Hochschule für Angewandte
Wissenschaften Hof

Seminararbeit

Aufbau und Funktionsweiße eines Prozessors

Marco Vogel

unter Aufsicht von
Stefan Müller

3. Dezember 2017

Inhaltsverzeichnis

1 Motivation **3**

2 Zahlensysteme **3**

2.1 Binäre Darstellung von Zahlen 4

3 Logische Schaltglieder **6**

3.1 AND-Gatter 6

3.2 OR-Gatter 6

3.3 NOR-Gatter 6

3.4 XOR-Gatter 6

3.5 NOT-Gatter 6

3.6 Flip-Flops 6

4 Prozessorarchitekturen **6**

4.1 Von-Neumann Architektur 6

4.2 Harvard Architektur 8

4.3 CISC-Prozessoren 8

4.4 RISC-Prozessoren 9

4.5 Klassifizierung 9

4.5.1 Klassifizierung nach Flynn 9

4.5.2 Erlanger Klassifizierung 9

5 Aufbau und Funktion **9**

5.1 Register 9

5.1.1 Universalregister 10

5.1.2 Spezialregister 10

5.2 Steuerwerk 11

5.3 Adresswerk 12

5.4 Arithmetisch Logische Einheit 12

5.5 Memory Management Unit(evtl) 12

5.6 Bussysteme 12

6 Speicher **12**
6.1 RAM/ROM 12
6.2 Stack 12

7 Befehlsausführung **12**
7.1 Befehlszyklus 12
7.2 Schleifen 12

8 Besondere Ausführungsarten **12**
8.1 Interrupts 12
8.2 Exceptions 12
8.3 Subroutinen 12

9 Planung und Entwurf eines Prozessors **13**
9.1 Befehlsbreite 13
9.2 Befehlssatz 14
9.3 Speicher 16
9.3.1 RAM/ROM 16
9.3.2 Stack 16

10 Implementierung einer Prozessorsimulation in Logisim **16**
10.1 Logisim 16
10.2 Prozessor Komponenten 16
10.3 Entwicklung und Ausführung eines Programmes 19

List of Code Listings

1 C++ Code Primzahlenzählen 19
2 Assemblercode der main Methode 20
3 Assemblercode der checkIfPrime Methode 22

1 Motivation

2 Zahlensysteme

Unser geläufiges Zahlensystem ist das Dezimalsystem. Das bedeuted, dass Zahlen mit folgender Formel gebildet werden:

$$Z = \sum_{i=0}^{n-1} a_i * 10^i$$

Somit wird die Dezimalzahl 135 folgendermaßen gebildet:

$$Z = 1 * 10^2 + 3 * 10^1 + 5 * 10^0 = 135$$

Die Basis der Wertepotenz spiegelt das Zahlensystem wieder welches dargestellt wird, weshalb die Formel im Allgemeinen darstellbar ist für die Zahl Z mit Basis B:

$$Z = \sum_{i=0}^{n-1} a_i * B^i$$

Das dezimale Zahlensystem ist für Menschen sehr intuitiv zu verstehen. Da wir zehn Finger haben können wir optimal mit diesem Dezimalsystem zählen. Für Computer ist dieses Zahlensystem allerdings ungeeingnet. Ein Prozessor besteht aus vielen kleinen Transistoren, diese können entweder Strom fliesen lassen oder nicht. Somit bietet sich ein Zahlensystem an, welches nur zwei Zustände kennt. AN und AUS, Strom kann fliesen oder Strom kann nicht fliesen. Der deutsche Mathematiker Gottfried Wilhelm Leibniz entwickelte die Dyadik, die Darstellung von Zahlen durch 1 und 0. Diese Dar-

stellungsform ist für Prozessoren viel intuitiver, da sie selbst ebenfalls nur zwei Zustände kennen.[**?**].

2.1 Binäre Darstellung von Zahlen

Zahlen im Dualsystem können vorzeichenlos und vorzeichenbehaftet dargestellt werden.Vorzeichenlose Binärzahlen können mittels folgender Formel gebildet werden:

$$Z = \sum_{i=0}^{N-1} a_i * 2^i$$

Die Dezimalzahl 135 würde dann im Dualsystem dem Bitmuster 10000111 entsprechen, dargestellt durch folgende Konvertierung:

$$10000111b = 1*2^7+0*2^6+0*2^5+0*2^4+0*2^3+1*2^2+1*2^1+1*2^0 = 128+4+2+1 = 135d$$

Prozessoren haben immer eine begrenzte Anzahl an Bits zur Verfügung mit denen sie arbeiten können. Deshalb kann es während der Ausführung mit vorzeichenlosen Zahlen zu einem Überlauf kommen. Ein Überlauf tritt auf, wenn zum Beispiel auf einer 8-Bit CPU die Operation 255+1 ausgeführt wird, da als Ergebnis 0 geliefert wird. Das geschieht, da die Zahl 255s die Dualdarstellung 11111111 besitzt. Da 255d die größte darstellbare Zahl in 8-Bit ist wird die Addition von 1 einen Fehler verursachen. Das Ergebnis $256d$ benötigt zur dualen Darstellung 9 Bit($100000000b$), allerdings können nur 8 Bit gespeichert werden. Deshalb werden die ersten 8 Bit verwendet und das Ergebnis ist 0.

Tabelle 1: Rechnung mit Übertrag

Übertrag	**Binär**	**Dezimal**
-	11111111b	255d
-	00000001b	+1d
1	00000000b	256d
Ergebnis:	00000000b	0d

Diese Rechenoperation würde in der CPU das Carry Flag (Übertragsbit) setzen

um dem Programmierer darauf hinzuweißen, dass die letzte Operation keine richtigen Werte erzeugt hat. Den vorzeichenlosen Dualzahlen fehlt allerdings die Möglichkeit, negative Werte anzunehmen. Diese Eigenschaft bieten vorzeichenbehaftete Dualzahlen. Eine Dualzahl im sogenannten Zweierkomplement wird folgendermaßen gebildet:

$$Z = -a_{N-1} * 2^{N-1} + \sum_{i=0}^{N-2} a_i * 2^i$$

Die Formel kann zu Erklärungszwecken in zwei Teile gegliedert werden. Zum einen die erste Teil $-a_{N-1} * 2^{N-1}$, das Vorzeichenbit. Dieser sagt aus, dass das höchstwertige Bit einer vorzeichenbehafteten Zahl negativ gewertet wird. Der hintere Teil $\sum_{i=0}^{N-2} a_i * 2^i$ ist bereits aus der Erzeugung von vorzeichenlosen Dualzahlen bekannt. Die Bits werden nach ihrer Position gewichtet und ihre Wertigkeit aufaddiert. Die Zahl $10000111b$ im Zweierkomplement wird also wie folgt interpretiert:

$$10000111b = -1*2^7+0*2^6+0*2^5+0*2^4+0*2^3+1*2^2+1*2^1+1*2^0 = -128+4+2+1 = -121d$$

Bei den vorzeichenlosen Dualzahlen konnte es, wie oben beschrieben, zu einem Übertrag kommen, wenn der darstellbare Zahlenbereich überschritten wurde. Ein ähnliches Verhalten besitzen Zahlen im Zweierkomplement, allerdings kommt es zu einem Überlauf statt einem Übertrag. Zur Erklärung soll der 8-Bit Prozessor die Rechnung 127+1 durchführen. Hier wird nun nicht 128 als Ergebnis geliefert, sondern -128. Dies geschieht aufgrund der Interpretation von vorzeichenbehafteten Dualzahlen. Da das vorderste Bit nun gesetzt ist, interpretiert der Prozessor die Wertigkeit nun mit -128 statt 128, und da die restlichen sieben Bit null sind wird das Ergebnis als -128 interpretiert.

Tabelle 2: Rechnung mit Überlauf

Überlauf	**Binär**	**Dezimal**
-	01111111	127
-	00000001	+1
Ja	10000000	-128
Ergebnis:	10000000	-128

3 Logische Schaltglieder

3.1 AND-Gatter

3.2 OR-Gatter

3.3 NOR-Gatter

3.4 XOR-Gatter

3.5 NOT-Gatter

3.6 Flip-Flops

4 Prozessorarchitekturen

Mikroprozessoren besitzen immer einen eigenen, meist einzigartigen, Aufbau. Allerdings haben sich im Laufe der Entwicklung einige Architekturmerkmale ausgeprägt, welche die Prozessoren verbindet. Ziel dieser Architekturen ist es stets, die Ausführungsgeschwindigkeit eines Programmes zu beschleunigen.

4.1 Von-Neumann Architektur

Die Von-Neumann Architektur ist nach dem ungarisch-US-amerikanischen Mathematiker John von Neumann benannt. Er hat 1945 in dem Bericht (First Draft of a Report on the EDVAC) das Prinzip erstmals beschrieben. Die Von-Neumann Architektur besteht grundlegend aus folgenden Komponenten(siehe Abbildung 1):

- CPU
- Speicherwerk
- Ein-/Ausgabewerk
- Bus-System

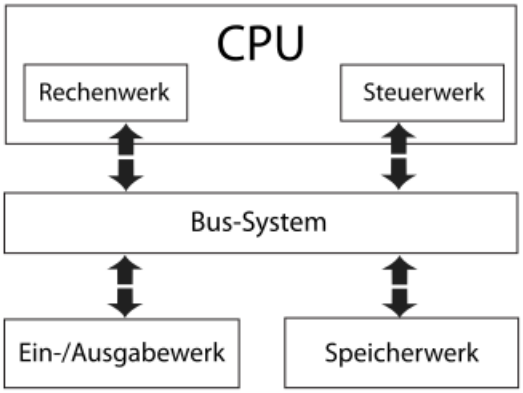

Abbildung 1: Komponenten Von-Neumann Architektur

Bevor John von Neumann dieses Architekturprinzip beschrieben hatte musste für eine bestimmte Aufgabe ein speziell darauf ausgelegter Rechner entworfen und gebaut werden. Mit der Von-Neumann Architektur war das nicht mehr nötig, es konnten verschiedene Programme auf dem gleichen Prozessor ausgeführt werden. Diese Funktion gab dem Prinzip den Namen programmgesteuerter Universalrechner(Stored-Program Machine)[1]. Ein sehr zentrales Prinzip dieser Archtitektur ist die Speicherung von Programmcode und Daten im gleichen Speicher. Das führt allerdings auch zu dem Problem, dass die CPU nicht unterscheiden kann ob geladenene Bytes Programmcode oder Daten enthalten. Diese Unterscheidung muss also der Programmierer vornehmen. Außerdem kann mit dieser Archtitektur nur jeweils Daten oder Code geladen werden und somit nur ein Befehl ausgeführt werden. Dieser Umstand erfordert einen speziellen Programmablauf der CPU, den so genannten Von-Neumann Zyklus. Dieser besteht aus den folgenden fünf Schritten welche nacheinander abgelaufen werden.

1. Instruction Fetch
2. Instruction Decode
3. Fetch Operands
4. Execute
5. Increment Program Counter (PC)

Im ersten Schritt wird aus dem Speicher der zu abzuarbeitende Befehl in das Befehlsregister geladen. Daraufhin wird im zweiten Schritt der Befehl vom Befehlsdekodierer verarbeitet und die nötigen Steuersignale an die CPU Komponenten weitergeleitet. Dann werden die Operanden welche für den Befehl benötigt werden geladen. Im vierten Schritt wird der Befehl schließlich von der ALU ausgeführt. Im letzten Schritt wird der Befehlszähler (Program Counter - PC) inkrementiert damit er im nächsten Zyklus bereits die Adresse des nächsten auszuführenden Befehls enthält.

[2].

4.2 Harvard Architektur

Die Harvard Architektur ist eine abgewandelte Form der Von-Neumann Architektur. Der größte Unterschied besteht darin, Codesegment und Datensegment in separaten Speichern zu verwalten. Diese Konzeption bringt den Vorteil, dass im Gegensatz zur Von-Neumann Architektur Befehle und Daten gleichzeitig geladen werden können. Um diesen Vorteil ausnutzen zu können benötigt ein Harvard Rechner allerdings auch getrennte Daten und Adressbusse.

In modernen x86 Prozessoren ist eine klare Unterscheidung zwischen Von-Neumann und Harvard Architektur nur schwer möglich. So zeigen die modernen CPU sich dem Entwickler zwar als pure Von-Neumann Maschinen, also mit gemeinsamen Code und Datenspeicher (RAM), allerdings besitzen sie intern einen getrennten Level-1 Cache für Instruktionen und Daten, was der Harvard-Architektur entspricht. Die beiden Architekturen haben also jeweils Vor- und Nachteile, wobei in modernen Prozessoren die beiden Konzepte verwendet werden um die Nachteile zu minimieren.

4.3 CISC-Prozessoren

Neben den beiden vorherigen CPU-Architekturen gibt es noch zwei weitere Designphilosophien für die Entwicklung von Prozessoren welche sich geschichtlich ergeben haben. In den Anfängen der Prozessorentwicklung gab es einige Faktoren, welche berücksichtigt werden mussten. So wurden Prozessoren bis in die 1970'er Jahre oft in Assembler pro-

grammiert. Um den Entwicklern für jeden möglichen Anwendungsfall einen einzelnen Assemblerbefehl zur Verfügung stellen zu können, begannen Prozessorhersteller, immer komplexere Befehle in den Befehlssatz zu integrieren. Diese Befehle beinhalteten oft mehrere Unterschritte, zum Beispiel das Lesen aus dem Speicher und dem Verrechnen zweier Variablen. Um solche Befehle ausführen zu können mussten die komplexen Befehle in mehrere Zwischenschritte aufgeteilt werden, welche dann vom Prozessor nacheinander abgearbeitet wurden. Prozessorhersteller entwickelten deshalb Microcode für den komplexen Befehlssatz, welche einen CISC-Befehl in mehrere Microcode Befehle dekodiert und diese ausführt. Das kostet zwar mehr Platz auf dem Chip für den Befehlsdekodierer, allerdings musste somit nicht mehr oft auf den Befehlsspeicher zugegriffen werden, was sehr viel Zeit kostet. Durch diesen Prozess wuchs die Befehlssatzgröße stark an und wurde immer komplexer. Solch eine ISA (Instruction Set Architecture) wird CISC(Complex Instruction Set Computer) genannt.

4.4 RISC-Prozessoren

Eine von IBM durchgeführte Studie hat Anfang der 1980er Jahre herausgefunden, dass Programme, welche auf einer CPU mit CISC ISA liefen, nur einen geringen Teil der zur Verfügung stehenden Instruktionen überhaupt verwenden.

4.5 Klassifizierung

4.5.1 Klassifizierung nach Flynn

4.5.2 Erlanger Klassifizierung

5 Aufbau und Funktion

5.1 Register

Register sind die schnellste Speichereinheit innerhalb einer CPU. Prozessoren besitzen eine vielfach höhere Ausführungsgeschwindigkeit als Arbeitsspeicher. Die CPU müsste ohne Register viele Taktzyklen auf Daten warten bevor sie diese verarbeiten könnte.

Register bieten deshalb die Möglichkeit, sehr kleine Datenmengen mit einer sehr geringen Latenz prozessorintern lesen und schreiben zu können. Übliche Registergrößen sind 8,16,32 oder 64 Bit.[3] Sie werden aus Flip-Flops aufgebaut welche jeweils genau ein Bit speichern können, das heißt ein 64 Bit Register besteht aus 64 gemeinsam gesteuerten Flip-Flops.[3] Diese Art der Datenspeicherung hat allerdings auch einige Nachteile. So verbrauchen Register sehr viel Energie und Platz auf dem Prozessordie, es werden deshalb keine großen Speichermengen zur Verfügung gestellt. (Nachteile evtl streichen)

5.1.1 Universalregister

Es werden zwei Arten von Registergruppen unterschieden.In einem Universalregister kann ein Programm Werte und Variablen abspeichern.Sie stehen außerdem einem Programmierer von außen offen, das heißt er kann auf jedes Universalregister direkt zugreifen und seinen Wert verändern.

5.1.2 Spezialregister

Spezialregister werden von einer CPU für interne Zwecke genutzt. Oft sind in Prozessoren ähnliche Spezialregister zu finden.

Der StackPointer(SP) ist ein Register welcher auf die aktuelle Position des Stacks im Speicher zeigt. Wenn der Befehl zu Speicherung eines Werts auf dem Stack ausgeführt wird inkrementiert die CPU automatisch, durch die interne Verschaltung des SP, den Wert des StackPointers. Dadurch zeigt das Register immer auf die nächste freie Speicheradresse im Stack.

Der InstructionPointer(IP) enthält die Adresse des nächsten Befehls im Programmspeicher der ausgeführt werden muss. Auch er wird nach der Abarbeitung eines Befehlszyklus als letzter Schritt inkrementiert. Dieses Register bietet allerdings die Möglichkeit einen anderen Wert zu laden. Das wird zur Realisierung von Sprüngen innerhalb des Programmcodes benötigt.

Das Statusregister(SR) werden zur Ausführung von bedingten Sprunganweisungen gebraucht. Sie werden auch Flagregister genannt da die ALU, in Abhängigkeit der zuletzt ausgeführten Rechenoperation, einzelne Bit(Flags) setzen kann. Auf die einzelnen

Flags und ihre Bedeutung wird im Abschnitt der ALU näher eingegangen

5.2 Steuerwerk

Das Steuerwerk ist für die Steuerung von internen Bussystemen des Prozessors zuständig. Es besteht aus zwei wesentlichten Komponenten. Im Befehlsregister (Programm Counter bzw. PC) ist die Adresse des nächsten Befehls enthalten, welcher ausgeführt werden soll. Der Befehl wird von der Adresse des Befehlsregisters in den Befehlsdekodierer geladen und analysiert. Falls nötig wird der Befehl in mehrere Schritte unterteilt, die nacheinander abgearbeitet werden müssen. Ob und wie viele solcher Schritte benötigt werden um einen bestimmten Befehl auszuführen bestimmt zum einen die Architektur des Prozessors und zum anderen der Befehl an sich. Bei RISC Prozessoren (→4.4) ist keine weitere Unterteilung in mehrere Befehle notwendig, bei CISC Prozessoren (→4.3) besitzt der Befehlssatz sehr viel kompliziertere Befehle welche nicht in einem Takt abgearbeitet werden können. Hier wird der Befehlsdekodierer den Befehl in die nötigen Teilbefehle umwandeln und nacheinander ausführen.

Da über ein Bussystem immer nur zwei Komponenten miteinander kommunizieren können, muss das Steuerwerk die Busse für die jeweiligen Komponenten wie zum Beispiel CPU zu Speicher und CPU zu Peripherie freischalten. [**?**]

5.3 Adresswerk

5.4 Arithmetisch Logische Einheit

5.5 Memory Management Unit(evtl)

5.6 Bussysteme

6 Speicher

6.1 RAM/ROM

6.2 Stack

7 Befehlsausführung

7.1 Befehlszyklus

7.2 Schleifen

8 Besondere Ausführungsarten

8.1 Interrupts

8.2 Exceptions

8.3 Subroutinen

9 Planung und Entwurf eines Prozessors

Der Inhalt der bisherigen Arbeit handelte von den Komponenten einer CPU und deren Funktionsweißen. Um den dargestellten Inhalt praktischer Vermitteln zu können, wird nun mittels einer Simulationssoftware eine CPU von Grund auf erstellt. Dieser Prozessor stellt keinen Vergleich zu modernen Prozessoren her. Er soll lediglich die Funktionsweiße der essentiellsten Bauteile beschreiben und einfache Operationen wie Sprünge und Subroutinen unterstützen.

9.1 Befehlsbreite

Am Anfang der Planung jeder CPU steht die Festlegung der benötigten Befehlsbreite. Je nachdem welche Features eingebaut werden sollen kann der Befehlssatz eingeteilt werden. Logisim bietet die Möglichkeit, einen 32-Bit Bus zu nuzten. Zu Erklärungszwecken werden die 32-Bit wie folgt aufgeteilt:

Tabelle 3: Befehlsbus

8-Bit	Opcode
8-Bit	Argument
16-Bit	Value

Opcode: Der Opcode beinhaltet den Befehl welche die CPU als nächstes Ausführen soll(z.B. MOV oder ADD). Es werden nicht mehr als 8-Bit benötigt, da nicht viele Befehle vorhanden sein müssen um die Basisfunktionalität einer CPU zu erzielen.

Argument: Das Argument wird nicht bei jedem Befehl verwendet. Diese 8-Bit sind eine Hilfestellung für Operationen bei denen eine genauere Spezifikation der zu ausführenden Tätigkeit benötigt wird. Beispielsweise wird bei der arithmetischen Operation ADD mit Hilfe des Argumentes angegeben, in welches Register das Ergebnis gespeichert werden soll.

Value: Die verbleibenden 16-Bit werden als Wertangabe benutzt. Durch diese 16-Bit wird gleichzeitig die Befehlsbusbreite innerhalb des Prozessors festgelegt, das heißt der Prozessor kann mit Zahlen arbeiten welche innerhalb der 16-Bit Grenze liegen (ohne

Vorzeichen maximal 65536). Einige Befehle in dieser CPU benötigen allerdings drei Parameter zur Ausführung. Um mit dem Argument drei Parameter bereitzustellen können die letzten 16-Bit in zwei 8-Bit Blöcke gespalten werden. Diese werden hier Quelle und Ziel genannt. Der Befehlssatz sieht bei diesen speziellen Befehlen folgendermaßen aus:

Tabelle 4: Befehlsbus mit drei Parametern

8-Bit	Opcode
8-Bit	Argument
8-Bit	Ziel
8-Bit	Quelle

Befehle, welche diese Aufteilung benötigen sind zum Beispiel ALU-Operationen oder der MOV Befehl, welcher den Wert eines Register in ein anderes schiebt.

9.2 Befehlssatz

Der Befehlssatz beschreibt die Befehle, welche die CPU ausführen kann.

Tabelle 5: Befehlssatz von VI-17

00000000	NOP
00000001	MOV
00000010	IN
00000011	STO
00000100	LEA
00000101	PUSH
00000110	POP
00000111	—
00001000	—
00001001	CALL
00001010	RETURN
00001011	ADD
00001100	SUB
00001101	INC
00001110	DEC
00001111	COMP
00010000	SHIFTL
00010001	SHIFTR
00010010	ROTL
00010011	ROTR
00010100	AND
00010101	OR
00010110	NOR
00010111	NAND
00011000	XOR
00011001	XNOR
00011010	JIT
00011011	JIF
00011100	JUMP

Die CPU soll die grundlegenden Aufgaben eines Prozessors erfüllen können. Die einzelnen Befehle des obigen Befehlssatzes werden nun kurz beschrieben.

00000000 NOP: No Operation. Es wird keine Operation ausgeführt.

00000001 MOV: Move. Überschreibt den Wert des Zielregisters mit dem Wert des Quellregisters.

9.3 Speicher

9.3.1 RAM/ROM

9.3.2 Stack

10 Implementierung einer Prozessorsimulation in Logisim

10.1 Logisim

Logisim ist ein Open Source Werkzeug für den Entwurf und die Simulation digitaler Schaltungen. Es bietet die Möglichkeit, größere Schaltungen aus kleineren Schaltungen herzustellen. Damit ist es möglich, ganze Prozessoren in Logisim zu entwerfen. Ein solch einfacher Prozessor soll nun im Folgenden implementiert werden.

10.2 Prozessor Komponenten

Der Prozessor besteht aus fünf Hauptkomponenten:

- Control Unit - Steuerungseinheit
- ALU - Arithmetisch Logische Einheit
- Registersatz
- RAM/Stack
- ROM

Control Unit - Steuerungseinheit: Die CU verarbeitet die Daten des Befehlsbusses und dekodiert die einzelnen Befehle, welche die CPU als nächstes ausführen muss. Der Befehlsbus wird mittels Komparatoren mit dem gesamtem Befehlssatz verglichen. Wenn ein Befehl gefunden wird sendet die CU die notwendigen Steuersignale an die einzelnen Komponenten des Prozessors, um zum Beispiel die Register zum beschreiben freizuschalten.

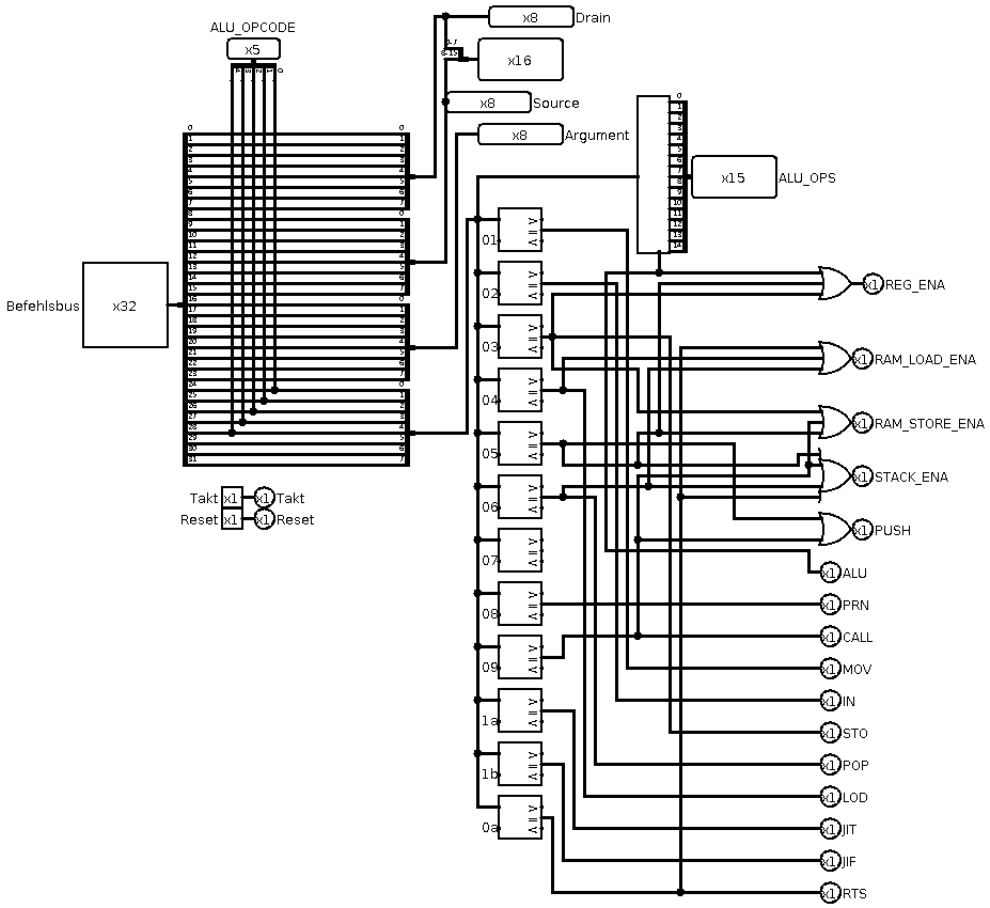

Abbildung 2: Darstellung des Steuerwerks

Registersatz:

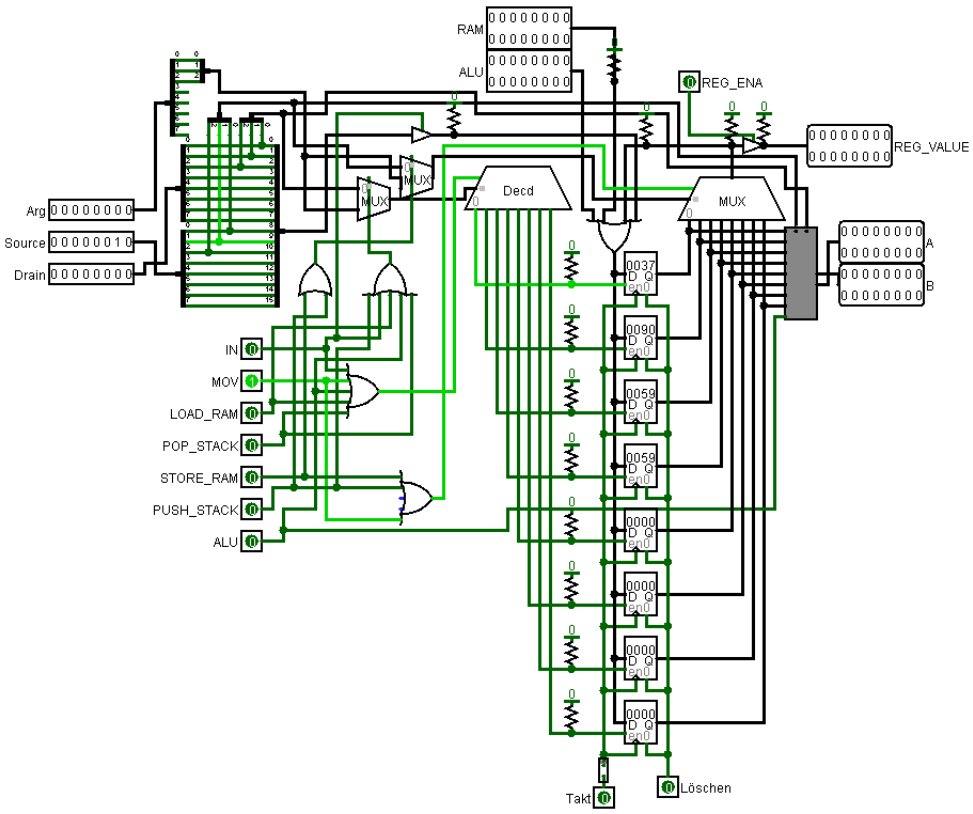

Abbildung 3: Darstellung des Registersatzes

10.3 Entwicklung und Ausführung eines Programmes

Um nun die Funktionalität der CPU zu zeigen wurde ein C++ Programm entwickelt welches alle Primzahlen bis $2^{16} = 65536$ ausrechnet und die Anzahl der Primzahlen auf dem Terminal ausgibt. Dieses Programm wurde unter einem aktuellen Ubuntu kompiliert.

```
bool checkIfPrime(unsigned int x){
        if(x<2) return false;
        unsigned int i=2;
        for(i;i<x;i++){
                if(x%i == 0){
                        return false;
                }
        }
        return true;
}

int main(int argc, char const *argv[])
{
        int counter=0;
        for(unsigned int i=1;i<65536;i+=2){
                if(checkIfPrime(i)){
                        counter++;
                }
        }
        std::cout << counter << std::endl; //Ausgabe 6492
        return 0;
}
```

Code Listing 1: C++ Code Primzahlenzählen

Um dieses Programm auf der VI-17 ausführen zu können muss es im Assembler der CPU neu geschrieben werden. Da Assembler eine sehr hardwarenahe Sprache ist, erleichtern wir uns die Entwicklung und betrachten den Assemblercode des C++ Programms, um die grobe Struktur sehen zu können, welche die CPU ausführt. Der Assemblercode kann mittels GDB betrachtet werden. Um den Umfang der Erklärungen nicht zu sprengen muss allerdings ein grundsätzliches Verständnis für den x86 Befehlssatz vorhanden sein.

```
Dump of assembler code for function main:
   0x000000000040085e <+0>:     push   %rbp
   0x000000000040085f <+1>:     mov    %rsp,%rbp
   0x0000000000400862 <+4>:     sub    $0x20,%rsp
   0x0000000000400866 <+8>:     mov    %edi,-0x14(%rbp)
   0x0000000000400869 <+11>:    mov    %rsi,-0x20(%rbp)
   0x000000000040086d <+15>:    movl   $0x0,-0x8(%rbp)
   0x0000000000400874 <+22>:    movl   $0x1,-0x4(%rbp)
   0x000000000040087b <+29>:    cmpl   $0xffff,-0x4(%rbp)
   0x0000000000400882 <+36>:    ja     0x40089c <main+62>
   0x0000000000400884 <+38>:    mov    -0x4(%rbp),%eax
   0x0000000000400887 <+41>:    mov    %eax,%edi
   0x0000000000400889 <+43>:    callq  0x400816 <_Z12checkIfPrimej>
   0x000000000040088e <+48>:    test   %al,%al
   0x0000000000400890 <+50>:    je     0x400896 <main+56>
   0x0000000000400892 <+52>:    addl   $0x1,-0x8(%rbp)
   0x0000000000400896 <+56>:    addl   $0x2,-0x4(%rbp)
   0x000000000040089a <+60>:    jmp    0x40087b <main+29>
   0x000000000040089c <+62>:    mov    -0x8(%rbp),%eax
   0x000000000040089f <+65>:    mov    %eax,%esi
   0x00000000004008a1 <+67>:    mov    $0x601060,%edi
   0x00000000004008a6 <+72>:    callq  0x4006a0 <_ZNSolsEi@plt>
   0x00000000004008ab <+77>:    mov    $0x400700,%esi
   0x00000000004008b0 <+82>:    mov    %rax,%rdi
   0x00000000004008b3 <+85>:    callq  0x4006f0 <_ZNSolsEPFRSoS_E@plt>
   0x00000000004008b8 <+90>:    mov    $0x0,%eax
   0x00000000004008bd <+95>:    leaveq
   0x00000000004008be <+96>:    retq
End of assembler dump.
```

Code Listing 2: Assemblercode der main-Methode

In der Zeile main +15 wird die Variable counter mit 0 initialisiert und auf dem Stack an Offset 0x8 des Base Pointers platziert. Anschließend wird die Laufvariable i der for-Schleife in Zeile main+22 mit dem Wert 1 an Offset 0x4 des Base Pointers im Stack initialisiert. Erster Durchlauf for-Schleife: Zeile +29 Die Laufvariable i wird mit 0xffff (dezimal: 65536) verglichen. Anschließend wird mittels des Assembler-Befehls ja (jump if above) geprüft, welche Flag der vorherige Compare Befehl gesetzt hat. Wenn im Flagregister das greater Bit gesetzt wurde, springt das Programm an die Adresse $0x000000000040089c$ (main+62), also aus for-Schleife raus, da die Schleifenbedingung

($i < 65536$) nicht mehr erfüllt ist. Wenn kein Sprung auftritt, läuft das Programm weiter und ruft an Stelle main+43 die Funktion checkIfPrime auf. Diese Funktion erwartet allerdings einen Übergabeparameter, dieser wird in Register $edi (main+41)abgelegt. Der Rückgabewert der Funktion steht daraufhin, wenn die Funktion durchlaufen und beendet wurde, in Register al. Da checkIfPrime den Rückgabetyp boolean besitzt steht in Register al entweder eine 0 wenn es keine Primzahl war, oder 1 wenn es eine Primzahl war die übergeben wurde. Der Befehl test an Stelle main+48 führt ein bitweise logisches UND zwischen al und al aus. Hier Prüft der Prozessor, ob das Ergebnis ungleich null war und setzt das ZF-Bit (Zero Flag). Wenn das Flag-Bit nicht gesetzt wurde wird das Programm ganz normal weitergeführt. Die counter Variable wird inkrementiert (main+52) und die Laufvariable i wird um zwei erhöht (main+56), daraufhin wird an Stelle main+29 gesprungen und der nächste Schleifendurchgang beginnt.

Das Code Listing 3 zeigt den Assemblercode der Funktion checkIfPrime. In Zeile 4 wird der Übergabeparameter, welcher sich in Register edi befindet, auf den Stack verschoben. Daraufhin wird mit dem Befehl cmpl dieser Übergabeparameter mit dem Wert 1 verglichen. Dafür werden die beiden Werte subtrahiert und das Ergebnis ausgewertet. Bei dieser Auswertung setzt die CPU automatisch die Flags für die Subtraktion. Wenn Beispielweise eine -2 übergeben wird und vom Befehl compl mit dem Wert 1 verglichen werden soll, so wird die ALU -2-1=-3 rechnen und dabei die Sign Flag(SF) setzen, da das Ergebnis negativ ist. Der nächste Befehl ist jg (Jump if greater), dieser Sprung wird laut Intel-Architektur-Dokumentation nur ausgeführt, wenn die beiden Flags ZF und SF **nicht** gesetzt, also null, sind. Diese sind null, wenn das Ergebnis zum einen nicht negativ (SF) und nicht null(ZF) ist.
Kurz gesagt: Die beiden Zeilen 7 und 11 stellen sicher, dass der Übergabeparameter größer als 1 ist. Im C++ Programm entspricht das der ersten Zeile der Funktion. Sollte eine der beiden Flags ZF bzw. SF nicht gesetzt sein, wird nicht gesprungen und in Zeile 13 eine 0 in das Rückgaberegister geschrieben. Daraufhin wird zum Ende der Funktion gesprungen und die Funktion ist beendet. Wenn der Sprung in Zeile 11 ausgeführt wird, dann springt das Programm zu Zeile 20 in der die Laufvariable i mit dem Wert 2 initialisiert wird. Die Zeilen 27 bis 30 sind analog zu Codelisting 2 die Prüfung der Laufvariable in der for-Schleife, ob die Abbruchbedinggung bereits erfüllt ist. In der

```
Dump of assembler code for function _Z12checkIfPrimej:
   0x0000000000400816 <+0>:     push   %rbp
   0x0000000000400817 <+1>:     mov    %rsp,%rbp
   0x000000000040081a <+4>:     mov    %edi,-0x14(%rbp)
   0x000000000040081d <+7>:     cmpl   $0x1,-0x14(%rbp)
   0x0000000000400821 <+11>:    ja     0x40082a <_Z12checkIfPrimej+20>
   0x0000000000400823 <+13>:    mov    $0x0,%eax
   0x0000000000400828 <+18>:    jmp    0x40085c <_Z12checkIfPrimej+70>
   0x000000000040082a <+20>:    movl   $0x2,-0x4(%rbp)
   0x0000000000400831 <+27>:    mov    -0x4(%rbp),%eax
   0x0000000000400834 <+30>:    cmp    -0x14(%rbp),%eax
   0x0000000000400837 <+33>:    jae    0x400857 <_Z12checkIfPrimej+65>
   0x0000000000400839 <+35>:    mov    -0x14(%rbp),%eax
   0x000000000040083c <+38>:    mov    $0x0,%edx
   0x0000000000400841 <+43>:    divl   -0x4(%rbp)
   0x0000000000400844 <+46>:    mov    %edx,%eax
   0x0000000000400846 <+48>:    test   %eax,%eax
   0x0000000000400848 <+50>:    jne    0x400851 <_Z12checkIfPrimej+59>
   0x000000000040084a <+52>:    mov    $0x0,%eax
   0x000000000040084f <+57>:    jmp    0x40085c <_Z12checkIfPrimej+70>
   0x0000000000400851 <+59>:    addl   $0x1,-0x4(%rbp)
   0x0000000000400855 <+63>:    jmp    0x400831 <_Z12checkIfPrimej+27>
   0x0000000000400857 <+65>:    mov    $0x1,%eax
   0x000000000040085c <+70>:    pop    %rbp
   0x000000000040085d <+71>:    retq
End of assembler dump.
```

Code Listing 3: Assemblercode der checkIfPrime-Methode

Schleife werden die Zeilen 35 bis 59 ausgeführt. Der Befehl idvl führt eine Division aus, wobei der Rest in Register EDX gespeichert wird. Nachdem EDX in EAX verschoben wurde wird mittels des Befehls test EAX,EAX (Zeile 48) geprüft, ob das Register null ist, also auch der Rest der Division null ist. Sollte dem so sein, so springt das Programm ans Ende und schreibt eine 0 in Übergaberegister EAX.

Literatur

[1] *Taschenbuch Mikroprozessortechnik*. Hanser Fachbuchverlag, 2010.

[2] Universität-Köln, "Arbeitsweise einer cpu - von neumann-zyklus."

[3] K. Wüst, *Mikroprozessortechnik, Grundlagen, Architekturen, Schaltungstechnik und Betrieb von Mikroprozessoren und Microcontrollern*. Vieweg+Teubner, 4 ed., 2011.