UMach Spezifikation

2. Mai 2012

Inhaltsverzeichnis

1 Einführung

UMach ist eine einfache virtuelle Maschine (VM), die einen definierten Instruktionssatz und eine definierte Architektur hat. UMach orientiert sich dabei an Prinzipien von RISC Architekturen: feste Instruktionslänge, kleine Anzahl von einfachen Befehlen, Speicherzugriff durch Load- und Store-Befehlen, usw. Die UMach Maschine ist Register-basiert. Der genaue Aufbau dieser Rechenmaschine ist im Abschnitt 2.1 ab der Seite 7 beschrieben.

Für den Anwender der virtuellen Maschine wird zuerst eine Assembler-Sprache zur Verfügung gestellt. In dieser Sprache werden Programme geschrieben und anschließend kompiliert. Die kompilierte Datei (Maschinen-Code) wird von der virtuellen Maschine ausgeführt.

Obwohl in diesem Dokument Namen von Assembler-Befehlen angegeben werden (siehe Kapitel 3, Instruktionssatz) spezifiziert dieses Dokument die UMach Maschine auf Maschinencode Ebene (Register, Bussystem, Instruktionen). Die Implementierung eines Assemblers ist frei, zusätzliche Befehle, Instruktionsformate, Aliase und sprachliche Konstrukte auf der Assembler-Ebene zu definieren. Z.B. auf Maschinencode-Ebene, sind die Befehle

```
ADD R1 R2 5
SUB R2 4
```

fehlerhaft, denn das Format der `ADD` und `SUB` Befehle verlangt die Angabe von drei Registern. Der Assembler ist frei, diese zusätzliche Formate zu definieren und zu erkennen, solange er gültigen UMach Maschinencode produziert. Gültiger Maschinencode für die obigen Befehle wäre

```
ADDI R1 R2 5 # Maschinencode 0x52 0x01 0x02 0x05
SUBI R2 R2 4 # Maschinencode 0x5A 0x02 0x02 0x04
```

2 Organisation der UMach VM

2.1 Aufbau

Die virtuelle Maschine besteht aus einem Kern (der eigentlichen UMach Maschine) und aus einem Bussystem. Das Bussystem wird im Abschnitt 2.4 ab der Seite 15 beschrieben.

Der Kern der Maschine besteht aus den folgenden Komponenten:

1. Befehlsabruf Einheit (Instruction Fetch Unit)
2. Decodierungseinheit
3. Recheneinheit
4. Register

Die Decodierungseinheit ist dafür zuständig, eine abgerufene Instruktion zu decodieren, bzw. in ihren Komponenten zu zerteilen.

Die Recheneinheit ist für die tatsächliche Ausführung der Instruktionen zuständig.

Die Register sind die Speichereinheiten, die sich in der Maschine befinden.

2.1.1 Betriebsmodi

Ein Betriebsmodus bezieht sich auf die Art, wie die UMach VM läuft. Die UMach VM kann in einem der folgenden Betriebsmodi laufen:

1. Normalmodus
2. Einzelschrittmodus

Der standard-Modus ist der Normalmodus. Zur Jeder Zeit kann das Betriebsmodus geändert werden. Siehe die Instruktion `CRM`.

Normalmodus Die virtuelle Maschine führt ohne Unterbrechung ein Programm aus. Nach der Ausführung befindet sich die Maschine in einem Wartezustand, falls sie nicht ausdrücklich ausgeschaltet wird.

Einzelschrittmodus Die virtuelle Maschine führt immer eine einzige Instruktion aus und nach der Ausführung wartet sie auf einen externen Signal um mit der nächsten Instruktion fortzufahren. Dieser Modus soll dem Entwickler erlauben, ein Programm schrittweise zu debuggen.

2.1.2 Von Neumann Zyklus

Der Von Neumann Zyklus der UMach ist auf 4 Schritte verkürzt. Dies ist möglich, da die Instruktionsbreite immer aus 4 Wörtern besteht, und somit der „Fetch“ von Befehl und zugehörigen Operanden in einem gemeinsamen Schritt durchgeführt werden kann. Somit besteht der Zyklus aus einem beginnenden Fetch. Bei diesem wird der an der im Programmcounter `PC` liegende Adresse gespeicherte Befehl in das Instruktionsregister `IR` geladen. Danach wird der im ersten Byte liegende Befehl mit Hilfe des Befehlsdecoders decodiert und an der ALU entsprechend eingestellt.

In einem dritten Schritt Execute wird die Instruktion ausgeführt. Der theoretisch 4. Schritt, das Inkrementieren des Programmcounters PC, wird parallel zu den Fetch und Decode Vorgängen ausgeführt. Diese Tatsache ist bei Instruktionen, welche den Inhalt des PC manipulieren, zu berücksichtigen. Siehe auch Abbildung 2.1 auf der Seite 9.

Auflistung der Schritte:

1. Fetch – Holen der Instruktion aus dem Speicher von der Adresse, welche im PC vorliegt.

2. Decode – Decodieren des Befehles und Einstellen der ALU.

3. Execute – Auführen des Befehles in der ALU.

4. Update `PC` – Inkrementieren des `PC`. Findet parallel zu Fetch und Decode statt.

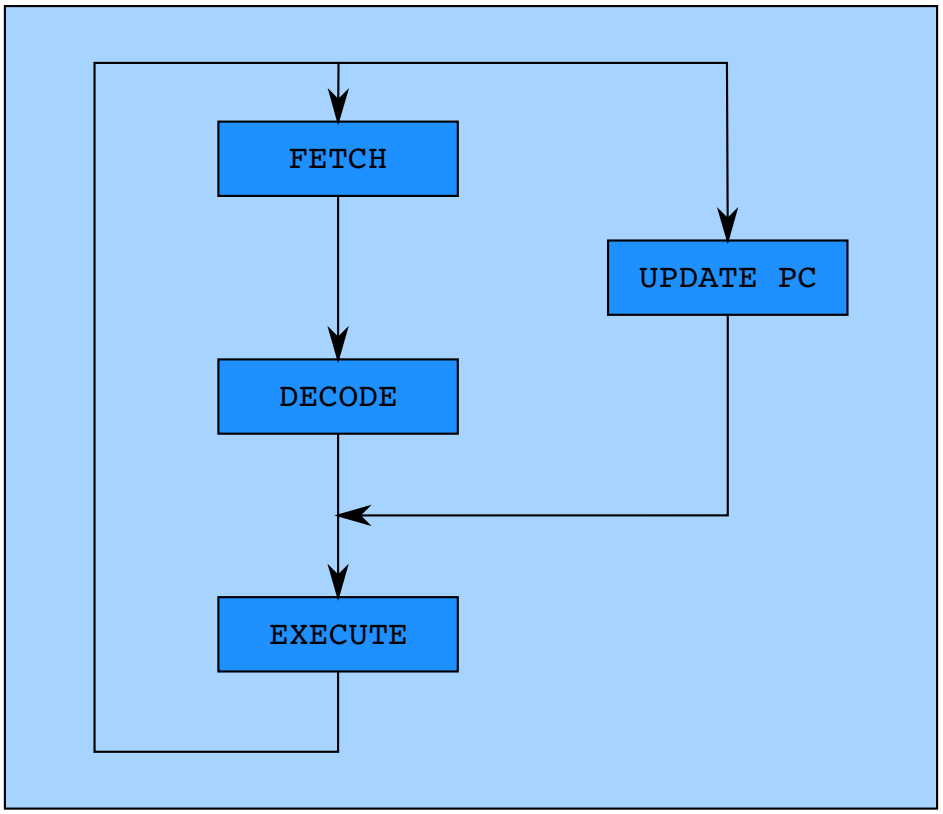

Abbildung 2.1: Von Neumann Zyklus

2.2 Register

Die Register sind die Speichereinheiten im Prozessor. Die meisten Anweisungen an die UMach Maschine operieren auf einer Art mit den Registern.

Für alle Register gilt:

1. Jedes Register ist ein Element aus der Menge $\mathcal{R}$, die alle Register beinhaltet. Die Notation $x \in \mathcal{R}$ bedeutet, dass x ein Register ist.

2. Die Speicherkapazität beträgt 32 Bit.

3. Es gibt eine eindeutige Maschinenzahl, die innerhalb der Maschine das Register identifiziert. Diese Zahl heißt Maschinenname und wird von einer Instruktion auf Maschinencode-Ebene verwendet, wenn sie das Register anspricht.

4. Die UMach Maschine erwartet die Angabe eines Registers als numerischer Wert (Maschinenname). Jedoch verwendet der Programmierer der Maschine auf Assembler Ebene einen eindeutigen Namen dieses Registers. Dieser Name heißt Assemblername.

Die UMach Maschine hat zwei Gruppen von Registern: die Allzweckregister und die Spezialregister.

2.2.1 Allzweckregister

Es gibt 32 Allzweckregister, die dem Programmierer für allgemeine Zwecke zur Verfügung stehen. Diese 32 Register werden beim Hochfahren der Maschine auf Null (`0x00`) gesetzt. Außer dieser Initialisierung, verändert die Maschine den Inhalt der Allzweckregister nur auf explizite Anfrage, bzw. infolge einer Instruktion.

Die 32 Register werden auf Maschinencode-Ebene von 1 bis 32 nummeriert (`0x01` bis einschliesslich `0x20` im Hexadezimalsystem). Diese Nummer ist der Maschinenname des Registers. Auf Assembler-Ebene, werden sie mit den Namen $R1, R2 \ldots$ bis $R32$ angesprochen (Assemblername). Die Zahl nach dem Buchstaben R ist im Dezimalsystem angegeben und ist fester Bestandteil des Registernamens.

Assemblername	`R1`	`R2`	`R3`	`...`	`R32`
Maschinenname	`0x01`	`0x01`	`0x03`	`...`	`0x20`

Register mit Nummer Null ($R0$) gibt es nicht und die Verwendung des Null-Registers wird von der Maschine als Fehler gemeldet.

Beispiel für die Verwendung von Registernamen:

Assembler	`ADD`	`R1`	`R2`	`R3`
Maschinencode	`0x50`	`0x01`	`0x02`	`0x03`
Bytes	erstes Byte	zweites Byte	drittes Byte	viertes Byte
Algebraisch	$R_1 \leftarrow R_2 + R_3$			

2.2.2 Spezialregister

Die Spezialregister werden von der UMach Maschine für spezielle Zwecke verwendet, sind aber dem Programmierer sichtbar. Der Inhalt der Spezialregister kann von der Maschine während der Ausführung eines Programms ohne Einfluss seitens Programmierers verändert werden.

Nicht alle Spezialregister können durch Instruktionen überschrieben werden (sind schreibgeschützt).

Die Maschinennamen der Spezialregister setzen die Nummerierung der Allzweckregister zwar fort, die Assemblernamen aber nicht: es gibt kein Register $R33$. Die Tabelle 2.1 auf Seite 11 enthält die Liste aller Spezialregister. In der ersten Spalte steht der Assemblername, so wie er vom Programmierer verwendet wird. In der zweiten Spalte steht der Maschinenname im Hexadezimalsystem, so wie er im Maschinencode steht. Die dritte Spalte enthält eine kurze Beschreibung und Bemerkungen. Falls die Beschreibung nicht spezifiziert, dass ein Register schreibgeschützt ist, ist das Register nicht schreibgeschützt.

Tabelle 2.1: Liste der Spezialregister

`PC`	`0x33`	„Instruction Pointer“. Enthält zu jeder Zeit die Adresse der nächsten Instruktion. Wird auf Null gesetzt, wenn die Maschine hochfährt. Wird nach dem Abfangen einer Instruktion in das Register `IR` automatisch inkrementiert. Schreibgeschützt.
`SP`	`0x34`	„Stack Pointer“. Enthält die Speicheradresse des höchsten Eintrag auf dem Stack. Wird beim Hochfahren der Maschine auf die maximal erreichbare Adresse im Speicher gesetzt. Siehe auch 2.3.3, Seite 14.
`FP`	`0x35`	„Frame Pointer“. Enthält die Startadresse der lokalen Variablen einer Subroutine und unterstützt höhere Programmiersprachen.

LIN	0x36	„Last Instruction“. Enthält den Maschinencode der zuletzt ausgeführten Instruktion. Schreibgeschützt.
IR	0x37	„Instruction Register“. Enthält die gerade ausgeführte Instruktion. Schreibgeschützt.
NIN	0x38	„Next Instruction“. Enthält die nächste Instruktion. Während der Ausführung eines Instruktion, zeigt der Register PC auf diese Instruktion, aber im Speicher. Schreibgeschützt.
STAT	0x39	Enthält Status-Informationen
ERR	0x3A	„Error“. Fehlerregister. Die einzelnen Bits geben Auskunft über Fehler, die mit der Ausführung des Programms verbunden sind. Liegt kein Fehler vor, so ist der Inhalt dieses Registers gleich Null.
ERRM	0x3B	„Error message“. Enthält zusätzliche Informationen zu dem Fehler signalisiert im Register ERR.
HI	0x3C	„High“. Falls eine Multiplikation durchgeführt wird, enthält dieses Register die höchstwertigen 32 Bits des Ergebnisses der Multiplikation und bildet zusammen mit dem Register LO das volle Ergebnis der Multiplikation. Falls eine Division durchgeführt wird, enthält dieses Register den Quotient der Division.
LO	0x3D	„Low“. Falls eine Multiplikation durchgeführt wird, enthält dieses Register die niedrigstwertigen 32 Bits des Ergebnisses der Multiplikation und bildet zusammen mit dem Register HI das volle Ergebnis der Multiplikation. Falls eine Division durchgeführt wird, enthält dieses Register den Rest der Division.
CMP	0x3E	„Compare Register“. Enthält das Ergebnis eines Vergleichs. Siehe auch Abschnitt 3.7, Seite 64.
ZERO	0x3F	Enthält die Zahl Null. Schreibgeschützt.

2.3 Der Speicher

2.3.1 Adressierungsarten

Als RISC-orientierte Maschine, greift die UMach lediglich in zwei Situationen auf den Speicher zu: zum Schreiben von Registerinhalten in den Speicher (Schreibzugriff) und zum Lesen von Speicherinhalten in einen Register (Lesezugriff). Die Adressierungsart beschreibt dabei, wie der Zugriff auf den Speicher erfolgen sollte, bzw. wie die angesprochene

Speicheradresse angegeben wird. Anders ausgedrückt, beantwortet die Adressierungsart die Frage „wie kann eine Instruktion der Maschine eine Adresse angeben?“.

Die UMach Maschine kennt eine einzige Adressierungsart: die indirekte Adressierung, die unten beschrieben wird. Die direkte Adressierung, die aus einer direkten Angabe eines Speicheradresse besteht, wird von der indirekten Adressierung überdeckt.

Indirekte Adressierung

Die indirekte Adressierung verwendet zwei Register B und I, die von der Maschine verwendet werden, um die endgültige Adresse zu berechnen: Eine Instruktion, die diese Adressierung verwendet, hat also das Format RRR (siehe auch 3.1.7).

erstes Byte	zweites Byte	drittes Byte	viertes Byte	Algebraisch
Ladebefehl	R	B	I	$R \leftarrow mem(B + I)$
Speicherbefehl	R	B	I	$R \rightarrow mem(B + I)$

Die fünfte Spalte gibt jeweils den äquivalenten algebraischen Ausdruck wieder. $mem(x)$ steht dabei für den Inhalt der Adresse x.

Die zweite Zeile (Ladebefehl) bedeutet, dass die UMach Maschine die Inhalte der Register B und I aufaddieren soll, diese Summe als Adresse im Speicher zu verwenden und den Inhalt an dieser Adresse in den Register R zu kopieren.

Die dritte Zeile (Speicherbefehl) bedeutet: die Maschine soll den Inhalt des Registers R an die Adresse $B + I$ schreiben.

Üblicherweise enthält B eine Startadresse und I einen Versatz oder Index zur Adresse in B.

Vorteil der indirekten Adressierung ist, dass sie $2^{33} - 1$ mögliche Adressen ansprechen kann. Nachteil ist, dass zwei oder mehrere Instruktionen gebraucht werden, um diese Adressierung zu verwenden, denn die Register B und I erst entsprechend geladen werden müssen.

Die Register R, B und I stehen für beliebige Register.

Name	Bits	Vorzeichenbehaftet		Vorzeichenlos	
		Min	Max	Min	Max
Byte	8	-128	127	0	255
Half	16	-2^{15}	$2^{15}-1$	0	$2^{16}-1$
Word	32	-2^{31}	$2^{31}-1$	0	$2^{32}-1$

Tabelle 2.2: Wertebereiche der UMach Datentypen

2.3.2 Datentypen

Ein Datentyp im Kontext der UMach Maschine ist ein fester Wertebereich, der von der Maschine in einer einzigen Instruktion angesprochen werden kann.

Die UMach Maschine kennt 3 Datentypen, die sich lediglich in der Anzahl der verwendeten Bytes unterscheiden. Die Tabelle 2.2 auf der Seite 14 gibt eine Übersicht der Wertebereiche der einzelnen Datentypen.

Die Einteilung dieser Datentypen wirkt sich auf die Gestaltung mancher Maschinen-Instruktionen, die mit verschiedenen Datentypen arbeiten. Z.B. `PUSHB` arbeitet mit dem Datentyp „Byte“, `PUSHH` arbeitet mit dem Datentyp „Half“.

2.3.3 Der Stack

Der Stack ist ein spezieller Bereich im Speicher. Mit „Speicher“ wird hiermit der Speicher-Adressraum, der am Speicher-Port auf dem Bussystem angeschlossen ist gemeint (nicht also der gesamte Speicherraum innerhalb des Busses).

Dieser Bereich fängt am Ende des Speichers mit der größten Adresse an und erstreckt sich bis zur derjenige Adresse, die im Register `SP` gespeichert ist. Die Stack-Größe ist damit dynamisch, denn das Register `SP` wird sowohl durch die Instruktionen `PUSH` und `POP`, als auch direkt vom Programmierer geändert. Das Wachsen des Stacks bedeutet, dass das Register `SP` immer kleinere Werte annimmt. Das Schrumpfen des Stacks bedeutet, dass `SP` immer größere Werte annimmt. Wird versucht, den Inhalt von `SP` kleiner Null oder größer als die maximale Speicheradresse zu setzen, so wird dies von der Maschine verweigert und als Fehler im Register `ERR` signalisiert.

Beim Hochfahren der Maschine, wird das Register `SP` auf die maximal-erreichbare Speicheradresse gesetzt (Stack-Größe ist ein Byte).

2.4 Die Peripherie

Die UMach Maschine ist an einem Bussystem angeschlossen. Dieses Bussystem enthält unter anderem Baukomponenten (Ports), an denen peripherische Geräte physisch angeschlossen sein können. Der Speicher der Maschine ist eine solche peripherische Komponente, die an einem Port angeschlossen ist (am Speicherport). Eine andere peripherische Komponente ist das Display, das an einem Video-Port des Busses angeschlossen ist. Eine weitere Komponente ist die Tastatur. Wie diese Geräte intern funktionieren, weiß die Maschine und das Bussystem nicht. Was sie „wissen“, ist dass diese Geräte eine Reihe von Befehlen ausführen können (hauptsächlich „read/write“ oder „load/store“ Befehle). Siehe dazu die Abbildung 2.2 auf der Seite 16.

2.4.1 Speicherbasierte Kommunikation

Die UMach Maschine kommuniziert nicht direkt mit den am Bus angeschlossenen peripherischen Geräten, sondern sie verwendet dafür eine Art speicherbasierter Kommunikation, die eine vereinfachte Form von „Memory Mapped I/O“ darstellt. Wie dieses Verfahren funktioniert wird im folgenden beschrieben.

Die Bus-Ports sind im Bussystem physisch fest eingebaut[1]. Deren Anzahl und physikalische Eigenschaften sind zu jeder Zeit dem Bussystem bekannt. Zu diesen physikalischen Eigenschaften zählt die Fähigkeit, bestimmte Befehle aufzunehmen und auszuführen (mittels Port-lokalen Register, die Befehle speichern können), sowie die Fähigkeit, Daten und Signale zu produzieren. Diese Eigenschaften erlauben dem Bussystem, den Port als eine Speichereinheit zu betrachten. Zusätzlich wird für jeden Port festgelegt, welche Befehle soll das angeschlossene Gerät ausführen können. Dies ermöglicht die Abwicklungs eines einfachen Abfrage-Protokolls am entsprechenden Port.

Da jeder Port als Speicher adressierbar ist, sieht das Bussystem jeden Port als in einem einheitlichen, kontinuierlichen 32-Bit Speicherraum eingebettet (da die UMach Maschine 32-Bit Adressen verwendet). Dieser gesamte Speicherraum ist nicht physikalischer Natur, denn es gibt ja keinen Speicher, wo die Ports eingebaut sind, sondern virtueller Natur: jedem Port wird vom Bussystem einen festen Adressbereich innerhalb des gesamten Adressraums vergeben und bildet somit einen Speicherunterraum innerhalt des gesamten virtuellen Speicherraums.

Da an jedem Port Geräte mit unterschiedlichen physikalischen Eigenschaften angeschlossen sein können, wie z.B. Speicher mit unterschiedlicher Speicherkapazität, kann im

[1] Wir schließen nicht aus, dass unterschiedliche Realisierungen der UMach Maschine unterschiedliche Ports haben.

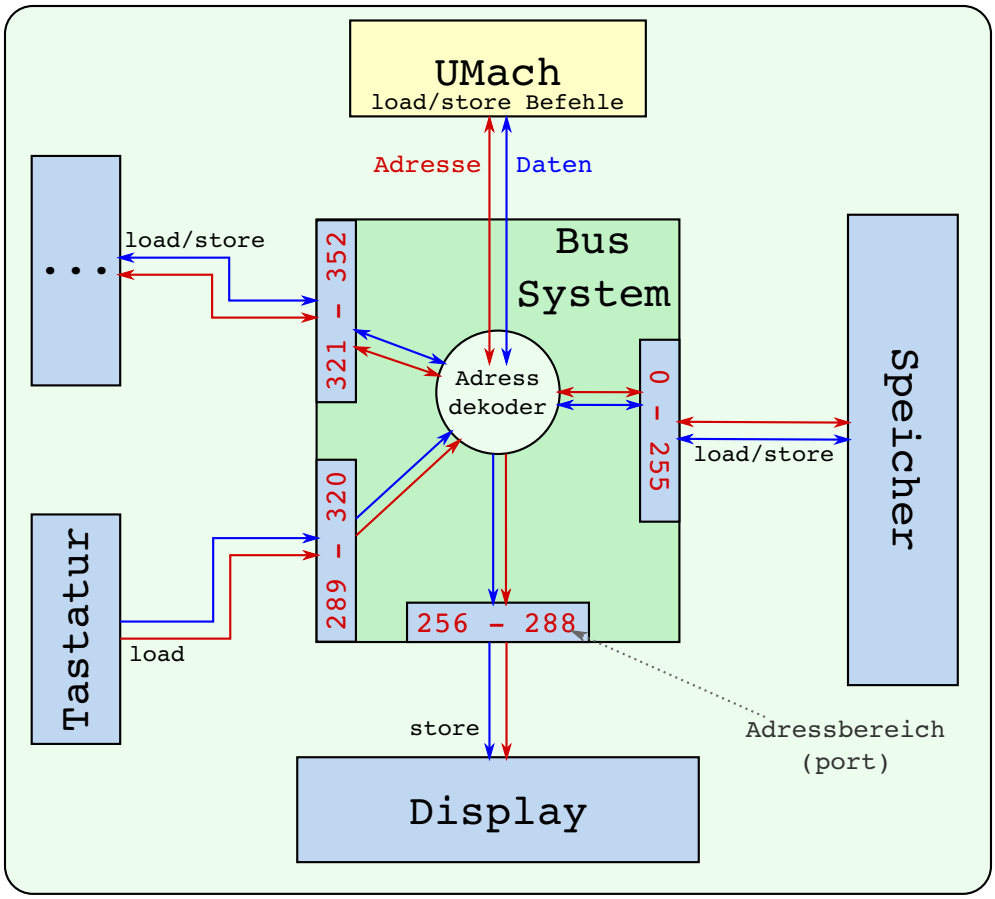

Abbildung 2.2: Das UMach Bussystem verwendet eine einfache speicherbasierte Adressierung der peripherischen Komponenten, die an „Memory Mapped I/O“ angelehnt ist. Jeder peripherischen Komponente wird vom Bussystem beim Hochfahren der Maschine einen eindeutigen Adressbereich zugeweisen. Die „load/store“ Befehle der Maschine werden anhand deren Adressen vom Adressdekoder, der wie ein Demultiplexer funktioniert, an die entsprechende Komponente weitergeleitet. Die Adressbereiche $[0, 255]$, $[256, 288]$ usw. sind nur als Beispiel gegeben. Der interne Speicher des Busses, der die Abbildungen der einzelen Ports auf Adressbereiche enthält, ist nicht dargestellt.

Allgemeinen das Bussystem nicht im Vorraus festlegen, wie groß ist der Adressbereich jedes einzelnen Ports. Diese Größe muss dynamisch vom Bus festgelegt werden.

Beim Hochfahren der gesamten Maschine testet das Bussystem an jedem Port das angeschlossene Gerät. Mittels eines einfachen Protokolls werden die konkrete physikalische Eigenschaften des Geräts abgefragt. Nach der Abfrage und Testphase hat das Bussystem in einem eigenen, internen Speicher eine Liste von Geräten und deren Eigenschaften aufgebaut. Anhand dieser Liste legt das Bussystem fest, wie groß ist der Adressbereich jedes einzelnen Ports. Damit ist das Bussystem in der Lage, eine Adressraumverteilung vorzunehmen, die später für die Adressierung der einzelnen Ports benutzt wird. Siehe dazu auch den Abschnitt 2.4.3 ab der Seite 18. Nach der Verteilung kann das Bussystem jede 32-Bit Adresse einem bestimmten Port zuweisen.

Die Abbildung 2.2 auf der Seite 16 zeigt sehr vereinfacht eine solche Verteilung der Speicherunterräume auf die Adressbereiche innerhalb des gesamten Speicherraums.

Adressierung

Wie bereits oben erklärt, erstellt das Bussystem eine Abbildung der einzelnen Bus-Ports auf Adressbereiche innerhalb eines 32-Bit Adressraums. Der Speicher selbst belegt einen solchen Adressbereich.

Die Kommunikation der UMach Maschine mit den peripherischen Geräten besteht aus Lade- und Speicherbefehle, die von der Maschine dem Bussystem übergeben werden (siehe dazu den Abschnitt 3.4 ab der Seite 31). Das Bussystem fängt diese Befehle in seinem Adressdekoder ab und entscheidet dort anhand der verwendeten Adresse, für welchen Port sind die Befehle gemeint. Wenn das Bussystem den Port identifiziert hat, leitet er den Befehl und die eventuellen Daten an den Port weiter. Siehe als Beispiel die Abbildung 2.2 auf der Seite 16. In diesem Beispiel, möchte die Maschine Daten an das Display senden, so verwendet sie einen Speicherbefehl („store“), der den Adressbereich des Displays verwendet. In der Abbildung 2.2 wäre das eine Adresse im Bereich $[256, 288]$. Um den Buchstaben ’A’ auf dem Display anzuzeigen, sagt die Maschine dem Bussystem „speichere ’A’ an die Adresse 256“. Um von der Tastatur zu lesen, sagt die Maschine dem Bus „lese aus der Adresse 289“.

Um die Adressbereiche bekannt zu machen, muss das Bussystem seinen eigenen internen Speicher in den gesamten Adressbereich einbinden, und zwar an eine feste Adresse (siehe auch Abschnitt 2.4.3 auf der Seite 18). Somit kann der Programmierer der Maschine beim Start des Programms die einzelnen Bereiche abfragen, verwenden und weiter unterteilen.

2.4.2 Bus-Ports

Die folgende Tabelle zeigt die minimale Port-Ausstattung des UMach Bussystems. Zu jedem Port wird eine Typkennung angegeben.

Typ	Port	Bedarf in Bytes
0	Speicher	?
1	Terminal Ausgabe	64
2	Tastatur Eingabe	64

2.4.3 Grundzüge der Adressraumverteilung

Das Bussystem teilt den Adressraum zuerst in zwei große Bereiche: negativen Adressen und positiven Adressen. Die negativen Adressen enthalten die Adressraumverteilung selbst, die eine spezielle Struktur hat. Die positiven Adressen werden an die Ports vergeben. Die Verwendung von negativen Adressen wird damit begründet, dass der Umgang mit Adressen möglichst wenig von den internen Datenstrukturen der Maschine gestört werden sollte. Der Programmierer sollte die Adresse Null oder 512 verwenden können, ohne zu bedenken, dass an diesen Adressen vielleicht Maschinen-Internen Daten vorhanden sind. Gleichzeitig sollten die Maschinen-Informationen offen und auf einfacher Weise zugänglich sein.

Negative Adressen

Der Bereich der negativen Adressen speichert die Verteilung der Ports auf die positiven Adressen. Dieser Bereich beginnt an der Adresse -1 und endet mit der Adresse -2306. Er hat also eine feste Größe von 2306 Bytes[2]. Dieser Speicherbereich hat die folgende Struktur:

- Das Byte an der Adresse -1 speichert die Anzahl der nachfolgenden Einträge und entspricht der Anzahl der eingetragenen Ports. Es sind höchstens 256 Einträge möglich (entspricht 256 mögliche Ports).

- Ab der Adresse -2 werden die Einträge aufgelistet. Jeder Eintrag hat dabei die folgende Struktur:

[2]Die Zahl -2306 ergibt sich aus 256 mögliche Einträge mal 9 Byte plus 1.

$-2306 \leftarrow \ldots$	1 Byte	4 Byte	4 Byte	$\ldots \rightarrow -1$
	Typ	Länge	Startadresse	

Jeder solcher Eintrag belegt 9 Bytes und speichert die Startadresse des Ports, die Länge des Adressbereichs (wieviel Bytes) und den Port-Typ. Der Port-Typ ist eine identifizierende Zahl (Typkennung) aus der Tabelle im Abschnitt 2.4.2 auf der Seite 18.

Siehe Abbildung 2.3 auf der Seite 20.

Byte-Order der negativen Adressen Die negativen Adressen werden vom Adressdekoder auf den internen Speicherbereich des Bussystems abgebildet und zwar so, dass sie betragsmässig gleich sind. Es besteht daher für den Programmierer der Maschine keinen Unterschied zwischen den negativen und der positiven Adressen - außer dem Vorzeichen.

Beispiele Der folgende Code-Abschnitt liest die Anzahl der Bus-Ports in das Register $R1$:

```
  SET R1 0
  LBI R1 R1 -1
```

Die Instruktionen `SET` und `LBI` werden im Abschnitt 3.4, ab der Seite 31 beschrieben.

Der folgende Code sucht nach der Startadresse des Terminals, speichert diese Adresse in das Register $R7$ und gibt das Zeichen 'A' aus.

```
  SET   R1 -10       # R1 zeigt auf erste Typ-Adresse
search:
  LBI   R2  R1  0    # Typ in R2
  CMPIU R3  R2  1    # R3 ist Null falls Typ ist 1
                     # Terminal Typ ist 1
  BZ    R3  finish   # gefunden, mach weiter
  SUB   R1  R1  9    # Naechster Typ Eintrag
  JMP   search
finish:
  ADDI  R1  R1  8    # R1 zeigt auf Anfangsadresse
  LBI   R7  R1  0    # R7 zeigt auf Anfang der Terminaladresse
  SET   R2  65       # R2 = 'A'
  SW    R2  R7 ZERO  # Ausgabe von 'A'
```

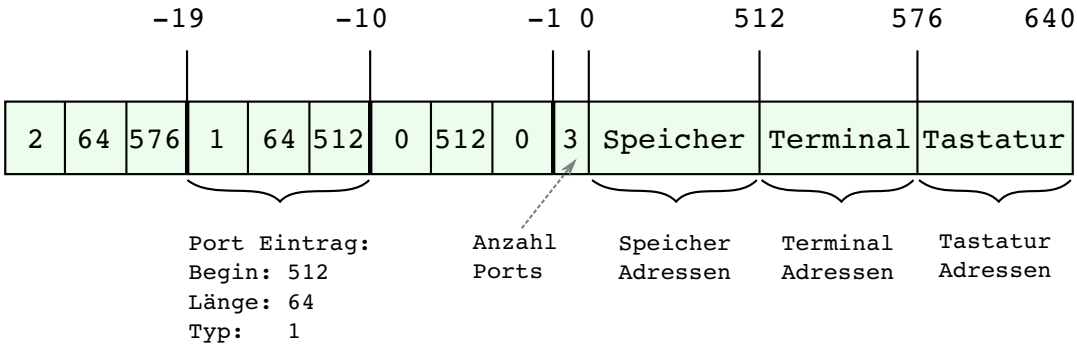

Abbildung 2.3: Adressraumverteilung im Bussystem

Positive Adressen

Der Speicher bekommt einen Adressraum, der bei der Adresse Null anfängt. Die Größe seines Adressraums ist von den Adressraum-Bedarf der anderen Ports und von der eigenen Speicherkapazität (Adressraum-Bedarf) begrenzt. Nach dem Adressraum des Speichers befinden sich die Adressräume der anderen Ports, normalerweise in der Reihenfolge ihrer Typkennung, die im Abschnitt 2.4.2 auf Seite 18 angegeben sind.

Reicht der gesamte Adressraum nicht für alle Ports, so wird der Adressraum des Speichers gekürzt, bis der Bedarf aller Ports mit einem festen Adressraum-Bedarf erfüllt ist.

3 Instruktionssatz

In diesem Kapitel werden alle Instruktionen der UMach VM vorgestellt.

3.1 Instruktionsformate

Eine Instruktion besteht aus einer Folge von 4 Bytes. Das Instruktionsformat beschreibt die Struktur einer Instruktion auf Byte-Ebene. Das Format gibt an, ob ein Byte als eine Registerangabe oder als reine numerische Angabe zu interpretieren ist.

Instruktionsbreite Jede UMach-Instruktion hat eine feste Bitlänge von 32 Bit (4 mal 8 Bit). Instruktionen, die für ihren Informationsgehalt weniger als 32 Bit brauchen, wie z.B. `NOP`, werden mit Nullbits gefüllt. Alle Daten und Informationen, die mit einer Instruktion übergeben werden, müssen in diesen 32 Bit untergebracht werden.

Byte Order Die Byte Order (Endianness) der gelesenen Bytes ist big-endian. Die zuerst gelesenen 8 Bits sind die 8 höchstwertigen (Wertigkeiten 2^{31} bis 2^{24}) und die zuletzt gelesenen Bits sind die niedrigstwertigen (Wertigkeiten 2^7 bis 2^0). Bits werden in Stücken von n Bits gelesen, wobei $n = k \cdot 8$ mit $k \in \{1, 4\}$ (byteweise oder wortweise).

Allgemeines Format Jede Instruktion besteht aus zwei Teilen: der erste Teil ist 8 Bit lang und entspricht dem tatsächlichen Befehl, bzw. der Operation, die von der UMach virtuellen Maschine ausgeführt werden soll. Dieser 8-Bit-Befehl belegt also die 8 höchstwertigen Bits einer 32-Bit-Instruktion. Die übrigen 24 Bits, wenn sie verwendet werden, werden für Operanden oder Daten benutzt. Beispiel einer Instruktionszerlegung:

Instruktion (32 Bit)	`00000001`	`00000010`	`00000011`	`00000100`
Hexa	`01`	`02`	`03`	`04`
Byte Order	erstes Byte	zweites Byte	drittes Byte	viertes Byte
Interpretation	Befehl (8 Bit)	Operanden, Daten oder Füllbits		

Die Instruktionsformate unterscheiden sich lediglich darin, wie sie die 24 Bits nach dem 8-Bit Befehl verwenden. Das wird auch in der 3-buchstabigen Benennung deren Formate wiedergeben.

In den folgenden Abschnitten werden die UMach-Instruktionsformate vorgestellt. Jede Angegebene Tabelle gibt in der ersten Zeile die Reihenfolge der Bytes an. Die nächste Zeile gibt die spezielle Belegung der einzelnen Bytes an.

3.1.1 000

erstes Byte	zweites Byte	drittes Byte	viertes Byte
Befehl	nicht verwendet		

Eine Instruktion, die das Format 000 hat, besteht lediglich aus einem Befehl ohne Argumenten. Die letzen drei Bytes werden von der Maschine nicht ausgewertet und sind somit Füllbytes. Es wird empfohlen, die letzten 3 Bytes mit Nullen zu füllen.

3.1.2 NNN

erstes Byte	zweites Byte	drittes Byte	viertes Byte
Befehl	numerische Angabe N		

Die Instruktion im Format NNN besteht aus einem Befehl im ersten Byte und aus einer numerischen Angabe N (einer Zahl), die die letzten 3 Bytes belegt. Die Interpretation der numerischen Angabe wird dem jeweiligen Befehl überlassen.

3.1.3 R00

erstes Byte	zweites Byte	drittes Byte	viertes Byte
Befehl	R_1	nicht verwendet	

Die Instruktion im Format R00 besteht aus einem Befehl im ersten Byte gefolgt von der Angabe eines Registers im zweiten Byte. Die letzten zwei Bytes werden nicht verwendet, bzw. werden ignoriert.

3.1.4 RNN

erstes Byte	zweites Byte	drittes Byte	viertes Byte
Befehl	R_1	numerische Angabe N	

Eine Instruktion im Format RNN besteht aus einem Befehl, gefolgt von einer Register Nummer R_1, gefolgt von einer festen Zahl N, die die letzten 2 Bytes der Instruktion belegt. Die genaue Interpretation der Zahl N wird dem jeweiligen Befehl überlassen. Zum Beispiel, die Instruktion

erstes Byte	zweites Byte	drittes Byte	viertes Byte
`0x20`	`0x01`	`0x02`	`0x03`

wird folgenderweise von der UMach Maschine interpretiert: die Operation mit Nummer `0x20` soll ausgeführt werden, wobei die Argumenten dieser Operation sind das Register mit Nummer `0x01` und die numerische Angabe `0x0203`.

3.1.5 RR0

erstes Byte	zweites Byte	drittes Byte	viertes Byte
Befehl	R_1	R_2	nicht verwendet

Eine Instruktion im Format RR0 besteht aus einem Befehl im ersten Byte, gefolgt von der Angabe zweier Register in den folgenden 2 Bytes. Das dritte Byte wird nicht verwendet, bzw. wird ignoriert. Entspricht einer unären Operation auf das Register R_2.

3.1.6 RRN

erstes Byte	zweites Byte	drittes Byte	viertes Byte
Befehl	R_1	R_2	numerische Angabe N

Eine Instruktion im Format RRN besteht aus einem Befehl, gefolgt von der Angabe zweier Registers R_1 und R_2, jeweils in einem Byte, gefolgt von einer numerischen Angabe N (festen Zahl) im letzten Byte. Zum Beispiel, die Instruktion

erstes Byte	zweites Byte	drittes Byte	viertes Byte
`0x52`	`0x01`	`0x02`	`0x03`

soll wie folgt interpretiert werden: die Operation mit Nummer `0x52` soll ausgeführt werden, wobei die Argumenten dieser Operation sind Register mit Nummer `0x01`, Register mit Nummer `0x02` und die Zahl `0x03`.

3.1.7 RRR

erstes Byte	zweites Byte	drittes Byte	viertes Byte
Befehl	R_1	R_2	R_3

Eine Instruktion im Format RRR besteht aus der Angabe eines Befehls im ersten Byte, gefolgt von der Angabe dreier Register R_1, R_2 und R_3 in den jeweiligen folgenden drei Bytes. Die Register werden als Zahlen angegeben und deren Bedeutung hängt vom jeweiligen Befehl ab.

3.1.8 Zusammenfassung

Im folgenden werden die Instruktionsformate tabellarisch zusammengefasst.

Format	erstes Byte	zweites Byte	drittes Byte	viertes Byte
000	Befehl	nicht verwendet		
NNN	Befehl	numerische Angabe N		
R00	Befehl	R_1	nicht verwendet	
RNN	Befehl	R_1	numerische Angabe N	
RR0	Befehl	R_1	R_2	nicht verwendet
RRN	Befehl	R_1	R_2	numerische Angabe N
RRR	Befehl	R_1	R_2	R_3

3.2 Verteilung des Befehlsraums

Zur besseren Übersicht der verschiedenen UMach-Instructionen, unterteilen wir den Instruktionssatz der UMach virtuellen Maschine in den folgenden Kategorien:

Maschinencodes	Kategorie
00 - 0F	Kontrollbefehle
10 - 4F	Lade-/Speicherbefehle
50 - 8F	Arithmetische Befehle
90 - AF	Logische Befehle
B0 - CF	Vergleichsbefehle
D0 - DF	Sprungbefehle
E0 - EF	Unterprogrambefehle
F0 - FF	Systembefehle

Tabelle 3.1: Verteilung des Befehlsraums nach Befehlskategorien. Die Zahlen sind im Hexadezimalsystem angegeben.

1. Kontrollinstruktionen, die die Maschine in ihrer gesamten Funktionalität betreffen, wie z.B. den Betriebsmodus umschalten.

2. Lade- und Speicherbefehle, die Register mit Werten aus dem Speicher, anderen Registern oder direkten numerischen Angaben laden und die Registerinhalte in den Speicher schreiben.

3. Arithmetische Instruktionen, die einfache arithmetische Operationen zwischen Registern veranlassen.

4. Logische Instruktionen, die logische Verknüpfungen zwischen Registerinhalten oder Operationen auf Bit-Ebene in Registern anweisen.

5. Vergleichsinstruktionen, die einen Vergleich zwischen Registerinhalten angeben.

6. Sprunginstruktionen, die bedingt oder unbedingt sein können. Sie weisen die UMach Maschine an, die Programmausführung an einer anderen Stelle fortzufahren.

7. Unterprogramm-Steuerung, bzw. Instruktionen, die die Ausführung von Unterprogrammen (Subroutinen) steuern.

8. Systeminstruktionen, die die Unterstützung eines Betriebssystem ermöglichen.

Die oben angegebenen Instruktionskategorien unterteilen den Befehlsraum in 8 Bereiche. Es gibt 256 mögliche Befehle, gemäß $2^8 = 256$. Die Verteilung der Kategorien auf die verschiedenen Maschinencode-Intervallen wird in der Tabelle 3.1 auf Seite 25 angegeben.

Die Tabelle 3.2 auf der Seite 26 enthält eine Übersicht aller Befehle und deren Maschinencodes. Diese Tabelle wird folgenderweise gelesen: in der am weitesten linken Spalte wird die erste hexadezimale Ziffer eines Befehls angegeben (ein Befehl ist zweistellig im

Tabelle 3.2: Befehlentabelle

	0	1	2	3	4	5	6	7	
0	NOP	RST	CRM	CSM	DIE	RSR	AUTSM	SOCL	cntrl
	HATE	TRST	ZMB	ALIV					
1	SET	SETU							set copy
	COPY		MOVE						
2	LB	LBU	LH	LHU	LW	LWU			load
	LBI	LBUI	LHI	LHUI	LWI	LWUI			
3	SB	SBU	SH	SHU	SW	SWU			store
	SBI	SBUI	SHI	SHUI	SWI	SWUI			
4	PUSHB	PUSHH	PUSH						push/pop
	POPB	POPH	POP						
5	ADD	ADDU	ADDI	ADDIU					add
	SUB	SUBU	SUBI	SUBIU					sub
6	MUL	MULU	MULI	MULIU	MULD				mult
	DIV	DIVU	DIVI	DIVIU	DIVD				div
7	MOD		MODI						mod
	ABS								unary
8	NEG	INC	DEC						
9	AND	ANDI	OR	ORI	XOR	XORI	NOT	NOTI	and or...
	NAND	NANDI	NOR	NORI					
A	SHL	SHLI	SHR	SHRI	SHRA	SHRAI			shift
	ROTL	ROTLI	ROTR	ROTRI					rotate
B	CMP	CMPU	CMPI	CMPIU					compare
C									
D	BE	BNE	BL	BLE	BG	BGE			branch
	JMP	JMPR						GO	
E	CALL	RET							call
F	WAKE								syscall
	KILL								
	8	9	A	B	C	D	E	F	

Hexadezimalsystem). Jede solche Ziffer hat rechts Zwei Zeilen, die von links nach rechts gelesen werden: eine Zeile für die Ziffern von 0 bis 8, die anderen für die übrigen Ziffern 9 bis F (im Hexadezimalsystem). Die Assemblernamen (Mnemonics) der einzelnen Befehle sind an der entsprechenden Stelle angegeben.

Definitionsstruktur Im den folgenden Abschnitten werden die einzelnen Instruktionen beschrieben. Zu jeder Instruktion wird der Assemblername, die Parameter, der Maschinencode (Maschinenname) und das Instruktionsformat, das die Typen der Parameter definiert, formal angegeben. Zudem werden Anwendungsbeispiele angegeben. Die Instruktionsformate können im Abschnitt 3.1 ab der Seite 21 nachgeschlagen werden.

Zur Notation Mit $\mathcal{R}$ wird die Menge aller Register gekennzeichnet[1]. Die Notation $X \in \mathcal{R}$ bedeutet, dass X ein Element aus dieser Menge ist, mit anderen Worten, dass X ein Register ist. Analog bedeutet die Schreibweise $X, Y \in \mathcal{R}$, dass X und Y beide Register sind.

Gilt $X, Y \in \mathcal{R}$ und ist $\sim$ eine durch einen Befehl definierte Relation zwischen X und Y, so bezieht sich die Schreibweise $X \sim Y$ nicht auf die Maschinennamen von X und Y, sondern auf deren Inhalte. Zum Beispiel, haben die Register $R1$ und $R2$ die Maschinencodes `0x01` und `0x02` und sind sie mit den Werten 4 bzw. 5 belegt, so bedeutet $R1 + R2$ das gleiche wie $4 + 5 = 9$ und nicht `0x01 + 0x02 = 0x03`.

Andere verwendeten Schreibweisen:

$\mathbb{N}$	Menge aller natürlichen Zahlen: $0, 1, 2, \ldots$
$\mathbb{N}_{\setminus 0}$	$\mathbb{N}$ ohne die Null: $1, 2, \ldots$
$\mathbb{Z}$	Menge aller ganzen Zahlen: $\ldots, -2, -1, 0, 1, 2, \ldots$
$\mathbb{Z}_{\setminus 0}$	$\mathbb{Z}$ ohne die Null: $\ldots, -2, -1, 1, 2, \ldots$
$N \in \mathbb{N}$	N ist Element von $\mathbb{N}$, oder liegt im Bereich von $\mathbb{N}$
$X \leftarrow Y$	X wird auf Y gesetzt
$mem(X)$	Speicherinhalt an der Adresse X (1 Byte)
$mem_n(X)$	n-Bytes-Block im Speicher ab Adresse X
`mem[n]`	äquivalent zu $mem(n)$

[1] Nicht verwechseln mit den Symbolen $\mathbb{R}$ und $\mathbb{R}$, die die Menge aller reellen Zahlen bedeuten.

3.3 Kontrollinstruktionen

3.3.1 `NOP`

Assemblername	Parameter	Maschinencode	Format
`NOP`	keine	`0x00`	000

Diese Instruktion („No Operation“) bewirkt nichts. Der Sinn dieser Instruktion ist, den Maschinencode mit Nullen füllen zu können, ohne dabei die gesamte Ausführung zu beeinflussen, außer, Zeitlupen zu schaffen.

3.3.2 `RST`

Assemblername	Parameter	Maschinencode	Format
`RST`	keine	`0x01`	000

Setzt alle Allzweckregister auf Null.

3.3.3 `CRM`

Assemblername	Parameter	Maschinencode	Format
`CRM`	$N \in \mathbb{N}$	`0x02`	NNN

„Change Run Mode“. Setzt das Betriebsmodus. Dabei ist das Parameter einer der folgenden konstanten Werten:

0	Normalmodus
1	Einzelschrittmodus

Siehe Abschnitt 2.1.1 auf Seite 7.

3.3.4 `CSM`

Assemblername	Parameter	Maschinencode	Format
`CSM`	$N \in \mathbb{N}$	`0x03`	NNN

Change System Mode.

3.3.5 `DIE`

Assemblername	Parameter	Maschinencode	Format
`DIE`	keine	`0x04`	000

Die Maschine ausschalten.

3.3.6 `RSR`

Assemblername	Parameter	Maschinencode	Format
`RSR`	keine	`0x05`	000

„Resurrect“. Die Maschine neustarten.

3.3.7 `AUTSM`

Assemblername	Parameter	Maschinencode	Format
`AUTSM`	keine	`0x06`	000

„Become autistic“. Bewirkt, dass alle Lese- und Schreibbefehle, die sich nicht ausschließlich auf Register beziehen, wirkungslos sind. Praktisch wird die Kommunikation mit dem Bussystem ausgeschaltet.

3.3.8 SOCL

Assemblername	Parameter	Maschinencode	Format
SOCL	keine	0x07	000

„Become social“. Schaltet die Kommunikation mit dem Bussystem ein.

3.3.9 HATE

Assemblername	Parameter	Maschinencode	Format
HATE	keine	0x08	000

„Hate“. Schaltet alle Schutzmechanismen ein.

3.3.10 TRST

Assemblername	Parameter	Maschinencode	Format
TRST	keine	0x09	000

„Trust“. Schaltet alle Schutzmechanismen aus.

3.3.11 ZMB

Assemblername	Parameter	Maschinencode	Format
ZMB	keine	0x0A	000

„Become a zombie“. Schaltet alle Registeränderungen aus. Nach der Ausführung dieses Befehls, alle Operationen, die einen Register modifizieren sollen, sind wirkungslos. Die Maschine ändert ihren Zustand nicht mehr, außer, dass sie weitere Befehle liest (das IP Register wird weiter erhöht).

3.3.12 ALIV

Assemblername	Parameter	Maschinencode	Format
ALIV	keine	0x0B	000

„Become alive“. Nach der Ausführung dieses Befehls, alle Register können wie normal modifiziert werden.

3.4 Lade- und Speicherbefehle

3.4.1 SET

Assemblername	Parameter	Maschinencode	Format
SET	$X \in \mathcal{R}$, $N \in \mathbb{Z}$	0x10	RNN

Setzt den Inhalt des Registers X auf den ganzzahligen Wert N. Da N mit 16 Bit und im Zweierkomplement dargestellt wird, kann N Werte von -2^{15} bis $2^{15}-1$ aufnehmen, bzw. von -32768 bis $+32767$. Werte außerhalb dieses Intervalls werden auf Assembler-Ebene entsprechend gekürzt (es wird modulo berechnet, bzw. nur die ersten 16 Bits aufgenommen).

Beispiele:

```
label:
  SET R1  8     # R1 ← 8
  SET R2 -3     # R2 ← −3
  SET R3 65536  # R3 ← 0, da 65536 = 2^16 ≡ 0 mod 2^16
  SET R4 70000  # R3 ← 4464 = 70000 mod 2^16
  SET R7 label  # Adresse 'label' ins R7
```

3.4.2 SETU

Assemblername	Parameter	Maschinencode	Format
SETU	$X \in \mathcal{R}$, $N \in \mathbb{N}$	0x11	RNN

Setzt den Inhalt des Registers X auf den positiven natürlichen Wert N. N wird vorzeichenlos interpretiert. Entsprechend kann N Werte von 0 bis +65535 aufnehmen. Wird dem Assembler einen Wert außerhalb dieses Bereichs gegeben, so schneidet der Assembler alle Bits außer den ersten 16 weg und betrachtet das Ergebnis als vorzeichenlose Zahl.

Beispiele:

```
SETU R1    8      # R1 ← 8
SETU R2    70000  # R2 ← 4464
SETU R2   -70000  # R2 ← 61072
```

3.4.3 COPY

Assemblername	Parameter	Maschinencode	Format
COPY	$X, Y \in \mathcal{R}$	0x18	RR0

Kopiert den Inhalt des Registers Y in das Register X. Register Y wird dabei nicht geändert. Entspricht

$$X \leftarrow Y$$

Beispiel:

```
SET   R1 5   # R1 ← 5
COPY  R2 R1  # R2 ← 5
```

3.4.4 MOVE

Assemblername	Parameter	Maschinencode	Format
MOVE	$X, Y \in \mathcal{R}$	0x1A	RR0

Ähnlich wie COPY, kopiert dieser Befehl den Inhalt des Registers Y in das Register X. Anders aber als COPY, setzt MOVE das Register Y auf Null. Entspricht einer echten Verschiebung eines Wertes von Y nach X. Gemäß der Formatspezifikation, können nur Register als Argumenten verwendet werden. Um den Inhalt eines Registers auf einen konstanten Wert zu setzen, ist SET zu verwenden. Um den Inhalt eines Registers mit einem Speicherinhalt zu belegen, ist z.B. LW zu verwenden.

Entspricht

$$X \leftarrow Y$$
$$Y \leftarrow 0$$

```
SET  R1  5   # Inhalt von R1 ist 5
MOVE R2 R1   # R1 = 0, R1 = 5
MOVE R1 R2   # R1 = 5, R1 = 0
MOVE R2  7   # Fehler, da 7 kein Register
```

3.4.5 LB

Assemblername	Parameter	Maschinencode	Format
LB	$X, Y, Z \in \mathcal{R}$	0x20	RRR

Lade ein Byte aus dem Speicher mit Adresse $Y + Z$ in das niedrigstwertige Byte des Registers X. Die anderen Bytes von X werden von diesem Befehl nicht betroffen. Besonders, sie werden nicht auf Null gesetzt. Alle Registerinhalte werden als vorzeichenbehaftet behandelt. Algebraisch äquivalent:

$$X \leftarrow (X \div 2^8) \cdot 2^8 + \left(mem(Y + Z) \bmod 2^8\right)$$

Äquivalenter C Code:

```
x = (x & 0xFFFFFF00) | (mem(y + z) & 0x00FF);
```

Beispiel Angenommen, der Speicher an den Adressen 100 und 101 hat den Wert 5, bzw. 6.

```
SET  R1   100   # Basisadresse R1 = 100
SET  R2     0   # Index R2 = 0
SET  R3     0
LB   R3 R1 R2   # R3 = 5 (mem(100+0))
SHLI R3 R3  8   # shift left 8 Bit, R3 = 1280
INC  R2         # R2++, R2 = 1
LB   R3 R1 R2   # R3 = 1286 (R3 + mem(100+1))
```

Hier werden zwei nacheinander folgenden Bytes aus dem Speicher gelesen und in die zwei niedrigstwertigen Bytes von $R3$ abgelegt. Das gleiche kann man kürzer mit dem Befehl LH erreicht werden.

3.4.6 LBU

Assemblername	Parameter	Maschinencode	Format
LBU	$X, Y, Z \in \mathcal{R}$	0x21	RRR

„Load Byte Unsigned“. Analog dem Befehle LB mit dem Unterschied, dass alle Registerinhalte als vorzeichenlos interpretiert werden. Somit ergeben sich größere Wertebereiche für die Adressen.

3.4.7 LH

Assemblername	Parameter	Maschinencode	Format
LH	$X, Y, Z \in \mathcal{R}$	0x22	RRR

„Load Half“. Lade ein halbes Word (2 Bytes) aus der Adresse $Y + Z$ in die zwei niedrigstwertigen Bytes des Registers X. Die beiden höchstwertigen Bytes von X werden nicht verändert.

Beispiel Angenommen, der Speicher an den Adressen 100 und 101 hat den Wert 5, bzw. 6. Die Adressen 100 und 101 bilden also zusammen den Wert 0x0506 (1286).

```
SET  R1   100      # Basisadresse R1 = 100
SET  R2     0      # Index R2 = 0
SET  R3     0
LH   R3 R1 R2      # R3 = 1286 (mem2(100+0))

SET  R3  0x010000 # R3 = 65536 (3. Byte auf 1 gesetzt)
LH   R3 R1 R2     # R3 = 66822 = 65536+1286
```

3.4.8 LHU

Assemblername	Parameter	Maschinencode	Format
LHU	$X, Y, Z \in \mathcal{R}$	0x23	RRR

„Load Half Unsigned“. Analog zur Instruktion LH mit dem Unterschied, dass alle Registerinhalte als vorzeichenlos interpretiert werden.

3.4.9 LW

Assemblername	Parameter	Maschinencode	Format
LW	$X, Y, Z \in \mathcal{R}$	0x24	RRR

„Load Word“. Lade ein Wort (4 Byte) aus dem Speicher mit Adresse $Y + Z$ in das Register X. Alle Bytes von X werden dabei überschrieben. Die Bytes aus dem Speicher werden nacheinander gelesen. Es werden also die Bytes mit Adressen $Y + Z + 0$, $Y + Z + 1$, $Y + Z + 2$ und $Y + Z + 3$ zu einem 4-Byte Wort zusammengesetzt und so in X ablegt.

Beispiel Abgenommen, die Adressen von 100 bis 103 sind mit den Werten 0, 1, 2 und 3 belegt und bilden somit den Wert 66051.

```
SET R1   100
SET R2     0
LW  R3 R1 R2  # R3 ← mem4(R1 + R2) = 66051
LW  R3 R1  8  # Fehler! 8 ist kein Register nutze LWI dafuer
```

3.4.10 LWU

Assemblername	Parameter	Maschinencode	Format
LWU	$X, Y, Z \in \mathcal{R}$	0x25	RRR

„Load Word Unsigned“. Analog zur Instruktion LW mit dem Unterschied, dass alle Registerinhalte vorzeichenlos interpretiert werden.

3.4.11 LBI

Assemblername	Parameter	Maschinencode	Format
LBI	$X, Y \in \mathcal{R}$, $N \in \mathbb{Z}$	0x28	RRN

„Load Byte Immediate“. Lade ein Byte aus dem Speicher mit Adresse $Y + N$ in das niedrigstwertige Byte des Registers X. N ist dabei eine feste, konstante Zahl, die zum Inhalt von Y hinzuaddiert wird.

Beispiel Das folgende Beispiel setzt das niedrigstwertige Byte des Registers $R2$ auf denn Speicherinhalt mit Adresse 116:

```
SET R1    100   # R1 ← 100
SET R2      0
LBI R2 R1  16   # R2 ← mem(100 + 16)
```

3.4.12 LBUI

Assemblername	Parameter	Maschinencode	Format
LBUI	$X, Y \in \mathcal{R}$, $N \in \mathbb{N}$	0x29	RRN

„Load Byte Unsigned Immediate“. Analog zur Instruktion LBI mit dem Unterschied, dass sowohl Y als auch N vorzeichenlos interpretiert werden.

3.4.13 LHI

Assemblername	Parameter	Maschinencode	Format
LHI	$X, Y \in \mathcal{R}$, $N \in \mathbb{Z}$	0x2A	RRN

„Load Half Immediate“. Lädt aus dem Speicher ab der Adresse $Y + N$ zwei nacheinander folgenden Bytes in die zwei niedrigstwertigen Bytes des Registers X. Y ist ein Register und N eine ganze Zahl im Bereich $[-128, 127]$. Die Instruktion funktioniert wie LH, außer, dass N eine feste Zahl und kein Register ist.

3.4.14 LHUI

Assemblername	Parameter	Maschinencode	Format
LHUI	$X, Y \in \mathcal{R}$, $N \in \mathbb{N}$	0x2B	RRN

„Load Half Unsigned Immediate“. Analog zur Instruktion LHI mit dem Unterschied, dass alle Operanden vorzeichenlos interpretiert werden.

3.4.15 LWI

Assemblername	Parameter	Maschinencode	Format
LWI	$X, Y \in \mathcal{R}$, $N \in \mathbb{Z}$	0x2C	RRN

„Load Word Immediate“. Lädt ein Wort (4 Bytes) ab der Adresse $Y + N$ in das Register X. Entspricht dem algebraischen Ausdruck

$$X \leftarrow mem_4(Y + N)$$

Funktioniert wie LW mit dem Unterschied, dass N eine feste konstante ganze Zahl ist.

Beispiel Angenommen, an den Speicheradressen 100 bis 104 stehen die Werte 0x01, 0x02, 0x03, 0x04 und 0x05.

```
SET R1 100
LWI R2 R1 0 # R2 = 0x01020304
LWI R2 R1 1 # R2 = 0x02030405
```

3.4.16 LWUI

Assemblername	Parameter	Maschinencode	Format
LWUI	$X, Y \in \mathcal{R}$, $N \in \mathbb{N}$	0x2D	RRN

„Load Word Unsigned Immediate“. Analog zur Instruktion LWI mit dem Unterschied, dass alle Werte (insbesondere N) vorzeichenlos interpretiert werden.

3.4.17 SB

Assemblername	Parameter	Maschinencode	Format
SB	$X, Y, Z \in \mathcal{R}$	0x30	RRR

„Store Byte“. Speichert den Inhalt des niedrigstwertigen Byte von X an der Speicherstelle $Y + Z$. X, Y und Z sind dabei Register. Deren Inhalt wird als vorzeichenbehaftet interpretiert.

Entspricht dem algebraischen Ausdruck

$$X \to mem_1(Y + Z)$$

$mem_1(x)$ bedeutet dabei 1 Byte an der Adresse x.

```
SET R1 128      # R1 = Speicheradresse 128
SET R2 513      # R2 = 0x0201
SB  R2 R1 ZERO # Speicher mit Adresse 128 wird auf 1 gesetzt
```

`ZERO` ist dabei ein Spezialregister mit konstantem Wert 0.

3.4.18 SBU

Assemblername	Parameter	Maschinencode	Format
SBU	$X, Y, Z \in \mathcal{R}$	0x31	RRR

„Store Byte Unsigned“. Analog zur Instruktion SB mit dem Unterschied, dass X, Y und Z vorzeichenlos behandelt werden.

3.4.19 SH

Assemblername	Parameter	Maschinencode	Format
SH	$X, Y, Z \in \mathcal{R}$	0x32	RRR

„Store Half“. Speichert die 2 niedrigstwertigen Bytes von X (rechte Hälfte von X) an die Adressen $Y + Z$ und $X + Y + 1$ so, dass diese zwei Adressen mit den letzten Bytes von X übereinstimmen. Die folgende Tabelle zeigt die bitweise Übereinstimmung der betroffenen Stellen.

X Wertigkeiten	2^{15}	2^{14}	2^{13}	2^{12}	2^{11}	2^{10}	2^9	2^8	2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0
Speicherstellen	$X + Y$								$X + Y + 1$							

Entspricht dem algebraischen Ausdruck

$$X \to mem_2(Y + Z)$$

Beispiel Dieses Programm speicher die 2 niedrigstwertigen Bytes von $R2$ in den Speicher an die Adressen 128 und 129.

```
SET R1 128      # R1 = Speicheradresse 128
SET R2 513      # R2 = 0x0201
SH  R2 R1 ZERO # Speicher mit Adresse 128: 0x02
               # Speicher mit Adresse 129: 0x01
```

3.4.20 SHU

Assemblername	Parameter	Maschinencode	Format
SHU	$X, Y, Z \in \mathcal{R}$	0x33	RRR

„Store Half Unsigned“. Analog zur Instruktion SH mit dem Unterschied, dass X, Y und Z vorzeichenlos interpretiert werden.

3.4.21 SW

Assemblername	Parameter	Maschinencode	Format
SW	$X, Y, Z \in \mathcal{R}$	0x34	RRR

„Store Word“. Speichert den Inhalt aller Bytes in X an die Speicheradressen $Y + Z$ bis $X + Z + 3$.

$$X \to mem_4(Y + Z)$$

Beispiel Es wird das Register $R2$ mit dem Wert 0x01020304 geladen und an die Adresse 128 gespeichert. Dabei werden die Byte-Werten in „big-endian“ Reihenfolge gespeichert: das höchstwertige Byte aus $R2$ (0x01) wird an der Adresse 128 gespeichert, das niedrigstwertige Byte (0x04) an die Adresse 131.

```
SET R1 128          # R1 = Speicheradresse 128
SET R2 0x01020304   # Wert zum Speichern
SH  R2 R1 ZERO      # mem[128] = 0x01
                    # mem[129] = 0x02
                    # mem[130] = 0x03
                    # mem[131] = 0x04
```

3.4.22 SWU

Assemblername	Parameter	Maschinencode	Format
SWU	$X, Y, Z \in \mathcal{R}$	0x35	RRR

„Store Word Unsigned“. Analog zur Instruktion SW, aber die Register X, Y und Z werden vorzeichenlos gelesen.

3.4.23 SBI

Assemblername	Parameter	Maschinencode	Format
SBI	$X, Y \in \mathcal{R}$, $N \in \mathbb{Z}$	0x38	RRN

„Store Byte Immediate“. Speichert das niedrigstwertige Byte aus dem Register X an die Adresse $Y + N$. Der Unterschied zur Instruktion SB ist, dass N eine ganzahlige direkte Angabe ist.

Beispiel Der folgende Code schreibt die Zahl 0x0304 an die Adressen 128 und 129, wobei die zwei Bytes vertauscht werden.

```
SET  R1 0x0304   # Wert zum Speichern in R1
SET  R2 128      # Adresse in R2
SBI  R1 R2 0     # mem[128] = 0x04
SHRI R1 R1 8     # R1 = 0x0003 (shift right 8 Bit)
SBI  R1 R2 1     # mem[129] = 0x03
```

Nützlich, wenn die Adresse konstant ist und keine variablen Indizes verwendet werden.

3.4.24 SBUI

Assemblername	Parameter	Maschinencode	Format
SBUI	$X, Y \in \mathcal{R}$, $N \in \mathbb{N}$	0x39	RRN

„Store Byte Unsigned Immediate“. Analog zur Instruktion SBI mit dem Unterschied, dass X, Y und N vorzeichenlose Angaben sind.

3.4.25 SHI

Assemblername	Parameter	Maschinencode	Format
SHI	$X, Y \in \mathcal{R}$, $N \in \mathbb{Z}$	0x3A	RRN

„Store Half Immediate“. Speichert die 2 niedrigstwertigen Bytes von X an die Adressen $Y + N$ (3. Byte von X) und $Y + N + 1$ (4. Byte von X). N ist dabei eine konstante ganze Zahl, die direkt angegeben wird.

```
SET R1 0x0304 # Wert zum Speichen.
              # Byte 3 = 0x03
              # Byte 4 = 0x04
SET R2 128    # Die Adresse in R2
SHI R1 R2  0  # mem[128] = 0x03
              # mem[129] = 0x04
```

3.4.26 SHUI

Assemblername	Parameter	Maschinencode	Format
SHUI	$X, Y \in \mathcal{R}$, $N \in \mathbb{N}$	0x3B	RRN

„Store Half Unsigned Immediate“. Funktioniert wie SHI aber mit vorzeichenlosen Werten.

3.4.27 SWI

Assemblername	Parameter	Maschinencode	Format
SWI	$X, Y \in \mathcal{R}$, $N \in \mathbb{Z}$	0x3C	RRN

„Store Word Immediate“. Speichert den Inhalt des Registers X an die Byte-Adressen von $Y + N + 0$ bis $Y + N + 3$. Die Bytes werden in „Big-Endian“ Reihenfolge gelesen und gespeichert. N ist eine direkt angegebene ganze Zahl aus dem Intervall $[-128, 127]$.

Entspricht dem algebraischen Ausdruck

$$X \to mem_4(Y + N)$$

```
SET R1 0x01020304 # Wert zum speichern in R1
                  # 1. Byte = 0x01
                  # 2. Byte = 0x02
                  # 3. Byte = 0x03
                  # 4. Byte = 0x04
SET R2 200        # Adresse 200 in R2
SWI R1 R2 56      # Addiere den Versatz 56 zu 200
                  # mem[256] = 0x01
                  # mem[257] = 0x02
                  # mem[258] = 0x03
                  # mem[259] = 0x04
```

3.4.28 SWUI

Assemblername	Parameter	Maschinencode	Format
SWUI	$X, Y \in \mathcal{R}$, $N \in \mathbb{N}$	0x3D	RRN

„Store Word Unsigned Immediate“. Wie SWI mit dem Unterschied, dass alle Register und Werten vorzeichenlos gelesen und gespeichert werden. Besonders gilt $N \in [0, 255]$. Wird eine größere Zahl angegeben, so wird die modulo 256 berechnet.

3.4.29 PUSHB

Assemblername	Parameter	Maschinencode	Format
PUSHB	$X \in \mathcal{R}$	0x40	R00

„Push Byte“.

Erniedrigt das Register SP um 1 und speichert das vierte (niedrigstwertige) Byte aus dem Register X an die Adresse, die in SP gespeichert ist. Entspricht

$$\begin{aligned} SP &\leftarrow SP - 1 \\ X &\rightarrow mem_1(SP) \end{aligned}$$

3.4.30 PUSHH

Assemblername	Parameter	Maschinencode	Format
PUSHH	$X \in \mathcal{R}$	0x41	R00

„Push Half“. Erniedrigt das Register SP um 2 und speichert das 3. und 4. Byte (die zwei niedrigstwertigen) aus X an die Adresse SP.

Entspricht

$$\begin{aligned} SP &\leftarrow SP - 2 \\ X &\rightarrow mem_2(SP) \end{aligned}$$

Beispiel Dieser Code „pusht“ das 3. und 4. Byte von $R1$ auf den Stack:

```
SET   R1 0x0102
PUSHH R1          # mem[SP + 0] = 0x01
                  # mem[SP + 1] = 0x02
```

3.4.31 PUSH

Assemblername	Parameter	Maschinencode	Format
PUSH	$X \in \mathcal{R}$	0x42	R00

„Push Word“. Erniedrigt das Register SP um 4 und kopiert das ganze Register X auf den Stack, wobei der „Stack“ ist der Speicherbereich mit Anfangsadresse in SP. Die Byte-Reihenfolge der Lese- und Schreiboperationen ist „Big-Endian“ und wird in der nachfolgenden Tabelle dargestellt:

X Wertigkeiten	$2^{31} \leftrightarrow 2^{24}$	$2^{23} \leftrightarrow 2^{16}$	$2^{15} \leftrightarrow 2^{8}$	$2^{7} \leftrightarrow 2^{0}$
	↓	↓	↓	↓
Stack-Bereich	mem[SP + 0]	mem[SP + 1]	mem[SP + 2]	mem[SP + 3]

Dieser Befehl ist somit äquivalent zu

```
SUBI SP SP 4
SW   X  SP ZERO
```

Entspricht

$$SP \leftarrow SP - 4$$
$$X \rightarrow mem_4(SP)$$

Beispiel Der folgende Code speichert das 4-Byte Wort 0x01020304 auf den Stack. Die Stack-Struktur wird in Kommentaren gezeigt.

```
SET  R1 0x01020304 # Wert zum pushen
PUSH R1            # mem[SP + 0] = 0x01
                   # mem[SP + 1] = 0x02
                   # mem[SP + 2] = 0x03
                   # mem[SP + 3] = 0x04
```

3.4.32 POPB

Assemblername	Parameter	Maschinencode	Format
POPB	$X \in \mathcal{R}$	0x48	R00

„Pop Byte“. Ladet ein Byte aus dem Stack-Bereich in das niedrigstwertige Byte des Registers X und erhöht das Spezialregister `SP` um 1. Entspricht somit den Befehlen

```
LB   X SP ZERO
INC SP
```

Algebraische Formel:

$$X \leftarrow mem_1(SP)$$
$$SP \leftarrow SP + 1$$

Beispiel Angenommen, der „Top of Stack“ (Speicherinhalt an der Adresse `SP`) ist mit dem Byte `0x05` belegt:

```
SET  R1 0x0201 # irgendein Wert im Byte 3 und 4 von R1
POPB R1        # R1 = 0x0205
```

Der obige Code zeigt, dass nur das letzte Byte von $R1$ überschrieben wird.

3.4.33 `POPH`

Assemblername	Parameter	Maschinencode	Format
`POPH`	$X \in \mathcal{R}$	`0x49`	R00

„Pop Half“. Lädt die ersten zwei Bytes aus dem Stack-Bereich in die 2 niedrigstwertigen Bytes von X und erhöht das Register `SP` um 2. Äquivalente Instruktionen:

```
LHI   X SP 0
ADDI SP SP 2
```

Algebraische Formel:

$$X \leftarrow mem_2(SP)$$
$$SP \leftarrow SP + 2$$

Beispiel Angenommen, die ersten 4 Bytes vom Stack-Bereiche sind `0xAA 0xBB 0xCC 0xDD`

```
                # mem[SP + 0] = 0xAA
                # mem[SP + 1] = 0xBB
POPH R1         # R1 = 0xAABB
                # mem[SP + 0] = 0xCC
                # mem[SP + 1] = 0xDD
```

3.4.34 POP

Assemblername	Parameter	Maschinencode	Format
POP	$X \in \mathcal{R}$	0x4A	R00

„Pop Word“. Speichert 4 Bytes ab der Adresse SP in das Register X und erhöht SP um 4. Die Byte-Reihenfolge der Lese- und Schreiboperationen ist „Big-Endian“ und wird in der nachfolgenden Tabelle dargestellt.

X Wertigkeiten	$2^{31} \leftrightarrow 2^{24}$	$2^{23} \leftrightarrow 2^{16}$	$2^{15} \leftrightarrow 2^{8}$	$2^{7} \leftrightarrow 2^{0}$
	↑	↑	↑	↑
Stack-Bereich	mem[SP + 0]	mem[SP + 1]	mem[SP + 2]	mem[SP + 3]

Diese Instruktion kann algebraisch so ausgedrückt werden:

$$X \leftarrow mem_4(SP)$$
$$SP \leftarrow SP + 4$$

Die Instruktion POP ist äquivalent zu den folgenden Instruktionen:

```
LWI   X SP 0
ADDI SP SP 4
```

Beispiel Angenommen, die ersten 4 Bytes vom Stack-Bereiche sind 0xAA 0xBB 0xCC 0xDD.

```
POPH R1         # R1 = 0xAABBCCDD
```

3.5 Arithmetische Instruktionen

3.5.1 ADD

Assemblername	Parameter	Maschinencode	Format
ADD	$X, Y, Z \in \mathcal{R}$	0x50	RRR

Vorzeichen behaftete Addition der Registerinhalte Y und Z. Das Ergebnis der Addition wird in das Register X gespeichert. Entspricht dem algebraischen Ausdruck

$$X \leftarrow Y + Z$$

Beispiel:

```
SET   R1 1      # R1 ← 1
SET   R2 2      # R2 ← 2
ADD   R3 R1 R2  # R3 ← R1 + R2 = 1 + 2 = 3
#     X  Y  Z
SET   R2 -2     # R2 ← −2
ADD   R3 R3 R2  # R3 ← R3 + R2 = 3 + (−2) = 1
ADD   R3 R4  5  # Fehler! 5 kein Register
```

Vorzeichenlose Addition wird durch den Befehl ADDU ausgeführt.

3.5.2 ADDU

Assemblername	Parameter	Maschinencode	Format
ADDU	$X, Y, Z \in \mathcal{R}$	0x51	RRR

„Add Unsigned“. Vorzeichenlose Addition der Register Y und Z. Das Ergebnis wird in das Register X gespeichert. Enthält Y oder Z ein Vorzeichen (höchstwertiges Bit auf 1 gesetzt), so wird es nicht als solches interpretiert, sondern als Wertigkeit, die zum Betrag des Wertes hinzuaddiert wird ($+2^{31}$).

```
SET   R1 1      # R1 ← 1
SET   R2 -2     # R2 ← −2
ADDU  R3 R1 R2  # R3 ← (1 + 2 + 2^31) = 2147483651
```

3.5.3 ADDI

Assemblername	Parameter	Maschinencode	Format
ADDI	$X, Y \in \mathcal{R}$, $N \in \mathbb{Z}$	0x52	RRN

„Add Immediate“. Hinzuaddieren eines festen vorzeichenbehafteten ganzzahligen Wert N zum Inhalt des Registers Y und speichern des Ergebnisses in das Register X. Entspricht dem algebraischen Ausdruck

$$X \leftarrow Y + N$$

N wird als vorzeichenbehaftete 8-Bit Zahl in Zweierkomplement-Darstellung interpretiert und kann entsprechend Werte von -128 bis 127 aufnehmen.

Beispiel:

```
SET   R1 1      # R1 ← 1
ADDI  R2 R1 2   # R2 ← R1 + 2 = 1 + 2 = 3
#     X  Y  N
ADDI  R2 R2 -3  # R2 ← R2 + (−3) = 3 + (−2) = 1
ADDI  R2 R3 R4  # Fehler! R4 kein n ∈ Z
```

3.5.4 ADDIU

Assemblername	Parameter	Maschinencode	Format
ADDIU	$X, Y \in \mathcal{R}$, $N \in \mathbb{N}$	0x53	RRN

„Add Unsigned Immediate“. Vorzeichenlose Addition des ganzzahligen Wertes N zum Inhalt des Registers Y und speichern des Ergebnisses in das Register X. Der Inhalt des Registers Y, die Zahl N und das Ergebnis $Y + N$ werden als vorzeichenlose Werte interpretiert. Die Feste natürliche Zahl N kann Werte aus dem Bereich $[0, 255]$ aufnehmen. Wird eine größere Zahl angegeben, so wird sie modulo 256 berechnet.

3.5.5 SUB

Assemblername	Parameter	Maschinencode	Format
SUB	$X, Y, Z \in \mathcal{R}$	0x58	RRR

Subtrahiert die Registerinhalte von Y und Z und speichert das Ergebnis in das Register X. Entspricht dem Ausdruck

$$X \leftarrow (Y - Z)$$

Wobei X, Y und Z Register sind.

3.5.6 SUBU

Assemblername	Parameter	Maschinencode	Format
SUBU	$X, Y, Z \in \mathcal{R}$	0x59	RRR

„Subtract Unsigned“. Analog zur Instruktion SUB mit dem Unterschied, dass alle Werte und Operationen vorzeichenlos sind.

3.5.7 SUBI

Assemblername	Parameter	Maschinencode	Format
SUBI	$X, Y \in \mathcal{R}$, $N \in \mathbb{Z}$	0x5A	RRN

„Subtract Immediate“. Funktioniert wie SUB aber N ist eine direkt angegebene Zahl (kein Register).

Beispiel Folgendes Beispiel demonstriert die Verwendung von SUBI und zeigt zugleich einen Fehler.

```
SET  R1 50      # R1 ← 50
SUBI R2 R1 30   # R2 ← (R1 − 30) = 20
SUBI R2 R1 R1   # Fehler! da R1 ∉ Z
```

3.5.8 SUBIU

Assemblername	Parameter	Maschinencode	Format
SUBIU	$X, Y \in \mathcal{R}$, $N \in \mathbb{N}$	0x5B	RRN

„Subtract Immediate Unsigned“. Funktioniert wie die Instruktion `SUBI` mit dem Unterschied, dass `SUBIU` ausschliesslich mit vorzeichenlosen Werten arbeitet.

3.5.9 `MUL`

Assemblername	Parameter	Maschinencode	Format
`MUL`	$X, Y \in \mathcal{R}$	`0x60`	RR0

„Multiply“. Multipliziert die Inhalte der Register X und Y und speichert das Ergebnis in die Spezialregister `HI` und `LO`. Diese zwei Spezialregister werden als eine 64-Bit Einheit betrachtet, wobei jedes eine Hälfte des 64-Bit Ergebnisses enthält. Dabei werden die höchstwertigen 32 Bit des Ergebnisses in das Register `HI` und die 32 niedrigstwertigen Bits des Ergebnisses in das Register `LO` gespeichert. Siehe auch die Tabelle 2.1 auf der Seite 11.

Falls das Ergebnis der Multiplikation gänzlich in den 32 Bit des Registers `LO` passt, wird das Register `HI` trotzdem auf Null gesetzt.

Äquivalenter algebraischer Ausdruck:

$$(HI, LO) \leftarrow X \cdot Y$$

Beispiel Der folgende Code demonstriert die Verwendung der `MUL` Instruktion.

```
SET   R1 4     # R1 ← 4
SET   R2 5     # R2 ← 5
MUL   R1 R2    # HI ← 0
               # LO ← 20

SET   R1 0xAAAAAAAA
MUL   R1 R1    # R1^2
               # HI = 0x71C71C70
               # LO = 0xE38E38E4
COPY  R2 LO    # R1^2 mod 2^32
```

Falls es bekannt ist, dass das Ergebnis der Multiplikation sich mit 32 Bit darstellen lässt, kann man den Weg über die `LO` und `HI` Register mit der Instruktion `MULD` umgehen.

3.5.10 MULU

Assemblername	Parameter	Maschinencode	Format
MULU	$X, Y \in \mathcal{R}$	0x61	RR0

„Multiply Unsigned“. Funktioniert wie die Instruktion MUL mit dem Unterschied, dass die Multiplikationoperanden X und Y vorzeichenlos behandelt werden.

3.5.11 MULI

Assemblername	Parameter	Maschinencode	Format
MULI	$X \in \mathcal{R}$, $N \in \mathbb{Z}$	0x62	RNN

„Multiply Immediate“. Multipliziert den Inhalt des Registers X mit der ganzen Zahl N und speichert das 64-Bit Ergebnis in die Register HI und LO, die als ein einziges 64-Register betrachtet werden: HI enthält die ersten 32 Bits (die höchstwertigen) und LO die letzten 32 Bits (die niedrigstwertigen). Siehe auch die Instruktion MUL.

3.5.12 MULIU

Assemblername	Parameter	Maschinencode	Format
MULIU	$X \in \mathcal{R}$, $N \in \mathbb{N}$	0x63	RNN

„Multiply Immediate Unsigned“. Funktioniert wie die Instruktion MULI mit dem Unterschied, dass sowohl die Operanden X und N als auch das Ergebnis vorzeichenlos sind.

3.5.13 MULD

Assemblername	Parameter	Maschinencode	Format
MULD	$X, Y, Z \in \mathcal{R}$	0x64	RRR

„Multiply Direct“. Multipliziert die Inhalte der Register Y und Z und speichert die niedrigstwertigen 32 Bits des Ergebnisses in das Register X. Anders wie bei der Instruktion `MUL`, werden die Register `HI` und `LO` nicht verändert.

`MULD` entspricht der Instruktionen:

```
MUL   Y Z
COPY  X LO
```

wobei der alte Wert von `LO` erhalten bleibt.

Algebraisch geschrieben:

$$X \leftarrow (Y \cdot Z) \bmod 2^{32}$$

3.5.14 `DIV`

Assemblername	Parameter	Maschinencode	Format
`DIV`	$X, Y \in \mathcal{R}$	`0x68`	RR0

„Divide“, ganzzahlige Division. Dividiert den Inhalt des Registers X durch den Inhalt des Registers Y und speichert den Quotient in das Register `HI` und den Rest in das Register `LO`. Nach der Ausführung gilt

$$X = Y \cdot HI + LO$$

Algebraisch ausgedrückt:

$$HI \leftarrow \lfloor X/Y \rfloor$$
$$LO \leftarrow X \bmod Y$$

$\lfloor x \rfloor$ bedeutet in diesem Kontext, dass x auf die betragsmässig nächstkleinste ganze Zahl gerundet wird, oder die Nachkommastellen von x werden abgeschnitten.

Um den Weg über die Register `HI` und `LO` zu umgehen, dafür aber den Rest der Division zu verlieren, kann man die Instruktion `DIVD` verwenden.

Beispiel Der folgende Code demonstriert die Verwendung von `DIV`.

```
SET  R1  10    # R1 ← 10
SET  R2   3    # R2 ← 3
DIV  R1  R2    # HI ← 3
               # LO ← 1
```

3.5.15 DIVU

Assemblername	Parameter	Maschinencode	Format
DIVU	$X, Y \in \mathcal{R}$	0x69	RR0

„Divide Unsigned“. Funktioniert wie DIV mit dem Unterschied, dass ganzzahlige vorzeichenlose Division durchgeführt werden. Die Ergebnis-Register HI und LO enthalten entsprechend vorzeichenlose Werte.

3.5.16 DIVI

Assemblername	Parameter	Maschinencode	Format
DIVI	$X \in \mathcal{R}$, $N \in \mathbb{Z}_{\setminus 0}$	0x6A	RNN

„Divide Immediate“. Dividiert den Inhalt des Registers X durch die feste ganze Zahl N und speichert den Quotient in das Register HI und den Rest in das Register LO. N nimmt Werte aus dem Intervall $[-2^{15}, 2^{15} - 1] \setminus 0$. Nach der Durchführung der Division gilt:

$$X = HI \cdot N + LO$$

Beispiel Der folgende Code demonstriert die Verwendung von DIVI.

```
SET  R1  10    # R1 ← 10
DIVI R1   3    # HI ← 3
               # LO ← 1
```

3.5.17 DIVIU

Assemblername	Parameter	Maschinencode	Format
DIVIU	$X \in \mathcal{R}$, $N \in \mathbb{N}_{\setminus 0}$	0x6B	RNN

„Divide Immediate Unsigned“. Analog zur Instruktion DIVI mit dem Unterschied, dass X und N vorzeichenlose Werte haben. Insbesondere nimmt N Werte aus dem Intervall $[1, 2^{16} - 1]$.

3.5.18 DIVD

Assemblername	Parameter	Maschinencode	Format
DIVD	$X, Y, Z \in \mathcal{R}$	0x6C	RRR

„Divide direct“. Dividiert ganzzahlig den Inhalt des Registers Y durch den Inhalt des Registers Z und speichert den Quotient in das Register X. Die Spezialregister HI und LO werden nicht verändert. Entspricht den Instruktionen:

```
DIV  Y  Z
COPY X HI
```

Algebraische Schreibweise:

$$X \leftarrow \left\lfloor \frac{Y}{Z} \right\rfloor$$

Beispiel für die Verwendung der Instruktion DIVD:

```
SET  R1 10     # R1 ← 10
SET  R2  3     # R2 ← 3
DIVD R3 R1 R2  # R3 ← ⌊10/3⌋ = 3
MOD  R4 R1 R3  # R4 ← (10 mod 3) = 1
```

3.5.19 MOD

Assemblername	Parameter	Maschinencode	Format
MOD	$X, Y, Z \in \mathcal{R}$	0x70	RRR

Modulo Operation. Berechnet den Rest der Division Y/Z und speichert den Rest in das Register X. Äquivalent zu den Instruktionen:

```
DIVU Y Z
COPY X LO
```

Algebraische Schreibweise:

$$X \leftarrow Y \bmod Z$$

oder

$$X \leftarrow \left(Y - \left\lfloor \frac{Y}{Z} \right\rfloor \cdot Z\right)$$

3.5.20 MODI

Assemblername	Parameter	Maschinencode	Format
MODI	$X, Y \in \mathcal{R}$, $N \in \mathbb{N}$	0x72	RRN

„Modulo Immediate“. Analog zur Instruktion MOD, berechnet MODI den Rest der ganzzahligen Division Y/N und speichert ihn in das Register X. Der Unterschied liegt darin, dass N eine fest angegebene natürliche Zahl ist.

$$X \leftarrow Y \bmod N$$

3.5.21 ABS

Assemblername	Parameter	Maschinencode	Format
ABS	$X, Y \in \mathcal{R}$	0x78	RR0

„Absolute“. Speichert den absoluten Wert des Registers Y in das Register X. Algebraisch ausgedrückt:

$$X \leftarrow \begin{cases} Y & \text{falls } Y \geq 0 \\ (-1) \cdot Y & \text{falls } Y < 0 \end{cases}$$

3.5.22 NEG

Assemblername	Parameter	Maschinencode	Format
NEG	$X, Y \in \mathcal{R}$	0x80	RR0

„Negate“. Wechselt das arithmetische Vorzeichen des Registers Y und speichert das Ergebnis in das Register X. Entspricht der Zweierkomplement Bildung. Algebraische Schreibweise:

$$X \leftarrow \Big((-1) \cdot Y\Big)$$

Um eine bitweise Inversion zu erreichen (Einerkomplement), siehe die Instruktion NOT.

3.5.23 INC

Assemblername	Parameter	Maschinencode	Format
INC	$X \in \mathcal{R}$	0x81	R00

„Increment“. Inkrementiert den Inhalt des Registers X.

$$X \leftarrow (X + 1)$$

3.5.24 DEC

Assemblername	Parameter	Maschinencode	Format
DEC	$X \in \mathcal{R}$	0x82	R00

„Decrement“. Dekrementiert den Inhalt des Registers X.

$$X \leftarrow (X - 1)$$

3.6 Logische Instruktionen

3.6.1 AND

Assemblername	Parameter	Maschinencode	Format
AND	$X, Y, Z \in \mathcal{R}$	0x90	RRR

Berechnet bitweise $Y \wedge Z$ (und-Verknüpfung) und speichert das Ergebnis in das Register X.

$$X \leftarrow (Y \wedge Z)$$

3.6.2 ANDI

Assemblername	Parameter	Maschinencode	Format
ANDI	$X, Y \in \mathcal{R}$, $N \in \mathbb{N}$	0x91	RRN

„And Immediate“. Berechnet bitweise $Y \wedge N$ (und-Verknüpfung) und speichert das Ergebnis in das Register X. N ist dabei eine feste Zahl aus dem Bereich $[0, 255]$. Wird eine größere Zahl angegeben, so wird sie modulo 256 berechnet – mit anderen Worten, nur das letzte Byte zählt. Andere Schreibweise:

$$X \leftarrow (Y \wedge (N \bmod 256))$$

Bemerkung Die natürliche Zahl N wird auf die Länge von 32 Bit mit Nullen verlängert (Links-Verlängerung). Deshalb setzt diese Instruktion alle Bits mit Wertigkeiten von 2^8 bis 2^{31} im Register X auf Null.

Beispiel für die Verwendung von `ANDI`:

```
SET  R1    0x0102
ANDI R2 R1   0x01  # R2 = 0
```

3.6.3 `OR`

Assemblername	Parameter	Maschinencode	Format
`OR`	$X, Y, Z \in \mathcal{R}$	`0x92`	RRR

Berechnet bitweise $Y \vee Z$ (oder-Verknüpfung) und speichert das Ergebnis in das Register X.

$$X \leftarrow (Y \vee Z)$$

3.6.4 `ORI`

Assemblername	Parameter	Maschinencode	Format
`ORI`	$X, Y \in \mathcal{R}$, $N \in \mathbb{N}$	`0x93`	RRN

„Or Immediate“. Berechnet wie `OR` ein bitweises „oder“zwischen Y und N und speichert das Ergebnis in das Register X. Dabei wird die natürliche Zahl $N \in [0, 255]$ auf die Länge von 32 Bit mit Nullen verlängert (Links-Verlängerung). Wird für N einen Wert größer als 255 angegeben, so wird er modulo 256 berechnet.

$$X \leftarrow (Y \vee (N \bmod 256))$$

3.6.5 XOR

Assemblername	Parameter	Maschinencode	Format
XOR	$X, Y, Z \in \mathcal{R}$	0x94	RRR

Berechnet bitweise $Y \oplus Z$ (xor-Verknüpfung) und speichert das Ergebnis in das Register X.

$$X \leftarrow (Y \oplus Z)$$

3.6.6 XORI

Assemblername	Parameter	Maschinencode	Format
XORI	$X, Y \in \mathcal{R}$, $N \in \mathbb{N}$	0x95	RRN

„Xor Immediate“. Analog zur Instruktion XOR mit dem Unterschied, dass N eine direkt angegebene Zahl aus dem Intervall $[0, 255]$ ist.

$$X \leftarrow (Y \oplus (N \bmod 256))$$

3.6.7 NOT

Assemblername	Parameter	Maschinencode	Format
NOT	$X, Y \in \mathcal{R}$	0x96	RR0

Invertiert alle Bits aus dem Register Y und speichert das Ergebnis in das Register X. Entspricht der Einerkomplement-Bildung.

$$X \leftarrow \overline{Y}$$

Beispiel Der folgende Code invertiert alle Bits aus dem Register $R2$ und speichert das Ergebnis in das Register $R1$.

```
NOT R1 R2 # R1 ← ¬R2
```

3.6.8 NOTI

Assemblername	Parameter	Maschinencode	Format
NOTI	$X \in \mathcal{R}$, $N \in \mathbb{N}$	0x97	RNN

„Not Immediate“. Wie die Instruktion NOT aber N ist eine natürliche Zahl aus dem Intervall $[0, 2^{15} - 1]$. Diese konstante Zahl nach links auf die Länge von 32 Bit mit Nullen verlängert.

Beispiel Der folgende Code invertiert alle Bits der Zahl 5 und speichert das Ergebnis in das Register $R1$.

```
NOT R1 5 # R1 ← 5̄
```

3.6.9 NAND

Assemblername	Parameter	Maschinencode	Format
NAND	$X, Y, Z \in \mathcal{R}$	0x98	RRR

Berechnet $Y \overline{\wedge} Z$ (nand-Verknüpfung) und speichert das Ergebnis in das Register X. Gemäß dem Format RRR sind X, Y und Z Register. Algebraische Schreibweise:

$$X \leftarrow (Y \overline{\wedge} Z)$$

oder

$$X \leftarrow \overline{(Y \wedge Z)}$$

3.6.10 NANDI

Assemblername	Parameter	Maschinencode	Format
NANDI	$X, Y \in \mathcal{R}$, $N \in \mathbb{N}$	0x99	RRN

„Nand Immediate“. Berechnet eine nand-Verknüpfung zwischen dem Inhalt des Registers Y und der konstanten Zahl N und speichert das Ergebnis in das Register X. $N \in [0, 255]$.

$$X \leftarrow \big(Y \overline{\wedge} (N \bmod 256)\big)$$

3.6.11 NOR

Assemblername	Parameter	Maschinencode	Format
NOR	$X, Y, Z \in \mathcal{R}$	0x9A	RRR

Verknüpft bitweise die Inhalte der Register Y und Z mit der nor-Operation und speichert das Ergebnis in das Register X. „nor“ ist die Negation von „or“.

$$X \leftarrow (Y \triangledown Z)$$

```
NOR R1 R2 R3 # R1 ← ¬(R2 ∨ R3)
```

3.6.12 NORI

Assemblername	Parameter	Maschinencode	Format
NORI	$X, Y \in \mathcal{R}$, $N \in \mathbb{N}$	0x9B	RRN

„Nor Immediate“. Analog zur Instruktion NOR mit dem Unterschied, dass N eine konstante Zahl aus dem Bereich $[0, 255]$ ist. Diese Zahl wird modulo 256 berechnet und auf der linken Seite auf die Länge von 32 Bit mit Nullen aufgefüllt.

3.6.13 SHL

Assemblername	Parameter	Maschinencode	Format
SHL	$X, Y, Z \in \mathcal{R}$	0xA0	RRR

„Shift Left“. Verschiebt die Bits aus dem Register Y Z Stellen nach links und speichert das Ergebnis in das Register X. Auf der rechten Seite wird X mit Nullen aufgefüllt.

Andere Schreibweise:

$$X \leftarrow Y << Z$$

oder

$$X \leftarrow (Y \cdot 2^Z) \bmod 2^{32}$$

3.6.14 SHLI

Assemblername	Parameter	Maschinencode	Format
SHLI	$X, Y \in \mathcal{R}$, $N \in \mathbb{N}$	0xA1	RRN

„Shift Left Immediate“. Verschiebt die Bits im Register Y N Stellen nach links und speichert das Ergebnis in das Register X. N ist eine positive Zahl im Bereich $[0, 255]$. Anstelle der nach links verschobenen Bits werden Nullen aufgefüllt.

3.6.15 SHR

Assemblername	Parameter	Maschinencode	Format
SHR	$X, Y, Z \in \mathcal{R}$	0xA2	RRR

„Shift Right“. Verschiebt die Bits aus dem Register Y Z Stellen nach rechts und speichert das Ergebnis in das Register X. Anstelle der verschobenen Bits werden im Register X auf der linken Seite Nullen aufgefüllt. Gemäß dem Instruktionsformat RRR sind X, Y und Z Register. Alle Register-Inhalte werden vorzeichenlos interpretiert.

Bemerkung Wird in dem Register Y eine negative Zahl gespeichert, so löscht diese Verschiebung das Vorzeichen, da auf der linken Seite Nullen aufgefüllt werden. Um das Vorzeichen zu behalten, sollte man die Instruktion SHRA verwenden.

Beispiel Der folgende Code verschiebt die Bits im $R1$ eine Stelle nach rechts. Es wird praktisch $R3 \leftarrow (5 \div 2)$ berechnet.

```
SET   R1  5        # R1 ← 5
SET   R2  1        # R2 ← 1
SHR   R3  R1  R2   # R3 ← 2
```

3.6.16 SHRI

Assemblername	Parameter	Maschinencode	Format
SHRI	$X, Y \in \mathcal{R}$, $N \in \mathbb{N}$	0xA3	RRN

„Shift Right Immediate“. Das Bitmuster im Register Y wird N Stellen nach rechts verschoben und das Ergebnis in das Register X gespeichert. Dabei ist N eine positive natürliche Zahl aus dem Intervall $[0, 255]$. Auf der linken Seite werden die versetzten Bits mit Nullen ersetzt. Siehe auch SHRAI.

3.6.17 SHRA

Assemblername	Parameter	Maschinencode	Format
SHRA	$X, Y, Z \in \mathcal{R}$	0xA4	RRR

„Shift Right Arithmetical“. Funktioniert wie die Instruktion SHR mit dem Unterschied, dass auf der linken Seite nicht mit Nullen, sondern mit dem ersten (höchstwertigen) Bit aufgefüllt wird. Dies führt dazu, dass das Vorzeichenbit in Y erhalten bleibt.

3.6.18 SHRAI

Assemblername	Parameter	Maschinencode	Format
SHRAI	$X, Y \in \mathcal{R}$, $N \in \mathbb{N}$	0xA5	RRN

„Shift Right Arithmetical Immediate“. Wie SHRA, N ist aber eine natürliche Zahl aus dem Intervall $[0, 255]$.

3.6.19 ROTL

Assemblername	Parameter	Maschinencode	Format
ROTL	$X, Y, Z \in \mathcal{R}$	0xA8	RRR

„Rotate Left“. Die Bits aus dem Register Y werden so viele Stellen nach links „rotiert“ wie der Inhalt des Registers Z und das Ergebnis in das Register X gespeichert.

$$X \leftarrow Y \circlearrowleft Z$$

Die Rotation bedeutet, dass die Bits nach links verschoben werden und diejenigen Bits, die über die linke Grenze hinausfallen auf der rechten Seite wieder eingefügt werden. Dies bedeutet, dass mit jedem verschobenen Bit, ein Bit ganz links (Wertigkeit 2^{31}) fällt aus und wird wieder ganz rechts mit Wertigkeit 2^0 eingefügt. Nach 32 Rotationen eine Stelle nach links ist das ursprüngliche Bitmuster wieder hergestellt.

Beispiel Der folgende Code zeigt ein Beispiel für die Verwendung dieser Instruktion.

```
SET  R1 2
SET  R2 1
ROTL R3 R1 R2       # links-rotation von R1 eine Stelle
                    # R3 = 4
SET  R1 0x80000000 # nur das linke Bit gesetzt
ROTL R4 R1 R2       # R4 = 0x01
```

3.6.20 ROTLI

Assemblername	Parameter	Maschinencode	Format
ROTLI	$X, Y \in \mathcal{R}$, $N \in \mathbb{N}$	0xA9	RRN

„Rotate Left Immediate“. Wie ROTL aber N ist eine konstante Zahl aus dem Intervall $[0, 255]$.

Beispiel Verwendung:

```
ROTL R4 R1  6 # rotiere die Bits in R1 6 stellen
ROTL R4 R1 -6 # Fehler! da −6 ∉ ℕ
```

3.6.21 ROTR

Assemblername	Parameter	Maschinencode	Format
ROTR	$X, Y, Z \in \mathcal{R}$	0xAA	RRR

„Rotate Right“. Funktioniert genauso wie die Instruktion ROTL mit dem Unterschied, dass die Rotationen (Verschiebungen) nach rechts geführt werden.

$$X \leftarrow Y \circlearrowright Z$$

3.6.22 ROTRI

Assemblername	Parameter	Maschinencode	Format
ROTRI	$X, Y \in \mathcal{R}$, $N \in \mathbb{N}$	0xAB	RRN

„Rotate Right Immediate“. Funktioniert genauso wie ROTLI, nur die Rotationen sind nach rechts geführt.

3.7 Vergleichsinstruktionen

Die Vergleichsinstruktionen vergleichen den Inhalt eines Registers mit dem Inhalt eines anderen Registers oder mit einer angegebenen ganzen Zahl. Das Ergebnis der Vergleichsinstruktion wird in das Register CMP gespeichert. Dieses Ergebnis ist -1, 0 oder $+1$ und wird folgenderweise berechnet: werden zwei Werte x und y vergliechen, so wird das Register CMP wie folgt gesetzt:

$$CMP \leftarrow \begin{cases} -1 & \text{falls } x < y \\ \pm 0 & \text{falls } x = y \\ +1 & \text{falls } x > y \end{cases}$$

3.7.1 CMP

Assemblername	Parameter	Maschinencode	Format
CMP	$X, Y \in \mathcal{R}$	0xB0	RR0

„Compare“. Vergleicht die Inhalte der Register X und Y.

3.7.2 CMPU

Assemblername	Parameter	Maschinencode	Format
CMPU	$X, Y \in \mathcal{R}$	0xB1	RR0

„Compare Unsigned“. Vergleicht analog zu CMP – aber vorzeichenlos – X mit Y (die Inhalte der Register X und Y werden als vorzeichenlose Werte ausgewertet).

3.7.3 CMPI

Assemblername	Parameter	Maschinencode	Format
CMPI	$X \in \mathcal{R}$, $N \in \mathbb{Z}$	0xB2	RNN

„Compare Immediate“. Vergleicht X mit angegebenen festen Wert N. Dabei nimmt N Werte aus dem Intervall $[-2^{15}, 2^{15} - 1]$, entsprechend dem Datentyp „Half“.

3.7.4 CMPIU

Assemblername	Parameter	Maschinencode	Format
CMPIU	$X \in \mathcal{R}$, $N \in \mathbb{N}$	0xB3	RNN

„Compare Immediate Unsigned“. Vergleicht, wie CMPI auch, X mit angegebenen Wert N und setzt entsprechend das Register CMP. Der Unterschied ist, dass X und N als vorzeichenlos betrachtet werden. N hat den Datentyp „Half“.

3.8 Übersprungsbefehle

Alle Sprungbefehle, außer dem GO Befehl, veranlassen einen relativen Übersprung im Programmcode – relativ im Sinne, dass die Parameter der Übersprungbefehle einen

ganzzahligen Versatz zur aktuellen Programmadresse angeben. Dabei bedeutet der Versatz, wieviele Instruktionen müssen bis zur Zielinstruktion übersprungen werden. Z.B. die Instruktion

```
JMP 2
```

bedeutet „Überspringe 2 Instruktionen und fahre mit der 3. fort“. Die Instruktion

```
BE 5
```

bedeutet „Falls das Register CMP den Wert 0 hat, überspringe 5 Instruktionen und fahre mit der 6. fort“.

Der Versatz wird nicht in Bytes angegeben, sondern in Instruktionen – wobei eine Instruktion der UMach Maschine 4 Bytes beträgt.

Die Übersprungbefehle bauen auf die Vergleichsbefehle auf: jeder Übersprungbefehl (außer JMP und GO) untersuchen das Spezialregister CMP und verzweigen die Programmausführung anhand seines Wertes.

3.8.1 BE

Assemblername	Parameter	Maschinencode	Format
BE	$N \in \mathbb{Z}$	0xD0	NNN

„Branch Equal“. Wenn das Register CMP den Wert 0 hat, wird über N Instruktionen vorwärts oder rückwärts gesprungen. Ein negatives N bedeutet einen Sprung rückwärts, ein positives N bewirkt einen Sprung vorwärts. Der Sprung wird dadurch erreicht, dass das Register PC gemäß der folgenden Formel modifiziert wird:

$$PC \leftarrow PC + 4 \cdot N$$

Die Multiplikation mit 4 wird deshalbt ausgeführt, weil ein Befehl immer aus 4 Bytes besteht, sodass der Adressoffset zwischen zwei Befehlen immer 4 ist. Somit ist N die Anzahl der zu überspringenden Befehle bis zur nächsten Instruktion. Der dazu benötigte Offset N wird vom Assembler automatisch berechnet.

Bemerkung Die UMach Maschine erhöht den Programmcounter (Register PC) nach der Ausführung jeder Instruktion (siehe 2.1.2, Seite 8). Die Modifizierung des PC-Registers durch den BE Befehl wirkt sich nicht störend auf die automatische Inkrementierung des Programmcounters.

Beispiel Der folgende Code lädt zwei Bytes in die Register $R2$ und $R3$ und addiert diese arithmetisch, falls sie ungleiche Werte haben. Sind die Werte gleich, wird stattdessen der Inhalt von $R2$ mit 2 multipliziert. Ein mögliches Überlaufen wird nicht berücksichtigt.

```
  SET   R1 100     # R1 ← 100
  LBUI  R2 R1  0   # Lade Byte von Adresse 100 nach R2
  LBUI  R3 R1  1   # Lade Byte von Adresse 100+1 nach R3
  CMPU  R2 R3      # Vergleiche Inhalt von R2 und R3
                   # Ergebnis geht ins CMP
  #BE    equal     # Asm Schreibweise
  BE    3          # Wenn CMP gleich 0 ist, überspringe
                   # die folgenden 3 Instruktionen
                   # (gehe zum label equal)
                   # N ist in diesem Fall 3
  ADDU  R2 R2 R3   # Addiere Inhalt von R2 und R3
  SBUI  R2 R1 0    # Speichere R2 nach Adresse 100
  #JMP   finish    # Asm Schreibweise
  JMP   2          # Überspringe die folgenden 2 Befehle,
                   # N von JMP ist 2.
equal:
  MULIU R2 2       # Multipliziere Inhalt von R2 mit 2
  SBUI  LO R1 0    # Speichere Inhalt von LO nach Adresse in R1
finish:
```

3.8.2 BNE

Assemblername	Parameter	Maschinencode	Format
BNE	$N \in \mathbb{Z}$	0xD1	NNN

„Branch Not Equal “. Entspricht dem Verhalten von BE mit dem Unterschied, dass der angegebene Übersprung ausgeführt wird, wenn CMP nicht 0 ist.

3.8.3 BL

Assemblername	Parameter	Maschinencode	Format
BL	$N \in \mathbb{Z}$	0xD2	NNN

„Branch Less“. Überspringt N Instruktionen, wenn der Inhalt von CMP kleiner 0 ist.

3.8.4 BLE

Assemblername	Parameter	Maschinencode	Format
BLE	$N \in \mathbb{Z}$	0xD3	NNN

„Branch Less Equal“. Überspringt N Instruktionen, wenn der Inhalt von CMP kleiner oder gleich 0 ist.

3.8.5 BG

Assemblername	Parameter	Maschinencode	Format
BG	$N \in \mathbb{Z}$	0xD4	NNN

„Branch Greater“. Überspringt N Instruktionen, wenn der Inhalt von CMP größer als 0 ist.

3.8.6 BGE

Assemblername	Parameter	Maschinencode	Format
BGE	$N \in \mathbb{Z}$	0xD5	NNN

„Branch Greater Equal“. Überspringt N Instruktionen, wenn der Inhalt von CMP größer oder gleich 0 ist.

3.8.7 JMP

Assemblername	Parameter	Maschinencode	Format
JMP	$N \in \mathbb{N}$	0xD8	NNN

„Jump“. Überspringt N Instruktionen.

3.8.8 JMPR

Assemblername	Parameter	Maschinencode	Format
JMPR	$X \in \mathcal{R}$	0xD9	R00

„Jump Register“. Überspringt X Instruktionen. X ist ein Register.

3.8.9 GO

Assemblername	Parameter	Maschinencode	Format
GO	$N \in \mathbb{N}$	0xDF	NNN

Setzt PC auf die angegebene absolute Adresse. Hierbei ist zu beachten, dass nicht in die Mitte eines Befehles gesprungen wird. Dies zu gewährleisten liegt in der Verantwortung des Programmierers.

3.9 Unterprogramminstruktionen

3.9.1 CALL

Assemblername	Parameter	Maschinencode	Format
CALL	$N \in \mathbb{N}$	0xE0	NNN

3.9.2 RET

Assemblername	Parameter	Maschinencode	Format
RET	keine	0xE1	000

3.10 Systeminstruktionen

3.10.1 WAKE

Assemblername	Parameter	Maschinencode	Format
WAKE	$N \in \mathbb{N}$	0xF0	NNN

Ruft das Betriebssystem auf, eine Aktion zu unternehmen. Die Aktion wird als numerischer Code angegeben und ist systemspezifisch. Vergleichbar mit einem „syscall“.

3.10.2 KILL

Assemblername	Parameter	Maschinencode	Format
KILL	keine	0xF8	000

„Kill OS“. Schaltet die Auswirkung des WAKE-Befehls aus. Damit wird praktisch das Betriebssystem ausgeschaltet.

Tabellenverzeichnis

Glossar

A | B | I | M | R | S

A

Adressdekoder
: Das Herz des Bussystems. Wählt anhand einer eingegebenen Adresse einen Bus-Port, an den ein Befehl weitergeleitet wird. Funktioniert wie eine Art Demultiplexer.

Adressierungsart
: Die Art, wie eine Instruktion die Umach Maschine dazu veranlasst, einen Speicherbereich zu adressieren. Siehe auch Abschnitt 2.3.1.

Assemblername
: Der Name eines Registers oder eines Befehls, so wie er in einem textuellen Programm (ASCII) verwendet wird. $R1$, $R2$, ADD sind Assemblernamen von Registern und Befehlen.

B

Befehl
: Die ersten 8 Bits in einer Instruktion. Operation code.

Befehlsraum
: Die Anzahl der möglichen Befehle, abhängig von der Befehlsbreite. Beträgt die Befehlsbreite 8 Bit, so ist der Befehlsraum $2^8 = 256$.

Betriebsmodus
: Die Art, wie die UMach Maschine die einzelnen Instruktionen abarbeitet. Siehe auch 2.1.1.

Bussystem
: Die Mainboard.

Byte
: Eine Reihe oder Gruppe von 8 Bit.

I

Instruktion
: Eine Anweisung an die UMach VM etwas zu tun. Eine Instruktion besteht aus einem Befehl (Operation Code) und eventuellen Argumenten.

Instruktionsformat
: Beschreibt die Struktur einer Instruktion auf Byte-Ebene und zwar es gibt an, ob ein Byte als eine Registerangabe oder als reine numerische Angabe zu interpretieren ist. Siehe 3.1.

Instruktionssatz
: Die Menge aller Instruktionen, die von der UMach Maschine ausgeführt werden können.

M

Maschinenname
: Der Name eines Registers oder eines Befehls, so wie er im Maschinencode erscheint. `0x01`, `0x02`, `0x40` sind Maschinennamen von Registern und Befehlen.

R

Register
: Eine sich im Prozessor befindende Speichereinheit. Das Register ist dem Programmierer sichbar und kann mit Werten geladen werden. Siehe Abschnitt 2.2, Seite 10.

S

Speicher
: Baukomponente, die am Bussystem angeschlossen ist und die eine feste Anzahl von Bytes für die Laufzeit der Maschine aufnehmen kann.

Index