Universität Hamburg
DER FORSCHUNG | DER LEHRE | DER BILDUNG

Universität Hamburg
Fakultät für Mathematik, Informatik und Naturwissenschaften

# Abschlussbericht

---

64-189 Projekt
Entwurf, Realisierung und Programmierung eines Mikrorechners
Wintersemester 13/14

---


Verfasser:

Marcel Hellwig
Felix Ortmann
David Weber
Felix Wiedemann
Nasif Yueksel

Betreuer:

Dr. Andreas Mäder
Bernd Schütz



Vorgelegt am: 23. März 2014

# Inhaltsverzeichnis

# 1 Einleitung

Ein Computer ist keine komplizierte Maschine. Die Meisten wissen, dass er aus einem Prozessor, einem Arbeitsspeicher und einer Festplatte besteht. Der Prozessor steuert die Maschine und treibt die Datenverarbeitung voran, der Arbeitsspeicher dient als Puffer für die Aufträge und die Festplatte ist für die persistente Speicherung der Daten zuständig. Weiterhin können noch Hardwareteile wie eine Grafikkarte, ein DVD-Laufwerk oder eine Webcam verbaut sein. Beschäftigt man sich etwas genauer mit dem Thema, wird man schnell feststellen, dass nicht all diese Komponenten zwingend notwendig sind. Dass die Webcam oder das DVD-Laufwerk nicht unbedingt zu jedem PC gehören erkennen wohl die Meisten. Dass auch die Grafikkarte und die Festplatte nicht zur Grundidee eines Computers gehören ist nicht so offensichtlich.

Der österreichisch-ungarische Mathematiker John von Neumann publizierte im Jahr 1945 eine Architektur, die die Basis moderner PCs, Laptops, Smartphones darstellt. Die nach ihm benannte Architektur besteht aus einer Recheneinheit, der Central Processing Unit (CPU), einem Speicher für die Daten und Befehle, einer Ein- und Ausgabe-Einheit, damit der Mensch mit der Maschine interagieren kann und einem Bussystem, das die genannten Komponenten miteinander verbindet. Die CPU steuert dabei alle anderen Komponenten. Aber woher weiß die CPU, was zu tun ist? Denn der Prozessor ist auch nur ein Hardwareteil, das ohne Anweisungen nicht weiß, was zu tun ist. Ein Algorithmus muss die Regeln vorgeben, wie Befehle auszuführen sind. Dieser Algorithmus bestimmt, wie schnell und effizient Befehle ausgeführt werden. Der Prozessor kann noch so modern sein, wenn der Algorithmus schlecht ist, wird auch der Computer keine Wunder vollbringen. Nicht umsonst verdienen Unternehmen wie Intel, AMD und neuerdings auch Qualcomm Milliarden mit dem Verkauf von Prozessoren.

Wie wichtig der Algorithmus ist, sieht man auch gut am Beispiel von Apple. Das Top-Smartphone von Apple läuft mit einem 1,3 GHz-Dual-Core-Prozessor. Das Top-Modell von Samsung läuft mit einem 2,5 GHz-Quad-Core-Prozessor. Dennoch laufen beide Smartphones mit einem ähnlichen Arbeitstempo und erreichen ähnliche Werte in Benchmarks. Natürlich kann man die 2 Prozessoren nicht direkt miteinander vergleichen, da es sich um andere Architekturen und Hersteller handelt. Außerdem läuft nicht das gleiche System auf beiden Geräten. Dennoch ist der Unterschied der Taktfrequenz deutlich.

Mit diesem „magischen“ Algorithmus durften und mussten wir uns in dem Projekt „Entwurf, Realisierung und Programmierung eines Mikrorechners“ beschäftigen.

## 1.1 Das Team

Unser Team besteht aus verschiedensten technikbegeisterten Mitgliedern. Zum einen aus Nasif Yüksel, Student der Wirtschaftsinformatik im fünften Semester. Er ist der strahlende Stern des Testens in unserem Projekt. Marcel Hellwig, Student der Informatik im fünften Semester. Der unbestrittene König der Hardware. David Weber, Student der Wirtschaftsinformatik. Willenloser Sklave in der Softwarefraktion. Felix Wiedemann, Student der Informatik im fünften Semester. Er ist der Anführer der Softwarerebellen. Felix Ortmann, Student der Informatik im sechsten Semester. Er war der Verfechter der Konventionen.

Formulierung sehr fluffig aber schon iwie witzig ;)

## 1.2 Motivation

Zum einen waren wir einfach neugierig zu erfahren, wie die Software mit der Hardware im Detail zusammenspielt. In der Veranstaltung Rechnerstrukturen haben wir ersten Kontakt mit diesem Thema bekommen. Um unsere großes Interesse weiter zu vertiefen und in die Praxis umzusetzen, hat sich dieses Projekt ideal angeboten. Ein weiterer Grund für dieses Projekt war, dass wir unbedingt unser Projekt im Arbeitsbereich TAMS machen wollten. Denn bereits im Seminar wurden wir durch interessante Themen und einer tollen Unterstützung durch das Lehrpersonal überrascht. Außerdem hat die Vorstellung einen eigenen Mikrorechner zu konzipieren, jeden von uns gereizt.

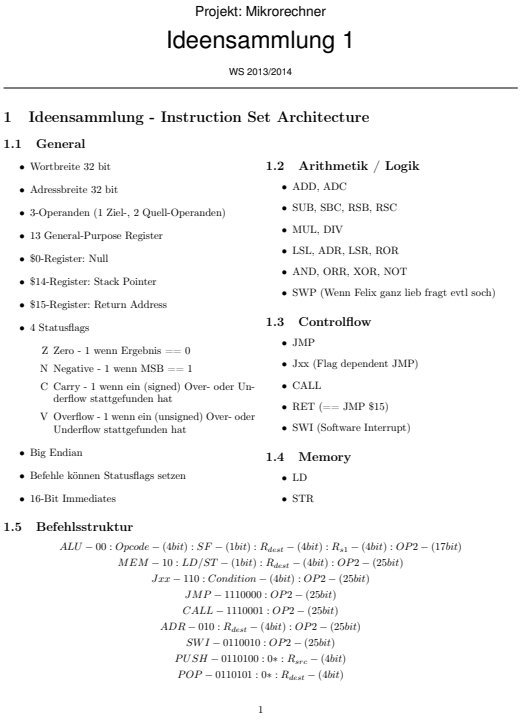
## 1.3 Material

Um unsere Testergebnisse sichtbar zu machen, wurde ein FPGA-Prototyp, u.a. ausgestattet mit 4 LEDs, 4 Tastern, einem 800x400-Touchscreen und diversen Schnittstellen. Unser Ziel war es die LEDs durch drücken der Taster zum leuchten zu bringen.

Ob wir unser Ziel erreicht haben, was besonders gut oder schlecht lief, auf was für Schwierigkeiten wir gestoßen sind und welche Herausforderungen wir gemeistert haben,wird auf den nächsten Seiten detailliert darstellen.

# 2 Allgemein

Am 17. Oktober 2013 war der Startschuss der Veranstaltung. Die ersten Termine waren im Sinne der Wissensreaktivierung gestaltet. In mehreren Terminen wurden die grundlegenden Konzepte von Rechnerarchitekturen wiederholt, um Wissenslücken zu schließen und alle auf einen Stand zu bringen. Erst später ging die eigentliche Projektphase los. Nach den Vorlesungsterminen ging es in die Gruppen. Jede Gruppe bestand aus 5-8 Personen. Unsere Gruppe bestand aus 5 Leuten und war damit die Kleinste. Ab diesem Zeitpunkt waren die Gruppen auf sich gestellt. Die Lehrkräfte waren nicht mehr für die Organisation zuständig, sondern hatten nur noch unterstützende und beratende Aufgaben. In den ersten Wochen wurde noch nicht so viel Praktisches gemacht. Wir teilten uns zunächst in eine Hardware- und Softwaregruppe auf. Damit beide Teilgruppen an einem Strang ziehen, investierten wir die Zeit in den ersten Wochen für eine kleine Ideensammlung bezüglich unserer ISA und unserer Befehlsstruktur. Es wurden Fragen grundlegende Fragen geklärt wie z.B. die Wort- und Adressbreite, Little oder Big Endian und wie unsere Befehlsstruktur aufgebaut ist. Ein Teil unserer Ideensammlung der ersten Wochen ist in der Abbildung Abbildung 2.1 zu sehen.



Projekt: Mikrorechner

Ideensammlung 1

WS 2013/2014

**1 Ideensammlung - Instruction Set Architecture**

**1.1 General**

- Wortbreite 32 bit
- Adressbreite 32 bit
- 3-Operanden (1 Ziel-, 2 Quell-Operanden)
- 13 General-Purpose Register
- $0-Register: Null
- $14-Register: Stack Pointer
- $15-Register: Return Address
- 4 Statusflags
  - Z Zero - 1 wenn Ergebnis == 0
  - N Negative - 1 wenn MSB == 1
  - C Carry - 1 wenn ein (signed) Over- oder Underflow stattgefunden hat
  - V Overflow - 1 wenn ein (unsigned) Over- oder Underflow stattgefunden hat
- Big Endian
- Befehle können Statusflags setzen
- 16-Bit Immediates

**1.2 Arithmetik / Logik**

- ADD, ADC
- SUB, SBC, RSB, RSC
- MUL, DIV
- LSL, ADR, LSR, ROR
- AND, ORR, XOR, NOT
- SWP (Wenn Felix ganz lieb fragt evtl soch)

**1.3 Controlflow**

- JMP
- Jxx (Flag dependent JMP)
- CALL
- RET (== JMP $15)
- SWI (Software Interrupt)

**1.4 Memory**

- LD
- STR

**1.5 Befehlsstruktur**

$ALU - 00 : Opcode - (4bit) : SF - (1bit) : R_{dest} - (4bit) : R_{s1} - (4bit) : OP2 - (17bit)$

$MEM - 10 : LD/ST - (1bit) : R_{dest} - (4bit) : OP2 - (25bit)$

$Jxx - 110 : Condition - (4bit) : OP2 - (25bit)$

$JMP - 1110000 : OP2 - (25bit)$

$CALL - 1110001 : OP2 - (25bit)$

$ADR - 010 : R_{dest} - (4bit) : OP2 - (25bit)$

$SWI - 0110010 : OP2 - (25bit)$

$PUSH - 0110100 : 0* : R_{src} - (4bit)$

$POP - 0110101 : 0* : R_{dest} - (4bit)$

1

Abbildung 2.1: Ideensammlung

Um eine gute Kommunikation zu gewährleisten, trafen wir uns jeden Donnerstag am Informatikum. Wir arbeiteten alle in einem Raum, weil uns eine direkte Kommunikation wichtig war. Bei Problemen konnte so der direkte Kontakt hergestellt werden. Denn es stellte sich schnell raus, dass Abweichungen von der ersten Ideensammlung unumgänglich waren. So konnten Konflikte schnell diskutiert und beseitigt werden.

Darüber hinaus nutzten wir Git, um die Ergebnisse zusammenzutragen und alle auf einem Stand zu halten. Außerdem konnten so Fragen sofort geklärt werden und mussten nicht bis auf den nächsten Donnerstag verschoben werden.

## 2.1 vorgehensweise inkl organisation

## 2.2 Warum Python

In unserem Pojekt haben wir an vielfältigen Stellen auf die Sprache Python zurückgegriffen. Python wurde sowohl zu Modellierungsunterstüzung als auch zum Bau von Emulatoren und UIs verwendet. Diese waren zwar nicht zwingend relevant für das Abschließen des Projektes, erleichterten unsere

Arbeit allerdings sehr.
Die Entscheidung für die Sprache Python wurde in unserer Gruppe sehr früh getroffen. Schon beim ersten Treffen waren alle Gruppenmitgleider ausreichend über VHDL Vor- und vorallem Nachteile informiert, um eine Wrapper-Sprache ausprobieren zu wollen. Bereits beim ersten Treffen haben wir beschlossen, dass wir Python nicht nur nutzen wollen, weil die Sprache intuitiv ist, sondern weil wir ein Python Package fanden, dass den gesamten Hardware-Modellierungsprozess von VHDL auf Python Konstrukte abstrahieren konnte - MyHDL. Mit diesem Package konnte die Modellierungsarbeit und später auch die Verifikation erheblich erleichtert werden. MyHDL bildet eine Art Brücke zwischen den Eigenheiten der mit VHDL möglichen Prozessbeschreibung hin zu dem bekannten Pythonkonzept der Generatoren. Mit Pythons Unit Test Framework war es viel leichter den MyHDL Code zu testen, als tatsächliche Hardwaretests durchzuführen. Zumindest in den frühen Stadien des Projektes hat es einen großen Teil der Arbeit sehr erleichtert.
Abgesehen von dem Wrapper-Package MyHDL hat die Sprache Python noch an anderen Stellen viel Einfluss auf unser Projekt gehabt. Als erstes war ein Compiler für unseren eigenen Assembler wichtig. Wir wollten unsere Assembler Befehle in Bitcode übersetzen können. Dieser Assembler wurde in Python geschrieben und übersetzt alle Assemblerbefehlsworte in den Bitcode, wie er in userer zu dem Zeitpunkt jeweils aktuellen ISA spezifiziert war. Unsere endgültige ISA ist in Tabelle 6.1 zu finden. Mit dem Bitcode war allerdings noch nicht viel anzufangen, da die Hardwarespezifikation noch lange nicht fertig war. Folglich brauchten wir eine Möglichkeit, unsere Assemblerprogramme auf Gültigkeit zu testen, am besten sogar eine Ausführungsmöglichkeit. Es folgte ein weiteres kleines Unterprojekt in Python.
Wir haben uns zum Ziel gesetzt, einen Emulator zu schreiben, der den später modellierten Microcontroller vollständig widerspiegeln sollte. Damit wollten wir es uns ermöglichen, bereits kleine Programme in unserem eigenem Assembler-Code schreiben und ausprobieren zu können. Den Emulator haben wir ebenfalls in Python geschrieben, er emuliert analog zur HDL Hardwarestruktur eine CPU, eine RAM/ROM und einen Programmcode in der ROM. Auf dieser Ausführungseinheit fehlte nur noch die Introspektion, die wir brauchten, um den Ablauf der Programme zu verifizieren und auch Datenflüsse in der emulierten Hardware zu optimieren. Da wir den Emulator eng an die tatsächliche Hardwarespezifikation angelehnt haben, war es uns möglich das spätere Verhalten des Microcontrollers am Beispiel des Emulators zu beobachten und zu verbessern. Um also den nötigen Grad an Einsicht in unsere Programminterna zu erhalten, schrieben wir unsere eigene „Emulator UI", eine Art Debugger für Programm- und Hardwarezustände. Für das grafische Frontend des Emulators verwendeten wir das UI-Framework GTK für Python.

## 2.3 Ein Wort zur Lizenz

*GPLv3, GNU General Public License version 3*
Von Beginn an haben wir unser Projekt weltöffentlich und zugänglich auf github gemacht. Im Zuge dieser Art der Veröffentlichung erschien es uns nötig, den Code unter Lizenz zu stellen. Es gibt zu genau diesem Zweck viele verschiedene Arten von Lizenzen; viele haben den Zweck den tatsächlichen Autor des Codes davor zu schützen, dass Dritte diese Arbeit verkaufen und den Autor ausbeuten. Dabei darf jedoch nicht außer acht gelassen werden, dass der Code open source ist und jedem zugänglich sein *soll*.
Wir haben uns in unserem Projekt für die Lizenz *GNU General Public License version 3* entschieden. Damit sind vorallem die Nutzer unserer Software nicht benachteiligt, die Nutzung ist uneingeschränkt möglich und erlaubt. Der Download, das Verteilen und auch die Veränderung zu eigenen Zwecken werden in der Lizenz behandelt und explizit erlaubt. Was jedoch nicht erlaubt ist, ist die kommerzielle Distribution unseres Codes ohne uns als Autoren zu informieren und zu beteiligen. Es lässt sich festhalten, dass die Lizenzsierung hauptsächlich aus dem Wunsch heraus motiviert wurde, unseren

Code nicht an einen unbekannten Dritten rechtlich zu verlieren.

## 2.4 tools

### 2.4.1 python

### 2.4.2 git

### 2.4.3 pycharm

### 2.4.4 latex

### 2.4.5 altera

### 2.4.6 hades

### 2.4.7 gtkwave

## 2.5 testen

# 3 Hardware(Marcel)

## 3.1 Modelliersprache

Zur Beschreibung wurde, entgegen der in dem Projekt genannten Sprache, MyHDL[1] genutzt. Es wurde uns von einem Kommilitonen nahe gelegt *nicht* VHDL zu nutzen, da dies einen Großteil der Zeit in Anspruch nimmt die Sprache zu lernen. Diese Zeit würde später fehlen, wenn man den Mikrocontroller implementieren möchte MyHDL ist ein Python-Modul, welches dazu genutzt wird Python als Hardwarebeschreibungssprache zu nutzen. Außerdem können damit auch Tests geschrieben werden, die zur Verifikation der Devices dient. Ein Beispiel ist zu finden unter Abbildung 3.1.

```
1  from myhdl import *
2
3  def dff(q, d, clk):
4
5      @always(clk.posedge)
6      def logic():
7          q.next = d
8
9      return logic
```

Abbildung 3.1: D-FlipFlop

In Zeile drei werden sämtliche Ports der Komponente deklariert. Dabei werden – im Gegensatz zu VHDL – keine Spezifikationen wie input/output oder Bitlänge definiert. Diese werden laut Konvention nur in dem Kommentar der "Funktion" definiert.
Der Dekorator[2] in Zeile 5 wird benutzt um zu spezifizieren, wann die dekorierte Funktion ausgeführt werden soll. In diesem Falle wird gesagt, dass bei einer steigenden Flanke des clk-Ports die Funktion ausgeführt werden soll.
Die eigentliche Logik der Komponente befindet sich in Zeile 7. Hier wird gesagt, dass im nächsten Augenblick der Wert von $q$ $d$ sein soll. In Zeile 9 wird die dekorierte Funktion zurückgegeben, d.h. sie wird in der Simulation und Transformation berücksichtigt.
Gut zu sehen sind an diesem Beispiel die Vor- und Nachteile von Python als Hardwarebeschreibungssprache. Ein Nachteil ist, dass man die genaue Spezifikation eines Ports nicht direkt angeben kann, z.B. deren Bitweite oder Typ. Dies kann zu Fehlern führen und zu viel Frust. MyHDL entscheidet die Eigenschaften durch statische Analyse des Quellcodes.
Vorteile hingegen sind der große Umfang der Python-Bibliotheken, die man sehr gut zum Testen der Komponenten nutzen kann, z.B. reguläre Ausdrücke, Listen oder Hashmap, sowie die einfache Erlernbarkeit und viele Syntaktische Feinheiten.

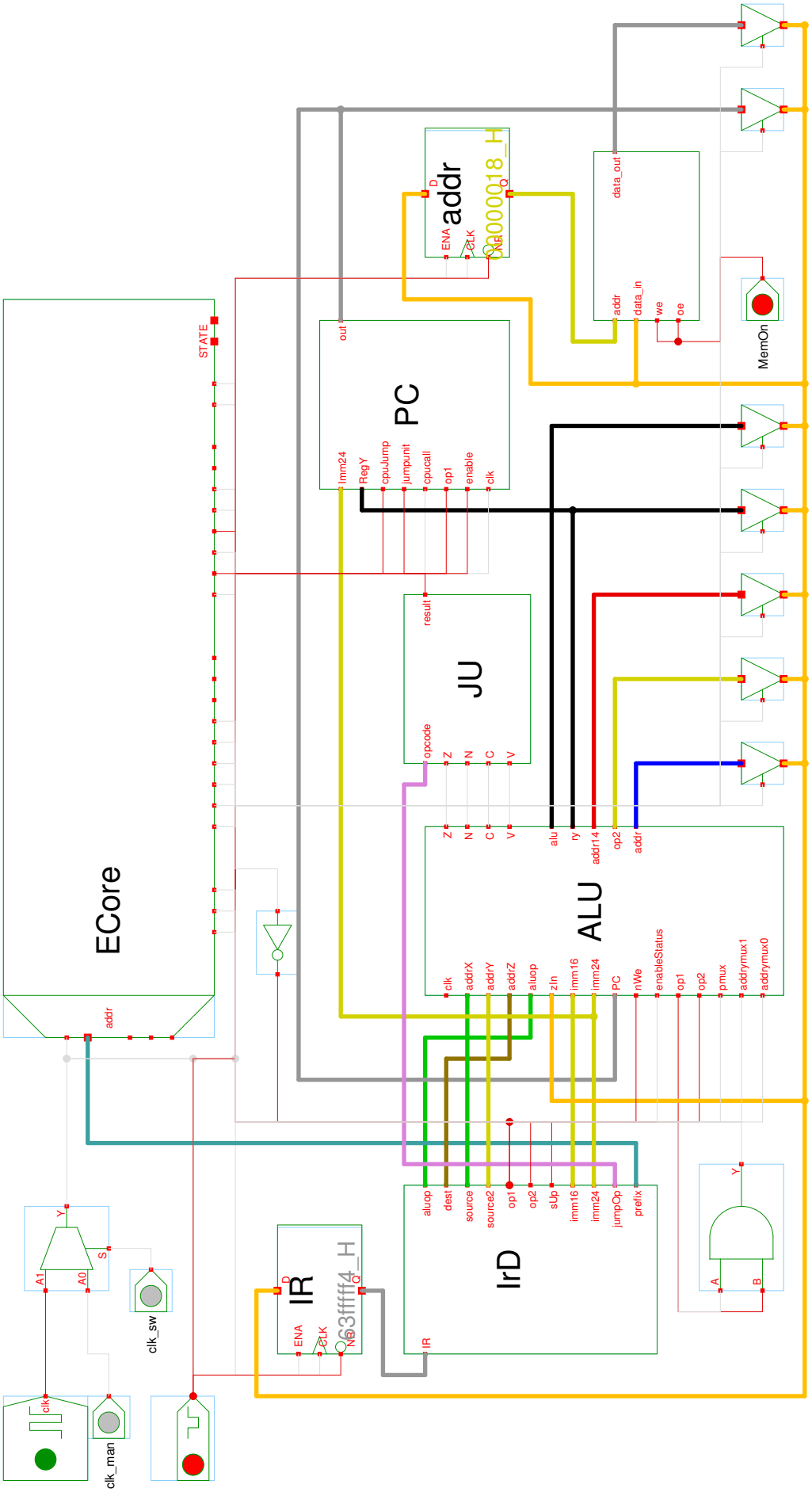


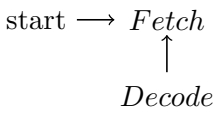

Abbildung 3.2: Übersicht des MK

## 3.2 Design

Das Design der Mikrocontrollers ist sehr stark am D-Core[3] orientiert, mit einem zentralen Bus, einem Steuerwerk, einer Abstraktion für den Speicher, sowie Elementen, die mit Hilfe des Busses miteinander kommunizieren. Wie man in der Abbildung 3.2 sehr gut sehen kann ist die Verbindung zwischen den einzelnen Komponenten auf ein Minimum gehalten. Einzige Ausnahme bildet hierbei der Instruction-Decoder. Auf die einzelnen Komponenten wird in den Folgenden Abschnitten noch genauer eingegangen.
Die Entscheidung zu dieser Anordnung kam aus der Überlegung heraus, dass wir uns an etwas bestehendem orientieren wollten. Deswegen ist verfügt er über einen Bus und verschiedene TriState-Dioden, die von einer Zentralen Einheit angesteuert werden, in unserem Fall das Steuerwerk oder CPU genannt. Doch genau hier liegt auch das Problem, denn die CPU ist nicht in der Lage aktiv werte an Komponenten zu senden, sondern schaltet nur Leitungen an oder aus. Auf dieses Problem wird später in der ALU auch noch genauer eingegangen. So gibt es an einigen Stellen für bestimmte Probleme maßgeschneiderte Lösungen.
Ein dennoch großer Vorteil eines gemeinsamen Busses ist die einfache Erweiterbarkeit des Systems. So hatten wir am Anfang gar nicht mit einer RS232-Schnittstelle gerechnet, konnte diese später jedoch Problemlos einfügen, indem sie am Bus lauscht/schreibt und von der CPU geschaltet wird.
Die eigentliche Beschreibung des Mikrocontrollers erfolgt letztendlich in Python und kann im Beigelegten Projektordner eingesehen werden.

### 3.2.1 Steuerwerk

Unser Steuerwerk (CPU) ist eine State-Maschine (3.3), die anhand eines Präfix in den nächsten Status gelangt. In einem Status ist genau festgehalten zu welchem Zeitpunkt welche Steuerleitung (de-)aktiviert wird. Hier ist es möglich Substates zu haben (realsiert mit Hilfe eines hochlaufen Counters), falls dieser Befehl mehr als einen Takt benötigt um z.B. auf die Antwort der MMU zu warten. Das Bereit sein wird asynchron gemeldet, das heißt über eine eingehen Steuerleitung.




TODO: state chart machen

Abbildung 3.3: Cpu

### 3.2.2 Instruction Decoder

Der Instruction Decoder (IrD) ist zuständig dafür, einen 32Bit langen Wert in seine Einzelteile zu zerlegen und den einzelnen Komponenten zur Verfügung zu stellen. Dies soll ohne Takt geschehen, sondern sofort. Eine Entsprechende Abbildung ist unter Abbildung 3.4 zu sehen.
Der IrD stellt 8 Werte bereit und 3 Flags. Die Einzelnen Werte stehen für:

- aluop – Der Op-code für die ALU
- dest – Die Zieladresse für die Registerbank (Z)

[1] http://myhdl.org
[2] https://wiki.python.org/moin/PythonDecorators
[3] http://tams-www.informatik.uni-hamburg.de/applets/hades/webdemos/60-dcore/t3/processor.html

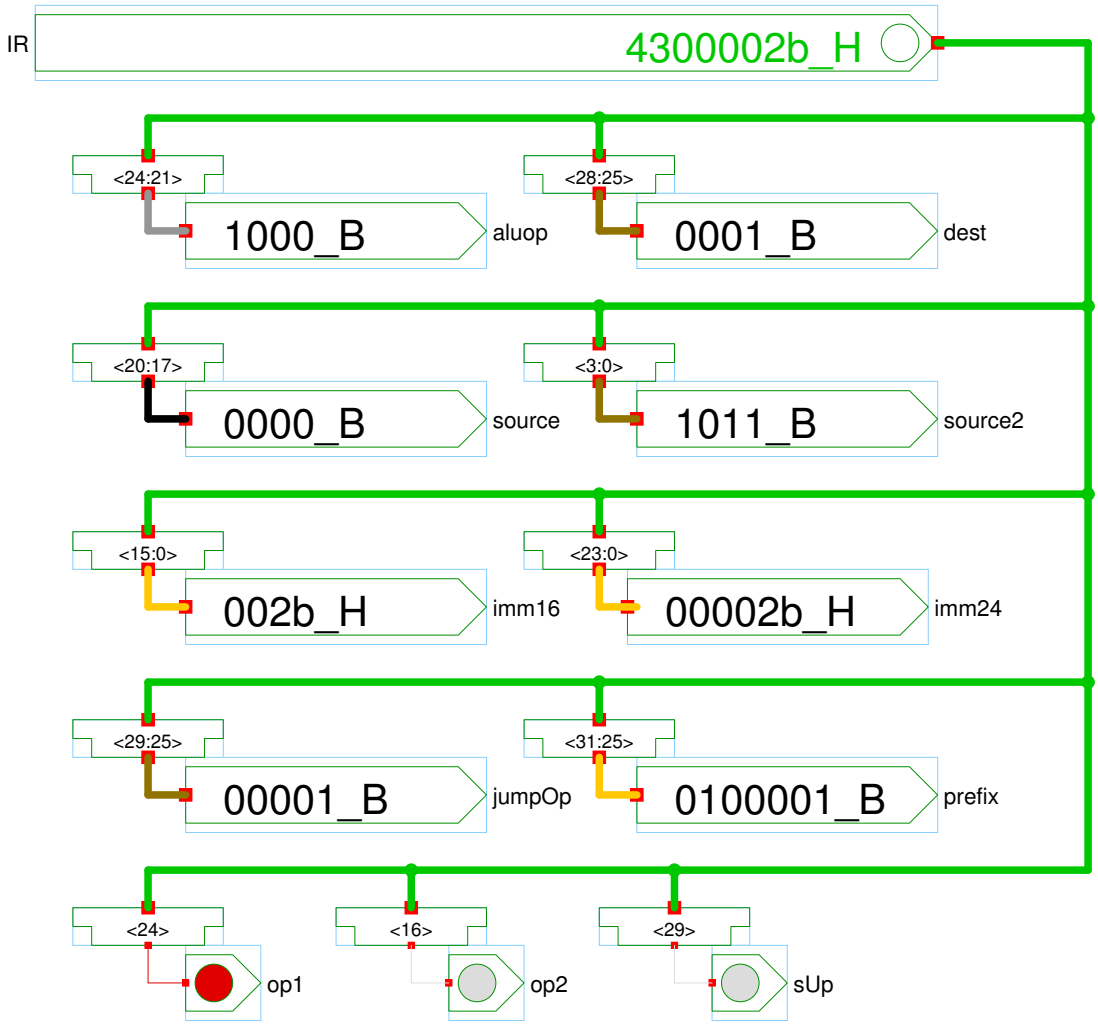



Abbildung 3.4: Instruction Decoder

- source – Die Quelladresse für die Registerbank (X)
- source2 – Die zweite Quelladresse für die Registerbank (Y)
- imm16 – Ein 16Bit breiter Immediate Wert für Berechnungen
- imm24 – Ein 24Bit breiter Immediate Wert für Berechnungen
- jumpop – Der Op-code für die Sprungeinheit.
- prefix – Das Instruktionsprefix für die CPU

Mit Hilfe der einzelnen Werte und Flags ist es nun möglich einen Befehl gemäß unserer Befehlsstruktur auszuführen. Da Werte nicht immer sinnig sein müssen, z.B. bei einem Sprungbefehl kann ein unsinniger Aluop entstehen, ist für die Hardware nicht weiter schlimm. Denn die berechnet einfach, wird ihr Ergebnis aber nie auf den Bus schreiben, denn die CPU unterbindet dies.

### 3.2.3 ALU

### 3.2.4 Jumpunit

### 3.2.5 Programmcounter

### 3.2.6 Memory

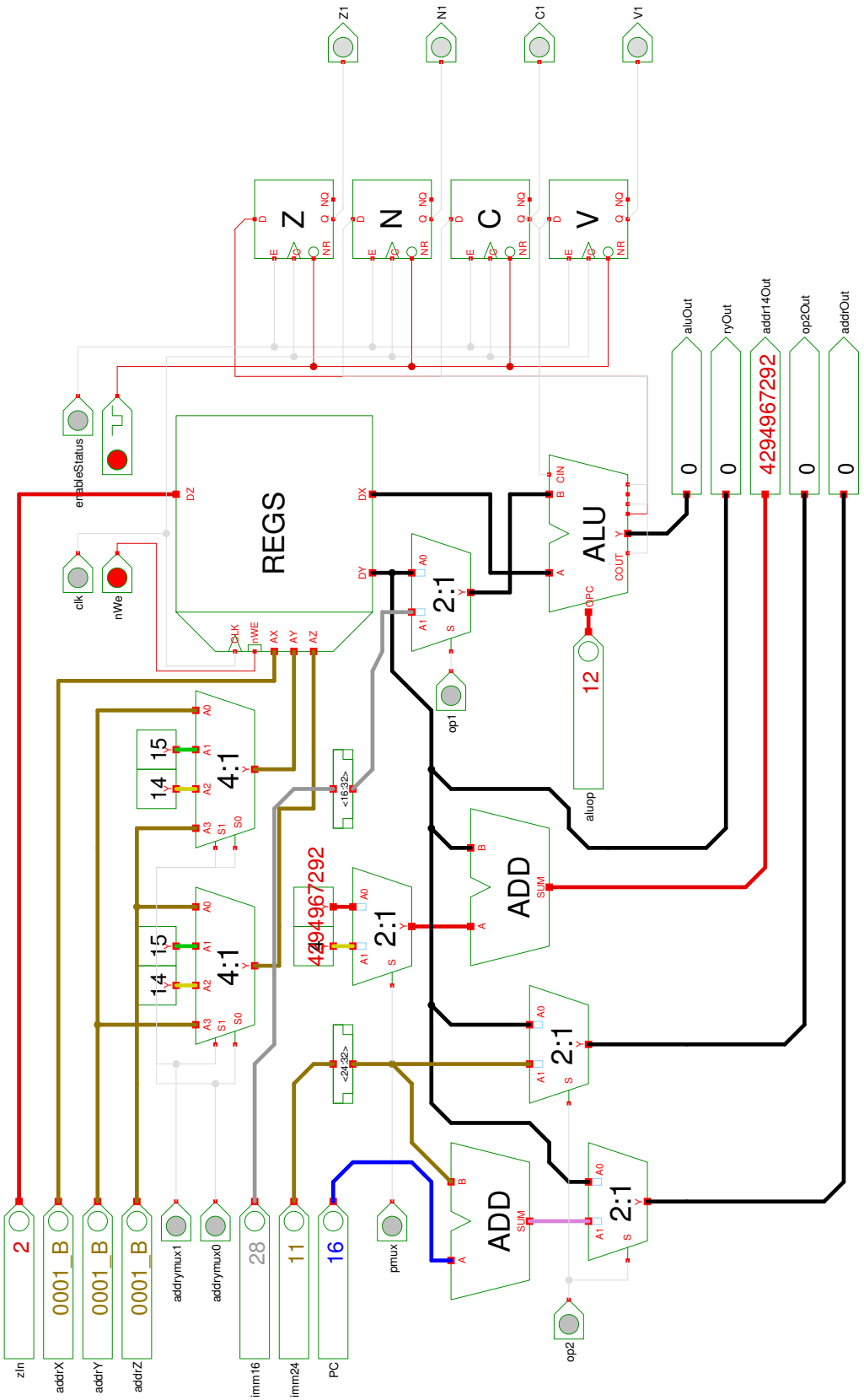



Abbildung 3.5: Address, Arithmetisch und Logische Einheit

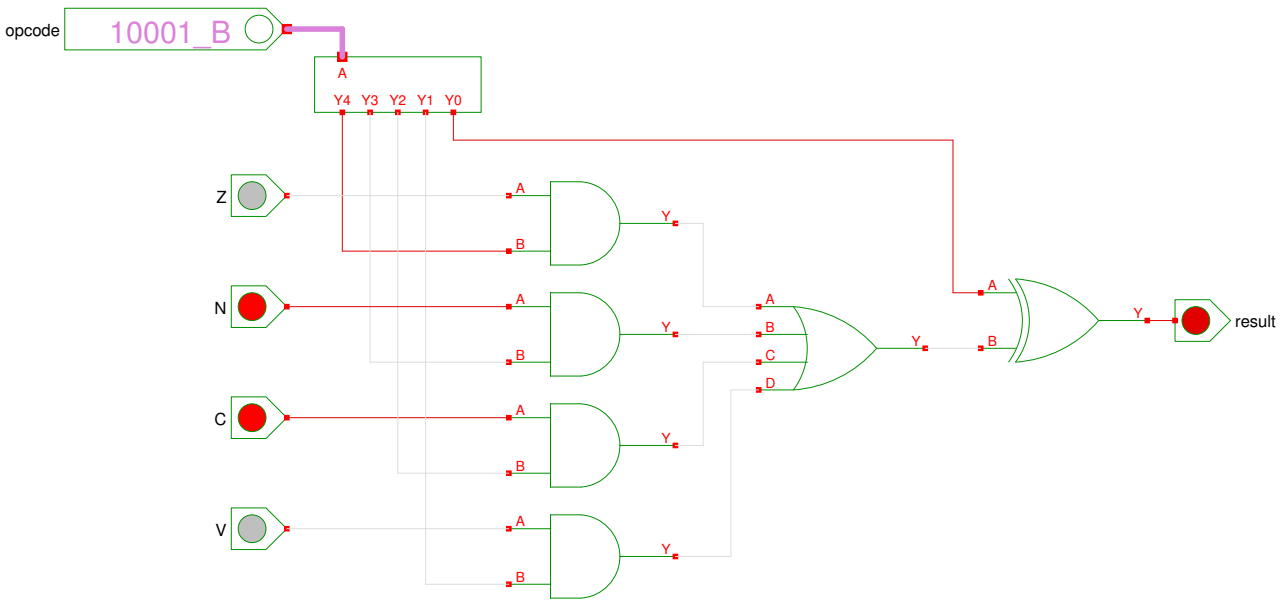



Abbildung 3.6: Sprung-Einheit

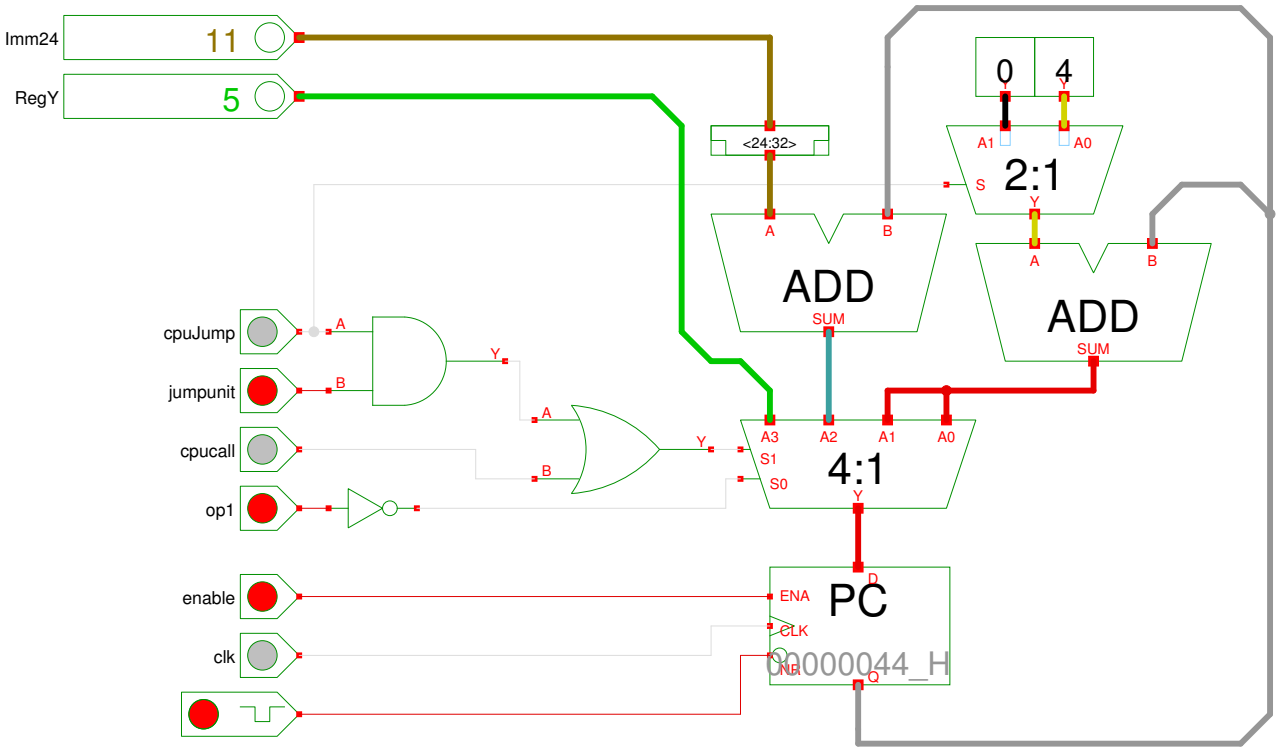



Abbildung 3.7: Programmcounter

# 4 Software

## 4.1 ISA(Felix O.&Marcel&Felix W.)

Den Aufbau unserer Instrution Set Architecture haben wir bereits beim ersten Gruppentreffen begonnen und stark umstritten. Im Verlauf des Projekts hat sich viel geändert, zum Teil motiviert durch Wünsche und Verbessungsideen unserer Mitglieder, zum Teil begründet in Fehlern oder Problemen mit den Befehlsworten oder deren Bitcode-Layout. In der Übersicht unter 6.1 ist unsere endgültige Instruction Set Architecture zu sehen. Wir haben von Beginn an den Fokus auf eine überschaubare Komplexität gelegt. Unsere arithmetischen Operationen zum Beispiel enthalten nur Befehle für Addition, Subtraktion und Multiplikation. Division haben wir nicht behandelt, da das ein tiefergehendes Auseinandersetzen mit Gleitkommazahlen nach sich gezogen hätte. Gleitkomma-Arithmetik haben wir von Beginn an als schwieriges, nicht notwendiges Feature eingestuft.

Für die Speicherzugriffsbefehle nutzen wir lediglich kustomisierte Varienten von load und store. Bemerkenswert ist jedoch unser breites Angebot an Sprungbefehlen. Diese haben wir so definiert, dass sie auf Labels referenzieren können. Labels sind in unserem Assembler Punkte im Code, die einen funktional zusammengehörigen Block markieren. Dies war am Anfang des Semesters noch völlig offen und hat sich erst mit der Zeit ergeben. Die Verwendung von Labels hatten wir zunächst, ähnlich wie die Gleitkomma-Arithmetik, als nicht so wichtig eingestuft. Für viele Programmabläufe sind Sprünge jedoch unabdingbar und daher umso dringender als Kommando im Assembler benötigt. Das endgültige Design der Springbefehle ist derart gestaltet, dass sie konditionierbar sind; die bekannte Befehle, die bei einer Vergleichsoperation ausgelöst werden können, etwa jump-less-or-equal, haben wir um Befehle erweitert, die auf jedem Statusflag-Status unserer ALU aufsetzen. So gibt es beispielsweise einen Jump-Befehl für den Overflow Fall, der aus Rahmen eines herkömmlichen Minimalassemblers herausfällt. Schlussendlich haben sich in unserer ISA auch noch Befehle als nützlich erwiesen, die eine spezielle Funktion direkt an der Hardware abstrahieren. Die ISA bietet uns die Möglichkeit direkt auf Buttons und LEDs auf dem FPGA zuzugreifen und diese Hardwarelemente in logisch unären Operationen anzusprechen. Dies ist sehr praktisch, denn eine Abfrage der Buttonelemente für jedes Programm als Sprachkonstrukt in herkömmlichem Assembler implementieren zu müssen, erschien uns sehr umständlich.

All diese Feinheiten der ISA haben sich mit der Zeit entwickelt. Beginnend bei unserem Primärziel, ein FPGA zu einem kleinen Taschenrechner werden lassen, hatten wir eine entsprechend kleine ISA. Sie umfasste lediglich den Befehlsatz für arithmetische Operationen, der fast genau so bis zur endgültigen ISA geblieben ist, Kontrollfluss regulierende Befehle wie call und jump, die wir zu diesem Zeitpunkt noch nicht weiter durchdacht hatten, sowie einen load und store Befehl. Wie 6.1 zeigt, hat sich die ISA im vergleich zu Abbildung 2.1 stark erweitert.

## 4.2 Emulator(Felix W.)

## 4.3 Assembler(Felix W.)

## 4.4 GUI/Debugger(Marcel)

# 5 Fazit(Alle)

## 5.1 ergebnisse

## 5.2 probleme

## 5.3 erkenntnisse

## 5.4 fazit

# 6 Appendix

## 6.1 ISA

| | | | |
|---|---|---|---|
| **Arithmetic** | Add | add $d, $s, $t | 00 S dddd 0000 ssss op3 |
| | Addc | adc $d, $s, $t | 00 S dddd 0001 ssss op3 |
| | Sub | sub $d, $s, $t | 00 S dddd 0100 ssss op3 |
| | Subc | sbc $d, $s, $t | 00 S dddd 0101 ssss op3 |
| | Revsub | rsb $d, $s, $t | 00 S dddd 0110 ssss op3 |
| | Revsubc | rsc $d, $s, $t | 00 S dddd 0111 ssss op3 |
| | Mul | mul $d, $s, $t | 00 S dddd 0010 ssss op3 |
| **Logical** | And | and $d, $s, $t | 00 S dddd 1000 ssss op3 |
| | Andn | adn $d, $s, $t | 00 S dddd 0011 ssss op3 |
| | Or | orr $d, $s, $t | 00 S dddd 1001 ssss op3 |
| | Xor | xor $d, $s, $t | 00 S dddd 1010 ssss op3 |
| | Orn | orn $d, $s, $t | 00 S dddd 1011 ssss op3 |
| **Shift / Rotate** | lsl | lsl $d, $s, $t | 00 S dddd 1100 ssss op3 |
| | asr | asr $d, $s, $t | 00 S dddd 1101 ssss op3 |
| | lsr | lsr $d, $s, $t | 00 S dddd 1110 ssss op3 |
| | ror | ror $d, $s, $t | 00 S dddd 1111 ssss op3 |
| **Jump** | Jump | jmp .label | 01 00001 op2 |
| | Jump | jmp op2 | 01 00001 op2 |
| | Jump Equal | jeq .label | 01 10000 op2 |
| | Jump Not Equal | jne .label | 01 10001 op2 |
| | Jump Less Than | jlt .label | 01 01000 op2 |
| | Jump Less Equal | jle .label | 01 11000 op2 |
| | Jump Greater Than | jgt .label | 01 11001 op2 |
| | Jump Greater Equal | jge .label | 01 01001 op2 |
| | Jump Overflow | jo .label | 01 00010 op2 |
| | Jump Not Overflow | jno .label | 01 00011 op2 |
| | Jump Carry | jc .label | 01 00100 op2 |
| | Jump Not Carry | jnc .label | 01 00101 op2 |
| **Memory** | Load | ld $d, $s | 100 dddd op2 |
| | Store | st $d, $s | 101 dddd op2 |
| | adr | adr $d, #imm | 110 dddd * imm24 |
| **Push / Pop** | Push | push $s | 11100 * op2 |
| | Pop | pop $d | 11101 * op2 |
| **Special** | Call | call .label | 111100 * op2 |
| | Clock | clk $d | 111101 * dddd |
| | Get Button | but $d | 1111100 * dddd |
| | Set Led | led $l | 1111101 op2 |
| | Rs232 read | rsr $d | 1111110 * dddd |
| | Rs232 transmit | rst $d | 1111111 op2 |

Tabelle 6.1: ISA

## 6.2 Abbildungsverzeichnis

# Abbildungsverzeichnis



## 6.3 Tabellenverzeichnis

# Tabellenverzeichnis