ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

TIK Institut für Technische Informatik und Kommunikationsnetze

Florian Kaufmann

Design and implementation of a modular emulator for Topsy

Diploma Thesis DA-2003.25
March 2003 to July 2003

Supervisor: Lukas Ruf
Co-Supervisor: Matthias Bossardt
Professor: Bernhard Plattner

Zusammenfassung

Dieses Dokument dokumentiert die Diplomarbeit von Florian Kaufmann während des Sommersemesters 2003.

Es wurde ein modularer Emulator designed und implementiert. Modular bedeutet, dass verschiedene Implementationen von Architekturen hinzugefügt werden können. Als Beispiel wurde die Cpu MipsR3051 und das IO-System Mipsboard implementiert.

Dieser Emulator wird benötigt, um erstens die alternden Mipsboards zu ersetzen, und zweitens um eine Platform zu haben, mit der das freie Betriebsystem Topsy bequem debugged werden kann. Damit der Emulator auch für andere Hosts als nur die Mipsarchitektur verwendet werden kann, muss er modular sein.

Die Implementation war erfolgreich; Der Emulator kann das Topsy OS ausführen.

Dieses Dokument beschreibt das Softwaredesign und bildet eine detaillierte Anleitung, wie Implementationen von neue Architekturen hinzugefügt werden können.

Diese Arbeit kann sinnvoll erweitert werden, indem ein Interface zum bekannten GDB Debugger der GNU Tools designed und implementiert wird.

abstract

This document describes the diploma thesis of Florian Kaufmann during the summer term 2003. A modular emulator was designed and implemented. Modular signifies that arbitrary implementations of architectures can be added. As an example, the MipsR3051 cpu and the iosystem Mipsboard have been implemented.
This emulator is needed to replace the Mipsboards which are growing old and to have a platform with which the emulated program can easely be debugged. In order to use the emulator for other hosts than Mips architecture, it has to be modular.
The implementation was very successful. The emulator is able to run the Topsy OS.
This document describes the design of the software and builds a detailed manual about how new implementations of architectures can be added.
The presented work can reasonably be extended by designing and implimenting an inteface to the famous GDB debugger of the GNU tools.

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

1 Einführung

1.1 Motivation

Die Aufgabe eines Emulators ist es, die Hardware eines Computersystems in Software nachzubauen. Diese Hardware beinhaltet die Cpu, das I0-System und deren Devices.
Somit können Programme, die für die reale Hardware geschrieben worden sind, auch auf dem Emulator ausgeführt werden. Da sich der Emulator genau so verhält wie die reale Hardware, merkt das Programmm nichts von der Substitution. Es ist beispielsweise möglich, ein Programmm, das für ein ATA-kompatibles PC-System entwickelt wurde, auf einem Macintosh laufen zu lassen.
Ein weiterer grosser Vorteil des Emulators gegenüber der realen Hardware ist, dass er jederzeit gestoppt und der Status der Cpu, des I0-Systems und der Devices ausgegeben werden können. Dies ist nützlich für das Debuggen eines Programmes. Nach jeder Instruktion kann der Emulator anghalten und der Status der Hardware ausgelesen werden. Ein Debugger liefert zwar ähnliche Dienste, ist aber auf den Inhalt der Register der Cpu beschränkt. Zudem ist der Debugger selbst ein Programm. Das heisst,im Hintergrund muss bereits ein Betriebssystem sein. Möchte man jedoch ein Betriebsystem selbst während dem Aufstarten debuggen, kann demzufolge kein Debugger eingesetzt werden.
In diesem Konkreten Fall gieng es auch darum, dass die vorhandenen Mipsboards, die zu Studienzwecken verwendet werden, durch den Emulator erstetzt werden können. Denn diese sind am Veralten, und es lohnt sich nicht, sie zu ersetzen. Desweitern sollen die genannten Debuggfähigkeiten des Emulators genutzt werden, um das freie Betriebsystem Topsy auf neue Architekturen zu portieren.
Es existieren bereits verschiedene Emulatoren, wie zum Beispiel MipsSim, vmips, vmware oder bochs. Nur sind diese auf jeweils genau eine Architektur fixiert: MipsSim und vmips auf den Mips, vmware und bochs auf die Ia32 Architektur. Das Bedürfnis nach einem modularen Emulator bleibt also bestehen. Des weitern war MipsSim in Java implementiert. Wieso dies ein Nachteil ist, wird anschliessend erklärt.
Die Implementation soll in C oder C++ geschrieben sein. Java stand nicht zur Option. Denn Java ist abhängig von gewissen Konfigurationen der Hardware. Zum Beispiel ergeben sich mit Java Probleme, wenn auf eine Netzwerkkarte zugegriffen wird, die nicht so konfiguriert ist, wie es die Java erwartet. Diese Probleme hat C bzw C++ nicht. Die Implementation wurde in C++ implementiert. C++ ist im Gegensatzt zu C Objektorientiert, was ein grosser Vorteil ist wenn ein modulares System entwickelt werden muss.

1.2 Entwicklungsumgebung

Entwickelt wurde der Emulator auf einem Pentium 3 mit Suse Linux 8.0. Als Kompiler wurde GCC 3.2.2 verwendet. Da exzessiver Gebrauch von Templates gemacht wird, haben ältere Kompiler Mühe, den Code zu kompilieren. Visual C++ 6.0 von Microsoft und GCC 2.95 waren nicht in der Lage, den Code zu kompilieren.

2 Übersicht

In diesem Kapitel werden die wichtigsten Eigenschaften einer Cpu und eines I0-Systems erklärt, die relevant sind für eine modulare Emulation dieser beiden Komponenten. Anschliessend wird aufgezeigt, wie diese Eigenschaften im Software Design des Emulators umgesetzt werden.

2.1 Einführung

Die Hardware von Computersystemen wie sie im Büro- und Heimbereich verwendet werden, ist vereinfacht gesehen aus zwei Komponenten aufgebaut: dem **Prozessor**, abgekürzt **Cpu** (Central Processing Unit), und dem **IO-System** (Input/Output-System). Die Cpu führt sehr schnell Rechenoperationen aus. Diese Operationen benötigen Daten als Input, und es resultieren Daten als Output. Die Cpu verfügt über eine beschränkte Anzahl an Speicherplätzen, **Register** genannt, um diese Daten abzulegen. Werden mehr Speicherplätze gebraucht, was in der Realität immer der Fall ist, werden sie vom **Hauptspeicher**, dem **Ram** (Random Access Memory), bereit gestellt. Die Übertragung von Daten zwischen Ram und Cpu wird durch das IO-System veranlasst. Grafik 1 veranschaulicht diese Beziehung. Die Cpu kann Daten via IO-System in das Ram schreiben, beziehungsweise vom Ram lesen.

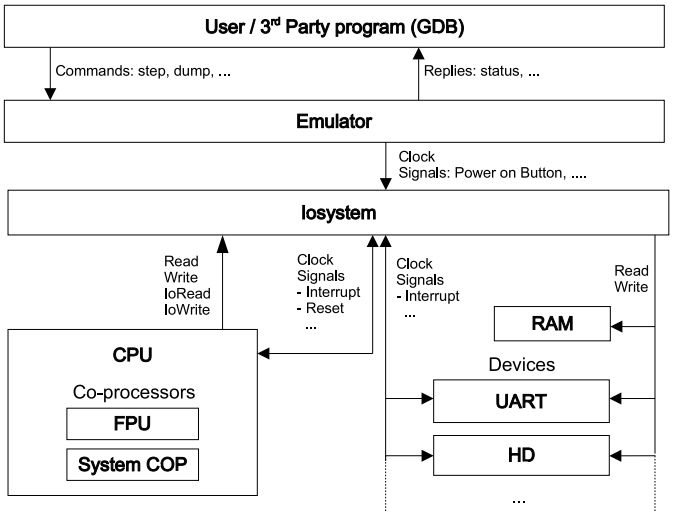


Abbildung 1: Untere Hälfte veranschaulicht sowohl die Realen Hardwarekomponenten, sowie auch deren Implementeationen. Die Interaktionen sind die selben, sowohl in der realen Hardware wie auch in der Implementation. Die obere Hälfte zeigt, das der Emulator ein Interface zwischen User und der Implementation bildet.

2.1.1 I0-System

Aus der Grafik 1 ist ersichtlich, dass das IO-System aus dem Ram und weiteren Komponenten besteht, den **Devices**. Die Devices können beliebige Aufgaben übernehmen: Die Harddisk beispielsweise stellt grosse Mengen an persistentem Speicherplatz zur Verfügung oder die Soundkarte liefert Musik. Allen Devices ist gemeinsam, dass das IO-System die Kommunikation zwischen Cpu und Device übernimmt.
Möchte die Cpu Daten ins Ram schreiben beziehungsweise davon daraus lesen, muss sie dem IO-System mitteilen, an welcher Adresse die Daten abzulegen sind, wie umfangreich die Daten sind und natürlich die Daten selbst. Die Grösse der Daten, die pro Sequenz übertragen werden, beschränkt sich in der Regel auf 1 bis 8 Byte. Die Daten selbst werden über den Datenbus verschickt, deshalb ist die Breite des Datenbus der beschränkende Faktor für diese Grösse. Ein Datenbus mit einer Breite von 32 Bit kann folglich Daten von 1 bis 4 Byte übertragen. Analog beschränkt die Breite des Adressbuses die Grösse der Adresse.

Möchte die Cpu Daten ins Ram schreiben oder davon lesen, muss sie dem IO-System eine Adresse mitliefern. Die Adresse bestimmt, an welcher Stelle im Ram die Daten lokalisiert sind. Das Total möglicher Adressen wird Adressraum genannt. In den meisten aktuellen Architekturen umfasst der **Adressraum** 32 Bit, in einigen bereits 64 Bit.
Möchte die Cpu Daten an die Devices schicken und von ihnen empfangen, hat sie in der Regel zwei Möglichkeiten: Memory mapped IO oder **Port IO**. Bei **Memory mapped IO** wird der normale Adressraum, wie oben beschrieben, benutzt. Jedem Device wird ein spezieller Bereich im Adressraum zugeordnet. Wird auf diesen Bereich zugegriffen, wird nicht - wie üblich - aufs Ram zugegriffen, sondern auf das entsprechende Device. Wird zum Beispiel in einem AT/IO-System auf die Adressen 0xA000-0xAF9F zugegriffen, können damit Bildpunkte der Grafikkarte (im VGA Modus) setzen oder abfragen. Wo im Adressraum diese Bereiche liegen, wird durch das IO-System und/oder dem jeweiligen Device festgelegt.
Bei Port IO wird ein zweiter Adressraum benutzt, der ausschliesslich für Devices verwendet wird. Der Adressraum des Ram und der Devices sind in diesem Fall getrennt. Welche Devices welche Bereiche in diesem Port-Adressraum benutzen, ist wiederum Aufgabe des IO-Systems und/oder des jeweiligen Devices. Port IO wird jedoch nicht von allen CPUs und IO-Systemen unterstützt.

2.1.2 Cpu

Die Hauptaufgabe der Cpu, Rechenoperationen auszuführen, übernimmt die arithmetische Einheit, abgekürzt **ALU**(Arithmetic Logic Unit).In den Registern werden, wie oben erläutert, Zwischenresultate abgelegt. **Co-Prozessoren** helfen der Cpu in bestimmten Aufgabenbereichen. Die **FPU** (Floating Point Unit) beispielsweise, führt Rechenoperationen mit Fliesskommazahlen aus. Der **System Co-Processor** führt spezielle Aufgaben aus, die von Betriebsystemen benutzt werden. Welche Co-Prozessoren eine bestimmte Cpu implementieren, ist ihnen vollkommen frei gestellt. Die zwei letztgenannten (Subjekt fehlt) sind häufig anzutreffen, sind jedoch keinesfalls zwingend vorgeschrieben.

2.1.3 Signale

Die Kommunikation zwischen Cpu und IO-System beschränkt sich nicht auf den Datenaustausch. Bekannte Beipsiele sind **Interrupts**. Im bisher definierten Datenaustausch ist immer die Cpu der Initiator. Braucht jedoch das IO-System die Aufmerksamkeit der Cpu, weil ein Device Daten für die Cpu bereitgestellt hat, signalisiert es dies mit einem Interrupt. Ein anderes Beispiel ist das folgende: Das IO-System kann der Cpu kann mit einem Error- Signal anzeigen, ob der von der Cpu verlangte Datenaustausch erfolgreich war oder nicht. In jedem dieser Fälle erfolgt die Kommunikation via Signale.

2.2 API Design

Modularität ist eine zentrale Eigenschaft dieses Emulators. Dies bedeutet in erster Linie, dass er nicht auf die Emulation einer bestimmten Architektur beschränkt ist. **Architektur** bezeichnet ein ganzes Computersystem, der Verbund einer bestimmten Cpu und eines bestimmten IO-Systems. Der Emulator muss mit verschiedenen Implementationen von Cpu's und IO-Systemen umgehen können. Vor allem müssen die Implementationen der Cpu und des IO-Systems untereinander kommunizieren können, ohne jeweils die andere Implementation zu kennen. Eine solche Problemstellung wird am besten mit einer objektorientierten Sprache wie C++ mit abstrakten Klassen gelöst.
Im folgenden bezeichnet **API** (Application Programmming Interface) das Software-Interface, das heisst die Deklaration der C++ Klassen. Mit **Interface** allein ist das Hardware-Interface zwischen zwei Hardware-Komponenten gemeint, oder dessen Repräsentation als C++ Klasse. In Anlehnung an die Grafik 1, welche die Beziehung des Cpu zum IO-System aus Hardwaresicht zeigt, veranschaulicht die Grafik 2 deren Umsetzung in das API.

CEmulator steuert das IO-System, welches dann die Cpu steuert. Dieses Steuern ist im Normalfall auf das Auslösen des Clock-Impulses durch Clock reduziert. Der Emulator stellt ein API zwischen emulierter Hardware und Aussenwelt in mehrfacher Hinsicht dar:

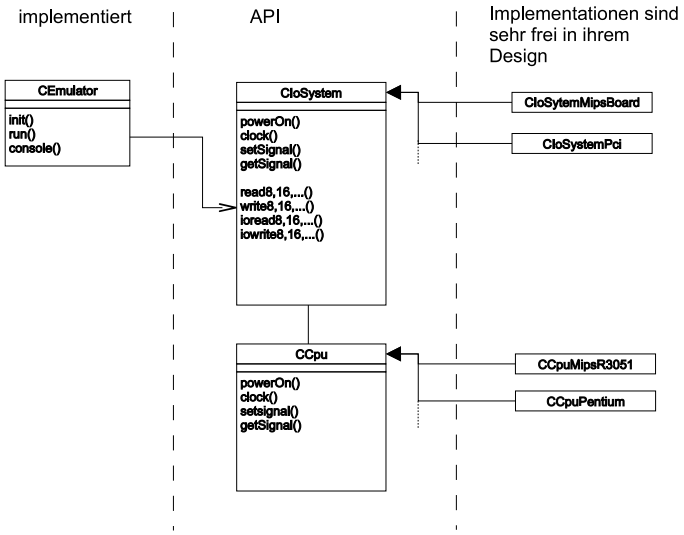


Abbildung 2: Übersicht der wichtigsten Klassen. CCpu und CIoSystem bilden ein API zwischen dem Emulator und einer spezifischen Implementation einer Architektur.

- Alle Knöpfe des Computers, beispielsweise Power-On oder Reset, werden vom Emulator emuliert. Drückt der User einen dieser Knöpfe, wird dies via Set-Signal() an das I0-System weitergeleitet.
- Kommandozeilen-Argumente, die dem Emulator beim Programmstart gegeben werden, werden an CCpu und CIoSystem per Init-Methode weitergeleitet.
- Während der Ausführung kann der User Befehle eingeben,zum Beispiel “internen Status ausgeben” oder “ein Clock-Zyklus voranschreiten”.

Eine weitere wichtige Aufgabe des Emulators ist die Unterstützung im Debuggen der emulierten Software und im Debuggen der Implementation einer Architektur.

CIoSystem steuert CCpu und die Devices. Im Normalfall bedeuted dies, dass sie den Clock-Impuls via Clock-Methode an die Cpu und die Devices weiterreicht. Die zweite wichtige Aufgabe liegt im Versorgen der Cpu mit Daten durch die Methoden readx/writex für Memory mapped IO und readI0x/writeI0x für den Port IO. x steht für die Anzahl der zu übertragenden Bytes. Für weitere, beliebige Kommunikation zwischen Cpu und I0-System sind die Methoden setSignal/getSignal zuständig. Das I0-System ist frei im Design seiner Devices, deshalb sind diese in der API nicht eingezeichnet.

CCpu bewältigt die Hauptaufgabe: Ausführen von Instruktionen, die sie via IO-System aus dem Ram holt. Die CCpu ist frei im Design ihrer ALUs und Koprozessoren; daher sind dieser in der API nicht eingezeichnet.

CCpu/I0-Interface ist für das Mapping von Signalen zwischen IO-System und Cpu zuständig.

Die Klassen CCpu, CIoSystem und CCpuI0-Interface haben genau eine Instanz, die global defininiert ist. Diese Instanzen haben immer den gleichen Namen - **Cpu**, **IoSystem** und **CpuIoInterface** - unabhängig davon, welche spezifische Architektur sie implementieren.
Der Leser beachte, dass die API sehr nahe an die reale Hardware anlehnt. Abgesehen von den Read/Write-Methoden kommunizieren die Cpu und das I0-System nur mittels setSignal/getSignal. Auch das I0-System empfängt Befehle von aussen nur via setSignal. Es werden also keine Methoden wie Reset,oder SetInterrupt verwendet, die eine bestimmte Funktionalität implizieren. Der Grund liegt darin, die API möglichst allgemeingültig zu halten, um so ein grösstmögliches Mass an Modularität zu erreichen.
Auffällig ist, dass der Emulator nicht mit der Cpu direkt kommuniziert. Der Grund liegt darin, dass auch in der realen Hardware die Cpu vom I0-System kontrolliert wird. Das I0-System

soll die Kontrolle darüber führen, wie oft die Cpu und die Devices den Clock bekommen, und wann genau es Signale an die Cpu oder die Devices schickt. Dieses Schema kann von der Implementation der Cpu weitergeführt werden, so dass auch die Co-Prozessoren von der eigentlichen Cpu gesteuert werden. Allgemein gesprochen sind alle Komponenten des gesamten Systems hierarchisch geordnet, wobei das IO-System die Rootkomponente darstellt.
Einige Methoden sind allen Hardware-Komponenten gemeinsam, wie zum Beispiel Dump oder PowerOn. Auf die genaue Bedeutung dieser Methoden wird später eingegangen; sie ist hier noch nicht von Interesse. Durch den im letzten Absatz erwähnten hierarchischen Aufbau der Hardware-Komponenten, muss der Emulator nur die Methode des IO-System aufrufen, um alle Komponenten des gesamten Systems zu erreichen. Jede Komponente muss einfach die entsprechende Methode ihrer untergeordenten Komponenten aufrufen.

2.3 Wiederverwendbare Komponenten

Im Folgenden geht es darum, wie dem Programmmierer einer neuen Architektur schon möglichst viele Code zur Verfügung gestellt werden kann. Wiederverwenden von bestehendem Code bedeutet weniger Schreib- und Designarbeit. Des weiteren wurde er schon einmal auf Fehlerlosigkeit überprüft. Es bedeutet nicht, dass keine Fehler mehr vorhanden sind, aber die grössten Fehler dürften beseitigt sein.
Alle in der Grafik 2 vorgestellten Klassen sind bis auf CEmulator abstrakte Klassen, haben also noch keinerlei Funktionalität vorprogrammmiert. Dies deshalb, weil die verschiedenen Architukturen einfach zu verschieden sein können. Zum Beipsiel wird der Clock-Zyklus einer Cpu mit Pipeline komplett verschieden sein von dem einer Cpu ohne Pipeline. Trotzdem haben verschiedene CPUs beziehungsweise I0-Systeme gewisse Gemeinsamkeiten. Diese Gemeinsamkeiten liegen aber eher in Teilaufgaben. So sind die arithmetisch logischen Befehle wie Addieren, Dividieren, Shift etc. bei praktisch allen CPUs wiederzufinden. Auch das Setzen der Flags wird beinahe ausnahmslos nach demselben Muster vollzogen.
Kleine, wiederverwendbare Komponenten haben den Vorteil, dass einige davon benutzt werden können, andere nicht.Im Baukastensystem kann eine neue Implementierung implementiert werden.
Die anschliessende Auflistung soll einen Überblick über die verfügbaren und wiederverwendbaren Komponenten geben. Die weiteren Kapitel dieses Dokumentes gehen dann stärker auf die einzelnen Komponenten ein. Die nachfolgend verwendeten Begriffe wie Instruktion oder Assembler Mnemonic, die bis jetzt noch nicht eingeführt worden sind, werden im jeweiligen Kapitel erklärt.

CAlu stellt eine Vielzahl von Operationen zur Verfügung, die den Kern einer Instruktion, das eigentliche Ausführen der arithmetischen/logischen Funktion, und das Setzen der Flags modelieren.

CPUEnhanced Stellt eine Templatemethode zur Verfügung, die den Ablauf einer Instruktion modeliert.

CInstruction stellt Methoden zur Verfügung, mit denen Assembler Mnemonics von beliebigen Prozessoren in das jeweilige Instruktionswort assembliert. Auch der umgekehrte Weg, das Disassemblieren, wird unterstützt.

CDevice / CDeviceInterface

3 Design Überlegungen

Es werden einige allgemeine Ideen vorgestellt, die mich beim Desgin der Interfaces und der Mips-Architektur geleitet haben. Sie dienen auch als Leitfaden bei der Implementierung einer neue Architektur.

3.1 Modular

Das Design von Implementierten Hardwarekomponenten ist so nahe wie möglich bei der entsprechenden realen Hardwarekomponente. Dies wird auch durch Grafik 1 verdeutlicht. Die untere Hälfte wiederspiegelt sowohl die reale Hardware, wie auch das Design der Implementation. Jede Komponente kommt sowohl in der realen Hardware als auch in der Implementation als eigenständige Komponente daher. Die Interaktionen sind auf beiden Seiten dieselben: Die Komponenten kommunizieren hauptsächlich mit Signalen. Die Cpu kann Daten vom IO-System anfragen, bzw damit Devices steuern. Dies geschiet mit read/write für Memmory Mapped IO und ioread/iowrite für Port IO.
Die Komponenten sind hierarchisch strukturiert. Das IO-System kontrolliert die Devices und die Cpu. Die Cpu kontrolliert ihre Co-Prozessoren und ihre Alu. Je nach Implementation kann sich diese Hierarchie fortsetzen. Wegen dieser Hierarchi muss der Emulator nur mit dem IO-System kommunizieren, da dieses die Wurzel der Hierarchie bildet.
Grafik 2 hat denselben Informationsgehalt wie Grafik 1. Grafik 2 zeigt die Sicht des API. Es sei verdeutlicht, dass die angegebenen Methoden genügen, die Funktonalität der Hardware zu modelieren.
Weil die Hardware so nahe modeliert wird, und weil die Basisklassen so klein sind, ist das Design modular. Denn neue Implementationen haben nur sehr wenige Vorgaben zu erfüllen. Es gibt keine Methoden, die eine spezielle Funktionalität vordern, die eine spezielle Architektur nicht erfüllen könnte. Gäbe es zum Beispiel eine reset Methode, ist durchaus denkbar, dass es Architekturen gibt, die eine solche Funktionalität gar nicht anbieted. Dies zum Beispiel, weil die Architektur implizit gereseted wird, sobald sie mit Spannung versorgt wird.

3.2 Portabel

Die Implementation wurde in ANSI C++ programmiert. Dies wurde jedoch nicht verifiziert.
Damit die Implementation portabel ist, dürfen keine Assemblerinlines verwendet werden, obwohl dies im Hinblick auf die Performance sehr nützlich wäre.
Die Datentypen TBit und TReg werden benützt, um Aliases für die Kompilerabhängigen primitiven Datentypen wie short,int und long zu Bilden. Kapitel 4.3 geht näher auf diese Thematik ein.
Es ist anzumerken, dass für viele Compiler ein 64 Bit Typ als “long long” definiert werden muss; dies ist jedoch kein vom C++ Standard unterstützter Datentyp. Wird long long trotzdem verwendet, schränkt dies die Portabilität des Emulators ein. Da jedoch für einen 64 Bit Typ fast immer long long verwendet werden muss, bleibt nur der Ausweg, TBit64 nicht zu verwenden. Dies ist jedoch kaum machbar, soll eine Cpu mit 64-Bit-Registern implementiert werden. Auch der Zähler der Clock sollte 64 Bit gross sein. Mit 32 Bit ist der Zähler, gesetzt der Fall dass 1 Million Instruktionen pro Sekunde emuliert werden, was sehr schnell aber realistisch ist, bereits nach 72 Minuten am Ende angelangt. Da Portabilität keine hohe Priorität hatte, und die meisten Compiler für ein 32Bit System einen 64 Bit Datentyp unterstützten, wurde dieses Problem nicht weiter verfolgt.

3.3 Performance

Hinsichtlich dem Verhältniss zwischen Komplexität und Performance, wurden für die Cpu und für die Devices unterschiedliche Ansätze gewählt. Die Cpu ist schnell, dafür aber komplex. Devices hingegen haben ein schönes OO Design, sind dafür jedoch langsamer. Dieser Ansatz macht durchaus Sinn. Die Cpu wird sehr viel öfter benutzt als die Devices. Desshalb muss sie schnell sein. Die Devices hingegen werden eher selten benutzt. Sie sind ausserdem auch in der realen Hardware eher langsam.

Um die Performance der Cpu zu erhöhen, wurde dort auf virtuelle Methoden verzichted. Dies bringt jedoch die erwähnte Komplexität mit sich. Bei den Devices lohnt es sich nicht, aus Performancegründen auf virtuelle Methoden zu verzichten. Denn schönes OO Design wird erst durch virtuelle Methoden möglich.
Wie in Kapitel 8.2 gezeigt wird, werden die Devices in mehrfacher Hinsicht wiederverwendet. Desshalb ist es angebracht, dass sie ein durchdachtes OO Design haben.

4 Miscellaneous

In diesem Kapitel werden Konzepte und dazugehörige Klassen vorgestellt, die globale Gültigkeit haben. Es sind dies Klassen, die im Vergleich zu CCpu, CIoSystem und CEmulator eine eher untergeordnete Rolle spielen, jedoch wichtig genug sind, um in dieser Dokumentation erwähnt zu werden.

4.1 Signale

`Source Signal.h`

Um die Modularität zu erhöhen, kommunizieren Implementationen von Hardware-Komponenten über Signale. Alle Klassendeklarationen, die eine Hardware-Komponente repräsentieren, haben eine setSignal- und eine getSignal- Methode, wie sie im nachfolgenden Code deklariert ist. Jede Klasse vergibt die Nummern für ihre Input- und Output-Signale selbst. Zum Beipsiel definiert CCpuMipsR3051 die Konstante SIGNAL_RESET für ihr eigenes reset Inputsignal.

```
enum ESignalState
{ DE_ASSERTED,
  ASSERTED
};
```

void setSignal(int no, ESignalState s);
Setzt ein Inputsignal einer Hardwarekomponente.
ESignalState getSignal(int no);
Fragt ein Inputsignal einer Hardwarekomponente ab.

4.2 Fehlerbehandlung

Bei der Fehlerbehandlung, müssen zwei grundsätzliche Fehler unterschieden werden: Fehler, die auch in der realen Hardware vorkommen können, und Fehler, die von einer fehlerhaften Implementation herrühren. Beispiele für den ersten Fall sind versuchtes Ausführen eines ungültigen Instruktionswortes, Zugriff auf eine ungültige Speicheradresse oder wenn im Usermode auf Resourcen zugegriffen wird, die nur im Kernelmode zugänglich sind. Ein Beispiel für den zweiten Fall ist, wenn der Parameter einer Methode ausserhalb des gültigen Wertebereichs liegt.
Wie mit den Fehlern der ersten Art umzugehen ist, ist einfach: die Spezifikation der Hardware gibt vor, was zu tun ist. Die Implementation muss sich genau so verhalten, wie die reale Hardware, und dieses Verhalten ist durch die Spezifikation vorgegeben. Der MipsR3051 löst zum Beispiel eine “Reserved Instruction” Exception aus, wenn das Instruktionswort ungültig ist. Ein IO-System, das von der Cpu eine ungültige Adresse erhält, wird meist ein Error-Signal an die Cpu auf High setzen. Signale können oft zur Fehlerbehandlung eingesetzt werden. Das sollten sie auch, wenn die Spezifikation es so vorsieht.
Tritt ein Fehler zweiter Art auf, ist es oft am Besten, die Ausführung des Programmes abzubrechen und in einer Fehlermeldung den Ort im Sourcecode, wo der Fehler aufgetreten ist, anzugeben. Dazu wird das ASSERT Makro benutzt, auf das anschliessend tiefer eingegangen wird. Wenn ein Debugger benutzt wird, kann im ASSERT Makro ein Breakpoint gesetzt werden; somit wird die Ausführung des Programms vom Debugger unterbrochen, sobald ein Fehler aufgetreten ist, und der Programmierer kann alle aktuellen Variablen auslesen und somit hoffentlich die Ursache des Fehlers bestimmen.

4.2.1 Undefined / Unpredictable Behaviour

Einige Fehler haben gemäss Spezifikation jedoch “undefined” oder “unpredictable” Verhalten zur Folge. Diese Therme werden im Detail von der jeweiligen Spezifikation spezifiziert; im Allgemeinen ist damit gemeint, dass der Inhalt der Register und das weitere Verhalten der Hardware in einem solchen Fall eben “undefiniert” ist, kann also irgendetwas sein. Dies lässt sich am Besten nachbilden, indem alle Register mit zufälligen Werten beschrieben werden. Auf jeden Fall sollte sich die Hardware mit einem Flag merken, dass undefiniertes Verhalten aufgetreten ist.

An dieser Stelle wird die Methode shallEmulatorWait kurz eingeführt. Eine detailierte Beschreibung ist in Kapitel 5.4.1. Diese Methode gibt true zurück, wenn der Emulator die Ausführung des emulierten Programmes unterbrechen soll. In diesem Fall gibt der Emulator dem User dann die Möglichkeit, den aktuellen Status der Hardware auszulesen. Ausserdem weiss der User, in welchem Clockzyklus das undefinierte Verhalten aufgetreten ist. Siehe auch Kapitel 5.4, welches solche Debugtechniken genauer erläutert.

4.2.2 ASSERT Makro

```
Source                  Error.h, .cpp
Aufruf                  ASSERT( cond )
```

Das ASSERT Makro wird benutzt, um Fehler in der Implementation zu finden. Ist der Ausdruck cond false, wird die Zeile und das File, in dem das ASSERT Marko steht, in einer Fehlermeldung ausgegeben und das Programm angehalten. Dies kann benutzt werden, um sich “zusichern” zu lassen, dass eine Variable einen bestimmten Wert hat oder in einem bestimmten Wertebereich liegt. Der Leser findet im beigelegten Code viele Beispiele zur Verwendung von ASSERT. Das ASSERT Makros wird jedoch nur dann vom Compiler verwendet, wenn im Konfigurations-Headerfile DEBUG_ASSERT auf 1 ist. Ansonsten werden vom Preprozessor alle ASSERT Makros entfernt, und es wird kein Code dafür generiert. Da, zumindestens theoretisch, in der Retailversion eines Programmes keine Fehler mehr in der Implementation vorhanden sind, braucht es auch die ASSERT Tests nicht mehr. Somit wird auch die Performance der Retailversion gesteigert, da diese Tests wegfallen. Siehe Kapitel 5.4.4, wie ASSERT für das Debugging verwendet werden kann.

4.2.3 Class CError

```
Source                  Error.h, .cpp
```

Um Fehler zurückzugeben, verwenden das IO-System und die Devices die Klasse CError. Nach jedem Aufruf der write/read-Methode muss mit getError abgefragt werden, ob die read/write-Methode erfolgreich war. Ist ein Fehler aufgetreten, muss er behandelt und anschliessend mit clearError zurückgesetzt werden.

virtual CError* getError();
Gibt null Pointer zurück, wenn kein Fehler aufgetreten ist. Ansonsten wird ein Pointer auf eine Instanz der CError Klasse zurückgegeben, die Informationen über den aufgetretenen Fehler enthält.

virtual void clearError();
Löscht den Fehlerstatus.

Damit die Instanz, auf die der zurückgegebene CError-Pointer zeigt, jederzeit verfügbar ist, und nicht etwa bereits mit dem Operator delete gelöscht wurde, sollte die Instanz dieselbe Lebensdauer wie das Device bzw. das IOSystem haben.
Die Verwendung der CError Klasse widerspricht der Einleitung, wo davon gesprochen wird, dass Hardware-Komponenten Fehler mit Signalen zurückgeben sollen. In der Tat sollte die eben beschriebene Methode der Fehlerrückgabe aus dem Projekt verbannt werden. Erstens, weil die Verwendung von Signalen näher bei der Hardware liegt. Zweitens, weil mit der CError-Methode, nach jedem Aufruf von read/write auch getError aufgerufen werden muss. Da bei jedem Ausführen einer Intruktion mindestens einmal (Laden der Instruktion) mit read aufs IO-System zugegriffen wird, wäre es aus Performancegründen durchaus wünschenswert, getError nicht jedesmal aufrufen zu müssen. Die Eingangsignale der Cpu müssen hingegen jedoch in jeder Instruktion überprüft werden. Aus Zeitgründen wurde die Verwendung von CError noch nicht durch das Verwenden von Signalen ersetzt.

4.3 Datentypen TBit, TReg

```
Source                  Host-Headerfile
```

```
                        Beispiel: Host/Ia32/HostIa32.h
```

C++ definiert die Grösse der primitiven Datentypen char, short, int und long nicht. Welche Grösse char, short, int und long haben, ist eine Eigenschaft des verwendeten Compilers. Hierbei wird GCC der für einen Ia32 Code erzeugt, als ein “anderer” Compiler angesehen als GCC, der für einen Sparc Code erzeugt. Wenn es nun aber für das Programm wichtig ist, dass eine Variable eine ganz bestimmte Grösse hat, dann müssen mit Hilfe von typedefs neue Datentypen erstellt werden, wie das nachfolgende Beispiel zeigt:

```
typedef unsigned short      TBitu16;
typedef short               TBits16;
```

TBit und TReg gibt es jeweils in einer unsigned und einer signed Variante, und in Grössen von 8,16,32 oder 64 Bit. Zum Beispiel TBitu32 für unsigned 32 Bit oder TBits8 für signed 8 Bit.
Der Unteschied zwischen TBit und TReg ist der, dass TBit genau so gross ist, wie es der Name besagt - TBitu32 ist genau 32 Bit gross - während TReg mindestens so gross ist. Die eventuell vorhandenen extra Bits von TReg sind 0 für unsigned Werte und 0 oder 1 für signed Werte. Der tatsächliche Wert einer TReg Variable muss immer dem beabsichtigten Wert entsprechen. Der Clou an TReg ist: Muss ein 8 Bit Register emuliert werden, kann dies mit einer Variablen in der Grösse der Register der Host Cpu getan werden. Deshalb werden TReg Typen meist als int definiert, da in C++ ein int der Registergrösse des Host entspricht. Dies erhöht die Performance des Emulators. Dies bedingt jedoch, dass der Programmmierer dafür sorgt, dass die eventuell vorhandenen extra Bits einen korrekten Wert aufweisen. Dazu werden die Methoden signExtend und zeroExtend benutzt. Trotz des Mehraufwandes, denke ich, dass sich der Datentyp TReg lohnt. Die genannten Methoden weden nur dann benötigt, wenn sich nach einer Operation die extra Bits verändert haben könnten, was nicht oft der Fall ist. Eine Veränderung kommt allenfalls in der Alu der Cpu vor. Der Compiler weiss dies jedoch nicht. Würden ausschliesslich TBit Typen verwendet, würde der Compiler im Fall von TBits8 andauern, sofern die Variable nicht gerade temporär in einem Register abgelegt ist. Vor jedem Zugriff die Variable vorzeichenerweitern auf die interne Registergrösse.
Die oben genannten typedefs müssen für jeden Host, für den der Emulator compiliert werden soll, angepasst werden. Genau genommen für jeden Compiler, da die Grössen der primitiven Datentypen Compilerabhängig sind. Damit diese Anpassung möglichst einfach vor sich geht, und pro Compiler nur einmal gemacht werden muss, sind diese Defines in einem Host-Headerfile definiert. Dies wird in Kapitel 10.3 erläutert.

4.4 Datentypen TVAddr, TPAddr, TIoAddr

```
Source                  Konfigurationsheaderfile
                        Beispiel: Configs/ConfigMipsDebug.h
```

Da Adressen (physikalische, virtuelle oder Port IO) in vielen verschiedenen Komponenten verwendet werden, ist deren Typ global mit typedef in Config.h definiert. Siehfe Tabelle 1 Somit ist der Emulator flexibel für verschieden grosse Adressysteme.

TVAddr	Virtuelle Adresse. Eigenschaft der Cpu.
TPAddr	Physikalische Adresse. Eigenschaft der Cpu und des IO-Systems.
TIoAddr	Port IO Adresse.Eigenschaft der Cpu und des IO-Systems.

Tabelle 1: Addresstypen

Für einen 32 Bit Adressraum, und 16 Bit Port IO Adressraum, sieht der entsprechende Teil in Config.h zum Beispiel folgendermassen aus:

```
typedef TRegu32 TPAddr;
typedef TRegu32 TVAddr;
typedef TRegu16 TIoAddr;
```

4.5 Class CClock, Methode clock

`Source` `Emulator/Clock.h`

Die Aufgabe der Klasse CClock liegt im Zählen der verstrichenen Clockzyklen. CClock ist global definiert, deshalb ist der aktuelle Zählstand von überall her abrufbar. Ein Clockzyklus entspricht dem Aufruf der Methode clock von CIoSystem.
Jede Klasse, die eine Hardware-Komponente repräsentiert, besitzt die Methode Clock. Diese Methode soll das tun, was die Hardware in einem Clockzyklus tut. Für eine Cpu kann das heissen, eine ganze Instruktion auszuführen, oder einen Stage der Pipeline abzuarbeiten.
Das “Weiterreichen” des Clockimpules ist hierarchisch aufgebaut. Der Emulator ruft Clock des IO-Systems auf, welches dann Clock der Cpu und der Devices aufruft. Je nach Implementation der Cpu und der Devices, können diese den Clockimpuls auch an ihre untergeordneten Komponenten verteilen.
An dieser Stelle sei der Leser auf einen Fehler im Design der CClock Klasse hingewiesen. Das IO-System ist frei, den Clockimpuls in einer anderen Frequenz an die Cpu oder an die Devices weiterzureichen. CClock ist also nur die “Grundfrequenz”.

5 Emulator

Dieses Kapitel beschäftigt sich mit den Aufgaben der CEmulator Klasse. Ihre Aufgabe ist erstens die Steuerung des I0-Systems, welches seinerseits die Cpu steuert. Zweitens Bilden eines Interfaces zwischen User und emulierter Hardware und drittens Bereitstellen von Funktionalität, um die emulierte Software sowie die Implementation einer bestimmten Architektur zu debuggen.

5.1 Class CEmulator

`Source` `Emulator/Emulator.h, .cpp`

Die Klasse CEmulator hat drei wichtige Aufgaben, die sich zum Teil überschneiden.

- Steuert das IO-System aus Hardwaresicht.
  - Gibt den Clockimpuls ans IO-System
  - Setzt Eingangssignale des IO-Systems. Diese Signale sind beispielsweise Reset oder PowerOn.

  Das IO-System steuert dann seinerseits die Cpu.
- Ist Interface zwischen User und Software von dritter Seite einerseits und Cpu / I0-System andererseits.
  - Kommandozeilen-Argumente, mit denen der Emulator gestartet wurde, werden an Cpu und I0-System weitergegeben, um diese zu initialisieren.
  - User kann via Konsole des Emulators Cpu und I0-System steuern und dessen Status abfragen.
  - DrittProgrammme wie zum Beispiel der Debugger GDB kommunizieren via Emulator mit der Hardware. Dies ist jedoch noch nicht implementiert.
- Hilft beim Debuggen der Implementation einer Architektur und beim Debuggen der emulierten Software.

5.2 Initialisations-Reihenfolge

Anhand von Pseudo-Codes soll dem Leser aufgezeigt werden, wie der Emulator alle Komponenten initialisiert, und wie das Aufstarten des Systems vor sich geht.

```
/* --- file Globals.cpp --- */
// 1) constructors are called even before main method
CONFIG_CPU              Cpu            CONFIG_CPU_ARGS;
CONFIG_IOSYSTEM         IoSystem       CONFIG_IOSYSTEM_ARGS;
CONFIG_CPU_IO_INTERFACE CpuIoInterface CONFIG_CPU_IO_INTERFACE_ARGS;

/* --- file CEmulator.cpp --- */
// 2) intialize all components with command line arguments
CEmulator::init(...)
{ Cpu.init(...)
  IoSystem.init(...)
}

CEmulator::run()
{ // 3) give power to the computer
  IoSystem.powerOn()
  for (i=0; i<POWER_ON_CYCLES; i++ )
    clock()
```

```
  // 4) press computer's power on button
  IoSystem.setSignal( SIGNAL_POWER_ON_BUTTON, ASSERTED )
  for (i=0; i<; i++ )
    clock()
  IoSystem.setSignal( SIGNAL_POWER_ON_BUTTON, DE_ASSERTED )

  // 5)
  while(1)
    clock()
}

CEmulator::clock()
{ IoSystem.clock()
  //...emulator specific actions
}
```

1) Da die Cpu-, I0-System- und Cpu-I0-Interface-Instanzen global definiert sind, werden deren Konstruktoren implizit beim Aufstarten des Emulator-Programmes aufgerufen. Die Konstruktoren werden noch vor der main Methode ausgeführt. Der Typ der Instanzen (CONFIG_CPU etc.) sowie die Argumente der Konstruktoren (CONFIG_CPU_ARGS etc) sind als #define Konstanten in Config.h definiert.

2) Als nächstes werden Kommandozeilen-Argumente, mit denen der Emulator aufgerufen worden ist, verarbeitet. Argumente, die mit "-CPU" beginnen, werden an die Cpu, Argumente mit "-IO" oder "-DEV" beginnend, werden an das IO-System weitergeleitet. Das IOSystem leitet Argumente beginnend mit "-DEV" an die Devices weiter.

3) Versorgt des I0-Systems mit Spannung durch die Methode PowerOn. Dies ist noch nicht das Einschalten des Systems, sondern erst das Einstecken des Stromkabels; d.h. die Versorgung mit Spannung.

4) Anschliessend wird automatisch der PowerOn-Knopf gedrückt, um das System einzuschalten. Ab wann genau die Cpu und die Devices mit Spannung versorgt werden, wann sie eventuell benötigte Resetsignale erhalten, und ab welchem Zeitpunkt sie den Clockimpuls erhalten, ist Sache des IO-Systems. Beliebige dieser Aktionen können auch schon in Schritt 3 oder erst in Schritt 4 erfolgen.

5) Es ist falsch anzunehmen, ab diesem Zeitpunkt beginne die Ausführung von Instruktionen aus dem BIOS oder vom OS Kernel. Es hängt vielmehr von der Implementation des I0-Systems ab, ab welchem Zeitpunkt damit begonnen wird. Im Normalfall wird dies ein oder mehrere Zyklen nach dem Drücken des PowerOn-Knopfes sein, je nachdem wieviele Zyklen das Reset Signal (das das I0-System der Cpu im Normalfall asserten wird) asserted sein muss, bis die Cpu anfängt, Instruktionen ab einer von der Cpu definierten Adresse auszuführen.

In jedem Clock-Zyklus wird der Clockimpuls zuerst an das IO-System geleitet, danach hat der Emulator selbst Gelegenheit, irgendwelche Aufgaben auszuführen wie das Ausgeben von Debugg-Informationen oder das Empfangen von Kommandos.
Diese Initalisierung scheint auf den ersten Blick relativ umständlich. Sie ist jedoch dem Prinzip treu, dass sich die API sehr nahe an die realen Hardware-Gegebenheiten zu halten hat, um ein hohes Mass an Modularität zu erreichen.

5.2.1 Methoden init, powerOn, setSignal

Die Methoden konstructor, init, powerOn, setSignal finden sich in allen Klassen-Definitionen wieder, die Hardware-Komponenten repräsentieren. Nachfolgend soll aufgezeigt werden, welche Methode welche Initialisierungen vornehmen sollte. Der Sinn dieser "Vorgaben" ist, dem Programmierer neuer Implementationen eine Idee zu geben, wie er seine Implementation möglichst hardwaregetreu initialsieren kann. Es besteht keine Notwendigkeit, Implementationen genau so zu initialisieren.

Konstruktor
Soll implementationsspezifische Initialisierungen vornehmen, wie Speicher allozieren oder Lookup-Tables initialisieren. Beispiel CCpuMipsR3051: Die Instruktionstabellen werden intialisiert.
init(int argc, char* argv[])
Initialisiert die Implementation mit Kommandozeilen-Argumenten. Der erfahrene C/C++ Programmierer wird die Funktionsparameter sofort erkennen: es sind dieselben, mit denen die main aufgerufen wird; deshalb wird hier nicht auf deren Bedeutung eingegangen. Diese Kommandozeilen-Argumente können einerseits verwendet werden, um das Verhalten der Implementation zu steuern. Zum Beispiel vieviel Debugg-Information ausgegeben werden soll, oder wie exakt die Hardware zu emulieren ist. Andererseits können eventuell vorhandene ROMs (insbesondere das BIOS) mit Daten aus Dateien initialisiert werden, deren Dateinamen als Argument übergeben werden.
Beispiel CIoSystemMips: Dumplevel wird gesetzt. Anhand des Konfigurationsheaderfile werden alle Devices erstellt und anhand der Kommandozeilen-Argumenten initialisiert, indem alle Argumente beginnent mit “-DEV´´ an die init Methode des entsprechenden Devices übergeben werden. Argument “-IOki image´´ gibt Kernelimage an, das ins Ram geladen wird.
In der aktuellen Version werden die init Methoden noch nicht korrekt verwendet. Anstatt die Optionen in Form von Kommandozeilenargumenten zu übergeben, werden die einzelnen Elemente des argv Parameters direkt mit dem Wert der Option gesetzt. Dies wurde so implementiert, weil es einfacher ist, und somit wertvolle Zeit spart.
Beispiel CCpuMipsR3051: Dumplevel wird gesetzt
powerOn()
Versorgt die Hardware-Komponente mit Spannung. Diese Methode soll genau das machen, was die Spezifikation der Hardware vorgibt. In den meisten Fällen bedeuted dies, dass einige Register einen definierten Wert erhalten, andere undefiniert bleiben. Um diese “Undefiniertheit” hervorzuheben, wird vorgeschlagen, dem Inhalt solcher Register einen zufälligen Wert zuzuweisen. Falls die zu emulierende Software fälschlicherweise von initialisierten Registern ausgeht, wird dieser Fehler schneller entdeckt.
Beispiel CIoSystemMips: Deasserted alle Signale zur Cpu und zu den Devicen. Versorgt die Cpu und Devices jedoch nocht nicht mit Spannung.
Beispiel CCpuMipsR3051: Initialisiert alle Register mit zufälligen Werten. Ruft powerOn der Co-Prozessoren auf.
Beispiel CCopMipsR3051Cp0: Intialisiert alle Register mit zufälligen Werten, ausser dem konstantem, read-only PRID Register (bestimmt die Version des Prozessors); dieses wird mit Werten aus dem Konfigurationsheaderfile initialisiert.
setSignal(int no, ESignalState s)
Viele Hardware-Komponenten werden erst durch ein Resetsignal richtig intialisiert. Wiederum gibt die Spezifikation der Hardware vor, was genau zu tun ist. Neben dem Resetsignale können natürlich auch andere Signale können einen initialisierenden Charakter haben. In jedem Fall wird die setSignal Methode benutzt (es gibt keine reset Methode).
Beispiel CIoSystemMips, no==SIGNAL_POWER_ON_BUTTON: Versorgt Cpu und Devices mit Spannung. Asserted das Resetsignal der Cpu.
Beispiel CCpuMipsR3051, no==SIGNAL_RESET: Löst die Resetexception aus, welche einige wenige Register mit festgelegten Werten intialisiert.

5.3 Debuggen der emulierten Software

Eine wichtige Aufgabe eines Emulators ist es, Hilfe beim Debuggen der emulierten Software zu bieten. Da der GDB-Debugger der GNU-Tools dies schon vortrefflich beherrscht, und auch schon eine Funktionalität implementiert ein remote target zu debuggen, ist er erste Wahl für diese Aufgabe.
Ein Interface zum GDB wurde jedoch noch nicht implementiert. Trotzdem beherrscht die jetzige Version rudimentäre Features, die zum Debuggen benutzt werden können. Da von Anfang an klar war, dass GDB einmal die Hauptaufgabe im Debuggen übernehmen wird, wurden diese Featers nicht weiter ausgebaut. Es sind dies auch Features, die in erster Linie gebraucht werden, um die implementierte Architektur zu debuggen, aber eben auch zum Debuggen der emulierten Software gebraucht werden können.

5.4 Debuggen der implementierten Architektur

Der Emulator bieted hilfreiche Features, um die implementierte Architektur zu debuggen:

- Die Ausführung des emulierten Programmes kann aufgrund verschiedener Bedingungen unterbrochen werden.
- Im angehaltenen Zustand können dem Emulator über die Konsole verschiedene Befehle eingegegeben werden, um der Ursache des Fehlers auf den Grund zu gehen. Wahlweise kann auch ein konventioneller I0-System wie GDB oder dessen Frontend DDD benutzt werden, sobald eine solche Bedingung eingetroffen ist.
- Nach jedem Clockzyklus kann der interne Status der Implementation automatisch mit einem Verifyfile verifiziert werden.
- Nach jedem Clockzyklus kann der interne Status in einem File ausgegeben werden.

Die nachfolgenden Kapitel gehen tiefer auf diese Punkte ein.

5.4.1 Wait Flags und Wait Level

Die Ausführung des emulierten Programmes kann aufgrund verschiedener Bedingungen unterbrochen werden. Im angehaltenen Zustand kann dem Emulator über die Konsole verschiedene Befehle eingegegeben werden, um Informationen über den internen Status des Systems zu erhalten, und um der Ursache des Fehlers auf den Grund zu gehen. Diese Befehle sind in Tabelle 2 aufgelisted.

Befehl	Mnemonic	Bedeutung
c	continue	Emulator läuft bis zur nächsten Wait Condition.
s	step	Emulator schickt einen Clockimpuls ans IO-System.
d	dump	Interner Status der Hardware ausgeben
q	quit	Beenden des Emulators

Tabelle 2: Befehle, die in der Konsole des Emulator eingegeben werden können.

Im Folgenden werden die **Wait Flags** vorgestellt. Ein Wait Flag ist die Bedingung, unter welcher der Emulator angehalten wird. Die verschiedenen Wait Flags können in einer logischen oder in einer Verknüpfung kombiniert werden. Angegeben ist jeweils der Wert des Flags.

Breakpoint (0x01) Setzen eines Breakpoints. Sobald die Clock den gegebenen Wert erreicht hat, wird der Emulator angehalten, noch bevor der entsprechende Clockimpuls an das IO-System gesendet wird. Breakpoints werden mit dem Kommandozeilen-Argument **-bp n** gesetzt, wobei n für den Clockzyklus steht, in dem der Emulator angehalten werden soll.

Implementation Request (0x02) Der Emulator fragt die Implementation, ob sie den Emulator anhalten möchte. Dies könnte sie verlangen, wenn zum Beispiel undefined behaviour aufgetreten ist. Dazu ruft der Emulator, nach dem Clockimpuls, die shallEmulatorWait Methode des IO-Systems auf (welches shallEmulatorWait der Cpu aufruft). Gibt sie true zurück, wird der Emulator angehalten.

Oft ist es wünschenswert, dass der Emulator unter unterschiedlich gravierenden Bedingungen angehalten wird. Der **Wait Level** bestimmt, wie gravierend eine Bedingung sein muss, damit der Emulator angehalten wird. Der Level reicht von 1 (sehr schwer) bis zu 4 (sehr schwach). Für die Cpu könnte dies folgendermassen umgesetzt werden. Level 1: wenn undefined behaviour aufgetreten ist. Level 2: Bedingungen des Level 1 plus wenn eine Exception aufgetreten ist. Level 3: Bedingungen des Level 2 plus wenn irgendein Signal asserted ist. Der Wait Level wird mit dem Kommandozeilenargument **-wl n** gesetzt, wobei n den Level angibt.

Assertion (0x04) Der Emulator wird angehalten, falls eine Assertion ausgelöst wurde.

Verify (0x08) Der Emulator wird angehalten, wenn der interne Status der Implementation nicht mit aufgezeichnetem Status im Verifyfile übereinstimmt.

Am folgenden Beispiel eines Aufrufs soll dies verdeutlicht werden:

```
./emulator -wf 3 -wl 1 -bp 100
```

Mit -wf 3 werden die Flags 0x01 (Breakpoint) und 0x02 (Implementation Request) ausgewählt. -bp 100 setzt den Breakpoint auf den Clockzyklus 100. -wl 1 setzt den Wait Level auf sehr schwer.
Die isVerifyOk Methode ist Member von CCpu und von CIOSystem. Jede Implementation muss sie folgendermassen implementieren:
bool shallEmulatorWait(ostream& msg); Gibt true zurück, falls der Emulator angehalten werden soll. Die Entscheidung soll unter Berücksichtigung des Wait Levels stattfinden. Unter Wait Level 1 wird nur unter schwerwiegenden unter Wait Level 4 auch unter trivialen Bedingungen angehalten. In den Outputstream msg wird eine Meldung geschrieben, wieso der Emulator angehalten werden soll.

5.4.2 Verify

Die effektivste Art, die Implementation zu debuggen, ist der automatische Vergleich des internen Status nach jedem Clockzyklus mit einem Verifyfile, das den korrekten Status für jeden Zyklus enthält. Somit kann die Korrektheit der Implementation sehr schnell veriziert werden. Mit Hilfe der Wait Flags wird die Ausführung des Emulators sofort unterbrochen, wenn der Status nicht mehr übereinstimmt. In diesem Fall kann der Emulator neu gestartet werden, mit einem gesetzten Breakpoint im fraglichen Clockzyklus. Mit Hilfe eines konventionellen Debuggers kann die Ausführung der fehlerhaften Instruktion Schritt für Schritt verfolgt werden. Durch dieses Verfahren werden fehlerhafte Instruktionen sehr schnell identifiziert, da in wenigen Sekunden hundertausende Instruktionen überprüft werden.
Das grösste Problem an diesem Verfahren dürfte sein, ein Programm zu finden, das ein solches Verifyfile erstellen kann. Im Falle der MipsR3051 Implementation wurde MipsSim dahingehend erweitert, dass ein solches File erstellt wird. Denn MipsSim war schon genügend fehlerfrei um als Referenz dienen zu können. Trotzdem wurden auch in MipsSim noch ein paar kleine Fehler gefunden werden.
Natürlich muss während dem Erstellen des Verifyfiles und dem Verifizieren auf dem Emulator dasselbe Programm ausgeführt werden. Im Falle der MipsR3051 war dies das Topsy OS.
Das Format des Verifyfile ist sehr frei. Der Emulator stellt nur zwei Bedingungen an das Format. Erstens: pro Zeile ist der Status eines Clockzyklus abgespeichert. Zweitens: jede Zeile beginnt mit dem Clockzyklus, für den sie steht. Diese Zahl ist im Textformat abgeschlossen durch einen Leerschlag. Beispiel für ein Verifyfile für einen Prozessor mit zwei Registern:

```
0 0 0
1 5 0
2 5 7
```

Am Ende des Clockzyklus 0 sind beide Register 0. Am Ende des Zyklus 1 hat Register 1 auf 5 gewechselt. Am Ende des Zyklus 2 hat Register 2 auf 7 gewechselt.
Die anfallenden Datenmengen wachsen sehr schnell an. Im Falle des MipsR3051, werden pro Status 195 32 Bit Register abgespeichert (32 General Purpose Register, PC, HI, LO, 32 System Koprocessor Register und 128 TLB Register). Jedes Regiser braucht im Textformat, inklusive Leerschlag, 9 Bytes zur Abspeicherung. Nach nur 100'000 Instruktionen ist das Verifyfile demzufolge auf 175 Megabyte angewachsen. Zum Vergleich: Topsy braucht etwa 350'000 Instruktionen, um ganz Aufzustarten, wobei dies im Vergleich zu anderen OS sehr wenig ist. Aus diesem Grunde muss es möglich sein, dass im Verifyfile nur ein bestimmtes Intervall von Clockzyklen abgespeichert wird. Dies ist die Aufgabe des Programms, das das Verifyfile erstellt. Im Falle von MipsSim wird dies mit den Kommandozeilen-Argumenten **-first n** und **-last m** erreicht, wobei n für den ersten und m für letzten abzuspeichernden Zyklus steht.

Dem Emulator wird mit den Kommandozeilen-Argumenten **-vf o** und **-vl p** angewiesen, dass o der erste und p der letzte Clockzyklus ist, der verifiziert werden soll. Dabei müssen o und p nicht zwingenderweise dem ersten bzw. dem letzten Zyklus entsprechen, der im Verifyfile aufgezeichnet worden ist; sie müssen jedoch im Intervall der abgespeicherten Zyklen liegen. **-vff q** gibt dem Emulator an, welches der erste Zyklus ist, der im Verifyfile steht. [1] **-sc r** Bestimmt den Wert, mit dem die Clock beim Starten des Emulators initialisiert wird. Der folgende Abschnitt führt ein Beispiel an, wieso diese Angabe Sinn macht.
MipsR3051 kann nicht ab Clockzyklus 0 mit dem Verifyfile von MipsSim verifiziert werden. Dies einerseits, weil MipsR3051 auf eine andere Art und Weise intialisiert wird als dies in MipsSim der Fall ist. Andererseits, weil dadurch auch der im BIOS enthaltene Code unterschiedlich ist (MipsSim initialisiert das Bootstackregister im Konstruktor der Cpu, MipsR3051 initialisiert das Bootstackregister während dem Ausführen des BIOS). Zum Zeitpunkt, wo MipsSim und MipsR3051 die erste Instruktion des Topsykernel ausführen, sind unterschiedlich viele Clockzyklen vergangen. Damit während dem Verify jedoch keine Verwirrung aufkommt, müssen 'dieselben' Clockzyklen auf beiden Seiten dieselbe Nummer haben. [2] Deshalb fängt MipsR3051 nicht bei Clockzyklus 0 an, sondern bei -5. MipsR3051 braucht im Vergleich zu MipsSim zusätzlich 3 Zyklen für das resetSignal, das die CPU initialisiert. Und 2 weitere Zyklen, weil das BIOS infolge der Initialisierung des Bootstackregisters um 2 Instruktionen grösser ist. Da, wie erwähnt, das BIOS unterschiedlich ist, kann während der Ausführung des BIOS der MipsR3051 nicht mit MipsSim verglichen werden. Erst ab Zyklus 4, in dem die erste Instruktion des Kernels ausgeführt wird, stimmen beide Seiten wieder überein. Das folgende Beispiel zeigt, wie die ersten 100 000 Clockzyklen von MipsR3051 mit MipsSim verifiziert werden können.

```
java Simulator -first 0 -last 100000
./emulator -sc -5 -vf 4 -vl 100000
```

Das Verifyfile enthält die Zyklen 0 bis und mit 100'000. Der Emulator initialisiert seine Clock mit -5, und verifiziert von Clock 4 bis und mit 100'0000.
Die isVerifyOk-Methode ist Member von CCpu und von CIOSystem. Jede Implementation muss sie folgendermassen implementieren:
bool isVerifyOk(istream& sin_verify, ostream& msg); Vergleicht den internen Status mit dem im Inputstream aufgezeichneten Status. Ist ein Element nicht kohärent, wird das Auslesen des Inputstreams abgebrochen. Eine Fehlermeldung wird in den Outputstream geschrieben, und die Methode wird mit false verlassen. Andernfalls wird nichts in den Outputstream geschrieben und die Methode wird mit true verlassen.

5.4.3 Dump

Hilfreich beim Debuggen der emulierten Software wie auch beim Debuggen der Implementation ist die Aufzeichnung des Status der implementierten Hardware für jeden Clockzyklus. Dazu ruft der Emulator nach jedem Clockimpuls die dump Methode des IO-Systems auf, die dann wiederum dump der Cpu und der Devices aufruft. Dieses Schema setzt sich rekursiv fort, bis die dump Methode jeder Komponente des Systems aufgerufen wurde.
Die Menge an Information, die ausgegeben wird, wird über den **Dump Level** gesteuert, der von 1 bis 4 reicht. Level 1 bedeutet ein Minimum an Information, Level 4 das Maximum. Gesetzt wird der Dump Level mit dem Kommandozeilen-Argument **-dl n**. Der Dump Level ist nur eine Richtlinie. Schlussendlich ist es jeder Komponente überlassen, wieviel Information sie ausgibt und welche ihrer untergeordneten Komponenten sie aufruft. [3]
Die dump Methode ist allen Implementationen von Hardware-Komponenten gemeinsam, und soll folgendermassen implementiert werden:
void dump(ostream& sout); Schreibt den internen Status in den gegebenen Outputstream. Das Format ist frei wählbar. Abhängig vom Dumplevel, der in init gesetzt wurde, soll unterschiedlich viel Information ausgegeben werden. Der Level reicht von 1 minimal) bis 4 (maximal).

[1] Diese Information könnte der Emulator im Prinzip selbst aus dem Verifyfi le auslesen, da jede Zeile mit dem Clockzyklus beginnt. Dies wurde jedoch noch nicht implementiert.

[2] Dies ist der Grund, wieso jede Zeile des Verifyfi le mit dem Clockzyklus beginnen muss: nämlich dass der Emulator überprüfen kann, ob seine Clock mit der Clock des Verifyfi le übereinstimmt.

[3] In der aktuellen Version speichert jede Komponente ihren eigenen individuellen Dumplevel ab Es ist jedoch nicht möglich, jeder Komponente ihren individuellen Level zuzuweisen. Per init Methode wird der durch -dl gesetzte globale Level rekursiv an alle Komponenten verteilt.

Beispiel CCpuMipsR3051: Bei Dumplevel 1 wird nur die Adresse der zuletzt ausgeführten Instruktion plus die Instruktion selbst ausgegeben. Bei Level 4 werden alle verfügbaren Register der Cpu und der Co-Prozessoren und Informationen über Branch-/Loaddelay und über Exceptions ausgegeben.

5.4.4 Konventioneller Debugger

Um die Ursache des Fehlers zu finden, der den Emulator durch die wait Flags zum Anhalten gebracht hat, ist ein konventioneller Debugger wie GDB oder dessen Frontend DDD den rudimentären Befehlen, die dem Emulator über die Konsole gegeben werden können, weit überlegen. Um einen Fehler in der Implementation zu finden, wird folgendes Vorgehen vorgeschlagen:

1. Der Vorahnung entsprechend werden die Wait Flags gesetzt. Wenn möglich wird ein Verifyfile benutzt, und der Vorahnung entsprechend ein Intervall abgesteckt, in dem der Status verifiziert wird.

2. Tritt ein Fehler auf, wird der Emulator wegen den Wait Flags angehalten. Aus der Konsole des Debuggers kann abgelesen werden, in welchem Clockzyklus der Fehler aufgetreten ist. Beispielsweise zeigt die Zeile

   ```
   (mpem clk-1000):->
   ```

   an, dass der Emulator während oder nach dem Ausführen des Clockzyklus 1000 angehalten wurde. Zudem wird meist auch eine Meldung ausgegeben, weshalb der Emulator angehalten worden ist.

3. Mit dem Befehl d kann der Status der Implementation betrachted werden. Wahlweise kann auch das Dumpfile zu Rate gezogen werden, was den Vorteil hat, dass Informationen von vergangenen Zyklen auch verfügbar sind. Zusammen mit der ausgegebenen Meldung hat man oft schon eine Idee, wo der Fehler zu finden ist.

4. Im Sourcecode von Emulator/Emulator.cpp wird folgende Zeile gesucht:

   ```
   if ( mWaitAfterClock )
     console( );
   ```

   Auf die Zeile “console()” wird mit einem konventionellen Debugger ein Breakpoint gesetzt. Diese Zeile wird nur dann ausgeführt, wenn der Emulator angehalten werden soll.

5. Der Emulator wird mit dem Debugger neu gestartet. Im fraglichen Clockzyklus wird mit -bpein Breakpoint gesetzt:

   ```
   emulator -bp 1000
   ```

   Die Anleitung des verwendeten Debuggers gibt Auskunft darüber, wie Kommandozeilen-Argumente an das zu debuggende Programm übergeben werden können.

6. Der Debugger wird nun den Emulator anhalten, sobald der in Schritt 4 gesetzte Breakpoint erreicht worden ist. Dies ist der Fall, sobald der Breakpoint des Emulators erreicht ist, der in Schritt 5 wurde. Mit dem Debugger kann nun Schritt für Schritt der Programmcode durchlaufen werden, was hoffentlich zum Entdecken des Fehlers führt.

6 Instruction

In diesem Kapitel wird der Leser zuerst in die Aufgaben und den Aufbau einer Instruktion eingeführt. Danach wird die Klasse CInstruction vorgestellt, die eine Basisklasse für alle Instruktionen jeglicher Cpu-Implementationen darstellt.

6.1 Einführung

Instruktionen bilden die Bausteine eines Computeprogrammes. Eine **Instruktion** enhält eine Anweisung an die Cpu, welche Funktion sie ausführen soll, und alle dazu benötigten Operanden. Anweisungen können unterschiedenstlicher Art sein. Zum Beispiel arithmetische Anweisungen wie Addieren, Subtrahieren. Oder aber Anweisungen, die den Programmfluss steuern wie bedingte und unbedingte Sprünge. Operanden sind meist bestimmte Register der Cpu; Direktwerte, die mit der Instruktion mitgeliefert werden, oder Speicheroperanden. Die Summe der möglichen Anweisungen bildet den **Instruktionsatz** einer Cpu. Jede Cpu definiert ihren eigenen Instruktionssatz. Des weitern definiert jede Cpu, welche Operanden für eine bestimmte Anweisung zulässig sind.
Ein Instruktion ist im Prinzip ein normales Speicherwort, dessen Bits nach einem bestimmten Muster von der Cpu interpretiert werden. Das Speicherwort wird in diesem Fall auch **Instruktionswort** genannt. Jede Cpu definiert ihr eigenes Muster, wie sie die Bits eines Instruktionswortes interpretiert. Die Grösse einer Instruktion, das heisst wieviele Bytes sie beansprucht, ist von Cpu zu Cpu variabel, und kann sogar innerhalb derselben Cpu varieren. Dies ist zum Beispiel bei der Ia32 Architektur der Fall. Innerhalb einer Risch Cpu wie dem MipsR3051 haben jedoch alle Instruktionen dieselbe Grösse. Sie entspricht der Grösse der Register im Falle des MipsR3051 32 Bit. Figur 3 zeigt als Beispiel den Aufbau von zwei MipsR3051 Instruktionstypen.

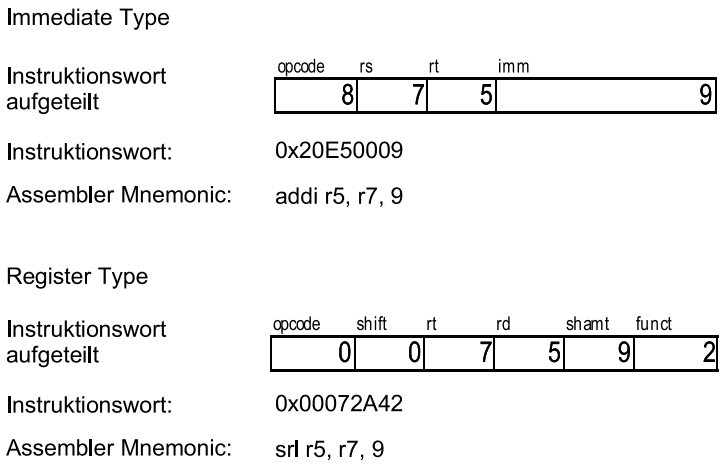


Abbildung 3: Aufbau einer MipsR3051 Instruktion anhand von zwei Beispielen.

Dem Beispiel sollen zwei Grundideen entnommen werden. Erstens: Bestimmte Bitfelder wie rs, rt, immidiate kommen in verschiedenen Instruktionen vor und haben immer die gleiche Bedeutung. Der Inhalt des rt Feldes selektioniert immer ein bestimmtes Register. Das nächste Kapitel zeigt, wie dies ausgenutzt werden kann. Zweitens: Es gibt zwei Hauptgruppen von Bitfeldern: die eine wählt die auszuführende Anweisung aus. Dazu gehören die Felder op und funct. Die andere Gruppe wählt die Operanden der Anweisung aus. Dazu gehören die Felder rs, rt, immidiate. Die erstgenannte Gruppe interessiert uns im Folgenden.
Im folgenden wird Figur 4 eräutert. Das Feld op wählt die Anweisung aus. Da es jedoch nur 6 Bit breit ist, würde dies bedeuten, dass der MipsR3051 nur 64 verschiedene Anweisungen verstehen würde. Dies genügt nicht. Daher sind bestimmte Werte von opcode reserviert, um anzudeuten, dass ein weiteres Feld im Instruktionswort die Anweisung auswählt. Ist der Inhalt von op 0, dann wählt das Feld func die Anweisung aus. Der gesamte Instruktionssatz hat demzufolge eine Struktur von verlinkten Listen.

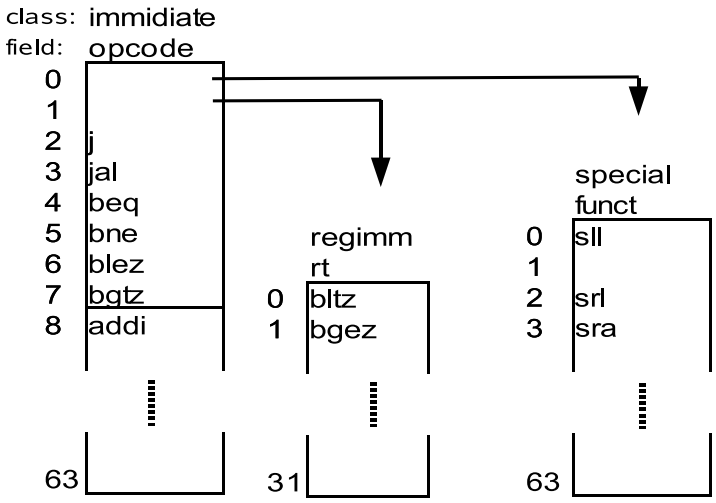


Abbildung 4: Der Instruktionsatz kann mit verlinkten Listen dargestellt werden.

Die Liste, in der eine Anweisung nun tatsächlich steht, bestimmt die **Klasse** der Instruktion. Der Index innerhalb der Liste, an dem die Anweisung steht, bezeichnet den **Index** der Instruktion. Die Instruktion srl (shift right logical) hat also die Klasse register und den Index 2. Der Instruktionsatz kann folglich auch als zwei- dimensionales Array angesehen werden. Wobei die zwei Dimensionen von Klasse und Index gebildet werden. Dies wird durch Figur 5 illustriert. Da nicht alle Listen, wie sie in Figur 4 vorkommen, gleich lang sind, sind einige Felder des Arrays unbenutzt.

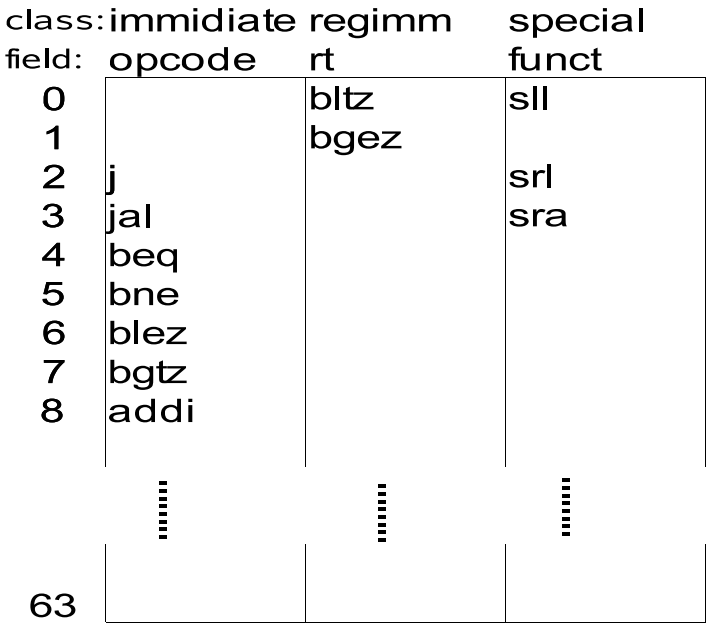


Abbildung 5: Der Instruktionssatz kann als zweidimensionales Array dargestellt werden.

Aus Figure 3 ist ersichtlich, dass es verschiedene Arten gibt, eine bestimmte Instruktion zu repräsentieren. Die Repräsentation durch eine Zahl wie 0x20E5009, wird Instruktionswort ge-

nannt. Die Repräsentation durch Text wie add “r5, r7, 9” wird **assembler mnemonic** genannt. Das Umwandeln von assembler mnemonics in Instruktionsworte wird **assemblieren** genannt. Der umgekehrte Weg, das Umwandeln von Instruktionsworte in assembler mnemonics heisst **disassemblieren**. Es sei angemerkt, dass das in Figure 3 verwendete Format nicht dem offiziellen Format von MipsR3051 assembler mnemonics entspricht, sondern einem vereinfachten Format. Im Folgenden wird das vereinfachte Format benutzt.

6.2 Class CInstruction

```
Source:                Cpu/Instruction.h, .cpp
Implementionsbeispiel: Arch/CpuMipsR3051/InstructionMipsR3051.h, .cpp
```

Die Vorteile einer Klasse, die assemblieren und disassemblieren kann, liegen auf der Hand. Das Disassemblieren kann benützt werden, um die Dump Informationen, die eine Cpuarchitektur ausgibt, verständlicher zu machen. Denn assembler mnemonics sind für den Menschen natürlich sehr viel einfacher zu verstehen als Instruktionsworte. Das Assemblieren kann benutzt werden, um einzelne Instruktionen auf einer neuen Architektur zu testen. Es macht keinen Sinn, auf einer frühen Version einer neuen Implementation gleich ein ganzes Programm ausführen zu wollen. Es würden zuviele Fehler auftreten.
CInstruction ist eine abstrakte Basisklasse, von der alle implementierenden Instruktionsklassen abgeleitet werden müssen. Die Hauptaufgabe der Basisklasse liegt im Bereitstellen von Methoden, die einfaches Assemblieren und Disassemblieren erlauben. Darauf wird in Kürze eingegangen.
Die Hauptaufgaben einer implementierenden Instruktionsklasse liegen im Bereitstellen des Zugriffs auf die einzelnen Bitfelder und im Bestimmen der Instruktionsklasse und des Indexes. Es ist nicht Aufgabe dieser Klasse, die in der Instruktion enthaltenen Anweisung auszuführen. Dies ist Aufgabe der CPU. Denn nur diese hat Zugriff auf die benötigten Resourcen wie Register, ALU oder IO-System. Die Instruktion teilt der CPU mit, was sie zu tun hat, und die CPU führt die Anweisung aus.
Jede CPU und jeder Co-Prozessor hat seine eigene Instruktionsklasse zugeordnet. Zum Beispiel hat CCpuMipsR3051 die Klasse CInstructionMipsR3051, CCopMipsR3051Cp0, CInstructionMipsR3051Cp0 zugeordnet. Alle Instruktionsklassen sind von der Basisklasse CInstruction abgeleitet. Wieso Co-prozessoren ihre eigene Instruktionsklasse haben, wird in Kapitel 6.3 erläutert.
Im Folgenden wird erklärt, wie die Basisklasse beliebige Instruktionssätze assemblieren/disassemblieren kann. Im vorhergehenden Kapitel wurde gezeigt, dass für jede Instruktion Bitfelder definiert sind, die Informationen über die Operanden der Instruktion enthalten. Die assembler mnemonics der jeweiligen Instruktion enthalten dieselben Operanden, einfach in einer anderen Repräsentation. Kennt man also die Operanden einer Instruktion, kann sowohl das Instruktionswort wie auch das assembler mnemonic gebildet werden. Wie Figur 3 zeigt, hat die zum Beispiel addi Instruktion die Operanden rs, rt und immidiate.
Des weitern wurde gezeigt, dass sich alle Instruktionen des gesamten Instruktionssatzes in einem zweidimensionalen Array speichern lassen. Die benötigten Informationen zum Assemblieren/Disassemblieren werden daher in einem zweidimensionalen Array abgelegt. Jedes Element speichert die benötigten Informationen für eine Instruktion. Die Position eines Elementes im Array bestimmt die Klasse und den Index der Instruktion. Der Inhalt des Elementes bestimmt die benötigten Operanden und den Namen der Instruktion, wie er in der mnemonic Repräsentation verwendet wird. Die folgende Struktur definiert den Typ eines solchen Elementes:

```
struct CAssInfo
{ const char*  mName;
  int          mOperand[];
};
```

Operanden werden durch die implementierende Klasse definiert. Folgendes Beispiel ist eine Auswahl der Operanden, die CInstructionMipsR3051 definiert.

```
enum  EOperand
{ NONE = 0,
```

```
  OPCODE, FUNCT,
  RT, RS,
  SIMM  // signed immidiate
};
```

Durch folgendes Beispiel sind nun alle benötigten Informationen vorhanden, um die addi instruktion zu assemblieren oder zu disassemblieren:

```
CLASS_IMMIDATE = 0;
INDEX_ADDI     = 8;
CAssInfo table[][];
table[ CLASS_IMMIDATE ][ INDEX_ADDI ] = { "addi" , { RT, RS, SIMM } };
```

Die Postion des Elementes bestimmt die Klasse und den Index der Instruktion, was denselben Informationsgehalt hat wie der String “addi”. Das Element selbst bestimmt die benötigten Operanden. Der Wert der Operanden, wird entweder durch das Instruktionswort geliefert im Falle des Disassemblierens, oder durch das assembler mnemonic im Falle des Assemblierens. “addi r5, r7, 9” zum Beispiel bestimmt, dass Operand rt den Wert 5 hat, Operand rs den Wert 7 und Operand signed immidiate hat den Wert 9. Diese Werte werden beim Assemblieren ins entsprechende Bitfeld des Instruktionswortes eingetragen.
Die implementierende Klasse erstellt somit das zweidimensionale Array, und übergibt es auf Anfrage (Methode typeInfo()) der Basisklasse. Konkret ist dies die Struktur **CInstruction::CTypeInfo**, die etwas komplexer ist als das vorgestellte zweidimensionale Array. Für die spezifischen Unterschiede sei der Leser auf den Code verwiesen.

6.3 Instruktionen der Co-Prozessoren

`Implementionsbeispiel: Arch/CpuMipsR3051/InstructionMipsR3051Cp0.h, .cpp`

Bevor darauf eingegangen wird, wie Instruktionen assembliert bwz disassembliert werden, behandeln wir die Instruktionen der Co-Prozessoren.
Der Instruktionssatz der Co-Prozessoren ist eine Erweiterung des Instruktionssatzes der Cpu. Es ist daher naheliegend, dass die Klasse, die die Instruktionen eines Co-Prozessors repräsentiert, von der Instruktionsklasse der Cpu abgeleitet ist.Beispiel: CInstructionMipsR3051Cp0 ist von CInstructionMipsR3051 abgeleitet.
Die Instruktionsklasse der Cpu kann bzw. sollte die Instruktionen der Co-Prozessoren aus dem folgendem Grund nicht selbst aufnehmen: Gewisse Co-Prozessoren sind erstens optional und zweitens nicht genau spezifiziert. Zum Beispiel ist gemäss Mips32 (nicht MipsR3051) Spezifikation [cite] der Co-Prozessor 2, d.h. seine Aufgaben und folglich auch sein Instruktionssatz, nicht genau spezifiziert. Es wird lediglich vorgegeben, wo sich dieser Instruktionsatzt in den Instruktionssatz des Mips32 einklinken kann. Figur 4 verdeutlicht, was mit ’Einklinken’ gemeint ist. Hätten die Co-Prozessoren, in diesem Fall der Co-Prozessor 2, keine eigene Klasse, müsste, jedesmal wenn ein anderer Co-Prozessor 2 benutzt wird, die Instruktionsklasse der Cpu abgeändert werden.
Die Instruktionsklasse der Cpu kennt also die Instruktionen der Co-Prozessoren nicht. Dafür ’kennt’ sie aber die Instruktionsklassen ihrer Co-Prozessoren. Diese werden ihr beim Aufstarten des Programms mit der **addDerivedType** Methode [4] mitgeteilt. C++ kennt keinen Mechanismus, mit dem eine beliebige Klasse identifiziert werden kann, wie dies in Java beispielsweise mit der Klasse Class und der Methode Object.getClass möglich ist. Deshalb steht ’kennt’ in Anführungszeichen. Der addDerivedType Methode wird eine Instanz der Instruktionsklasse des Co-Prozessors übergeben. Erst die folgenden Ausführungen werden verdeutlichen, inwiefern diese Instanz von Nutzen ist.
Dieses Prinzip kann rekursiv fortgesetzt werden. Auch von den Instruktionsklassen der Co-Prozessoren können weitere Instruktionsklassen abgeleitet werden. Wiederum werden der Instruktionsklassse des Prozessors durch addDeriveType alle direkt abgeleiteten Instruktionsklassen mitgeteilt.

[4] addDerivedType ist klassenbezogen und deshalb eine statistische Methode. Sie muss von allen Instruktionsklassen, von denen abgeleitet wird, implementiert werden. Es würde Sinn machen, sie in CInstruction als true virtual zu deklarieren. Methoden können jedoch nicht gleichzeitig static und virtual sein. Jede implementierende Klasse muss selbst daran denken, addDerivedType zu implementieren.

6.4 Assemblieren

```
Source:    Cpu/Instruction.h, .cpp
Methode:   int CInstruction::assemble( istream& sin )
```

Im Folgenden wird der Algorithmus vorgestellt, mit dem assembler mnemonics assembliert werden können. Implementiert wird er von der Methode Assemble, implementiert in CInstruction. Ihr wird ein assembler mnemonic übergeben, das assembliert wird und in einer Membervariable der Klasse als Instruktionswort abgespeichert wird. Der Leser soll sich klar sein, dass die Methode assemble via einer Instanz einer implementierenden Instruktionsklasse aufgerufen wird. [5]

```
CInstructionMipsR3051 instruction;
instruction.assemble( "addi r5, r7, 9" );
instruction.get() // returns 0x20E50009
```

Die Methode assemble lässt sich von typeInfo, implementiert in der implementierenden Klasse (im Beispiel CInstructionMipsR3051), den Pointer auf die bereits vorgestellte CTypeInfo Struktur übergeben:

```
CTypeInfo* typeinfo = typeInfo();
```

Assemble verfügt nun über alle Informationen, um den Inputstream zu assemblieren. Der der Methode Assemble als Parameter übergebene Inputstream enthält das assembler mnemonic. Daraus wird der Name extrahiert, zum Beispiel “addi”. Dieser wird in dem in typeinfo enthaltenen zweidimensionalen Array gesucht. [6] Ist er gefunden, ist durch die Position des gefundenen Arrayelementes die Klasse und der Index der Instruktion bekannt. Und die entsprechenden Bitfelder im Instrukionswort können gesetzt werden. Dies geschieht mittels der Methode setClassAndIndex, die in der implementierenden Klasse implementiert ist.
Danach wird für jeden Operanden, der im Arrayelement (Typ CAssInfo) vermerkt ist, die Methode assembleOperand aufgerufen, die wiederum in der implementierenden Klasse implementiert ist. Sie liest selbständig den nächsten Operanden aus dem Inputstream, dessen Typ sie ja nun kennt. Entsprechend den eingelesenen Daten und dem Wissen über den Typ des Operanden setzt sie das entsprechende Bitfeld ins Instruktionswort.
Wurde der Name der Instruktion nicht in der CTypeInfo Struktur gefunden, werden alle direkt abgeleiteten Klassen angefragt, ob sie diesen Namen kennen und folglich assemblieren können. Dank der vorgestellten Methode addDerivedType enthält die CTypeInfo Struktur, neben dem zentralen zweidimensionalem Array, je eine Instanz aller direkt abgeleiteten Klassen. Durch diese Instanz kann die Methode Assemble der abgeleiteten Klasse aufgerufen werden. Dieses Prinzip wird rekursiv fortgesetzt. Kennt keine Klasse im Vererbungsgraphen die Instruktion, wird ein Fehler zurückgemeldet.

6.5 Disassemblieren

```
Source:  Cpu/Instruction.h, .cpp
Methode: int CInstruction::disassemble(ostream& sout, bool newline )
```

Die Methode Disassemble assembliert das interne Instruktionswort und schreibt das resultierende assembler mnemonic in den als Parameter übergebenen Outputstream. Die geschriebene Zeile wird mit einem Newline-Zeichen abgeschlossen, falls Newline true ist.
Der Disassembler Algorithmus ist im Wesen dem Assemblier-Algorithmus sehr ähnlich. Daher wird an dieser Stelle nur auf die wesentlichen Unterschiede eingegangen.
Mit den Methoden iclass und index, beide in der implementierenden Klasse implementiert, wird die Klasse und der Index der Instruktion abgefragt. Diese Information kann direkt verwendet werden, um im zweidimensionalen Array das CAssInfo Element zu finden.

[5] Die Methode assemble ist mehrfach überladen, damit mnemonics bequem in verschiedene Formate übergeben werden können.Deshalb kann an Stelle eines Inputstreams auch ein String übergeben werden. Dieser wird automatisch in einen Inputstream umgewandelt.

[6] Der Suchalgorithmus ist der einfache, aber langsame “linear search” Algorithmus. In einer verbesserten Version wäre es angebracht, eine Hashtable zu benutzten, die ideal für diese Problemstellung wäre.

Bezeichnen die Klasse und der Index jedoch eine Instruktion, die Teil des Instruktionssatzes eines Co-Prozessors ist, muss die Instruktionsklasse des Co-Prozessors die Instruktion disassemblieren.
Im Element ist der mnemonic name der Instruktion enthalten, der in den Outputstream geschrieben werden kann. Für jeden im Element vermerkten Operanden wird assembleOperand aufgerufen, der das entsprechende Bitfeld disassembliert und als mnemonic in den Outputstream schreibt.

6.6 Einschränkungen

Im Folgenden werden einige Einschränkungen der Assemble / Disassemble Methoden erläutert.
Die Methode Assemble kann assembler mnemonics nur zeilenweise assemblieren. Demzufolge können keine Labels gesetzt werden, die von Jump-Instruktionen angesprungen werden können. Es können auch keine Bezeichner für Variablen verwendet werden. Direktwerte (ein Label ist schlussendlich ein Direktwert) müssen immer als Zahl angegeben werden. Folgende mnemonics können nicht assembliert werden:

```
       j label
label: addi r5, r7, 9
```

Die Einzelnen Operanden müssen mit einem Komma abgetrennt werden. Folgendes mnemonic kann demzufolge nicht assembliert werden:

```
lw  r1, 10(r2)  // error
lw  r1, r2, 10  // workaround
```

Diese Einschränkung könnte behoben werden, indem zum Beispiel in CInstruction die rein virtuelle Methode skipOperandSeprator(istream& sin, int left_operand, int right_operand) eingeführt wird, die von der implementierenden Klasse implementiert werden muss.
Es sei angemerkt, dass die mnemonics, wie sie in CInstructionMipsR3051 verwendet werden, nur aus Zeitnot vereinfacht wurden[7]. Würde die Methode CInstructionMipsR3051::assembleOperand entsprechend erweitert, könnte auch die komplexere Form der mnemonics verwendet werden.

[7] Anstatt der normalerweise verwendeten “addi $a0,$a1,100” nur “addi r4,r5,100”

7 Cpu

In diesem Kapitel wird die abstrakte Basisklasse CCpu vorgestellt, von der alle Cpu-Implementationen abgeleited sein müssen.

7.1 Class CCpu

```
Source:                 Cpu/Cpu.h, .cpp
Implementionsbeispiel: Arch/CpuMipsR3051/CpuMipsR3051.h, .cpp
```

Die Hauptaufgabe der Cpu liegt darin, Instruktionen via IO-System aus dem Ram zu lesen und auszuführen. Auf das IO-System wird mittels der global definierten Instanz “I0-System” zugegriffen.
Eine neue Cpu Implementation muss alle Methoden von CCpu überschreiben. All diese Methoden wurden bereits vorgestellt; Tabelle 3 gibt Auskunft darüber, in welchem Kapitel welche Methode erklärt wird.

isVerifyOk **dump** **shallEmulatorWait**	Kapitel 5.4
powerOn **init** **setSignal**	Kapitel 5.2.1
setSignal **getSignal**	Kapitel 4.1
clock	Kapitel 4.5

Tabelle 3: Bereits eingeführte Methoden von CCpu

Obwohl die Methoden von CCpu formal nicht als rein virtuell deklariert worden sind, müssen sie trotzdem alle überschrieben werden. Wird eine dieser Methoden nicht überschrieben, wird folglich deren Implementation in CCpu aufgerufen, welche das Programm durch ASSERT(0) abbricht.

7.2 Class CCpuEnhanced

```
Source:                 Cpu/CpuEnhanced.h, .hxx
Implementionsbeispiel: Arch/CpuMipsR3051/CpuMipsR3051.h
```

Mit der Klasse CCpuEnhanced werden dem Programmierer, der eine neue Cpu Implementation schreiben will, einige elementare Methoden zur Verfügung gestellt, die er höchstwahrscheinlich brauchen wird. Zum einen ist dies eine template Methode, die eine allgemeine Instruktion modeliert, zum andern Methoden, die auf das IO-System zugreifen und Probleme wie Endianess und Alignement behandeln.
Zunächst soll auf den etwas speziellen Aufbau der Klasse CCpuEnhancded eingegangen werden. CCpuEnhanced ist eine Template-Klasse. Als erstes Templateargument muss die Klase bezeichnet werden, die schlussendliche die Cpu-Klasse darstellt, zum Beispiel CCpu-MipsR3051. Das zweite Template-Argument ist die zur Cpu gehörende Instruktionsklasse. Im File Arch/CpuMipsR3051/CpuMipsR3051.h befindet sich ein Beispiel, wie CCpuEnhanced korrekt instanziert wird.

```
template< class TCCpu, class TCInstruction >
class CCpuEnhanced : public CCpu { /*...*/ };
```

Es wäre durchaus auch möglich, CCpuEnhanced als normale Klasse zu definieren. Dann müssten jedoch alle ihre Methoden als Templatemethoden definiert werden, die alle TCCpu als Template-Argument hätten, manche noch zusätzlich TCInstruction. TCCpu und TCInstruction

sind logisch gesehen eher Parameter der Methoden von TCCpuEnhanced, als von der Klasse selbst. Ich denke, diese Lösung wäre noch komplexer bzw. umständlicher gewesen.
TCCpu wird aus folgenden Grund benötigt: Gäbe es das Template-Argument TCCpu nicht, und CCpuMipsR3051 wäre direkt von CCpuEnhanced abgeleitet, könnte die CCpuEnhanced- Methoden die Methoden von CCpuMipsR3051 nur via Virtual-Mechanismus aufrufen. Dies würde zuviel Zeit kosten, erst recht in der Methode CCpuEnhanced:instructionGeneral, die viele Methoden von TCCpu aufruft. Dadurch, dass durch TCCpu nun CCpuEnhanced den Typ der schlussendlichen Klasse kennt, in diesem Beispiel CCpuMipsR3051, kann CCpuEnhanced die Methoden von TCCpu bzw CCpuMipsR3051 inline aufrufen. Im Code von CCpuEnhanced befindet sich oft die Zeile

```
TCCpu*  me = (TCCpu*)this;
```

Mit der Variable me können alle Methoden von TCCpu bzw. CCPuMipsR3051 inline aufgerufen werden.

7.2.1 Templatemethode für Instruktionen

Die meisten Instruktionen, die eine Cpu ausführt, haben einen ähnlichen Aufbau. Dies gilt insbesondere für die arithmetischen und logischen Instruktionen, wie Addieren, Shift oder Bitweise und Verknüpfen.

1. Operanden laden
2. Operation ausführen
3. Resultat abspeichern

Bei vielen Architekturen sind die Anzahl möglicher Operatoren und Resultat-Destinationen beschränkt. Beim MipsR3051 sind mögliche Operatoren zum Beispiel rs-,rd- oder rt-Register, oder aber ein Direktwert. Diese Operatoren finden sich in vielen Instruktionen wieder. Andererseits finden sich auch viele Operationen in verschiedenen Prozessoren wieder. Wenn im folgenden von Operationen wie Addieren die Rede ist, dann ist das Setzen der Flags immer eingeschlossen. Zum Beispiel haben praktisch alle Prozessoren eine Addier- und eine Shift-Operation.
Im Sinne des Prinzips, dass Code wiederverwendet werden kann, zeichnen wir folgenden Ansatz einer allgemeinen Instruktion auf, in Pseudocode geschrieben:

```
class CCpuEnhanced : public CCpu
{
  template< OP1, OP2, DST, FUNC >
  instructionGeneral( instructionword )
  { op1 = getOperand( instructionword, OP1 );
    op2 = getOperand( instructionword, OP2 );
    result = FUNC( op1, op2 );
    saveResult( instructionword, DST, result );
  }
}

class CCpuMipsR3051 : public CpuEnhanced
{
  getOperand( instructionword, OP )
  { switch ( OP )
    { case OP_RS: return mRegister[ instructionword.rs() ];
      case OP_RD: return mRegister[ instructionword.rd() ];
    }
  }
  saveResult( instructionword, DST, result )
  { switch ( op )
    { case DEST_RD: mRegister[ instructionword.rd() ] = result;
      case DEST_RT: mRegister[ instructionword.rt() ] = result;
```

```
    }
  }
}

class CAlu
{
  add( op1, op2 )
  { result = op1 + op2;
    /* ... setzen der flags ... */
    return result;
  }
}
```

InstructionGeneral ist in der Basisklasse CCpuEnhanced definiert, und kann deshalb in allen von CCpuEnhanced abgeleiteten Klassen verwendet werden. In der schlussendlichen Cpu Klasse, in diesem Fall CCpuMipsR3051, werden getOperand und saveResult definiert. Sie können von allen MipsR3051-Instruktionen wiederverwendet werden. getOperand und saveResult sind prozessorspezifisch.Es macht daher keinen Sinn, sie in CCpuEnhanced zu definieren.
Sei i ein eingelesenes Instruktionswort, dann hat instructionGeneral<OP_RD, OP_RS, DEST_RT, CAlu::add>(i) die Funktionalität der MipsR3051 “add” -Instruktion. Die add-Instruktion addiert das rs- und das rt-Register, und speichert das Resultat im rt-Register. Somit können die meisten Instruktionen einer Cpu im Baukastensystem zusammengesetzt werden, indem für jede Instruktion die entsprechenden Template-Agumente verwendet werden.
Für ein Beispiel verweise ich den Leser auf den mitgelieferten Code. Eine eingehendere Betrachtung würde den Rahmen dieser Dokumentation sprengen. In Cpu/CCpuEnhaced.hxx befinden sich die Methode InstructionGeneral, in Arch/CpuMipsR3051/CpuMipsR3051.h befinden sich getOperand und saveResult und schliesslich in Cpu/Alu.h die add-Methode. Die Template-Argumente von instructionGeneral werden beim Erstellen der Instruktionstabelle in Arch/CpuMipsR3051/CpuMipsR3051.cpp in der Methode InitInstructionTable definiert. Tatsächlich aufgerufen wird die Instruktion in der Clockmethode, welche in Arch/CpuMipsR3051/CpuMipsR3051.cpp zu finden ist. Die CAlu Klasse wird im nächsten Kapitel noch genauer beschrieben.
Der tatsächliche Code ist in einigen Punkten komplexer als der vorgestellte Pseudocode. Auf wichtige Unterschiede geht die folgende Liste ein. Um sie zu verstehen, müssen die entsprechenden Textstellen konsultiert werden.

TTReg InstructionGeneral, getOperand, saveResult sind Templatemethoden, die unter anderem den Template-Parameter TTReg besitzen. TTReg ist der Datentyp (TRegu32, int, short etc) des Operanden bzw. des Resultates. Für den Mips, der nur 32-Bit-Register hat, wird deshalb für TTReg immer TRegu32 eingesetzt. Für einen x86, der 8Bit, 16Bit und 32-Bit-Register hat, würde je nach Instruktion TRegu8, TRegu16 bzw. TRegu32 eingesetzt.

helpCompiler Der Sinn von helpCompiler ist, den Code effizienter zu gestalten, indem dem Compiler implizites Wissen mitgeteilt wird. In CMipsR3051::helpCompiler wird zum Beispiel das “hat-eine-Exception”-Flag auf false gesetzt. Dies ist erlaubt, da die Instruktion gar nicht ausgeführt würde, wenn dem nicht so wäre. Dieses Flag wird in den häufig auftretenden hasException()-Methoden abgefragt. Alle Methoden sind inline. Deshalb werden alle Aufrufe von hasException, und die dazugehörigen if-Blöcke wegoptimiert, wenn der Compiler sieht, dass die getOperand-, saveResult- und die FUNC-Methode das “hat-eine-Exception”-Flag nicht verändern, da er dank helpCompiler eben weiss, dass es false ist.

Es sei angemerkt, dass für jede einzelne Instruktion, ausgewählt werden kann, wie sie implementiert werden soll; es ist nicht zwingend, dass generalInstruction verwendet wird. Einige spezielle Instrukionen, die gar nicht in dieses generelle Schema passen, werden am besten in eigenständigen Methoden implementiert. Als Beispiel sei hier CCpuMipsR3051::instructionLwr oder CCpuMipsR3051Cp0::instructionTlbp genannt. Des weitern können auch neue Templates geschrieben werden. Obwohl generalInstruction meines Erachtens schon sehr allgemein und sehr flexibel ist, zieht es der Programmierer einer neuen Cpu Architektur eventuell vor, eine eigene “generelle Instruktion” zu implementieren. Zum Beispiel könnte für den ARM-Prozessor

eine Templatemethode CCpuArm::instructionGeneralArm implementiert werden, die als Basis für die meisten ARM- Instruktionen dienen würde.
Da nun alle vorgestellten Methoden inline definiert sind, bis auf instructionGeneral, ist auch die Performance gewährleistet. Tests haben gezeigt, dass die gesamte instructionGeneral-Methode vom Compiler in etwa 18 Assembler-Instruktionen umgesetzt wird, ohne jegliche call-Instruktionen zu generieren. Siehe auch Kapitel 11.
Natürlich ändert sich die Grösse der instructionGeneral-Methode je nach Template-Argumenten. .

7.2.2 Zugriff auf das IoSystem

Das Eigentliche Senden/Empfangen von Daten zum/vom IO-System ist oft nur ein Teil des gesamten Zugriffes auf das IO-System. Es stellt sich noch das Problem der Adresstranslation, der Endianness und des Alignment. Die Methoden readMemx/writeMemx, wobei für x 8,16,32 stehen kann, behandeln diese Themen.

7.3 Class CAlu

```
Source:                Cpu/Alu.h
Implementionsbeispiel: Arch/CpuMipsR3051/AluMipsR3051.h
```

In Kapitel 7.2.1 wurde schon angedeutet, dass der Kern einer Instruktion, die eigentliche arithmetische Operation wie Add, Shift etc. und das dazugehörige Setzen der Flags erstens bei verschiedenen Prozessoren immer wieder vorkommt und zweitens auch innerhalb des Instruktionssatzes ein- und derselben Cpu öfters vorkommen kann. Beim MipsR3051 Instruktionssatz zum Beispiel kommen die meisten arithmetischen/logischen Operationen in einer Variante vor, in der beide Operanden Register sind, und in einer zweiten Variante, wo einer der Operanden ein Direktwert ist. Noch krasser ist das Beispiel des x86 Instruktionssatzes, wo es für die meisten arithmetischen/logischen Instruktionen bis zu 14 Varianten gibt.
Würde die Operation nur aus dem Addieren bestehen, lohnte es sich kaum, dafür extra eine Methode zu schreiben, da die Addition schon vom C++ Operator + übernommen wird. Dies gilt auch für die anderen arithmetischen/logischen Operationen. Nach jeder Operation müssen auch die Flags berechnet werden; somit hat eine Add-Methode durchaus ihre Berechtigung.
Die Flags Carry, Zero, Sign (auch Negative genannt) und Overflow kommen in vielen Prozessoren vor. Sie haben überall die gleiche Bedeutung, und werden deshalb meist nach denselben Regeln berrechnet. Aber nach Operationen, wo der Inhalt eines Flags gemäss seiner Definition undefiniert ist, setzen unterschiedliche Prozessoren das Flag auf einen anderen Wert. Zum Beispiel hat das Overflowflag nach der logischen and Operationen keinen eindeutig definierbaren Wert. Bei einem x86 Prozessor wird es auf 0 gesetzt, bei einem ARM bleibt es unverändert. Das Setzen der Flags muss also noch von Parametern abhängen.
Nach diesen Überlegungen wird nun eine

```
class CAlu
{
  add( op1, op2 , flag_treat )
  { result = op1 + op2;
    setZeroFlag( result == 0 , flag_treat );
    setSignFlag( result < 0  , flag_treat );
    /*...uebrige flags setzen...*/
  }
}

class CCpuX86
{
   setZeroFlag( value , flag_treat )
   { switch( CFlagTreat::z(flag_treat) )
     { case ALWAYS_0  : mFlagZero = 0;
       case ALWAYS_1  : mFlagZero = 1;
       case ALU_RESULT: mFlagZero = value;
```

```
        case ALU_INVERS: mFlagZero = !value;
      }
    }
}
```

In CAlu werden die Flags nun allgemein berechnet. In setZeroFlag, setSignFlag etc. kann Dank dem flag_treat Parameter aber entschieden werden, wie das Flag nun tatsächlich gesetzt werden soll. Da setZeroFlag, setSignFlag etc. in der Cpu-Klasse definiert sind, kann jede Cpu-Implementation selber entscheiden, auf welche Arten der “Vorschlag” der ALU überschrieben oder übernommen werden kann.
Damit die Methoden von CAlu nicht mit zu vielen Parametern überladen sind, gibt es nur einen einzigen flag_treat-Parameter für alle Flags, statt jeweils einen Parameter pro Flag. Deshalb muss der eine Parameter nun in Bitfelder aufgespalten werden; jedes Flag hat sein Bitfeld. Wie diese Aufteilung von statten geht, ist jeder Cpu selbst überlassen. Wahlweise kann dazu die Hilfsklasse CFlagTreat verwendet werden. Zu beachten ist, dass sich die CAlu Methoden überhaupt nicht um den flag_treat-Parameter kümmern. Sie reichen ihn nur weiter. Es ist die Cpu selbst, die die CAlu-Methoden aufruft. Dies ist letztlich der Grund, warum die Cpu vollkomen frei ist, wie sie den flag_treat- Parameter behandelt.
CAlu ist eine Template-Klasse mit dem Template-Parametern TCCpu, TTReg, TTRegSign und BIT_CNT. TCCpu wird wie bei CCpuEnhanced dazu verwendet, Methoden von TCCpu inline aufrufen zu können. Die restlichen drei Parameter sind logisch gesehen redundant. Alle drei bezeichnen die Registergrösse, mit der die Methoden der Alu rechnen sollen. Soll die Alu mit 32-Bit-Registern rechnen, wird für TTReg TRegu32 eingesetzt. Folglich ist TTSignReg TRegs32 und BIT_CNT ist 32. Für TTReg wird häufig TRegu32, TRegu16 etc. verwendet. Aus Sicht des Compilers lassen sich aber TRegu32 und TRegu16 nicht immer unterscheiden. Siehe Kapitel 4.3. Deshalb braucht es den Templateparamter BIT_CNT, um auch für den eineindeutig zu spezifizieren, wie gross die zu verwendenden Register sind. Dass die Anzahl der Bits eines Registers durch BIT_CNT so direkt gegeben ist, hat sich dann nebenbei häufig als sehr komfortabel erwiesen. Der Leser sei hierbei auf die Methoden CAlu::shiftR oder CAlu::rotateR verwiesen.
Bevor die CAlu Template-klasse bequem benutzt werden kann, muss sie instanziert werden, d.h. alle Template-Parameter müssen angegeben werden. Im Beispielcode ist dies die CAluMipsR3051Base-Klasse. Es wird vorgeschlagen, davon die effektiv verwendete Klasse abzuleiten (Beispiel CAluMipsR3051). In der abgeleiteten Klasse könnten nun spezielle Methoden, die nicht in CAlu vorhanden sind, hinzugefügt werden. Des weiteren können auch alle oder einige CAlu Methoden überschrieben werden, um zusätzliche Flags zu setzen. Eine x86 -Implementation müsste zusätzlich zu den Standardflags der CAlu, also Sign, Overflow, Zero und Carry, auch noch das Parityflag und das Auxiliarz Carryflag setzen. Die Klasse CCpuX86 (nicht im beigelegtem Code enthalten) hätte dann zusätzlich die Methoden setParityFlag und setAuxCarryFlag.
Beispiele zu all diesen Ideen findet der Leser in CAluMipsR3051, wo das CAlu instanziert wird und wovon die CAluMipsR3051-Klasse abgeleitet wird. In CCpuMipsR3051::initInstructionTable werden via CCpuEnhanced::instructionGeneral die CAlu-Methoden aufgerufen. CCpuMipsR3051::setOverflowFlag ist ein Beispiel für die das s Setzen eines Flags.
Alle beteiligten Methoden sind inline, und der Parameter flag_treat wird als Konstante übergeben. Der Compiler optimiert alle nicht benötigten Case Labels in den Methoden setZeroFlag etc. weg. Auch die Berechnung des Flags wird wegoptimiert, falls das Resultat in setZeroFlag etc. nicht gebraucht wird.

8 IoSystem

8.1 Class CIoSystem

```
Source:                IoSystem/CIoSystem.h,.cpp
Implementionsbeispiel: Arch/IoSystemMips/IoSystemMips.h,.cpp
```

Die Hauptaufgabe des IO-Systems liegt darin, den Cpu-Zugriff auf das Templatemethode und die Devices bereitzustellen. Devices werden mit Memory-Mapped IO angesprochen oder mit Port-Mapped IO. Das IO-System ist auch dafür zuständig, die Cpu und die Devices zu steuern, indem es den Clockimpuls weitergibt und Signale setzt.
Wie eine IO-System-Implementation seine Devices implementiert, ist völlig freigestellt. Es kann jedoch der im Kapitel 8.2 vorgeschlagene Ansatz verwendet werden. Eine neue IO-System-Implementation muss nur die Methoden von CI0-System implementieren. Diese werden im folgenden vorgestellt, falls dies nicht schon geschehen ist. Tabelle 4 gibt Auskunft, in welchen Kapiteln die jeweilige Methode beschrieben wird. Obwohl die Methoden von CIoSystem formal nicht als rein virtuell deklariert worden sind, müssen sie trotzdem alle überschrieben werden. Wird eine dieser Methoden nicht überschrieben, wird deren Implementation im CIoSystem aufgerufen, die das Programm mit ASSERT(0) abbricht.

isVerifyOk **dump** **shallEmulatorWait**	Kapitel 5.4
powerOn **init** **setSignal**	Kapitel 5.2.1
setSignal **getSignal**	Kapitel 4.1
clock	Kapitel 4.5
getError **clearError**	Kapitel 4.2.3

Tabelle 4: Bereits eingeführte Methoden von CIoSystem

getExtraSignals(int& cnt, char& names, ESignalType*&)**
CIoSystem definiert nur das Signal SIGNAL_POWER_ON_BUTTON. Damit der User via Emulator auch eventuell zusätzliche Signale eines bestimmten IO-Systems benutzen kann, muss das IO-System dem Emulator eine Liste seiner zusätzlichen Signale geben. Ein Beispiel eines solchen Signals wäre ein Reset-Knopf-Signal. In der aktuellen Version des Emulators wird getExtraSignals noch nicht dahingehend verwendet, dass der User nun beliebige Knöpfe des IO-Systems betätigen kann. Dies wurde aus Zeitnot noch nicht implementiert.
Beispiel CIoSystemMips: Das Mipsboard IO-System hat keine zusätzlichen Signale. Die Implementation enthält trotzdem ein Beispiel, wie die Werte syntaktisch korrekt zurückzugeben sind.

isLittleEndian()
Gibt true zurück, falls das IO-System als little Endian konfiguriert ist, andernfalls false. Ist als Methode definiert, damit auch IO-Systeme unterstützt werden, die ihre Endianness dynamisch ändern können.
Beispiel CIoSystemMips: Returniert die Konstante MIPSBOARD_IS_LITTLE_ENDIAN, welche im Konfigurations-Headerfile definiert ist.

TRegu32 read32 (TPAddr paddr)
void write32 (TPAddr paddr, TRegu32 data)
Zugriff aufs Templatemethode, bzw. auf Memmory-Mapped Devices. 32 Steht für 32 Bit Zugriff und kann auch durch 8,16,64 ersetzt werden.

TRegu32 readIo32 (TIoAddr ioaddr)
void writeIo32(TIoAddr ioaddr, TRegu32 data)
Zugriff aufs auf Port IO-Mapped Devices. 32 Steht für 32 Bit Zugriff und kann auch durch 8,16,64 ersetzt werden.

8.2 Devices

Ein Framework für Devices sollte folgenden Punkt berücksichtigen: Der Kern der Devices, der die eigentliche Funktionalität implementiert, sollte losgelöst von der Implementation des IO-Systems bestehen. So kann er in verschiedenen IO-Systemen wiederverwendet werden. Dies ist auch bei den realen Devices der Fall. Ein Interface übernimmt die Kommunikation zwischen dem Device-Kern und dem IO-System. Figur 6 gibt diesen Sachverhalt wider. Die Devices DUART (Universal Asynchronous Receiver-Transmitter) und PIT (Programable Interrupt Timer) können für verschiedene IO-Systeme wiederverwendet werden.

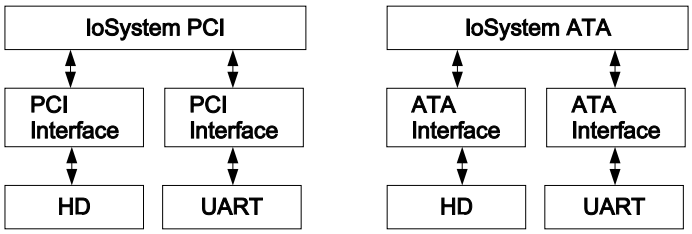


Abbildung 6: Ein Interface übernimmt Kommunikation zwischen Devicekern und dem IoSystem. Devicekern und Deviceinterface können wiederverwendet werden.

Figur 7 illustriert die Umsetzung in C++ Klassen. Wurde der Kern des Devices, zum Beispiel CDeviceUart, und ein bestimmtes Interface, zum Beipsiel CDeviceInterfaceMips, implementiert, kann damit das eigentliche Device, CDeviceMipsUart, sehr einfach implementiert werden: Von der Interface-Klasse ableiten, und eine Instanz des Device-Kerns als Member instanzieren. Methoden müssen meist keine mehr implementiert werden, da diese schon in CDeviceInterfaceMips implementiert sind.

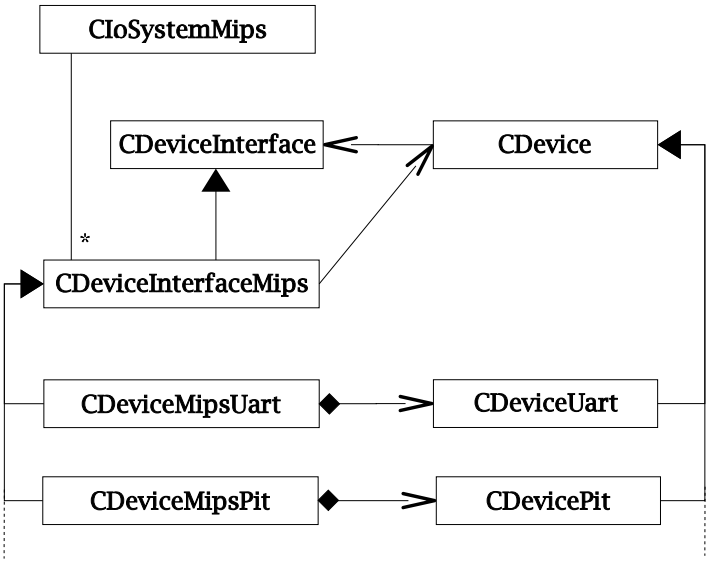


Abbildung 7: Umsetzung der Grafik 6 in ein UML Diagram.

Der Leser sei darauf hingewiesen, dass die Assoziation zwischen CDeviceMipsUart und CDeviceUart gerichtet ist. CDeviceUart kennt CDeviceMipsUart nicht, sondern nur CDeviceInterface (via CDevice). Der Kern des Devices kann nur mit der abstrakten Klasse CDeviceInterface kommunizieren. Dies ist durchaus logisch, da der Kern nicht wissen kann, an welches Interface er einmal angeschlossen wird. Umgekehrt jedoch, kann CDeviceMipsUart mit CDeviceUart kommunzieren. Es ist gerade der Sinn der Klasse CDeviceMipsUart, dass sie alle Eigenheiten des Interfaces kennt sowie alle Eigenheiten des Device-Kerns.
Im Gegensatz zu den Klassen CCpu und CIoSystem, sind alle Methoden virtual. Somit ist ein schönes OO-Programmieren möglich. Hier ist der Geschwindigkeitsvorteil, der durch das Vermeiden von Virtual-Methoden erreicht würde, vernachlässigbar. Die Devices werden relativ zum Abarbeiten einer Instruktion nur selten verwendet.
Vom Prinzip, dass zwischen Device-Kern und IO-System immer ein Interface steckt, ist im Beispiel des Mipsboard IO-Systems das Ram ausgenommen. CRam ist auch von CDevice abgeleitet, aber es wird direkt von CIoSystemMips angesprochen. Deshalb, weil das Ram sehr oft benützt wird, durch den Instruktionsfetch pro Instruktion mindestens einmal. Ohne Interface ist der Zugriff aufs Ram schneller. Weil CRam direkt in CIoSystem instanziert ist, also kein Pointer gebraucht wird, tritt auch der Virtual-Mechanismus nicht in Kraft.Und als virtuell deklarierte Methoden werden sie direkt, ohne Umweg über die Virtualtable, aufgerufen.

8.2.1 Class CDevice

```
Source:                 Devices/Device.h,.cpp
Implementionsbeispiel: Devices/DevicePit82C54.h,.cpp
                        Devices/DeviceUartSnc2681.h,.cpp
```

Die Klasse CDevice ist die Basisklasse aller Device-Kerne. Viele ihrer Methoden wurden bereits vorgestellt. Tabelle 5 zeigt, welche Methoden in welchen Kapiteln beschrieben werden. Die restlichen Methoden werden nachfolgend eingeführt.

isVerifyOk **dump** **shallEmulatorWait**	Kapitel 5.4
powerOn **init** **setSignal**	Kapitel 5.2.1
setSignal **getSignal**	Kapitel 4.1
clock	Kapitel 4.5
getError **clearError**	Kapitel 4.2.3

Tabelle 5: Bereits eingeführte Methoden für CDevice

const bool mIsLittleEndian;
Hat den Wert true, wenn die Methoden read/write auf little Endian Basis operieren. Jedes Device hat demzufolge seine individuelle Endianess. Es ist Aufgabe des Device-Interfaces, die Endianess des Device der Endianness des IO-Systems anzupassen. Folgender Code, der in jedem Device vorkommen kann, soll verdeutlichen, wieso jedes Device seine Endianness bestimmen können muss.

```
char ram[256];
memset( ram, 0, sizeof(ram) );
ram[0] = 0xFF;
TBitu32 data1 = *(TBitu32*)(&ram[0]);
TBitu32 data2 = 0x12345678;
```

Ist der Host ein little Endian System, hat data1 den Wert 0xFF. Ist der Host jedoch ein big Endian-System, hat data1 den Wert 0xFF000000. Demzufolge ist data1 von der Endianness des Host abhängig. data2 hingegegen hat immer den Wert

0x12345678, unabhänig von der Endianness des Host. Die Endianness eines Devices ist also variabel, entspricht aber nicht unbedingt der Endianness des Hosts.
virtual TRegu32 read32 (TPAddr);
virtual void write32(TPAddr , TRegu32);
Read/Write-Zugriff auf das Device, jeweils in einer 8,16,32 oder 64Bit Variante. TPAddr ist relativ zum Adressraum des Devices. Adresse 0 bezeichnet demzufolge das erste Byte im Adressraum des Devices. Die Spezifikation des Devices gibt vor, wie auf Lese- und Schreibzugriffe zu reagieren . Es ist die Aufgabe des Interfaces, die physikalische Adresse, die die Cpu dem IO-System miteilt, in eine Adresse relativ zum Adressraum des Devices umzuwandeln.

8.2.2 Class CDeviceInterface

```
Source:                Devices/DeviceInterface.h,.cpp
Implementionsbeispiel: Arch/IoSystemMips/DeviceInterfaceMips.h,.cpp
```

Das API von CDeviceInterface ist darauf ausgerichted, dass ein CDevice mit CDeviceInterace kommunizieren kann, und ist demzufolge sehr klein gehalten. Alles, was darüber hinausgeht, muss von der abgeleiteten Klasse, zum Beispiel CDeviceInterfaceMips deklariert werden. Der Grund liegt darin, dass möglichst wenige Vorgaben an eine Interface-Implementation gestellt werden, damit sie möglichst frei in ihrem Design ist.
virtual int id()
Gibt eine id zurück, die das Device innerhalb des IO-Systems eineindeutig identifiziert. Diese wird vom Device-Kern meistens verwendet, um sie zusammen mit einer Debuggmeldung auszugegeben, damit der User weiss, welches Device die Meldung ausgegeben hat.
virtual void setDeviceSignal(int no, ESignalState s)
virtual ESignalState getDeviceSignal(int no)
Wird vom Device-Kern benutzt, um sein Ausgangssignal, bzw. das Eingangssignal des Interfaces zu setzen. Für no wird die Nummer eines Ausgangssignals des Device-Kerns verwendet.

9 Class CCpuIoInterface

```
Source:                IoSystem/CIoSystem.h,.cpp
Implementionsbeispiel: Arch/IoSystemMips/IoSystemMips.h,.cpp
```

Die Augabe der Klasse CCpuIoInterface liegt darin, die Outputsignale des IO-Systems mit den entsprechenden Inputsignalen der Cpu zu verbinden und umgekehrt. Es ist demzufolge eine sehr kleine Klasse, die in der Regel nur zwei Lookuptables hat, und die nachfolgend vorgestellten Methoden. Es kann jedoch durchaus sein, dass das Routing der Signale etwas komplexer ist als nur 1:1.
CCpuIoInterface muss wie CCpu und CIoSystem abgeleitet und implementiert werden. Denn eine CCpuIoInterface Implementation muss sowohl die implementierende CCpu-Klasse sowie auch die implementierende CIoSystem-Klasse kennen. Nur so kennt sie die Konstanten der Input-/Outputsignale.

void setCpuSignal(int no, ESignalState s)
ESignalState getCpuSignal(int no)
Wird vom IO-System aufgerufen, um ein Inputsignal der Cpu zu setzen, beziehungsweise abzufragen. Für den Parameter no verwendet das IO-System die selbstdefinierte Konstante des eigenen Outputsignals.

void setIoSystemSignal(int no, ESignalState s)
ESignalState getIoSystemSignal(int no)
Wird von der Cpu aufgerufen, um ein Inputsignal des IO-Systems zu setzen beziehungsweise abzufragen. Für den Parameter no verwendet die Cpu die selbstdefinierte Konstante des eigenen Outputsignals.

10 Hinzufügen neuer Implementationen am Beipsiel des MipsR3051

Dieses Kapitel dient als Anleitung zum Implementieren von neuen Architekturen und zum Bearbeiten vom Makefile Makefile.user, das zum Kompilieren des Projektes verwendet wird.

10.1 Einleitung

Diese Anleitung setzt das in den vorherigen Kapiteln vermittelte Wissen über die Basisklassen CCpu, CIoSystem, CCpuIoInterface, CDevice und CDeviceInterface voraus. Sie erklärt, wie neue Implementationen formal zu Implementieren sind.

10.2 Konfigurationsheaderfile

Konfiguriert wird das gesamte Projekt durch ein Konfigurationsfile, das nichts anderes als eine C++ Headerfile ist. In erster Linie bedeuted dies, dass die vom Emulator zu emulierende Cpu- und IO-Systemarchitektur ausgewählt wird. Zum Beispiel kann als Cpu der MipsR3051 und als IO-System das MipsBoard ausgewählt werden. Des weitern werden Konfigurationen gemacht, die globale Gültigkeit haben und in jedem Fall verwendet werden. Gleichgültig welche Architektur verwendet wird. Als letztes kann jede Architektur ihre individuellen Konfigurationen vornehmen. Für ein und dieselbe Architektur können auch mehrere Konfigurationsfiles erstellt werden. Zum Beispiel eine Debuggversion und eine Retailversion. Zusammengefasst bestimmt ein Konfigurationsfile also die emulierte Architektur plus deren Konfiguration.
Die folgenden Schritte beschreiben das Erstellen eines neuen Konfigurationsfiles. Als Referenz kann das Beispiel Emulator/Configs/ConfigMipsDebug.h betrachtet werden.

1. Einen Namen für die Konfiguration festlegen.Beispiel: MipsDebug.
2. Eine Kopie der Vorlage Emulator/Configs/ConfigTemplate.h erstellen, und analog zum Beispiel Emulator/Configs/ConfigMipsDebug.h benennen.
3. Bestimmen der zu emulierenden Cpu, IoSystem und CpuIoInterface. Im neu erstelltem Konfigurationsfile, müssen folgende Konstanten angepasst werden.

```
#define CONFIG_CPU                          CCpuMipsR3051
#define CONFIG_CPU_INCLUDE                  "CpuMipsR3051.h"
#define CONFIG_CPU_ARGS                     ( 0 )

#define CONFIG_CPU                          CCpuMipsR3051
#define CONFIG_IOSYSTEM_INCLUDE             "IoSystemMips.h"
#define CONFIG_IOSYSTEM_ARGS

#define CONFIG_CPU_IO_INTERFACE             CCpuIoInterfaceMipsR3051Mips
#define CONFIG_CPU_IO_INTERFACE_INCLUDE     "CpuIoInterfaceMipsR3051Mips.h"
#define CONFIG_CPU_IO_INTERFACE_ARGS
```

   Die erste Zeile eines Blocks definiert den Namen der Klasse. In der zweiten Zeile wird das Headerfile angegeben, in dem die Klasse deklariert ist. Die dritte Zeile gibt die Argumente an, die dem Konstruktor übergeben werden. Braucht der Konstruktor keine Argumente, wird wie im Beispiel `CONFIG_IOSYSTEM_ARGS` die Konstante “leer” definiert. Braucht er eines oder mehrere Argumente, werden sie inklusive Klammern angegeben, wie im Beispiel `CONFIG_CPU_ARGS`, wo 0 als Argument übergeben wird. [8]
4. Die Typen `TPAddr` (physikalische Adresse), `TVAddr` (virtuelle Adresse) und `TIoAddr` (Port IO Adresse) müssen via typedef definiert werden. Im Beispiel haben wir 32 Bit physikalische, 32 Bit virtuelle und 16 Bit Port IO Adressen. [9]

[8] In Globals.cpp, teilweise auch in Kapitel 5.2, sieht man die Verwendung dieser Konstanten.
[9] TPAddr, TVAddr und TIoAddr werden in Kapitel 4.4 behandelt

```
typedef TRegu32 TPAddr;
typedef TRegu32 TVAddr;
typedef TRegu16 TIoAddr;
```

5. Nun können beliebige Defines oder Deklarationen angefügt werden, die die verwendete Architektur konfigurieren. Für Beispiele sei der Leser auf die Beispielkonfiguration Emulator/Configs/ConfigMipsDebug.h verwiesen.

Beim Kompilieren des Projektes wird angegeben, welches Konfigurationsheaderfile verwendet werden soll. Dies wird in Kapitel 10.8 erklärt, und ist momentant noch nicht von Bedeutung. Der Leser sei daran erinnert, dass auch durch Kommandozeilenargumente Konfigurationen vorgenommen werden können.

10.3 Host-Headerfile / Host-Makefile

Damit das Projekt portabel ist, und an jeden Host optimal angepasst werden kann, müssen für jeden Host ein Host-Headerfile und ein Host-Makefile erstellt werden. Der Host ist das System, auf dem der Emulator ausgeführt wird.
Ein neuer Host wird folgendermassen zum Projekt hinzugfügt. Als Beispiele dienen die Files im Verzeichnis Emulator/Host/Ia32.

1. Einen Namen für den Host festlegen. Beispiel: Ia32.
2. Eine Kopie des Verzeichnis (inklusive Inhalt) Emulator/Host/Template anlegen und in den in 1 festgelegten Namen umbenennen. Auch das darin enthaltenene Host-Headerfile “HostTemplate.h” muss analog unbenannt werden. Beispiel der neuen Namen:

   ```
   Emulator/Host/Ia32/HostIa32.h
   Emulator/Host/Ia32/Makefile.host
   ```

3. Im Host-Headerfile folgende Einstellungen vornehmen.
   (a) HOST_IS_LITTLE_ENDIAN auf 1 setzen, falls der Host ein little Endian-System ist, 0 andernfalls.
   (b) Die TBit Datentypen so definieren, dass sie genau gleich gross sind, wie es der Name besagt. TBitu32 muss also genau 32 Bit gross sein. Das Manual des verwendeten Compilers gibt Auskunft darüber, wie gross die primitiven Datentypen wie char,short,int,long implementiert sind. Kapitel 4.3 geht näher auf die Datenypen TBit und TReg ein.
   (c) Die TReg Datentypen so definieren, dass sie mindestens so gross sind, wie es der Name besagt. Die Definitionen sollten so erfolgen, dass die Registergrössen, die die Hostcpu bereitstellt, ausgenutzt werden können. Deshalb sollte möglichst int verwendet werden, da int meist der Registergrösse der Cpu entspricht. Im Falle des x86 kann auch char und short verwendet werden, da der x86 auch 8 Bit und 16 Bit-Register hat.
4. Im Host-Makefile wird der Variable GCC der zu verwendende Compiler zugewiesen, zum Beispiel g++, der Variable LD der zu verwendende Linker, zum Beispiel g++.

Für die restlichen Einstellungsmöglichkeiten im Host-Headerfile sei der Leser auf die Dokumentation im Code verwiesen.

10.4 Cpu

Die folgende Liste beschreibt die wenigen notwendigen Schritte, um eine neue Cpu-Architektur zu implementieren.

1. Im Verzeichnis Emulator/Arch ein neues Unterverzeichnis erstellen, vorteilhaft mit dem Namen der Cpu. Beispiel: Emulator/Arch/CpuMipsR3051. In diesem Verzeichnis können beliebig viele Sourcefiles erstellt werden, wobei deren Namen irrelevant sind. Alle Sourcen, die für Co-Prozessoren, Alu oder ähnliches gebraucht werden, kommen in dieses Verzeichnis.

2. In allen Sourcefiles mit `#include CONFIG_INCLUDE` das Konfigurationsfile einbinden.

3. Eine neue Klasse erstellen, die von CCpu abgeleitet ist. Alle Methoden von CCpu müssen überschrieben werden. Kapitel 7.1 beschrei die Funktionalität, die diese Methoden besitzen müssen. Das Beispiel des MipsR3051 wählt jedoch die etwas kompliziertere Variante, die gleich anschliessend vorgestellt wird.

Damit sind bereits alle notwendigen Schritte getan. Wie eventuell vorhandene Co-Prozessoren oder ALU's 30 implementiert werden, ist vollkommen freigestellt. Sollen die vorgefertigten Methoden verwendet werden, die CCpuEnhanced und CAlu bieten, ersetzt man Schritt 3 durch die folgenden:

3. Instanzieren der CCpuEnhanced Klasse.

   ```
   class   CCpuMipsR3051; // forward declaration
   typedef CCpuEnhanced< CCpuMipsR3051, CInstructionMipsR3051 >
           CCpuMipsR3051Base;
   ```

   Als erstes Template-Argument ist die schlussendliche Cpu Klasse, als zweites die verwendete Instruktionsklasse anzugeben. Die schlussendliche Klasse ist im Beispiel CCpuMipsR3051. Diese Instruktionsklasse muss nicht, wie dies im Beispiel CInstructionMipsR3051 der Fall ist, von CInstruktion abgeleitet sein.

4. Von der im Schritt 3 instanzierten Klasse die schlussendliche Klasse ableiten.

   ```
   class   CCpuMipsR3051 : public CCpuMipsR3051Base
   { /*.....*/
   };
   ```

   Alle Methoden von CCpu (CCpuEnhanced ist von CCpu abgeleitet) müssen implementiert werden. Zusätzlich müssen einige Methoden von CCpuEnhanced überschrieben werden. Im File CCpuEnhanced.h sind die Methoden, die noch implementiert werden müssen, gekennzeichnet.

5. Sourcefiles, die Elemente oder Methoden der Klasse CCpuEnhanced referenzieren, müssen das Headerfile CCpuEnhanced.hxx beinhalten, statt wie normalerweise üblich das .h Headerfile.

10.5 Iosytem

Die folgende Liste beschreibt die wenigen notwendigen Schritte, um eine neue IO-System-Architektur zu implementieren.

1. Im Verzeichnis Emulator/Arch ein neues Unterverzeichnis erstellen, vorteilhaft mit dem Namen des IoSystem. Beispiel: Emulator/Arch/IoSystemMips. In diesem Verzeichnis können beliebig viele Sourcefiles erstellt werden, wobei deren Namen irrelevant sind. Alle Sourcen, die für die Implementation des IO-Systems gebraucht werden, kommen in dieses Verzeichnis.

2. In allen Sourcefiles mit `#include CONFIG_INCLUDE` das Konfigurationsfile einbinden.

3. Eine neue Klasse erstellen, die von CIoSystem abgeleited ist. Alle Methoden von CIoSystem müssen überschrieben werden. Kapitel 8.1 die Funktionalität, die diese Methoden besitzen müssen.

4. Die Methoden getSignal und setSignal müssen das Signal `SIGNAL_POWER_ON_BUTTON`, definiert in CIoSystem, erkennen. Deshalb darf beim Definieren der Konstanten für Input-/Outputsignale auch nicht vergessen werden, dass CIoSystem bereits die Konstante `SIGNAL_POWER_ON_BUTTON` definiert hat. Allgemein gesehen sind alle Konstanten von 0 bis exklusiv `CIoSystem::SIGNAL_COUNT` vergeben.

10.6 Devices

Der Leser sei darin erinnert, dass ein IO-System komplett frei ist, wie es seine Devices implementiert. Dieser Abschnitt kann übersprungen werden, wenn der im Kapitel 8.2 vorgeschlagene Ansatz nicht verwendet wird.
Um ein neues Device zu implementieren, müssen zuerst die zwei Kompomenten Device-Interface und Device-Kern implementiert sein. Wie diese zwei Komponenten selbst implementiert werden, wird nachfolgend erklärt.

1. Im Verzeichnis Device eine neue Klasse erstellen, die von CDevice abgeleitet ist.
2. Alle Methoden überschreiben, deren Standard-Implementation (in Devices/Device.cpp) unzureichend ist.

1. Im Verzeichnis Device eine neue Klasse erstellen, die von CDevice abgeleitet ist.
2. Alle Methoden überschreiben, deren Standard-Implementation (in Devices/Device.cpp) unzureichend ist.

10.7 Cpu-Io Interface

Die folgende Liste beschreibt die wenigen notwendigen Schritte, um eine neues Cpu-IO Interface zu implementieren.

1. Im Verzeichniss Emulator/Arch/CpuIoInterfaces eine neue Klasse erstellen, die von CCpuIoSystem abgeleitet ist. Alle Methoden von CCpuIoSystem müssen überschrieben werden. Kapitel 9 beschreibt die Funktionalität, die diese Methoden besitzen müssen.
2. In allen Sourcefiles mit `#include CONFIG_INCLUDE` das Konfigurationsfile einbinden.

10.8 Kompilieren

Nachdem nun die gewünschte Architektur implementiert worden ist, muss das Projekt kompiliert und gelinkt werden. Die nötigen Einstellungen werden im Makefile “Makefile.user” gemacht. Dabei gibt es 4 wichtige Parametergruppen, die einzustellen sind.

Konfiguration bestimmt die Eigenschaften des Target. Jede Konfiguration hat ihr Konfigurationsheaderfile.

Host bestimmt die Architektur, auf der der Emulator läuft. Im Beispiel ist dies Ia32. Jeder Host hat ein Hostheaderfile und ein Host-Makefile. Das Hostheaderfile definiert Datentypen und Methoden, die vom Host abhängig sind. Dies sind vor allem die TReg und TBit Datentypen. Es wird automatisch in das bereits vorgestellte Konfigurationsheaderfile eingebunden[10]. Das Makefile bestimmt die Tools (Compiler, Linker, Assembler etc), die von make verwendet werden müssen. Das Hostheaderfile hat einen Namen in der Form Emulator/Host/Ia32/HostIa32.h, das Host-Makefile einen Namen in der Form Emulator/Host/Ia32/Makefile.host; dabei muss natürlich “Ia32” durch den Namen des jeweiligen Host ersetzt werden.

Target bestimmt die Architektur, die emuliert werden soll. Im Beispiel ist dies die MipsR3051 Cpu und das Mipsboard IO-System.

[10] Mit der Anweisung #include HOST_INCLUDE

Version fasst alle drei vorhergehenden Punkte zusammen. In einer Version ist der Host, der Target und dessen Konfiguration festgelegt. Zusätzlich sind in einer Version auch noch Compileroptionen festgelegt. Für eine bestimmtes Konfiguration-Host-Target Tripel kann somit eine Debugg-Version erstellt werden, wo mit -g (Informationen für den Debugger erstellen) kompiliert wird, und eine Retail Version, wo mit -O3 (optimieren) kompiliert wird.

Mit den folgenden Schritten wird eine neue Version in das Makefile.user eingefügt.

1. Das File Makefile.user öffnen.
2. Einen Namen für die Version bestimmen. Beispiel: MipsDebug oder MipsRetail.
3. Eine Kopie des “TEMPLATE_VERSION” Blocks erstellen. Danach überall im kopiertem Block “TEMPLATE_VERSION” durch den in 2 bestimmten Namen ersetzen.
4. Der Variable CONFIG muss der Name der zu verwendenden Konfiguration zugewiesen werden. Dieser Name wurde in Abschnitt 10.2 festgelegt. Beispiel: MipsDebug.
5. Der Variable EMUHOST wird der Name des zu verwendenden Hosts zugewiesen. Dieser Name wird in Abschnitt 10.3 bestimmt. Beispiel: Ia32.
6. Den Variablen ARCHCPU und ARCHIO werden die Namen der Verzeichnisse zugewiesen, in denen sich die Targetarchitektur befindet. Beispiel: CpuMipsR3051 bzw IoSystemMips. *Achtung*: Auch im Konfigurationsheaderfile wurde die Architektur schon angegeben. Natürlich müssen die Definitionen an den beiden Orten kohärent sein.
7. Die Optionen für den Compiler bestimmen.
8. Der Variable VERSION (befindet sich zuoberst in Makefile.user), den Namen der Version zuweisen, die kompiliert werden soll. Dies ist der im Punkt 2 bestimmte Namen.
9. Makefile.user abspeichern. [11]

Mit dem Befehl “make” wird das Projekt kompiliert und gelinkt. Eine Meldung am Schluss dieses Prozesses gibt den Ort an, an dem sich das erzeugte Executable befindet.
Wurden einmal alle gewünschten Versionen im Makefile.user erstellt, kann vor dem “make” Befehl bequem zwischen den Versionen hin- und hergeschalten werden, in dem die VERSION Variable im Makefile.user auf den Namen der zu erstellenden Version gesetzt wird. Jede Version hat ihr eigenes Ausgabeverzeichnis. Somit können verschiedene Versionen bequem nebeneinander getestet und verglichen werden.

[11] Die Variablen im Teilblock “input/output” müssen normalerweise nicht geändert werden. Sollte dies trotzdem nötig sein, sei der Leser auf die Dokumentation im Makefi le.user verwiesen.

11 Evaluation

In diesem Kapitel wird

Korrektheit 1 In diesem Punkt soll die Korrektheit des Emulators überprüft werden. Sie wird überprüft, indem Topsy auf dem Emulator ausgeführt wird.

Kompiliert wird die Platform mit der Version MipsDebug. Die Methode CDevice-Pit82C54::clock ist deaktiviert, indem als erstes Statement “return;” eingefügt ist. Siehe auch Punkt Korrektheit 4. Im Folgenden wird die Version und die verwendeten Optionen aufgelistet:

```
-- Makefile.user --
VERSION = MipsDebug

-- Optionen --
kernel image     'kernel.srec'
bios image       'bios.srec'
mpem console in  'stdin'
mpem console out 'stdout'
uart in          'stdin'd
uart out         'stdout'
verify           off
wait             flags 0xffef
                 level 1
dump             occasion 0
                 level 0
                 dump output to 'mpemlog'
                 log output to 'mpemlog'
```

Topsy kann aufgestarted werden. Nach 8 Stunden Laufzeit werden in der Topsy Shell die folgenden Befehle ohne Probleme ausgeführt:

```
hello world
help
ps
start shell
exit
```

Die Korrektness des Emulators wurde bestätigt: Topsy kann für lange Zeit ausgeführt und benützt werden, ohne dass Fehler auftreten. Treten Fehler auf sind sie zurückzuführen auf unkorrektes Verhalten von Topsy, Fehlermeldungen von Topsy, Fehler der Implementation (ASSERTION macros) und undefined/undpredictable Behaviour.

Topsy konnte auch mit anderen Versionen und mit andern Optionen aufgestartet werden, die hier nicht einzeln aufgelisted werden. Es sind keine Versionen/Optionen bekannt, mit denen Topsy nicht aufgestartet werden kann. Es sind keine Topsy-Befehle bekannt, die nicht ausgeführt werden können, so lange der Clock des Devices Pit82C54 deaktiviert ist.

Korrektheit 2 In diesem Punkt soll wiederum die Korrektheit des Emulators überprüft werden. Die Überprüfung erfolgt, indem das Verifing des Emulators verwendet wird.

Überprüft werden die ersten 300’000 Clockzyklen. Das Verifyfile wird von MipsSim erstellt. 300’000 Zyklen werden gewählt, weil sie genügen, um Topsy aufzustarten und in den Loop zu bringen, in dem die Topsy Shell auf die Eingabe des Users wartet. Betrachtet werden jegliche Register des MipsR3051 und dessen System-Co-Prozessors, inklusive der gesamten TLB. Nicht betrachtet werden die Register der Devices.

Kompiliert wird die Platform mit der Version MipsDebug. Die Methode CDevice-Pit82C54::clock war aktiviert. Im Folgenden wird die Version und die verwendeten Optionen aufgelistet:

```
-- Makefile.user --
VERSION = MipsDebug

-- Optionen --
kernel image     'kernel.srec'
bios image       'bios.srec'
mpem console in  'stdin'
mpem console out 'stdout'
uart in          'stdin'
uart out         'stdout'
verify           '../../MipsSimSensor/verify'
                 from 5 to 300000
                 verifyfile starts at 0
wait             flags 0xf
                 level 1
dump             occasion 0
                 level 0
                 dump output to 'mpemlog'
                 log output to 'mpemlog'
```

Die Korrektness konnte wieder bestätigt werden. Das Verifying hat keine Inkonsistenzen zwischen Emulator und MipsSim gefunden.

Korrektheit 3 Die Korrektheit des Verifying, wie in Punkt 4 verwendet, wird hier überprüft.

Dazu soll der Inhalt des Verifyfile absichtlich verfälscht werden, um zu testen, ob daraus tatsächlich eine Fehlermeldung entsteht. Das in Punkt 2 verwendete Verifyfile ist jedoch zu gross (500 Megabytes), um editiert zu werden. Deshalb wird ein neues Verifyfile erstellt, das lediglich die Zyklen bis 1000 enthält.

Es wird dieselbe Kompilitation wie in Punkt Korrektheit 2 verwendet, ohne neu zu kompilieren. Der einzige Unterschied sind die folgenden Verify-Optionen. Sie wurden in beiden Fällen mit Kommandozeilen-Argumenten gesetzt.

```
verify           '../../MipsSimSensor/verify'
                 from 5 to 1000
                 verifyfile starts at 0
```

Da hier ein anderes Verifyfile verwendet wird als in Punkt Korrektheit 2, wird sicherheitshalber zunächst nichts am Verifyfile geändert. Der Verify-Prozess wird gestartet. Es werden korrekterweise keine Inkonsistenzen entdeckt.

Im Zyklus 100 wird der Wert des ersten Generalpurpose-Registers von 0 auf 1 geändert. Der Verify-Prozess wird gestartet. Der Fehler wird korrekt erkannt:

```
INCOHERENCY after clock 100 : gpr 0x0 invalid : mipsim = 1!=0
```

Danach die gesammte Zeile des Zyklus 100 gelöscht. Der Verify Prozess wird gestartet. Der Fehler wird korrekt erkannt.

```
INCOHERENCY after clock 100 : invalid clk : verify=101 != 100
```

Daraus lassen sich zwei Schlüsse ziehen. Erstens: Das Verifying funktioniert. Zweitens: Der Test in Punkt Korrektheit 2 ist aussagekräftig, da das Verifying nicht in jedem Fall, ob Inkonsistenzen auftreten oder nicht, meldet, dass alles in Ordnung sei. Dies wurde mit diesem Test bewiesen.

Es sind keine Konfigurationen/Optionen bekannt, mit denen das Verifying nicht funktionieren würde.

Korrektheit 4 In diesem Punkt werden zwei bekannte Fehler im Zusammenhang mit dem Device Pit82C54 angeführt. Es sei vorweggenommen, dass beide Fehler in der aktuellen Version nicht auftreten; sie wurden jedoch nie explizit behoben.

Erster Fehler: Wird in der Topsy Shell der Befehl “start reserve” eingegeben, ist das Resultat des ausgeführten Prozesses deterministisch.

Zweiter Fehler: Wird die Methode CDevicePit82C54::clock nicht durch ein frühzeitiges “return;”-Statement abgebrochen, versteht die Topsy Shell lange Befehle nicht richtig. Wird zum Beispiel “start reserve” eingegeben, wird dies von Topsy als “start reserse” interpretiert. Offensichtlich wird jeweils der zwölfte Buchstabe durch den ersten ersetzt.

Die beiden Fehler in der aktuellen Version nicht mehr auf. [12] CDevicePit82C54::clock wird nicht mehr verfrüht abgebrochen. Zwischen der vorhergehenden Version, in der die Fehler auftraten und der aktuellen Version, wurden ein paar unwesentliche Änderungen im CEmulator und CCpuIoInterfaceMipsR3051Mips vorgenommen. Da die Ursache der Fehler nie explizit behoben wurde, wird angenommen, dass die Ursache der Fehler noch besteht, obwohl sie nicht mehr auftreten.

Korrektheit 5 In diesem Punkt wird auf Probleme mit veschiedenen Compilern hingewiesen. Entwickelt wurde die Platform mit gcc 3.2.2.

Wegen dem exzessivem Gebrauch von Templates kann die Platform von älteren Compilern nicht kompiliert werden. Dies schliesst insbesondere gcc 2.95 und Visiual C++ 6.0 mit ein. Dies ist ein Fehler der jeweiligen Compiler; gcc gibt eine entsprechende Bugmeldung aus.

Die Datei Emulator/Emulator.cpp kann von gcc 3.2.2 nicht mit der Option “-finline-limit=5000” kompiliert werden. Dies ist wiederum ein Fehler dieses Compilers, denn er gibt eine entsprechende Bugmeldung aus. Diese Option wird benützt, damit die “grossen” inline-Methoden wie CCpuMipsR3051::getOperand wirklich inline eingesetzt werden. Der Leser sei daran erinnert, dass der eingesetzte Code der einzigen Zeile eines case labels entspricht, da der Parameter der switch Anweisung zur Kompilierungszeit bekannt und konstant ist. Für einen Workaround sei der Leser auf die Dateien Makefile verwiesen.

Gcc 3.3.1 konnte das Projekt nicht kompilieren, da er eine 64 Bit Integer-Literalkonstante nicht akzeptiert. Es wird angenommen, dass dies mit dem in Kapitel 3.2 besprochenem Punkt zusammenhängt, wonach “long long” Typen nicht dem ANSI C++ Standard entsprechen. Dieser Fehler wurde nicht weiter untersucht.

Geschwindigkeit 1 In diesem Punkt wird die Geschwindigkeit des Emulators gemessen. Als Referenz wird die Geschwindigkeit von MipsSim genommen. Beide Emulatoren implementieren aus Hardwaresicht diesselbe Funktionalität.[13]

In diesem Emulator wie auch in MipsSim wird die Zeit gemessen, die die ersten 1’000’0000 Instruktionen von Topsy benötigen, um auf dem jeweiligen Emulator ausgeführt zu werden. Die Zeit, die der jeweilige Emulator braucht, um sich selbst aufzustarten, zu initialiseren, und um das Kernelimage einzulesen, ist explizit nicht eingerechnet. Als Overhead kommt auf den beiden Seiten hinzu, die Zeit zu messen.

Der Emulator wird mit der Version MipsRetail kompiliert. Da in dieser Version alle Debuggfeatures ausgeschalten sind, werden an dieser Stelle die Optionen nicht angezeigt, da sie irrelevant sind. Im Sourcefile Emulator.cpp müssen die Konstanten gesetzt sein:

```
#define EMULATOR_MEASURE_TIME            1
#define EMULATOR_MEASURE_TIME_FROM       5
#define EMULATOR_MEASURE_TIME_TO         1000000
```

Die erste Instruktion des Kernels wird im Zyklys 5 ausgeführt. Die vorherigen Zyklen gehören zum BIOS. Natürlich wurde auch auf der MipsSim Seite erst ab Clockzyklus 5 gemessen.

[12] Bezüglich des ersten Fehler kann es aber durchaus sein, dass der Ausgeführte Prozess bis zu zehn mal daselbe Verhalten zeigt.

[13] Im Gegensatz zu MipsSim implentiert dieser Emulator noch kein Netzwerkinterface. Es wurde jedoch schlussendlich von MipsSim nicht benutzt, da es nicht initialisiert werden konnte. Dass MipsSim nicht mehr, bzw andere, ’schwierigere’ Instruktionen ausführen muss als dieser Emulator, wurde in Punkt Korrektheit 2 bewiesen.

Es werden je 10 Messungen gemacht. Tabelle 6 zeigt die Resultate [14] Daraus ist ersichtlich, dass dieser Emulator um cirka den Faktor vier schneller ist als der MipsSim [15]. Demzufolge hat sich die Summe der Design-Entscheidungen gelohnt.

	MipsSim	dieser Emulator
Mittelwert	963 ms	243 ms
Standartabweichung	33 ms	26 ms

Tabelle 6: Benötigte Zeit zum Ausführen der ersten 1'000'000 Instruktionen von Topsy. Es wurden je 10 Messungen gemacht.

Effizienz von CCpuEnhanced::instruction2Op und von CAlu In diesem Punkt wird die ..... verifiziert, dass die Templatemethode CCpuEnhanced::instruction2Op effizient kompiliert werden kann.

Dies wird folgendermassen erreicht: Die Anzahl Assembler-Instruktionen, die der Kompiler für eine Instanz der Templatemethode generiert, werden verglichen mit der Anzahl Instruktionen die er generiert für eine direkte Implementation der Methode. Beispiel: In CCpuMipsR3051::initInstructionTable wird für die "addi"-Instruktion die instruction2Op Templatemethode instanziert. In CCpuMipsR3051::instructionAddi wird dieselbe Funktionalität direkt implementiert. InstructionAddi wird nur für diesen Test verwendet. Nach dem Kompilieren wird das vom Compiler erstellte Assemblerfile geprüft, und die Anzahl generierter Assembler-Instruktionen werden verglichen. Da Methoden der CAlu-Klasse aufgerufen werden, gilt das Resultat dieses Tests auch für diese Klasse.

Als Beispiel wurden die Instruktionen "addi" und "add" gewählt. "Add" deshalb, weil es eine "normale" einfache Instruktion repräsentiert. "Addi" auch deshalb, weil darin die Flags der Cpu gesetzt werden müssen. Somit kann die Effizienz des Mechanismus gemessen werden, der die Flags der Cpu setzt.

Der Emulator wird mit der Version MipsRetail kompiliert. Da in dieser Version alle Debuggfeatures ausgeschaltet sind, werden an dieser Stelle die Optionen nicht angezeigt, da sie irrelevant sind. Damit Assemblerfiles generiert werden, muss mit "make assfiles" kompiliert werden.

Tabelle 7 zeigt die Resultate. Es ist evident, dass trotz der hohen Komplexität des Methoden-Aufrufbaumes mit instruction2Op als Wurzel, der Kompiler instruction2Op sehr effizient kompilieren kann. Sie sind nur unwesentlich grösser als die direkte Implementation [16]

Die wiederverwendbaren Komponenten CALU und instruction2Op sind demnach nicht langsamer als eine direkte Implementation der Methoden. Wie zu vermutet war.

1 MipsR3051 Instruktion	Template	direkte Implementation
addi	19	18
addiu	45	40

Tabelle 7: Anzahl Assemblerinstruktionen, die vom Kompiler erzeugt werden.

Es wurden nicht gemessen, wie gross der sogenannte Slowdownfaktor ist. Der Slowdownfaktor bezeichnet das Verhältnis der Zeit, die ein Programm braucht, um auf Architektur X ausgeführt

[14] Ca 30% der Messungen für diesen Emulator hatten negative Werte. Wegen der beschränkter Zeit wurde nicht auf diess Phänomen eingegangen. Diese Werte wurden einfach ignoriert.

[15] In der Presentation wurde behaupted, dieser Emulator sei sieben mal schneller als MipsSim. Diese Fehlerhafte Angabe ist darauf zurückzuführen, dass während der Messungen für MipsSim im Hintergrund ein weiterer Cpulastiger Prozess ausgeführt wurde. Dies veränderte die Resultate zu Ungusten von MipsSim. Bei den hier angeführten Tests war kein anderer Cpulastiger Prozess im Hintergrund.

[16] Es ist anzumerken, dass die Anzahl der generierten Assembler-Instruktionen noch kein defi nitives Mass für Effi zienz einer Methode darstellt.

zu werden, gegenüber der Zeit, die dasselbe Programm braucht, wenn es durch den Emulator auf Architektur X ausgeführt wird. Dabei emuliert der Emulator die Architektur X.
Es wurde noch keine Vergleiche mit Emulatoren - ausser mit MipSim - aufgestellt. Kandidaten dafür wären SimOs oder vmips.

12 Fazit

In diesem Kapitel werden als erstes die wichtigsten erreichten Eigenschaften zusammengefasst. Als zweites wird in einem Ausblick vermittelt, inwiefern diese Arbeit erweitert und verbessert werden kann.

12.1 Erreichtes

Im Folgenden werden die wichtigsten erreichten Eigenschaften aufgelistet. Die aufgestellten Behauptungen wurden im Kapitel 11 bewiesen.

Topsy lauffähig Das wichtigste Ziel wurde erreicht: Topsy kann auf der emulierten MipsR3051 - MipsBoard Architektur ausgeführt werden. Es treten keine Fehler auf. Kapitel 11 geht auf zwei frühere Fehler ein, die nicht mehr auftreten, aber auch nie explizit behoben wurden.

Modular Das API ist einfach, klein und orientiert sich sehr nahe an der realen Hardware. Somit wurde eine gute Modularität erreicht. Die Eigenschaft der Modularität wurde nicht bewiesen. Dies wird sich zeigen, wenn einige Implementationen für diesen Emulator implementiert worden sind.

Debugging Zum Debuggen der Implementation einer Architektur wird hilfreiche Unterstützung angeboten:

- Der Status der emulierten Architektur kann nach jedem Clockzyklus mit unterschiedlichem Informationsgehalt ausgegeben werden.
- Mit sogenannten Wait Flags kann der Emulator unter konfigurierbaren Bedingungen angehalten werden. Somit werden fehlerhafte Clockzyklen schnell gefunden.
-

Diese Eigenschaften können auch dazu genutzt werden, das emulierte Programm zu debuggen. Zum Beispiel kann der Emulator angehalten werden, wenn eine Instruktion des emulierten Programmes undefined/unpredictable Behaviour auslöst. Die wertvollste Hilfe zum Debuggen des emulierten Programmes wäre ein Interface zum GDB. Dies konnte wegen der beschränkten Zeit nicht implementiert werden.

Fast Die emulierte MipsR3051 - MipsBoard Architektur führt Topsy viermal schneller aus als der MipsSim Emulator. Die Verwendung von Inline- Methoden und gleichzeitige Vermeidung von virtuellen Methoden hat sich gelohnt.

12.2 Ausblick

Im Folgenden wird vermittelt, inwiefern diese Arbeit ergänzt und verbessert werden kann.

- Ein Interface zum GDB der GNU Tools implementieren. Dies wäre sehr hilfreich für das Debuggen der emulierten Software.
- Der Emulator sollte durch einen Remote-Process gesteuert werden können.
- Der Emulator, und die emulierte Architektur sollte während der Ausführung des emulierten Programmes neu konfiguriert werden können. Zum Beispiel könnte der Dump Level von tief auf hoch umgeschaltet werden, sobald das emulierte Programm eine kritische Phase erreicht. Aus demselben Grund könnte der Level, wie akurat eine Implementierte Architektur die Hardware emuliert, während der Ausführung geändert werden. Ich schlage vor,die Basisklassen CCpu und CIoSystem um die eine Methode ähnlich zu der Init- Methode zu erweitern. Der Unterschied wäre, dass die neue Methode mehrmals aufgerufen werden kann, init hingegen nur einmal beim Aufstarten des Emulators.

- Hinsichtlich der Performance der MipsR3051-MipsBoard Architektur, können die folgenden Punkte angegangen werden. Auf Grund des Fetchen des Instruktionswortes wird für jede ausgeführte Instruktion auf das IO-System zugegriffen. Das beinhaltet die Uebersetzung der virtuellen in die physikalische Adresse. Optimierungen in diesen zwei Punkten lohnen sich deshalb besonders. Die vorgeschlagenen Weiterentwicklungen sind in der Reihenfolge der zu erwartenden Performance-Gewinne aufgelistet.
  - In CDeviceRam soll das Swappen der Endianness effizienter implementieren. Zum Beispiel, indem alle Bytes einzeln gelesen und richtig zusammengesetzt werden.
  - An Stelle der linearen Suche, die TLB mit Hilfe einer Hashtable durchsuchen
  - Devices melden dem IOSystem, wann (gemessen in Clockzyklen) sie vom IOSystem den Clockimpuls bekommen möchten. Somit müsste das IOSystem nicht nach jedem Clockzyklus (oder in einem längeren Intervall) allen Devices den Clockimpuls geben. In den allermeisten Fällen sind die Devices gar nicht daran interessiert, den Clock zu bekommen. Dies sind sie in der Regel nur, wenn sie von der CPU einen Auftrag bekommen. Dann müssen sie nach einer bestimmten Zeit den Clockimpuls bekommen, um einen Interrupt auslösen zu können.
- Neue Devices implementieren. Das MimpsBoard IO-System braucht noch eine Netzwerkkarte.

A Aufgabenstellung

Institut für
Technische Informatik und
Kommunikationsnetze

Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich
Ecole polytechnique fédérale de Zurich
Politecnico federale di Zurigo

Sommersemester 2003

Diplomarbeit

für

Florian Kaufmann

Betreuer: Lukas Ruf
Stellvertreter: Matthias Bossardt

Ausgabe: 10.03.2003
Abgabe: 10.07.2003

Design und Implementation eines modularen Emulators für Topsy

1 Einführung

Die Entwicklung von Betriebssystemen erfordert Entwicklungsumgebungen, welche es gestatten, den Zustand des gesamten Systems zu jedem beliebigen Zeitpunkt in der Betriebssystem-Ausführung zu betrachten und nötigenfalls zu verändern. Ansätze, dieses Problem zu lösen, sind durch Plattformen mit hardware-seitiger Unterstützung für die Fehlersuche, wie z.B. dem JTAG [5] Interface, oder durch Software-Emulatoren gegeben, die gesamte Computer-System nachbilden. Bekannte Beispiele der letztgenannten Gruppe sind vmware [14], bochs [1] und dessen Abkömmling Plex86 [9].
Am Institut für Technische Informatik und Kommunikationsnetze (TIK) der ETH Zürich (ETHZ) wurde im Rahmen der Arbeit zu Topsy v1 [2] von George Fankhauser ein Software Emulator für die MIPS-Plattform entwickelt. Durch eine Erweiterung während der Arbeit zur ersten Implementierung eines IP [10] Stacks [11] wurde der Emulator um Netzwerkfähigkeiten bereichert.
Der vorliegende Emulator wurde in Java [8] implementiert. Er unterstützt die Möglichkeiten des Debuggings nur sehr rudimentär und ist auf die MIPS-Plattform beschränkt. Obwohl Java Portabilität und Plattform-Unabhängigkeit nachgesagt wird, ist Java sehr von der eingesetzten Version der Virtual Machine abhängig. Zudem stellten sich beim Einsatz des Emulators, verschiedene Merkwürdigkeiten im Zusammenhang mit der Netzwerk-Emulation heraus.

2 Aufgaben: Design und Implementation eines modularen Emulators für Topsy

Im Rahmen dieser Diplomarbeit soll ein modularer Emulator entwickelt werden, der es gestattet, die für die Verarbeitung des Binär-Codes eingesetzten Application Binary Interfaces (ABIs) resp. System Binary Interfaces (SBIs) auf einfache Weise für verschiedene Plattformen zur Verfügung zu stellen. In einer ersten Version soll der Emulator die MIPS-Plattform unterstützen.

Der zu entwickelnde, Software-basierte Emulator soll in der Lage sein, Zusatzkomponenten wie Bussysteme, mehrere Netzwerkkarten oder Harddisk-Interfaces zur Basis des Emulators hinzuzufügen.
Der Emulator soll die Möglichkeit bieten, die Entwicklung von Betriebssystemen (Topsy) auf geeignete Weise zu unterstützen. Wenn möglich, soll ein Interface zum weitverbreiteten gdb [3] der GNU Tools entwickelt werden.
Anhand theoretischer Ueberlegungen für eine StrongARM [12] Plattform soll die Einsatzfähigkeit des Emulators für verschiedene Hardware-Plattformen gezeigt werden.

3 Vorgehen

- Richten Sie sich eine Entwicklungsumgebung (GNU Tools) unter Linux ein.
- Machen Sie sich vertraut mit den Unterlagen (Dokumentation und Source Code) zu MIPS, StrongARM, dem vorliegenden MIPS-Emulator und den eingesetzten Netzwerk-Interface Karten [7, 6].
- Erstellen Sie einen Zeitplan, in welchem Sie die von Ihnen zu erreichenden Meilensteine Ihrer Arbeit identifi zieren.
- Entwickeln Sie eine Architektur für den zu erstellenden Emulator.
- Defi nieren Sie die Interfaces.
- Implementieren Sie Ihre Architektur.
- Verifi zieren, evaluieren und demonstrieren Sie das Erreichte durch eine Beispielapplikation.
- Dokumentieren Sie die Resultate ausführlich.

Auf eine klare und ausführliche Dokumentation wird besonders Wert gelegt. Es wird empfohlen, diese laufend nachzuführen und insbesondere die entwickelten Konzepte und untersuchten Varianten vor dem defi nitiven Variantenentscheid ausführlich schriftlich festzuhalten.

4 Organisatorische Hinweise

- Am Ende der zweiten Woche ist ein Zeitplan für den Ablauf der Arbeit vorzulegen und mit dem Betreuer abzustimmen.
- Mit dem Betreuer sind regelmässige, zumindest wöchentliche Sitzungen zu vereinbaren. In diesen Sitzungen sollen die Studenten mündlich über den Fortgang der Arbeit und die Einhaltung des Zeitplanes berichten und anstehende Probleme diskutieren.
- Am Ende des ersten Monates muss eine Vorabversion des Inhaltsverzeichnis zur Dokumentation dem Betreuer abgegeben und mit diesem besprochen werden.
- Nach der Hälfte der Arbeitsdauer soll ein kurzer mündlicher Zwischenbericht abgegeben werden, der über den Stand der Arbeit Auskunft gibt. Dieser Zwischenbericht besteht aus einer viertelstündigen, mündlichen Darlegung der bisherigen Schritte und des weiteren Vorgehens gegenüber Professor Plattner.
- Am Schluss der Arbeit muss eine Präsentation von **20 Minuten** im Fachgruppen- oder Institutsrahmen gegeben werden. Anschliessend an die Schlusspräsentation soll die Arbeit Interessierten praktisch vorgeführt werden.
- Die Arbeit muss regelmässig auf dem CVS-Server cvs.topsy.net gesichert werden. Es ist darauf zu achten, dass die **richtige CVS-Branch** verwendet wird.
- Der Topsy [13] Coding Style muss eingehalten werden.
- Bereits vorhandene Software kann übernommen und gegebenenfalls angepasst werden.
- Die Dokumentation ist mit dem Satzsystem LaTeX zu erstellen.

- Es ist ein mit Bindespiralen gebundener Schlussbericht (am TIK vorhanden) über die geleisteten Arbeit abzuliefern (4 Exemplare). Dieser Bericht besteht aus einer Zusammenfassung , einer Einleitung, einer Analyse von verwandten und verwendeten Arbeiten, sowie einer vollständigen Beschreibung der Konfi guration von den eingesetzten Programmen. Der Bericht ist in Deutsch oder Englisch zu halten. Die Zusammenfassung muss in Deutsch und Englisch verfasst werden.
- Die Arbeit muss auf CDROM archiviert abgegeben werden. Stellen Sie sicher, dass alle Programme sowie die Dokumentation sowohl in der lauffähigen, resp. druckbaren Version als auch im Quellformat vorhanden, lesbar und verwendbar sind.
  Mit Hilfe der abgegebenen Dokumentation muss der entwickelte Code zu einem ausführbaren Programm erneut übersetzt und eingesetzt werden können.
- Diese Arbeit steht unter der GNU General Public License (GNU GPL) [4].
- Diese Arbeit wird als Diplomarbeit an der ETH Zürich durchgeführt. Es gelten die Bestimmungen hinsichtlich Kopier- und Verwertungsrechte der ETH Zürich.

Literatur

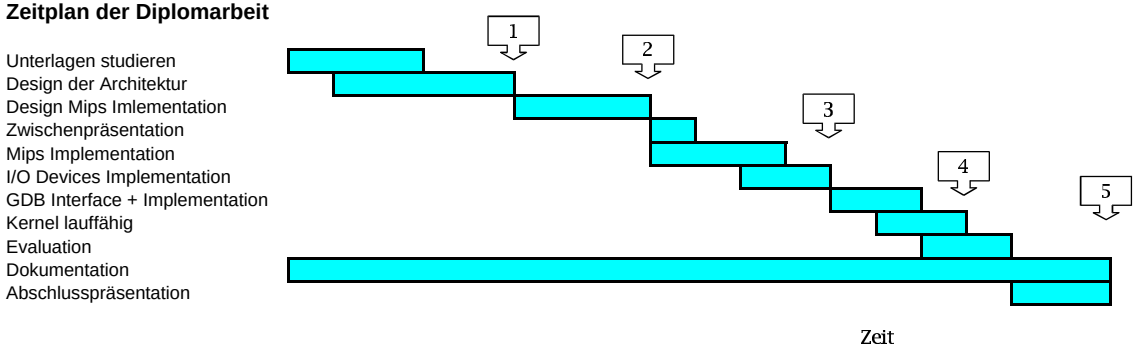
[1] Bochs. https://sourceforge.net/projects/bochs, 1998-2003.
[2] G. Fankhauser, C. Conrad, E. Zitzler, and B. Plattner. *Topsy – A Teachable Operating System*. TIK, ETH Zurich, 1997.
[3] GDB, the GNU symbolic debugger. http://www.gnu.org, 2001.
[4] GNU General Public License v2. http://www.gnu.org/copyleft/gpl.html, June 1991.
[5] IEEE-1149. JTAG. http://grouper.ieee.org/groups/1149, 1997-2003.
[6] AMD Inc. *Am79C970A PCnet PCI II Single-Chip Full-Duplex Ethernet Controller for PCI Local Bus, publication# 19436*, 1999.
[7] AMD Inc. *Am79C972 PCnet-FAST+ Enhanced 10/100 Mbps PCI Ethernet Controller with OnNow Support, publication# 21485*, 1999.
[8] JAVA. http://www.java.org.
[9] plex86. https://sourceforge.net/projects/plex86, 2001-2003.
[10] J. Postel. *RFC–0791: Internet Protocol*, September 1981.
[11] D. Schweikert. *A lightweight and high-performance TCP/IP stack for Topsy*. TIK, ETH Zurich, 1998.
[12] David Seal. *ARM Architecture Reference Manual*. Addison-Wesley, 2 edition, 2000.
[13] The Topsy Core Team. Topsy coding style. http://www.topsy.net/Standards, 2002.
[14] vmWare. http://www.vmware.com, 1998-2003.

Zürich, den 12. März 2003

B Zeitplan

Sheet1

Zeitplan der Diplomarbeit



Milestones

1	Design der Architektur abgeschlossen
2	Design der Mips Archtitektur abgeschlossen
3	Alle Implementationen bis auf gdb Interface abgeschlossen.
4	Topsy lauffähig, inklusive remote gdb interface.
5	Abschluss der Arbeit.

C Dateien

```
emulator.doxygen         Konfiguriert Doxygen ( Code-documenting tool )
API                      Rootverzeichnis der API generiert von Doxygen

TopsyHelp                Snipsets von Topsy, die auf Hardware zugreifen
mipssim_errors           Gefundene Fehler in MipsSim
todo2                    Notizen, was am Emulator verbessert werden könnte

emulator.kdevprj         Für Kdevelop ( Entwicklunsumgebung )
emulator.kdevses         Für Kdevelop ( Entwicklunsumgebung )
Makefile                 Makefile
Makefile.user            Makefile, das abgeändert werden darf
kernel.srec              Topsy Kernelimage


Emulator                                    Rootverzeichnis der Sourcedateien
|-- Arch                                    Sammlung von verschiedenen Architekturen
|   |-- CpuIoInterfaces                     Sammlung von Cpu - IoSystem Interfaces
|   |   |-- CpuIoInterfaceMipsR3051Mips.cpp Interface Cpu MipsR3051 <-> IoSystem MipsBoard
|   |   `-- CpuIoInterfaceMipsR3051Mips.h
|   |-- CpuMipsR3051                        Implementation der MipsR301 Cpu Architektur
|   |   |-- AluMipsR3051.h                  Alu
|   |   |-- CopMipsR3051Cp0.cpp             System Koprozessor
|   |   |-- CopMipsR3051Cp0.h
|   |   |-- CpuMipsR3051.cpp                Hauptprozessor
|   |   |-- CpuMipsR3051.h
|   |   |-- CpuMipsR3051.s
|   |   |-- InstructionMipsR3051.cpp        Instruktion für Hauptprozessor
|   |   |-- InstructionMipsR3051.h
|   |   |-- InstructionMipsR3051Cp0.cpp     Instruktion für System Koprozessor
|   |   `-- InstructionMipsR3051Cp0.h
|   `-- IoSystemMips                        Implemenation des MipsBoard IO-System
|       |-- DeviceInterfaceMips.cpp         Deviceinterface zum MipsBoard IO-System
|       |-- DeviceInterfaceMips.h
|       |-- DeviceMipsEthertapAm79C970A.h   Device: Netzwerkkarte
|       |-- DeviceMipsFpga.h
|       |-- DeviceMipsPit82C54.h            Device: Programable interrupt timer (clock)
|       |-- DeviceMipsRom.cpp               Device: Rom
|       |-- DeviceMipsRom.h
|       |-- DeviceMipsUartSnc2681.h         Device: Uart
|       |-- IoSystemMips.cpp
|       `-- IoSystemMips.h

|-- Configs                                 Sammlung von Konfiguration
|   |-- ConfigMipsDebug.h                   Architektur = MipsR3051-MipsBoard , Debug Features on
|   |-- ConfigMipsRetail.h                  Architektur = MipsR3051-MipsBoard , High Performance
|   `-- ConfigTemplate.h                    Vorlage eines Konfigurationsfiles

|-- Cpu                                     Basisklasse + wiederverwendbare Komponenten für Cpu
|   |-- Alu.h
|   |-- Cpu.cpp
|   |-- Cpu.h
|   |-- CpuEnhanced.h
|   |-- CpuEnhanced.hxx
|   |-- FlagTreat.h
|   |-- Instruction.cpp
|   `-- Instruction.h

|-- Devices                                 Basisklassen + Sammlung von Devicekernen
|   |-- Device.cpp                          Basisklasse
|   |-- Device.h
|   |-- DeviceEthertapAm79C970A.cpp         Netzwerkkarte
|   |-- DeviceEthertapAm79C970A.h
|   |-- DeviceFpga.cpp
|   |-- DeviceFpga.h
|   |-- DeviceInterface.cpp                 Basisklasse
|   |-- DeviceInterface.h
|   |-- DevicePit82C54.cpp                  Programable Interrupt Timer (clock)
|   |-- DevicePit82C54.h
|   |-- DeviceRam.cpp                       Ram
|   |-- DeviceRam.h
```

```
|   |-- DeviceRom.cpp                   Rom
|   |-- DeviceRom.h
|   |-- DeviceUartSnc2681.cpp           Uart
|   `-- DeviceUartSnc2681.h

|-- Emulator
|   |-- Clock.h
|   |-- Emulator.cpp
|   `-- Emulator.h

|-- Host                                Sammlung von Host-Headerfiles und Host-Makefiles
|   |-- Host.h                          gemeinsam verwendet
|   |-- Ia32
|   |   |-- HostIa32.h
|   |   `-- Makefile.host
|   `-- template                        Vorlage für neuen Host
|       |-- HostTemplate.h
|       `-- Makefile.host

|-- IoSystem
|   |-- CodeLoader.cpp
|   |-- CodeLoader.h
|   |-- CpuIoInterface.cpp              Basisklasse
|   |-- CpuIoInterface.h
|   |-- IoSystem.cpp                    Basisklasse
|   `-- IoSystem.h

|-- Endianness.h                        implementiert swapEndianness
|-- Error.cpp
|-- Error.h
|-- Globals.cpp                         definition aller globalen Variablen
|-- Main.cpp                            implementation der main methode
`-- Signal.h                            declariert ESignalState
```

Literatur

[1] AMD Datasheets, *AM79C970A PCnet/PCI II Single/Chip Full-Duplex Ethernet Controller for PCI Local Bus Product*;
http://www.amd.com, 1999.

[2] George Fankhauser, Christian Conrad, Eckart Zitzler, Bernhard Plattner., *Topsy - A Teachable Operating System, Version 1.1*;
Computer Engineering and Networks Laboratory, ETH Zuerich, 1996, 2000.

[3] Integrated Device Technology Inc., *The IDTR3051, R3052 RISController Hardware User's Manual*;
http://www.idt.com, 1992, 1994.

[4] MIPS Technologies, *MIPS32 Architecture for Programmers, Volume I-III*;
http://www.mips.com, 2002.

[5] David A. Patterson, John L. Hennessy, *Computer Organization and Design, Second Edition*;
Morgan Kaufmann Publishers, Inc.,1997.

[6] Martin Schader, Stefan Kuhlins, *Programming in C++, Fifth Edition*;
Springer, 1998.

[7] //David Seal, *Architecture Reference Manual, Second Edition*;
ARM Limited, 1996-2000.

[8] W. Richard Stevens, *UNIX Network Programming, Second Edition*;
Prentice-Hall, Inc., 1998.

[9] Matthias Uphoff, *VGA- und Super-VGA-Programmierung, Second Edition*;
Addison-Wesley GmbH, 1994.

[10] Bertram Wohak, Reinhold Maurus, *80x86/Pentium Assembler, First Edition*;
IWT Verlag GmbH, 1995.