



HOCHSCHULE FÜR ANGEWANDTE
WISSENSCHAFTEN HOF

SEMINARARBEIT

**Aufbau und Funktionsweise eines
Prozessors**

Marco Vogel

unter Aufsicht von
Stefan Müller

December 18, 2017

Contents

1	Motivation	3
2	Zahlensysteme	3
2.1	Binäre Darstellung von Zahlen	4
3	Prozessorarchitekturen	6
3.1	Von-Neumann Architektur	6
3.2	Harvard Architektur	8
3.3	CISC-Prozessoren	8
3.4	RISC-Prozessoren	9
3.5	Klassifizierung nach Flynn	11
4	Aufbau und Funktion	11
4.1	Register	11
4.1.1	Universalregister	11
4.1.2	Spezialregister	12
4.2	Steuerwerk	12
4.3	Adresswerk	13
4.4	Arithmetisch Logische Einheit	13
4.5	Memory Management Unit	16
4.6	Bussysteme	17
5	Speicherverwaltung	17
5.0.1	Paging	17
5.0.2	Speichersegmentierung	19
5.1	Virtueller Speicher	19
5.2	Caches	19
6	Befehlsausführung	19
6.1	Befehlszyklus	19
6.2	Schleifen	19

6.3	Subroutinen	22
7	Besondere Ausführungsarten	24
7.1	Interrupts	24
7.2	Exceptions	26
8	Leistungsaspekte eines Prozessors	27
8.1	Pipelines und Skalarität	27
8.2	Mehrkern-Prozessoren	27
9	Planung und Entwurf eines Prozessors	27
9.1	Befehlsbreite	27
9.2	Befehlssatz	28
9.3	Speicher	30
9.3.1	RAM/ROM	30
9.3.2	Stack	30
10	Implementierung einer Prozessorsimulation in Logisim	30
10.1	Logisim	30
10.2	Prozessor Komponenten	30
10.3	Entwicklung und Ausführung eines Programmes	33

List of Code Listings

1	C Code einfache Schleife	20
2	Assembler Code einfache Schleife	21
3	C Code Funktionen	22
4	Assembler Code Funktionen	23
5	C++ Code Primzahlen zählen	33
6	Assemblercode der main Methode	35
7	Assemblercode der checkIfPrime Methode	36

1 Motivation

2 Zahlensysteme

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

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

Die Dezimalzahl 135 wird z.B. wie folgt gebildet:

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

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

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

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 ungeeignet. Ein Prozessor besteht aus vielen kleinen

Transistoren. Diese können entweder Strom fließen lassen oder nicht. Somit bietet sich ein Zahlensystem an, welches nur zwei Zustände kennt: AN und AUS. Strom kann fließen oder Strom kann nicht fließen. Der deutsche Mathematiker Gottfried Wilhelm Leibniz entwickelte die Dyadik; die Darstellung von Zahlen durch 1 und 0. Diese Darstellungsform ist für Prozessoren optimal geeignet, 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.

Diese Rechenoperation würde in der CPU das Carry Flag (Übertragsbit) setzen um dem Programmierer darauf hinzuweisen, dass die letzte Operation keine richtigen Werte erzeugt hat. Den vorzeichenlosen Dualzahlen fehlt allerdings die Möglichkeit, negative

Table 1: Rechnung mit Übertrag

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

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 Erklärungs Zwecken 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.

Table 2: Rechnung mit Überlauf

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

3 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.

3.1 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

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



Figure 1: Komponenten Von-Neumann Architektur

Machine)“[1]. Ein sehr zentrales Prinzip dieser Architektur ist die Speicherung von Programmcode und Daten im gleichen Speicher. Das führt allerdings auch zu dem Problem, dass die CPU nicht selbstständig unterscheiden kann ob geladenene Bytes Programmcode oder Daten enthalten. Diese Unterscheidung muss also der Programmierer vornehmen. Außerdem kann mit dieser Archtitektur zu jedem Takt 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 ablaufen.

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

Im ersten Schritt wird aus dem Speicher der 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].

3.2 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 sich die modernen CPU 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 maximale Leistung zu erzielen.

3.3 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 programmiert. 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. Der Befehlsdekodierer nimmt zwar auf einem Prozessor mehr Platz ein, allerdings musste somit nicht mehr oft auf den Befehlsspeicher zugegriffen werden, was viele Taktzyklen Zugriffszeit gekostet hat. 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.

3.4 RISC-Prozessoren

Mehrere Faktoren führten dazu, dass die CISC-ISA einige Nachteile entwickelte, weshalb die RISC-ISA (Reduced Instruction Set Computer) entwickelt wurde.

Eine von IBM durchgeführte Studie hat Anfang der 1980er Jahre ergeben, dass Programme, welche auf einer CPU mit CISC ISA liefen, nur einen geringen Teil der zur Verfügung stehenden Instruktionen überhaupt verwendeten. Das bedeutet, dass Programme den komplexen Befehlssatz, welche eine CISC CPU bietet, gar nicht ausnutzen sondern nur die simplen und universellen Befehle verwenden.

Ein weiterer Grund für die Entwicklung der RISC Computer war, dass die Zahl der Entwickler, welche in Assembler programmierten, immer weniger wurden. Mit dem Auftreten der ersten Hochsprachen, insbesondere C, und ihren Compilern, mussten Programmierer nicht mehr den Befehlssatz eines Prozessors kennen. Die Compiler wandelten den Programmcode der Hochsprache erst in Assembler und daraufhin in eine maschinenlesbare Form um. Durch diese Abstraktionsschicht zwischen Prozessor und Programmierer war die kompakte Struktur von Assemblerbefehlen kein Designfaktor für Prozessoren mehr, da die meisten Programmierer keinen Assemblercode mehr betrachten mussten.

Die Latenz zwischen Prozessor und Hauptspeicher in den 1980er Jahren stetig

gesunken. Es wurden also deutlich weniger Taktzyklen des Prozessors geopfert, wenn er aus dem Hauptspeicher Befehle anforderte. Dadurch entfiel der Vorteil von nur wenigen Zugriffen auf den Hauptspeicher.

Daraus hat sich dann die RISC Philosophie ergeben. Im Gegensatz zu den vorher genannten Punkten implementiert RISC folgende Eigenschaften.

Der Befehlssatz ist auf die wesentlichen Befehle beschränkt. Während CISC Befehlssätze oft 200 oder mehr Befehle beinhalten sind in RISC Befehlssätzen oft nur 100 oder weniger Befehle vorhanden. Auf die Mikroprogrammierung wird verzichtet. Es werden also nicht mehr die Befehle aus dem Befehlsspeicher intern in Mikrobefehle dekodiert. Das spart auf dem Chip Fläche, da der Dekodierer keine komplexen Aufgaben mehr hat. Alle Befehle sind also fest in dem Prozessor implementiert.

Ein Befehl soll genau einen Takt zur Ausführung benötigen. Wenn bei CISC ein Befehl in mehrere Mikrobefehle dekodiert wird, benötigen diese mehrere Takte bis alle ausgeführt wurden. Dies wird bei RISC vermieden. Jeder geladene Befehl wird dekodiert und spricht genau eine Hardwareeinheit an.

Eine weitere Eigenschaft der RISC Philosophie ist die Kommunikation mit dem Hauptspeicher nur über Load und Store Befehle. Dadurch wird die Anzahl der Zugriffe auf den Hauptspeicher reduziert, was die Ausführungszeit senkt. Daraus folgt, dass ein ALU Befehl nur auf Register anwendbar ist, da Register auf Prozessortakt betrieben werden und somit keine Latenz zwischen den beiden Hardwareeinheiten entsteht. Der Befehl, zwei Register zu verrechnen, kann in einem Taktzyklus abgearbeitet werden. Dadurch entfällt wiederum die Notwendigkeit der Mikroprogrammierung. In einer CISC CPU ist zwar eine Speicherverrechnung zwischen Register und Speicherzelle etc. möglich, allerdings nicht skalar, also in einem Takt, sondern in mehreren Teilschritten, zum Beispiel Speicherzelle laden, verrechnen, zurückschreiben.

Da der Speicherzugriff auch in modernen Prozessoren eine zeitkritische Komponente des Funktionsmodells ist besitzen RISC Prozessoren einen sehr großen Registersatz, üblicherweise 16 oder mehr Register. Das ermöglicht einem Programm, viele seiner Variablen in den Registern zu halten um Hauptspeicherzugriffe zu vermeiden. Hier sind allerdings die Compiler der Programmiersprachen in der Pflicht, optimierten Binärcode für RISC Prozessoren zu generieren. Alle beschriebenen Vorteile treten nicht

ein, wenn sie von den Compilern nicht unterstützt werden. Wenn ein für CISC Prozessoren geschriebenes Programm auf einem RISC Prozessor ausgeführt wird, funktioniert es, aber die Variablen werden zum größten Teil im Stack gespeichert, einem Teil des Hauptspeichers, und der Großteil zur Verfügung stehender Register wird nicht verwendet.

Aktuelle Prozessoren sind meist CISC Prozessoren. Mit RISC-V existiert aber ein Projekt welches die Absicht hat wieder moderne und konkurrenzfähige RISC Prozessordesigns zu entwickeln und als freie ISA zur Verfügung zu stellen .

3.5 Klassifizierung nach Flynn

4 Aufbau und Funktion

4.1 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)

4.1.1 Universalregister

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

4.1.2 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) wird zur Ausführung von bedingten Sprunganweisungen gebraucht. Sie werden auch Flagregister genannt da die ALU, in Abhängigkeit der zuletzt ausgeführten Rechenoperation, einzelne Bits (Flags) setzen kann. Auf die einzelnen Flags und ihre Bedeutung wird im Abschnitt der ALU näher eingegangen

4.2 Steuerwerk

Das Steuerwerk ist für die Steuerung von internen Bussystemen des Prozessors zuständig. Es besteht aus zwei wesentlichen Komponenten. Im Befehlsregister (Program 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 (\rightarrow 4.4) ist keine weitere Unterteilung in mehrere Befehle notwendig, bei CISC Prozessoren (\rightarrow 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. [?]

4.3 Adresswerk

4.4 Arithmetisch Logische Einheit

Die ALU (Arithmetic Logic Unit) ist der Teil einer CPU, welcher die eigentliche Datenverarbeitung durchführt. Sie verfügt über keinen eigenen Speicher, die Ergebnisse müssen also in den Registern gespeichert werden. Die ALU ist ein Schaltwerk, welches einfache Operationen auf meist zwei Operanden ausführen kann. So kann das Rechenwerk einer CPU meistens arithmetische, logische und bitschiebende Operationen ausführen. Arithmetische Operationen umfassen Addition und Subtraktion, seltener auch Multiplikation und Division. Addierer und Subtrahierer sind als Hardwareeinheit in die ALU integriert, wohingegen Multiplikation und Division meist algorithmisch durchgeführt werden. Die ALU alleine hat wenig Wirkung, sie obliegt der Kontrolle des Steuerwerkes, das steuert, welche Operation ausgeführt wird, mit welchen Operanden sie ausgeführt wird und in welches Register das Ergebnis gespeichert werden soll.

Rechenwerke sind innerhalb eines Prozessors oft unterschiedlich aufgebaut. So haben sehr einfache Prozessoren häufig ein spezielles Register, in welchem das Ergebnis einer ALU Operation automatisch gespeichert wird, den so genannten Akkumulator. Solch ein Register hat den Vorteil, das auf die Angabe eines Zieles bei der Programmierung verzichtet werden kann, da der Ausgang der ALU fest mit dem Akkumulator verbunden ist. Des weiteren ist der Ausgang des Akkumulators fest mit einem der zwei Eingänge der ALU verbunden. Man kann sich den Akkumulator also als fest vorgeschaltetes Register des Rechenwerks vorstellen, in dem auch das Ergebnis gespeichert wird. Deshalb muss bei einer Rechenoperation vom Programmierer nur ein Operand angegeben werden.

Beispiel: Wenn im Akkumulator der Wert 10d steht und der Assemblerbefehl ADD RAX ausgeführt wird, dann wird die ALU vom Steuerwerk angewiesen eine Addition

auszuführen und übergibt als Parameter am zweiten Eingang den Wert des Registers RAX. Der erste Eingang ist mit dem Akkumulator verbunden und übergibt deshalb den Wert 10d. Das Ergebnis der Berechnung wird wiederum in den Akkumulator gespeichert und überschreibt den vorherigen Wert 10d.

Diese feste Konfiguration der ALU hat allerdings einige Nachteile weshalb in den meisten modernen Prozessoren eine ALU implementiert ist, die mit keinem Register fest verbunden ist. Eine Rechenwerksoperation muss also immer zwei Quellregister und ein Zielregister angeben. Diese Konfiguration ist zwar sehr flexibel in der Übergabe der Speicherorte, allerdings benötigt sie zusätzliche Schaltungslogik auf dem Chip und die Codekomplexität nimmt zu.

Beispiel: Bei der Ausführung des Befehls `ADD RAX,RBX,RCX` werden die Werte der beiden Register RAX und RBX addiert und das Ergebnis in RCX gespeichert.

Die ALU speichert zwar das Ergebnis in dem ihr zugewiesenen Register, allerdings kann der Prozessor keine Eigenschaften der letzten Operation unterscheiden. Diese Fähigkeit wird aber benötigt, wenn der Programmierer beispielsweise anfordert, dass das Programm an eine andere Adresse springt, sollte das letzte Rechenergebnis 0 ergeben haben. Damit der Prozessor diesen Verlauf der letzten Rechenoperation speichern kann ist den meisten Prozessoren ein Statusregister integriert. Das Statusregister speichert nicht wie die Universalregister einen Wert in mehreren Bits ab, sondern jedes Bit steht im Register für eine bestimmte Eigenschaft der vorhergehenden Rechenoperation. Diese Bits werden Flags genannt. Eine 1 bedeutet dass die letzte Operation diese Bedingung erfüllt hat, eine 0 das Gegenteil. Welche Flags ein Prozessor unterstützt und wie diese angeordnet sind unterscheidet sich von Prozessor zu Prozessor. Allerdings sind die fünf grundlegendsten Flags in fast allen Prozessoren implementiert.

Zero-Flag (Nullbit): Das Zero-Flag ist sehr simpel aufgebaut. Wenn das letzte Ergebnis der ALU gleich Null war wird das Bit auf 1 gesetzt, ansonsten auf 0. Mit diesem Flag lässt sich eine Abbruchbedingung für eine Schleife leicht prüfen. Die Anweisung:

$$\text{for}(\text{int } i = 5; i > 0; i - -)\{\}$$

wird solange durchlaufen bis das Ergebnis der ALU Operation ($i - -$) gleich Null ist

und damit das Zero-Flag gesetzt wird. Dann wird die Schleife abgebrochen und der Programmablauf fortgesetzt.

Table 3: Schritte der for-Schleife

i	Zero-Flag	Nächster Schritt
5	0	Weiter
4	0	Weiter
3	0	Weiter
2	0	Weiter
1	0	Weiter
0	1	Abbruch

Tabelle 3 zeigt die Nutzung des Zero-Flags zur Prüfung der Abbruchbedingung von Abbildung TODO.

Carry-Flag (Übertragsbit): Das Carry-Flag zeigt an, ob es bei der letzten ALU-Operation zu einem Übertrag gekommen ist. Wie in Abbildung TODO(Übertrag) bereits dargestellt, kommt es zu einem Überlauf, wenn das Ergebnis nicht mehr korrekt mit den vorhandenen Bits dargestellt werden kann. Diese Flag wird beim Rechnen mit vorzeichenlosen Zahlen verwendet.

Rechnung	Ergebnis (d)	Ergebnis (b)	Carry-Flag
253+2	255	11111111	0
253+3	256	00000000	1

Table 4: Beispiele Carry-Flag

In Abbildung ?? sind beispielhaft zwei Rechenoperationen dargestellt. Es wird angenommen dass die Operation von einer 8-Bit ALU ausgeführt wird. In der ersten Zeile wird die Summe aus 253 und 2 berechnet. Das Ergebnis kann gerade noch als 8-Bit Dualzahl dargestellt werden ($11111111b = 255d$), das Carry-Bit wird deshalb nicht von der ALU gesetzt. Das Ergebnis der Rechnung in der zweiten Zeile würde neun Bits benötigen um korrekt dargestellt werden zu können ($100000000b = 256d$). Da das Rechenwerk aber nur eine Breite von 8 Bit hat wird im Zielregister der Wert der ersten 8 Bit des Ergebnisses, also null, gespeichert. Um diese fehlerhafte Rechnung

anzuzeigen wird nun das Carry Bit im Flagregister gesetzt. Sollte der letzte ausgeführte ALU Befehl ein Schiebefehl gewesen sein, so wird das Carry Flag dazu verwendet den Wert des herausgeschobenen Bits anzuzeigen.

Sign-Flag (Vorzeichenbit): Das Vorzeichenbit entspricht dem MSB (most significant bit) des Ergebnisses. Das MSB ist das höchstwertige Bit einer Dualzahl und repräsentiert im Zweierkomplement das Vorzeichen der Zahl.

Overflow-Flag (Überlauf-Bit): Diese Flag wird beim Rechnen mit Zahlen im Zweierkomplement benötigt. Es wird gesetzt, wenn bei einer Addition oder Subtraktion ein Übertrag auf das MSB stattfindet, also das höchstwertige Bit verändert wird.

Rechnung (d)	Ergebnis (Zweierkomplement)	Ergebnis (b)	Overflow-Bit
122+5	127	01111111	0
122+6	-128	10000000	1

Table 5: Rechnen im Zweierkomplement mit 8-Bit Zahlen

In Abbildung ?? sind zwei Rechnungen dargestellt. Im Zweierkomplement hat eine Zahl mit n Bits folgenden darstellbaren Zahlenbereich:

$$-2^{n-1} + \dots + 2^{n-1} - 1$$

In Tabelle ?? wird mit 8-Bit Zahlen gearbeitet, also reicht der darstellbare Zahlenbereich für das Beispiel von -128 bis $+127$. Die erste Rechnung ergibt 127, die höchste positive darstellbare Zahl im Zweierkomplement mit 8-Bit. Es kommt zu keinem Übertrag auf das MSB, also wird das Overflow Bit nicht gesetzt. In der zweiten Zeile kommt es zu einem Übertrag auf das MSB und das Ergebnis wird verfälscht, da $+128$ nicht mehr im darstellbaren Zahlenbereich für diese Rechnung liegt. Um das falsche Ergebnis anzuzeigen wird die ALU das Overflow Bit auf 1 setzen.

4.5 Memory Management Unit

Die Speicherverwaltungseinheit (MMU) ist ein relativ neuer Bestandteil einer CPU. Sie befindet sich am Adressbus und steuert die Adressierung zwischen der CPU und

dem Speicher. Ihre Aufgabe ist die Umwandlung von virtuellen in physische Speicheradressen.

4.6 Bussysteme

5 Speicherverwaltung

Am Anfang der Prozessorentwicklung war Hauptspeicher nur in kleinen Kapazitäten verfügbar. Ein Programm war häufig zu groß um komplett in den Hauptspeicher geladen zu werden. Deshalb wurden so genannte Overlays verwendet. Wenn ein Programm geladen werden sollte, wurde ein Bereich im Hauptspeicher für Overlays reserviert. Der Programmierer musste dann sein Programm in Segmente (Overlays) unterteilen und diese dann während der Laufzeit nacheinander in den Hauptspeicher zur Ausführung laden. Diese Speicherverwaltungsmethode hatte aber Nachteile. Der Entwickler musste neben seinem geschriebenen Programm auch noch auf die Speicherverwaltung achten, was mehr Arbeit verursachte. Außerdem erhöhte das ständige Nachladen von Overlays in den Speicher die Ausführungszeit erheblich.

Um die Programmierer von der zeitraubenden Arbeit der manuellen Speicherverwaltung zu entlasten, hat sich das System des virtuellen Speichers etabliert. Dabei sieht der Programmierer nur einen großen zusammenhängenden Speicherbereich und muss keine Rücksicht auf die Größe des Hauptspeichers nehmen. Sollte ein Programm mehr Hauptspeicher benötigen als physisch in dem System vorhanden ist, dann wird das Betriebssystem die Daten auf ein anderes Speichermedium auslagern. Diese Technik wird Paging genannt und muss vom Betriebssystem unterstützt werden.

5.0.1 Paging

Ein Beispiel: Ein Programmierer hat ein Programm geschrieben welches 4 MiByte (4096 KiByte) Arbeitsspeicher benötigt. Das System auf dem das Programm ausgeführt werden soll besitzt allerdings nur 1 MiByte (1024 KiByte) physischen Hauptspeicher. Wenn das Betriebssystem Paging unterstützt wird es folgendermaßen vorgehen. Das Betriebssystem teilt den benötigten Adressraum von 4 MiByte in vier gleich große Teile (jeweils

1024 KiByte) ein, welche einzeln in den Hauptspeicher geladen werden können. Diese Speicherfenster nennt man "Page Frames". Zum Start des Programmes wird eines der vier Pageframes in den Hauptspeicher geladen und der Rest auf einem Massenspeicher gespeichert. Die Seiten werden vom Betriebssystem in einer Seitentabelle ("Page-Table") verwaltet.

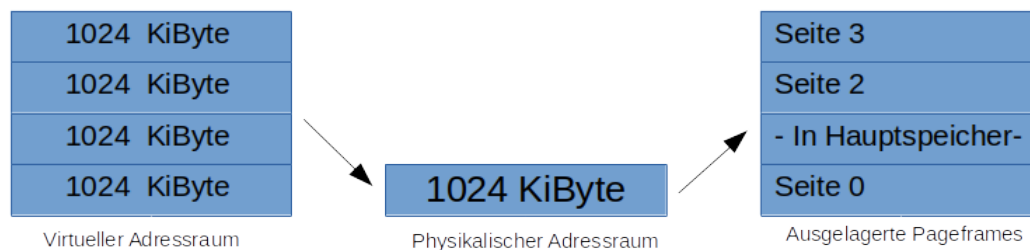


Figure 2: Paging

Fordert das Programm nun eine virtuelle Adresse an welche nicht im physikalischen Speicher liegt wird die Ausnahme Seitenfehler ("Pagefault Exception") ausgelöst. Wie in Kapitel ?? ausgeführt wird bei einer Exception eine Behandlungsroutine angestoßen. Im Falle der Seitenfehler Ausnahme wird die Routine die Seite ("Page Frame"), welche die benötigte Adresse enthält, aus dem Massenspeicher in den Hauptspeicher laden. Paging ist ein kompliziertes Verfahren um das Zusammenspiel von virtuellen und physischen Speicher zu verwalten. Dieses Konzept in Software zu realisieren würde wie die Overlays zu viel Leistung kosten. Deshalb besitzen die meisten modernen Prozessoren einen Controller für solche Speicherverwaltungsaufgaben, die so genannte MMU ("Memory Management Unit").

5.0.2 Speichersegmentierung

5.1 Virtueller Speicher

5.2 Caches

Moderne Prozessoren wie zum Beispiel der AMD R7 1700x takten mit bis zu 3.60 GHz. Arbeitsspeicher (DDR4) hingegen taktet effektiv mit etwa 1500 MHz. Wenn die CPU Daten aus dem Arbeitsspeicher benötigt, muss sie mitunter mehrere Taktzyklen warten bis der Arbeitsspeicher Antworten liefert.

6 Befehlsausführung

6.1 Befehlszyklus

6.2 Schleifen

Programmschleifen sind für jedes Programm von essentieller Bedeutung und in jeder modernen Programmiersprache implementiert. Ohne sie wäre kein effizienter und kompakter Programmaufbau möglich. In Hochsprachen sind zwei Grundlegende Arten von Schleifen geläufig. Zum einen die Schleifen mit festgelegter Anzahl an Durchläufen:

$$\textit{for}(\textit{int } i = 0; i < 100; i++) \{ \dots \}$$

Diese Schleife wird genau 100 mal durchlaufen bis sie abbricht. Es gibt aber auch Szenarien in den die benötigte Zahl an Durchläufen nicht bekannt ist:

$$\textit{while}(\textit{"Bedingung"}) \{ \dots \}$$

Diese Schleifen werden solange durchlaufen bis die Abbruchbedingung "true" ergibt. Die gezeigten Beispiele entsprechen Schleifen in einer Hochsprache, aber wie werden diese Konstrukte auf der Hardware durch die CPU ausgeführt? Prozessoren durchlaufen besitzen grundlegend einen linearen Programmablauf. Sie holen sich den Befehl

aus der Speicherzelle auf dessen Adresse der Program Counter (PC) zeigt. Diesen Befehl dekodieren sie und führen ihn aus. Daraufhin wird der Wert des PC inkrementiert. Um Schleifen realisieren zu können besitzen Prozessoren spezielle Befehle mit denen sie den Wert des PC ändern können. Diese Befehle werden Sprungbefehle ("Jump") genannt. Der Jump-Befehl lädt den Program Counter mit einer angegebenen Adresse. Das bedeutet, dass der Programmablauf an der neuen Adresse weitergeführt wird. Im Grunde muss bei einer Schleife ein bestimmter Codeabschnitt immer wieder durchlaufen werden. Zur Veranschaulichung wird nun eine einfache Schleife in der Hochsprache C mit dem daraus erzeugten Assemblercode für einen Intel Prozessor verglichen. Kompiliert wurde der Code unter Ubuntu 17.10 mit dem Linux-Kernel 4.13.

```
int main(int argc, char const *argv[])
{
    unsigned int counter = 0;
    while(counter < 100){
        counter++;
    }
    return 0;
}
```

Code Listing 1: C-Code für eine einfache Schleife

Dieses Programm initialisiert eine Variable counter mit dem Wert 0 und inkrementiert diesen in einer Schleife so lange bis er den Wert 100 erreicht. Der Assemblercode für dieses Programm ist folgendermaßen aufgebaut:

Die erste Zeile ("push rbp") und die letzten beiden Zeilen ("pop rbp & ret") sind bei jeder Funktion gleich und dienen dazu, einen Bereich im Stack für das Programm zur Verfügung zu stellen und diesen nach Programmende wieder freizugeben. In Zeile 11 wird im Stack an der Adresse mit Offset 0x4 der Wert 0 gespeichert. Das entspricht der Initialisierung der Variable counter im C-Programm. Danach kommt es zu einem unbedingten Sprung. Der Program Counter wird mit dem angegebenen Wert geladen und zeigt nun auf Adresse 0x612. Im nächsten Ausführungszyklus wird nun das Programm an Adresse 0x612 weiter ausgeführt. Mit diesem Sprung wird genau eine Zeile übersprungen. Diese Zeile (main+20) inkrementiert den Wert im Stack an der Adresse

```

0x000000000000005fa <+0>:    push    rbp
0x000000000000005fb <+1>:    mov     rbp, rsp
0x000000000000005fe <+4>:    mov     DWORD PTR [rbp-0x14], edi
0x00000000000000601 <+7>:    mov     QWORD PTR [rbp-0x20], rsi
0x00000000000000605 <+11>:   mov     DWORD PTR [rbp-0x4], 0x0
0x0000000000000060c <+18>:   jmp     0x612 <main+24>
0x0000000000000060e <+20>:   add     DWORD PTR [rbp-0x4], 0x1
0x00000000000000612 <+24>:   cmp     DWORD PTR [rbp-0x4], 0x63
0x00000000000000616 <+28>:   jbe     0x60e <main+20>
0x00000000000000618 <+30>:   mov     eax, 0x0
0x0000000000000061d <+35>:   pop     rbp
0x0000000000000061e <+36>:   ret

```

Code Listing 2: Assembler-Code für eine einfache Schleife

mit Offset 0x4, welche die Variable counter enthält. Diese Zeile wird zum Beginn des Programmes genau einmal übersprungen, da die Abbruchbedingung beim ersten Durchlauf noch nicht kontrolliert wurde. Das Programm wird also in Zeile 24 weiter ausgeführt. Der Befehl `cmp` ("Compare") vergleicht nun den Wert der Variablen counter ("rbp-0x4") mit der Konstante 0x63 (99d). Da `cmp` eine ALU Operation ist wird dieser Befehl die entsprechenden Flags im Statusregister setzen, je nachdem ob "counter" kleiner oder gleich 0x63 ist. Wenn der Befehl abgearbeitet ist wird die Zeile 28 geladen. Der Befehl `jbe` ("Jump if below or equal") ist ein bedingter Sprung. Das heißt es wird nur gesprungen wenn die Bedingung erfüllt ist. Daraufhin wird das Statusregister geprüft ob das Carry-Flag (CF) oder das Zero-Flag (ZF) gesetzt ist. Wenn die ODER-Verknüpfung dieser beiden Werte 1 ergibt, wird der PC mit der angegebenen Adresse geladen, wenn sie eine 0 ergibt, wird mit dem nächsten Befehl weitergearbeitet. Die Zeile 28 entspricht also der Abbruchprüfung der Schleife (*counter* < 100). Wenn der Wert kleiner 100 ist wird in Zeile 20 gesprungen und damit die Variable inkrementiert, wenn der Wert größer oder gleich 100 ist wird das Programm in Zeile 30 weitergeführt. Da die Schleife dann beendet ist wird das Programm beendet.

6.3 Subroutinen

Subroutinen sind neben den Schleifen eine der wichtigsten und elementarsten Kontrollstrukturen in der Softwareentwicklung. Den meisten Programmierern sind Subroutinen eher als Funktionen bekannt. Sie sind wie die Schleifen in jeder modernen Programmiersprache enthalten. Funktionen ermöglichen es Programmcode übersichtlich zu gestalten und zu kapseln. Wenn ein bestimmter Algorithmus häufiger ausgeführt werden soll kann er in eine Funktion ausgelagert werden. Dann muss der gesamte Algorithmus nicht mehrfach implementiert werden sondern kann als Funktion aufgerufen werden. Der Prozessor hat für die Verzweigung des Programmflusses einen eigenen Befehl, die sogenannte Call-Anweisung. Nach einem Call Befehl wird eine Adresse angegeben und mit dieser Adresse wird der Program Counter geladen. Zur Veranschaulichung wieder ein kleines Beispiel in der Sprache C.

```
int add(int a, int b){
    return a+b;
}

int main(int argc, char const *argv[])
{
    int a = 3;
    int b = 5;
    int c = add(a, b);
    return 0;
}
```

Code Listing 3: C-Code mit Funktionsaufruf

Dieses kleine Programm initialisiert zwei Variablen, *a* und *b*, mit *int* Werten. Eine dritte Variable erhält den Rückgabewert der Funktion *add*. Der Assemblercode für dieses Programm ist folgender:

In der *main* Methode werden in Zeilen *main*+15 und *main*+22 die beiden Variablen *a* und *b* auf dem Stack initialisiert. In Zeile *main*+35 wird der *call*-Befehl vom Prozessor ausgeführt. *Call* ist in seiner grundlegenden Funktion dem Befehl *JMP* ähnlich. Beides sind unbedingte Sprungbefehle, das heißt sie laden die übergebene Adresse ohne weitere Prüfung in den PC. Der Unterschied zwischen den beiden besteht darin, dass das Pro-

Function main			
0x0000000000000060e	<+0>:	push	rbp
0x0000000000000060f	<+1>:	mov	rbp, rsp
0x00000000000000612	<+4>:	sub	rsp, 0x20
0x00000000000000616	<+8>:	mov	DWORD PTR [rbp-0x14], edi
0x00000000000000619	<+11>:	mov	QWORD PTR [rbp-0x20], rsi
0x0000000000000061d	<+15>:	mov	DWORD PTR [rbp-0xc], 0x3
0x00000000000000624	<+22>:	mov	DWORD PTR [rbp-0x8], 0x5
0x0000000000000062b	<+29>:	mov	edx, DWORD PTR [rbp-0x8]
0x0000000000000062e	<+32>:	mov	eax, DWORD PTR [rbp-0xc]
0x00000000000000631	<+35>:	mov	esi, edx
0x00000000000000633	<+37>:	mov	edi, eax
0x00000000000000635	<+39>:	call	0x5fa <add>
0x0000000000000063a	<+44>:	mov	DWORD PTR [rbp-0x4], eax
0x0000000000000063d	<+47>:	mov	eax, 0x0
0x00000000000000642	<+52>:	leave	
0x00000000000000643	<+53>:	ret	
Function Add			
0x000000000000005fa	<+0>:	push	rbp
0x000000000000005fb	<+1>:	mov	rbp, rsp
0x000000000000005fe	<+4>:	mov	DWORD PTR [rbp-0x4], edi
0x00000000000000601	<+7>:	mov	DWORD PTR [rbp-0x8], esi
0x00000000000000604	<+10>:	mov	edx, DWORD PTR [rbp-0x4]
0x00000000000000607	<+13>:	mov	eax, DWORD PTR [rbp-0x8]
0x0000000000000060a	<+16>:	add	eax, edx
0x0000000000000060c	<+18>:	pop	rbp
0x0000000000000060d	<+19>:	ret	

Code Listing 4: Assembler-Code mit Funktionsaufruf

programm nach dem Beenden einer Subroutine wieder an seinen Aufrufort im Hauptprogramm zurückspringt. Um an die richtige Stelle im Hauptprogramm zurückzuspringen wird der call-Befehl deshalb den Wert der Rücksprungadresse auf dem Stack speichern. Nach dem Aufruf von call wird nach Adresse 0x5fa gesprungen, wo sich der Startpunkt der Funktion *add* befindet. Die Funktion hat nur die Funktion die beiden übergebenen Werte zu addieren und das Ergebnis zurückzugeben. In Zeile add+16 werden die beiden Register mit den Variablen addiert. Daraufhin wird das Programm beendet. Hier wird ein interessantes Detail über Funktionsaufrufe in Assembler ersichtlich. In

Hochsprachen ist es ein Entwickler gewohnt, dass eine Funktion Übergabeparameter und Rückgabeparameter besitzt. Diese Übergaben sind aber in Assembler nicht explizit möglich. Die Parameter werden deshalb in Registern gespeichert welche vom Compiler verwaltet werden müssen. In den Zeilen 35 und 37 der main-Funktion werden die beiden Variablen *a* und *b* in den Registern *esi* und *edi* gespeichert. Diese beiden Register dienen nun als Übergabeparameter für die Funktion *add*. In der Funktion werden die Variablen dann erst auf dem Stack gespeichert und dann in die Register *edx* und *eax* verschoben. Diese werden daraufhin addiert und das Ergebnis wird in *eax* gespeichert. Als letzter Schritt wird die Funktion mit dem Befehl *ret* (Zeile *add*+19) beendet. Dieser Befehl lädt den Program Counter wieder mit dem Wert der Rücksprungadresse welche vom *call*-Befehl auf dem Stack gespeichert wurde.

7 Besondere Ausführungsarten

7.1 Interrupts

Das Konzept von Interrupts ist für moderne Prozessoren von enormer Bedeutung, weshalb es in so gut wie jedem modernen Prozessor implementiert ist. Bisher wurde in dieser Arbeit nur betrachtet wie ein Prozessor intern logisch funktioniert. Allerdings muss dieser auch mit zahlreichen anderen Bausteinen auf einer Platine zusammenarbeiten um sinnvoll eingesetzt werden zu können. Eine CPU in einem Desktop PC muss beispielsweise auf Tastatureingaben und Mausbewegungen reagieren, oder auf eine Netzwerkschnittstelle welche Daten empfängt. Solche Service-Anforderungen sind von hoher Priorität, da sonst kein reibungsloser Ablauf gewährleistet werden kann. Auswirkungen können zum Beispiel sein, dass die Maus nicht auf Bewegung reagiert oder die Netzwerkschnittstelle keine Daten verarbeiten kann. Ein Problem bei solchen Anforderungen an die CPU ist, dass sie nicht vorhersehbar sind, also asynchron auftreten. Eine Mausbewegung kann zu jeder Zeit kommen genau so wie der Datenempfang an einer Netzwerkschnittstelle.

Ein mögliches Konzept um auf diese Anforderungen zu reagieren sind so genannte Polling-Schleifen. Wenn z.B ein Datenstrom über das Netzwerk übertragen werden

soll, dann besitzt die CPU intern ein Statusregister, welches anzeigt, ob die Netzwerkschnittstelle bereit ist das nächste Paket zu verschicken. Zeigt das Register an, dass es bereit ist schickt die CPU das nächste Paket zu der Netzwerkschnittstelle. Da die externe Schnittstelle um ein vielfaches langsamer taktet als der Prozessor werden viele Abfragezyklen an das Statusregister verschwendet. Dieses Verfahren ist sehr ineffizient und wird deshalb so gut wie nie verwendet.

Um Service-Anforderungen effizienter durch den Prozessor bearbeiten zu können hat sich ein Verfahren namens Interruptkonzept etabliert. Ein Interrupt ("Unterbrechung") ist ein Signal, welches externe Schnittstellen an den Prozessor senden können wenn sie einen Service anfordern wollen. Im Gegensatz zu Polling-Schleifen muss der Prozessor nicht ständig den Zustand der Schnittstellen abfragen sondern kann seinen internen Programmfluss ungestört abarbeiten bis ein Interrupt angefordert wird. Der Prozessor benötigt für ein solches Verfahren einen separaten Interrupt-Eingang. Wenn zum Beispiel eine Taste auf der Tastatur gedrückt wurde schickt das Betriebssystem einen Interrupt vom Tastatur-Controller an die CPU. Der Prozessor wird daraufhin seinen Programmfluss unterbrechen und in eine spezielle Interrupt Behandlungsroutine springen.

Im Unterschied zu Subroutinen können Interrupts jederzeit auftreten, sie sind deshalb asynchron. Das ist problematisch, da keinerlei Vorbereitungen auf die Verzweigung im Programmfluss getroffen werden können. Die ISR ("Interrupt Service Routine"), also der Programmcode welcher bei einem bestimmten Interrupt ausgeführt wird, muss aber Register- und Flagwerte verändern. Um einen reibungslosen Übergang in das unterbrochene Programm zu gewährleisten, sichern die ISR die Register- und Flagwerte zu Beginn ihrer Ausführung auf dem Stack. Sobald der Interrupt abgearbeitet ist, werden die Werte aus dem Stack wieder geladen und das Unterprogramm wird beendet. Somit kann das unterbrochene Hauptprogramm sofort weiter ausgeführt werden.

Ein Prozessor ist an mehrere externe Schnittstellen angebunden welche Interrupts auslösen können. Um die Priorität der Interrupts zu verwalten und den Prozessor zu entlasten besitzen die meisten Interruptfähigen Prozessoren einen Interrupt-Controller. Der Aufbau und die Funktionsweise des Controllers wird im Folgenden nur oberflächlich beschrieben. Die Vektorisierung, Kaskadierung und das Daisy-Chaining von Interrupts

wird hier nicht behandelt. Der Interrupt-Controller besitzt intern mehrere Register zur Verwaltung und Priorisierung von Interrupts. Das Interrupt-Masken-Register (IMR) ist dafür zuständig, die einzelnen Interrupteingänge für Interrupts freizuschalten oder zu sperren. Nur wenn im IMR das Interrupt-Freigabe-Bit für den jeweiligen Interrupt gesetzt ist wird der Interrupt weiterverarbeitet. Das Interrupt-Request-Register (IRR) enthält die Interrupts, welche vom IMR freigeschaltet und von den externen Interruptquellen angefordert wurden. Vereinfacht kann man sich die Werte des IRR als UND-Verknüpfung zwischen den Werten der Interrupteingänge und den Werten des IMR vorstellen. Da es möglich ist, dass mehrere Interrupts gleichzeitig auftreten muss der Interrupt-Controller entscheiden welche Unterbrechung Vorrang hat. Die Entscheidung trifft der Controller auf der Basis der Werte des Prioritätenregisters. In ihm ist die Reihenfolge der Interruptabarbeitung hinterlegt. Das Prioritätenschaltnetz (PSN) wird nun die Reihenfolge der Abarbeitung bestimmen indem es die Werte des IRR und des Prioritätenregisters beschreibt. Die als erstes abzuarbeitende Unterbrechung wird in das Interrupt-Service-Register (ISR) geschrieben. Das ISR zeigt also an welcher Interrupt zur Verarbeitung an den Prozessor weitergegeben wird.

Nahezu jeder Prozessor besitzt heutzutage einen Interruptcontroller und alle externen Schnittstellen sind interruptfähig. Dieses Konzept ist sehr effizient und die Probleme die es mitbringt werden durch den Interruptcontroller eliminiert.

7.2 Exceptions

Im Gegensatz zu den Interrupts, deren Ursprung außerhalb des Prozessors liegt, werden Ausnahmen ("Exceptions") vom Prozessor selbst ausgelöst.

8 Leistungsaspekte eines Prozessors

8.1 Pipelines und Skalarität

8.2 Mehrkern-Prozessoren

9 Planung und Entwurf eines Prozessors

Der Inhalt der bisherigen Arbeit handelte von den Komponenten einer CPU und deren Funktionsweisen. 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 Funktionsweise 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 nutzen. Zu Erklärungszwecken werden die 32-Bit wie folgt aufgeteilt:

Table 6: 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:

Table 7: 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.

Table 8: 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 gesamten 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.

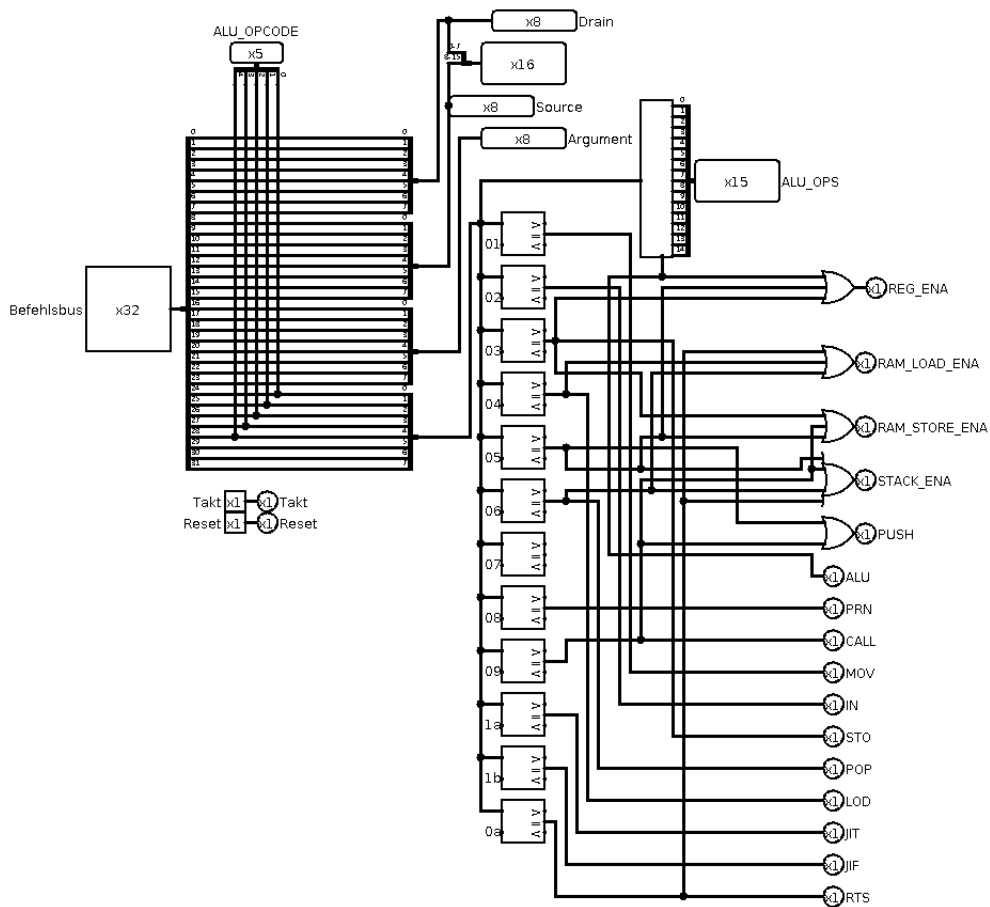


Figure 3: Darstellung des Steuerwerks

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 5: C++ Code Primzahlen zählen

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.

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 `0x00000000040089c` (`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 Rückgabebetyp `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 Beispielsweise eine -2 übergeben wird und vom Befehl `cmpl` 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

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  0(%rax,%rdi)
0x00000000004008be <+96>:   retq

```

End of assembler dump.

s

Code Listing 6: Assemblercode der main-Methode

```

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 7: Assemblercode der checkIfPrime-Methode

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 Abbruchbedingung bereits erfüllt ist. In der 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.

References

- [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.