Projektgruppe 433 PoCiLet:

Pocket Computer Interactive Learning Object

Endbericht

Betreuer:

Birgit Sirocic, Jens Wagner

Universität Dortmund
Fachbereich Informatik
Lehrstuhl Informatik XII
Prof. Dr. Peter Marwedel

Inhaltsverzeichnis

1 Einleitung **5**

1.1 PG-PoCiLet 5
1.1.1 Thema 5
1.1.2 Zeitraum 6
1.1.3 Umfang 6
1.1.4 Veranstalter 6
1.1.5 Internetadresse 6
1.2 Teilnehmervorrausetzungen 6
1.3 Teilnehmer 7
1.4 Nutzungsvereinbarung 7
1.5 Minimalziel 8

2 Realisierung des Projekts **11**

2.1 Befehlssatz und Operationscode 12
2.1.1 Struktur des Befehlscodes 12
2.1.2 Auscodierung des Operationscodes 14
2.1.3 Befehlscode 14
2.2 Das PoCiLet-Board - Vom Schaltplan zur Platine 17
2.2.1 Vorüberlegungen und erste Ideen 17
2.2.2 Auswahl der Bauteile und deren Funktionen 20
2.2.3 Auswahl der Entwicklungtools 24
2.2.4 Entwicklung des Schaltplans 28
2.2.5 Programmierung der CPLDs 29
2.2.6 Verifikation des Schaltplans durch Simulation 34
2.2.7 Vom Prototypen zum Board 38
2.2.8 Fertigstellung der Platine 44
2.3 Der PoCiLet-Simulationskern 46
2.3.1 Einleitung 46
2.3.2 Konzept 47
2.3.3 Simulationseinstieg an einem Flipflop-Beispiel 50

2.3.4 Entwurf . . . 52
2.4 Die grafische Benutzerschnittstelle . . . 59
2.4.1 Konzept . . . 60
2.4.2 Prototyp . . . 65
2.4.3 Entwurf . . . 67
2.4.4 Realisierung . . . 73

3 Evaluation 84
3.1 Methodik . . . 84
3.1.1 Evaluationsphasen . . . 84
3.2 Durchführung . . . 85
3.3 Ergebnisse . . . 86
3.3.1 Fragen über allgemeine Vorkenntnisse mit dem Rechner . 86
3.3.2 Fragen zu unserer PoCiLet-Software: . . . 91
3.3.3 Fragen zu unserer PoCiLet-Hardware? . . . 95
3.4 Bewertung der Ergebnisse . . . 99
3.5 Die Teststrategie . . . 101

4 Fazit & Ausblick 106
4.1 Persönliches Fazit . . . 106
4.2 Fachliches Fazit . . . 107

5 Benutzerhandbuch 109
5.1 Installationsanleitung . . . 109
5.2 Übersicht der GUI Komponenten . . . 111
5.3 Der PoCiLet Programmeditor . . . 111
5.3.1 Das Erstellen eines Programms . . . 111
5.3.2 Ein Programm editieren . . . 112
5.4 Erste Schritte im PoCiLet Simulator . . . 112
5.4.1 Starten/Stoppen des Simulators . . . 112
5.4.2 Teilschritte eines Befehls ansehen . . . 113
5.4.3 Rückwärtsschritte im Simulator . . . 113
5.4.4 Einen Zustand speichern und laden . . . 114
5.5 Die vier Ebenen von PoCiLet . . . 114
5.5.1 Die Schaltkreisebene . . . 115
5.5.2 Die Register-Transfer-Ebene . . . 115
5.5.3 Die Layout-Ebene . . . 116
5.5.4 Die 3D-Ebene . . . 116
5.6 Die drei Ansichtsmodi . . . 116
5.6.1 Vollbild . . . 117
5.6.2 Vierfach . . . 117

5.6.3 Integriert 118
5.7 Die PoCiLet-Hardware 118
5.7.1 Anschluss der Hardware 118
5.7.2 Anmeldung der Hardware 119
5.7.3 Benutzung der Hardware 119
5.8 Die PoCiLet Simulator GUI beenden 121

6 Anhang **122**
6.1 Zustandsautomat des Steuerwerks Seite 31 122

Kapitel 1

Einleitung

1.1 PG-PoCiLet

1.1.1 Thema

Ziel der Projektgruppe war es, ein Lehr-/Lernobjekt zu entwickeln, das den Aufbau und die Funktionsweise eines einfachen Rechnersystems auf verschiedenen Entwicklungsstufen, wie Registertransferebene, Schaltkreisebene und Layoutebene, gleichzeitig und nebeneinander visualisiert.
Durch das Nebeneinanderstellen der verschiedenen Entwicklungsstufen (Sichten), in denen ein Rechnersystem beschreibbar ist, wird es möglich, die Brücke zwischen dem im Alltag benutzbaren System und den technischen Grundlagen zu schlagen.

Der Einstieg in das Lehr-/Lernobjekt soll über eine interaktive 3D-Visualisierung des einfachen Rechnersystems erfolgen. In dieser 3D-Visualisierung soll das Rechnersystem möglichst wirklichkeitsgetreu dargestellt werden und interaktiv bedienbar sein. Dadurch wird in erster Linie der direkte Bezug zum tatsächlichen Rechnersystem (Gerät) hergestellt. Über ein virtuelles Tastenfeld soll dem Benutzer die Möglichkeit gegeben werden, mit dem virtuellen Rechnersystem zu spielen. Die Simulation und Visualisierung des dynamischen Verhaltens des Rechnersystems in den verschiedenen Sichten zeigt dann, was im Inneren abläuft.
Diese Art und Weise der interaktiven Visualisierung der Zusammenhänge gibt es bisher noch nicht. Die Veranschaulichung der Zusammenhänge, als auch der Einstieg in ein Thema über ein alltagsnahes Beispielszenario, hier die virtuelle und wirklichkeitsnahe Darstellung und Bedienung des Gerätes, kann sich auf nicht technisch interessierte Studierende oder andere Lerntypgruppen motivierend auswirken.

Ein weiterer Vorteil der 3D-Visualisierung innerhalb einer wirklichkeitsgetreuen Beispielszene liegt darin, dass man zunächst die Realität nachstellt, dann aber die Dinge zeigen, bzw. visualisieren kann, die man mit dem menschlichen Auge so nicht sehen kann. So werden zum Beispiel in der 3D-Einstiegsszene die Signale auf den Leiterbahnen visualisiert. Das Vorhandensein von Signalen auf den Leiterbahnen kann in der Realität nur punktuell über Messgeräte erfolgen.

1.1.2 Zeitraum

Zeitraum: Wintersemester 2003/2004 und Sommersemester 2004

1.1.3 Umfang

Umfang: Jeweils 8 Semesterwochenstunden

1.1.4 Veranstalter

Dipl.-Inf. Birgit Sirocic
Otto-Hahn-Str. 16, Raum E04
Tel.: 0231 / 755 6329
EMail: Birgit.Sirocic@udo.edu

Dipl.-Inf (FH), Ing. Jens Wagner
Otto-Hahn-Str. 16, Raum E20
Tel.: 0231 / 755 6263
EMail: Jens.Wagner@udo.edu

1.1.5 Internetadresse

Unsere PG-Webseite:
http://ls12-www.cs.uni-dortmund.de/pocilet

1.2 Teilnehmervorrausetzungen

Kenntnisse in mindestens einem der folgenden Gebiete sind notwendig:
Rechnerarchitektur, Prozeßrechnertechnik / Eingebettete Systeme, Graphische Systeme, Softwareergonomie, Didaktik der Informatik.
Wünschenswert sind folgende Kenntnisse:
Fortgeschrittene Kenntnisse der E-Technik, evtl. EPra.

1.3 Teilnehmer

Nihal Aksoy
Mandana Asfa
Monika Enns
Martina Friese
Désirée Kraus
Christiane Lacmago
Frank Benneker
Sven Berling
Holger Bihr
Daniel Fengler
Benjamin Lachenicht
Kai-Uwe Reichert

1.4 Nutzungsvereinbarung

Die Software ist unter der Zwei-Klausel-BSD-Lizenz gestellt. Diese erlaubt die freie Verbreitung der Software in binärer Form, die Quelltexte bleiben in dem Besitz der Projektteilnehmer. Es ist jedoch vorgesehen, besonders in Bereichen der Forschung und Lehre, die Quelltexte zugänglich zu machen, unter der Voraussetzung sie nicht kommerziell zu nutzen.

Copyright 2004 - PG 433 PoCiLet
Univerity of Dortmund, NRW, Germany All rights reserved.
Email: pocilet@ls12.cs.uni-dortmund.de

Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

1. Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.

2. Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.

THIS SOFTWARE IS PROVIDED BY THE REGENTS AND CONTRIBUTORS „AS IS“ AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE REGENTS OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT

LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

1.5 Minimalziel

Das gewählte Thema hat einen thematischen Anschluß zu Forschungsarbeiten am Lehrstuhl Informatik XII der Universität Dortmund. Insbesondere die thematische Nähe zum Simba Projekt ermöglicht es den Teilnehmern, Einblick in aktuelle Arbeiten auf dem Gebiet der Wissensvermittlung im Fachgebiet Technische Informatik zu erhalten. Die Arbeit am Projekt kann weiterhin zur Wissensvertiefung auf dem Gebiet der Rechnerarchitektur genutzt werden. Ein Großteil der am Lehrstuhl verfassten Diplomarbeiten, entsteht in thematischer Nähe zu den ehemaligen Projektgruppen. Das Projekt bietet den Teilnehmern somit die Möglichkeit, sich für eine Reihe von möglichen Diplomarbeiten am Lehrstuhl zu qualifizieren.
Im Laufe der Projektgruppe soll ein System entstehen, welches den Aufbau eines einfachen Rechensystems in vier verschiedenen Sichten visualisiert:

- Schematische Sicht auf Registertransferebene
- Schematische Sicht auf Schaltkreisebene
- Layout Sicht auf Schaltkreisebene
- 3D Modell des Gerätes / körperlich vorhanden Hardware

Wunschsystem

Ein minimales Rechnersystem wird komplett in den vorgeschlagenen vier Schichten realisiert. Denkbar wäre z.B. ein System, bestehend aus einem kleinen mikroprogrammierbaren Rechenwerk, Speicher und Ein-/Ausgabeeinheit. Der vereinfachte Aufbau soll keinen Anspruch auf die Darstellung eines aktuellen Systems erheben. Es ist vielmehr ein einfacher *Pocketcomputer* oder auch Taschenrechner als Explorationsbasis angedacht. Damit wird das Ziel eines einfachen Lernsystems nicht aus dem Auge verloren. Die geringe Komplexität der verwendeten Komponenten trägt zum leichten Verständnis jedes einzelnen verwendeten Bausteines bei. Die Kleinheit des Gesamtsystems soll es dem Lernenden ermöglichen, die Funktionsweise in kurzer Zeit vollständig zu erfassen. Das fertige System soll die

didaktische Lücke zwischen abstrakter Beschreibung eines Systems und Realisierung in Hardware schliessen. Die haptische Komponente ist mit dem Ziel hinzugefügt, dem Lernenden das Erlebnis zu geben, dass die Anwendung der erarbeiteten Prinzipien wirklich zu einem funktionierenden Rechner führt. Das abstrakte Modell wird lebendig.

Die Simulation des Verhaltens soll in allen vier Sichten parallel erfolgen. Denkbar wäre eine Teilung einer Bildansicht in vier Sichten. Eine schrittweise Abarbeitung des Modells soll ein einfaches Verstehen erleichtern. Einzelne Simulationsschritte sollten sich zurücknehmen lassen. Die Simulation sollte mit einem Java Komponentenmodell erfolgen. Java wird wegen der Portabilität zwischen möglichst vielen Plattformen vorgeschlagen. Am Lehrstuhl existieren bereits Erfahrungen mit solchen Systemen.

Das gesamte Design der Hardware soll von vornherein auf einfache Nachbaubarkeit ausgelegt sein. Der aktuell in der Literatur veröffentliche Stand soll als Basis in die Arbeit eingehen. Die Ergebnisse sollen unbedingt für Dritte nachvollziehbar und nutzbar sein.

Realisierung

Die Realisierung unterteilt sich in drei Bereiche:

- Simulationskern

  Die Realisierung des Simulationskerns umfaßt den Entwurf von Komponenten des Simulationsmodells und der Testfälle. Dabei soll der Aufbau des Simulationskerns den strukturellen Aufbau des Rechners widerspiegeln. Die Arbeiten verlangen ein relativ gutes Verständnis der Rechnerkomponenten auf Registertransferebene. Die Möglichkeiten des objektorientierten Entwurfs sollen ausgenutzt werden, um eine modellhafte Abbildung realer Architekturkomponenten zu erhalten. Die Komponenten sollen ähnlich eines Baukastensystems zu einem Komplettsystem zusammengesetzt werden können.

- Frontend

  Das Fronted beinhaltet alle Komponenten, die zur Interaktion mit dem Anwender nötig sind. Diese Teilaufgabe erfordert Einarbeitung in die ergonomischen, künstlerischen und teilweise auch didaktischen Aspekte einer solchen Schnittstelle. Das Fronted

soll die gewünschten Informationen kompakt übermitteln und intuitive Bedienung zulassen. Zum besseren Erlernen der nötigen Grundlagen sollen weiterführende Elemente schrittweise eingeblendet werden. Die Symbolik soll sich an der Literatur üblichen Normen orientieren. Eine dreidimensionale Darstellung soll einen Eindruck der Realisierung in einem Gerät vermitteln.

- Applikationsentwicklung

  Diese Teilaufgabe beschäftigt sich mit der Realisierung des darzustellenden Modells. Das Modell soll in Hardware und Software entwickelt werden. Eine Beschreibung aus Anwendersicht ist notwendig. Dieses könnte z.B. ein Tutorial mit Übungsaufgaben beinhalten. Die für diese Aufgabe nötigen Grundkenntnisse der Elektrotechnik können während des Projekts problemlos erlernt werden. Diese Teilgruppe wird auch die benötigte Hardware entwerfen und realisieren.

Das Ziel ist nunmehr die Visualisierung eines einfachen Rechnersystems in vier verschiedenen Sichten. Desweiteren soll das System die Bedienung eines solchen Rechnersystems veranschaulichen. Das derzeit jedem Rechnersystem zugrundeliegende Prinzip der Registermaschinen und ihre Programmierung mittels einer sequentiellen Liste von Befehlen, gilt auch für PoCiLet. Der Lernende erhält somit eine Einführung in der Assemblerpogrammierung.

Kapitel 2

Realisierung des Projekts

Nach der Einführung im vorherigen Kapitel und dem dort dargestellten Wunschziel, geht es nun in diesem Kapitel darum, wie der Pocketcomputer im Einzelnen realisiert wurde.

Am Anfang der Projektgruppe haben wir uns in drei Untergruppen aufgeteilt, um an unterschiedlichen Teilaufgaben parallel zu arbeiten. Die drei Gruppen haben folgende Teilaufgaben bearbeitet:
Die Umsetzung des Pocketcomputers in Hardware, die Entwicklung des Simulationskerns und die Entwicklung der graphische Benutzerschnittstelle.
Im weiteren Verlauf dieses Kapitels wird auf die Umsetzung dieser drei Teilaufgaben genauer eingegangen.

Im ersten Unterkapitel wird zunächst der Befehlssatz festgelegt, welcher im späteren Verlauf der Entwicklung in der Hardware, als auch in der Software verwendet wird. Im zweiten Unterkapitel wird geschildert, wie die Hardwaregruppe vom erstellten Schaltplan bis hin zur Platine ihre Ideen umgesetzt hat. Im dritten Unterkapitel wird erklärt, wie die Umsetzung dieses Schaltplans in Software durch die Realisierung eines Simulationskerns erfolgt.

Der Simulationskern simuliert die Abstraktionsschicht der realen Hardware, wie sie auf der Registertransferebene (kurz RT-Ebene) dargestellt ist. Die graphische Benutzerschnittstelle hingegen stellt die Simulation in den vier Ebenen (Schaltplan-, RT-, Layout- und 3D-Ebene) dar, welche weitere Interpretationen der RT-Ebene des Simulationskern sind. Sie wird im Unterkapitel 2.4 beschrieben.

2.1 Befehlssatz und Operationscode

Ein Befehlssatz stellt die Gesamtheit der unterschiedlichen Befehle dar, die von einem Mikroprozessor ausgeführt werden können.

> „Ein Befehl ist die kleinste bezüglich einer Programmiersprache nicht weiter zerlegbare Einheit und gibt einen Arbeitsschritt an. Befehle sind die Grundbestandteile eines Programms. Die Bezeichnung „Befehl“ wird meist nur bei maschinenorientierten Sprachen verwendet. Jeder Befehl beeinflußt den Zustand einer Rechenanlage, insbesondere die Register, den Speicher usw.“ [VC97]

Relativ am Anfang der Entwicklung haben wir den Befehlssatz für unseren Pocketcomputer festgelegt, welcher nicht zu kompliziert, jedoch den Minimalansprüchen gerecht ist, um den Lernenden auch ein realistisches Bild der Hardware zu vermitteln.
Um dies festlegen zu können, musste eine grobe Struktur der Hardware definiert werden. Von Bedeutung sind hier die Ressourcen, die für die Auskodierung benötigt werden:

- Realisiert werden 4 Register
- Das Offset für die Sprungadressen beträgt 4 Bit
- Die Wortbreite der Daten beträgt 8 Bit

Anhand dieser Daten haben wir die Wortbreite des Befehlssatzes auf eine 12 Bit breite Darstellung festgelegt. Insgesamt setzt sich der Befehlssatz aus 24 Befehlen (siehe Tabelle 2.1) zusammen, in denen verschiedene Befehlstypen enthalten sind. Hierzu zählen Transportbefehle, arithmetische und logische Befehle, sowie bedingte und unbedingte Sprungbefehle.

2.1.1 Struktur des Befehlscodes

Der strukturelle Aufbau des Befehlcodes besteht aus zwei Teilen. Die höherwertigen Bits bilden den Operationscode, dessen Binärwert eine Abbildung auf die einzelnen Befehle des Befehlssatzes liefert. Die übrigen niederwertigen Bits bilden den Datencode. Sie kodieren die Information auf welche Register ein Befehl operiert und enthalten eventuell weitere Konstanten, die ein bestimmter Befehlstyp zur Ausführung benötigt.

Die Bitbreite des Befehlscode	
Operationscode	Datencode

Vorgehensweise bei der Auscodierung an einem konkreten Beispiel:

Um in der Hardware eine *Addition* durchführen zu können, müssen bestimmte Bedingungen im Vorfeld erfüllt sein. Die internen Werte zweier beliebiger Register müssen mit den zu addierenden Werten initialisiert sein. Der Datencode kodiert nun die Addressierung von drei Registern. Die ersten Beiden referenzieren die Register, deren interne Werte addiert werden sollen, das letzte gibt das Register an, in welchem das Endergebnis der Addition hineingeschrieben wird.

Wie zuvor erwähnt, werden in der PoCiLet-Hardware vier Register dargestellt. Die Adressierung dieser erfolgt über 2 Bit.

Struktur der arithmetischen (teilweise logischen) Operationen:

Es werden drei Register, bei der Addition für die Quell- und Zieladressen der jeweiligen Werte, benötigt. Diese werden je über 2 Bit adressiert. Somit werden insgesamt 6 Bit für die Adressierung dieser Register benötigt ($3 \cdot 2$ Bit). Die übrigen 6 Bit bilden den Operationscode.
Eine allgemeine Struktur für arithmetische und logische Operationen sieht wie folgt aus:

Opcode	reg2	reg1	reg
6Bit	2Bit	2Bit	2Bit

Struktur der Vergleichsoperationen:

Ein Beispiel für eine logische Vergleichsoperation ist der equal- oder less-Befehl. Hier werden zwei Register benötigt, deren Inhalte miteinander verglichen werden und ein Offset, das die jeweilige Sprungadresse angibt.
Somit werden für die Adressierung der Register insgesamt 4 Bit benötigt und für das Offset ebenfalls 4 Bit. Dem Operationscode bleiben nur noch 4 Bits zur Verfügung.

Opcode	Offset	reg1	reg
4Bit	4Bit	2Bit	2Bit

Struktur bei Sprung Operationen:

Ein Beispiel für Sprungbefehle sind die Befehle call, return oder goto. Für die Sprungadresse wird eine Konstante der Bit-Breite 8 angegeben, der Rest steht dem Operationscode zur Verfügung.

Opcode	Konstante
4 Bit	8 Bit

Struktur des* ldi *Befehls:

Als letztes Beispiel sei der Befehl ldi aufgeführt, welcher keine allgemeine Struktur wie die anderen Befehlen aufweist.
Der Befehl ldi benötigt eine Zieladresse in der die angegebene Konstante abgespeichert werden soll. Das heißt, 2 Bit werden für die Adressierung des Zielregisters und 8 Bit für die Konstante benötigt. Somit verbleiben für den Operationscode nur noch 2 Bit.

Opcode	reg	Konstante
2 Bit	2 Bit	8 Bit

2.1.2 Auscodierung des Operationscodes

Der Befehlscode (12 Bit) setzt sich aus dem Operations- und dem Datencode zusammen. Die Angaben des *Datencodes* für die jeweiligen Befehle ist die Darstellung einer *eindeutigen Wortbreite*. Der *Operationscode* hingegen ist die Darstellung einer Zuordnung einer bestimmten Bitzahl zu einer Operation, welche eine *variierende Größe* vorweist.

Um jedem Befehl einen eindeutigen Operationscode zuzuweisen, wird für die Auskodierung eine Baumstruktur verwendet. Die Baumstruktur ist so aufgebaut, dass die Knoten, die tiefer in der Hierarchie liegen, eine größere Wortbreite erhalten. Die Blätter stellen den Operationscode dar. Je kleiner die Wortbreite des Datencodes ist, desto tiefer liegt der Operationscode in der Baumstruktur. In der Abbildung 2.1 ist die Baumstruktur des Operationscodes dargestellt.

2.1.3 Befehlscode

Anhand der Daten aus den vorangegangenen Unterkapiteln kann nun die vollständige Tabelle des Befehlssatzes angegeben werden.

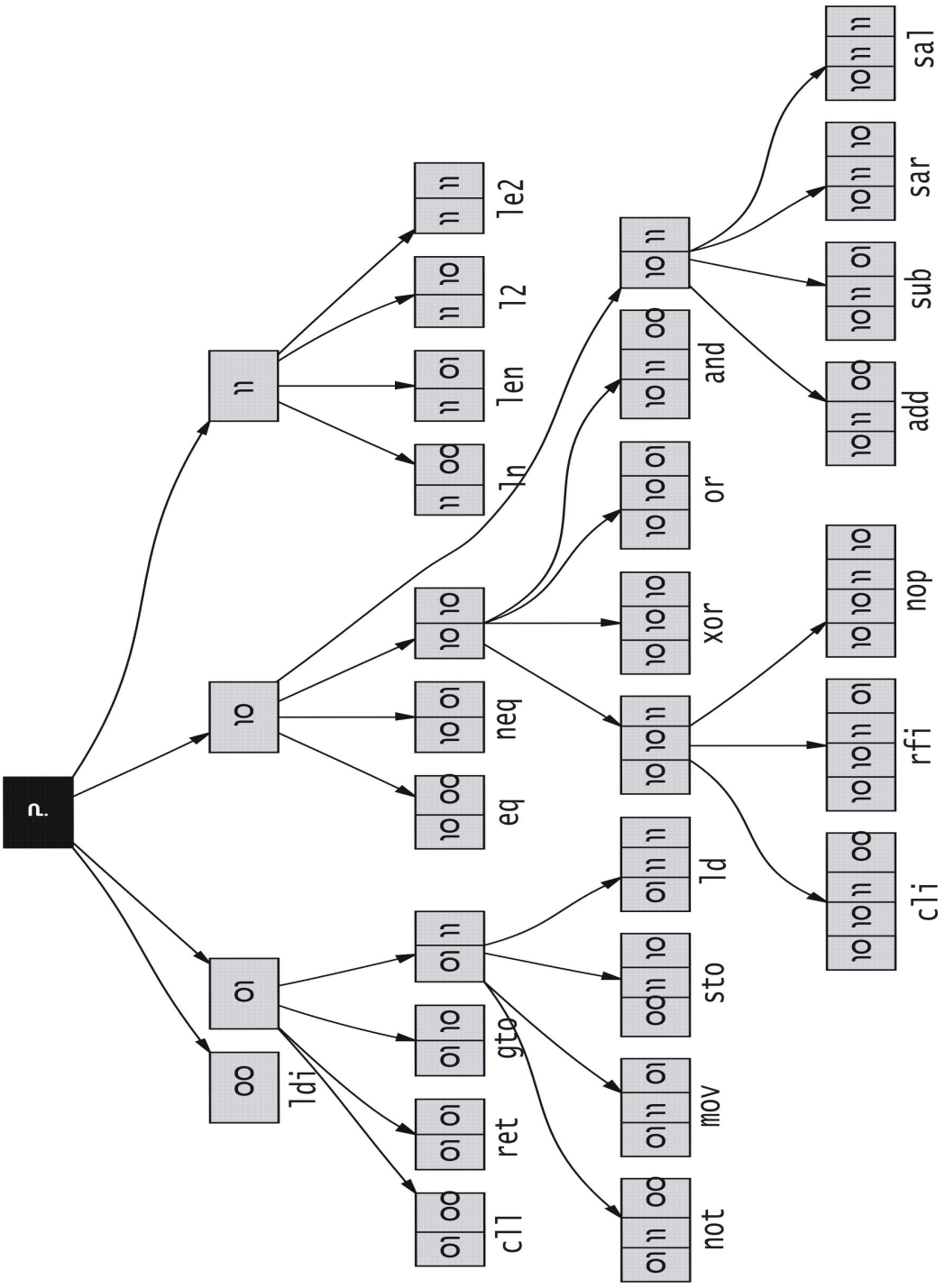


Abbildung 2.1: Baumstruktur des Operationscodes

	1	2	3	4	5	6	7	8	9	10	11	12
load integer	**0**	**0**	rd	rd	k	k	k	k	k	k	k	k
call	**0**	**1**	**0**	**0**	k	k	k	k	k	k	k	k
return	**0**	**1**	**0**	1	k	k	k	k	k	k	k	k
goto	**0**	**1**	**1**	**0**	k	k	k	k	k	k	k	k
not	**0**	**1**	**1**	**1**	**0**	**0**	-	-	rs	rs	rd	rd
move	**0**	**1**	**1**	**1**	**0**	**1**	-	-	rs	rs	rd	rd
store	**0**	**1**	**1**	**1**	**1**	**0**	-	-	rs	rs	rd	rd
load	**0**	**1**	**1**	**1**	**1**	**1**	-	-	rs	rs	rd	rd
equal	**1**	**0**	**0**	**0**	-	-	rs	rs	rs	rs	rd	rd
not equal	**1**	**0**	**0**	**1**	-	-	rs	rs	rs	rs	rd	rd
and	**1**	**0**	**1**	**0**	**0**	**0**	rs	rs	rs	rs	rd	rd
or	**1**	**0**	**1**	**0**	**0**	**1**	rs	rs	rs	rs	rd	rd
xor	**1**	**0**	**1**	**0**	**1**	**0**	rs	rs	rs	rs	rd	rd
callinter	**1**	**0**	**1**	**0**	**1**	**1**	**0**	**0**	-	-	-	-
returninter	**1**	**0**	**1**	**0**	**1**	**1**	**0**	**1**	-	-	-	-
noop	**1**	**0**	**1**	**0**	**1**	**1**	**1**	**1**	-	-	-	-
add	**1**	**0**	**1**	**1**	**0**	**0**	rs	rs	rs	rs	rd	rd
sub	**1**	**0**	**1**	**1**	**0**	**1**	rs	rs	rs	rs	rd	rd
shift arithmetical right	**1**	**0**	**1**	**1**	**1**	**0**	-	-	rs	rs	rd	rd
shift arithmetical left	**1**	**0**	**1**	**1**	**1**	**1**	-	-	rs	rs	rd	rd
less (nat)	**1**	**1**	**0**	**0**	f	f	f	f	rs	rs	rd	rd
less equal	**1**	**1**	**0**	**1**	f	f	f	f	rs	rs	rd	rd
less (int)	**1**	**1**	**1**	**0**	f	f	f	f	rs	rs	rd	rd
less equal	**1**	**1**	**1**	**1**	f	f	f	f	rs	rs	rd	rd

Tabelle 2.1: Befehlscode-Tabelle

rs:	Source Register	k:	Konstante
rd:	Destination Register	f:	Offset

Wie aus Tabelle 2.1 ersichtlich, haben wir die allgemeine Struktur der Befehle durch die Anordnung der Register, Konstanten und des Offsets an festen Positionen gleichmäßig dargestellt. Dies dient zum einen zu einer Vereinheitlichung der Befehle und zum anderen zur Vereinfachung der in der Hardware noch anzulegenden Leitungen.

2.2 Das PoCiLet-Board - Vom Schaltplan zur Platine

2.2.1 Vorüberlegungen und erste Ideen

Die erste Entscheidung, die zu treffen war, war die Architektur des Rechners festzulegen. Als Optionen hatten wir die Von-Neumann- und die Harvard-Architektur bestimmt, da diese uns aufgrund der weiten Verbreitung vom didaktisch größten Wert erschienen.
Letztendlich entschieden wir uns für die Harvard-Architektur. Grund dafür war die saubere Trennung zwischen Programm- und Datenspeicher, die die Verständlichkeit des Rechneraufbaus erhöht. Als weiterer Vorteil erschien uns eine einfachere Realisierung, da die Busse für Programm- und Datenspeicher so sehr leicht verschieden breit sein können. Dies erlaubt es, relativ einfach einen Prozessor zu entwerfen, der das Instruktionswort in nur einem Takt lesen kann. Eine Von-Neumann-Architektur hätte mehrere Takte benötigt; dies ist der Verständlichkeit eher abträglich.
Bei der Busbreite entschieden wir uns beim Datenbus für 8 Bit, da der Großteil der erhältlichen Logik-ICs (Bustreiber, Speicher etc.) für diese Busbreite ausgelegt sind. Ein weiterer Grund für diese Entscheidung war didaktischer Natur; fast alle heutigen Rechner arbeiten mit Bussen, deren Breite 8 Bit oder ein vielfaches davon ist. So wird die Ähnlichkeit von PoCiLet zu anderen Rechnern maximiert. Der Bus zum Programmspeicher wurde bereits durch den Befehlssatz auf 12 Bit festgelegt.
Ferner waren die I/O-Schnittstellen des Rechners zu entwerfen. Wir entschieden uns für eine Memory-mapped I/O-Schnittstelle. Motivation hierfür war, daß keine zusätzlichen Befehle für die I/O-Operationen notwendig sind. Als Eingabeeinheit wurde eine Tastatur vorgesehen; diese Lösung bietet dem Benutzer eine komfortablere Eingabemöglichkeit als etwa Dip-Schalter. Für die Ausgabeeinheit erschien eine Hexadezimalanzeige, die 8 Bit breite Datenwerte anzeigen kann, angebracht.
Als nächstes stand für die jeweiligen Komponenten die Entscheidung an, in welchem Detailgrad diese realisiert werden sollen. Die Möglichkeiten reichten von einer Black-Box-Realisierung (Komponente für den Benutzer nicht sichtbar, z.B. in einem CPLD[1]) über einen Aufbau mit Logik-ICs zu einer Realisierung mit diskreten Bauelementen (Transistoren).
Letztere Variante wurde in keinem Fall verwendet, um den Hardwareaufwand und den benötigten Platz auf der Platine überschaubar zu halten. Ebenso hätte die resultierende Komplexität für den Benutzer keinen didaktischen Wert gehabt, da diese Komplexität eher von den wesentlichen Abläufen im Rechner ablenken

[1] Complex Programmable Logic Device

würde. Der Registersatz sollte für den Benutzer definitiv sichtbar sein. Daher entschieden wir uns, jedes Register einzeln über Logik-ICs zu realisieren; ebenso sollte zu jedem didaktisch wichtigen Register eine Anzeige vorhanden sein, um den Prozessorzustand zu visualisieren. Für diese Aufgabe wurden in einer Reihe angeordnete LEDs verwendet, die einen 8-Bit-Binärwert darstellen. Diese Anzeigemöglichkeit wurde auch für die auf dem Datenbus anliegenden Werte gewählt.
Für die Speicherkomponenten wurde die Realisierung als Black-Box ausgewählt. Grund dafür ist der sehr regelmäßige interne Aufbau dieser Komponenten, der didaktisch nicht sehr wertvoll ist. Wir entschieden uns daher, komplette Speicherbausteine einzusetzen.
Das Steuerwerk sollte als CPLD realisiert werden. Diese Lösung war hardwareseitig überschaubar und bot auch Raum, den Befehlssatz noch nachträglich verändern zu können. Eine für den Benutzer detailliertere Realisierung wäre kaum möglich gewesen, da sich die Zustände des Steuerautomaten schlecht verständlich hätten visualisieren lassen. Die ALU[2] war zunächst als Logik-IC geplant, jedoch stellte sich heraus, dass ein entsprechender Baustein nicht zu beziehen war. Eine diskrete Realisierung schied wegen der hohen Komplexität ebenfalls aus. Letztendlich entschieden wir uns ebenfalls für eine CPLD-Realisierung.
Die I/O-Komponenten werden jeweils über einen Microcontroller an den PoCiLet-Rechner angeschlossen. Diese Lösung erlaubt es, komplexe Steuereinheiten der I/O-Schnittstellen mit geringem Aufwand zu realisieren. Dazu gehörte auch eine Interrupt-Steuerung, die einen Interrupt Request sendet, wenn eine Taste der Tastatur gedrückt wird. Da diese Mikrocontroller nicht zum eigentlichen PoCiLet-Rechner gehören, sind sie auf der Unterseite der Platine angebracht, um sie vor dem Benutzer zu verstecken. So sind nur die eigentlichen Komponenten des PoCiLet-Rechners sichtbar.
Anschließend galt es, aus diesen Komponenten eine komplette Rechnerarchitektur zu synthetisieren:

- Datenbus

  Dieser Bus verbindet die Register und den Datenspeicher.

- Programmdatenbus

  Dieser Bus verbindet den Programmspeicher mit dem IR[3]. Dieses IR-Register wird vom Steuerwerk ausgelesen.

[2] Arithmetic Logic Unit
[3] Instruction Register

- Adressbus

  Dieser Bus dient dazu, den Datenspeicher zu adressieren. Mittels des Registers MAR[4] können Werte auf diesen Bus gelegt werden.

- Speicher

  Es wurden getrennte Speicher für Daten und Programmcode vorgesehen (RAM, ROM).

- Programmzähler

  Dieses Register (PC[5]) dient dazu, den Programmspeicher zu adressieren.

- Allgemeiner Registersatz

  Der allgemeine Registersatz (Reg1 bis Reg4) steht dem Benutzer zur Verfügung.

- Stack

  Es gibt einen Stack, der zur Zwischenspeicherung des Programmzählers dient, wenn Unterprogramme aufgerufen werden.

- ALU

  Die ALU bezieht ihre Eingangswerte über die speziellen Eingangsregister ALUE1 und ALUE2. Ergebnisse von Vergleichsoperationen werden in ein separates Register (Flag-Register) geschrieben. Ergebnisse der Berechnungen werden auf den Datenbus geschrieben.

- Interrupts

  Die Realisierung benutzt Zwei Leitungen - IRQ[6] und IntACK[7], die das Steuerwerk mit dem Mikrocontroller, an den die Tastatur angeschlossen ist, verbinden. Zusätzlich gibt es ein Register INTVEC[8], in dem die Speicheradresse abgelegt ist, an der die Interrupt-Routine beginnt.

[4] Memory Address Register
[5] Program Counter
[6] Interrupt Request
[7] Interrupt Acknowledge
[8] Interrupt Vector

- Konstantenregister

  Dieses Register erlaubt es, Daten aus dem Instruktionswort auf den Datenbus zu legen. Dies ist für z.B. LDI-Instruktionen wichtig.

- Takt- und Resettaster

  Diese Komponenten dienen zur Steuerung des PoCiLet-Rechners. Ebenso wurde ein automatischer Taktgenerator vorgesehen.

- I/O-Einheiten

  Tastatur als auch Anzeige werden auf die Speicheradresse 255 abgebildet.

2.2.2 Auswahl der Bauteile und deren Funktionen

In diesem Unterkapitel sollen die wesentlichen Bausteine, aus denen Pocilet aufgebaut ist, beschrieben werden. Die verwendeten ICs sind aus der HC-Serie[9]. Gebenüber TTL[10] zeichnen sich diese Bausteine durch eine geringe statische Verlustleistung und eine geringere Störempfindlichkeit aus.
Generell wurde versucht, SMD[11]-Bausteine zu verwenden, da diese Gehäuse weniger Platz auf der Platine verbrauchen. Zusätzlich zu den in diesem Unterkapitel erwähnten Bauteilen werden noch einige weitere Standard-Logik-ICs und passive Bauteile verwendet, die nicht im Detail erklärt werden.

Spannungsversorgung

Die ausgewählten ICs benötigen eine Versorgungsspannung von 5V. Zur Bereitstellung dieser Spannung wurde ein Gleichspannungsnetzteil ausgewählt. Zur Sicherung des Boards wurde eine Schmelzsicherung vorgesehen.
Um die Spannung zu stabilisieren, wurden 4 Elektrolytkondensatoren mit einer Kapazität von je 100 μF in die Schaltung aufgenommen. Ferner wurde jedem IC ein Abblockkondensator zugeordnet, um ein Einbrechen der Spannung bei den bei Schaltvorgängen entstehenden Stromspitzen zu verhindern. Diese Kondensatoren haben eine Kapazität von 10 nF und sind auf dem Board über möglichst kurze Leiterbahnen mit den Pins zur Spannungsversorgung des entsprechenden ICs verbunden.

[9] High Speed CMOS
[10] Transistor-Transistor Logic
[11] Surface Mounted Device

Latches

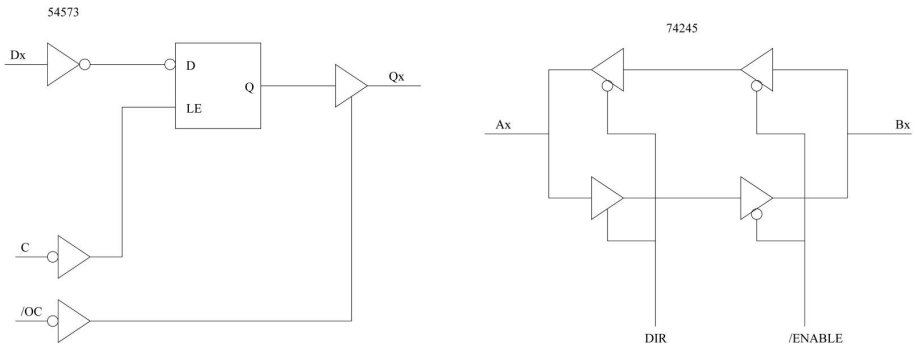


Abbildung 2.2: Schematischer Aufbau der Bausteine 74573, 74245

Kernelement des Rechners sind Registerbausteine. Hier wurde der Typ 74573 (Abbildung 2.2) ausgewählt. Als Gehäuse wurde das TSSOP[12]-Modell verwendet, um den Platzbedarf auf der Platine zu minimieren. Der Baustein kann 8 Datenbits speichern. Diese Daten werden von den Eingängen D1 bis D8 übernommen, wenn der Pin C auf High gesetzt wird. Um die gespeicherten Daten an den Ausgangspins auszugeben, muss der Pin /OC auf GND gezogen werden. Ansonsten sind die Ausgangspins hochohmig.
Da die Treiberleistung dieser Chips gering ist, wurde für die Register, die mittels LEDs angezeigt werden, jeweils ein weiterer Registerbaustein parallel geschaltet. Dessen Ausgänge wurden mit den LEDs verbunden.

LEDs

Bei der LED-Auswahl wurde Wert darauf gelegt, dass sie direkt nebeneinander angeordnet werden können. Das heißt, dass die Kontaktpads nicht die gesamte Gehäusebreite einnehmen. Ebenso war ein nicht transparentes Gehäuse erwünscht, um nebeneinander angeordnete LEDs besser differenzieren zu können.
Zur Begrenzung des Durchlassstroms sind die LEDs mit 470 Ω Widerständen in Reihe geschaltet. Dabei werden zur Minimierung des Platzbedarfs Widerstandsarrays mit jeweils vier Widerständen verwendet.
Um die Übersichtlichkeit der angezeigten Information zu verbessern, werden LEDs in verschiedenen Farben verwendet. Für die Anzeige von in Registern gespeicherten Daten werden gelbe LEDs verwendet. Steuersignale, die ein Register beschreiben, sind mit einer roten LED verknüpft. Steuersignalen, die ein Registers auslesen, wird eine grüne LED zugeordnet.

[12]Thin Shrink Small Outline Package

Für den Ausgabeport des Rechners werden zwei 5x7-LED-Punktmatrixanzeigen verwendet, auf denen sich jeweils eine einstellige Hexadezimalzahl darstellen lässt. Diese Anzeigematrix lässt sich dabei über Zeilen- und Spaltenleitungen ansprechen, wobei die Zeilenleitungen mit den Anoden, die Spaltenleitungen mit den Kathoden der LEDs verbunden sind. Die Zeichendarstellung erfolgt im Spaltenscan-Verfahren, das heisst, es werden nacheinander alle Spaltenleitungen einzeln aktiviert. Zur Aktivierung einer Spaltenleitung wird dieser Pin auf Masse gezogen. Soll eine LED in der aktivierten Spalte leuchten, so muß, wenn diese Spalte aktiviert ist, die entsprechende Zeilenleitung über einen Vorwiderstand mit 5V, ansonsten mit Masse verbunden werden.

CPLDs

Steuerwerk und ALU sollten mittels CPLDs realisiert werden, da diese Komponenten für einen diskreten Aufbau zu komplex sind. Gewählt wurde das Modell 9572-15 von Xilinx. Dieser Chip hat ein PLCC84[13]-Gehäuse, bietet 69 I/O-Pins und 1600 interne Logikgatter, die in 72 Makrozellen aufgeteilt sind. Zur Programmierung dient ein JTAG-Interface. Da für dieses Interface dedizierte Pins vorhanden sind, muss der Chip dazu nicht aus der Schaltung ausgebaut werden. Dieser Chip benötigt eine Betriebsspannung von 5V und kann sowohl in 5V- als auch 3.3V-Umgebungen eingesetzt werden.
Gewählt wurde dieser Chip, weil die dazu passenden Entwicklungstools (Webpack, ModelSim[14]) kostenlos vom Hersteller bereitgestellt werden und kein aufwendiger Programmieradapter erforderlich ist.
Da die Treiberleistung dieser Chips nicht sehr groß ist, wurden, soweit erforderlich, die Signale dieser Pins mit Bausteinen vom bereits erwähnten Typ 74573 verstärkt.

RS232-Schnittstelle

Die Steuerung des PoCiLet-Boards sollte durch einen Mikrocontroller realisiert werden. Hier fiel die Wahl auf den Typ PIC18F452 der Firma Microchip[15]. Dieser Mikrocontroller besitzt ein integriertes RAM und ein Flash-ROM für Programmcode. Zur Beschreibung dieses Flash-ROMs bietet der Chip ein Interface mit dedizierten Pins. So kann der Mikrocontroller auch noch programmiert werden, wenn er bereits in der Schaltung eingebaut ist.

[13] Plastic Leaded Chip Carrier, 84 Pins
[14] siehe auch Kapitel 2.2.3
[15] microchip.com

Weiterhin zeichnet sich der Baustein durch eine Vielzahl an frei verfügbaren I/O-Pins aus, die alternativ auch von speziellen Funktionseinheiten genutzt werden können. Von diesen Funktionseinheiten wird beispielsweise die eingebaute UART-Schnittstelle[16] genutzt.
Der Takt des Chips muss extern erzeugt und an den vorgesehenen Eingängen angelegt werden. Von den mehreren zur Verfügung stehenden Varianten wurde die Takterzeugung mit einem Quarzoszillator verwendet, da diese Möglichkeit eine für das Timing der UART-Schnittstelle ausreichende Präzision bietet. Ein Chip vom Typ MAX233 wurde vorgesehen, um die auf dem Board verwendeten Signalpegel von +5V und 0V in die für die Schnittstelle verwendeten +12V beziehungsweise -12V umzuwandeln. Dies geschieht durch eine in den Chip integrierte Ladungspumpe.
Der Chip besitzt jeweils 2 Kanäle für die Wandlung von TTL-Signalpegeln auf RS232-Pegel sowie für die Rückwandlung der RS232-Pegel zu 0V/5V-Signalen.

Speicher

Als Datenspeicher wurde ein SRAM gewählt. Der Vorteil des SRAMs gegenüber eines DRAMs ist, dass keine den Hardwareaufbau verkomplizierenden Refresh-Steuersignale erzeugt werden müssen. Gewählt wurde der Chip 62256-70M, der 64k x 8 Bit speichern kann. Es wird jedoch lediglich ein Adressraum von 8 Bit benutzt.
Der Programmspeicher sollte nicht flüchtig sein. Diese Eigenschaft und eine einfache Reprogrammierbarkeit wird von EEPROMs geboten. Ein Flash-Speicher würde diese Eigenschaften ebenfalls erfüllen. Er hat jedoch den Nachteil, dass er nicht wortweise beschrieben werden kann. Gewählt wurde schließlich der Baustein 28C64, der eine Kapazität von 8k Wörten mit jeweils 8 Bit besitzt. Um die erforderliche Wortbreite zu erhalten, wurden zwei Bausteine parallel geschaltet.

Tastatur

Als Eingabemöglichkeit wird eine Tastatur mit 16 Tasten verwendet. Diese Tasten sind in einer 4x4-Matrix angeordnet. Jeder Zeile und jeder Spalte dieser Matrix ist ein Pin zugeordnet. Zur Überprüfung, ob eine Taste gedrückt ist, wird jeweils einer der Pins, der einer Spalte der Matrix zugeordnet ist, auf 5V gesetzt. Ist nun eine Taste gedrückt, wird der der entsprechenden Zeile zugeordnete Pin ebenfalls auf 5V gezogen. Um ein eindeutiges Signal zu erhalten, wenn keine Taste gedrückt ist, sind alle Zeilenleitungen mit einem Pulldown-Widerstand versehen.
Die Ansteuerung der Tastatur erfolgt durch den Microcontroller. Dieser Controller übernimmt auch softwareseitig das Entprellen der Tasten.

[16] Universal Asynchronous Receiver And Transmitter

Schmitt-Trigger

Zur Entprellung des Takttasters wird der Baustein 7414, ein invertierender Schmitt-Trigger verwendet. Dieser Baustein zeichnet sich dadurch aus, dass der Ausgang ab einer bestimmten Eingangsspannung U_1 auf Low schaltet und diesen Wert beibehalt, bis die Eingangsspannung auf weniger als U_2 fällt; dabei gilt $U_2 < U_1$.
Als Eingangssignal des Triggers dient dabei das Signal, das durch den Takttaster mit parallel geschaltetem Kondensator generiert wird.
Aufgabe des Schmitt-Triggers ist es, Eingaben im gesamten Spannungsbereich (im Gegensatz zur digitalen Logik) abzudecken sowie durch die unterschiedlichen Schwellwertspannungen U_1 und U_2 ein mehrmaliges Umschalten des Ausgangs zu verhindern.

Treiberbausteine

Da der verwendete Microcontroller keine ausreichenden Ströme für die gewünschte LED-Helligkeit liefern kann, wird der Treiberbaustein UDN2585 verwendet. Es handelt sich dabei um einen Source-Driver-Baustein, der an die Kathoden der LEDs angeschlossen wird.
Da der für die Tastatur und das Beschreiben des EEPROMs verwendete Mikrocontroller nicht ausreichend General-Purpose-I/O-Pins besitzt, musste mit Bustreibern ein Multiplexing der Leitungen vorgenommen werden. Als Bustreiber wurden die Bausteine 74244 und 74245 ausgewählt.
Der erste genannte Baustein kann 8 Leitungen (A1-A8) elektrisch mit weiteren 8 Pins (Y1-Y8) verbinden oder trennen. Diese Steuerung geschieht in Gruppen von jeweils 4 Leitungen. Der Baustein 74245 (Abbildung 2.2) kann 8 parallelen Leitungen gleichzeitig entweder trennen oder verbinden; die Verbindungen sind unidirektional. Der Pin DIR bestimmt die Durchlassrichtung.

2.2.3 Auswahl der Entwicklungtools

Schaltplan und Platinenlayout

Zur Erstellung der Platine wurde ein CAD/EDA[17]-Werkzeug, das auf diese Aufgabe ausgerichtet ist, benötigt. Neben der grundlegenden Funktionalität, einen Schaltplan zu entwerfen und aus diesem ein Platinenlayout zu erzeugen, war folgende Funktionalität erwünscht:

[17] CAD: Computer Aided Design, EDA: Electronic Design Automation

- Dateiexport

  Das fertige Platinenlayout sollte in einem vom Platinenhersteller unterstützten Format abspeicherbar sein.

- Autorouter

  Um zeitaufwendiges manuelles Routen der Leiterplatte zu vermeiden, sollte ein Autorouter vorhanden sein.

- Umfangreiche Bauteilbibliotheken

  Auch hier war das Ziel, den Arbeitsaufwand zu minimieren. Ebenso sollten neue Bibliotheken erstellt werden können.

Die geforderte Funktionalität wird von Eagle[18] der Firma Cadsoft geboten. Da bereits eine Lizenz für dieses Produkt vorhanden war, war die Benutzung naheliegend. Darüberhinaus bietet die Software noch weitere Funktionen, die sich als sehr nützlich herausgestellt haben:

- Electrical Rule Check, Design Rule Check

  Diese Funktionen können den Schaltplan und das Platinenlayout auf offensichtliche Fehler hin überprüfen.

- Scripting-Fähigkeit

  Diese Funktion wurde benutzt, um eine Bauteilliste zu erstellen, die als Grundlage der Komponenten diente. Ebenso war diese Funktion sehr hilfreich, um Schaltplan und Platinenlayout so zu exportieren, dass diese Daten von der Simulationssoftware genutzt werden konnten.

Inbetriebnahme

Für die Inbetriebnahme des Prototyps wurde dieser inkrementell aufgebaut. Dabei wurden in jedem Schritt neu aufgelötete Bauelemente getestet. Dazu war es erforderlich, die Hardware auf Pin-Ebene zu manipulieren. Als geeignete Ansatzpunkte, um einen Großteil des Tests abdecken zu können, wurden die CPLDs und die Microcontroller ausgemacht. Diese Bausteine sind in PLCC-Sockeln[19] ausgeführt. Zur Inbetriebnahme wurden die Sockel dieser Chips mit PLCC-Adaptern

[18] Easily Applicable Graphical Layout Editor
[19] Plastic Leaded Chip Carrier

bestückt, welche wiederum mit Breakout-Boxen (Abbildung 2.3) von Measurement Computing[20] verbunden wurden. Dabei handelt es sich um programmierbare I/O-Geräte, d.h., es lassen sich in Software die Pins als Ein- beziehungsweise Ausgänge konfigurieren und mit Werten belegen. Zur Abstraktion dieser Geräte diente die mitgelieferte Universal Library, die es erlaubt, die Breakout-Boxen aus Programmiersprachen heraus direkt anzusprechen.

Abbildung 2.3: Breakout-Box

Es wurde das Programm Labview von National Instruments[21] verwendet. Dabei handelt es sich um eine grafische Entwicklungsumgebung für die Signalerfassung, -analyse und -präsentation. Das Programm erlaubt es, sogenannte virtuelle Instrumente zu entwickeln. Diese Instrumente bestehen aus einer virtuellen Bedieneinheit (Front Panel) und einem Blockdiagramm. Erstere Komponente enthält Elemente wie Schalter und Anzeigen. Diese Elemente sind auch im Blockdiagramm enthalten und können dort - zusammen mit weiteren, nur im Blockdiagramm enthaltenen Elementen - verknüpft werden. Für Labview sprach ein vergleichsweise geringer Aufwand für die Realisierung der benötigten virtuellen Elemente, verglichen mit der Entwicklung von Testprogrammen (z.B. in C), die eine vergleichbare Funktionalität aufweisen.

[20] www.measurementcomputing.com
[21] www.ni.com

Programmierung der Microcontroller

Abbildung 2.4: In Circuit Debugger ICD2

Zur Programmierung der PIC-Microcontroller stellt der Hersteller die Entwicklungsumgebung MPLAB bereit. In diese Umgebung können Entwicklungstools wie Compiler, Assembler, Linker, Debugger und Programmieradapter integriert werden beziehungsweise werden bereits mitgeliefert.
Zur Programmierung und zum Debuggen wurde der In-Circuit-Debugger (ICD2) verwendet (Abbildung 2.4). Dieser Debugger wird sowohl an den Host-Rechner (d.h., der Rechner, auf dem MPLAB ausgeführt wird) als auch an den Programmieradapter des Microcontrollers angeschlossen. Das erlaubt, im Gegensatz zu reinen Software-Simulatoren, die Funktionsweise der Software in der realen Hardwareumgebung zu überprüfen.

Programmierung der CPLDs

Das logische Verhalten der CPLDs wurde in der Hardwarebeschreibungssprache VHDL[22] modelliert. Diese Beschreibungen können durch Synthesewerkzeuge in eine logische Schaltung übersetzt werden. Aus dieser Schaltung kann dann eine Konfigurationsdatei für einen programmierbaren Logikbaustein erzeugt werden. Da dieser Prozess vom verwendeten Baustein abhängig ist, gab es keine andere Wahl, als die Software des Herstellers zu verwenden, der auch die CPLDs herstellt. Dabei handelt es sich um das Xilinx Webpack[23]. Diese Software kann sowohl die Synthese als auch die Abbildung auf Hardware übernehmen.

[22] Very High Speed Integrated Circuit Description Language
[23] WebPACK 6.2i: http://www.xilinx.com/ise/wp_overview.pdf

Logische Verifikation des Hardwareaufbaus

Zur Verifikation des logischen Hardwaremodells wurde der entwickelte Rechner komplett in bereits erhältlicher Simulationssoftware simuliert. Da einige Komponenten in VHDL beschrieben sind, war es naheliegend, eine VHDL-Beschreibung für das gesamte System zu entwickeln. Zur Durchführung der Simulation wurde das Programm Modelsim (ebenfalls von Xilinx) ausgewählt, hauptsächlich aus Gründen der Verfügbarkeit.

2.2.4 Entwicklung des Schaltplans

Das erste Bauelement

Der Pocilet-Schaltplan wurde zunächst in mehrere logische Einheiten unterteilt (Registersatz, ALU, Steuerwerk, Speicher, I/O und Programmierinterface). Begonnen wurde die Entwicklung des Schaltplans mit der Verschaltung des Registersatzes. Danach wurde der Schaltplan um die anderen logischen Einheiten erweitert. Beim späteren Aufbau der Testplatinen als auch beim Aufbau des Prototyps wurden Fehler im Schaltplan erkannt und behoben. Änderungen, die diese Fehler beseitigen, wurden direkt in den Schaltplan übernommen.

Erstellen neuer Bibliotheken

Da die von Eagle mitgelieferten Bauteilbibliotheken nicht alle verwendeten Bauteile enthalten, wurde eine zusätzliche Bibliothek erstellt. Die in dieser Bibliothek enthaltenen Bauteilbeschreibungen definieren die Bauteile sowohl für den Schaltplan- als auch den Layouteditor. Als Grundlage der Beschreibungen wurden die Datenblätter der Hersteller verwendet. Nach Möglichkeit wurden bereits in Eagle definierte Gehäuseformen übernommen. Waren diese nicht vorhanden, so wurde ein neues Gehäuse definiert. Um die Korrektheit des Footprints zu überprüfen, wurde das Bauteillayout ausgedruckt und mit dem realen Bauteil verglichen.

Fertigstellung des ersten Schaltplans

Die Funktion von Eagle, Skripte auszuführen, hat sich bei der Fertigstellung des Schaltplans als sehr nützlich erweisen. Das Skript Bill-Of-Materials (BOM) zum Beispiel erzeugt eine Tabelle der verwendeten Bauteile. Diese Tabelle diente uns dann als Grundlage für die Bestellung der Bauteile.
Außerdem war es erforderlich, die Bauteilnamen, die im Simulator verwendet werden mit den im Schaltplan verwendeten abzugleichen. Anstatt aufwendig alle Namen direkt in Eagle zu ändern, wurden alle Bauteil- und Netznamen mit Hilfe von Eagle-Skripten exportiert. Es wurde eine Tabelle erstellt, die in einer Spalte

den jeweils alten und neuen Namen eines Elements enthielt. Diese Tabelle wurde dann genutzt, um Board- und Schematic-Datei zu patchen. Dies ist problemlos möglich, da die Namen in den Eagle-Dateien als Strings fester Länge gespeichert sind.

2.2.5 Programmierung der CPLDs

Bei den für unsere Zwecke gewählten CPLDs[24] XC95108 der Firma Xilinx[25] handelt es sich um programmierbare Logikbausteine. Sie werden über vier spezielle Programmiersignale (TDO[26], TDI[27], TMS[28] und TCK[29]) über die nach IEEE 1149.1 festgelegte JTAG-Schnittstelle[30] mit einem von uns zuvor in VHDL beschriebenen Verhalten versehen. Dazu wird eine VHDL-Beschreibung mit dem von Xilinx entwickelten Softwarepaket Webpack (siehe Kapitel 2.2.3) in ein spezielles Programmierfile übersetzt. Das im Internet frei verfügbare Programmiertool NAXJP[31] überträgt das generierte File über die parallele Schnittstelle eines PCs.

Realisierung der ALU

Bei der ALU handelt es sich um ein reines Schaltwerk (siehe [Jan03]), das eine logische Funktion realisiert. Es wird ein Ergebnisvektor O und ein Flagvektor F in Abhängigkeit zweier Eingangsvektoren A, B und dem Steuersignal St ermittelt.

$$\Phi(A, B, St) \rightarrow (O, F)$$

Bei den beiden Eingangsvektoren handelt es sich um zwei 8 Bit Werte aus den beiden Eingangsregistern. Das Steuersignal variiert je nach gewünschter Operation und wird vom Steuerwerk bereitgestellt. Der CPLD führt nun die gewünschte Operation aus und legt das Ergebnis im Ausgangsregister ab. Zusätzlich wird ein 4 Bit Vektor an das Flagregister übertragen, dessen Bits für Sign, Zero, Overflow, und Less-than stehen.

In VHDL wird nun genau diese Funktion verwirklicht, indem die Ein- und Ausgangsvektoren definiert werden und die gewünschte Funktion in einem sogenannten Prozess definiert wird. Dazu wird in einer case(St)..when(xxxx) Struktur das

[24] CPLD - complex programmable logic devices http://www.xilinx.com
[25] www.xilinx.com
[26] TDO - Test Data Out
[27] TDI - Test Data In
[28] TMS - Test Mode Select
[29] TCK - Test Clock
[30] JTAG - Joint Test Action Group http://www.jtag.com
[31] NAXJP project: http://www.nahitech.com/nahitafu/naxjp/naxjp-e.html

Steuersignal überprüft und die gewünschte Operation ausgeführt. Zusätzlich wird der Prozess mit den Eingangsvektoren verknüpft. Dies bedeutet, dass sobald sich ein Eingangssignal verändert, der Prozess ausgeführt wird. Ein Teil des VHDL Codes ist im unten stehenden Listing zu sehen.

Listing 2.1: VHDL Beschreibung der ALU

```
1  ENTITY alu IS PORT(
2      A, B : IN     STD_LOGIC_VECTOR(7 downto 0);
3      OPCODE       : IN     STD_LOGIC_VECTOR(4 downto 0);
4      OUTPUT       : OUT    STD_LOGIC_VECTOR(7 downto 0);
5      FLAG : OUT    STD_LOGIC_VECTOR(3 downto 0);
6      ALUAUSWE     : OUT    STD_LOGIC;
7      FLAGWE       : OUT    STD_LOGIC);
8
9      attribute pin_assign : string;
10     attribute pin_assign of A : signal is "12 13 14 15 17 18 19 20";
11     attribute pin_assign of B : signal is "3 4 5 6 7 9 10 11";
12     attribute pin_assign of OUTPUT : signal is "
           46 45 44 43 37 36 35 34";
13     attribute pin_assign of FLAG : signal is "66 67 68 69";
14     attribute pin_assign of FLAGWE : signal is "63";
15     attribute pin_assign of ALUAUSWE : signal is "55";
16 END;
17
18 ARCHITECTURE Behavior OF alu IS
19 BEGIN
20     PROCESS (OPCODE,A,B)
21     VARIABLE FLAGOUT : STD_LOGIC_VECTOR(3 downto 0);
22     VARIABLE OVF,LS : STD_LOGIC;
23     VARIABLE OPT : STD_LOGIC_VECTOR(7 downto 0);
24
25     BEGIN
26         CASE OPCODE IS                          --testing OPCODE
27             WHEN "00000" =>                     --NOT
28                 OPT := NOT A;
29
30             WHEN "00001" =>                     --EQUAL
31                 IF (A = B) THEN
32                     OPT := "00000001";
33                 ELSE
34                     OPT := "00000000";
35                 END IF;
36             WHEN "00010" =>                     -- AND
37                 OPT := (A AND B);
38             WHEN "00011" =>                     -- OR
39                 OPT := (A OR B);
40                     ...
```

In Zeile 1-19 wird die Struktur der ALU beschrieben, indem die Ein- und Ausgänge festgelegt werden. Desweiteren wird in den Zeilen 9-15 die Pinbelegung der Signale im PLCC84-Sockel festgelegt, wie sie auch auf dem PoCiLet- Board realisiert ist. Ab Zeile 21 wird in der Architecture das Verhalten der ALU beschrieben. Es werden interne Variablen festgelegt und schließlich im Process(OPCODE,A,B) dann die gewünschte Funktion gemäß OPCODE ausgeführt.

Realisierung des Steuerwerks

Das Steuerwerk ist ein taktgesteuerter Automat (siehe [Jan03]). Ein Takt veranlasst das Steuerwerk, seine Ausgangsignale in den Folgetakten so zu verändern, dass ein Befehl gemäß dem anliegenden Instruktionswort ausgeführt wird. Ein Taktsignal lässt das Steuerwerk also in Abhängigkeit seiner Eingänge und seines inneren Zustandes in wohldefinierte Zustände mit entsprechendem Ausgangsverhalten übergehen. Es handelt sich hier also um einen Moore-Automaten mit Zuständen Z und einer Ausgabe A, der als Eingänge ein Resetsignal R, ein Taktsignal T, das Instruktionswort IW, das Flagregister F und ein Interruptsignal IR besitzt. Hier gilt:

$$\Theta(Z_x, IW, R, T, F, IR) \rightarrow (Z_y)$$

$$\Phi(Z_y, IW, R, F, IR) \rightarrow (A_y)$$

Der konkret für unser Steuerwerk entwickelte Automat ist in Abbildung 2.5 und in der im Anhang zu findenden Tabelle 6.1 beschrieben. In der Grafik ist zu erkennen, wie das Steuerwerk funktioniert. Nach der Initialisierung gelangt das Steuerwerk in den Startzustand. Dort analysiert es das Instruktionswort und wechselt dementsprechend in einen der Folgezustände. Diese sehen in der Regel so aus, dass eine Datenquelle und ein Datenziel definiert werden. Ein Steuersignal lässt das zugehörige Bauelement ein Datum auf den Datenbus legen. Ein zweites Steuersignal führt zur Speicherung des Datums im adressierten Bauelement. Nach mehreren Zustandsübergängen ist jeder Befehl, wie in der Grafik ersichtlich, abgearbeitet und das Steuerwerk befindet sich wieder im Startzustand. Darauf kann der nächste Befehl bearbeitet werden.
Es handelt sich hier um einen optimierten Automaten, da viele Befehle gleiche Strukturen aufweisen und somit die Zustände zusammengefasst werden können. Dies geschieht bei allen Befehlen, bei der die ALU beteiligt ist. So werden zunächst die Eingangsregister (ALUREG_2, ALUREG_1a, ALUREG_1b) der ALU beschrieben und diese errechnet ein Ergebnis, dass Sprungbefehle (Auswertung) bzw. ein Speichern des Ergebnisses (REG_Zuweisung) auslöst.
In VHDL werden nun analog zur ALU die Ein- und Ausgänge inklusive der Pinbelegung definiert. Zusätzlich werden alle inneren Zustände in Form einer Variablen festgehalten, die das Steuerwerk von einem Initialzustand erreichen kann. Der

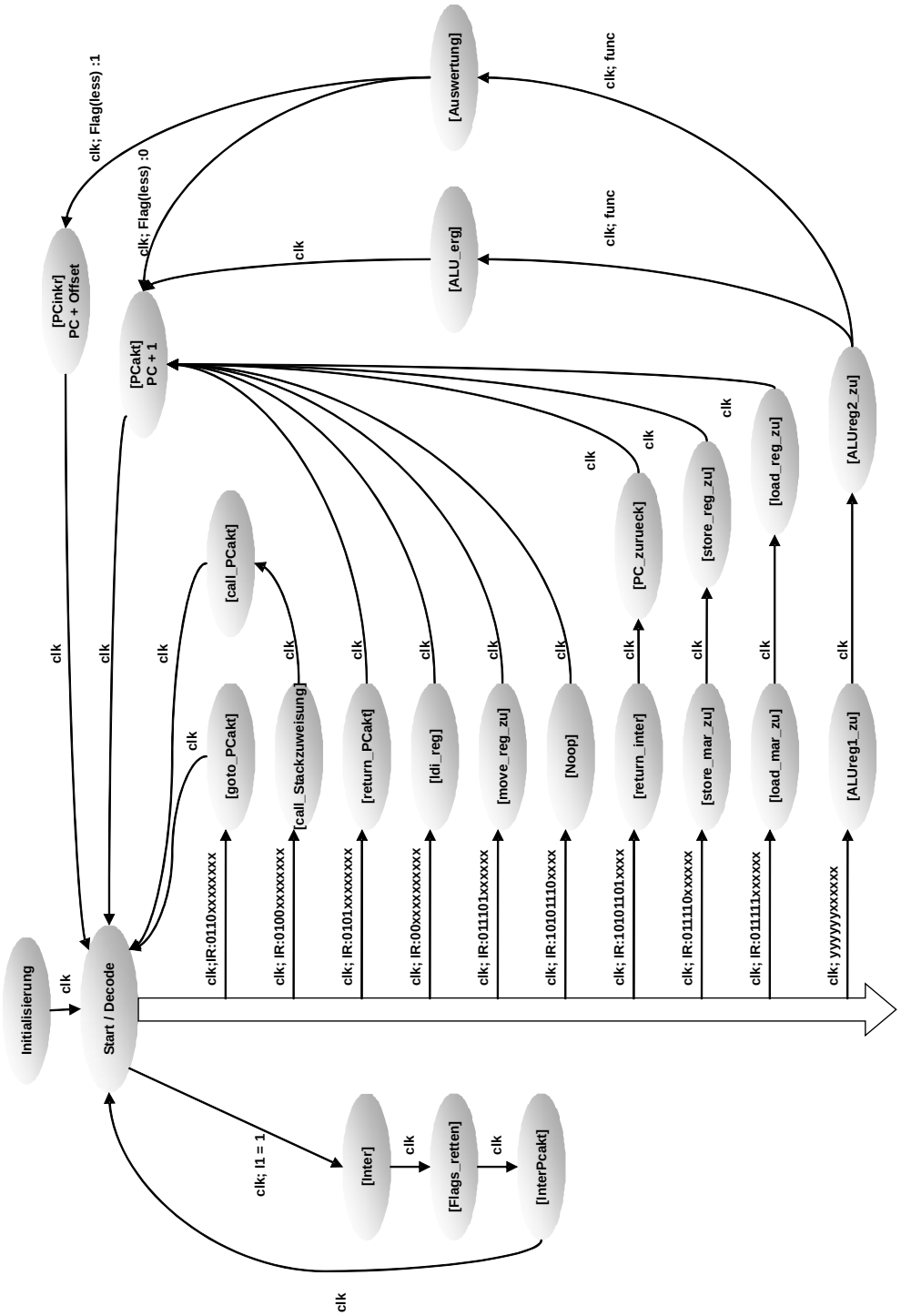


Abbildung 2.5: Zustandsübergangsdiagramm des Steuerwerks

nach außen sensible Prozess A reagiert nur auf das Taktsignal und das Resetsignal. Ein Reset setzt das Steuerwerk in seinen Initialzustand. Jedes folgende Taktsignal setzt das Steuerwerk gemäß Θ in einen Folgezustand. Nach dem Initialzustand gelangt das Steuerwerk in einen Startzustand. In diesem wird das Instruktionswort ausgelesen und somit wiederum der Folgezustand festgelegt. Dieser und weitere Zustände setzen jeweils die Steuersignale, um die Instruktion zu realisieren. Es werden Daten transportiert bzw. arithmetische und logische Operationen mittels ALU ausgeführt. Die Beschreibung der Zustandsübergänge und der zugehörigen Veränderung der Steuersignale geschieht in einem parallel laufenden Prozess B. Dieser wird vom Prozess A nach einem eingehenden Taktsignal durch Veränderung der Variable current_state aktiviert. Die Prozessstruktur des Steuerwerks sei in folgendem Ausschnitt (Listing 2.2) der VHDL Beschreibung festgehalten.

Listing 2.2: VHDL Beschreibung des Steuerwerks

```
1  Synch: process (CLK,Reset)
2  begin
3          IF (Reset='0') THEN
4                  Current_State <= Init;
5          ELSIF (CLK'event AND CLK = '1') THEN
6                  Current_State <= Next_State;
7          END IF;
8
9  end process Synch;
10
11 Automat: process (Current_State)
12
13 begin
14         CASE Current_State IS
15                 WHEN Init =>
16
17                 Next_State<=Start;
18                 -- PC on "00000000"
19                 -- reading instructionword from ROM
20
21                 ALUST<="11111", "11110" after 100 ns, "11111" after
                       400 ns;
22                 PCWE<='0', '1'after 200 ns, '0' after 300 ns;
23                 MAR<='0', '1'after 300 ns, '0' after 400 ns;
24                 IRWE <= '0', '1' after 600 ns, '0'after 700 ns;
25                 ROMRE <= '1', '0' after 500 ns;
26
27                 WHEN Start =>
28
29                 IF (InterReq = '1') THEN
30                         Next_State <= Inter;
31                 Interack <= '1';
32                 ELSE
```

```
33
34                          -- analysis of instructionword
35
36                  END IF;
37
38                  WHEN ...
```

2.2.6 Verifikation des Schaltplans durch Simulation

Einteilung der Simulation in Softwaresimulation und Testaufbauten

Die Vielzahl der auf dem PoCiLet-Board verwendeten Hardware verlangt eine Verifikation der Funktionalität und des Zusammenspiels der einzelnen Komponeneten. Die unterschiedliche Struktur, Komplexität, sowie Aufgabe der Elemente führt dazu, dass sich für die einen eine Simulation in Software, für die anderen eine Simulation in Hardware anbietet. Folglich wird die Funktionalität des Rechnersystems bestehend aus ALU, Steuerwerk, Registern, ROM, RAM und einigen Logikbausteinen in VHDL auf Softwareebene simuliert. Die Schnittstellen zur Außenwelt, Anzeige, Tastatur und Programmierschnittstelle werden in Testaufbauten nachgebaut und ihre Funtionalität durch diese Prototypen verifiziert. Da VHDL nur die logischen Verhaltensweisen der Komponenten simulieren kann nicht aber Signalpegel und Treiberleistung, werden zusätlich die CPLDs, die später die Aufgaben der ALU und des Steuerwerks übernehmen sollen, im Zusammenspiel mit einem Register getestet, um auch die elektrische Funktionalität gewährleisten zu können (Pegel, Treiberleistung). Die Aufteilung der Simulation auf Software- und Hardwareebene bedarf einer genauen Definition der Schnittstellen beider Teile. Eine Schnittstelle bildet das ROM, da es in der Softwaresimulation als vom Mikrocontroller fest programmiert angenommen werden kann. Desweiteren werden Takt und Reset Signale der Sofwaresimulation zugeführt.

Simulation in VHDL

VHDL ist in der Lage, Hardware zu beschreiben und somit eine Grundlage für die Simulation eines, aus einzelnen Komponenten zusammengesetzten Systems, zu schaffen. VHDL bietet also die Möglichkeit, Hardware zu modellieren und zu simulieren, bevor überhaupt ein reales Modell entstanden ist. Eine Simulationssoftware kann dann anhand dieses Modells eine analysierbare Ausgabe schaffen. Als Simulationssoftware bietet sich hier Modelsim an, das dem Webpack beiliegt. Anhand der Datenblätter[32] der verwendeten Komponenten haben wir in VHDL Beschreibungen der Struktur und des zeitlichen Verhaltens erstellt. Da die CPLDs

[32] ECA Datenlexikon - 74er digital

für ihre Programmierung[33] ebenfalls mit VHDL beschrieben werden, kann diese für die Simulation der ALU und des Steuerwerks übernommen werden. Als Beispiel (Listing 2.3) sei hier ein Auszug aus der Hardwarebeschreibung eines Registers aufgeführt:

Listing 2.3: VHDL Beschreibung eines Registers

```
1  library IEEE;
2  use IEEE.STD_LOGIC_1164.ALL;
3  use IEEE.STD_LOGIC_ARITH.ALL;
4  use IEEE.STD_LOGIC_UNSIGNED.ALL;
5
6  entity Registerelement is
7      Port ( Eingang : in std_logic_vector(7 downto 0);
8             Ausgang : out std_logic_vector(7 downto 0);
9             RE : in std_logic;
10            WE : in std_logic);
11 end Registerelement;
12
13 architecture Behavioral of Registerelement is
14 begin
15
16 Process(RE,WE, Eingang)
17
18 variable Puffer : std_logic_vector(7 downto 0);
19 variable Aus : std_logic_vector(7 downto 0);
20
21 begin
22
23         IF(RE = '1') THEN
24                 Aus :="ZZZZZZZZ";        -- highimp
25         ELSIF(RE = '0') THEN
26                 Aus := Puffer;           -- output
27         END IF;
28
29         IF(WE = '1') THEN
30                 Puffer := Eingang;       -- read
31                 IF(RE = '0') THEN
32                         Aus := Puffer;   -- output
33                 END IF;
34         END IF;
35
36 Ausgang<=Aus after 15 ns;
37
38 end process;
39
40 end Behavioral;
```

[33] Kapitel 2.2.5

Desweiteren habe wir ein Testbench in Form einer VHDL-Datei erstellt, in dem nun alle äußeren Stimuli festgelegt werden, die auf das Modell wirken. Diese äußeren Stimuli sind die eben schon erwähnten Signale, die in der Schnittstelle zwischen Hardwarssimulation und Softwaresimulation auftreten. So werden Zeitpunkt und Intervall des Reset- und Taktsignals, sowie der Inhalt der ROMs festgelegt.

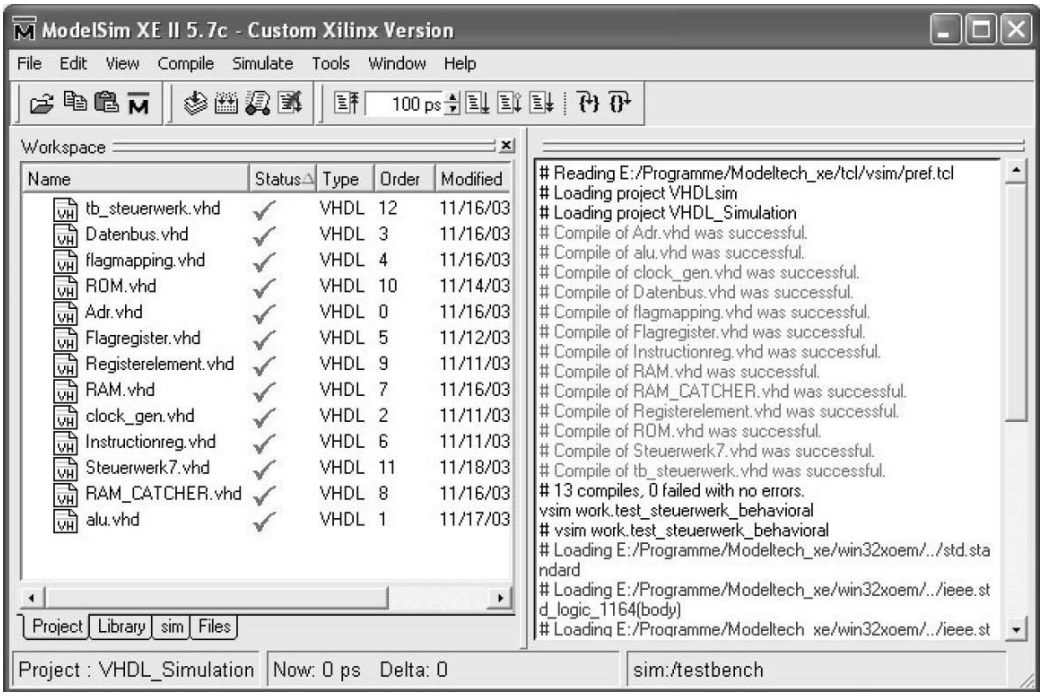


Abbildung 2.6: Projekt in Modelsim

Die VHDL-Dateien werden nun in Modelsim, wie in Abbildung 2.6 zu sehen, zu einem Projekt zusammengefasst. Modelsim generiert darauf hin nach unseren Eingaben und Beschreibungen ein Waveform, das eine Analyse der Funktionalität des Systems ermöglicht. In einem solchen Waveform sind alle Signale in ihrem binären Zuständen dargestellt. Ein solches Waveform ist in Abbildung 2.7 zu sehen. Einige Fehler konnten leicht entdeckt werden, da sie zu nicht erwarteten Ausgaben führten. Andere Fehler waren nur durch genaues Nachvollziehen der Waveforms zu entdecken. Eine Korrektur des Systems war nur durch Veränderung der VHDL-Beschreibung der ALU und des Steuerwerks möglich, da die Verhaltensweisen der Register, des RAMs und des ROMs fest durch ihr reelles Verhalten in der Hardware vorgegeben sind.

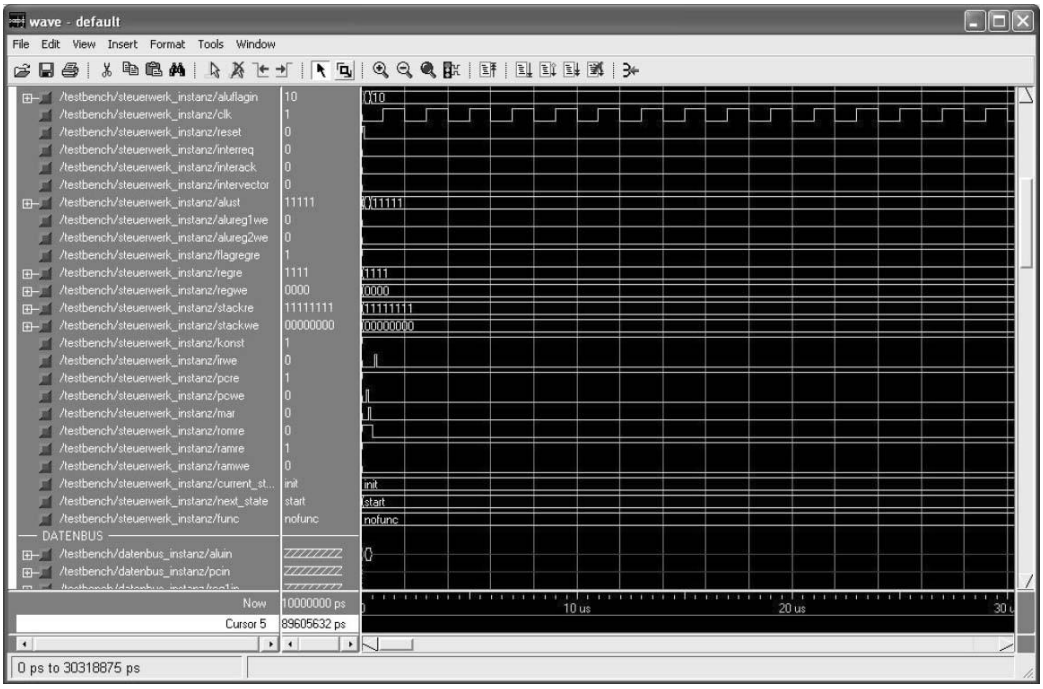


Abbildung 2.7: Waveform in Modelsim

Als in der Simulation alle Befehle des Befehlssatzes erfolgreich simuliert waren, konnte in einem Abschlußtest ein komplexes Programm in das ROM geschrieben werden. Dieses Programm lief einwandfrei. Somit war die Simulation mit VHDL erfolgreich beendet. Neben der Simulation erhielten wir ein korrigiertes Steuerwerk und eine ALU, die dann auch später in der Hardware verwendet wurden.

Aufbau von Testboards

Die beiden Mikrocontroller und ihre zugehörige Komponente, sowie ein CPLD wurden in Hardware aufgebaut, um ihre Funktionalität zu testen.
Die Anzeigematrix wurde zusammen mit dem Mikrocontroller und den zugehörigen Elementen (Quarzgenerator, Widerstände usw.) auf eine Lochrasterplatine aufgelötet und mit Kabel auf der Rückseite nach dem Schaltplan verdrahtet. Einzelne LEDs der Anzeige wurden mit Labview und den Breakout-Boxen angesprochen und sie funktionierten. Danach wurde der Mikrocontroller eingesetzt und mit einem kleinen Testprogramm geladen, welches einzelne LEDs ansprechen sollte. Dabei wurde festgestellt, dass die Verschaltung nicht das gewünschte Resultat lieferte, die Anzeige leuchtete zu schwach. Der Mikrocontroller lieferte nicht genug Strom, um die einzelnen LEDs der Anzeigematrix hell genug zu erleuchten. Die

Schaltung wurde geändert, indem ein Treiberbaustein UDN2585A zwischen Mikrocontroller und LEDs eingesetzt wurde. Dieser liefert einen viel höheren Strom, als der Mikrocontroller. Dieses Verhalten zeigt, daß eine Softwaresimulation der Signalpegel nicht ausreichend ist.

2.2.7 Vom Prototypen zum Board

Routen mit Eagle

Um Zeit zu sparen wurde mit den Vorbereitugen für das Routen des Boards begonnen, obwohl der Schaltplan noch nicht fertiggestellt war. Die Aufgabe war zunächst die 262 Bauteile auf einer vorgegebenen Fläche sinnvoll zu platzieren. Dabei sollten der Aufbau nicht nur übersichtlich, sondern aus didaktischer Sicht auch einfach und anschaulich sein. Das Ziel beim Platzieren ist einen günstigen Kompromiss zwischen mechanischen, elektrischen und thermischen Anforderungen zu schaffen und darauf zu achten, dass beim Routen die Airwires möglichst kurz sind und sich nicht kreuzen.

Bei der Wahl der Platinengröße galt es sich zwischen DIN A4 (210 x 279 mm) und DIN A5 (148 x 210 mm) zu entscheiden, da das Board in ins Handbuch geheftet werden soll. Wir haben uns besonders aus didaktischen Gründen für die zweite Variante entschieden.

Auf der Unterseite werden Bauteile plaziert, die nicht zum Kern des PoCiLet-Computers gehören, wie z.B. der Stack. Dadurch wird die Oberfläche übersichtlich gehalten, wodurch der PoCiLet- Computer leichter zu erfassen ist.

Die Abbildung 2.8 zeigt, dass sich die Oberseite räumlich in 3 Abschnitte aufteilen lässt:

1. Verarbeitung
   Der linke Abschnitt wird von den beiden CPLD's (ALU und Control Unit) gebildet werden.

2. Status
   Der mittlere Abschnitt besteht aus den Registern, den jeweils dazugehörigen 8 LED's und einem zweier Block aus „Write" und „Read" LED's. Die Reihe der LED's stellt den Datenbus dar.

3. I/O
   Der rechte Abschnitt beinhaltet Ein-/ und Ausgabe, bestehend aus dem Tastenfeld und den beiden Anzeigen

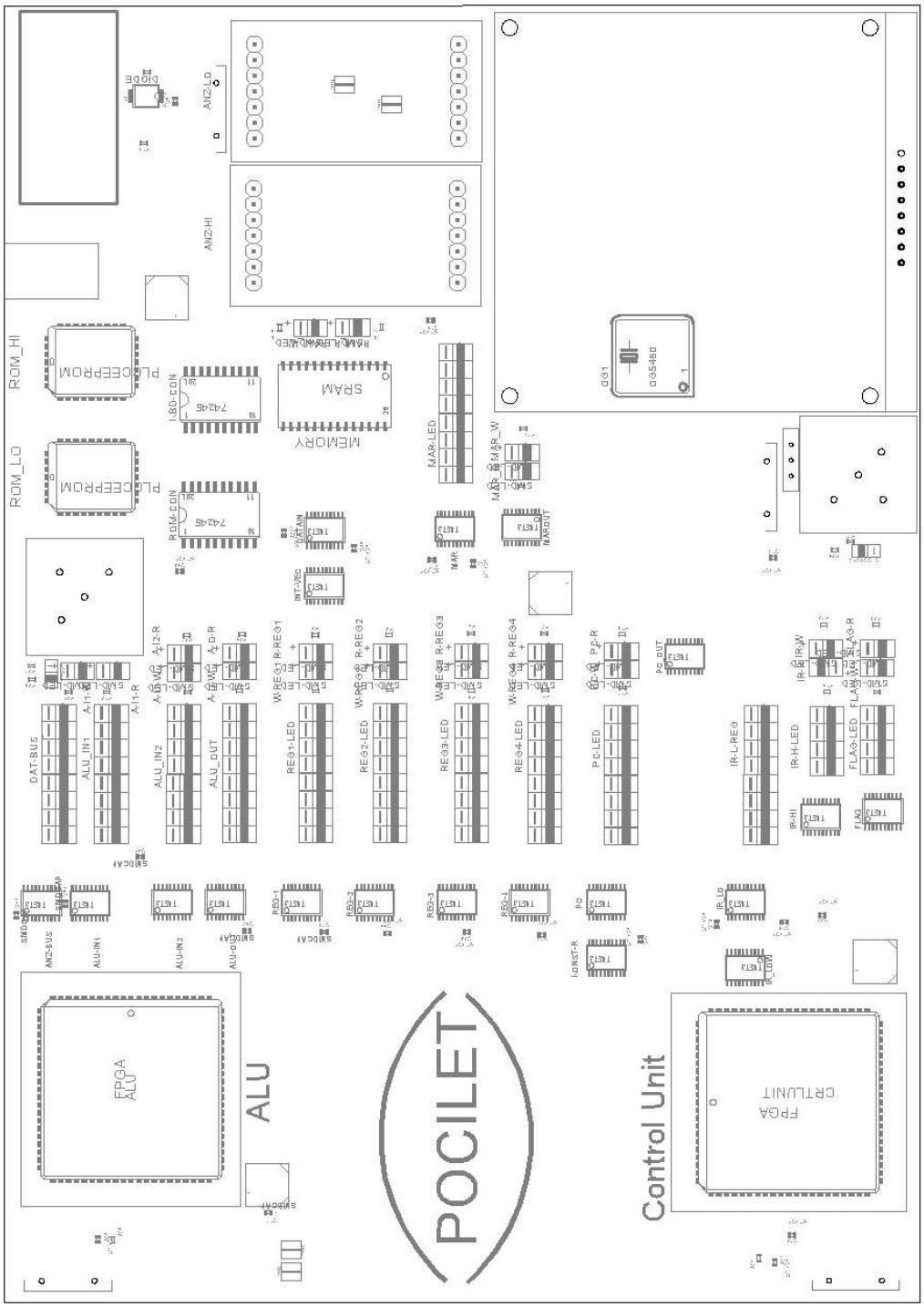


Abbildung 2.8: Oberseite des PoCiLet-Boards

Die Unterseite ist nicht so dicht bestückt wie die Oberseite. Hier ist die Logik platziert, sowie der Stack.
Zum besseren Routen und Einsparen von Vias wurden die meisten Widerstände auf der Unterseite platziert. Dies sollte beim späteren Routen ein wenig mehr Freiheit schaffen.

Nun konnte mit dem eigentlichen Routen der 1545 Airwires zu kreuzungsfreien Leiterbahnen begonnen werden. Hierbei werden aus Airwires Leiterbahnen mit realen Wegen, Breiten, Lötaugen und Durchkontaktierungen zwischen den beiden Layern. Eine einheitlich Vorzugsrichtung auf Unter-, bzw. Oberseite, wie anfänglich geplant, ließ sich aufgrund des Platzmangels leider größtenteils nicht realisieren. Abschließend ist es möglich, mit einem Design Rule Check (DRC) eventuelle Fehler, wie z.B. zu nah beieinander liegende Leiterbahnen oder zu eng an den Platinenrand plazierte Bauteile, festzustellen und anschließend zu korrigieren.

Schwierigkeiten traten immer wieder auf, da die Bearbeitung des Schaltplans zu Beginn des Routens nicht abgeschlossen war. Auch traten stets Fehler auf, die sowohl im Schaltplan als auch auf dem Board verändert werden mussten. Aufgrund der Konsistenz zwischen Schaltplan und Board wirkte sich jede Veränderung im Schaltplan direkt auf das Board aus. Es erschwerte das Routen insofern, dass stets Bauteile hinzugefügt und neu positioniert werden mussten. In unglücklichen Fällen führte das immer wieder dazu, bereits geroutete Leiterbahnen zu lösen und wieder neu zu routen. Auch aufgrund anfänglich fehlender Fachkenntnisse mussten bereits geroutete Leiterbahnen neu geroutet werden. Hinzu kam, dass wir zunächst davon ausgingen, dass die minimale Leiterbahnstärke bei 8 mil (1mm $\triangleq$ ca. 40 mil) lag. Der Wert konnte jedoch auf 6 mil korrigiert werden, was, minimal, mehr Spielraum beim Routen zur Folge hatte.

Autorouter vs. manuelles Routen

Eagle bietet die Option des Autorouters. Der Autorouter errechnet mit einem Algorithmus anhand von Verbindungslisten, der Pin-Geometrie der einzelnen Bauteile und zusätzlich vorgegebenen Parametern Wege für die Leiterbahnen. Der Autorouter sucht optimale Wege, wobei bereits verlegte Leiterbahnen durchaus automatisch entfernt und neu platziert werden können.

Die folgende Tabelle 2.2 soll die zahlreichen Versuche zeigen, welche Kriterien eine entscheidende Rolle spielten und welche Varianten zum eigentlichem Ziel führten. Letztendlich zeigt die Tabelle, dass die Ergebnisse des Autorouters nicht zufriedenstellend waren und wir das Board somit vollständig per Hand geroutet haben!

Variante	B∗L (mm)	verbleibende Airwires	Auto-router	Bemerkung
1	148 x 210	ca. 1700	-	starke Vorzugsrichtung nur Autorouter
2	148 x 210	510	70%	nur Autorouter
3	158 x 210	31	97%	die Hälfte per Hand geroutet, Rest Autorouter
...	...	...	...	
n-1	148 x 210	61	96%	die Hälfte per Hand geroutet, Rest Autorouter
n	148 x 210	0	-	komplett mit Hand geroutet; Endergebnis

Tabelle 2.2: getestete Varianten im Vergleich

Für kleine Platinen mit wenigen Bausteinen und viel Freiraum, scheint der Autorouter eine gute und schnelle Lösung zu sein. Für das PoCiLet-Board erwies sich der Autorouter aber als nicht sehr hilfreich, da das Board zu komplex war und der Autorouter meist die letzen 30 % nicht routen konnte.
Die Möglichkeit einer Kombination aus manuellem Routen und Autorouter zu verwenden, führte auch zu keiner zufriedenstellenden Lösung.
Im Endeffekt haben wir uns entschieden das Layout komplett manuell zu entwerfen, was zwar mehr Zeit in Anspruch nahm, aber unumgänglich war.

Das letzte Airwire

Das Routen der letzten Leiterbahnen erwies sich als am schwierigsten. Da der Freiraum überwiegend ausgeschöpft war, blieb meist nur noch der (Um-) Weg über den Aussenrand. In vielen Fällen mussten geroutete Leiterbahnen dabei umpositioniert werden, um neue Wege zu schaffen, was mit viel Aufwand verbunden war. Die Abbildungen 2.9 und 2.10 zeigen das derzeitig fertig geroutete PoCiLet-Board!

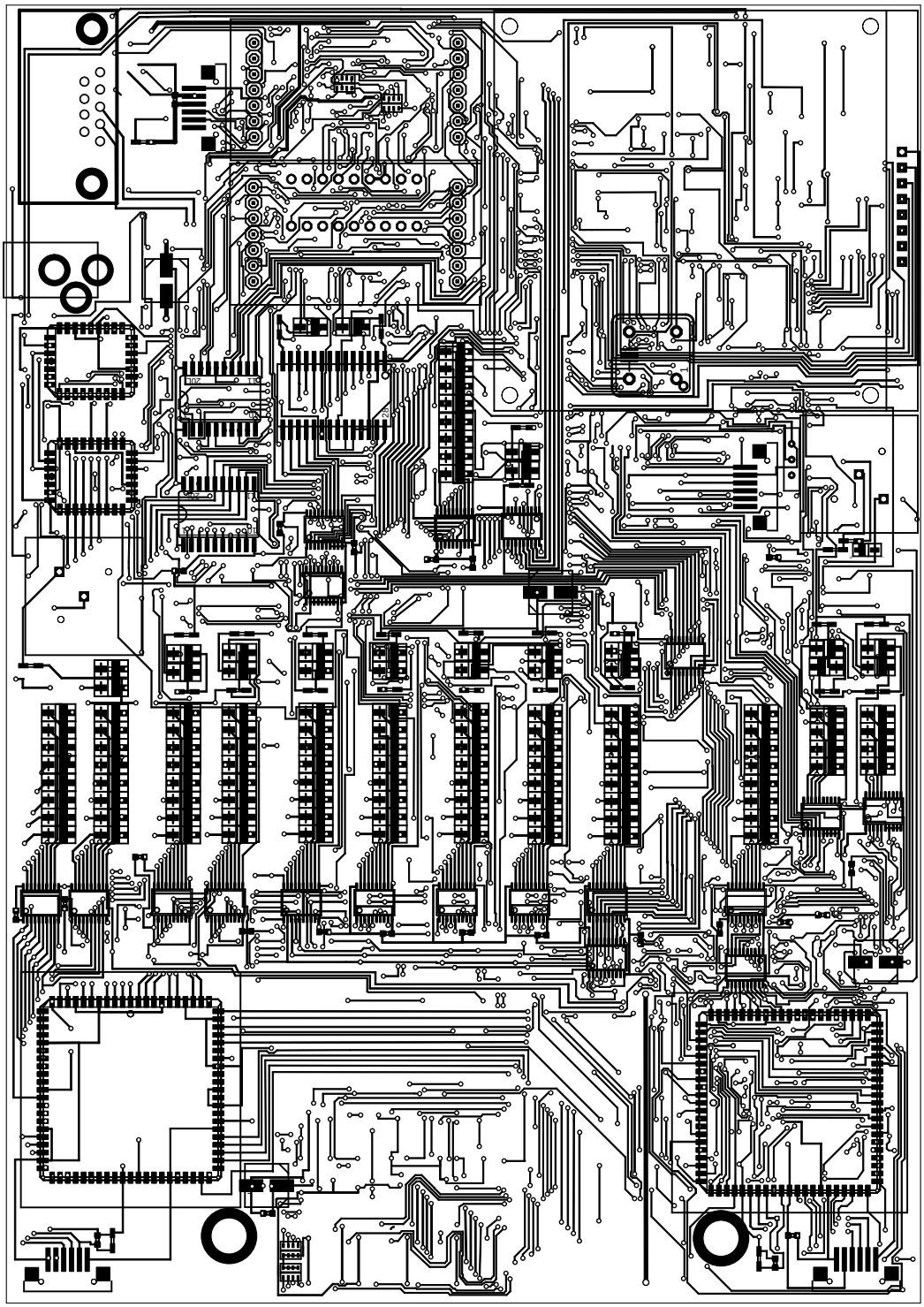


Abbildung 2.9: Das fertig geroutete PoCiLet-Board - Front

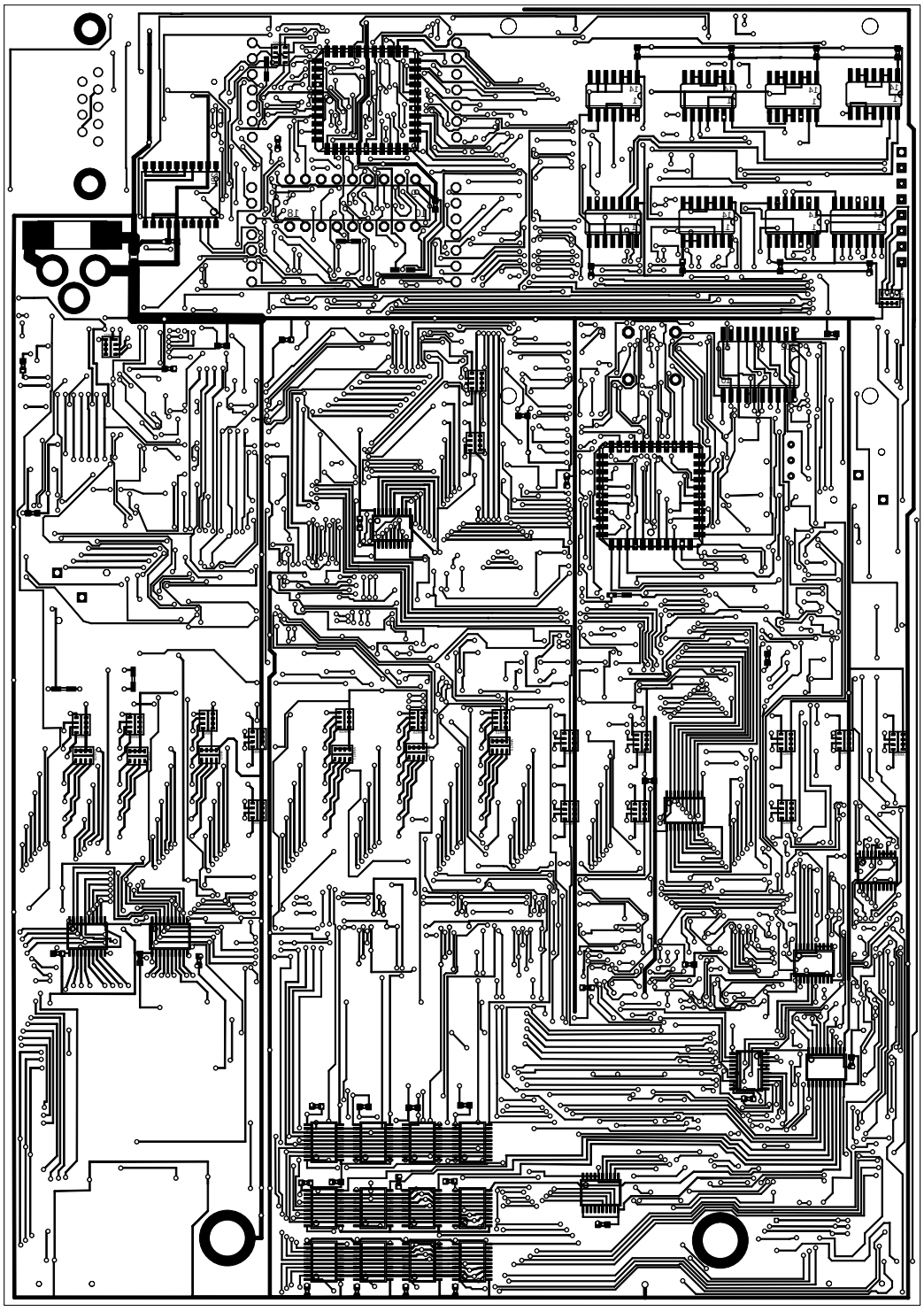


Abbildung 2.10: Das fertig geroutete PoCiLet-Board - Back

2.2.8 Fertigstellung der Platine

Vor dem Aufbau der Platine überlegten wir uns eine Strategie, um die Platine in Betrieb zu nehmen. Dabei wurde berücksichtigt, dass alle Bauteile in einer gewissen Reihenfolge auf die Platine gelötet werden, damit die einzelnen Bauteile getestet werden können. So können mögliche Fehler frühzeitig erkannt und erfolgreich behoben werden. Die Schaltung kann sofort im Schaltplan und im Board Layout geändert werden. So wird immer eine aktuelle Version der Schaltung und des Board realisiert und aufrecht gehalten, damit bei einer erneuten Platine, die alten Fehler nicht mehr aktuell sind und ein funktionales PoCiLet Board existiert.

Da manche Bauteile der Platine, besonders die Register und Widerstandarrays sehr klein waren, wurden die Bauteile unter einem Stereomikroskop aufgelötet. Die ersten Register wurden mit Lötpaste und einem Lötföhn auf die Platine gebracht. Es stellte sich aber heraus, dass es einfacher und sicherer ist, die Bauteile unter der Lupe mit Lötzinn und Lötkolben auf die Platine zu löten. Die Lötpaste konnte mit dem Fön nicht sicher genug verlötet werden, und Reste der Lötpaste bildeten Kurzschlüsse an den einzelnen Beinchen der Bauteile. Durch die Handverlötung konnten diese Kurzschlüsse vermieden werden und es bildeten sich keine unsichtbaren Kurzschlüsse unterhalb der Bauteile mehr.

Der Zusammenbau der Platine erfolgte auf einem ESD[34] sicheren Arbeitsplatz mit geerdeten Lötkolben. Die Inbetriebnahme erfolgte zunächst mit einem Labornetzteil. Der Strom wurde begrenzt, um im Fall eines Kurzschlusses die Schaltung nicht zu zerstören.

Zuerst wurde die Spannungsversorgung für die Platine bereitgestellt und die Sicherung und die großen Ablockkondensatoren auf die Platine gelötet. Danach wurde an relevante Stellen die Spannung auf der Platine geprüft und dabei wurde festgestellt, dass die beiden CPLDs nicht richtig auf der Platine verdrahtet waren. Das Layout der Bauteil war falsch in EAGLE nummeriert und so waren viele Leitungen der CPLDs falsch auf der Platine. Durch Änderung des Programms der einzelnen CPLD, die Leitungen wurden auf andere Ports des CPLDs umgeleitet, waren nur noch wenige Leitungen falsch. Diese wurden auf der Platine durchtrennt und mit dünnen Drähten mit den richtigen Stellen verbunden.

Anschließend wurden die ersten Register mit den Anzeigeregistern und LEDs aufgelötet. Zusätzlich die Sockel der CPLDs und die Treiberbausteine der CPLDs. Danach wurde die Schaltung mit Labview und der Breakout-Box getestet. Einzelne Register konnten gelesen und beschrieben werden. Da die Register in hochoh-

[34] Electro-Static Discharge

mige Zustände wechseln, wenn ihre Ausgänge inaktiv geschaltet werden, konnte sich der Datenbus nicht schnell entladen und zeigte die letzten Werte lange an. Um dies zu verhindern, haben wir zusätzliche Pulldown Widerstände an den Datenbus gelegt, damit der Datenbus die Werte löschen kann. Danach wurden die restlichen Register aufgelötet und getestet.

Anschließend wurde die Anzeigematrix mit dem Treiberbaustein und Mikrocontroller, inkl. Quarzgenerator, aufgelötet. Dabei wurde festgestellt, dass einzelne Widerstandarrays unter dem Sockel des Mikrocontroller lagen. Um dieses Problem zu lösen, musste der Sockel ein wenig ausgefräst werden, damit die Widerstandarrays Platz hatten und nicht den Mikrocontroller störten. Nachdem alle Bauteile aufgelötet waren, wurde der Mikrocontroller mit einem Testprogramm geladen und die Anzeigen leuchteten und zeigten den gewünschten Wert an. Danach wurde das endgültige Programm auf den Mikrocontroller geladen und getestet. Die Anzeigen zeigten wiederum den gewünschten Hexadezimalenwert an.
Im nächsten Schritt wurden die Logikbausteine des RAM, des ROM und der Tastatur, und zusätzlich das RAM und die Sockel für die ROMs und Tastatur aufgelötet.
Der zweite Mikrocontroller wurde zusammen mit den Treiberbausteinen und dem MAX233, der für die serielle Datenübertragung zuständig ist, aufgelötet und in Betrieb genommen. Es wurde ein Testprogramm auf den Mikrocontroller geladen, um die serielle Datenübertragung zu testen. Sie funktionierte einwandfrei und Daten konnten zwischen dem PoCiLet-Board und einem Terminalprogramm auf einem Personalcomputer ausgetauscht werden.
Die ALU (CPLD) wurde mit einem kleinen Testprogramm beschrieben, um zwei binär Zahlen mit einander zu addieren und die Zahlen ins Ausgangsregister zu schreiben. Durch eine Breakout-Box wurden zwei Zahlen in die ALU-Eingangsregister geschrieben und die ALU addierte die zwei Zahlen korrekt mit einander.
Danach wurden die restlichen Bauteile aufgelötet.

2.3 Der PoCiLet-Simulationskern

In diesem Kapitel wird die Realisierung des Simulationskerns beschrieben.
Im ersten Unterkapitel wird in das Thema Simulation eingeführt. Das zweite Unterkapitel stellt das Konzept des Simulationskern vor. Im dritten Unterkapitel wird die Realisierung des Konzeptes anhand eines kleinen Flipflop-Beispiels beschrieben. Im letzten Unterkapitel geht es um den Entwurf. Zunächst wird der Entwurf des Simutionskerns anhand von UML-Diagrammen (z.B. Klassendiagramm) dargestellt. Zum Schluss wird auf die Realisierung der wesentlichen Packages, die für die Simulation zuständig sind, eingegangen.

2.3.1 Einleitung

Ein Modell stellt ein Abbild eines Systems dar. Oft ist ein System zu komplex, um es gedanklich vollständig zu erfassen und zu untersuchen.

Hertz definiert die Entscheidungsmodelle wie folgt:

> *"Wir machen uns innere Scheinbilder oder Symbole der äußeren Gegenstände, und zwar machen wir sie von solcher Art, dass die denknotwendigen Folgen der Bilder stets wieder die Bilder seien von den naturnotwendigen Folgen der abgebildeten Gegenstände"*[35]

Unter einer Simulation versteht man den Prozess der Bildung einer Prognose mit Hilfe des Experimentierens innerhalb der Modellebene, also die Durchführung von Versuchen bzw. (Hoch-)Rechnungen in einem abstrakten Modell eines Systems. Ziel einer Simulation ist die Analyse des (zukünftigen) Systemverhaltens.

Die Simulation soll in unserem Fall dazu dienen, den Lernenden einen Einblick in die Hardware zu ermöglichen, um somit das Funktionsverhalten besser zu verstehen. Sie spiegelt den strukturellen Aufbau des Minicomputers wieder und umfasst den Entwurf der Bauteile.
Das Objekt unserer Simulation ist ein selbst erstellter Pocketcomputer, welcher nicht zu kompliziert jedoch den Minimalansprüchen gerecht ist.

Ziel ist es diesen Minicomputer auf der Registertransferebene, mit Verweis auf die anderen drei Ebenen, zu simulieren. Am Anfang der Entwicklung wurde zunächst einmal ein Konzept erstellt, um die Funktionalität der Simulation besser zu verstehen. Die Richtigkeit dieses Konzepts wurde nun anhand eines einfachen Flipflop-Beipiels überprüft. Der Einstieg über das Flipflop-Beispiel hat dazu beigetragen,

[35] http://www.learn-line.nrw.de/angebote/modell/aquarium/glossar.htm

dass Konzept zu verfeinern und einige Verbesserungen für die eigentliche Umsetzung vorzunehmen. Wie auch in dem nachfolgenden Abschnitt zu erkennen ist, sind bei der Umsetzung einige Klassen entfallen, welche in dem Einstiegsbeispiel noch vorhanden waren. Hier wurde das Konzept nach den Erfahrungen, die man während der Implementierung des Beispiels gemacht hat, und teilweise auch aus didaktischen Gründen, verbessert.

2.3.2 Konzept

Der Simulationskern stellt das Softwaremodell der Hardwarekomponenten des Minicomputers dar. Dieses Modell beschreibt den Minicomputer hinsichtlich seines Verhaltens.

Der im Rahmen des Projektes aufgebaute Minicomputer soll simuliert werden. Der Originalschaltplan des Rechners wurde nicht eins zu eins in der Software übernommen, da die Simulation nur auf der Register-Transfer-Ebene erfolgen sollte. Der Schaltplan enthält zusätzliche Komponente die nicht zur Register-Transfer-Ebene des Minicomputers gehören und erst auf der Schaltkreis-Ebene hinzukommen. Beispielsweise können hier die LEDs (Leuchtdioden) genannt werden, da diese für das eigentliche Verhalten des Minicomputers nicht relevant sind, sondern aus didaktischen Gründen verwendet werden müssen, um den Benutzer zu veranschaulichen, dass an den Leitungen im PoCiLet-Board Werte anliegen bzw. sich ändern. Im Grunde werden alle Komponenten bei der Simulation berücksichtigt, die für die didaktische Veranschaulichung der Funktionsweise des Minicomputers selbst benötigt werden.

In der PoCiLet-Hardware befinden sich keine Rückkopplungen, d.h. es gibt keine Zyklen innerhalb der Schaltung, die eine Berücksichtigung von Verzögerungszeiten der Bauteile in der Software erfordern würden. Aus diesem Grund kann dieses einfache Simulationsmodell auch auf die PoCiLet-Schaltung angewendet werden.

Simulation auf Register-Transfer-Ebene

Die softwaremäßige Simulation des Projektes beschränkt sich auf das Nachbilden und Berechnen der Register-Transfer-Ebene, von wo aus dann die weiteren Abstraktionsebenen abgeleitet werden können. Diese drei weiteren Ebenen sind, mit zunehmender Abstrahierung, die Schaltkreis-Ebene, die Layout-Ebene und die 3D-Ebene (siehe Kapitel 1.5 und Kapitel 2.4). Diese müssen, wie erwähnt, nicht explizit simuliert werden, die Simulationsergebnisse aus der Register-Transfer-Ebene müssen lediglich auf die anderen Ebenen übertragen und dort dargestellt werden. So wird sichergestellt, dass unnötiger Rechenaufwand, um die gleichen

Bauteile	*Anzahl*	*Bitbreite*	*Signale(Anzahl)*
REGISTER	4	8	18
STACK	8	8	18
PROGRAMMCOUNTER	1	8	18
KONST-REGISTER	1	8	18
MEMORY-ADRESS-REG	1	8	18
ALU-E1	1	8	18
ALU-E2	1	8	18
DATAIO(INPUT/OUTPUT)	1	8	18
FLAG-REG	1	4	10
INSTRUKTIONS-REG	1	12	26
MEMORY	1	[255*8 Bit]	26
ROM	1	[256*12 Bit]	21
ALU	1	8	33
AUTOCLOCK	1		1
RESET	1		1
CLOCK	1		1
STEUERWERK	1		1

Tabelle 2.3: Liste der simulierten Bauteile

Bauteile mehrfach zu simulieren, entfällt und die Simulation sich auf die wesentlichen Komponenten beschränken kann.
Die Simulation auf der Register-Transfer-Ebene erfolgt auf Basis der Bauteile, die für die Funktionalität des Minicomputers in ihrem Eingangs-Ausgangs-Verhalten benötigt werden. Nur die Komponenten, die zur Register-Transfer-Ebene gehören und bei der Abarbeitung eines Befehls relevant sind, sind hinsichtlich der Didaktik sinnvoll und werden simuliert. Komponenten die aus diesem Grund weggelassen werden sind z.B. die LED´s (Leuchtdioden) und das zweite PC-Register. Das zweite PC-Register wurde eingeführt damit die Werte des PC sowohl auf den Datenbus, als auch auf dem Programm-Adressbus gelegt werden können und gehört nicht zur RT-Ebene.
Die Tastatur (DATAIO) dient nur zur Ein- und Ausgabe durch die Oberfläche und wird nicht simuliert. Die Bauteile, die in der Simulation betrachtet werden, sind in der Tabelle 2.3 aufgelistet.

Für jede Komponente werden die Ausgänge solange berechnet, bis ein stabiler Zustand erreicht ist. Der stabile Zustand entspricht dann dem Ende eines Simulationsschrittes.

Das Ergebnis der Simulation wird gespeichert und auf die anderen Ebenen, die durch die GUI definiert sind, übertragen. Ein Simulationsschritt wird von einem Taktsignal der GUI eingeleitet. Die während den einzelnen Simulationsschritten abzuarbeitenden Befehle (in binärer Form) werden dem Simulator als ganzes Programm von der GUI geliefert, wie auch die einzelnen Taktsignale.

In Abbildung 2.11 kann man das Simulationsprinzip erkennen. Dem Konzept des PoCiLet-Simulationskerns liegt eine zweidimensionale Listenstruktur zugrunde. Zunächst einmal haben wir eine Liste, die die einzelnen Zwischenschritte(steps) eines Simulationsschrittes beinhaltet. Diese Liste (in Abbildung 2.11 horizontal dargestellt) wird im Folgenden StepList genannt. An jedem dieser Zwischenschritte hängt eine weitere, EventList genannte, Liste.

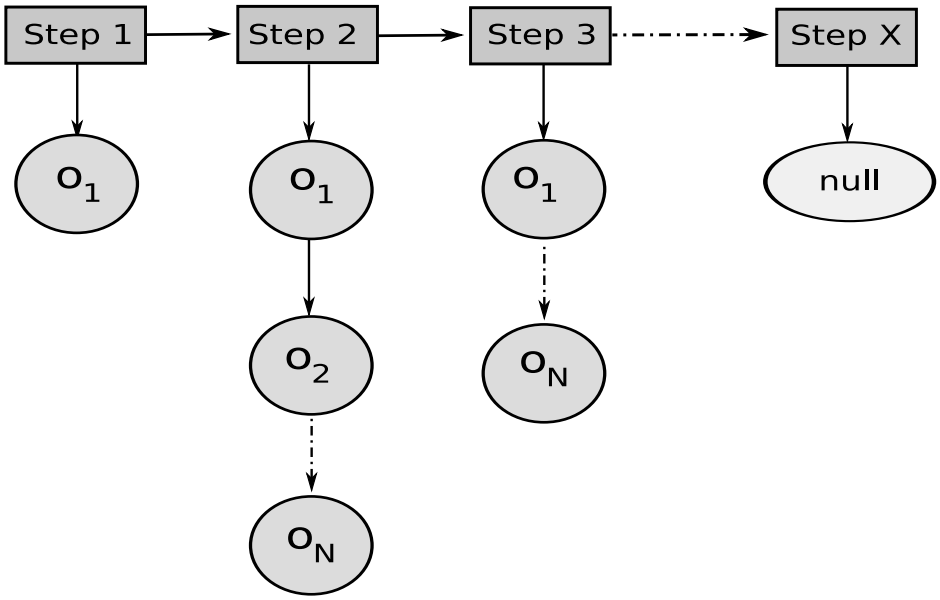


Abbildung 2.11: Ablauf eines Simulationsschrittes

Die Simulation funktioniert nun wie folgt: Am Anfang eines Simulationsschrittes werden alle Bauteile in die EventList des ersten Zwischenschrittes der StepList eingefügt.

Nun wird nach und nach jeder Zwischenschritt abgearbeitet. Hierbei wird die EventList des jeweiligen Zwischenschrittes durchlaufen und die darin enthaltenen Bauteile werden abgearbeitet. Jedes Bauteil muss seine Ausgangswerte und, falls vorhanden, seinen inneren Zustand, anhand der momentan anliegenden Eingangswerte neu berechnen. Haben sich die Ausgangswerte des aktuellen Bauteils ge-

ändert, müssen alle Bauteile, die an geänderten Leitungen angeschlossen sind, in die EventList des nächten Zwischenschrittes eingefügt werden, da sich deren Eingangswerte geändert haben. Wenn nun alle Bauteile, die sich in der EventList eines Zwischenschrittes befinden, abgearbeitet wurden, wird zum nächsten Zwischenschritt übergegangen.

Die Anzahl der Bauelemente, die während der Abarbeitung einer EventList in den jeweils nächsten Zwischenschritt eingefügt werden, konvergiert nun gegen Null. Auf diese Weise wird nach einigen Zwischenschritten ein Fixpunkt der Schaltung erreicht. Diesen erkennt man daran, dass man bei einem Zwischenschritt angelangt ist, in dessen EventList sich keine Bauteile mehr befinden. Das bedeutet, dass sich die Werte an den Ausgängen nicht mehr verändern. Somit ist die Abarbeitung des Simulationsschrittes beendet.

Die Richtigkeit dieses Konzeptes wurde, wie im folgenden Abschnitt beschrieben, anhand einer Flipflop-Testschaltung verifiziert.

2.3.3 Simulationseinstieg an einem Flipflop-Beispiel

Um das entworfene Konzept zu verifizieren, haben wir am Anfang versucht eine einfache Schaltung zu simulieren. Wir haben ein negiertes RS-Flipflop ausgewählt, welches sich aus 2 NAND-Gatter zusammensetzt (Abbildung 2.12).
Jedes Bauteil der Flipflop-Schaltung ist ein NAND-Gatter. Zur Simulation der Bauteile wurde die Klasse NandGate modelliert. Die Klasse Port definiert die Eingänge und Ausgänge des Bauteils. Die Verknüpfung der Bauteile wurde durch die Klasse Signal verwirklicht. Jedes Signal enthält einen Anfangs- und Endport, die identisch mit der Ports (IN, OUT) der Bauteile sind. Die Signale übertragen die logischen Werte, die durch die Klasse Logic definiert werden. Die Klasse Logic stellt die Attribute high, low und unknown zur Verfügung: high für Konstante **1**, low für Konstante **0**, unknown für Konstante **-1**.

Das Verhalten der NAND-Bauteile wird in der so genannten action() Methode der Klasse NandGate verwirklicht. Die Werte der Ausgangssignale werden an Hand der Eingangssignale in der Methode neu berechnet.
Die ganze Schaltung wurde in der Klasse Circuit durch die Methode buildCircuit() aufgebaut und initialisiert. Das heißt, für die benötigten Bauteile und Signale werden Objekte definiert und die Anfangs- und Endports der Signale werden auf die Ports der entsprechenden Bauteile gesetzt. Die Eingangs-, Ausgangssignale des gesamten Flipflops $\left(S, R, Q, \overline{Q}\right)$ werden mit sinnvollen Werten vorbelegt.

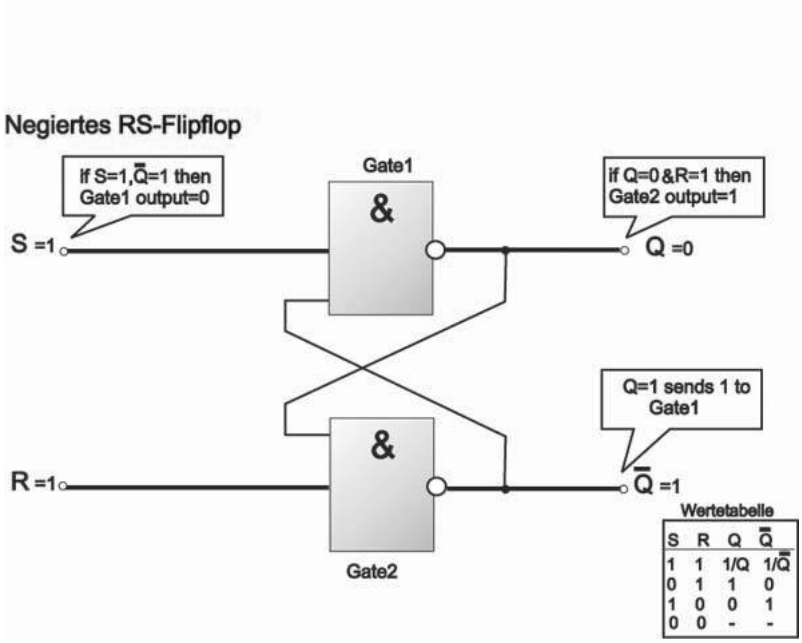


Abbildung 2.12: Negiertes RS-Flipflop

S	R	Q	$\overline{Q}$
1	1	1/Q	1/$\overline{Q}$
0	1	1	0
1	0	0	1
0	0	-	-

Tabelle 2.4: Wertetabelle des RS-Flipflop

Nachdem wir das Modell der Flipflop-Schaltung implementiert hatten, konnten wir anhand des Konzeptes das negierte RS-Flipflop simulieren:

In der sogenannten Klasse Simulator wird die Simulation gestartet und die Simulationsschritte abgearbeitet. Laut der Wertetabelle (Tabelle 2.4) werden die Eingangswerte des RS-Flipflops gesetzt (z.B. S=1 und R=0). Da es sich bei dem Flipflop um eine Schaltung mit Rückkopplungen handelt müssen die Ausgangssignalwerte vordefiniert sein und daher ist es wichtig, sie beim Simulationsstart mit sinnvollen Werten vorzubelegen. Die Ausführung jedes einzelnen Schrittes wird, wie im Konzept erklärt, in der Klasse Scheduler realisiert. Das heißt, beim Simulationsstart werden die zwei NandGates in die EventList des ersten Steps eingefügt. Nach der Berechnung der Ausgänge der NAND-Gatter wird überprüft, ob sich die Werte der Ausgangssignale geändert haben. Hat sich der Wert geändert, müs-

sen die Gatter, die an diesem Signal anliegen, in den nächsten Schritt eingefügt werden. Haben sich die Werte nicht geändert, wird an dieser Stelle nicht weiter simuliert. Die Liste des nächsten Schrittes enthält dann nur noch die Gatter, bei denen sich die Eingangssignale geändert haben und die daher ihre Ausgänge neu berechnen müssen.

Nach dieser Realisierung haben wir einen besseren Einblick in unser Konzept gewonnen. Es hat sich gezeigt, dass es im großen und ganzen funktioniert, aber noch verbessert werden kannte.

2.3.4 Entwurf

Simulationsentwurf in Java

Die Simulation erfolgt mit einem Java Komponentenmodell. Dieses Modell wurde aufgrund seiner Portabilität gegenüber anderen Plattformen bzw. Programmiersprachen gewählt. Alle wichtigen Eigenschaften von Java, als Objektorientierter Sprache, werden bei der Modellierung beachtet, um die Implementierung so weit wie möglich zu erleichtern. Zum Beispiel: Vererbung, Verwendung von Java-Bibliotheken zur Verwaltung von Listen, die in unserem Entwurf vorkommen, etc.

Jedes Bauteil wird im Prinzip als eine Klasse implementiert. Alle Bauteilklassen erben von der Oberklasse SimObject. Außerdem werden die Bauteile mit Ports ausgestattet, die mit Signalen verbunden werden, um die Verbindung mit anderen Bauteilen zu erstellen.

Für unser Modell haben wir eine dreiwertige Logik mit den Werten low, high und highimp gewählt. Hierfür haben wir einen eigenen Datentyp Logic erstellt, der eine variable Bitbreite hat und die Werte der einzelnen Bits zu zwei Integern lh (für high und low) und z (für highimp) zusammenfasst. So müssen logische Operationen nur auf den einzelnen Integern ausgeführt werden.

Signale beschreiben die Leitungen, die die Bauteile miteinander verbinden. Sie dienen zur Übertragung von Informationen (z.B. ein 8-bit codiertes Wort). Allerdings wird nicht jede Leitung einzeln implementiert, stattdessen hat ein Signal eine bestimmte Bitbreite. Auf diese Weise braucht man beispielsweise für ein 8bit breites Wort nicht acht Signalleitungen und acht Ports für die Übertragung. So wird in diesem Fall in Java statt acht Signal-Objekten und 8 Port-Objekten nur jeweils ein Objekt benötigt.

Ports sind Verbindungspunkte von Bauteilen, an die Signale gehängt werden. Genau wie die Signale, haben Ports eine variable Bitbreite.

Zusätzliches gibt es Vertices. Dies sind SimObjecte, deren einzige Funktionalität es ist, Signale aufzuspalten und weiterzuleiten.

Das Kernstück der Simulation ist der Simulator. Er ist einerseits die Schnittstelle zwischen dem PoCiLet-Simulationskern und der GUI, andererseits gehen von ihm alle Funktionen der Simulation in Java aus. Über diese Klasse kann z.B. die Simulation gestartet, ein weiterer Simulationsschritt ausgelöst oder ein Programm in den Simulator geladen werden.

Der Scheduler verwaltet die einzelnen Zwischenschritte, die innerhalb eines ganzen Simulationsschrittes ausgeführt werden. Jeder dieser Schritte (Step) enthält jeweils die SimObjekte, die innerhalb dieses Steps ihre action()-Methode ausführen müssen.

UML-Klassendiagramm

Abbildung 2.13: Klassendiagramm Package pocilet.simulation

Das Klassendiagramm des Simulationskerns ist in zwei Packages aufgeteilt. Zunächst einmal gibt es das Package simulation (Abbildung 2.13), das die Klassen beinhaltet, die die Simulation steuern, und das Sub-Package simobjects (Abbildung 2.14), in dem die Klassen für die einzelnen Bausteine, Ports und Leitungen enthalten sind.

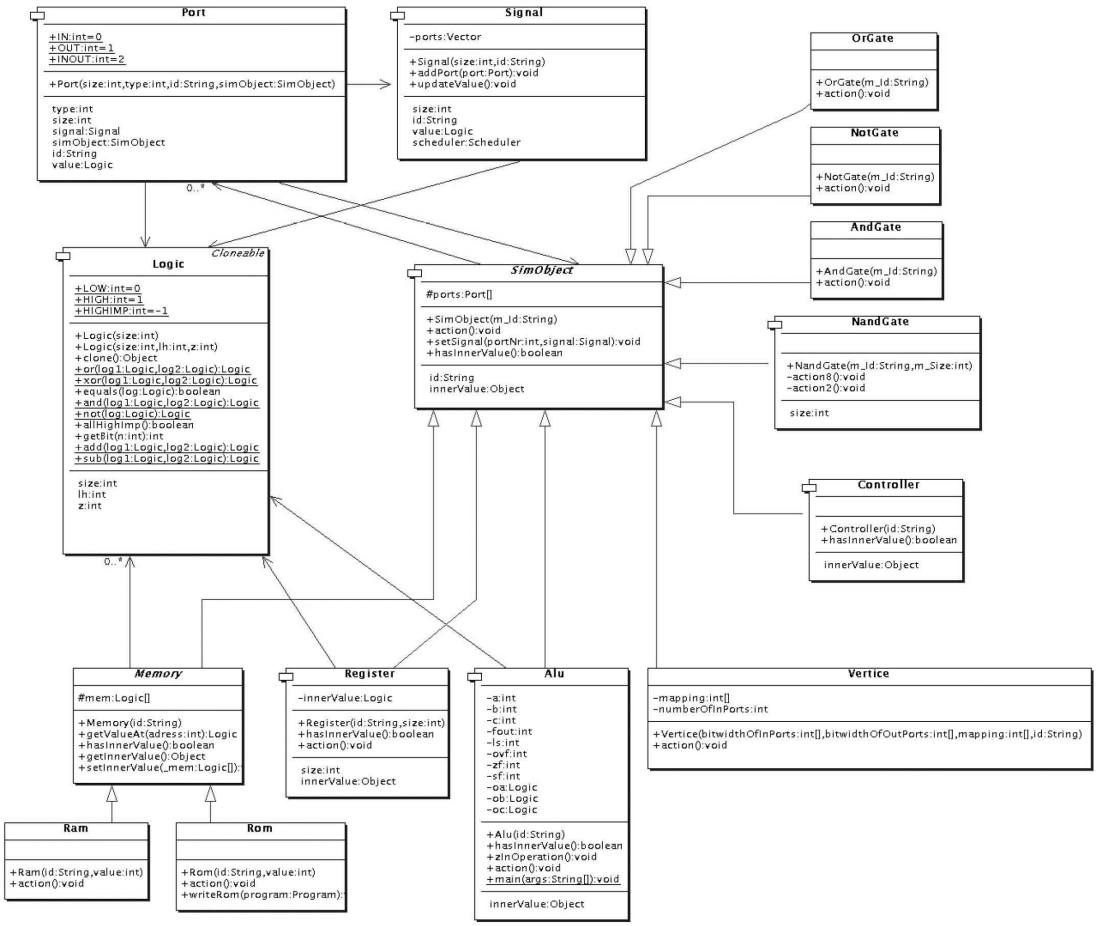


Abbildung 2.14: Klassendiagramm Package pocilet.simulation.simobjects

Das SimObject-Package

Wie im oberen Abschnitt aufgeführt worden ist, beinhaltet das Package Simobject sowohl die Klassen für die einzelnen Bauteile, als auch weitere Klassen, die untereinander assoziieren.
Es herrscht eine Klassenhierarchie, die eine Oberklasse aufweist, von der alle Bauteile, die auf der Hardware realisiert werden, ihre allgemeine Struktur erhalten. Die Subklassen weisen somit eine ähnliche Grundstruktur auf, wobei diese um ihre spezifischen Eigenschaften ergänzt werden.

Um dies genauer zu veranschaulichen, wird anhand der abstrakten Oberklasse SimObject die allgemeine Struktur dieses Packages erläutert:

Die abstrakte Oberklasse SimObject stellt Attribute zur Verfügung, die durch den Konstruktor der Klasse initialisiert werden. Das Attribut id, welches vom Typ String ist, wird beim Konstruktoraufruf initialisiert und stellt eine eindeutige ID für die jeweiligen Instanzen dar.
Als weiteres Attribut wird ein Array deklariert, dass Instanzen der Klasse Port enthält. Jedes Bauteil stellt eine unterschiedliche Anzahl von Ports zur Verfügung, an denen Leitungen in der entsprechenden Bitbreite angelegt werden. Dies wird jedoch erst in den Subklassen initialisiert, da die Anzahl von Instanz zu Instanz variabel ist.

Die Oberklasse weist Methoden auf, die in den Subklassen später überschrieben werden können oder müssen. Eine dieser, Methoden ist die action() Methode. Die action() Methode wird in den Subklassen überschrieben. Diese Methode führt das spezifische Verhalten der einzelnen Bauteile aus.

Weitere Methoden in SimObject sind hasInnerValue(), setInnerValue() und getInnerValue(). Zunächst einmal gibt die hasInnerValue() Methode wieder, ob ein innerer Zustand existiert. Gegebenenfalls wird diese Methode in den Subklassen mit dem Rückgabewert true überschrieben. Also haben die Subklassen, solange sie diese Methode nicht überschreiben, bei der Abfrage, ob ein innerer Zustand existiert, den Wert false anliegen. Die nächste Methode setInnerValue() dient dazu diesen inneren Wert manipulieren zu können und getInnerValue() gibt den Booleschen Wert wieder, welcher anliegt.

Dies ist soweit die Oberklasse SimObject, welche die allgemeine Struktur der Subklassen darlegt.

Weitere Klassen, die nicht von der Oberklasse SimObject erben, sind: Port, Signal und Logic.

Die Klasse Port stellt einen Verbindungspunkt zwischen der Signalleitung und dem Bauteil dar. Dieser setzt sich aus mehreren Attributen zusammen, die wie folgt lauten:

- size

  Das Attribut size ist ein Integer Wert, der die Bitbreite dieses Ports festlegt.

- signal

  Signal legt die jeweilige Signalleitung fest, die an diesem Port anliegt.

- value

  Das Attribut value gibt an, welcher Wert an diesem Port anliegt.

- type

  Type gibt an, ob dieser Port ein Eingangs-, Ausgangs- oder sowohl Eingangs- als auch Ausgangs-Port ist.

- id

  Die id ordnet dem Port einen eindeutigen Namen zu.

Diese Attribute werden durch den Konstruktoraufruf initialisiert, wobei die anliegende Leitung erst später durch die setSignal() Methode belegt wird.

In der Hardware wird die Codierung der Signalwerte binär dargestellt, jedoch werden die Signalwerte bei der Simulation, wegen der besseren Handhabung, durch Ganze Zahlen repräsentiert.
Binär gibt es drei Zustände, die auftreten können: low(0), high(1) und highimp (hochohmiger Zustand). Für die Darstellung der binären Zeichenkette, die aus Nullen und Einsen bestehen, wird stattdessen die Dezimalzahl lh dargestellt, ebenso existiert eine weitere Dezimalzahl z für den hochohmigen Zustand. Dieser Logik Wert setzt sich somit aus zwei Attributen vom Typ Integer zusammen.

Im Folgenden sind 2 Beispiele aufgeführt, wobei die Tabelle 2.5 einen Integer Wert ohne hochohmigen Zustand darstellt und die Tabelle 2.6 einen Integer Wert repräsentiert, wobei Teilleitungen einen hochohmigen Zustand aufweisen.

lh: low and high z : hochohmiger Zustand

Variable	Binär								Integer
lh	0	0	1	0	1	0	1	0	42
z	0	0	0	0	0	0	0	0	0
Anliegender Wert	l	l	h	l	h	l	h	l	

Tabelle 2.5: Beispiel für eine Instanz der Klasse Logic

In der Klasse Logic werden im Weiteren eine Menge von Methoden aufgeführt, die sowohl arithmetische Operationen und logische Verknüpfungen, als auch weitere Hilfsmethoden realisieren.

lh: low and high z : hochohmiger Zustand

Variable	Binär								Integer
lh	0	0	0	0	1	1	1	0	14
z	0	0	0	0	1	1	0	0	12
Anliegender Wert	l	l	l	l	z	z	h	l	

Tabelle 2.6: Beispiel für eine Instanz der Klasse Logic mit einem Anteil von hochohmigen Zustand

Das Package simulation setzt sich aus den folgenden Klassen zusammen:
Step, Scheduler, State, Simulator und dem oben aufgeführten Subpackage simObject.

Die Klasse Step hat eine Referenz auf eine eventList. Diese Referenz beinhaltet, wie schon im vorherigen Abschnitt beschrieben, am Anfang eine Liste aller Instanzen vom Typ SimObject, die in der Schaltung vorkommen. Die Liste wird durch die Methode processStep(), die in Scheduler implementiert ist, Schritt für Schritt abgearbeitet, indem die einzelnen Instanzen der Subklassen mit ihren jeweiligen action() Methoden aufgerufen werden. Ob eine Instanz in die eventList des nächsten Schrittes eingebunden wird, wird durch die Methode addObjToNextStep() realisiert, welche wiederum in der Klasse Signal aufgerufen wird. Wenn sich der Wert des Signals geändert hat und kein hochohmiger Zustand an der Leitung anliegt, werden die nachfolgenden Instanzen beim Scheduler hinzufügt.

Der Scheduler ist dafür zuständig, die Instanzen der Subklassen in die eventList einzubinden und die jeweilige action() Methode aufzurufen, bis die StepList keine Instanzen mehr enthält, also der Fixpunkt erreicht ist.

Nach der Abarbeitung eines Simulationsschrittes wird ein Zustand festgelegt, welcher auch von der GUI abgerufen wird, um ihn graphisch darzustellen. Dieser Zustand mit den benötigten Eigenschaften wird durch die Klasse State dargestellt. Für den Zustand sind zwei Hashmaps vorgesehen, die die SimObject- und Signalwerte beinhalten, und eine Variable Program, welche den Programmcode beinhaltet.

Die Klasse Simulator ist das Herz der ganzen Simulation. Sie ist die Schnittstelle zu der graphischen Oberfläche und zuständig für die Koordination der Simulation. Ihr Konstruktor ist so aufgebaut, dass bei der Initialisierung dieses Objekts sowohl die Signal- als auch die Objektliste übergeben werden. Hier werden die Attribute mit den Inhalten der beiden Listen belegt und eine Instanz der Klasse Scheduler gebildet. Weiterhin wird hier noch die Methode makeState() aufgerufen.

MakeState() initialisiert ein Objekt der Klasse State, durchläuft dabei die Liste aller Instanzen der Klasse SimObject und nimmt alle Instanzen, die einen inneren Zustand vorweisen, in die Hashmap auf. Zusätzlich wird die Signal-Liste durchlaufen und die Signale werden wiederum in die Hashmap des Objekts state eingebunden.

Zum Schluss wird nochmals zusammenfassend das Zusammenspiel der einzelnen Komponenten, bei der Abarbeitung eines Simulationsschrittes, in der Klasse Simulator geschildert:

Ein Simulationsschritt läuft wie folgt ab: Wenn der Simulator den Befehl bekommt, den nächsten Simulationsschritt auszuführen, werden im Scheduler zunächst alle vorhanden SimObjecte in den ersten Step eingefügt. Jedes dieser Objekte wird nun nacheinander abgearbeitet. Wenn nun ein Instanz mit seiner Berechnung fertig ist, legt es den/die neu berechneten Logic-Wert(e) an seinen/seine Ausgangsports an. Diese wiederum geben an die jeweiligen Signale weiter, dass dort ein neuer Logic-Wert berechnet werden muss. Im Signal wird nun der neue Wert berechnet und geprüft, ob sich der Wert des Signals überhaupt geändert hat. Wenn eine Änderung vorliegt, müssen alle SimObjecte, die an diesem Signal anliegen, in den nächsten Step eingebunden werden. Hat sich der Wert jedoch nicht geändert, ist in der Schaltung an dieser Stelle nichts passiert, werden die anliegenden Objekte nicht in den nächsten Step eingebunden. Dies darf aber nur geschehen, wenn sich auf dem jeweiligen Signal keine hochohmigen Anteile befinden. Eine Ausnahme hiervon sind Vertices. Sie werden nicht in den nächsten Step eingefügt, sondern direkt abgearbeitet, da sie ja nur Knotenpunkte darstellen, die in der Hardware gar nicht existieren. Wenn sich die Werte an den Vertices Eingängen ändern, folgt daraus, dass sich die Werte an den Ausgängen auch ändern. Somit werden die Instanzen, die mit den Vertices Ausgehen verbunden sind und die Werte sich geändert haben, in die nächste eventList eingebunden. Die Vertices werden auch abgearbeitet, wenn ein Wert, der an ihnen anliegt, einen hochohmigen Anteil beinhaltet.

2.4 Die grafische Benutzerschnittstelle

Die grafische Benutzerschnittstelle hat primär folgende Aufgaben:

Innerhalb der vier zuvor erwähnten Layoutsichten, stellt die GUI den Zustand der Hardware jeweils zu einem bestimmten Zeitpunkt dar. Das Zeitintervall zweier aufeinanderfolgender Zustände entspricht genau einem Taktzyklus, so wie er auch von der Hardware ausgeführt wird.

Der Zustand ist definiert durch die internen binären Daten signifikanter Bauelemente, welche für das rudimentäre Verständnis der PoCiLet-Architektur unerlässlich sind. Dies sind alle digitalen Bauelemente, welche in der Register-Transfer-Ebene vernetzt sind.

Der PoCiLet-Simulationskern, als weitere Softwarekomponente, simuliert gerade diese Abstraktionsschicht der realen Hardware, die RT-Ebene. Es besteht also eine Identität zwischen den zu simulierenden Hardware-Komponenten und denen, welche die GUI in der Register-Transfer-Ebene anzeigt.

Die drei weiteren Layoutsichten in der GUI sind mehr oder weniger weitere Interpretationen der RT-Ebene hin zur realen Hardware.

Die Schaltkreis-Ansicht entspricht dem Originalschaltplan, welcher benutzt wurde, um das Platinen-Layout der PoCiLet-Hardware zu bauen (siehe Kapitel 2.2.4, auf Seite 28). In der RT-Ebene sind bestimmte Bauelementgruppen ausgelassen, sofern sie nur für die elektrische Funktionsweise der Hardware relevant sind. Diese Ebene beschreibt nur das Ein- und Ausgangsverhalten der digitalen Bauelemente basierend auf einer boolschen Algebra. Hierzu zählen die ALU und das Steuerwerk, jeweils durch zwei FPGAs realisiert (siehe Kapitel 2.2.5), mehrere Registerbausteine und eine Tastatur (numerischer Ziffernblock). Diese Abstraktionsebene ist völlig unabhängig von der zugrunde liegenden technologischen Umsetzung. Beispielsweise ist es irrelevant, dass die binären Zustände „0” und „1” durch zwei verschiedene elektrische Spannungswerte realisiert werden. Des weiteren werden alle LEDs, welche auf der Hardware die internen Registerwerte darstellen, in der GUI nicht als Bauelemente umgesetzt, da man softwareseitig andere Visualisierungsmöglichkeiten hat (z.B. Farbkodierungen, dokumentierende Dialogelemente, usw.).

Die Layoutansicht zeigt dem Benutzer den nächsten Entwicklungsschritt. Die einzelnen Bauteile sind exakt so vernetzt, wie im Schaltplan definiert. Da die reale Hardware jedoch keine kreuzenden Leiterbahnen zulässt, wird der Schaltplan, wie in diesem Layout sichtbar, kreuzungsfrei auf zwei verschiedenen Ebenen reali-

siert. Die Farbkodierung der Leiterbahnen gibt hier lediglich die Position an, ob sich ein Leiterstück auf der Ober- oder Unterseite befindet.

Die 3D-Ansicht dient der weiteren Visualisierung der realen Hardware. Einzelne Bauelemente sind hier texturierte, naturgetreue Abbildungen, wie man sie auf der Platine wiederfindet. Mit dem Wissen der abstrakteren vorhergehenden Sichten, sollte der Benutzer nun die wichtigsten Bauelemente lokalisieren und ihre Funktionsweise besser verstehen können. Für den Fall, dass die reale Hardware nicht vorhanden ist, dient diese Ebene als Ersatz.

2.4.1 Konzept

Die graphische Benutzerschnittstelle (GUI) von PoCiLet bildet die Interaktionsfläche von Mensch und Maschine, welche in diesem Kapitel konzeptuell erfasst werden soll. Grundlegende Definitionen der wichtigsten Merkmale von angemessener Oberflächengestaltung werden erfasst, und die gewonnenen Kenntnisse in das Modell von PoCiLet übertragen.

Software-Ergonomie

Eine Benutzungsschnittstelle (hier auch: Dialog) muss bedienungsfreundlich sein und die Erledigung der Aufgaben, zu deren Bearbeitung das System eingesetzt wird, gut unterstützen [Bal98]. Diese Qualitätsanforderungen sind in der ISO-Norm 9241 (Teil 10) [fS] und in der DIN Norm 66234 [fNe85] beschrieben. Diese Normen nennen folgende Grundprinzipien als grundlegend für die Gestaltung von grafischen Benutzungsschnittstellen, insbesondere der grafischen Benutzerschnittstelle von PoCiLet:

- Aufgabenangemessenheit

  > Ein Dialog ist aufgabenangemessen, wenn er die Erledigung der Arbeitsaufgabe des Benutzers unterstützt, ohne ihn durch Eigenschaften des Dialogsystems unnötig zu belasten. Tätigkeiten, die sich aus der technischen Eigenart des Dialogsystems ergeben, sollten im Allgemeinen durch das System selbst ausgeführt werden.

- Selbstbeschreibungsfähigkeit

  > Ein Dialog ist selbstbeschreibungsfähig, wenn dem Benutzer auf Verlangen Einsatzzweck sowie Leistungsumfang des Dialogsystems erläutert werden kann, und wenn jeder Dialogschritt unmit-

telbar verständlich ist oder der Benutzer auf Verlangen über den jeweiligen Dialogschritt Erläuterungen erhalten kann.

- Steuerbarkeit

  Ein Dialog ist steuerbar, wenn der Benutzer die Geschwindigkeit des Ablaufs sowie die Auswahl und Reihenfolge von Arbeitsmitteln oder Art und Umfang von Ein- und Ausgaben beeinflussen kann.

- Erwartungskonformität

  Ein Dialog ist erwartungskonform, wenn er den Erwartungen der Benutzer entspricht, die sie aus Erfahrungen mit bisherigen Arbeitsabläufen oder aus der Benutzerschulung mitbringen sowie den Erfahrungen, die sie während der Benutzung des Dialogsystems und im Umgang mit dem Benutzerhandbuch gemacht haben.

- Fehlerrobustheit

  Ein Dialog ist fehlerrobust, wenn trotz erkennbar fehlerhafter Eingaben das beabsichtigte Arbeitsergebnis mit minimalem oder ohne Korrekturaufwand erreicht wird. Dazu müssen dem Benutzer die Fehler zwecks ihrer Behebung erkennbar gemacht werden.

- Individualisierbarkeit

  Ein Dialog ist individualisierbar, wenn er den individuellen Bedürfnissen und Fähigkeiten des Benutzers angepasst werden kann.

- Erlernbarkeit

  Ein Dialog ist erlernbar, wenn er den Benutzer in Lernphasen unterstützt und führt.

Definierte Interaktionskriterien

Bereits in der Seminarphase wurden wichtige psychologische und physiologische Gesichtspunkte in bezug auf die grafischen Schnittstelle erörtert und es wurde ein besonderer Wert auf diese Punkte gelegt, um auch benachteiligten Personen Zugang zu PoCiLet zu verschaffen.

- Rückmeldungen

  Der Benutzer sollte jederzeit Rückmeldungen vom Programm bekommen (können) über das, was gerade geschieht, und welche Einflussmöglichkeiten sich ihm bieten (siehe Kapitel 5.2). Das Programm soll hier vor allem die Möglichkeit bieten alle im aktuellen Kontext gegebenen Inhalte in mehrere Sprachen zu überführen, um Sprachbarrieren zu mindern. Desweiteren wird hierdurch ein Lerneffekt für den Anwender erzielt, da die betrachteten Inhalte im internationalen Sprachgebrauch betrachtet werden können und eine gemeinsame Kommunikationsbasis geschaffen wird.

- Beeinflussbarkeit

  Dies bedeutet, dass der Benutzer jederzeit in der Lage ist, die nächste Eingabe frei zu wählen und eventuelle Fehler rückgängig zu machen (siehe Kapitel 5.4.3).

- Ausführbarkeit

  Dies bezieht sich auf räumliche und zeitliche Parameter dergestalt, dass alle Handlungen der Benutzer flüssig und gezielt ausführbar sein sollten. Die grafische Gestaltung der Benutzerschnittstelle soll dementsprechend ausgelegt sein, unter Beachtung der gegebenen Konventionen (z.B. kulturelle Kohärenz, Aufgabenkonformität, Erwartungskonformität) und den Möglichkeiten der gewählten Umgebung „Java-Swing” und „Java3D”.

- Anpassbarkeit

  Es sollten Konzepte und Methoden angeboten werden, die es dem Benutzer erlauben eigene Operationsfolgen zu definieren, die zum Erreichen eines gewünschten Erscheinungsbildes oder Systemzustands erforderlich sind (siehe Kapitel 5.3 und Kapitel 5.4.4).

- Prägnanz

  Prägnanz bedeutet grafische Bedienelemente so zu gestalten, dass sie schnell erkannt werden, und Einzelelemente unter Vielen identifiziert werden können. Darunter fallen: Grafikprägnanz, Form- und Schriftprägnanz

- Strukturiertheit

  Dies bezieht sich auf die Anordnung von Bildschirmobjekten, d.h. das Wahrnehmungsfeld bzw. der Bildschirmaufbau sollten derart gestaltet sein, dass die Informationen zuverlässig erschlossen werden können. Optische, inhaltliche und aufgabenbezogene Aspekte müssen dabei im Zusammenhang betrachtet werden.

- Navigation

  Es sollte dem Benutzer klargemacht werden, wo er sich gerade im Programm befindet, wie er dort hinkam, und wie er diesen Ort wieder verlassen kann. Es sollten möglichst mehrere Zugriffe auf gleiche Funktionen angeboten werden, so dass der Benutzer sein präferiertes Navigationselement frei wählen kann (siehe Kapitel 5.6).

Anforderungsanalyse

Die Modellierung der tatsächlichen Anwendungsfälle von PoCiLet ist nun durch die softwareergonomischen Grundlagen und den gegebenen Interaktionskriterien möglich. Dabei wurden die analysierten Anwendungsfälle in UML-Aktivitätsdiagramme modelliert, um Zusammenhänge und Sequenzen im Handlungsablauf mit der GUI zu verdeutlichen. Zu Beginn der Projektgruppe wurden Minimalanforderungen an die grafische Schnittstelle genannt. Diese Übersicht diente als Grundlage zur genaueren Spezifikation der einzelnen Anwendungsfälle.

1. Datei
   (a) Neues Simulatorprogramm
   (b) Simulatorprogramm laden

(c) Simulatorprogramm speichern

(d) Simulatorprogramm editieren

2. Simulator

   (a) Simulator Schritt vor

   (b) Simulator Schritt zurück

3. Sichten

   (a) Sicht Vollbild

   (b) Sicht Integriert

   (c) Sicht Vierfach

4. Sprache

   (a) Sprache Deutsch

   (b) Sprache Englisch

Entwicklungsmerkmale im Überblick

- Protokollauszug vom 02.10.2003

  > „Zuerst wurden grobe Strukturmerkmale und Architekturmerkmale festgelegt, wie Harvard-Architektur und der Befehlssatz, sowie Bitbreiten."

- Protokollauszug vom 16.10.2003

  > „Es wurden 5 Hauptanwendungsfälle festgestellt, welche im Detail noch weitere Anwendungsfälle beinhalten."

- Protokollauszug vom 20.11.2003

  > "Die Gruppe hat in dieser Sitzung die Fortschritte der GUI präsentiert. Diese ist um die Funktionsweise der Internationalisierbarkeit aller Textelemente erweitert worden. Desweiteren ist die Rahmenstruktur abgeschlossen, so dass ab jetzt ... der Aufbau der einzelnen Sichtweisen des Rechnersystems angefangen werden kann. Als erste Sichtweise wird die Register-Transfer-Ebene in Angriff genommen, da sich diese unabhängig vom Schaltplan und der Hardware zum jetzigen Zeitpunkt aufbauen lässt."

2.4.2 Prototyp

Um einige Designideen auszuprobieren und den endgültigen Aufbau der Oberfläche von PoCiLet festzulegen, wurde zunächst ein Prototyp der GUI ohne Funktionalität erstellt.

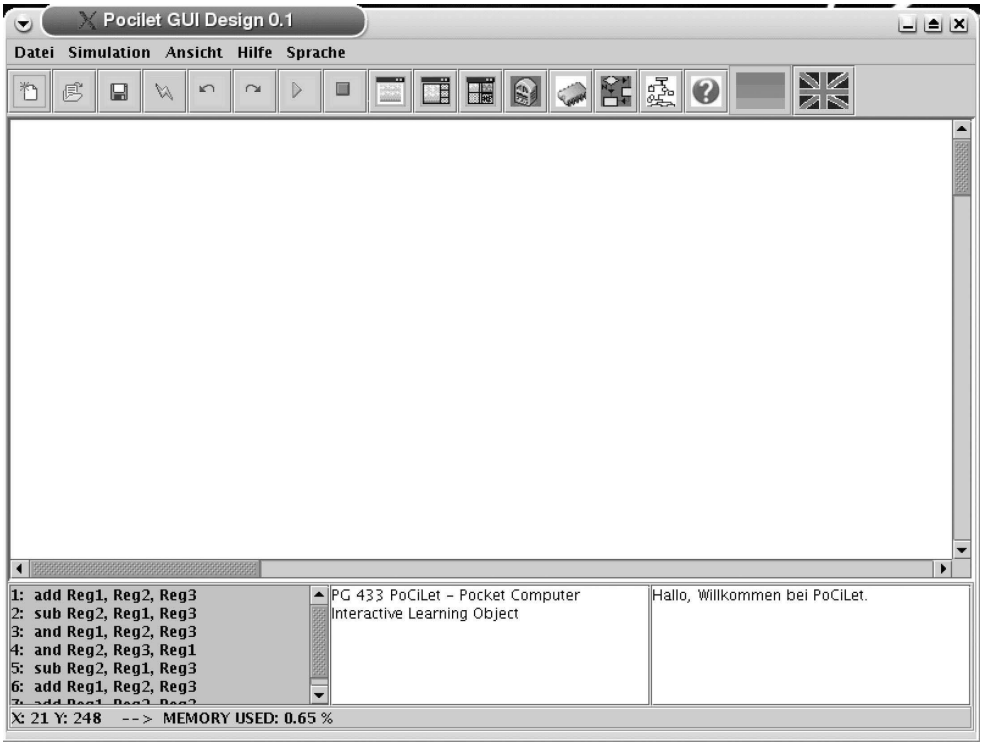


Abbildung 2.15: Prototyp - Vollbild Ansicht

In Abbildung 2.15 sieht man die Aufteilung des PoCiLet-Fensters. Wie es bei den meisten anderen Programmen mit grafischer Benutzerschnittstelle üblich ist, gibt es ganz oben eine Menüzeile. Mit Hilfe der Menüs „Datei", „Simulation", „Ansicht" und „Sprache" und deren Unterpunkten sollen alle Funktionen von PoCiLet direkt aufgerufen werden können.
Der Rest des Fensters ist zunächst einmal in drei Bereiche geteilt. Oben ist die ToolBar platziert, in der Mitte das Hauptfenster und unten einen weiteren Informationsbereich.

Die ToolBar enthält Buttons, mit denen man die gängigsten Funktionen von PoCiLet steuern kann. Ganz links findet man Buttons zum Erstellen eines neuen Programms, Laden und Speichern eines Programms und Verbinden der Software mit der Hardware. Rechts daneben haben wir vier Buttons zur Steuerung des

Simulators platziert. Hier kann jeweils ein Simulationsschritt vor oder zurück gegangen bzw. der Simulator gestartet oder gestoppt werden.
Als nächstes sieht man die Buttons zur Auswahl des Ansichtsmodus. Hier stehen die Möglichkeiten „Vollbild", „Integriert" und „Vierfach-Sicht" zur Auswahl.

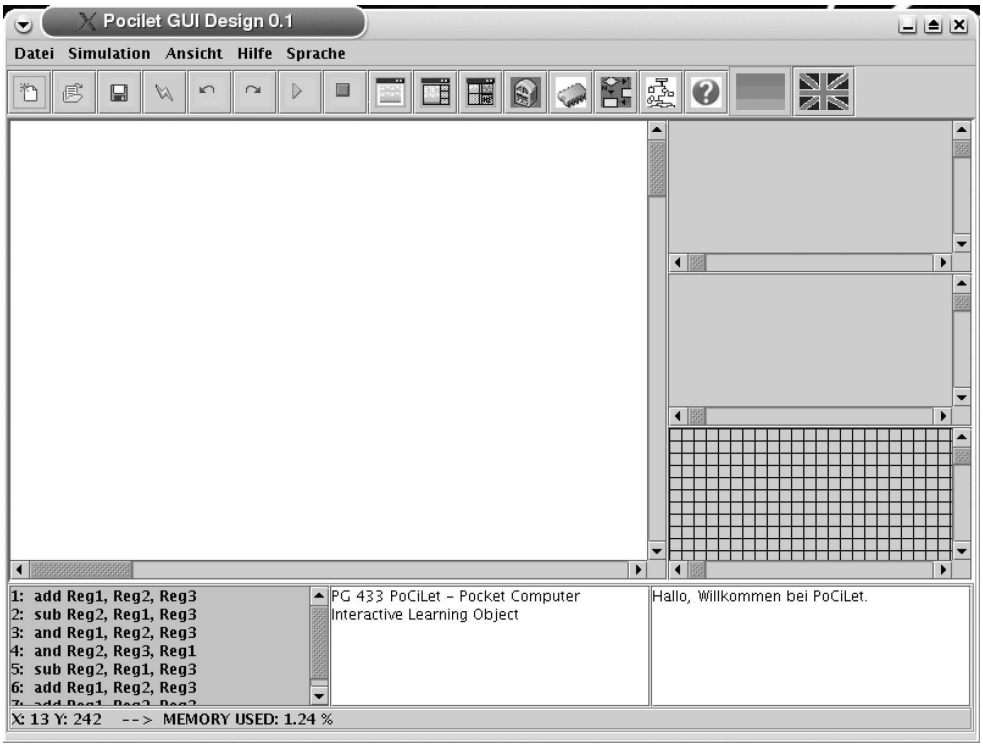


Abbildung 2.16: Prototyp - Integrierte Sicht

Den Ansichtsmodus „Vollbild" sieht man in Abbildung 2.15. Hier wird nur eine Ebene von PoCiLet im kompletten Hauptfenster angezeigt. Im Ansichtsmodus „Integriert" (Abbildung 2.16) hat man eine der vier Ebenen links in einem großen Fenster und die anderen drei Ebenen werden in den kleineren Fenstern an der rechten Seite angezeigt. Wählt man den Ansichtsmodus „Vierfach-Sicht", so werden alle vier Ebenen von PoCiLet in gleich großen Fenstern angezeigt (Abbildung 2.17).
Die nächsten vier Buttons sind zur Auswahl der Ebene (3D-, Layout-, Schaltkreis- oder RT-Ebene) im Vollbildmodus und in der integrierten Ansicht. Der dritte Button von rechts ist der Direkt-Hilfe-Button. Klickt man ihn an, kann man sich mit einem weiteren Klick auf ein Objekt einen Hilfetext zu den jeweiligen Objekt anzeigen lassen.
Ganz rechts in der ToolBar befinden sich die Buttons zur Auswahl der Sprache von PoCiLet.

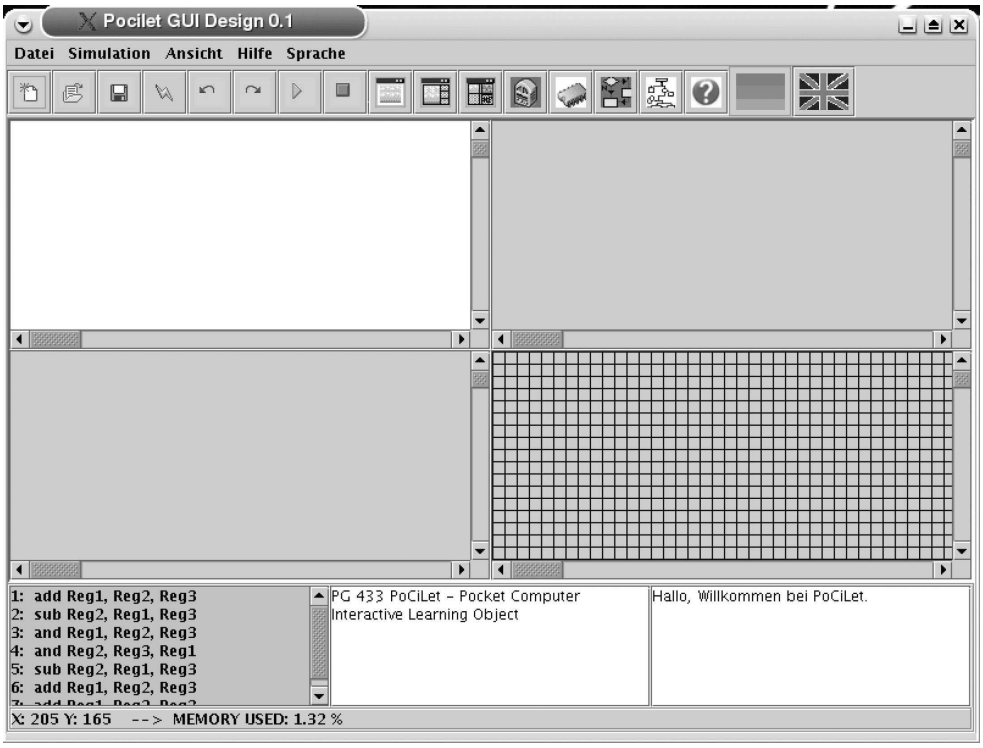


Abbildung 2.17: Prototyp - Vierfach-Sicht

Im unteren Informationsbereich findet man drei Fenster. Ganz links sieht man das aktuell im Speicher des PoCiLet-Computers geladene Programm. Der Befehl, der zur Zeit abgearbeitet wird, ist in diesem Fenster grün hervorgehoben. Das Fenster in der Mitte ist für Erklärungstexte zu gerade ausgewählten Bauteilen vorgesehen. Im rechten Fenster wird der Inhalt des RAM angezeigt.

2.4.3 Entwurf

Die Klassen und ihre Zusammenhänge

Bei erster grober Betrachtung des Klassendiagramms (siehe Abbildung 2.18), kann man die Klassen in drei Gruppen gliedern. Die erste Gruppe bilden hierbei die GUI-Klassen, die unsere Benutzerschnittstelle bilden. Die zweite Gruppe bilden die Utility-Klassen die einige Funktionen für die GUI-Klassen zur Verfügung stellen. Des weiteren gibt es die Gruppe der Control-Klassen, die die eigentliche Funktionalität, wie z.B. die Kommunkation mit dem Simulator und der Hardware oder das Parsen der Programme, beinhalten. Zu guter Letzt die StartUp-Klasse und die Program-Klasse, die in keine der genannten Gruppen fallen. Die StartUp-Klasse hat nicht viel Funktionalität - ihre einzige Aufgabe besteht darin PoCiLet zu in-

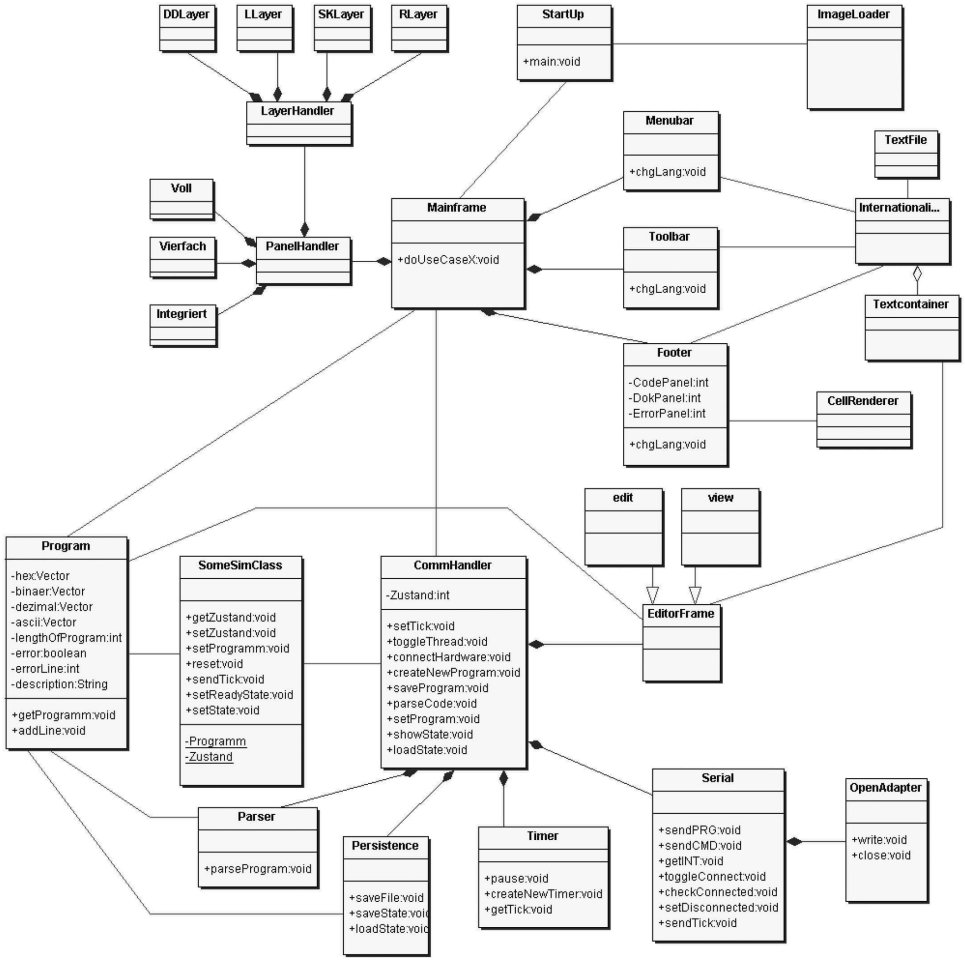


Abbildung 2.18: Das Klassendiagramm des Entwurfs

itialisieren und das Mainframe zu starten. Die Program-Klasse ist ein Container für ein PoCiLet Assembler Programm.

Die GUI-Klassen:

Um die grafische Benutzungsschnittstelle zu definieren, haben wir mit unserer GUI-Hauptklasse Mainframe begonnen. Diese Klasse bildet unser eigentliches Hauptfenster. Sie ist der Container für alle GUI-Elemente und hat die einzige Aufgabe die Interaktions-Events vom Benutzer abzufangen und dafür zu sorgen, dass diese ausgeführt werden.

Um, auf die Eingabemöglichkeiten bezogen, möglichst flexibel zu bleiben, hat das Hauptfenster sowohl eine Menubar als auch eine Toolbar. Beides sind eigene Klassen, die in das Mainframe eingebunden werden.

Desweiteren gibt es eine Footer Klasse, die genauso in das Mainframe eingebunden wird, sie bildet unseren Informationsbereich. Die Footer Klasse enthält drei Panels die Rückmeldungen an den Benutzer ermöglichen.

Mit diesen vier Klassen ist die Struktur unseres Hauptfensters bereits gegeben.

Damit der Benutzer sich die Oberfläche nach seinen Vorlieben einrichten kann, wollen wir drei verschiedene Ansichtsmodi anbieten. Alle drei Ansichtsarten werden in einer eigenen Klasse (Voll, Vierfach, Integriert) implementiert.
Voll enthält dabei immer nur ein einziges Panel in dem der aktuelle Layer in Vollbild angezeigt wird. Vierfach enthält vier Panels in denen alle vier möglichen Layer gleichzeitig und alle in gleicher Größe angezeigt werden. Integriert enthält auch vier Panels, in denen, genau wie in Vierfach, alle vier möglichen Layer angezeigt werden, wobei aber der aktuell ausgewählte Layer möglichst groß dargestellt wird und die übrigen drei nur als kleine Übersicht.

Die Panels werden wiederum von der Klasse PanelHandler verwaltet. Wenn das Mainframe einen Event vom Benutzer bekommt, die Ansicht zu ändern, gibt es diesen Aufruf nur an den PanelHandler weiter, der dann dafür sorgt, dass die erwünschte Sicht erstellt und angezeigt wird.

Damit der PanelHandler die Fenster seiner möglichen Darstellungsarten auch mit Inhalt füllen kann, hat der PanelHandler einen LayerHandler, der die verschiedenen Layer (mit den verschiedenen Sichten der Platine) verwaltet. Wenn der PanelHandler einen Aufruf bekommt, eine gewünschte Sicht zu erstellen, spricht er den LayerHandler an, damit dieser die gewünschten Sichten erzeugt.
Die Möglichen Sichten sind:

- Die Sicht der Register-Transfer-Ebene, umgesetzt in der Klasse Rlayer
- Die Schaltkreis-Ansicht umgesetzt in der Klasse SKLayer
- Die Layout-Ansicht, umgesetzt in der Klasse LLayer
- Die 3D-Ansicht, umgesetzt in der Klasse DDLayer.

Ausgehend vom Hauptfenster lässt sich ein weiteres Dialogfenster öffnen, in dem die PoCiLet-Befehle editiert werden können. Diese Aufgaben umfasst die Klasse EditorFrame.

Die Utility**-Klassen:**

Alle GUI-Elemente die in irgendeiner Form Text enthalten (z.B. als Label oder als ToolTip), greifen auf die Klasse Internationalizer zu und holen von dieser die aktuell benötigte Übersetzung, um den Text setzen zu können. Der Internationalizer beinhaltet eine weitere Klasse Textcontainer, die alle Übersetzungen der benötigten Begriffe verwaltet.

Um die aktuellen Meldungen im Footer farbig unterlegen zu können gibt es eine extra Klasse CellRenderer, die nur dafür zuständig ist, die entsprechenden Listeneinträge der Tabelle farbig zu unterlegen.

Desweiteren gibt es noch die Klasse ImageLoader, die von der StartUp Klasse angesprochen wird und dafür zuständig ist, die von PoCiLet benötigten Bilder (wie z.B. die Bilder von den Buttons in der Toolbar), zu laden.

Die Control**-Klassen:**

Die wichtigste unter den „Control"- Klassen ist der CommHandler. Der CommHandler arbeitet mit dem Mainframe zusammen. Jeden Benutzer-Event, den die Mainframe Klasse bekommt, leitet sie an den CommHandler weiter. Dieser ist also dafür zuständig, dass alle eintreffenden Anwendungsfälle ausgeführt werden. Zugleich bildet der CommHandler die Schnittstelle zur Simulation.
Außerdem erstellt der CommHandler ein EditorFrame, wenn dieser benötigt wird. Wenn der CommHandler vom EditorFrame ein Programm übergeben bekommt, dann gibt er dieses an die Parser-Klasse oder an die Persistence-Klasse weiter, damit es auf syntaktische Korrektheit überprüft oder abgespeichert wird.
Der CommHandler koordiniert auch die Kommunikation mit der Hardware, für die die Klasse Serial zuständig ist.

Eine weitere Control-Klasse ist die Klasse Timer. Diese Klasse erzeugt nebenläufig, in regelmäßigen Abständen von einigen Sekunden, einen sogenannten Tick. Das bedeutet, die Klasse Timer ruft die Methode setTick() des commHandlers auf. Dieser Timer wird verwendet um den Takt zu simulieren. Er wird auch an die Hardware weitergereicht um Simulation und Hardware synchron zu halten.

Der Parser ist dafür zuständig, das ihm übergebene Programm zu parsen und falls es syntaktisch korrekt ist, in Hex-, Dezimal- und Binärformat zu übersetzen. Falls das Programm nicht übersetzbar ist, vermerkt der Parser dies im Program-Objekt.

Die Persistence-Klasse hat nur die Aufgabe, alle Objekte die sie übergeben bekommt, unter dem gewünschten Namen abzuspeichern und dementsprechend Objekte die bereits abgespeichert wurden wieder zu laden.

Die Serial-Klasse ist der „Ansprechpartner“ des CommHandlers, wenn es darum geht mit der Hardware zu kommunizieren.
Serial vermittelt zwischen dem CommHandler und dem OpenAdapter. Dieser ist dafür da das aktuelle Betriebssystem zu erkennen und über das Netzwerk mit der eigentlichen Hardware zu kommunizieren. Um unter Windows Systemen mit der Hardware kommunizieren zu können, wird das javaCommunications Paket [jCom] von Sun verwendet. Unter Linux das RXTX Paket [RXTX] von RXTX.org das unter der GNU Lizenz steht.

Die SomeSimClass steht als Symbol für die Schnittstellenklasse zur Simulation.

Und abschliessend ist da die Klasse Program. Diese Klasse ist eine Art Container-Klasse ohne größere Funktionalität. Sie enthält vier Attribute vom Typ Vector, in denen das Programm in verschiedenen Formaten (Binär, Hexadezimal, als Integer und im ASCII-Format) zeilenweise gesetzt und ausgelesen werden kann. Zusätzlich verfügt die Klasse Program noch über Attribute, die es erlauben auf Fehler zu verweisen.

Textcontainer-Klasse und Property-Datei

Es wäre umständlich, bei der Erzeugung jedes einzelnen GUI-Elementes in einer umfangreichen Abfrage die Umgebung zu testen, die aktuelle Sprache zu ermitteln und fest im Quellcode verankerte Übersetzungen dementsprechend zu setzen. Hier scheint eine Art Übersetzer, der unter Angabe des Textes und der gewünschten Sprache aus einer Textdatei die richtige Beschriftung ermittelt, als bessere Lösung. Genau diese Funktionalität stellt unser Textcontainer zur Verfügung. Der Textcontainer enthält alle benötigten Texte in verschiedenen Übersetzungen und stellt eine Möglichkeit zur Verfügung, den gewünschten Text bei Angabe der Sprache auszulesen.

Die gewünschten Übersetzungen werden in Property-Dateien abgelegt, wobei für jede Übersetzung eine Datei angelegt wird. Im Namen der Datei wird angegeben, für welche Übersetzung der Inhalt gedacht ist, so wird eine Property-Datei mit deutschen Übersetzungen einen Namen enthalten der de_DE enthält. Hierbei steht das de für die Sprache deutsch und das DE für Deutschland.
Die Datei selbst enthält Paare (key, value) von allgemeinen Beschreibungen (z.B. Buttons oder Tooltips), die angeben für welchen Zweck der entsprechende Text

zu verwenden ist, und einen zugehörigen Text in der jeweiligen Sprache.
Zum Beispiel ist Editor.saveButton=Speichern ein Eintrag aus unserer deutschen Property Datei.

PoCiLet verwendet folgende Property-Dateien:

- MessageBundle_de_DE
- MessageBundle_en_US

Die PaketStruktur

Wie die Klassen sich bei näherer Betrachtung des Klassendiagramms (Abbildung 2.18) zu Gruppen gliedern, so haben wir auch die Paket Struktur gegliedert.
In erster Ebene liegt das Paket PoCiLet (Abbildung 2.19), welches das ganze Projekt umfasst. Darin befinden sich die Pakete:

- Gui - enthält alle GUI Klassen
- Simmanager - enthält alle Klassen die unter Control-Klassen beschrieben wurden, sowie die Program-Klasse
- Util - enthält alle Utility-Klassen
- Simulation - enthält die Klassen, die für die eigentliche Simulation zuständig sind

sowie die StartUp-Klasse.

Das Paket gui ist (wie in Abb. 2.20 zu sehen) weiter unterteilt.

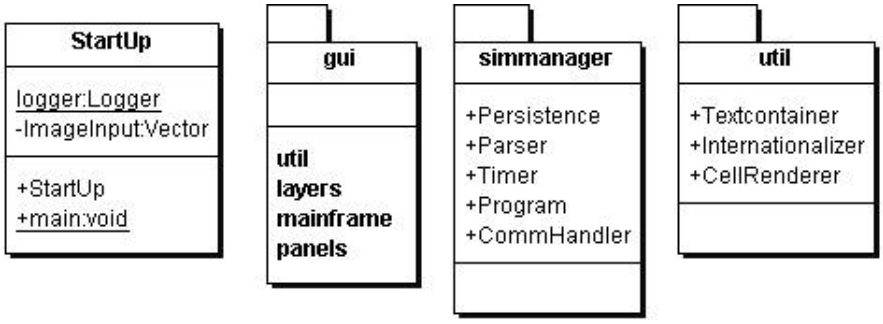


Abbildung 2.19: Das Paket PoCiLet

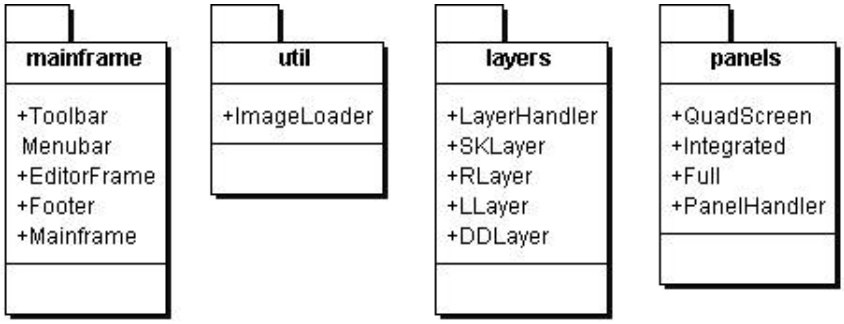


Abbildung 2.20: Das Paket gui

2.4.4 Realisierung

Alle vier Sichtweisen nutzen die gleiche Datenbasis zur Visualisierung eines Zustands. Die grafische Benutzerschnittstelle wartet auf ein weiteres Taktsignal, ausgelöst durch das manuelle Drücken eines Tasters auf der Hardware oder dem Klick auf einen der Buttons in der Toolbar der GUI, um den PoCiLet -Simulationskern anzuweisen einen weiteren Taktschritt zu simulieren.

Der PoCiLet-Simulationskern antwortet mit einem State-Objekt. Diese Instanz der Klasse State enthält zwei HashMaps. Diese füllt der Simulator mit den simulierten Objekten, respektive ihren internen Daten. Die Daten des State-Objektes werden über die Methode LayerHandler.setActualState an alle vier Sichten übergeben, welche daraus ihre Anzeigewerte aktualisieren.

Das State-Objekt wird darüber hinaus der Liste CommHandler.comPreviousStates hinzugefügt. Somit ist es möglich die Simulation in einen früheren Zustand zurückzuversetzen.

Die im PoCiLet-Simulationskern zu simulierenden Komponenten sind die gleichen, wie sie in der RT-Ebene dargestellt sind. Daher hat die Register-Transfer-Ebene die höchste Priorität in der Realisierung. Eine eigens für diese Ebene manuell realisierte Darstellung der Register-Transfer-Ebene zeigt die Abbildung 2.21 mit zugehöhriger Legende in Tabelle 2.7.

Die Register-Transfer-Ebene

In der RT-Ebene werden einige Signalleitungen zusammengefasst, einzelne dicke Leitungen repräsentieren einen Bus, also ein Bündel von Einzelsignalleitungen. Diese Abstraktion hält den Schaltplan übersichtlich und der Benutzer kann sich auf die wesentliche Vernetzung der Bauteile konzentrieren.

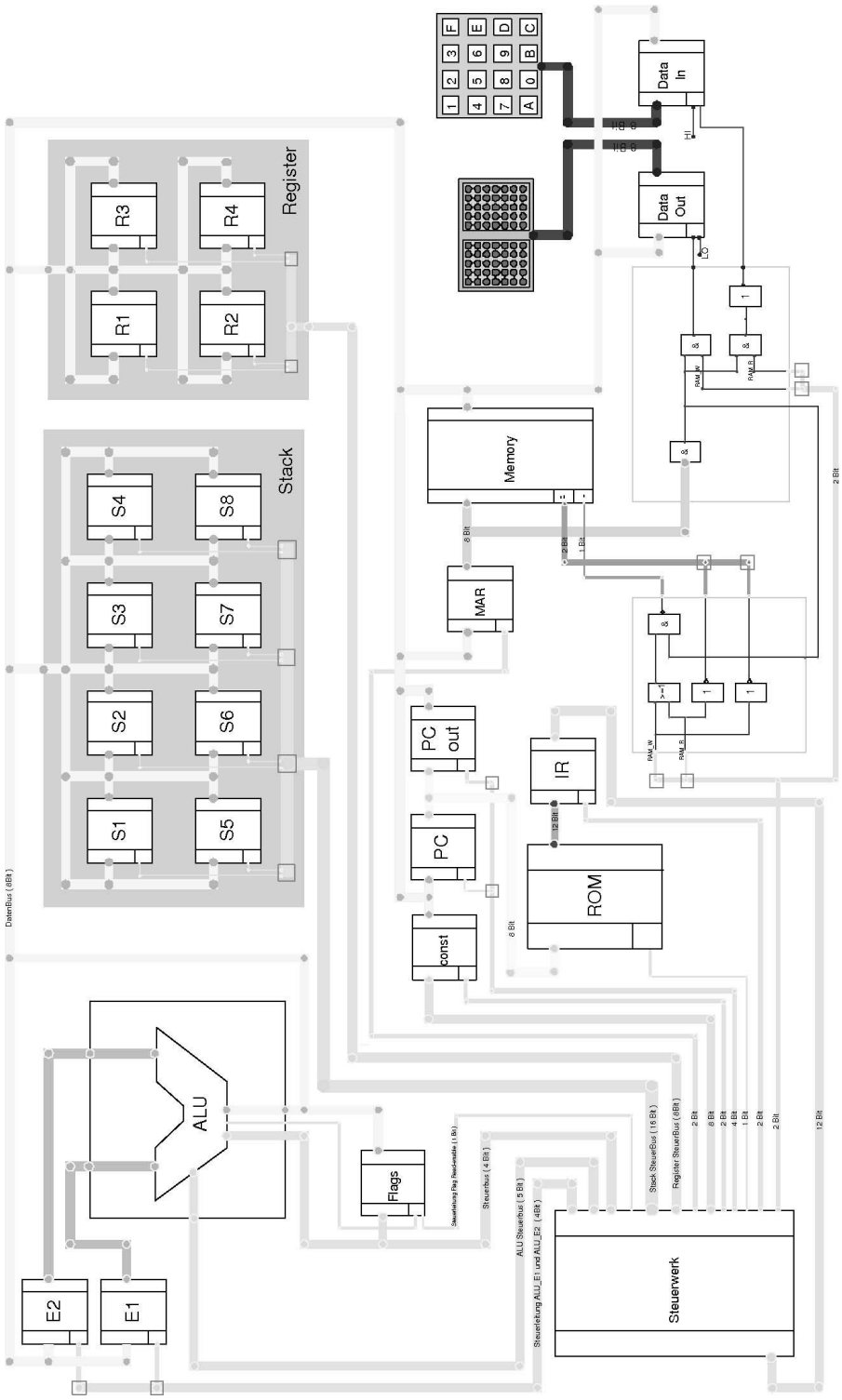


Abbildung 2.21: Register-Transfer-Ebene (Visio-Skizze)

Das Resultat ist ein an einzelnen Objekten (Bauteile, segmentierte Leitungen) stark reduzierter Netzplan, welcher manuell durch eine XML-Konfigurationsdatei (circuit.xml) beschrieben ist. Die Klasse DOMCircuitParser liest diese Datei ein und erstellt daraus zwei HashMaps, welche die einzelnen Bauelemente und Signal-Routen als Objekte speichern.

Aus dieser Objektsammlung generiert die Klasse GFXGenerator sowohl die sichtbaren GUI-Objekte der RT-Ebene, wie auch den kompletten Netzplan, auf dem der PoCiLet-Simulationskern arbeitet.
Im Package pocilet.gui.draw.simobjects beschreibt jeweils eine Java-Klasse das Aussehen der Bauelemente in der GUI. Das Logik-Verhalten der Bauelemente wird analog durch die jeweiligen Simulator-Klassen im package pocilet.simulation.simobjects beschrieben.

Das Zeichnen der RT-Ebene, sowie die Auswertung interaktiver Aktionen durch den Benutzer (Mausklicks auf Objekte) sind in der Klasse RLayer implementiert.

Ein Mausklick in dem Fensterbereich veranlasst die Methode findClickedComponent ein evtl. angeklicktes Bauteil oder Signal zu identifizieren.

Die Methode setState makiert farblich die Signalleitungen. Im voreingestellen Anzeigemodus werden die Signale in Abhängigkeit ihrer internen Werte eingefärbt (Farbkodierung: siehe Handbuch). Im differenziellen Anzeigemodus (im Ansichtsmenü der GUI umstellbar) werden nur diejenigen Signale markiert, welche eine Werteänderung zwischen zwei aufeinanderfolgenden Zuständen haben. Diese Signale sind in der Liste CommHandler.hotSignalList für jeden korrespondierenden Zustand hinterlegt.

E1,E2	ALU Eingangsregister
Flags	Flag-Register (4bit)
S1-S8	Stack-Register (8bit)
R1-R4	Programmierbare Register (8bit)
const	Konstanten-Register (8bit)
PC, PC OUT	Programmzähler-Register (8bit)
ROM	Programmspeicher (8bit)
IR	Befehlsregister (12bit)
Memory	Datenspeicher (8bit)
Data Out, Data In	Register (8bit), Pufferspeicher für Tastatureingaben

Tabelle 2.7: Legende zu Abbildung 2.21

Die Schaltkreis-Ebene

Die Umsetzung der Schaltkreis-Ebene erfolgte in mehreren Teilschritten. Zuerst wurden zwei Eagle-ULP-Skripte erzeugt, die den PoCiLet-Schaltplan und die PoCiLet-Libraries in textueller Eagle-Objekt-Beschreibung ausgeben, die dann mittels Parser importiert werden konnten. Auf diese Art und Weise war es möglich, die Darstellung des Schaltplans in PoCiLet automatisch und dynamisch zu erzeugen. Ein Beispiel für ein ULP-Skript ist im folgenden Abschnitt unter „Die Layout-Ebene“ zu finden.

Als erstes Zwischenergebnis wurden mehrere ASCII-Dateien erzeugt. Für jede der im Schaltplan verwendeten Libraries steht jetzt eine Text-Datei zur Verfügung, die die textuelle Objektbeschreibung der verwendeten Symbole enthält. Für den Schaltplan existiert eine Text-Datei mit den Parametern des Schaltplans, wie zum Beispiel die Größe, die Lage und Länge der Busse und Leitungen und die Objektpositionen und -ausrichtungen.
Diese Dateien wurden dann mit dem Parser eingelesen, zunächst die Text-Dateien der Libraries, dann die des Schaltplans. Während des Parsens wurde dann, entsprechend der von Eagle vorgegebenen Struktur, die entsprechende Java-Objektstruktur generiert.

Im Package pocilet.gui.draw.skcomponents beschreiben die Java-Klassen das Aussehen der Elemente im Schaltplan. Die Klasse GUI_SK_Schematic kapselt dabei alle Komponenten der Schaltkreis-Ebene. Folgende Java-Klassen sind hierbei erzeugt worden:

- GUI_SK_Circle
- GUI_SK_Device
- GUI_SK_Gate
- GUI_SK_Instance
- GUI_SK_Junction
- GUI_SK_Net
- GUI_SK_Part
- GUI_SK_Pin
- GUI_SK_Rectangle
- GUI_SK_Symbol

- GUI_SK_Text
- GUI_SK_Wire

Diese Klassen implementieren hierbei die Klasse Serializable, um alle Komponenten in einer Objekt-Datei zu speichern, damit beim Start von PoCiLet kein Zeitverlust durch erneutes Einlesen des ASCII-Exports auftritt. So fällt lediglich beim initialen Start von PoCiLet die Ladezeit der Objekt-Datei ins Gewicht. Die Objekt-Datei kann bei eventuell auftretenden Änderungen am Schaltplan durch diese Vorgehensweise leicht neu erzeugt werden.

Zusätzlich zu den reinen Objektinformationen - also deren Form und Größe - sind die Komponenten erweitert um Farbgebungen und einen Skalierungsfaktor, um einen Zoom innerhalb der Ebene zu ermöglichen und um innerhalb von PoCiLet ein leichtes Umschalten in ein neues Farbset zu gewährleisten. Es existieren drei vordefinierte Farbsets, welche die Lesbarkeit und Navigation in der Ebene vereinfachen. Ein spezielles DoubleBuffering innerhalb des Java JPanel ermöglicht ausserdem eine schnelle Navigation in der Layout-Ebene, da lediglich der BackBuffer neu gezeichnet werden muss. Dies verhindert zu lange Wartezeiten bei Benutzung des JScrollPane. Ein erneutes Zeichnen des BackBuffer ist lediglich bei Veränderung der Zoomstufe nötig.

Analog zu der RT-Ebene werden Zustände (State-Objekte) als Parameter der Methode setState in die Layout-Ebene transferiert und es werden diejenigen Einzelleitungen markiert, die im aktuellen Simulationsschritt verwendet werden. Um diese farblich von nicht verwendeten Signalen innerhalb eines Simulationsschrittes abzusetzen, ist z.B. im Standard-Farbset die Farbe Grün die Zeichenfarbe für nicht verwendete Signale und die aktiven Signale werden mit der Farbe Gelb gezeichnet.

Die Layout-Ebene

Die Realisierung der Layout-Ebene setzt sich aus mehreren Teilproblemen zusammen. Zuerst ist ein Eagle-ULP-Skript erzeugt worden, welches das geroutete PoCiLet-Board in textueller Eagle-Objektbeschreibung ausgibt, die dann mittels Parser importiert wurde. Dieser Weg ermöglichte es, Java-Klassen zu erzeugen, welche inhaltlich die Eagle-Objektstruktur erfassen. Es folgt als Beispiel ein Auszug aus dem ULP-Skript für das PoCiLet-Board:

Listing 2.4: Beispiel - Ausschnitt Eagle-ULP-Board-Skript

```
1  // VIA LISTE ERZEUGEN
2  printf("VIALIST\n");
3  B.signals(S) {
4     S.vias(V) {
5        // VIA: POSX POSY DRILL
6        printf("%10.0f %10.0f %10.0f\n",
7              mv(V.x)*100, mv(V.y)*100, mv(V.drill)*100);
8     }
9  }
10 printf("ENDVIALIST\n\n");
11
12 // SIGNALE LESEN
13 printf("SIGNALDETAILS\n");
14 B.signals(S) {
15    // SIGNALNAME
16    printf("%s\n", S.name);
17    S.wires(W) {
18       // WIRE FROMX FROMY TOX TOY WIDTH LAYER
19       printf("Wire %10.0f %10.0f %10.0f %10.0f %10.0f %d\n",
20             mv(W.x1)*100, mv(W.y1)*100, mv(W.x2)*100,
21             mv(W.y2)*100, mv(W.width)*100, W.layer);
22    }
23    printf("***\n");
24 }
25 printf("ENDSIGNALDETAILS\n\n");
```

Als erstes Zwischenergebnis ist eine ASCII-Datei entstanden, welche nun mit einem einfachen Java-Parser ausgelesen wird. Innerhalb dieses Parsers wurde gleichzeitig die entsprechende Java-Objektstruktur generiert und vernetzt. Im Package pocilet.gui.draw.boardcomponents beschreiben die Java-Klassen das Aussehen der Bauelemente in der Layout-Ebene. Die Klasse GUI_LL_Board kapselt dabei alle Komponenten der Layout-Ebene. Folgende Java-Klassen sind hierbei erzeugt worden:

- GUI_LL_Board
- GUI_LL_Contact
- GUI_LL_Element
- GUI_LL_Rectangle
- GUI_LL_Signal
- GUI_LL_Via
- GUI_LL_Wire

Diese Klassen implementieren die Klasse Serializable, um alle Komponenten in einer Objekt-Datei zu speichern, damit beim Start von PoCiLet kein Zeitverlust durch erneutes Einlesen des ASCII-Exports auftritt.

Zusätzlich zu den reinen Objektinformationen - also deren Form und Größe - sind die Komponenten erweitert um Farbgebungen und einen Skalierungsfaktor, um einen Zoom innerhalb der Ebene zu ermöglichen und um innerhalb von PoCiLet ein leichtes Umschalten in ein neues Farbset zu gewährleisten. Es existieren nun drei vordefinierte Farbsets, welche die Lesbarkeit und Navigation in dieser Ebene vereinfachen. Ein spezielles DoubleBuffering innerhalb des Java JPanel ermöglicht ausserdem eine schnelle Navigation in der Layout-Ebene, da lediglich der BackBuffer neu gezeichnet werden muss. Dies verhindert zu lange Wartezeiten bei der Benutzung des JScrollPane und ein erneutes Zeichnen des BackBuffer ist lediglich bei Veränderung der Zoomstufe nötig.

Analog zu der RT-Ebene werden Zustände (State-Objekte) als Parameter der Methode setState in die Layout-Ebene transferiert und es werden diejenigen Einzelleitungen markiert, die im aktuellen Simulationsschritt verwendet werden. Um sich farblich von nicht verwendeten Signalen innerhalb dieses Simulationsschrittes abzusetzen, ist z.B. im Standard-Farbset die Farbe Rot bzw. Blau die normale Zeichenfarbe auf der Ober- bzw. Unterseite der Platine und die markierten Signale haben die Farben Orange (für Rot im Zustand markiert) und Cyan (für Blau im Zustand markiert).

Die 3D-Ebene

Die Realisierung der 3D-Ebene erfolgt durch eine zu der Layout-Ebene analoge Verfahrensweise. Die Informationen, welche aus dem Eagle-Design des PoCiLet-Board bereits ausgelesen sind, stimmen inhaltlich mit den benötigten Informationen für die 3D-Ebene überein, da diese lediglich eine transparente Projektion der Ober- und Unterseite der Platine darstellen. Das ULP-Skript der Layout-Ebene ist dabei erweitert um die Layerinformationen von Eagle, damit man Ober- und Unterseite trennen kann und sowohl die Bauteile, als auch die Signalleitungen korrekt in eine 3D-Projektion übertragen kann.

Der Parser für die Generierung und Umsetzung der 3D-Bauteile ist eine an die neuen Java-Klassen angepasste Variante, welche nun auch die Layerinformationen von Eagle berücksichtigt, um die nötigen affinen Transformationen der Bauteilpositionen durchzuführen. Innerhalb dieses Parsers wird wiederum die Java-Objektstruktur generiert und vernetzt. Im Package pocilet.gui.draw.ddcomponents beschreiben die Klassen die Informationen die zur Erzeugung der 3D-Ebene be-

nötigt werden. Die Klasse GUI_DD_Board kapselt dabei alle Komponenten der 3D-Ebene analog zur Layout-Ebene. Die Klasse DDLayer im Package pocilet.gui.layers stellt dabei die Kernkomponente dar, welche die ursprünglichen 2D-Informationen durch intelligente Analyse der Daten mit Hilfe der Klasse BoardObjectFactory im Package pocilet.game in 3D-Bauteile transformiert und diese in der dritten Dimension positioniert. Folgende Java-Klassen sind hierbei erzeugt worden:

- GUI_DD_Board
- GUI_DD_Contact
- GUI_DD_Element
- GUI_DD_Rectangle
- GUI_DD_Signal
- GUI_DD_Via
- GUI_DD_Wire

Diese Klassen implementieren wiederum die Klasse Serializable, um alle Komponenten in einer Objekt-Datei zu speichern, damit beim Start von PoCiLet kein Zeitverlust durch erneutes Einlesen des ASCII-Exports auftritt. Die Objekt-Datei kann durch diese Vorgehensweise analog zur Layout-Ebene leicht neu erzeugt werden bei eventuell auftretenden Änderungen am Layout-Design, welche entsprechende Änderungen an der 3D-Ebene nach sich ziehen würde.

Zur tatsächlichen Darstellung der 3D-Komponenten sieht Java3D die Verwendung eines Canvas3D vor, welches in ein JPanel integriert werden kann, statt des üblichen Canvas, welches für die 2D-Darstellung verwendet wird. Folgende Klassen stellen nun die dreidimensionalen Objekte innerhalb des Canvas3D dar:

- Comport3D
- Diode3D
- ElementRectangleShape
- ElementWireShape
- FPGA3D
- Keypad3D
- LED3D

- LEDARRAY3D
- PerfectVia3D
- Signal3D
- SO14
- SOJ28A

Diese Objekte werden mit den Informationen der entsprechenden Bauteiltypen, deren Positionen und deren affine Transformation/Rotation innerhalb des DDLayer generiert. Es ergibt sich dadurch eine baumartige Struktur, welche den gewünschten SceneGraph aus Java3D-Objekten vom Typ Group, BranchGroup und TransformGroup mit Hilfe der Transformationsklasse Transform3D zusammensetzt. Der Lichteffekt Ambientlight erzeugt zusammen mit den generierten und positionierten Objekten den gewünschten Effekt eines vollständigen PoCiLet-Board, welches nun noch um so genannte Behaviors ergänzt wird, um Interaktivität zu erlauben. Es folgt ein Auszug aus der Erstellungsmethode des virtuellen Universums (Listing 2.5 :

Listing 2.5: Beispiel - Ausschnitt Java3D Universum generieren

```
1  // Initialisierung des 3D-Board mit Transformation der ViewPlatform
2
3  public void init() {
4    setLayout(new BorderLayout());
5    GraphicsConfiguration config = SimpleUniverse.
         getPreferredConfiguration();
6    c = new Canvas3D(config);
7    add("Center", c);
8    u = new SimpleUniverse(c);
9
10   // Erzeuge eine Scene und verbinde sie mit dem virtuellen
         Universum
11   BranchGroup scene = createSceneGraph(c);
12
13   // Erstellen der ViewingPlatform
14   ViewingPlatform viewingPlatform = u.getViewingPlatform();
15   viewingPlatform.setNominalViewingTransform();
16   u.getViewer().getView().setFrontClipDistance (0.005);
17   u.getViewer().getView().setBackClipDistance (500.);
18
19   // Orbit Behavior zur ViewingPlatform hinzufuegen
20   orbit = new OrbitBehavior(c, OrbitBehavior.REVERSE_ALL |
         OrbitBehavior.STOP_ZOOM);
21   BoundingSphere bounds = new BoundingSphere(new Point3d
         (0.0, 0.0, 0.0), 10000);
```

```
22      orbit.setSchedulingBounds(bounds);
23      viewingPlatform.setViewPlatformBehavior(orbit);
24
25      // Initiales Erstellen der ViewPlatformTransformation
26      TransformGroup vpTrans = null;
27      vpTrans = u.getViewingPlatform().getViewPlatformTransform();
28      Transform3D T3D = new Transform3D();
29      Vector3f translate = new Vector3f();
30      translate.set(0f, 0f, 25.0f);
31      T3D.setTranslation(translate);
32      vpTrans.setTransform(T3D);
33
34      // Scene zum Universum hinzufuegen
35      u.addBranchGraph(scene);
36  }
```

Die Navigation in der 3D-Ebene setzt sich zum Einen aus der Tastaturnavigation und zum Anderen aus der Mausnavigation zusammen, um eine optimale Änderung der Ansicht in der dritten Dimension zu gewährleisten. Die einzelnen Aktionen auf die Tastatur- bzw. Mauseingaben stellen lediglich eine affine Transformation des ViewPoint dar. Die folgende Übersicht zeigt die Belegung der Tastatur- und Mauskonfiguration:

1. Tastaturkonfiguration

    - ESCAPE — Reset der Viewposition in die Ausgangslage.
    - Pfeil Oben — Bewege Z-Achse (Vorwärts)
    - Pfeil Unten — Bewege Z-Achse (Rückwärts)
    - Pfeil Rechts — Rotiere Y Rechts
    - Pfeil Links — Rotiere Y Links
    - STRG + Pfeil Oben — Bewege Y-Achse (Aufwärts)
    - STRG + Pfeil Unten — Bewege Y-Achse (Abwärts)
    - STRG + Pfeil Rechts — Rotiere Z-Achse Rechts
    - STRG + Pfeil Links — Rotiere Z-Achse Links
    - ALT + Pfeil Oben — Rotiere X-Achse (Hoch schauen)
    - ALT + Pfeil Unten — Rotiere X-Achse (Hinunter schauen)
    - ALT + Pfeil Rechts — Bewege X-Achse (Seitwärts nach Rechts)
    - ALT + Pfeil Links — Bewege X-Achse (Seitwärts nach Links)

2. Mauskonfiguration

    - Selektiere eine Position auf dem Board und halte den linken Mausbutton gedrückt. Nun ist es möglich, das Board durch ein Bewegen der Maus rotieren zu lassen.
    - Selektiere eine Position auf dem Board und halte den rechten Mausbutton gedrückt. Nun befindet man sich im Grab-Modus und man bewegt das Board, als hätte man es in einer virtuellen Hand.

Kapitel 3

Evaluation

3.1 Methodik

3.1.1 Evaluationsphasen

Um eine qualitative Evaluation durchzuführen gibt es drei Phasen. Diese Phasen dienen dazu, ein systematisches Konzept für eine Evaluation zu entwickeln. Im folgenden gehen wir näher auf die Phasen ein.[1]

- Vorbereitungsphase
- Formative Phase
- Summative Phase

Vorbereitungsphase

Die Vorbereitungsphase dient der Zielpräzisierung, da jede Evaluation zielorientiert ist. Sie bedient sich bestimmter Kriterien, für die die Ziele genau formuliert werden müssen, um anschließend präzise Ergebnisse liefern zu können.

Formative Phase

Die formative Phase dient dazu, anhand der Analyseergebnisse das Produkt zu beurteilen und Verbesserungen vorzunehmen. Diese Phase umfasst die Qualitätsanalyse und Wirkungsanalyse. Bei der Qualitätsanalyse geht man einerseits auf

[1] http://dsor.uni-paderborn.de/de/forschung/publikationen/blumstengel-diss/Evaluationsmethodik.html

Ergonomie und Oberfläche ein, wobei die Bildschirmgestaltung in Bezug auf angewandte Farbkombinationen und Navigationskonzepten bewertet wird. Andererseits wird der Inhalt auf Vollständigkeit und Verständlichkeit untersucht. Zuletzt wird Bezug auf die Didaktik genommen, hierbei wird nach pädagogischen Kriterien geprüft, ob die verwendete Gestaltung den definierten Lernzielen gerecht wird. Bei der Wirkungsanalyse werden Testgruppen befragt und beobachtet.

Summative Phase

Die summative Phase tritt in Kraft, wenn die Entwicklung weitgehend abgeschlossen ist. Hierbei spielen Kosten-, Nutzen- und Effizienzanalyse eine wichtige Rolle.

3.2 Durchführung

Die Auswertung unseres Projektes wurde anhand der Ergebnisse verwirklicht, die am Tag der offenen Tür gesammelt wurden. Wir haben an diesem Tag mehr als 100 Interessenten an unserem Stand empfangen, von denen ein Teil freiwillig an unserer schriftlichen Auswertung teilgenommen hat.
Insgesamt wurden 104 Interessenten anhand der geführten Liste festgehalten, wie in Abbildung 3.1 zu sehen ist:

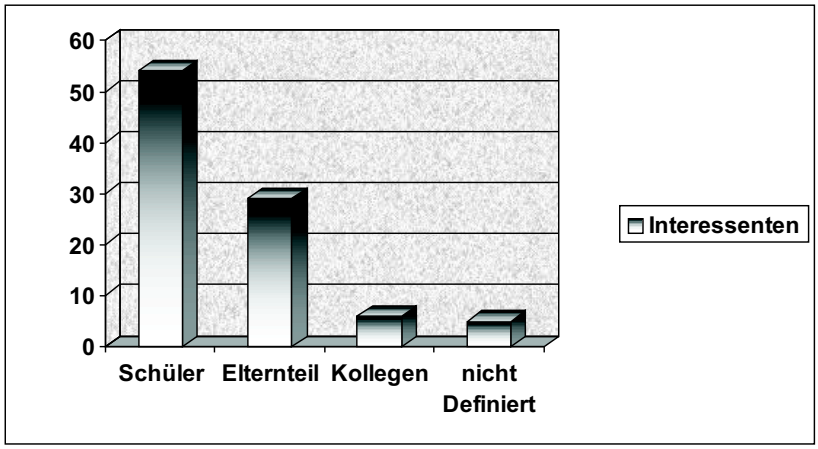


Abbildung 3.1: Gesamtzahl der Interessenten, die unseren Stand besucht haben

Im Anschluss haben sich 31 Personen an der Fragebogenaktion beteiligt, wobei sich diese in 25 Männer und 6 Frauen unterteilen.

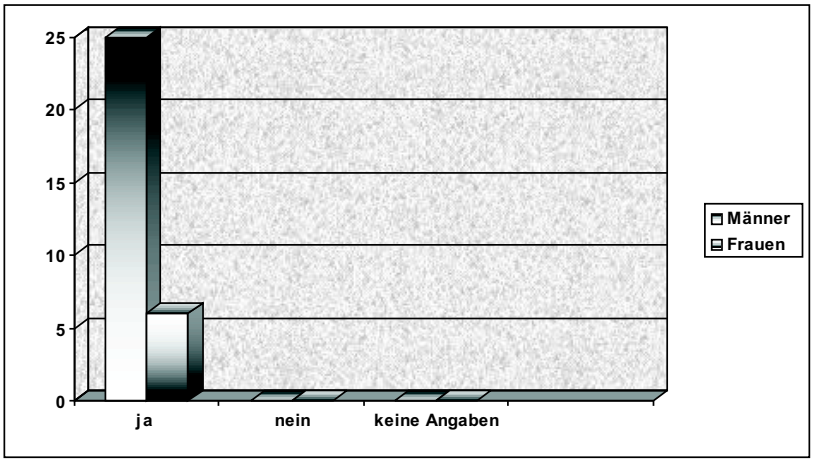
3.3 Ergebnisse

Unser Fragebogen setzte sich aus drei Teilbereichen zusammen, wobei im einzelnen Fragen zur Software, Hardware und den individuellen Vorkenntnissen gestellt wurden.
Anschließend zeigen die Diagramme die Ergebnisse zu den jeweiligen Fragen.
Hierbei beschreibt die vertikale Achse die Anzahl der beteiligten Personen, während die horizontale Achse sich auf die Antworten bezieht.

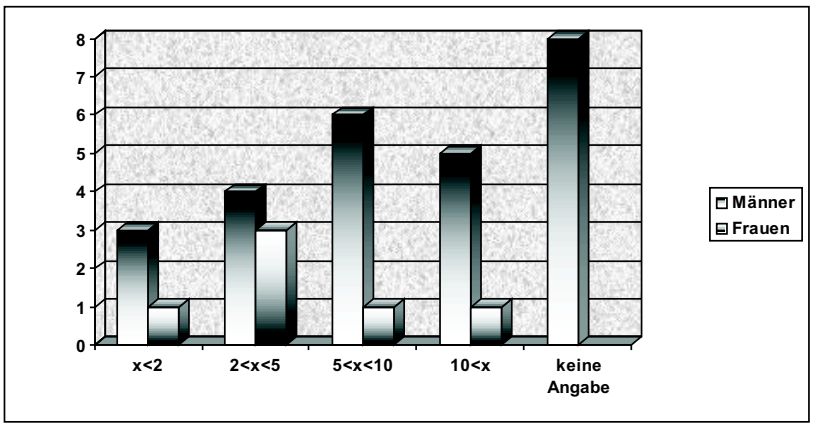
Fragebogen

3.3.1 Fragen über allgemeine Vorkenntnisse mit dem Rechner

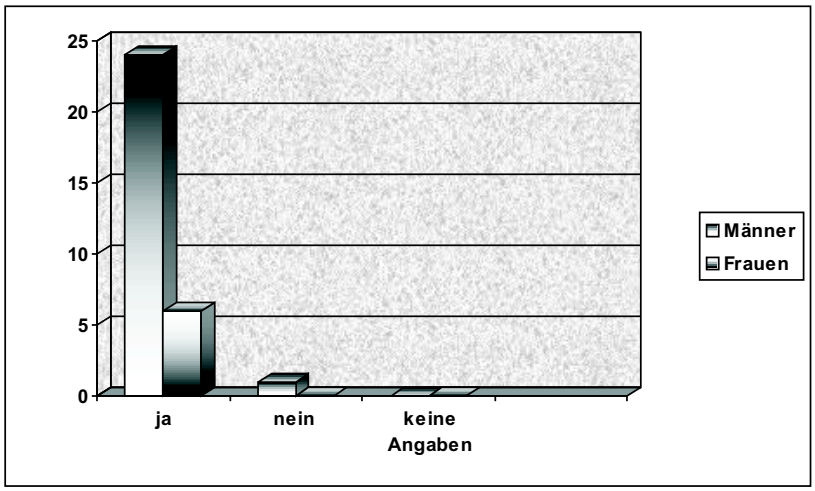
1. Ist ein Computer im Haushalt vorhanden?



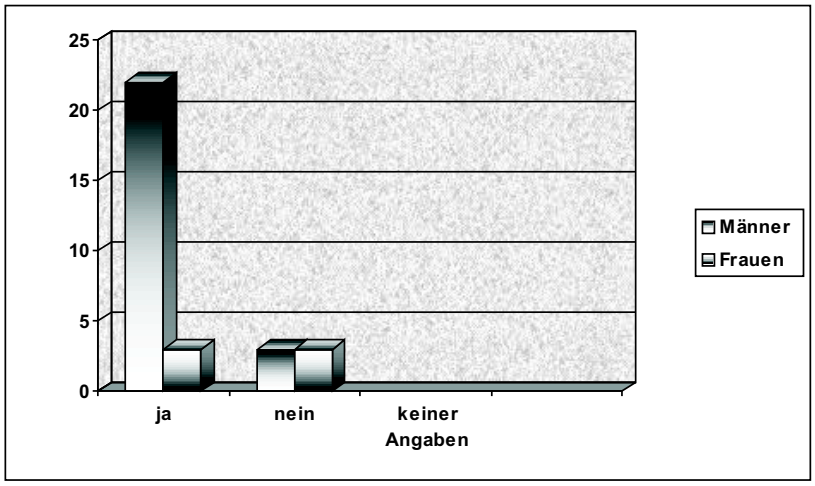
2. Seit wann am Computer?



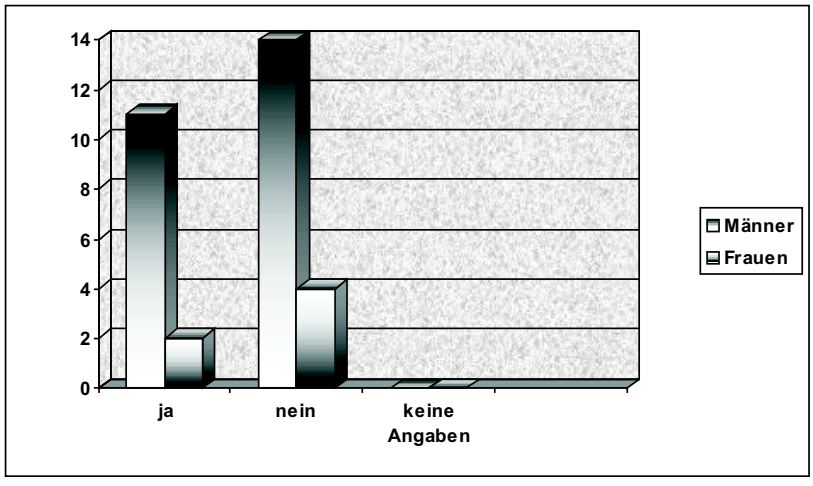
3. Hast Du Erfahrung mit dem Computer?



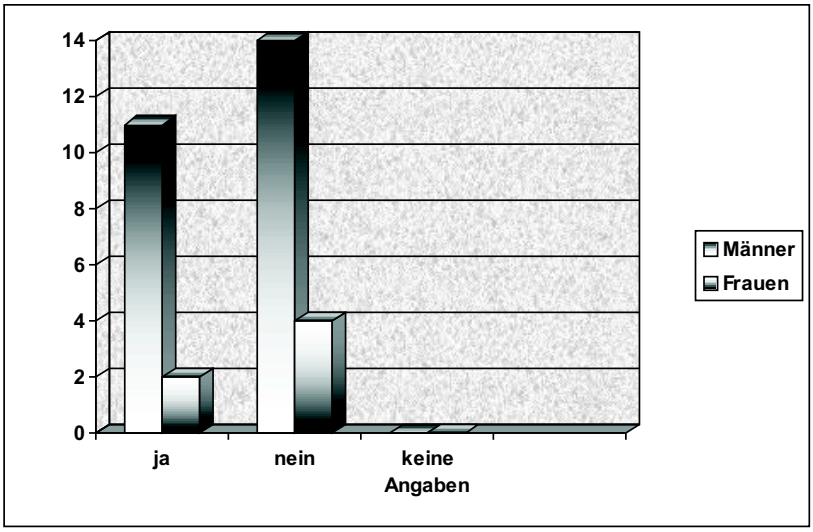
4. Bereits Programmiererfahrungen?



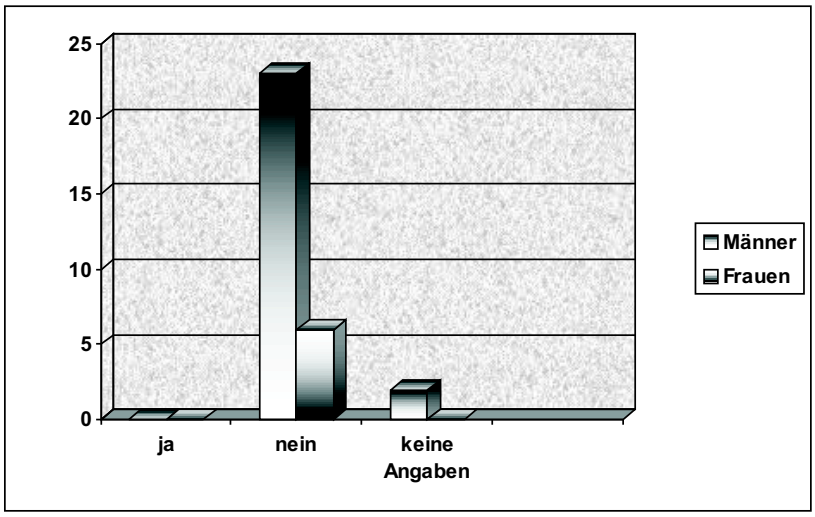
5. In der Schule Informatik Unterricht gehabt?



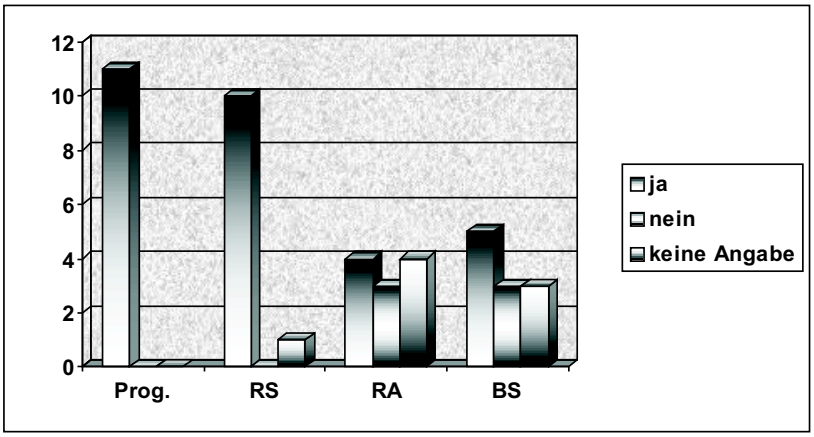
6. Studierst Du Informatik?



7. Studierst Du Informatik im Nebenfach?

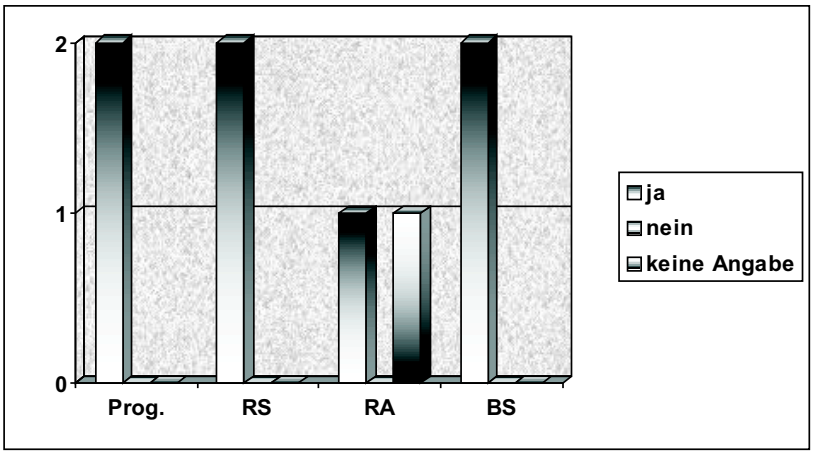


8a. Falls Du Informatik im Haupt- oder Nebenfach studierst, welche Vorlesungen hast Du bereits gehört?



Antworten der männlichen Befragten

8b. Falls Du Informatik im Haupt- oder Nebenfach studierst, welche Vorlesungen hast Du bereits gehört?



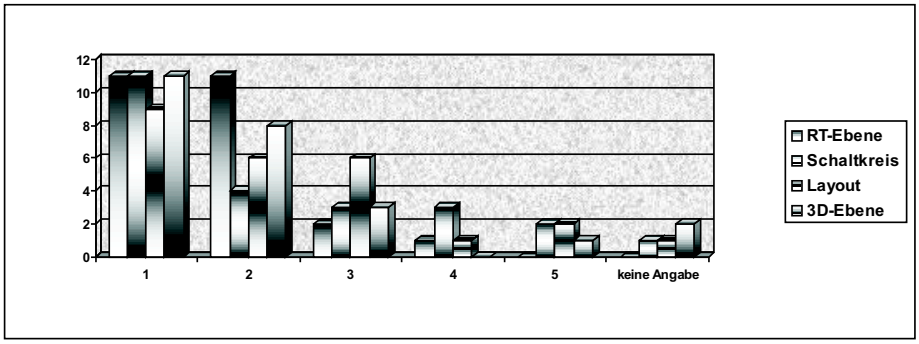
Antworten der weiblichen Befragten

3.3.2 Fragen zu unserer PoCiLet-Software:

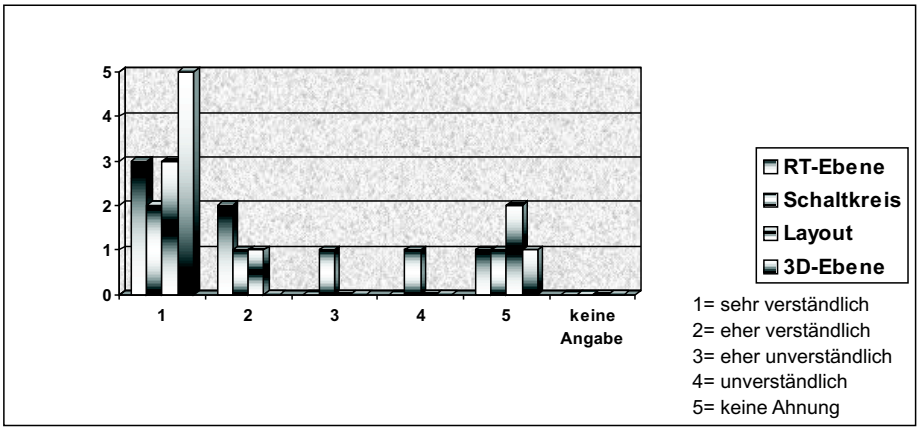
Mit den folgenden Fragen wollten wir herausfinden, wie gut die Oberfläche und die Farbwahl zum Verständnis der Bedienung des Programms beitrug. Ebenso wollten wir wissen, ob Mängel auftraten und inwiefern Verbesserungen vorgenommen werden können. Bei einigen Fragen war es interessant festzustellen, wie unterschiedlich die Ergebnisse zwischen Männern und Frauen ausfielen.

Die beiden folgenden Abbildungen zeigen, wie stark unsere Software zur Verständnis der verschiedenen Ebenen beitrug. Dabei sollten die Versuchspersonen PoCiLet auf einer Skala von 1 (sehr Verständlich) bis 4 (unverständlich) beziehungsweise 5 (keine Ahnung) bewerten. Die Mehrheit empfand die einzelnen Ebenen sehr versändlich.

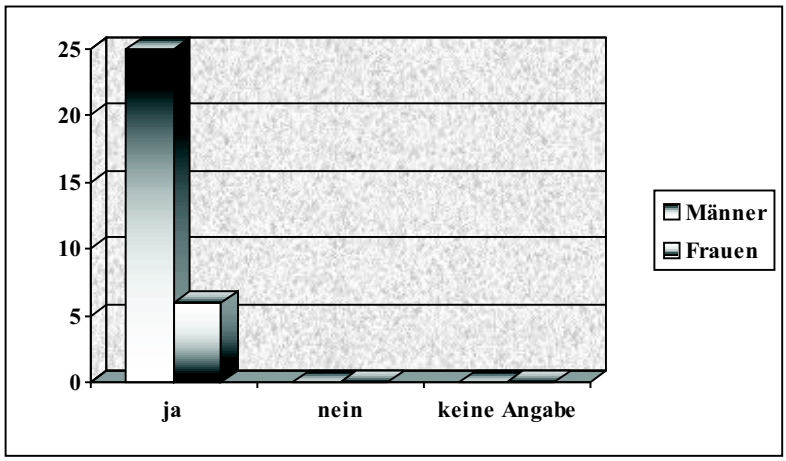
9a. Verständnis der verschiedenen Ebenen? *Männer*



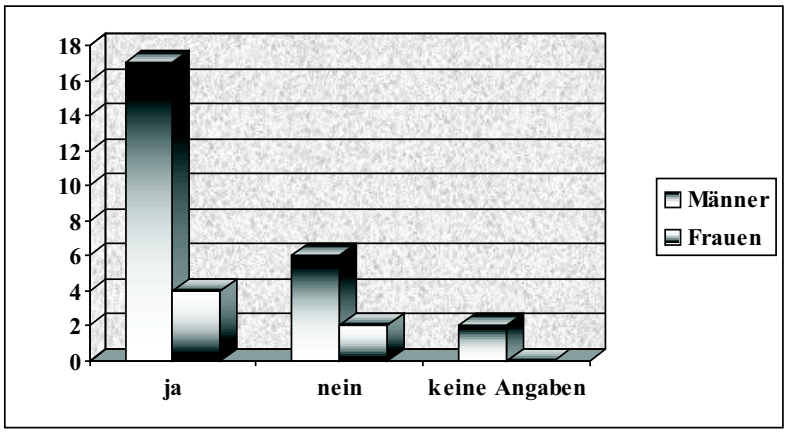
9b. Verständnis der verschiedenen Ebenen? *Frauen*



10. Kannst Du einen Bezug zwischen den einzelnen Ebenen finden?



11. Kannst Du alle Komponenten in allen Darstellungen wieder finden?



12. Wenn nein, was konntest Du nicht finden? (hinschreiben)

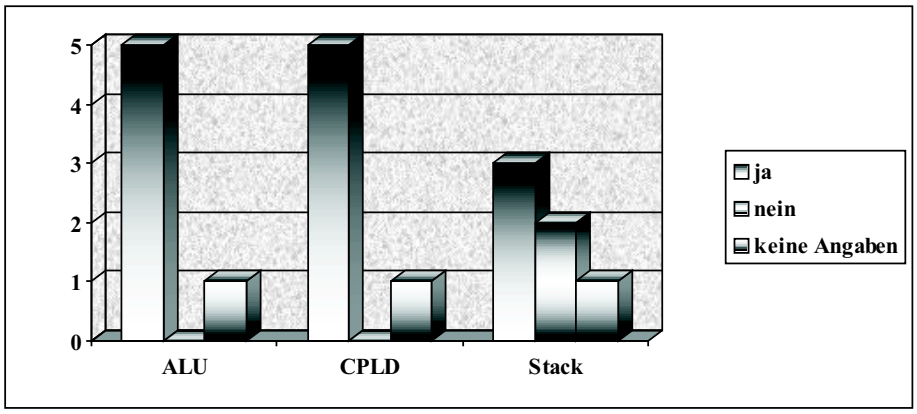
Die folgende Aufzählung enthält konkrete Antworten der befragten Personen:

- ROM
- Register-Transfer-Ebene
- Bus
- Das meiste

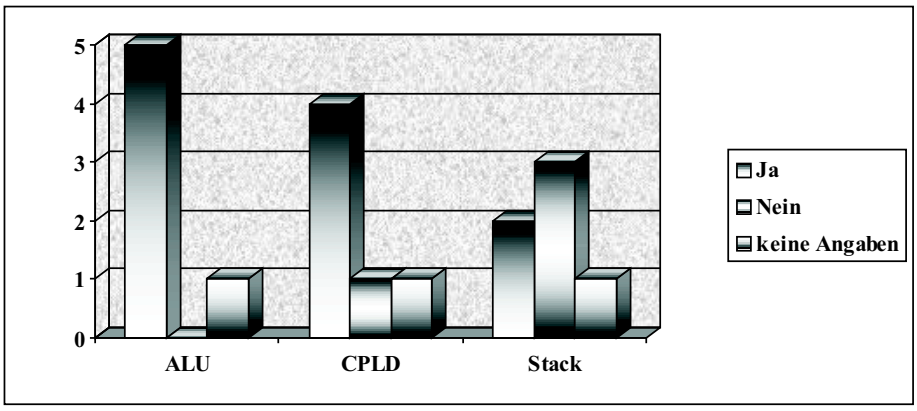
13. Wenn nein: Findest Du folgende Komponenten in den einzelnen Ebenen wieder?

Wie aus den Diagrammen zu entnehmen ist, sind ALU und Steuerwerk gut zu finden, während das Stackregister nicht ohne weiteres auffindbar war. Dabei ist zu erwähnen, dass das Stackregister in den Ebenen nicht explizit benannt ist, was wahrscheinlich die Suche erschwert.

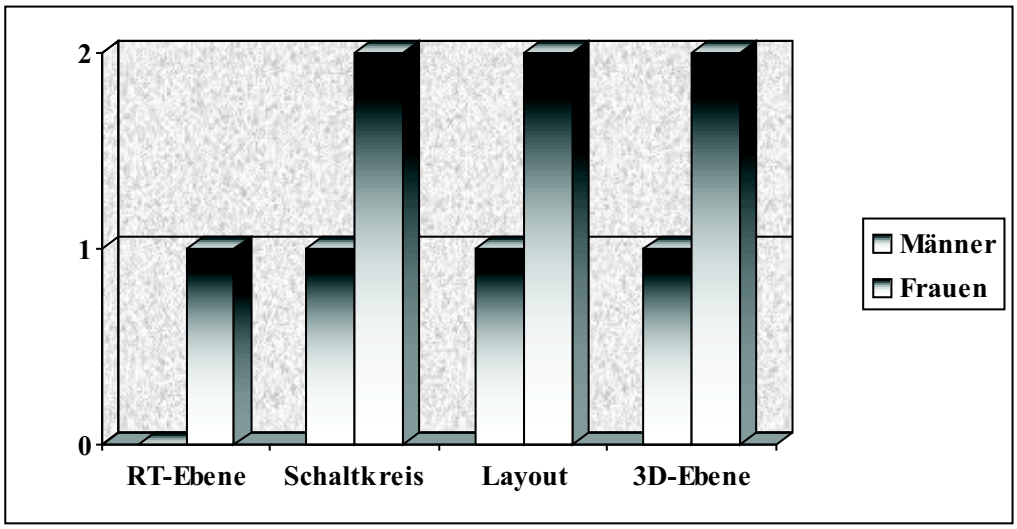
13a. Findest Du folgende Komponenten in den einzelnen Ebenen wieder? ***Männer***



13b. Findest Du folgende Komponenten in den einzelnen Ebenen wieder? ***Frauen***



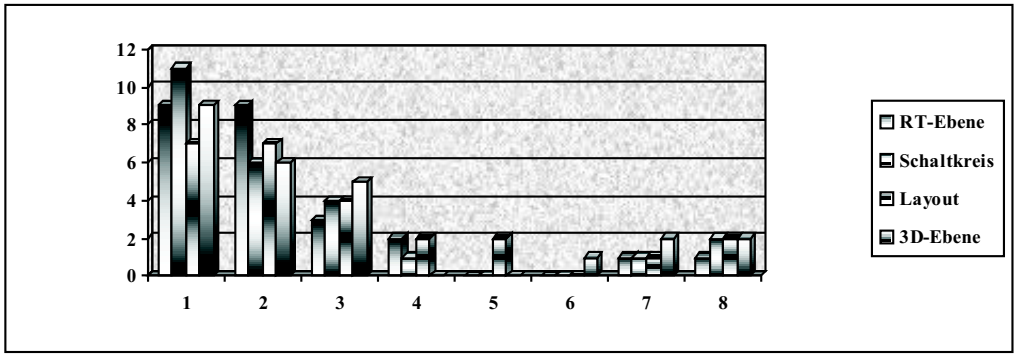
14. Falls Du eine Komponente nicht finden konntest, in welcher Ebene konntest Du die Komponenten nicht finden?



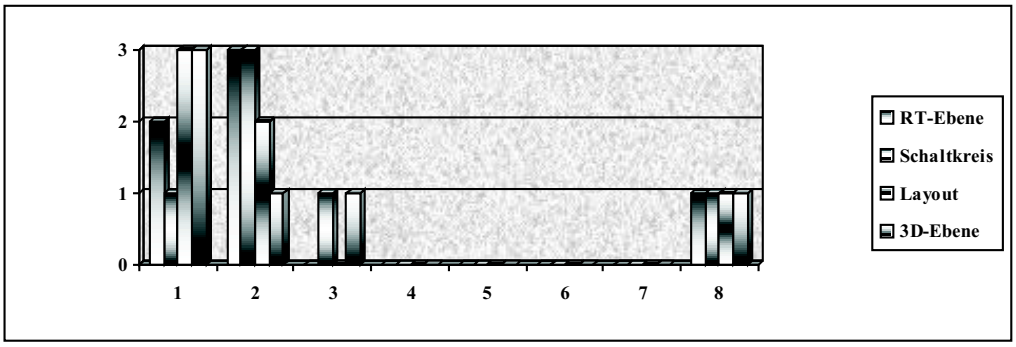
15. Bewerte die Farbgebung der einzelnen Ebenen (Signale und Leitungen etc.)

Die Auswertung zeigt, dass sowohl Männer als auch Frauen überwiegend die Farbgestaltung mit sehr gut oder gut einstuften. Hierbei sollten sie in einer Skala von 1 (sehr gut) bis 6 (sehr schlecht) beziehungsweise 7 (keine Angaben) bewerten.

15a. Männer



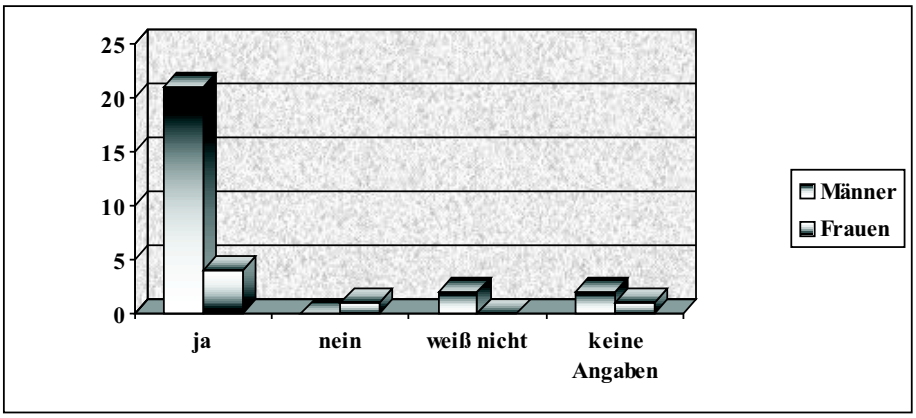
15b. Frauen



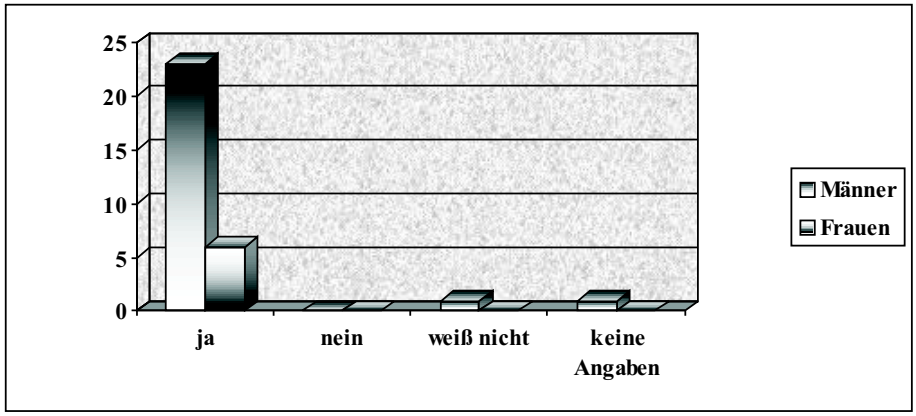
3.3.3 Fragen zu unserer PoCiLet-Hardware?

Nach einer kleinen Einführung unsererseits zeigen die folgenden Abbildungen, wie sich die Interessenten auf der Hardware zurechtgefunden haben.

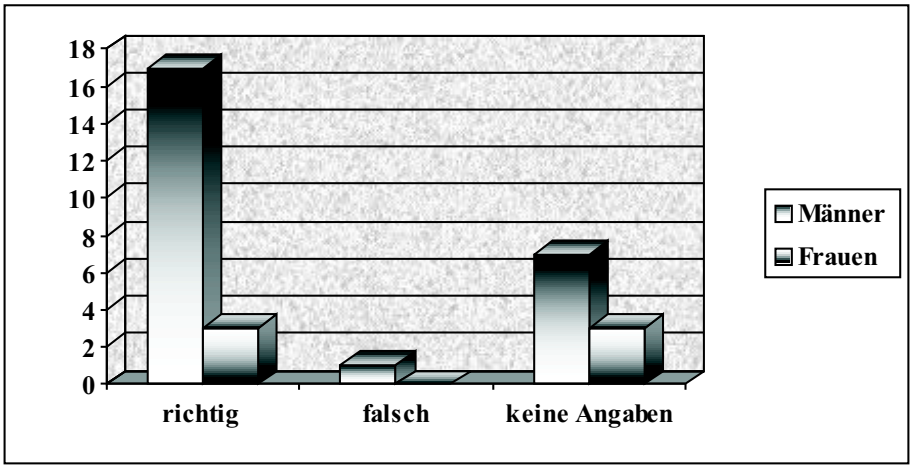
16. Wie haben Sie sich auf der Hardware zurechtgefunden?



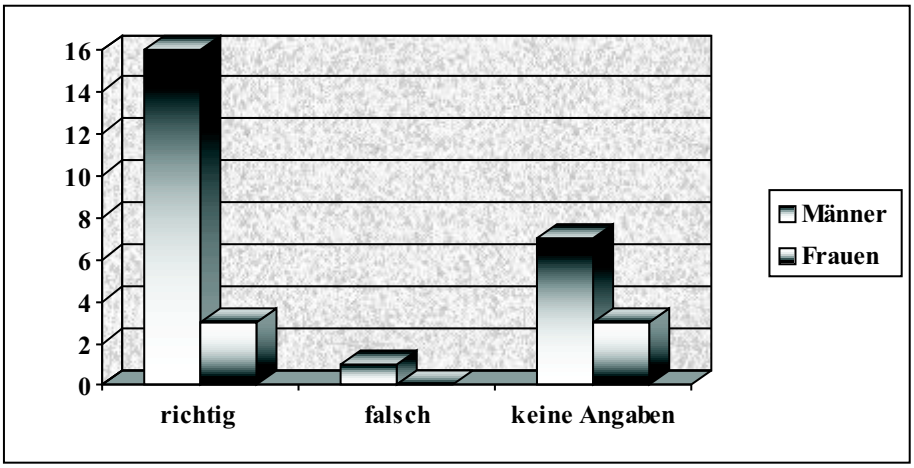
17. Findest Du markante Komponenten wieder (z.B. ALU, Steuerwerk)?



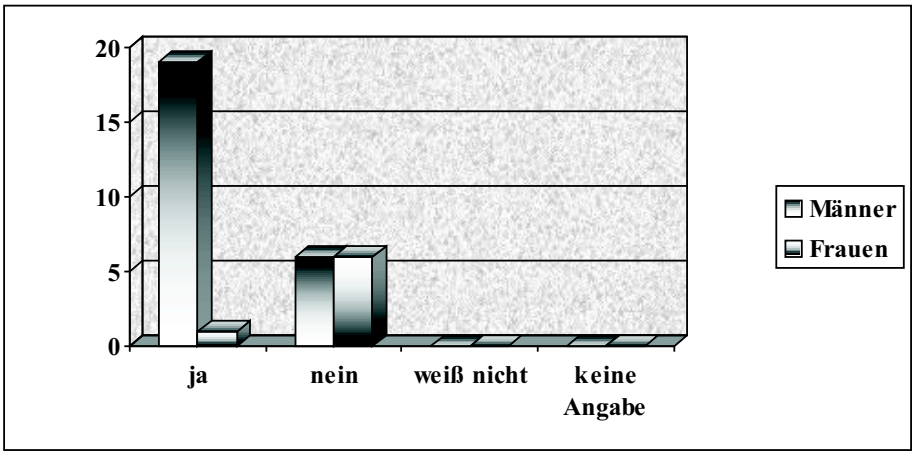
18a. Wenn Du sie finden kannst, beschreibe wo die ALU auf dem Board zu finden ist (insgesamt 6 Frauen, 25 Männer).



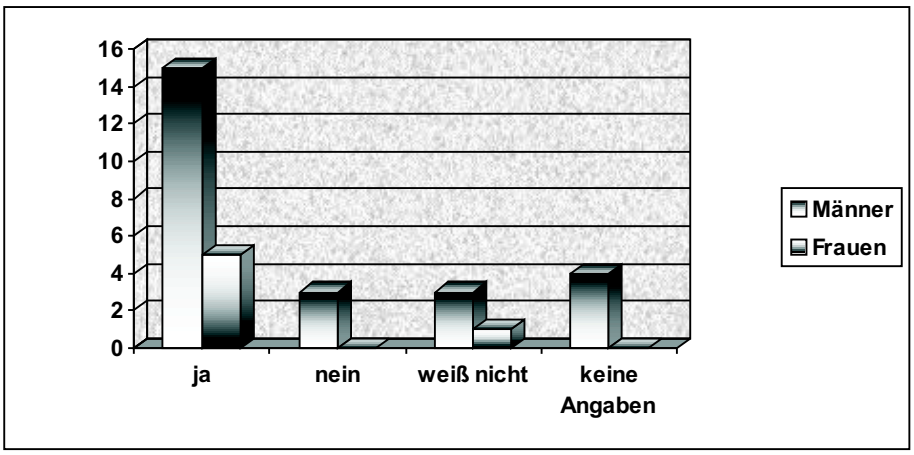
18b. Wenn Du sie finden kannst, beschreibe wo das Steuerwerk auf dem Board zu finden ist (insgesamt 6 Frauen, 25 Männer).



19. Hast Du vorher schon einmal eine Platine näher betrachtet?



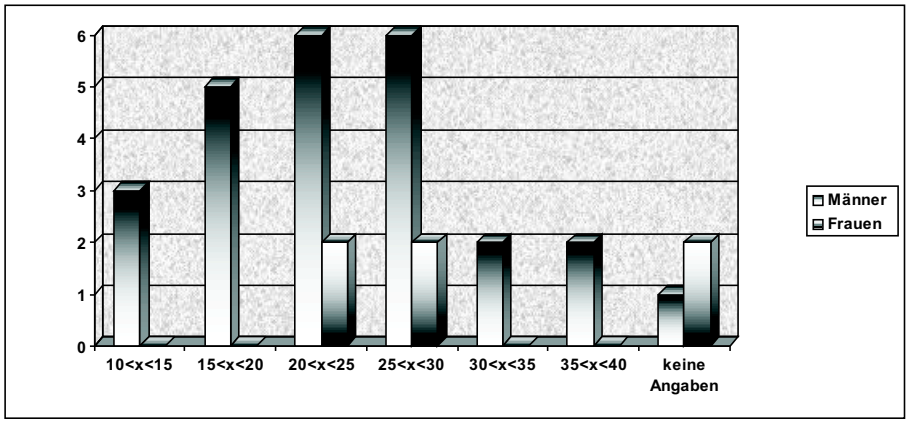
20. Findest du die Hardware leicht bedienbar?



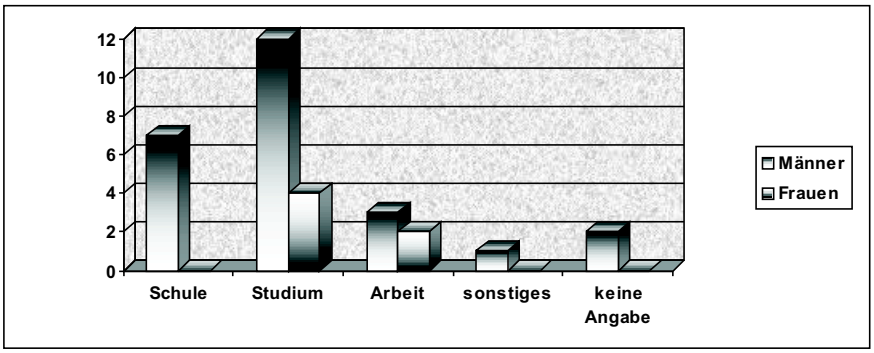
21. Persönliche Angaben der Interessenten

Um einen groben Überblick über den Alterskreis und der beruflichen Tätigkeit der Interessenden zu geben, sind die beiden folgenden Diagramme aufgeführt.

21a. Alter



21b. Beruf



3.4 Bewertung der Ergebnisse

Zunächst erstmal möchten wir darauf hinweisen, dass die Auswertung dieser 31 Fragebögen nur bedingt statistische Relevanz hat, da für den Erhalt eines signifikanten Ergebnisses wesentlich mehr Versuchspersonen hätten befragt werden müssen, als verfügbar waren.

Die Beobachtungen während der Untersuchungen zeigen jedoch, dass diese Ergebnisse ohne Vorkenntnisse nicht so leicht erzielt worden wären. Es musste zunächst eine kleine Einführung in die Thematik unsererseits erfolgen, damit die Interessenten eigenständig zurecht kamen. Was für uns sehr interessant war, sind die Ergebisse der Verbesserungsvorschläge zu unserer Software und Hardware.

Verbesserungsvorschläge zur Software:

- Höhere Programmiersprache, statt Assembler.
- Geschwindigkeitssteigerung
- Optionales Ausschalten der 3D-Ansicht
- Anderes/wechselbares Aussehen
- Verfügbarkeit im Internet zum Einsetzen in der Lehre
- Verbesserung der Zeichnungen
- Besseres Highlighting

Verbesserungsvorschläge zur Hardware:

- Knöpfe Leichtgängiger machen

Anhand dieser Ideen konnten noch Verbesserungen, wie z.B. das Ausschalten der 3D-Ansicht, an der Software vorgenommen. Weitere und größere Änderungen konnten allerdings im Rahmen dieser Projektgruppe nicht mehr vorgenommen werden.

3.5 Die Teststrategie

Die Teststrategie des Softwareteils des Projekts PoCiLet sieht eine möglichst umfassende Abdeckung aller dazugehörigen Klassen und möglichen Testfälle vor. Dennoch wurde im Verlauf der Entwicklung schnell klar, dass schon relativ früh Kompromisse zwischen möglichst umfangreichem Testen und benötigtem Arbeitsaufwand, also Nutzen und Kosten, gefunden werden mussten.

Nach Riedemann [Rie97] ist ein weiterer Anspruch an die Teststrategie, dass das Testen in einem möglichst frühen Stadium der Entwicklung einsetzen soll. So soll garantiert werden, dass Programmier- und Designfehler möglichst früh erkennbar werden. Dadurch wird eine Anhäufung von Fehlern durch das gesamte Projekt und das daraus resultierende Anwachsen des Test- und Korrekturaufwands, vermieden. Ein weiterer Vorteil des frühen Testens ist, dass dies das angestrebte hierarchische und inkrementelle Testkonzept von Klassen und Paketen vereinfachen soll. So kann man für Einzeltests von Klassen oder Paketen jeder Zeit auf schon positiv getestete Methoden oder Subklassen zurückgreifen, ohne dass man diese extra in dem aktuellen Prozess verifizieren muss.

Da das Testen innerhalb des Softwareprojekts überwiegend von uns selbst, also den verantwortlichen Entwicklern, und während der Entwicklung durchgeführt wurde, konnten und sollten die Tests weitgehend im Bereich *Whitebox* angesiedelt werden, also unter genauer Kenntnis des inneren Aufbaus der Testkandidaten geschehen. Es wurden demnach hauptsächlich Funktionentests, Klassentests und Modultests durchgeführt. Alle genannten Tests wurden der Strategie nach auf bereits getesteten Komponenten durchgeführt.

Das Testen der Softwareelemente in PoCiLet musste in zwei getrennte Bereiche unterteilt werden, die sehr unterschiedliche Teststrategien erforderten:

Auf der einen Seite das Testen der grafischen Elemente und auf der anderen Seite das Testen der darunterliegenden, nicht sichtbaren, funktionellen Komponenten der GUI sowie der Elemente des softwaremäßigen PoCiLet-Simulators. Beide Teile werden im Anschluss separat erläutert.

Grafische Elemente:

Auch für das Testen der einzelnen grafischen Klassen waren ausgiebige und möglichst automatisierte Tests vorgesehen, die als Minimum alle vorkommenden Anwendungsfälle abdecken sollten. Für einfache Tests von Knöpfen oder Menüeinträgen ist es leicht entsprechende Testklassen zu kreieren, die textuelles Feedback

über die reine Funktionalität der Testkandidaten liefern. Solche Tests wurden während der Entwicklung zu allen GUI Elementen erstellt und im weiteren Verlauf durch Tests mit darunterliegenden Komponenten ersetzt. Allerdings wurde schnell deutlich, dass es weniger leicht zu bewerkstelligen ist, für diese Elemente automatische Regressionstests (regelmäßige Wiederholungen von Tests mit Abgleichung der Ergebnisse) zu erstellen. Um dies zu ermöglichen, müssten Eingaben in die Benutzeroberfläche automatisiert erfasst werden und ebenso die Rückgaben von dort automatisch auszulesen und zu vergleichen sein. Identische Anforderungen an eine Testumgebung stellen auch Tests über das komplexe Zusammenspiel der einzelnen grafischen Komponenten untereinander, womit dies schon zwei Bereiche sind, die ohne Hilfsmittel nicht wie gewünscht abzudecken waren. Nach Erkundigungen über einige Testumgebungen (z.B. WinRunner, SilkTest), die die nötigen Anforderungen erfüllen würden, sind wir zu der Überzeugung gelangt, dass diese sowohl durchweg kommerziell, und somit sehr kostenintensiv, als auch sehr komplex zu bedienen wären [iX101]. Nach Abwiegen von Für und Wider haben wir erkannt, dass sowohl Preis als auch Einarbeitungsaufwand in einer nicht akzeptablen Relation zu dem erreichbaren Nutzen stehen. Folglich haben wir beschlossen, die Tests der einzelnen GUI Elemente mehr oder weniger von Hand durchzuführen und die Ergebnisse festzuhalten.

Einzelne einfache Klassen- und Modultests wurden wann immer möglich und sinnvoll durchgeführt. Komplexere Tests über die Gesamtfunktionalität wurden in Form von Integrations- und Systemtests durchgeführt, da die GUI dort auf Rückgaben aus anderen Teilen des Projekts angewiesen ist und es zu zeitaufwendig war, extra derart komplexe Testumgebungen zu erstellen.

Funktionelle Komponenten:
Das Testen des funktionellen Teils der PoCiLet Software sollte sich streng an die eben definierte Strategie halten, was also heißt:

- Frühzeitiges Testen
- Inkrementelles und hierarchisches Testen
- Automatisiertes Testen und Vergleichen von Ist-/Sollwerten
- Ständiges, automatisiertes Wiederholen von Tests

Zusätzlich sollte der Aufwand und die Zeit, die in das Testen flossen, möglichst gering gehalten werden. Hierzu mussten wir eine geeignete Testumgebung finden, die uns bei der Erstellung und Vereinfachung der Tests unterstützen konnte.

Nahezu alle gestellten Anforderungen werden durch das relativ weit verbreitete Testframework JUnit erfüllt. JUnit ist ein kostenloses Werkzeug, das unter der IBM Public License ständig weiterentwickelt wird und durch einen großen Umfang an erhältlichen Plugins an eine große Anzahl von Entwicklungsumgebungen angepasst werden kann [Lin02]. Es bietet umfangreiche, vordefinierte Testklassen und -methoden, die als Basis oder Rahmen für selbstdefinierte Testfälle eigener Klassen dienen können. JUnit unterstützt stark den angestrebten hierarchisch aufgebauten Testablauf über ein komplettes Projekt, indem es ermöglicht, einzelne Testfälle zu Testklassen und diese weiter zu sogenannten Testsuites zusammenzufassen. Es wird empfohlen, erstellte Testklassen in die im Projekt bestehende Paketstruktur einzugliedern, wodurch das Testen praktisch in das Entwickeln integriert wird.

Es wurden dann zu allen Testkandidaten Testklassen erstellt, die alle nötigen Testfälle enthalten, welche dann in jedem Paket zu einer Pakettestsuite zusammengefasst wurden. Es folgt als Beispiel ein Auszug aus der Testklasse zu der PoCiLet Klasse "Program":

Listing 3.1: Beispiel - Ausschnitt ProgramTest

```
1  package pocilet.simmanager;
2  import junit.framework.Test;
3  import junit.framework.TestCase;
4  import junit.framework.TestSuite;
5  import pocilet.simmanager.Program;
6
7  public class ProgramTest extends TestCase {
8    private Program _program;
9    private String _defaultString;
10
11   // Constructs a ProgramTest with the specified name.
12   public ProgramTest(String name) {
13     super(name);
14   }
15
16   // Sets up the test environment
17   // Called before every test case method.
18   protected void setUp() {
19     _program = new Program();
20     _program.addLineInAscii("testLine1");
21   }
22
23   // Deletes test environment.
24   // called after every test case method.
25   protected void tearDown() {
```

```
26      _program = null;
27    }
28
29    // Tests getting length of the program.
30    public void testProgramGetLengthOfProgram() {
31      assertEquals(1, _program.getLengthOfProgram());
32    }
33
34    // Tests getting line of the program.
35    public void testProgramGetLineAscii() {
36      assertEquals("testLine1", _program.getLineInAscii(0));
37    }
38
39    // Tests adding a line to the program.
40    public void testProgramAddLineAscii() {
41      _program.addLineInAscii("testLine2");
42      assertEquals(2, _program.getLengthOfProgram());
43      assertEquals("testLine2", _program.getLineInAscii(1));
44    }
45
46    // Tests the clearing of the program.
47    public void testEmpty() {
48      _program.initAscii();
49      assertEquals(0, _program.getLengthOfProgram());
50    }
51
52    // Assembles and returns a test suite for this test case
53    public static Test suite() {
54      TestSuite suite = new TestSuite(ProgramTest.class);
55      return suite;
56    }
57
58    // Runs the test case.
59    public static void main(String args[]) {
60      junit.textui.TestRunner.run(suite());
61    }
62  }
```

Das obige Beispiel zeigt Testfälle für die Methoden GetLengthOfProgram() (Z. 30-32), GetLineASCII() (Z. 35-37), AddLineASCII() (Z. 40-44) und InitASCII() (Z. 47-50). Weiter gibt es die Methoden SetUp() und TearDown() (Z. 18-27), um die Umgebung vor und nach Tests zu initialisieren bzw. zu löschen, sowie Methoden, die Testfälle zusammenzufassen und zu starten.

Als zweites Beispiel geben wir einen Teil der Testsuite des Pakets Simmanager, in dem sich Program befindet, an.

Listing 3.2: Beispiel - Ausschnitt SimmanagerTestSuite

```
1  package pocilet.simmanager;
2  import junit.framework.Test;
3  import junit.framework.TestSuite;
4
5  public class SimmanagerTestSuite {
6    // Assembles and returns a test suite
7    // containing all known tests.
8    // New tests should be added here!
9    public static Test suite() {
10
11     TestSuite suite = new TestSuite();
12     // add ProgramTest Suite
13     suite.addTest(ProgramTest.suite());
14     return suite;
15   }
16
17   // Runs the test suite.
18   public static void main(String args[]) {
19     junit.textui.TestRunner.run(suite());
20   }
21 }
```

Man sieht, dass es hier nur zwei Abschnitte gibt; im ersten werden alle gewünschten Suites zusammengefasst (Z. 9-15), im zweiten wird das Starten der Tests ermöglicht (Z. 18-20).

So waren wir in der Lage, regelmäßig während der Entwicklung Tests über einzelne Funktionen, Klassen, Pakete oder das gesamte Projekt durchzuführen, ohne großen Extraaufwand zu generieren. Die Durchführung selbst stellte sich denkbar einfach dar, indem einfach das entsprechende Testmodul gestartet wurde. Die Ergebnisse eines Testdurchlaufs konnten bequem textuell aus der Konsole oder über eine in das JUnit Paket integrierte GUI abgelesen, sowie Fehler leicht lokalisiert und bearbeitet werden.

Kapitel 4

Fazit & Ausblick

4.1 Persönliches Fazit

Wir haben in der Zeit der Durchführung der PG verschiedene neue Technologien kennen gelernt und erfahren wie man diese einsetzt. Damit haben wir unseren eigenen Wissenshorizont sinnvoll erweitert, um ein tieferes Fachwissen in den Bereichen der Hardware- und Softwareentwicklung zu erlangen. Durch das abwechslungsreiche Thema der PG, haben wir im Laufe der Zeit aber nicht nur unseren eigenen Wissenshorizont in verschiedenen Bereichen der Informatik erweitert, sondern wir haben darüber hinaus das Zusammenspiel dieser Bereiche für uns optimiert und die Nutzer des von uns entwickelten Lehr- und Lernobjektes PoCiLet dabei nicht aus den Augen verloren. Gerade diese Praxisorientierung und die Kombination und das vernetzte Denken in verschiedenen Fachbereichen der Informatik erweist sich als großer persönlicher Gewinn für jeden einzelnen von uns, da im späteren Beruf gerade dies von der heutigen Wirtschaft verstärkt verlangt wird.

Besonders die Vielschichtigkeit des Projektes setzte eine gut durchdachte Organisation voraus. Ergebnisse der Teilgruppen wurden in den PG-Sitzungen immer wieder kontrovers diskutiert, wodurch die Qualität der Arbeit jedes einzelnen deutlich zugenommen hat und die Teamfähigkeit gestärkt wurde. Die Aufteilung der Gruppe in kleinere Teilgruppen führte zu einer vernünftigen Planung und Selbstorganisation der Teilgruppen, wodurch einige von uns auch Erfahrungen im Bereich der Gruppenleitung und Gruppenorganisation sammeln konnten. Innerhalb der letzten zwei Semester wurde dadurch ein großes Maß an praktischer Erfahrung gesammelt und wir haben gelernt, ein großes Problem durch Zerlegung in Teilprobleme und koordinierte Bearbeitung der Teilprobleme in einem Team zu bewältigen.

Wir hatten die Möglichkeit dieses praxisorientierte Ergebnis in einer offizielen Präsentation zu erproben. Der Tag der offenen Tür ermöglichte es uns dabei unsere Arbeit nicht nur als akademische Dienstleistung zu betrachten, sondern gerade die wirkliche Präsentation vor einem großen Publikum zu erfahren, genauso wie es möglicherweise bei Präsentationen für Kunden oder Arbeitgeber in unserer späteren Berufslaufbahn erfolgen könnte. Gerade dieses Publikum bestärkte uns in unserer Erfahrung in diesem Jahr ein leistungsstarkes Team geworden zu sein.

Übersicht der verwendeten Technologien in Software und Hardware:

- XML
- Java I18N (Internationalisierung) API
- Eagle EDA Tool/PCB Designer
- Java3D API
- Java Comm API
- Java RXTX
- FPGA Anwendung/Logikentwurf
- Microcontroller-Applikation

4.2 Fachliches Fazit

Die Präsentation des Projekts im Rahmen des Tages der offenen Tür ergab von mehreren Besuchern zusätzlich zum positiven Feedback auch Anregungen, wie PoCiLet erweitert werden könnte. Dazu zählte beispielsweise, dass einzelne Ansichten, wie die Schaltkreis- und Layoutebene, mehr Interaktivität bieten könnten. So könnte in diesen Ebenen, ähnlich wie in der Register-Transfer-Ebene, der Zustand der einzelnen Komponenten in einem Popup-Fenster angezeigt werden. Um die Beziehungen zwischen den Ebenen stärker zu verdeutlichen, könnte die Funktionalität implementiert werden, ein bestimmtes Bauteil, beziehungsweise dessen Repräsentationen, in allen Ebenen zu markieren. Insbesondere könnte auch die 3D-Ebene über die bisher rein anschauliche Funktionalität hinaus interative Elemente bieten, die das beeinflussen der Simulation ermöglichen.
Weiterhin fehlt bisher eine Einstiegsdokumentation, die Benutzer an PoCiLet heranführt. Diese Funktion könnte durch ein interaktives Tutorial realisiert werden.

Bedingt durch die Komplexität des Hardwareaufbaus wurde die Möglichkeit, die Hardwarekomponente nachzubauen, signifikant eingeschränkt. Ansatzpunkte für eine Vereinfachung wären die Verkleinerung des Befehlssatzes, eine Verringerung der Anzahl der Register oder das Ersetzen der Matrixanzeige durch eine LED-Leiste. Dies ließe sich natürlich nur durch einen kompletten Neuaufbau realisieren.
Kritik an der Simulation könnte auf den Mangel an Flexibilität zielen, da der Simulationskern stärker auf Universalität ausgerichtet sein könnte.
Dazu zählt zum Beispiel, dass es nicht mehr nötig sein sollte, Timing und Reihenfolge der Abarbeitung der einzelnen Komponenten bei der Umsetzung der Schaltung beachten zu müssen. Vielmehr sollte der Simulator dieses selbst berechnen können. Dies würde es erleichtern, den Aufbau von PoCiLet zu modifizieren und um weitere Funktionalität zu erweitern. So könnten Perspektiven, den PoCiLet-Rechner um komplexere Hardwaretechniken, wie Caching und Pipelining, zu erweitern, verfolgt werden. Es könnten so verschiedene Hardwarearchitekturen simuliert werden, ohne Änderungen am Simulationskern durchführen zu müssen. Eine Modularisierung des Rechnerdesigns, so dass individuell ein Rechner assembliert werden kann, wäre die Perfektion dieses Konzepts.

Kapitel 5

Benutzerhandbuch

5.1 Installationsanleitung

Um PoCiLet zu verwenden ist eine lauffähige, aktuelle Installation der *Java 2 Runtime Environment (J2RE)* in der Version 1.4 oder höher erforderlich. Falls sie die *J2RE* neu installieren, sollten sie sicher gehen, dass die Installation der *J2RE* keine bestehende Version überschreibt, da diese eventuell von anderen Applikationen benötigt wird.

Minimale Anforderungen für PoCiLet:

- 500 MHz Pentium II
- 10 MB freier Festplattenplatz
- 256 MB freier Speicher für die JVM
- ATI-Radeon/nVidia-Geforce mit $\geq$ 8 MB Grafikspeicher
  (nur für 3D-Unterstützung)

Beachten sie hierzu auch die *J2RE Requirements* und die *Java3D Requirements* unter *http://java.sun.com*.

Installationsschritte:

1. Stellen sie sicher, dass sie eine lauffähige Java 2 (J2RE/JDK) Installation besitzen. Dies können sie am einfachsten, indem sie ihre Umgebungsvariablen einsehen.

   (a) Öffnen sie eine Konsole (Start -> Ausführen: `cmd` unter Windows-Systemen)

(b) Geben sie folgenden Befehl ein: `java -version`

(c) Setzen sie ggf. ihre Umgebungsvariable neu durch:
`set JAVA_HOME={Java.DIR}` wobei `{JAVA.DIR}` ihrem J2RE Pfad entspricht. Beispiel: `set JAVA_HOME=C:\JDK1.4.2_04`

Falls sie in ihrer Konsolenumgebung die Bourne-Shell verwenden, lautet die korrekte Syntax:
`export JAVA_HOME="{JAVA.DIR}"`

2. Die vorhandene J2RE Installation unter {JAVA.DIR} muss nun ergänzt werden durch die *Java3D Runtime Environment*. Die *Java3D Runtime Environment* erhalten sie in der aktuellen Version 1.3.1 unter:
http://java.sun.com/products/java-media/3D/download.html.
PoCiLet wurde für J2RE und Java3D 1.3.1 OpenGL/Win32 entwickelt.

   Führen sie das Java3D Setup aus, und wählen sie als Pfad ihr {JAVA.DIR}. Falls die Installation korrekt verlaufen ist, sollten sie unter ihrem {JAVA.DIR}/bin die Datei J3D.dll finden und unter {JAVA.DIR}/libs/ext die Datei j3dcore.jar. Falls diese Dateien nicht vorhanden sind, haben sie vermutlich den falschen Pfad zu ihrem {JAVA.DIR} gewählt. Falls sie die oben genannten Dateien zwar gefunden haben, diese aber in einem falschen Ordner liegen, versuchen sie entweder eine erneute Installation von Java3D oder kopieren sie die Dateien wie unten beschrieben.

   Manuelle Java3D Installation:
   Kopieren sie die Dateien aus ihrem Java3D-Ordner in ihr {JAVA.DIR} nach folgendem Schema:

   - J3D.dll nach {JAVA.DIR}/bin
   - J3DUtils.dll nach {JAVA.DIR}/bin
   - J3Daudio.dll nach {JAVA.DIR}/bin
   - j3dcore.jar nach {JAVA.DIR}/lib/ext
   - j3dutils.jar nach {JAVA.DIR}/lib/ext
   - j3daudio.jar nach {JAVA.DIR}/lib/ext
   - vecmath.jar nach {JAVA.DIR}/lib/ext

3. Um PoCiLet zu starten, führen sie `run.sh` unter Linux aus, oder `run.bat` unter Windows. Falls sie Probleme feststellen sollten, also PoCiLet nicht korrekt gestartet wird, öffnen Sie `run.bat` (respektive `run.sh` unter Linux) in einem Texteditor und ergänzen sie manuell den Pfad zu ihrem `JAVA.DIR`, so dass ihre `java.exe` (`java` unter Linux) gefunden wird.

5.2 Übersicht der GUI Komponenten

Die PoCiLet-GUI wird mit dem Befehl run.bat unter Windows oder run.sh unter Unix-System gestartet. Es erscheint zunächst ein Auswahldialog, über welchen man die 3D-Ebene optional aktivieren kann. Anschließend öffnet sich das Hauptfenster.

Dieses ist in vier Bereiche aufgeteilt. Im oberen Bereich findet man eine Menüleiste und darunter eine Toolbar. Im unteren Bereich sind drei weitere kleine Fenster zur Anzeige diverser Status- und Informationsmeldungen angeordnet. Den größten Bereich bildet der mittlere Teil, in dem die vier verschiedenen Ansichten der PoCiLet-Architektur angezeigt werden können.

Im folgenden werden die einzelnen Komponenten und ihre Bedienung erläutert.

5.3 Der PoCiLet Programmeditor

5.3.1 Das Erstellen eines Programms

Um ein neues Programm erstellen zu können, wählen Sie im Menü Datei den Menüpunkt Neues Programm. In das sich öffnende Editor Fenster können Sie jetzt ein Programm eingeben.

Wenn Sie die Eingabe abgeschlossen haben können Sie das Programm mit dem Button übernehmen in den PoCiLet Simulator übernehmen, mit dem Button speichern das Programm speichern oder mit dem Button schliessen das Editor Fenster schliessen.

Möchten sie ihr Programm übernehmen, wird das Programm automatisch an den Parser gereicht und auf syntaktische Fehler überprüft. Falls das Programm syntaktische Fehler enthält, wird jeweils der erste Fehler im Editor Fenster rot unterlegt und eine kurze Erklärung erscheint im unteren Bereich des Fensters. Falls das Programm fehlerfrei ist, wird das Programm in Simulator übernommen und das Editor Fenster geschlossen.

Möchten sie Ihr Programm speichern, so öffnet sich ein Datei-speichern Dialog und sie können auswählen, wo das Programm abgelegt werden soll.

Falls Sie schliessen auswählen wird das Editor Fenster geschlossen und die Eingabe verworfen.

5.3.2 Ein Programm editieren

Um ein bereits geladenes Programm zu editieren wählen sie im Menü Datei den Menüpunkt Programm editieren. Das Editor Fenster öffnet sich daraufhin mit dem aktuellen Programm. Sie haben jetzt die Möglichkeit das Programm zu editieren. Des weiteren verhält sich das Fenster wie bei dem Erstellen eines neuen Programmes (siehe Abbildung 5.3.1).

5.4 Erste Schritte im PoCiLet Simulator

5.4.1 Starten/Stoppen des Simulators

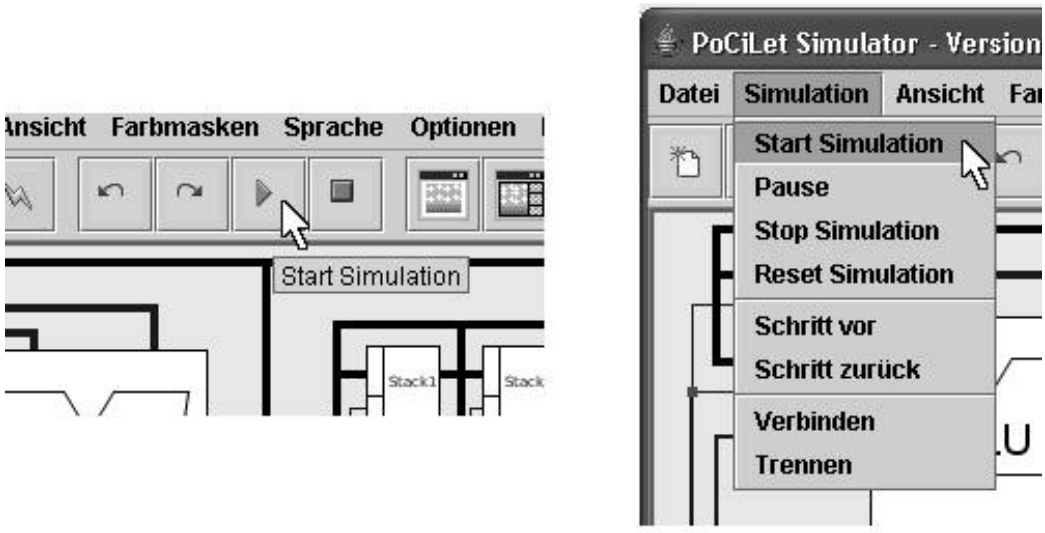


Abbildung 5.1: Starten des Simulators

Um den Simulator zu starten, muss ein Programm geladen worden sein. Wie Sie ein Programm erstellen können Sie in 5.3.1 nachlesen.

Sie haben nun zwei Möglichkeiten die Simulation zu starten. Klicken Sie entweder unter dem Menüpunkt Simulation auf Simulation starten oder betätigen Sie den Button mit dem grünen Pfeil (siehe Abbildung 5.1). Wenn Sie den Simulator das erste Mal starten, nachdem Sie ein neues Programm geladen haben, wird zunächst ein Initialisierungsschritt durchgeführt. Jeder weitere Start des Simulators arbeitet jeweils alle Einzelschritt bis zum Ende des aktuellen Befehls ab.

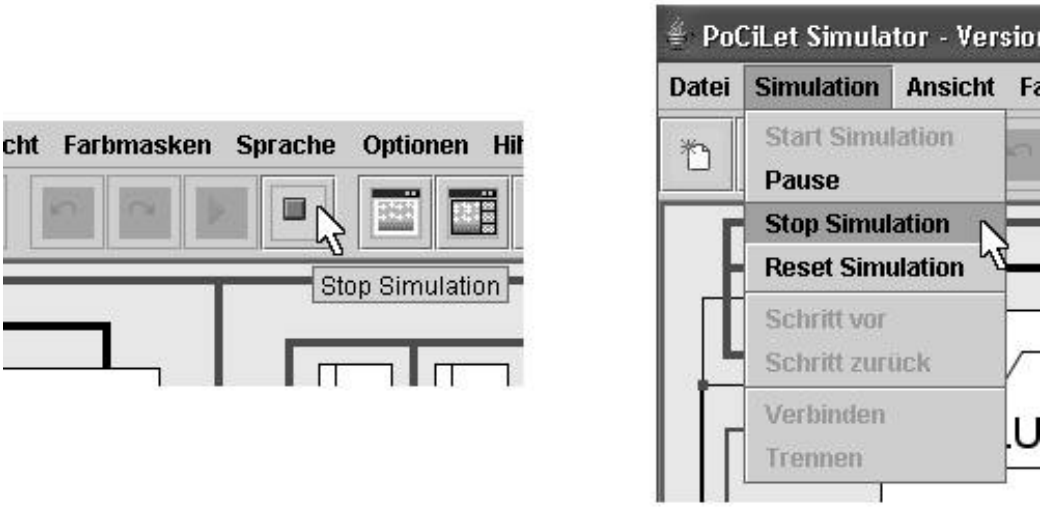


Abbildung 5.2: Stoppen des Simulators

Falls Sie den Simulator während der Abarbeitung eines Befehls stoppen wollen, um in einem bestimmten Zustand Details näher zu betrachten, drücken Sie den Button mit dem roten Quadrat oder wählen sie unter dem Menüpunkt Simulation den Eintrag Simulation stoppen (siehe Abbildung 5.2).

5.4.2 Teilschritte eines Befehls ansehen

Wenn Sie nicht möchten, dass der Simulator einen kompletten Befehl abarbeitet, da sie sich jeden Teilschritt eines Befehls in Ruhe ansehen möchten, klicken Sie auf den Button mit dem geschwungenen Pfeil nach rechts oder wählen Sie im Menü Simulation den Menüpunkt Schritt vor (siehe Abbildung 5.3). In diesem Fall wird nur der nächste Teilschritt des aktuellen Befehls abgearbeitet und der Simulator hält an.

5.4.3 Rückwärtsschritte im Simulator

Analog zu dem Vorgehen in Kapitel 5.3 besteht die Möglichkeit einen Teilschritt eines Befehls wieder zurückzugehen. Hierzu drücken Sie entweder den Button mit dem geschwungenen Pfeil nach links oder wählen im Menü Simulation den Menüpunkt Schritt zurück (siehe Abbildung 5.4). Wurden noch keine Schritte durchgeführt ist diese Funktion ausgegraut und kann nicht angewählt werden.

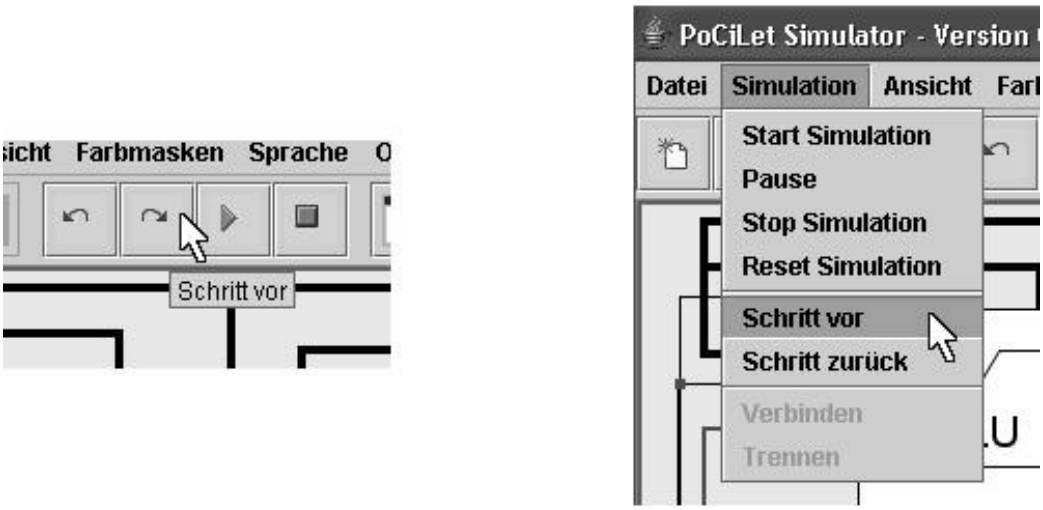


Abbildung 5.3: Einen Taktschritt vor gehen

5.4.4 Einen Zustand speichern und laden

Wenn sich der Simulator in einem Zustand befindet, kann dieser Zustand gespeichert werden. Hierzu wählen Sie im Menü Datei den Menüpunkt Zustand speichern (siehe Abbildung 5.5). Daraufhin öffnet sich ein Dialogfeld und Sie werden gebeten den Dateinamen und Ort auszuwählen, unter dem der Zustand gespeichert werden soll. Eine PoCiLet-Zustandsdatei hat die Endung .sta.

Wenn Sie eine zuvor gespeicherte PoCiLet-Zustandsdatei laden möchten, wählen Sie im Menü Datei den Menüpunkt Zustand laden (siehe Abbildung 5.6). Nun öffnet sich ein Dialogfenster und Sie können die Datei auswählen, die Sie laden möchten. Hierbei werden standardmässig nur Ordner und Dateien mit der Endung .sta angezeigt.

5.5 Die vier Ebenen von PoCiLet

Die vier Ebenen lassen sich entweder über den Menüpunkt Ansicht einzeln auswählen oder durch einen Klick auf die in Abbildung 5.7 dargestellen Buttons der Toolbar aktivieren. Die genaue Zuordnung der Buttons zu den Ebenen lässt sich über den entsprechenden Tooltipp leicht erfragen.

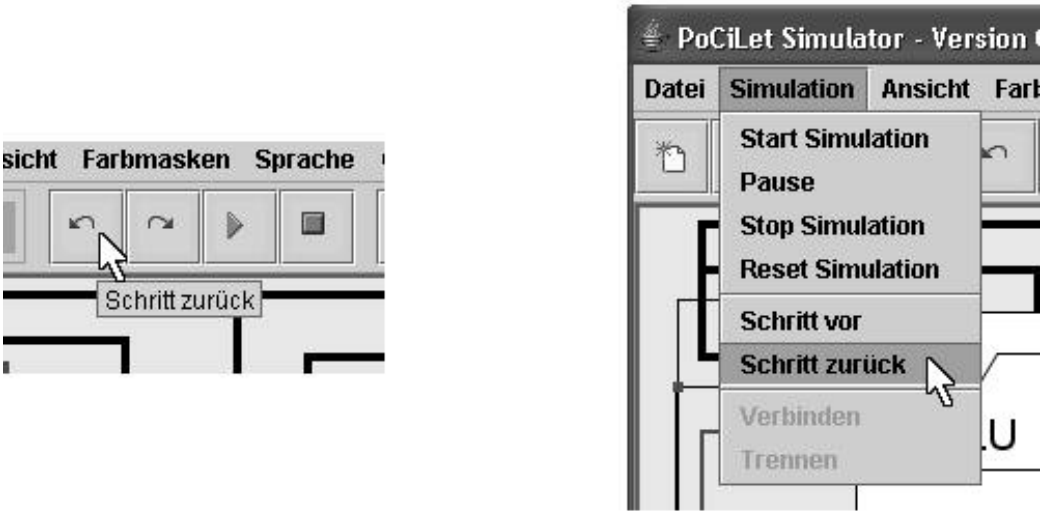


Abbildung 5.4: Einen Schritt zurück gehen

5.5.1 Die Schaltkreisebene

Die Schaltkreisebene zeigt detailiert den Schaltplan der PoCiLet-Hardware. Die Leitungen zwischen einzelnen Bauteilen sind als grüne Linien dargestellt, die bei Benutzung gelb eingefärbt werden. So kann der Benutzer während der Simulation verfolgen, auf welchen Teilen des Boards aktuell Funktionen ausgeführt und welche Komponenten dabei benutzt werden. Weitere interaktive Funktionalität bietet diese Ansicht nicht.

5.5.2 Die Register-Transfer-Ebene

Die Register-Transfer-Ebene ist die didaktische Hauptebene von PoCiLet. Hier sieht der Benutzer ausreichend abstrahiert die einzelnen Befehlsschritte und Befehlsunterschritte, die sowohl die PoCiLet-Hardware als auch der Simulator ausführen. Es ist jeder Zeit möglich einzelne Leitungen und Bauteile anzuklicken, um weitere Informationen über den aktuellen Zustand und innere Werte zu erhalten.
Hier kann der Benutzer den Ablauf der eingegebenen Programme verfolgen und auch Ergebniswerte ablesen und überprüfen. Auch Eingaben über die virtuelle Tastatur sind möglich, die sich analog zu der realen Tastatur auf der Hardware verhält.

Abbildung 5.5: Einen Zustand speichern

5.5.3 Die Layout-Ebene

Die Layout-Ebene zeigt detailiert den den Aufbau der PoCiLet Hardware, d.h. wie die Hardwareplatine wirklich aufgebaut ist. Es werden die Leitungen und Bauteile auf der Ober- und Rückseite der Platine zugleich angezeigt. Rot gezeichnete Leitungen befinden sich hierbei oben, blaue Leitungen sind rückseitig verlegt. Während des Ablaufs der Simulation werden auch hier aktive Leitungen orange bzw. türkis eingefärbt. Weitere interaktive Funktionalität bietet diese Ansicht nicht.

5.5.4 Die 3D-Ebene

Die 3D-Ebene zeigt eine virtuelle Ansicht des PoCiLet Boards. Diese Ebene soll hauptsächlich dazu dienen, Benutzern eine Vorstellung von der Hardware zu vermitteln, wenn die eigentliche Hardware nicht vorhanden ist. In dieser Ansicht ist es möglich, durch Benutzung von Maus und Tastatur den Blickwinkel, Blickpunkt und die Entfernung des Betrachters anzupassen. So kann der Benutzer jeden Winkel der Hardware detailiert erkunden.

5.6 Die drei Ansichtsmodi

PoCiLet bietet drei verschiedene Sichten an, bei denen sich die Aufteilung des Bildschirms strukturell unterscheidet. Folgende Bilder zeigen zum Einen die Möglichkeiten der Bedienung der Sichten innerhalb der Menüleiste und zum Anderen die Bedienung der Sichten in der Toolbar. Die Icons der Toolbar spiegeln dabei eine schematisierte Sicht der Einstellungen wieder, und zeichnen sich durch die

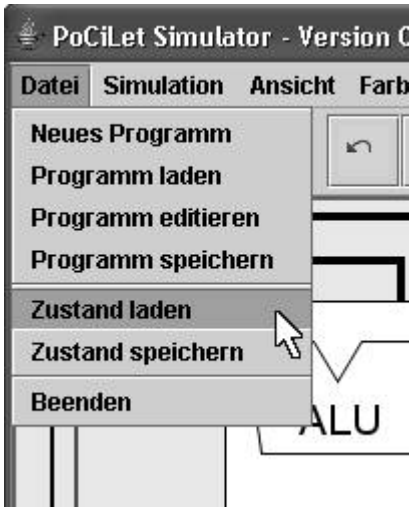


Abbildung 5.6: Einen Zustand laden

Abbildung 5.7: Auswählen der Hauptebene

intuitive Bedienung aus. Als weitere Hilfe besitzt die Toolbar sogenannte Tooltipps, welche aktiviert werden, wenn die Maus mehr als eine Sekunde über den Icons verweilt.

5.6.1 Vollbild

Wenn die Einstellung Vollbild gewählt wurde, ist jeweils nur eine der vier möglichen Ebenen von PoCiLet zu sehen. Der gesamte Bildschirmbereich steht so einer der vier Ebenen komplett zur Verfügung. Im Menü oder in der Toolbar ist die Ebene, die angezeigt werden soll, auswählbar.

5.6.2 Vierfach

Die Vierfachsicht bietet eine gleichzeitige Übersicht über alle vier Ebenen. Eine spezielle Umstellung der einzelnen Ebenen ist dabei nicht mehr möglich, da diese gleichberechtigt dargestellt werden. Eine Vergrößerung des sichtbaren Bildschirmbereichs ist durch das Abschalten des Footers möglich.

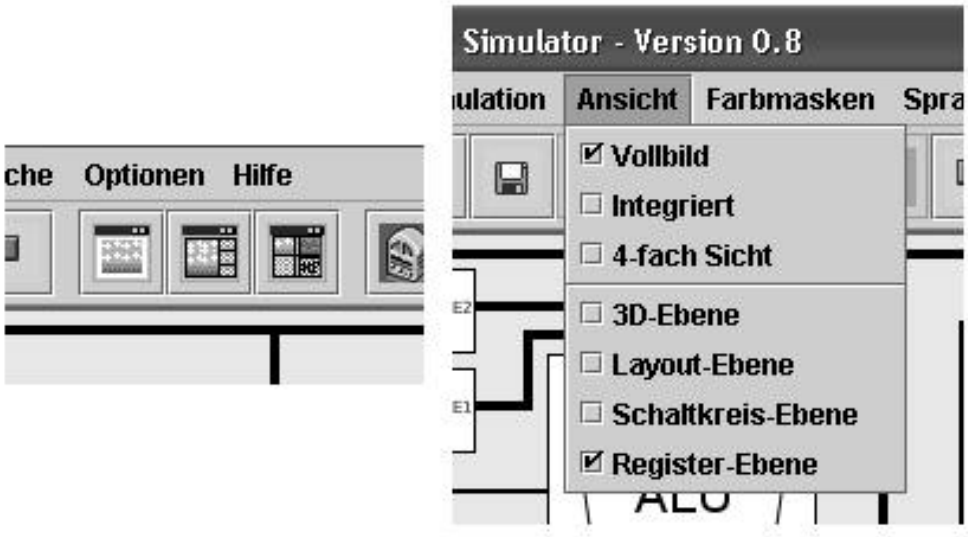


Abbildung 5.8: Auswählen des Ansichtsmodus

5.6.3 Integriert

Die integrierte Sicht stellt eine Kompromisslösung zwischen dem Vollbild und der Vierfachsicht dar, damit man für eine der vier Ebenen eine etwas größere Bildschirmfläche zur Verfügung hat und dabei die anderen drei Ebenen trotzdem noch in einer kleineren Ansicht verfolgen kann. Im Menü oder in der Toolbar ist die Ebene, welche angezeigt werden soll, umstellbar, dabei ist zu beachten, dass die gewählte Ebene in der größten Fläche der integrierten Sicht erscheint.

5.7 Die PoCiLet-Hardware

Das Lern-/Lehrobjekt PoCiLet bietet, neben der reinen Simulation eines Rechners auf Softwareebene, zusätzlich die Möglichkeit die PoCiLet-Hardware mit einzubeziehen.

5.7.1 Anschluss der Hardware

Voraussetzungen für die Verwendung der PoCiLet-Hardware sind

- Die PoCiLet-Hardware selbst
- Ein Windows oder Linux Rechner mit 15poliger serieller Schnittstelle

- Ein handelsübliches RS232-Kabel, passend zu der o.g. Schnittstelle (kein Nullmodemkabel)
- Installation der Java Unterstützung für die Serielle Schnittstelle (wie im read.me der PoCiLet-Installation beschrieben)

Zum Anschließen der Hardware sind keine besonderen Vorbereitungen nötig. Die Hardware kann jederzeit vor Inbetriebname oder während des Betriebs des Rechner oder der Software verbunden werden.

5.7.2 Anmeldung der Hardware

Nachdem die Hardware angeschlossen wurde, kann man über den Knopf Verbinden der Werkzeugleiste oder über den entsprechenden Menüpunkt unter Simulation der Software mitteilen, dass die Hardwarekomponenten vorhanden ist. Das aktuell geladene PoCiLet-Programm wird nun automatisch zur Hardware übertragen und diese in den Bereitschaftszustand versetzt.

VORSICHT: Um der Hardware einen Einstiegspunkt in die Simulation zu ermöglichen, ist es nötig, die Simulation in den Startzustand zu versetzen, d.h. es gehen alle bisher vorgenommenen Schritte verloren und das PoCiLet-Programm muss von vorne gestartet werden.

Nun haben sie die Möglichkeit, alle auf dem Monitor visualisierten Schritte auch auf der Hardware zu verfolgen und Eingaben über die Hardware vorzunehmen. Allerdings entfallen die Möglichkeiten, Rückschritte im laufenden Programm vorzunehmen und gespeicherte Zustände zu laden, da dies nicht von der Hardware unterstützt wird.

Um die Software wieder ohne Hardware zu betreiben, benutzen sie einfach den entsprechenden Werkzeugleistenknopf oder Menüpunkt, analog zum Verbinden der Hardware.

5.7.3 Benutzung der Hardware

Abbildung 5.9 zeigt die Anschlüsse und Bedienelemente des Boards. Diese sollen hier genauer erläutert werden.

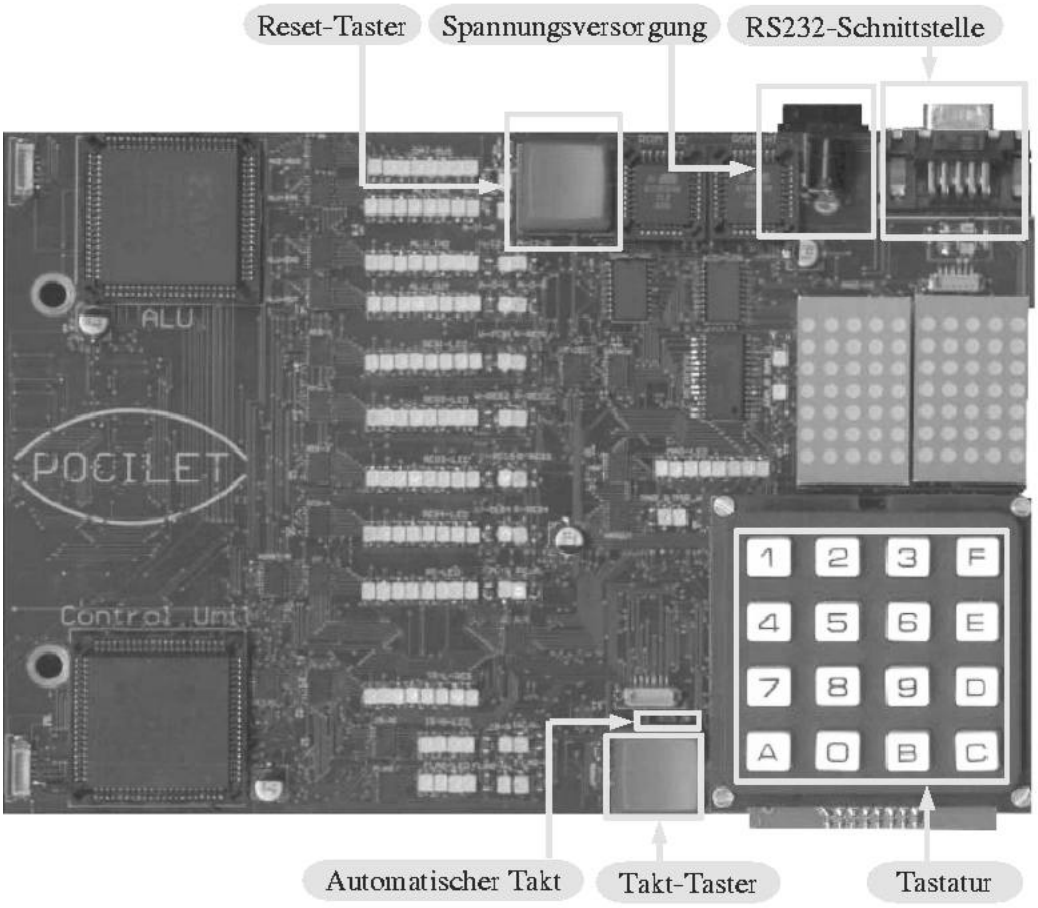


Abbildung 5.9: Anschlüsse und Bedienelemente des Boards

Spannungsversorgung

Der Betrieb des Boards erfordert eine Gleichspannung von 5V. Diese muss über einen Hohlstecker bereitgestellt werden. Dabei ist die Masseleitung innenliegend.

RS232-Schnittstelle

Diese Schnittstelle dient dazu, das Board mit dem Rechner zu verbinden, auf dem die PoCiLet-Software ausgeführt wird. Die Verbindung wird über ein serielles Kabel (kein Nullmodemkabel) hergestellt.

Reset-Taster

Mit diesem Taster kann der PoCiLet-Rechner in den Startzustand zurückversetzt werden. Die Programmausführung beginnt wieder bei Speicheradresse Null.

Takt-Taster

Durch das Drücken des Takt-Tasters führt der Rechner jeweils einen Taktschritt aus.

Automatischer Takt

Mit diesem Schalter kann eingestellt werden, ob der Takt über den manuellen Takttaster erzeugt werden soll (Schalter nach links schieben), oder aber ob die PoCiLet-Hardware den Systemtakt automatisch generieren soll (Schalter nach rechts schieben).

Tastatur

Mit der Tastatur können Daten für das PoCiLet-Programm eingegeben werden.

Sind Hard- und Software über den Menüpunkt Simulation/Verbinden in der Software miteinander gekoppelt, so werden Reset-, Takt- und Tastaturbefehle von der Hardware in den Simulator der Software übernommen.

5.8 Die PoCiLet Simulator GUI beenden

Wählen Sie im Menü Datei den Befehl Beenden aus.

Kapitel 6

Anhang

6.1 Zustandsautomat des Steuerwerks Seite 31

Befehl	Machinenwort	Schritt	Ausführung	Zustandscodierung
ldi	00rrffffffff	1. 2.	R:=IRconst PC:=PC+1	ldi_reg PC_akt
call	0100kkkkkkkk	1. 2.	Stack(x):=PC PC:= IRconst	call_Stackzuweisung call_PCakt
return	0101kkkkkkkk	1.	PC := Stack	return_PCakt
goto	0110kkkkkkkk	1.	PC := Irconst	goto_PCakt
not	011100ffr1r1r2r2	1. 2. 3.	ALUreg1 := r1r1 ALUerg := not reg1 PC:=PC+1	ALUreg1_zu ALUerg PC_akt
move	011101ffr1r1r2r2	1. 2.	r2r2 := r1r1 PC:=PC+1	move_reg_zu PC_akt
store	011110ffr1r1r2r2	1. 2. 3.	MAR := r1r1 RAM := r2r2 PC:=PC+1	store_mar_zu store_RAM_zu PC_akt
load	011111ffr1r1r2r2	1. 2. 3.	MAR := r1r1 r2r2 := RAM PC:=PC+1	load_mar_zu load_reg_zu PC_akt
equal	1000ffffr1r1r2r2	1. 2. 3. 4.	ALUreg1 := r1r1 ALUreg2 := r2r2 ALUerg := equal() PC:=PC+1	ALUreg1_zu ALUreg2_zu equal_ALUerg PC_akt

Befehl	Machinenwort	Schritt	Ausführung	Zustandscodierung
ne	1001fffr1r1r2r2	1. 2. 3. 4.	ALUreg1 := r1r1 ALUreg2 := r2r2 ALUerg := notequal() PC:=PC+1	ALUreg1_zu ALUreg2_zu notequal_ALUerg PC_akt
and	101000r1r1r2r2r3r3	1. 2. 3. 4. 5.	ALUreg1 := r1r1 ALUreg2 := r2r2 ALUerg := and() r3r3 := ALUerg PC:=PC+1	ALUreg1_zu ALUreg2_zu ALUerg REG_ZU PC_akt
or	101001r1r1r2r2r3r3	1. 2. 3. 4. 5.	ALUreg1 := r1r1 ALUreg2 := r2r2 ALUerg := or() r3r3 := ALUerg PC:=PC+1	ALUreg1_zu ALUreg2_zu ALUerg REG_ZU PC_akt
xor	101010r1r1r2r2r3r3	1. 2. 3. 4. 5.	ALUreg1 := r1r1 ALUreg2 := r2r2 AlUerg := or() r3r3 := ALUerg PC:=PC+1	ALUreg1_zu ALUreg2_zu ALUerg REG_ZU PC_akt
callint	10101100xxxx	1. 2. 3.	Stack(x):=PC Stack(x).=Flags PC:=ff	Inter Flagsretten InterPCakt
rfi	10101101xxxx	1. 2. 3.	Flags:=Stack(x) PC:=Stack(x) PC:=PC+1	returninter Pczurueck Pcakt
noop	10101110xxxx	1.	-	noop
add	101100r1r1r2r2r3r3	1. 2. 3. 4. 5.	ALUreg1 := r1r1 ALUreg2 := r2r2 ALUerg := add() r3r3 := ALUerg PC:=PC+1	ALUreg1_zu ALUreg2_zu ALUerg REG_ZU PC_akt
sub	101101r1r1r2r2r3r3	1. 2. 3. 4. 5.	ALUreg1 := r1r1 ALUreg2 := r2r2 ALUerg := sub() r3r3 := ALUerg PC:=PC+1	ALUreg1_zu ALUreg2_zu ALUerg REG_ZU PC_akt
sar	101110ffr1r1r2r2	1. 2. 3. 4. 5.	ALUreg1 := r1r1 ALUreg2 := r2r2 ALUerg := sar() r1r1 := ALUerg PC:=PC+1	ALUreg1_zu ALUreg2_zu ALUerg reg_zu PC_akt

Befehl	Machinenwort	Schritt	Ausführung	Zustandscodierung
sal	101111ffr1r1r2r2	1.	ALUreg1 := r1r1	ALUreg1_zu
		2.	ALUreg2 := r2r2	ALUreg2_zu
		3.	ALUerg := sal()	ALUerg
		4.	r1r1 := ALUerg	sal_reg_zu
		5.	PC:=PC+1	PC_akt
ln	1100ffffr1r1r2r2	1.	ALUreg1 := r1r1	ALUreg1_zu
		2.	ALUreg2 := r2r2	ALUreg2_zu
		3.	ALUerg := lessnat()	ALUerg
		4.	PC:=PC+1	PC_akt
len	1101ffffr1r1r2r2	1.	ALUreg1 := r1r1	ALUreg1_zu
		2.	ALUreg2 := r2r2	ALUreg2_zu
		3.	ALUerg := lessequalnat()	ALUerg
		5.	PC:=PC+1	PC_akt
l2	1110ffr1r1r2r2	1.	ALUreg1 := r1r1	ALUreg1_zu
		2.	ALUreg2 := r2r2	ALUreg2_zu
		3.	ALUerg := less2er()	ALUerg
		4.	PC:=PC+1	PC_akt
le2	1111ffffr1r1r2r2	1.	ALUreg1 := r1r1	ALUreg1_zu
		2.	ALUreg2 := r2r2	ALUreg2_zu
		3.	ALUerg := lessequal2er()	ALUerg
		4.	PC:=PC+1	PC_akt

Tabelle 6.1: Zustandstabelle des Steuerwerks

Index

Adressierung, 11
ALU, 32
Auskodierung, 10, 12
automatisierte Tests, 99

Baustein, 6, 19
    Logikbausteinen, 32
    Registerbausteine, 19
    SMD-Bausteine, 18
    Treiberbaustein, 22
Befehlssatz, 10
    Befehle, 10
        Datencode, 12
        Operationscode, 11
        Sprungbefehle, 12
    Befehlstypen, 10
Blickpunkt, 114
Blickwinkel, 114
Board, 36
    Routen, 36
Bustreiber, 22

CommHandler, 68
CPLDs, 20, 32
    JTAG-Schnittstelle, 27
    TDI, 27
    TDO, 27
    XC95108, 27
    Xilinx, 20
        Modell 9572-15, 20

Datenbus, 46
DRAM, 21
DRC, 38

EEPROM, *siehe* ROM
Entwicklungsstufen, 3
    Layoutebene, 3
    Registertransferebene, 3
    Schaltkreisebene, 3
    Sichten, 6
EventList, 47

Flipflop, 44
Footer, 115
Footer, 67
Funktionentests, 99

Gehäuse, 19
    PLCC84, 20
    TSSOP, 19
Gleichstromnetzteil, 18

Halbleiterbauelemente
    62256-70M, 21
    74245, 22
    74573, 20
    Chip, 20
    MAX233, 21
    UDN2585, 22
Hardware, 7
    Hardwareplatine, 114
Harvard-Architektur, 15
high, 50
highimp, 50

ICs, 18
id, 53
Initialisierungsschritt, 110
Integrationstests, 100

Internationalisierbarkeit, 62
ISO-Norm 9241, 58

J2RE, 108
Java3D, 108
JUnit, 101

Klassendiagramm, 44
Klassentests, 99
Komponent, 6
    Komponentenmodell, 7
Kondensatoren, 18

LABVIEW, 42
Layer, 67
LEDs, 19
    Kathoden, 20
Leitung, 54
    Vertice, 51
Lernphasen, 59
Lernsystem, 6
    Lehr-/Lernobjekt, 3
lh, 50
Logik, 50
Logikbausteine, *siehe* Baustein
low, 50

Mainframe, 66
Masse, 20
MAX233, 43
Menüleiste, 109
Mikroprozessor, 10
Modell, 7
Modelsim, 32
Modultests, 99
Multiplexing, 22

NAXJP, 27

Offset, 10
OpenGL/Win32, 108

Packages, 44
Panel, 67
PanelHandler, 67
Parser, 109
PC, 46
PC-Register, 46
Pins, 18
    /OC, 19
    Ausgangspins, 19
    C, 19
    I/O-Pins, 20
Platine, 15
Portabilität, 7
Ports, 50
Program, 55
Programm-Adressbus, 46
Programmspeicher, 21
Pulldown-Widerstand, 21

Rückkopplung, 45
RAM, 32
Rechnersystem, 3
    Pocketcomputer, 10
Regeressionstests, 100
Register, 10
    Flagregister, 27
ROM
    28C64, 21
RS232-Kabel, 117

Schaltplan, 15
Schaltung, 20
Scheduler, 51, 55
Schmelzsicherung, 18
Schnittstellen, 32
    Benutzerschnittstelle, 58
Signale, 4
    Interruptsignal, 29
    Resetsignal, 29
    RS232-Pegel, 21
    Signalpegel, 21
    Steuersignale, 19

Taktsignal, 34
Signalen, 50
Simba, 6
SimObject, 50
Simulation, 3
Hardwarssimulation, 34
Simulationsschritt, 47
Simulator, 64
simulation, 51
Simulationskern, 7
Simulator, 51
Spannung, 42
Schwellwertspannungen, 22
Versorgungsspannung, 18
Sprungbefehle, 10
Sprungadressen, 10
SRAM, 21
State, 55
Step, 51
StepList, 47
Steuerwerk, 32
System, 3
Rechnersystem, 3
Ein-/Ausgabeeinheit, 6
Rechenwerk, 6
Speicher, 6
Systemtests, 100

Taktschritte, 113
Taktunterschritte, 113
Tastatur, 21
reale Tastatur, 113
virtuelle Tastatur, 113
Teilschritt, 111
Testbench, 34
Testen
hierarchisches, 99
inkrementelles, 99
Testkonzept, 99
Teststrategie, 99
Testsuite, 101
Toolbar, 71, 109
Tooltipps, 115
Treiberbaustein, *siehe* Baustein
Trigger, 22
Schmitt-Trigger, 22

UDN2585A, *siehe* Halbleiterbauelemente
UML-Diagramm, 44

Vektor
Eingangsvektoren, 27
Ergebnisvektor, 27
Flagvektor, 27
Verzögerungen, 45
VHDL, 32
virtuelle Ansicht, 114
Visualisierung, 3, 8
3D Modell, 6
3D-Einstiegsszene, 4
3D-Visualisierung, 3
Von-Neumann-Architektur, 15

Waveform, 34
Werkzeugleiste, 117
Whitebox-Tests, 99
Wissensvermittlung, 6

Zieladresse, 12

Literaturverzeichnis

[Bal98] BALZERT, HELMUT: *Lehrbuch der Softwaretechnik, Band 1 und 2*. Spektrum-Verlag, 1998.

[fNe85] NORMUNG E.V., DEUTSCHES INSTITUT FÜR: *Bildschirmarbeitsplätze, Teil 7, Ergonomische Gestaltung des Arbeitsraumes. Beleuchtung und Anordnung*, 1985.

[fS] STANDARDIZATION, INTERNATIONAL ORGANIZATION FOR: *Ergonomic requirements for office work with visual display terminals (VDTs), Part 10: Dialogue principles*. Available on: `http://www.iso.org`.

[iX101] *iX Magazin*, 11, 2001.

[Jan03] JANSEN, THOMAS: *Grundvorlesung Rechnerstrukturen - Erster Teil*. Available on: `http://ls12-www.cs.uni-dortmund.de/edu/scripts-de.html`, 2003.

[Lin02] *Unit-Testing in Java*. Linux Magazin, 03, 2002.

[Rie97] RIEDEMANN, EIKE HAGEN: *Testmethodenfür sequentielle und nebeläufige Software-Systeme, Band 1*. Teubner-Verlag, 1997.

[VC97] VOLKER CLAUS, ANDREAS SCHWILL, HERMANN ENGESSER: *Informatik Duden*. Dudenverlag Mannheim.Leipzig.Wien.Zürich, 1997.