# Verifizierter Compiler und Mikroprozessor im Projekt SAFEMOS

Seminararbeit zum Kurs 21667 Sicherheitsgerichtete Echtzeitsysteme (SS07)

Alexander Kühn, Matr.-Nr.: 7122993

30. Juni 2007

## Inhaltsverzeichnis

# 1 Einleitung

Eingebette Computersysteme haben in unserer technisch hochentwickelten Welt einen wachsenden Stellenwert erlangt. Es ist gängige Praxis, eingebettete Computersysteme, bestehend aus Digitalrechnern mit dazugehöriger Software auch in sicherheitskritischen Anwendungen wie bspw. Bremsensteuergeräte in Automobilen, Fly-by-wire in Flugzeugen) einzusetzen. Diese Anwendungen stellen extrem hohe Anforderungen an die Sicherheit und an die Verfügbarkeit der Systeme. Mit dem Begriff *Sicherheit* ist hierbei gemeint, daß ein System die Fähigkeit besitzt, innerhalb vorgegebener Grenzen für eine gegebene Zeitdauer keine Gefahr zu bewirken oder zuzulassen[1]. Die Entwicklung eines solchen Systems muß daher zum Ziel haben, alle Fehler, die zu einem Sicherheitsrisiko führen können, rechtzeitig zu erkennen und Gegenmaßnahmen zu ergreifen.

Nach [1] werden die Maßnahmen in folgende Kategorien unterteilt:

1. Ausschluß von Fehlern und Ausfällen,
2. Verminderung der Wahrscheinlichkeit von Fehlern und Ausfällen sowie
3. Beeinflussung der Auswirkung von Fehlern und Ausfällen.

Die vorliegende Arbeit zeigt die Methoden zu der ersten Kategorie auf, die im Rahmen des SAFEMOS-Projektes erforscht wurden. Diese Maßnahmen verfolgen die Strategie der Perfektion des eingebetteten Systems, es wird also ein fehlerfreies System angestrebt. Der Ansatz, der hierbei Verwendung findet, ist der Einsatz formaler Methoden innerhalb des Entwicklungsprozesses.

## 1.1 Formale Methoden

Formale Methoden bezeichnen im allgemeinen die Anwendung von mathematischen Techniken innerhalb eines Entwicklungsprozesses. Man unterscheidet hierbei grob zwischen

- formaler Spezifikation und
- formaler Verifikation.

Die formale Spezifikation hat zum Ziel, eine mathematisch exakte Beschreibung des zu entwickelnden Systems zu generieren, während die formale Verifikation versucht, einen mathematisch exakten Beweis zu erbringen, daß ein implementiertes System der Spezifikation entspricht. Formale Methoden basieren vornehmlich auf den mathematischen Techniken der

- Logik und der
- diskreten Mathematik.

[1] siehe dazu [1], S. 7ff

## 1.2 Das SAFEMOS Projekt

In der Industrie ist der Einsatz formaler Methoden bisher nur in vereinzelten Projekten zu finden, obwohl diese Methoden schon seit mehreren Jahren Gegenstand der Forschung sind. Dies wird z.T. damit begründet, daß sich die Forschung bisher noch nicht zu Genüge mit der komfortablen industriellen Nutzbarkeit der Methoden beschäftige[2]. Daneben setzen formale Methoden einen hohen Wissenstand bei den beteiligten Personen voraus; Hinzu kommt der zeitliche Aufwand, der mit der Anwendung von formalen Methoden einhergeht. Ein Projekt, dass sich mit der Untersuchung von Techniken zur Anwendung der formalen Methoden im Hard- und Softwareentwicklungsprozeß beschäftigt, ist das SAFEMOS-Projekt.

Das SAFEMOS-Projekt wurde von der britischen Organisation UK-IED (Information Engineering Directorate) ins Leben gerufen und hatte eine Laufzeit von vier Jahren (1989-1993). Die folgenden Unternehmen und akademischen Einrichtungen waren Partner des Projekts:

- INMOS Ltd., Bristol;
- SRI International Cambridge Computer Science Research Centre;
- Oxford University Computing Laboratory, Programming Research Group und
- University of Cambridge Computer Laboratory.

Die Ziele des Projektes waren es,

- den Einsatz von maschinenunterstützten formalen Beweistechniken und deren Vorteile für Hard- und Softwareprojekte aufzuzeigen,
- Methoden und Werkzeuge zur Unterstützung dieser Methoden zu entwickeln und deren Kosten abzuschätzen sowie
- ein tieferes Wissen über die praktische Anwendung von bekannten formalen Methoden und Werkzeugen zu erlangen.

Besonderes Augenmerk lag hierbei auf einer durchgängigen Nutzung formaler Spezifikation und Verifikation innerhalb des Entwicklungsprozesses, beginnend bei der Anforderungsanalyse bis hin zu dem Hardwaredesign. Zentrale Komponenten des Projektes wurden abgeleitet von der Programmiersprache Occam und der Transputer Prozessorarchitektur. Als mathematische Methode wurde die Logik höherer Ordnung zusammen mit dem automatischen Theorembeweiser HOL angewandt.

## 1.3 Komponenten des SAFEMOS-Projektes

In einem Entwicklungsprozeß wird der Abstraktionsgrad einer Entwurfsspezifikation schrittweise verfeinert, d.h. ausgehend von einer Grobspezifikation wird ein immer feinerer Detaillierungsgrad bis zur endgültigen Implementierung angestrebt. Das

[2] vgl. [2] S. 7ff.

SAFEMOS-Projekt versucht durch eine Aufschichtung der untersuchten Techniken diese Struktur mit dem Ziel abzubilden, am Ende des Entwicklungsprozesses ein komplett verifiziertes System zu erhalten (siehe Abbildung 1).

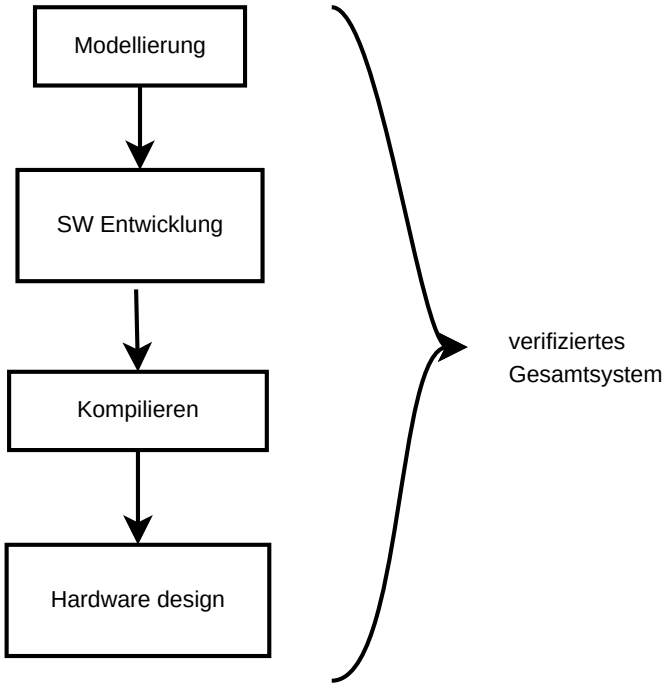


Abbildung 1: Komponenten des SAFEMOS-Projektes

Folgende Techniken wurden auf den verschiedenen Abstraktionsstufen eingesetzt:

**Modellierung:** "Timed Transition"-Systeme wurden mit HOL modelliert, um die Anforderungsanalyse und die Designspezifikationen abzudecken.

**Softwareentwicklung:** Hier kam die Theorie der "State-Transition Assertion" (STA) zum Tragen.

**Kompilierung:** Dafür wurden Intervallsemantiken basierend auf intervalltemporaler Logik angewandt, um den Kompiliervorgang von einer Occam-ähnlichen Programmiersprache auf einen Transputer-Prozessorinstruktionssatz zu modellieren.

**Hardwaredesign:** Ein inkrementelles Rahmenwerk zum Design von Mikroprozessoren basierend auf HOL wurde entwickelt.

Insbesondere die Ebenen Kompilierung und Hardwaredesign werden im folgenden näher behandelt.

## 1.4 Logik höherer Ordnung

### 1.4.1 Überblick über das HOL System

Ein Großteil der im SAFEMOS-Projekt angewendeten Methoden wurde mithilfe des HOL Theorembeweisers automatisiert.[3] Das HOL-System unterstützt hierbei den Benutzer interaktiv bei dem Beweis von Theoremen unter Anwendung der Logik höherer Ordnung. HOL basiert auf der funktionalen Programmiersprache ML[4]. Das System kann direkt zum Beweisen von Theoremen genutzt werden, aber auch als Umgebung für eigene, angepasste Theorembeweiser dienen. Die Geschichte des HOL-Systems begann in den frühen 1970er Jahren mit der Implementierung des LCF-Systems[5] durch Robin Miller in Edinburgh. Von diesem ursprünglichen System wurde FranzLISP abgeleitet, auf welchem wiederum das System CambridgeLCF basiert. Die Entwicklung mündete dann in dem aktuellen HOL90-System, das über [3] zu beziehen ist.

### 1.4.2 Einführung in die Logik höherer Ordnung

Die folgende Einführung basiert auf Beispielen, die in [4] zu finden sind.

Die Logik höherer Ordnung ist eine Erweiterung der Prädikatenlogik erster Stufe, bei der das getypte Lambda-Kalkül nach Church einbezogen wird. In dieser Logik können sich Variablen sowohl auf Funktionen als auch auf Prädikate beziehen; weiterhin existiert ein Typsystem.

Die einfache Aussagenlogik basiert auf dem Konstrukt des Satzes. Ein Satz ist entweder ein atomarer Satz oder ein komplexer Satz, der wiederum aus Sätzen zusammengesetzt ist. Hierzu ein Beispiel:

$A$ : Es regnet.
$B$ : Es ist kalt.
$A \wedge B$ : Es regnet und es ist kalt. (Konjunktion)

Die Prädikatenlogik basiert auf der Idee, daß die Welt aus Objekten besteht, die in Beziehung zueinander stehen. Beziehungen werden dabei durch Relationen ausgedrückt, während die Prädikate die Eigenschaften eines Objektes angeben.

Beispiel:

$\texttt{Montag}(x)$ : $x$ ist ein Montag
Es existieren weiterhin Quantoren:
$\forall x.\texttt{Montag}(x) \Rightarrow \texttt{kalt}(x)$ Alle Montage sind kalt.

Die Erweiterung zur Logik höherer Stufe wird durch die Einführung des Lambda-Kalküls erreicht. Letzteres besteht einerseits aus der Lambda-Abstraktion $\lambda x.A$, die eine Funktion definiert, die $x$ als Argument bekommt und den Funktionskörper $A$ besitzt. Eine weitere Eigenschaft des Lambda-Kalküls ist die Anwendung solcher Funktionen. $FA$ bedeutet, daß die Funktion $F$ auf den Ausdruck $A$ angewendet wird. Desweiteren existiert ein Typsystem; damit wird jedem Term in der Logik ein Typ zugeordnet.

[3] vgl. [2], S. 67ff.
[4] MetaLanguage
[5] Logic for Computable Functions

# 2 Verifizierte Compiler

Ein Hauptaspekt bei der Entwicklung eingebetteter Systeme ist das Überführen eines in einer höheren Programmiersprache vorliegenden Programms in den Instruktionssatz des eingesetzten Mikroprozessors, das sog. Kompilieren. Im SAFEMOS-Projekt wurde ein Weg aufgezeigt, mit dem es möglich ist, den Vorgang des Kompilierens von einer einfachen imperativen Programmiersprache SAFE in einen transputerähnlichen Prozessorbefehlssatz, der SAFE Maschinensprache, formal zu spezifizieren und zu verifizieren.

Es wurde dargestellt, daß der zugrundeliegende Algorithmus des Kompilierens in der Lage ist, die Programmanweisungen korrekt in das Zielsystem zu überführen. Dabei werden alle Echtzeitbedingungen erfüllt; wenn also ein Programm kompiliert, so erfüllt es auch alle zeitlichen Anforderungen. Zu diesem Zwecke wurde der Kompiliervorgang in HOL modelliert.

## 2.1 Modellierung der Programmiersprache SAFE

Die Programmiersprache SAFE wurde als einfache Implementierungssprache im SAFEMOS-Projekt entwickelt. Ein- und Ausgabe werden als Speicherzugriffe abgebildet; weiterhin können in der Sprache Zeitanforderungen spezifiert werden.

### 2.1.1 Syntax

Die Syntax der SAFE-Sprache besteht zunächst aus Prozessen, die in BackusNaur-Form (BNF) definiert sind als:

$$\begin{aligned} p ::= \texttt{SKIP} \mid \texttt{STOP} \mid x :=_t\ e \mid \texttt{READ}_t\ x_1\ x_2 \mid \texttt{WRITE}_t\ x\ e \mid \\ p_1; p_2 \mid \texttt{IF}_{t_1,t_2}\ b\ p_1\ p_2 \mid \texttt{WHILE}_{t_1,t_2}\ b\ p \mid \texttt{LOCAL}\ x\ p \end{aligned}$$

Die folgende Tabelle zeigt die Bedeutung der Prozeßdefinitionen:

Die Prozesse werden parallel ausgeführt, erst der Ausdruck der Prozeßsequenz ; führt dazu, dass die Ausdrücke hintereinander abgearbeitet werden. Das Besondere an der SAFE-Sprache ist die Möglichkeit, die Ausdrücke mit Zeitlimitierungen zu versehen. Dies ermöglicht den Einsatz der Sprache in Echtzeitumgebungen.

| Prozess | Bedeutung: |
|---|---|
| SKIP | der Prozeß wird sofort beendet |
| STOP | hält die Ausführung endgültig an |
| $x :=_t \ e$ | weist der Variablen $x$ den Wert des Ausdrucks $e$ zu; die Zuweisung dauert $t$ Zeiteinheiten |
| $\texttt{READ}_t \ x_1 \ x_2$ | weist den Inhalt des Eingangsports $x_1$ der Variablen $x_2$ zu |
| $\texttt{WRITE}_t \ x \ e$ | schreibt den Wert des Ausdrucks $e$ auf den Ausgabeport $x$ |
| $p_1; p_2$ | Prozeßsequenz: die Prozesse $p_1$ und $p_2$ werden direkt nacheinander ausgeführt |
| $\texttt{IF}_{t_1,t_2} \ b \ p_1 \ p_2$ | führe je nach Wert des Bool'schen Ausdrucks $b$ entweder $p_1$ oder $p_2$ aus. Dabei unterliegt die Auswertung von $b$ der Zeitgrenze $t_1$ und die Ausführung der bedingten Prozesse der Zeitgrenze $t_2$ |
| $\texttt{WHILE}_{t_1,t_2} \ b \ p$ | führt den Prozeß $p$ in einer Schleife aus, solange die Bedingung $b$ wahr ist. Das Auswerten der Bedingung darf nicht länger als $t_1$ dauern, während die Ausführung des Prozesses nicht länger als $t_2$ dauern darf. |

Ein Ausdruck $e$ ist definiert als

$$e ::= \texttt{CONST} \ c \mid \texttt{VAL} \ x \mid \texttt{IOP1} \ E1 \ e \mid \texttt{IOP2} \ E2 \ e1 \ e2$$

während Bool'sche Ausdrücke $b$ definiert sind als

$$b ::= \texttt{TRUE} \mid \texttt{FALSE} \mid \texttt{BOP1} \ B1 \ b \mid \texttt{BOP2} \ B2 \ b1 \ b2 \mid \texttt{ROP2} \ R2 \ e1 \ e2$$

Die monadischen und binären Operatoren sind hierbei generisch, d.h. jeder vom Prozessor unterstützte Operator kann eingesetzt werden.

### 2.1.2 Semantik

Die Semantik der SAFE-Sprache wurde mithilfe der intervalltemporalen Logik modelliert. Zunächst wurden in dem Projekt SAFEMOS die numerischen Ausdrücke

$$\begin{aligned}
\mathcal{M}^e(\texttt{CONST} \ n) &\equiv n \\
\mathcal{M}^e(\texttt{VAL} \ x) &\equiv \hat{\texttt{MEM}} \ x \\
\mathcal{M}^e(\texttt{IOP1} \ E1 \ e) &\equiv E1(\mathcal{M}^e \ e) \\
\mathcal{M}^e(\texttt{IOP2} \ E2 \ e_1 \ e_2) &\equiv E2(\mathcal{M}^e \ e_1)(\mathcal{M}^e \ e_2)
\end{aligned}$$

sowie die Bool'schen Ausdrücke

$$\begin{aligned}
\mathcal{M}^b(\texttt{TRUE}) &\equiv \texttt{T} \\
\mathcal{M}^b(\texttt{FALSE}) &\equiv \texttt{F} \\
\mathcal{M}^b(\texttt{BOP1} \ B1 \ b) &\equiv B1(\mathcal{M}^b \ b) \\
\mathcal{M}^b(\texttt{BOP2} \ B2 \ b_1 \ b_2) &\equiv B2(\mathcal{M}^b \ b_1)(\mathcal{M}^b \ b_2) \\
\mathcal{M}^b(\texttt{ROP2} \ R2 \ e_1 \ e_2) &\equiv R2(\mathcal{M}^e \ e_1)(\mathcal{M}^e \ e_2)
\end{aligned}$$

definiert. Die Semantik der Prozesse wird dann definiert als:

$$\begin{aligned}
\mathcal{M}^p(\texttt{SKIP}) &\equiv \texttt{Skip} \\
\mathcal{M}^p(\texttt{STOP}) &\equiv \texttt{Stop} \\
\mathcal{M}^p(x :=_t\ e) &\equiv (\texttt{MEM}\ x) :=_{0,t} (\mathcal{M}^e\ e) \\
\mathcal{M}^p(\texttt{READ}_t\ r\ x) &\equiv (\texttt{MEM}\ x) :=_{0,t} (\hat{\texttt{INP}}\ r) \\
\mathcal{M}^p(\texttt{WRITE}_t\ r\ e) &\equiv (\texttt{OUT}\ r) :=_{0,t} (\mathcal{M}^e\ e) \\
\mathcal{M}^p(p_1; p_2) &\equiv (\mathcal{M}^p\ p_1); (\mathcal{M}^p\ p_2) \\
\mathcal{M}^p(\texttt{IF}_{t_1,t_2}\ b\ p_1\ p_2) &\equiv \texttt{If}_{t_1,t_2}(\mathcal{M}^b\ b)(\mathcal{M}^p\ p_1)(\mathcal{M}^p\ p_1) \\
\mathcal{M}^p(\texttt{WHILE}_{t_1,t_2}\ b\ p) &\equiv \texttt{While}_{t_1,t_2}(\mathcal{M}^b\ b)(\mathcal{M}^p\ p) \\
\mathcal{M}^p(\texttt{LOCAL}\ x\ p) &\equiv \texttt{Local}(\texttt{MEM}\ x)(\mathcal{M}^p\ p)
\end{aligned}$$

## 2.2 Modellierung der Maschinensprache SAFE

### 2.2.1 Syntax

Die Maschinensprache SAFE orientiert sich an dem Instruktionssatz für Transputer-Prozessoren. Der Prozessor besitzt die drei Datenregister A, B und C, die sich wie ein dreistufiger Stack verhalten. Ebenso sind ein Programmzähler P, ein adressierbarer Speicher sowie speicherabgebildete Ein- und Ausgänge vorhanden. Die Typedefinition für den Speicher wurde folgendermaßen in HOL definiert:

$$m ::= \texttt{MEM}\ a \mid \texttt{REG}\ a \mid \texttt{INP}\ a \mid \texttt{OUT}\ a$$

`MEM` $a$ entspricht einer Speicherstelle an der Adresse $a$, wohingegen `REG` einem der ALU-Register entspricht. `IN` und `OUT` repräsentieren Ein- und Ausgabeports an der Adresse $a$. Die Adressen $a$ können numerisch oder symbolisch angegeben werden:

$$a ::= \texttt{REAL}\ w \mid \texttt{SYMB}\ x$$

$w$ repräsentiert eine physikalische Adresse, während $x$ einen symbolischen Verweis auf eine Variable darstellt. Die Syntax der Instruktionen ist definiert als

$$i ::= \texttt{LDC}\ w \mid \texttt{LDL}\ m \mid \texttt{STL}\ m \mid \texttt{JMP}\ w \mid \texttt{JMZ}\ w \mid \texttt{OP1}\ o1 \mid \texttt{OP2}\ o2 \mid \texttt{STP}$$

Die folgende Tabelle zeigt die Bedeutung der Instruktionen auf:

| Instruktion | Bedeutung: |
|---|---|
| `LDC` $w$ | lade Konstante $w$ in Register $A$ |
| `LDL` $m$ | lade Speicherinhalt $m$ in Register $A$ |
| `STL` $m$ | speichere Inhalt von Register $A$ an Speicherstelle $m$ |
| `JMP` $w$ | addiere $w+1$ auf Programmzähler $P$ (Springen) |
| `JMZ` $w$ | addiere $w+1$ auf Programmzähler, wenn Register $A$ null ist, anderenfalls addiere 1 auf $P$ |
| `OP1` $o1$ | wende monadische Operation $o1$ auf Register $A$ an |
| `OP2` $o2$ | wende binäre Operation $o2$ auf Register $A$ und $B$ an |
| `STP` | halte den Prozessor an |

### 2.2.2 Semantik

Mithilfe der intervalltemporalen Logik werden die Instruktionen in HOL folgendermaßen modelliert:

$$
\begin{array}{lcl}
\mathcal{M}^i(\texttt{LDC}\ w) & \equiv & A, B, C, P\ :=_{1,\mathcal{T}^i(\texttt{LDC}\ w)}\ w, \hat{A}, \hat{B}, \hat{P}+1 \\
\mathcal{M}^i(\texttt{LDL}\ m) & \equiv & A, B, C, P\ :=_{1,\mathcal{T}^i(\texttt{LDL}\ m)}\ \hat{m}, \hat{A}, \hat{B}, \hat{P}+1 \\
\mathcal{M}^i(\texttt{STL}\ m) & \equiv & m, A, B, P\ :=_{1,\mathcal{T}^i(\texttt{STL}\ m)}\ \hat{A}, \hat{B}, \hat{C}, \hat{P}+1 \\
\mathcal{M}^i(\texttt{JMP}\ w) & \equiv & A, B, C, P\ :=_{1,\mathcal{T}^i(\texttt{JMP}\ w)}\ \hat{A}, \hat{B}, \hat{C}, \hat{P}+1+w \\
\mathcal{M}^i(\texttt{JMZ}\ w) & \equiv & \\
 & & A, B, C, P\ :=_{1,\mathcal{T}^i(\texttt{JMZ}\ w)}\ \hat{A}, \hat{B}, \hat{C}, \hat{P}+1+w \lhd \hat{A} = 0 \rhd \\
 & & A, B, P\ :=_{1,\mathcal{T}^i(\texttt{JMZ}\ w)}\ \hat{B}, \hat{C}, \hat{P}+1 \\
\mathcal{M}^i(\texttt{OP1}\ o1) & \equiv & A, B, C, P\ :=_{1,\mathcal{T}^i(\texttt{OP1}\ o1)}\ o1\hat{A}, \hat{B}, \hat{C}, \hat{P}+1 \\
\mathcal{M}^i(\texttt{OP2}\ o2) & \equiv & A, B, P\ :=_{1,\mathcal{T}^i(\texttt{OP2}\ o2)}\ o2\hat{B}\ \hat{A}, \hat{C}, \hat{P}+1
\end{array}
$$

$\mathcal{M}^i\ i\ \sigma$ modelliert die Bedeutung der Instruktion $i$ im Intervall $\sigma$. Der Ausdruck $\mathcal{T}^i(i)$ beschreibt die Zeitdauer für die Instruktion $i$. Das Modell beschreibt den Zustandsübergang als Funktion des alten Zustands.

Das Verhalten eines Programms, das im ROM des Prozessors abgelegt ist, kann als kombinierte Auswirkung der Ausführung der einzelnen Instruktionen zwischen den ROM-Speicherstellen $n1$ und $n2$ modelliert werden:

$$\texttt{Run rom}\ n_1\ n_2 \equiv \texttt{Loop}(n_1 \leq P \wedge P < n_2)(\texttt{Effect}\ rom\ P)$$

wobei die Auswirkung `Effect` als die Bedeutung der Instruktion im Intervall $\sigma$ an eine ROM-Adresse $e$ modelliert wird:

$$\texttt{Effect}\ rom\ e\ \sigma \equiv \mathcal{M}^i(rom(e(\texttt{init}\ \sigma)))\ \sigma$$

## 2.3 Formale Spezifikation eines Compilers

Die Spezifikation eines Compilers wird durch Prädikate beschrieben, die rekursiv über die Syntax von Instruktionen und Ausdrücken definiert sind. Ein SAFE-Prozeß wird dabei als Prädikat über eine Symboltabelle, ein ROM und die Start- und Endadressen des kompilierten Objektcodes definiert:

$$\mathcal{C}^P\ p\ S\ n_1\ n_2\ rom$$

Dieses Prädikat ist erfüllt, wenn das ROM $rom$ den Objektcode des Prozesses $p$ in bezug auf die Symboltabelle $S$ zwischen den Speicherstellen $n_1$ und $n_2$ enthält. Da Ausdrücke einer Ausführungszeit unterliegen, wird das Prädikatenmodell für Ausdrücke um die Zeit erweitert:

$$\mathcal{C}^e\ e\ S\ t\ n_1\ n_2\ rom$$

Dieses Prädikat ist wiederum erfüllt, wenn der Objektcode $t$ Zeiteinheiten zur Ausführung benötigt. Die Symboltabelle stellt eine Zuordnung von Variablen auf Speicherstellen dar, auf die über folgende Funktionen zugegriffen werden kann:

- $\mathcal{L}\ S\ x$: gibt die Speicheradresse der Variablen $x$ aus;
- $\mathcal{E}\ S\ x$: fügt die Variable $x$ in die Symboltabelle ein

### 2.3.1 Modellierung des Kompilierens von Ausdrücken

Die Ausdrücke der SAFE-Sprache unterliegen zeitlichen Einschränkungen; daher ist es notwendig, daß die Zeitdauer der aus ihr resultierenden Maschineninstruktionen berechenbar ist. Das Kompilieren von zeitlich beschränkten Instruktionen wird durch das Prädikat

$$\mathcal{C}^{\texttt{INST}}\ i\ t\ n_1\ n_2\ \texttt{rom} \equiv (t = \mathcal{T}^i(i)) \wedge (rom\ n_1 = i) \wedge (n_2 = n_1 + 1)$$

modelliert. Dieses Prädikat sagt aus, ob das ROM die Instruktion $i$ zwischen den Adressen $n_1$ und $n_2$ enthält und ob die Zeitbedingung $t$ der Instruktion eingehalten wird. Die Verkettung zweier Kompilationsschritte $c_1$ und $c_2$ wird dann durch das Prädikat

$$(c_1 \oplus c_2)t\ n_1\ n_2\ rom \equiv \exists t_1\ t_2\ n \bullet\ c_1\ t_1\ n_1(n_1{+}n)\ rom \wedge\ c_2\ t_2\ (n_1{+}n)\ n_2\ rom \wedge\ t = t_1{+}t_2$$

ausgedrückt. Es ist gültig, wenn die Instruktion von $c_1$ im Rom zwischen $n_1$ und $n$ liegt und wenn die Instruktion von $c_2$ zwischen $n$ und $n_2$ liegt. Die Dauer der Ausführung zweier verketteter Instruktionen muß der Summe der einzelnen Ausführungszeiten entsprechen.

Das nachfolgende Prädikat spezifiziert das Anlegen von temporären Variablen, die beim Auswerten der Ausdrücke anfallen können, da der Prozessor nur aus einem dreielementigen Registersatz besteht:

$$\mathcal{C}^{\texttt{TMP}}\ c\ S \equiv (\mathcal{C}^{\texttt{INST}}(\texttt{STL}(\mathcal{L}(\mathcal{E}\ S\ \texttt{TMP})\texttt{TMP}))) \oplus (c(\mathcal{E}\ S\ \texttt{TMP})) \oplus (\mathcal{C}^{\texttt{INST}}(\texttt{LDL}(\mathcal{L}(\mathcal{E}\ S\ \texttt{TMP})\texttt{TMP})))$$

Im folgenden werden die Prädikate zur Kompilierung von SAFE-Ausdrücken modelliert:

$$\begin{aligned}
\mathcal{C}^e(\texttt{CONST}\ w)S &\equiv \mathcal{C}^{\texttt{INST}}(\texttt{LDC}\ w) \\
\mathcal{C}^e(\texttt{VAL}\ x)S &\equiv \mathcal{C}^{\texttt{INST}}(\texttt{LDL}\ (\mathcal{L}S(\texttt{MEM}\ x))) \\
\mathcal{C}^e(\texttt{IOP1}\ E1\ e)S &\equiv (\mathcal{C}^e\ e\ S)\ \oplus\ (\mathcal{C}^{\texttt{INST}}(\texttt{OP1}\ E1)) \\
\mathcal{C}^e(\texttt{IOP2}\ E2\ e_1\ e_2)S &\equiv (\mathcal{C}^e\ e_2\ S)\ \oplus\ (\mathcal{C}^{\texttt{TMP}}(\mathcal{C}^e\ e_1)S)\ \oplus\ (\mathcal{C}^{\texttt{INST}}(\texttt{OP2}\ E2))
\end{aligned}$$

Die Prädikate für Binärausdrücke werden entsprechend definiert:

$$\begin{aligned}
\mathcal{C}^b(\texttt{TRUE})S &\equiv \mathcal{C}^{\texttt{INST}}(\texttt{LDC}\ 1) \\
\mathcal{C}^b(\texttt{FALSE})S &\equiv \mathcal{C}^{\texttt{INST}}(\texttt{LDC}\ 0) \\
\mathcal{C}^b(\texttt{BOP1}\ B1\ b)S &\equiv (\mathcal{C}^b\ b\ S)\ \oplus\ (\mathcal{C}^{\texttt{INST}}(\texttt{OP1}\ B1)) \\
\mathcal{C}^b(\texttt{BOP2}\ B2\ b_1\ b_2)S &\equiv (\mathcal{C}^b\ b_2\ S)\ \oplus\ (\mathcal{C}^{\texttt{TMP}}(\mathcal{C}^b\ b_1)S)\ \oplus\ (\mathcal{C}^{\texttt{INST}}(\texttt{OP2}\ B2)) \\
\mathcal{C}^b(\texttt{ROP2}\ R2\ e_1\ e_2)S &\equiv (\mathcal{C}^e\ e_2\ S)\ \oplus\ (\mathcal{C}^{\texttt{TMP}}(\mathcal{C}^e\ e_1)S)\ \oplus\ (\mathcal{C}^{\texttt{INST}}(\texttt{OP2}\ R2))
\end{aligned}$$

### 2.3.2 Modellierung des Kompilierens von Prozessen

Prozesse unterliegen im allgemeinen keinen zeitlichen Beschränkungen; nur die Zuweisung und das Auswerten von Bedingungen können mit zeitlichen Einschränkungen versehen werden. Hierzu wird folgendes Prädikat definiert, das überprüft, ob der Objektcode nicht mehr Zeit benötigt als spezifiziert:

$$[c]_t\ n_1\ n_2\ rom \equiv \exists t' \bullet t' \leq t \wedge c\ t'\ n_1\ n_2\ rom$$

. Das Verbinden zweier Prozesse erfolgt analog dem Verbinden zweier Ausdrücke, diesmal jedoch ohne zeitliche Einschränkungen:

$$(c1 \otimes c2)n_1\ n_2\ rom \equiv \exists n \bullet\ c_1\ n_1(n_1+n)rom \wedge c_2(n_1+n)n_2\ rom$$

Somit können die Relationen für die Prozeßelemente modelliert werden:

$$\begin{aligned}
\mathcal{C}^{\mathtt{SKIP}}\ n_1\ n_2\ rom &\equiv n_2 = n_1 \\
\mathcal{C}^{\mathtt{STOP}}\ n_1\ n_2\ rom &\equiv \exists t \bullet \mathcal{C}^{\mathtt{INST}}\mathtt{STP}\ t\ n_1\ n_2\ rom \\
\mathcal{C}^{\mathtt{ASSIGN}}\ x\ e\ t\ n_1\ n_2\ rom &\equiv [e \oplus \mathcal{C}^{\mathtt{INST}}(\mathtt{STL}\ x)]_t\ n_1\ n_2\ rom \\
\mathcal{C}^{\mathtt{SEQ}}\ c_1\ c_2\ n_1\ n_2\ rom &\equiv (c_1 \otimes c_2)n_1\ n_2\ rom \\
\mathcal{C}^{\mathtt{JMZ}}\ n\ t\ n_1\ n_2\ rom &\equiv \mathcal{C}^{\mathtt{INST}}(\mathtt{JMZ}(n-(n_1+1)))t\ n_1\ n_2\ rom \\
\mathcal{C}^{\mathtt{JMP}}\ n\ t\ n_1\ n_2\ rom &\equiv \mathcal{C}^{\mathtt{INST}}(\mathtt{JMP}(n-(n_1+1)))t\ n_1\ n_2\ rom \\
\mathcal{C}^{\mathtt{IF}}\ c_b\ c_1\ c_2\ (t_1,t_2)\ n_1\ n_2\ rom &\equiv \mathtt{let} \\
&\quad \mathcal{C}^{\mathtt{IF}}\ c_b\ c(t_1,t_2)\ n_1'\ n_2' = ([c_b\ \oplus(\mathcal{C}^{\mathtt{JMZ}}n_2')]_{t_1} \otimes c \otimes [\mathcal{C}^{\mathtt{JMP}}\ n_2]_{t_2})n_1'\ n_2'\mathtt{in} \\
&\quad ((\mathcal{C}^{\mathtt{IF1}}\ c_b\ c_1\ (t_1,t_2)) \otimes c_2)\ n_1\ n_2\ rom \\
\mathcal{C}^{\mathtt{WHILE}}\ c_b\ p\ (t_1,t_2)\ n_1\ n_2\ rom &\equiv ([c_b \oplus (\mathcal{C}^{\mathtt{JMZ}}\ n_2)]_{t_1} \otimes p \otimes [\mathcal{C}^{\mathtt{JMP}}\ n_1]_{t_2})\ n_1\ n_2\ rom
\end{aligned}$$

Die Spezifikation des Prädikats $\mathcal{C}^p\ p\ S$ der Kompilierung eines Prozesses $p$ kann nun aus den obigen Prädikaten definiert werden:

$$\begin{aligned}
\mathcal{C}^p\ \mathtt{SKIP}\ S &\equiv \mathcal{C}^{\mathtt{SKIP}} \\
\mathcal{C}^p\ \mathtt{STOP}\ S &\equiv \mathcal{C}^{\mathtt{STOP}} \\
\mathcal{C}^p\ (x :=_t\ e)\ S &\equiv \mathcal{C}^{\mathtt{ASSIGN}}(\mathcal{L}\ S(\mathtt{MEM}\ x))(\mathcal{C}^e\ e\ S)\ t \\
\mathcal{C}^p\ (\mathtt{READ}_t\ r\ x)\ S &\equiv \mathcal{C}^{\mathtt{ASSIGN}}(\mathcal{L}\ S(\mathtt{MEM}\ x))(\mathcal{C}^{\mathtt{INST}}(\mathtt{LDL}(\mathcal{L}\ S\ (\mathtt{INP}\ r))))\ t \\
\mathcal{C}^p\ (\mathtt{WRITE}_t\ r\ e)\ S &\equiv \mathcal{C}^{\mathtt{ASSIGN}}(\mathcal{L}\ S(\mathtt{OUT}\ r))(\mathcal{C}^e\ e\ S)\ t \\
\mathcal{C}^p\ (p_1;p_2)\ S &\equiv \mathcal{C}^{\mathtt{SEQ}}(\mathcal{C}^p\ p_1\ S)(\mathcal{C}^p\ p_2\ S) \\
\mathcal{C}^p\ (\mathtt{IF}_{t_1,t_2}\ b\ p_1\ p_2)\ S &\equiv \mathcal{C}^{\mathtt{IF}}(\mathcal{C}^b\ b\ S)(\mathcal{C}^p\ p_1\ S)(\mathcal{C}^p\ p_2\ S)(t_1,t_2) \\
\mathcal{C}^p\ (\mathtt{WHILE}_{t_1,t_2}\ b\ p)\ S &\equiv \mathcal{C}^{\mathtt{WHILE}}(\mathcal{C}^b\ b\ S)(\mathcal{C}^p\ p\ S)(t_1,t_2) \\
\mathcal{C}^p\ (\mathtt{LOCAL}\ x\ p)S &\equiv \mathcal{C}^p\ p(\mathcal{E}\ S(\mathtt{MEM}\ x))
\end{aligned}$$

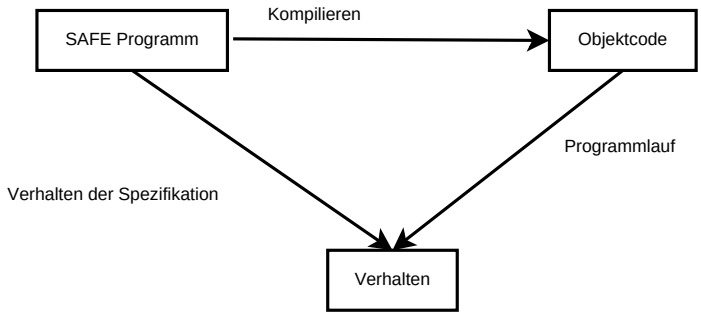


Abbildung 2: Korrektheit des Kompilierens

### 2.3.3 Beweis der Korrektheit

Das Kompilieren eines Prozesses wurde korrekt durchgeführt, wenn das Verhalten des Objektcodes dem Verhalten des SAFE-Programms entspricht (siehe Abbildung 2).

Der kompilierte Code eines Prozesses ist im ROM zwischen den Adressen $n_1$ und $n_2$ unter Verwendung der Symboltabelle $S$ abgelegt. Das Ausführen dieses Codes ab der Adresse $n_1$ muß nun das Verhalten des SAFE-Programms $\mathcal{M}_s^p\ p$ repräsentieren. Diese Aussage wird formal in HOL modelliert:

$$\forall\ S\ n_1\ n_2\ rom \bullet \mathcal{C}^p\ p\ S\ n_1\ n_2\ rom \Rightarrow (\ P :=_{0,0}\ \ n_1; \texttt{Run}\ rom\ n_1\ n_2 \sqsupseteq \mathcal{M}_s^p\ p; P :=_{0,0}\ \ n_2)$$

Der Beweis für das korrekte Kompilieren wurde für jeden Programmkonstrukt in HOL durchgeführt und kann als Beispiel für drei ausgewählte Konstrukte in [2], S.141ff. weiter vertieft werden.

# 3 Verifizierte Prozessoren

Im vorausgehenden Kapitel wurde aufgezeigt, wie im SAFEMOS-Projekt Objektcode entsteht, der sich garantiert wie das zugrundeliegende SAFE-Programm verhält.

Ein Mikroprozessor führt im allgemeinen den Objektcode, bestehend aus den Maschineninstruktionen, aus. Das Verhalten dieser Maschineninstruktionen ist spezifiziert. Ein Prozessor arbeitet dann korrekt, wenn die Implementierung des Prozessors dem spezifizierten Verhalten der Maschineninstruktionen entspricht. Eine Teilaufgabe des SAFEMOS-Projektes war es, das Entwickeln von Methoden für die Verifikation von Mikroprozessoren zu ermöglichen. Die allgemeine Strategie zur Verifikation von Prozessoren ist dabei die Auswirkung der Ausführungsschritte der Implementierung mit dem gewünschten Verhalten des abstrakten Modells zu vergleichen.

Ein Teilziel des SAFEMOS-Projektes war es, ein Abstraktions-Framework zur Spezifikation und Verifikation von Prozessoren zu erarbeiten. Dieses Framework hat den entscheidenden Vorteil, formale Methoden frühzeitig in den Entwicklungsprozeß einzubringen und somit einen inkrementellen Entwicklungsprozeß zu unterstützen. Die

folgenden Techniken wurden zur Erstellung des Abstraktions-Frameworks herangezogen:

**Hierarchie von Rechnermodellen:** Das Modell eines Prozessors kann als eine hierarchische Anordnung von Interpretern aufgefaßt werden. Jede Hierarchiestufe wird interpretiert durch eine Menge von Instruktionen auf der darunterliegenden Ebene. Im SAFEMOS-Projekt wurde jedoch nicht auf Modelle von Interpretern zurückgegriffen, sondern auf abstraktere Relationalmodelle.

**generische Argumente:** Generische Argumente erlauben es, diverse Aspekte des Designs außer acht zu lassen und sie nur als Parameter eines Designs zu betrachten. Die Wortbreite eines Prozessors kann beispielsweise als ein solcher Parameter definiert werden, da die Wortbreite zur reinen Verifikation von dem logischen Verhalten der Prozessorinstruktionen zunächt irrelevant ist.

**Verifikationstemplates:** Die Verifikation des Verhaltensmodells der Interpreterhierarchie kann mit Hilfe von Verifikationstemplates erleichtert werden, die für jede Interpreterstufe die Instruktionen und den dazugehörigen Interpreter verifizieren und das daraus resultierende Theorem zur Verifikation der nächsten Interpreterstufe heranziehen.

**inkrementelle Techniken:** Ein Framework zur Prozessorverifikation muß den inkrementellen Designprozeß unterstützen, d.h. wenn ein neuer Mikrocode zu den Prozessorinstruktionen hinzugefügt wird, sollten die bereits durchgeführten Verifikationsschritte wiederverwendbar sein.

## 3.1 Das Interpretermodell

Das Interpretermodell basiert auf einer Repräsentation des Verhaltens des Prozessors als Zustandsautomat, dessen Folgezustand sowohl von dem aktuellen Zustand als auch von der Umgebung abhängt. Ein solcher Zustandsübergang kann in HOL als Prädikat dargestellt werden. Die Semantik der Zustandsänderung durch die Instruktion $POP$ (= Stack leeren)eines einfachen Prozessors, dessen Zustand durch einen Programmzähler und einen drei-elementigen Stack abgebildet wird, kann beispielsweise folgendermaßen modelliert werden:

$$\texttt{POP\_SEM rep}\ ((pc', A', B', C', mem'), (pc, A, B, C, mem), env) =$$
$$(pc' = (\texttt{ADD1 rep})pc) \wedge (A' = B) \wedge (B' = C) \wedge (mem' = mem)$$

`rep` ist dabei der generische Parameter und $env$ eine Variable, die die Umgebung beschreibt. Das gewünschte Verhalten des Prozessors kann durch eine Menge von sog. Properties beschrieben werden. Jedes Property-Element besteht aus einem Tupel (= Tag; Prädikat). Das Tag dient zur Identifikation der Zustandsänderung. Die Beschreibung des Prozessors kann nun folgendermaßen aussehen:

$$[(\texttt{POP\_op}, \texttt{POP\_SEM rep}); (\texttt{JMP\_op}, \texttt{JMP\_SEM1 rep});$$
$$(\texttt{ADD\_op}, \texttt{ADD\_SEM rep}); (\texttt{JMP\_op}, \texttt{JMP\_SEM2 rep})];$$

Die Tags entsprechen den Instruktionscodes des Prozessors. Die Prädikate müssen nicht zwangsweise das vollständige Verhalten spezifizieren; so wurde im vorigen Beispiel keine Aussage über das Stack-Register C getroffen. Daher können die Tags in der Property-Menge mehrfach verschiedenenen Prädikaten zugeordnet werden. In diesem Beispiel wurden der Instruktion `JMP_op` die Prädikate `JMP_SEM1` und `JMP_SEM2` zugeordnet, was eine spätere Erweiterung der Verhaltensspezifikation ermöglicht.

Zur Modellierung des Zustandsautomaten mit den dazugehörigen Eigenschaften muß eine Auswahlfunktion definiert werden, die auf der Basis des aktuellen Zustands und der Umgebung ein Tag aus den Propertys aussucht. Die Definition des Zustandsautomaten ist in HOL folgendermaßen definiert:

$$\texttt{TRANSITION\_SYS}(\texttt{is\_selected}, \texttt{prop\_list})\ s\ e = (\forall t.\texttt{let a\_tag} = \varepsilon\texttt{tag.is\_selected}(\texttt{tag}, s, t, e\ t)\ \texttt{in}$$
$$(\forall \texttt{prop}.\ \texttt{prop MEM prop\_list}\ \wedge\ (\texttt{FST prop} = \texttt{a\_tag}) \Rightarrow \texttt{SND prop}(s(t+1), s, t, e, t)))$$

wobei der Ausdruck `MEM list` wahr wird, wenn $a$ Element der Liste ist. `FST` und `SND` greifen auf das erste bzw. das zweite Element eines Tupels zu. `is_selected()` wird dagegen zum Zeitpunkt $t$ wahr, wenn `tag` im aktuellen Zustand ausgewählt wurde. Der $\varepsilon$-Operator in HOL wird zu diesem `tag` aufgelöst. Die Definition von `TRANSITION_SYS` gilt für eine Auswahlfunktion und einer Menge von Properties, wenn ein Zustandsübergang ausgewählt wurde und dessen zugeordnete Properties gültig sind.

Die Definition des Mikrocodes des Prozessors muß folgender Bedingung genügen, um ein inkrementelles Modell zu gewährleisten:

$$\texttt{ALL\_UNIQUE}(\texttt{APPEND mcode1 mcode2}) \Rightarrow$$
$$\texttt{ROM}(\texttt{addr}, \texttt{out})(\texttt{APPEND mcode1 mcode2}) \Rightarrow \texttt{ROM}(\texttt{addr}, \texttt{out})\texttt{mcode1}$$

Die Bedingung `ALL_UNIQUE` wird wahr, wenn die Elemente einer Liste eindeutig sind, d.h. die Definition ist erfüllt, wenn bei dem Hinzufügen von weiterem Mikrocode der bereits enthaltene Mikrocode nicht beeinflusst wird.

## 3.2 Verifikation des Prozessors

Als Beispiel einer Verifikation wird in dem Projekt ein einfacher Prozessor herangezogen, dessen Zustand durch einen Programmzähler P, einen drei-elementigen Stack (A,B,C) und den Programmspeicher ROM sowie die Umgebung *env* definiert ist. Die Semantik der LDC-Instruktion kann beispielsweise in HOL folgendermaßen modelliert werden:

$$\begin{aligned}\texttt{LDC\_SEM rep}\ ((P', A', B', C', \texttt{ROM}'), (P, A, B, C, \texttt{ROM}), env) = \\ (\texttt{let instr} = \texttt{ROM}((\texttt{ADDR\_FN rep})\texttt{P}) \\ \texttt{in} \\ \texttt{let w} = (\texttt{ARG\_FN rep})\texttt{instr} \\ \texttt{in}((\texttt{P}' = (\texttt{ADD1 rep})\texttt{P}) \wedge \\ (A' = w) \wedge (B' = A) \wedge (C' = B) \wedge (\texttt{ROM}' = \texttt{ROM})))\end{aligned}$$

Die Implementierung des Prozessors wurde mikrocodebasiert entwickelt; somit kann das Verhalten der Implementierung durch das Prädikat

$$\texttt{MICRO\_MC MLIST\_A rep (SIG\_TUP8(mp\_index, P, A, B, C, IR, MAR, mem))}\ env$$

beschrieben werden, wobei `MLIST_A` den Mikrocode des Prozessors beschreibt.

Damit ist die Implementierung des Prozessors korrekt, wenn die Aussage

$$\begin{aligned}
&(\texttt{MICRO_MC MLIST_A rep}) \Rightarrow \\
\texttt{TRANSITION_SYS(IS_SELECT rep, [LDC_op, LDC_sem rep;\ JMP_op, JMP_SEM rep;}&\\
&\texttt{ADD_op, ADD_SEM rep;\ ...])} \\
&((\texttt{SIG_TUP5(P, A, B, C, mem)})\ \texttt{o}\ (\texttt{Temp_Abs}(\lambda\ t.\ \texttt{mp_index}\ t = 0))) \\
&(e\ o(\texttt{Temp_Abs}(\lambda\ t.\ \texttt{mp_index}\ t = 0)))
\end{aligned}$$

wahr ist.

Eine praxisnahe Anwendung dieses Verifikationsmodells wurde im SAFEMOS-Projekt anhand eines Prozessors, dessen Architektur sich an den Transputer-Prozessoren aus dem Hause INMOS orientiert, aufgezeigt. Die Architektur des Transputer-Prozessors ist in Abbildung 3 aufgezeigt.

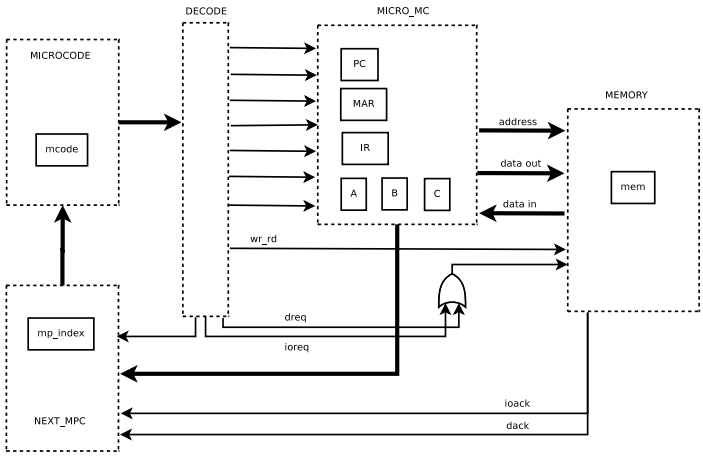


Abbildung 3: Architektur des Transputers

Der aktuelle Maschinenbefehl wird aus dem Speicher transferiert und das hinterlegte Mikroprogramm dem Befehlsdekoder zugeführt. Der Dekoder steuert das Rechenwerk des Prozessors an. Das Rechenwerk hat selsbt Zugriff auf den Speicher des Prozessors.

Weitere Details und der Projektbericht sind unter [2] S.167ff. zu finden.

# 4 Fazit

In dem Projekt SAFEMOS konnte gezeigt werden, daß die durchgängige Nutzung von formalen Techniken in einem Projekt möglich ist und somit ein verifiziertes eingebettetes System entstehen kann.

Wie aber auch aus dem Projekt ersichtlich ist, ist ein hohes mathematisches Fachwissen insbesondere in der formalen Logik unabdingbar, um diese Techniken zu nutzen.

Nach Laufzeit des Projektes konnten Methoden, die im Laufe des Projektes entwickelt wurden, in weiteren Projekten eingesetzt werden:

- Die Methode der State Transition Assertions floß direkt in die Entwicklung des CSP-Werkzeugs ein. Das CSP-System genoß eine hohe Akzeptanz in der Industrie.
- Das Unternehmen Inmos führte ein Neudesign der Transputer-Prozesoren aufgrund der gewonnenen Kentnisse aus dem SAFEMOS-Projekt durch.
- Auf Basis der SAFEMOS Technologie entstand das Projekt CLI. Aus diesem heraus, wurden die Subprojekte gegründet, die sich mit der Verfikation einer Teilmenge von ADA (AVA), dem Mach-Mikrokernel und dem Motorola 68020 Prozessoren beschäftigen.
- Ein weiteres Nachfolgeprojekt ist das ProCos Projekt, hierbei liegt der Fokus auf der Seite der Software.

# Literatur


[1] Halang, Wolfgang A.; Konakovsky, Rudolf: *Sicherheitsgerichtete Echtzeitsysteme* Oldenbourg Verlag München – Wien, 1999.

[2] J. Bowen, Hrsg.: *Towards Verified Systems* Elsevier Science B.V. Amsterdam, 1994.

[3] Internetseite zum HOL-System: *http://hol.sf.net*, Stand vom 08.03.2007.

[4] Ullrich, Dominik: *Interaktives Theorem Proving mit HOL*, Seminarausarbeitung, Ausgewählte Kapitel des Software Engineering, Westfälische Wilhelms-Universität Münster, 2006.