

# UMach Spezifikation

29. April 2012

# Inhaltsverzeichnis

<b>1</b>	<b>Einführung</b>	<b>6</b>
1.1	Anwendungsbeispiel . . . . .	6
<b>2</b>	<b>Organisation der UMach VM</b>	<b>7</b>
2.1	Aufbau . . . . .	7
2.1.1	Betriebsmodi . . . . .	7
2.2	Register . . . . .	8
2.2.1	Allzweckregister . . . . .	8
2.2.2	Spezialregister . . . . .	9
2.3	Der Speicher . . . . .	10
2.3.1	Adressierungsarten . . . . .	10
2.3.2	Datentypen . . . . .	12
2.3.3	Der Stack . . . . .	12
2.4	Die Peripherie . . . . .	12
2.4.1	Speicherbasierte Kommunikation . . . . .	12
2.4.2	Bus-Ports . . . . .	15
2.4.3	Adressraumverteilung . . . . .	15
<b>3</b>	<b>Instruktionssatz</b>	<b>18</b>
3.1	Instruktionsformate . . . . .	18
3.1.1	000 . . . . .	19
3.1.2	NNN . . . . .	19
3.1.3	R00 . . . . .	19
3.1.4	RNN . . . . .	20
3.1.5	RR0 . . . . .	20
3.1.6	RRN . . . . .	20
3.1.7	RRR . . . . .	21
3.1.8	Zusammenfassung . . . . .	21
3.2	Verteilung des Befehlsraums . . . . .	21
3.3	Kontrollinstruktionen . . . . .	25
3.3.1	NOP . . . . .	25
3.3.2	RST . . . . .	25
3.3.3	CRM . . . . .	25
3.3.4	CSM . . . . .	26
3.3.5	DIE . . . . .	26

3.3.6	RSR . . . . .	26
3.3.7	AUTSM . . . . .	26
3.3.8	SOCL . . . . .	27
3.3.9	HATE . . . . .	27
3.3.10	TRST . . . . .	27
3.3.11	ZMB . . . . .	27
3.3.12	ALIV . . . . .	28
3.4	Lade- und Speicherbefehle . . . . .	28
3.4.1	SET . . . . .	28
3.4.2	SETU . . . . .	28
3.4.3	COPY . . . . .	29
3.4.4	MOVE . . . . .	29
3.4.5	LB . . . . .	30
3.4.6	LBU . . . . .	30
3.4.7	LH . . . . .	31
3.4.8	LHU . . . . .	31
3.4.9	LW . . . . .	32
3.4.10	LWU . . . . .	32
3.4.11	LBI . . . . .	32
3.4.12	LBUI . . . . .	33
3.4.13	LHI . . . . .	33
3.4.14	LHUI . . . . .	33
3.4.15	LWI . . . . .	34
3.4.16	LWUI . . . . .	34
3.4.17	SB . . . . .	34
3.4.18	SBU . . . . .	35
3.4.19	SH . . . . .	35
3.4.20	SHU . . . . .	36
3.4.21	SW . . . . .	36
3.4.22	SWU . . . . .	37
3.4.23	SBI . . . . .	37
3.4.24	SBUI . . . . .	38
3.4.25	SHI . . . . .	38
3.4.26	SHUI . . . . .	38
3.4.27	SWI . . . . .	39
3.4.28	SWUI . . . . .	39
3.4.29	PUSHB . . . . .	40
3.4.30	PUSHH . . . . .	40
3.4.31	PUSH . . . . .	41
3.4.32	POPB . . . . .	41
3.4.33	POPH . . . . .	42
3.4.34	POP . . . . .	43
3.5	Arithmetische Instruktionen . . . . .	44
3.5.1	ADD . . . . .	44

3.5.2	ADDU	44
3.5.3	ADDI	45
3.5.4	ADDIU	45
3.5.5	SUB	45
3.5.6	SUBU	46
3.5.7	SUBI	46
3.5.8	SUBIU	46
3.5.9	MUL	47
3.5.10	MULU	47
3.5.11	MULI	47
3.5.12	MULIU	48
3.5.13	DIV	48
3.5.14	DIVU	48
3.5.15	DIVI	48
3.5.16	DIVIU	48
3.5.17	MOD	49
3.5.18	MODI	49
3.5.19	ABS	49
3.5.20	NEG	49
3.5.21	INC	50
3.5.22	DEC	50
3.6	Logische Instruktionen	50
3.6.1	AND	50
3.6.2	ANDI	50
3.6.3	OR	50
3.6.4	ORI	51
3.6.5	XOR	51
3.6.6	XORI	51
3.6.7	NOT	51
3.6.8	NOTI	51
3.6.9	NAND	52
3.6.10	NANDI	52
3.6.11	NOR	52
3.6.12	NORI	52
3.6.13	SHL	52
3.6.14	SHLI	53
3.6.15	SHR	53
3.6.16	SHRI	53
3.6.17	SHRA	53
3.6.18	SHRAI	54
3.6.19	ROTL	54
3.6.20	ROTLI	54
3.6.21	ROTR	54
3.6.22	ROTRI	55

3.7	Vergleichsinstruktionen . . . . .	55
3.7.1	CMP . . . . .	55
3.7.2	CMPU . . . . .	55
3.7.3	CMPI . . . . .	55
3.7.4	CMPIU . . . . .	56
3.8	Sprungbefehle . . . . .	56
3.8.1	BZ . . . . .	56
3.8.2	BNZ . . . . .	56
3.8.3	BLZ . . . . .	57
3.8.4	BLEZ . . . . .	57
3.8.5	BGZ . . . . .	57
3.8.6	BGEZ . . . . .	57
3.8.7	BEI . . . . .	58
3.8.8	BNI . . . . .	58
3.8.9	BLI . . . . .	58
3.8.10	BLEI . . . . .	58
3.8.11	BGI . . . . .	59
3.8.12	BGEI . . . . .	59
3.8.13	JMP . . . . .	59
3.8.14	GO . . . . .	59
3.9	Unterprogramminstruktionen . . . . .	60
3.9.1	CALL . . . . .	60
3.9.2	RET . . . . .	60
3.10	Systeminstruktionen . . . . .	60
3.10.1	WAKE . . . . .	60
3.10.2	KILL . . . . .	60
<b>Tabellenverzeichnis</b>		<b>61</b>
<b>Glossar</b>		<b>62</b>
<b>Index</b>		<b>64</b>

# 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, u.s.w. 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 die anschließend kompiliert werden. Die kompilierte Dateien (Maschinen-Code) wird von der virtuellen Maschine ausgeführt.

## 1.1 Anwendungsbeispiel

```
| LOAD R1 90  
| LOAD R2 09  
| REV R3 R1
```

## 2 Organisation der UMach VM

### 2.1 Aufbau

Dieser Abschnitt beschreibt den Aufbau der UMach virtuellen Maschine. Die virtuelle Maschine besteht aus internen und aus externen Komponenten. Dabei sind die externen Komponenten nicht wesentlich für die Funktionsfähigkeit der gesamten Maschine, d.h. die Maschine kann im Prinzip auch ohne die externen Komponenten funktionieren – in diesem Fall fehlt ihr eine Menge von Funktionen.

**Interne Komponenten** sind diejenigen Komponenten, die für die Funktionsfähigkeit der UMach Maschine wesentlich sind:

1. Recheneinheit
2. Logische Einheit
3. Register

**Externe Komponenten**

1. Anbindung an einem I/O-Port

#### 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

**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.

**Einzelstschrittmodus** 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.

## 2.2 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  $R1, R2 \dots$  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	0x40	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 Maschinenamen der Spezialregister setzen die Nummerierung der Allzweckregister zwar fort, die Assemblernamen aber nicht: es gibt kein Register  $R33$ . Die Tabelle 2.1 auf Seite 9 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

IP	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 CIN 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.
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.
CIN	0x37	„Current Instruction“. Enthält die gerade ausgeführte Instruktion. Schreibgeschützt.
NIN	0x38	„Next Instruction“. Enthält die nächste Instruktion. Während der Ausführung einer Instruktion, zeigt der Register IP 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.
ERRM	0x3B	„Error message“. Enthält zusätzliche Informationen zu den Fehler im signalisiert im Register ERR.
LO	0x3C	„Low“. Enthält die niedrigstwertigen 32 Bits des Ergebnisses einer Multiplikation. Speichert zusammen mit dem Register HI das volle Ergebnis der Multiplikation.
HI	0x3D	„High“. Enthält die höchstwertigen 32 Bits des Ergebnisses einer Multiplikation. Speichert zusammen mit dem Register LO das volle Ergebnis der Multiplikation.
ZERO	0x3E	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	$R$	$B$	$I$	$R \leftarrow \text{mem}(B + I)$
Speicherbefehl	$R$	$B$	$I$	$R \rightarrow \text{mem}(B + I)$

Die fünfte Spalte gibt jeweils den äquivalenten algebraischen Ausdruck wieder.  $\text{mem}(x)$  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.

### 2.3.2 Datentypen

Die UMach Maschine kennt 3 Datentypen:

1. Byte (8 Bit lang)
2. Half (2 Byte)
3. Word (4 Byte)

### 2.3.3 Der Stack

## 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 [2.1](#) auf der Seite [13](#).

### 2.4.1 Speicherbasierte Kommunikation

Die UMach Maschine kommuniziert nicht direkt mit den am Bus angeschlossen 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<sup>1</sup>. 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

---

<sup>1</sup>Wir schließen nicht aus, dass unterschiedliche Realisierungen der UMach Maschine unterschiedliche Ports haben.

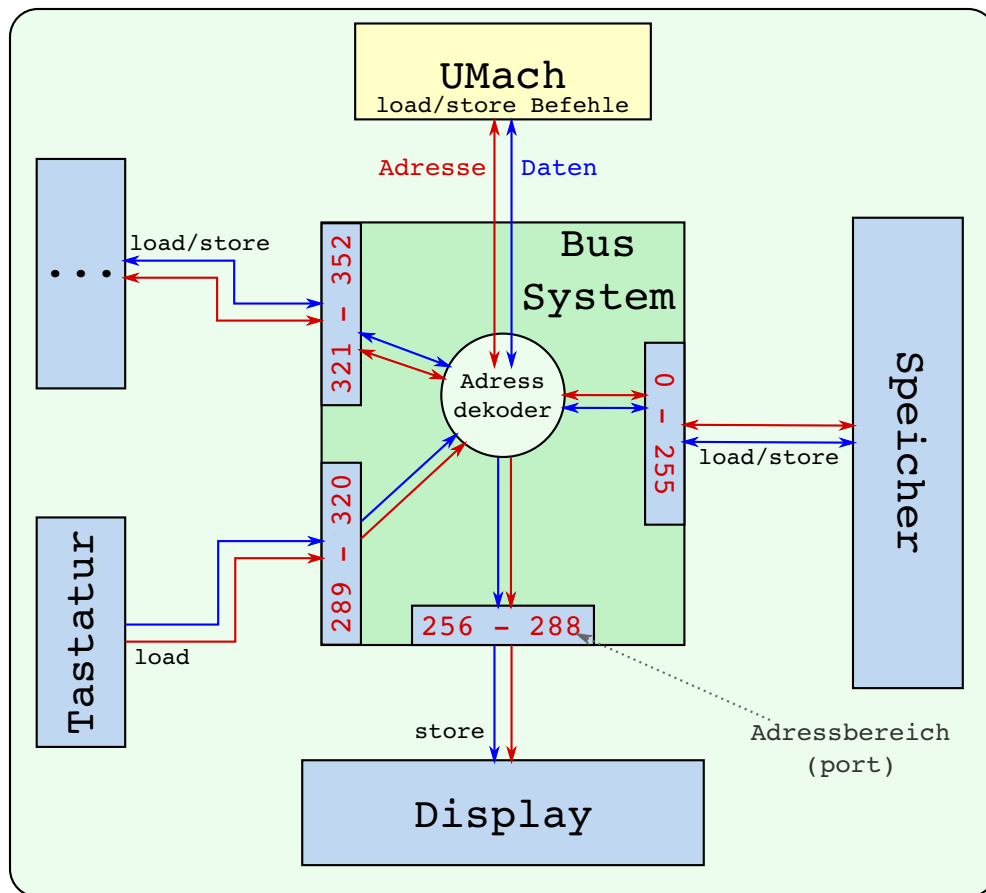


Abbildung 2.1: Das UMach Bussystem verwendet eine einfache speicherbasierte Adressierung der peripherischen Komponenten, die an „Memory Mapped I/O“ angelehnt ist. Jeder peripherischen Komponente wird vom Bussystem beim Hochfahren der Maschine ein eindeutigen Adressbereich zugewiesen. Die „load/store“ 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 einzelnen Ports auf Adressbereiche enthält, ist nicht dargestellt.

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 UMac 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 innerhalb 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 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 15. Nach der Verteilung kann das Bussystem jede 32-Bit Adresse einem bestimmten Port zuweisen.

Die Abbildung 2.1 auf der Seite 13 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 UMac 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 28). 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 2.1 auf der Seite 13. 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 2.1 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 15). 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<sup>2</sup>. 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:

$$-2306 \leftarrow \dots \quad \left| \begin{array}{c|c|c} 1 \text{ Byte} & 4 \text{ Byte} & 4 \text{ Byte} \\ \hline \text{Typ} & \text{Länge} & \text{Startadresse} \end{array} \right| \quad \dots \rightarrow -1$$

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 15.

Siehe Abbildung 2.2 auf der Seite 17.

**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 28 beschrieben.

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

<sup>2</sup>Die Zahl  $-2306$  ergibt sich aus 256 mögliche Einträge mal 9 Byte plus 1.



```

    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'

```

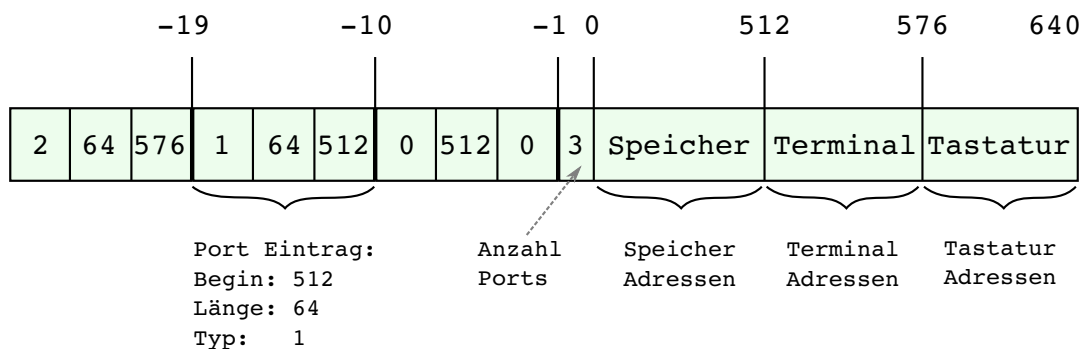


Abbildung 2.2: Adressraumverteilung im Bussystem

## 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 15 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 wiedergegeben.

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 letzten 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 Register  $R_1$  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 - BF	Vergleichsbefehle
C0 - DF	Sprungbefehle
E0 - EF	Unterprogrammbefehle
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 UMa-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 Betriebssystems 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 22 angegeben.

Die Tabelle 3.2 auf der Seite 23 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
	HATE	TRST	ZMB	ALIV				
1	SET	SETU						
	COPY		MOVE					
2	LB	LBU	LH	LHU	LW	LWU		
	LBI	LBUI	LHI	LHUI	LWI	LWUI		
3	SB	SBU	SH	SHU	SW	SWU		
	SBI	SBUI	SHI	SHUI	SWI	SWUI		
4	PUSHB	PUSHH	PUSH					
	POPB	POPH	POP					
5	ADD	ADDU	ADDI	ADDIU				
	SUB	SUBU	SUBI	SUBIU				
6	MUL	MULU	MULI	MULIU				
	DIV	DIVU	DIVI	DIVIU				
7	MOD		MODI					
	ABS							
8	NEG	INC	DEC					
9	AND	ANDI	OR	ORI	XOR	XORI	NOT	NOTI
	NAND	NANDI	NOR	NORI				
A	SHL	SHLI	SHR	SHRI	SHRA	SHRAI		
	ROTL	ROTLI	ROTR	ROTRI				
B	CMP	CMPU	CMPI	CMPIU				
C	BZ	BNZ	BLZ	BLEZ	BGZ	BGEZ		
	BEI	BNI	BLI	BLEI	BGI	BGEI		
D								
	JMP							GO
E	CALL	RET						
F	WAKE							
	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 18 nachgeschlagen werden.

**Zur Notation** Mit  $\mathcal{R}$  wird die Menge aller Register gekennzeichnet<sup>1</sup>. 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: $0, 1, 2, \dots$
$\mathbb{Z}$	Menge aller ganzen Zahlen: $\dots, -2, -1, 0, 1, 2, \dots$
$N \in \mathbb{N}$	$N$ 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 $mem(n)$

<sup>1</sup>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 Register außer dem Befehlszähler 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.

### 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 = 216 ≡ 0 mod 216
    SET R4 70000 # R3 ← 4464 = 70000 mod 216
    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 ein 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.

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.

```
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:

$$X \leftarrow (X \div 2^8) \cdot 2^8 + (\text{mem}(Y + Z) \bmod 2^8)$$

Äquivalenter C Code:

```

x = (x & 0xFFFFF00) | (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	<a href="#">RRR</a>

„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	<a href="#">RRR</a>

„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 #  $R3 \leftarrow mem_4(R1 + R2) = 66051$ 
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 den 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

$$X \rightarrow \text{mem}_1(Y + Z)$$

$\text{mem}_1(x)$  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

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

**Beispiel** Dieses Programm speichert 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

Assemblernamen	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

Assemblernamen	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  $X + Z + 3$ .

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

**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 ganzzahlige 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

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

```

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

$$SP \leftarrow SP - 1$$

$$X \rightarrow mem_1(SP)$$

**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

$$SP \leftarrow SP - 2$$

$$X \rightarrow mem_2(SP)$$

**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	<code>mem[SP + 0]</code>	<code>mem[SP + 1]</code>	<code>mem[SP + 2]</code>	<code>mem[SP + 3]</code>

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:

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

**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

Assemblernamen	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:

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

**Beispiel** Angenommen, die ersten 4 Bytes vom Stack-Bereich 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

Assemblernamen	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	$\text{mem}[SP + 0]$	$\text{mem}[SP + 1]$	$\text{mem}[SP + 2]$	$\text{mem}[SP + 3]$

Diese Instruktion kann algebraisch so ausgedrückt werden:

$$\begin{aligned}
 X &\leftarrow \text{mem}_4(SP) \\
 SP &\leftarrow SP + 4
 \end{aligned}$$

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-Bereich 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 + 231) = 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. 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 9.

**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
```

### 3.5.10 MULU

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

„Multiply Unsigned“.

### 3.5.11 MULI

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

„Multiply Immediate“.

**3.5.12 MULIU**

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

„Multiply Immediate Unsigned“.

**3.5.13 DIV**

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

**3.5.14 DIVU**

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

„Divide Unsigned“.

**3.5.15 DIVI**

Assemblername	Parameter	Maschinencode	Format
DIVI	$X, Y \in \mathcal{R}, N \in \mathbb{Z}$	0x6A	RRN

„Divide Immediate“.

**3.5.16 DIVIU**

Assemblername	Parameter	Maschinencode	Format
DIVIU	$X, Y \in \mathcal{R}, N \in \mathbb{N}$	0x6B	RRN



„Divide Immediate Unsigned“.

### 3.5.17 MOD

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

Modulo.

### 3.5.18 MODI

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

Modulo Immediate.

### 3.5.19 ABS

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

„Absolute“. Speichert den absoluten Wert des Registers  $Y$  in das Register  $X$ .

### 3.5.20 NEG

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

„Negate“. Wechselt das Vorzeichen des Registers  $Y$  und speichert das Ergebnis in das Register  $X$ .

**3.5.21 INC**

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

„Increment“. Inkrementiert den Inhalt des Registers  $X$ .

**3.5.22 DEC**

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

„Decrement“. Dekrementiert den Inhalt des Registers  $X$ .

**3.6 Logische Instruktionen****3.6.1 AND**

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

**3.6.2 ANDI**

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

**3.6.3 OR**

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

**3.6.4 ORI**

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

**3.6.5 XOR**

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

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

**3.6.6 XORI**

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

**3.6.7 NOT**

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

$$X \leftarrow \bar{Y}$$

| NOT R1 R2

**3.6.8 NOTI**

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

| NOT R1 5

**3.6.9 NAND**

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

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

**3.6.10 NANDI**

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

**3.6.11 NOR**

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

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

**3.6.12 NORI**

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

**3.6.13 SHL**

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

„Shift Left“.

**3.6.14 SHLI**

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

„Shift Left Immediate“. Shifte die Bits im Register  $Y$   $N$  Stellen nach links.  $N$  ist eine positive Zahl im Bereich  $[0, 255]$ .

**3.6.15 SHR**

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

„Shift Right“.

**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 geschiftet. 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“.

**3.6.18 SHRAI**

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

„Shift Right Arithmetical Immediate“.

**3.6.19 ROTL**

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

„Rotate Left“.

**3.6.20 ROTLI**

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

„Rotate Left Immediate“.

**3.6.21 ROTR**

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

„Rotate Right“.

**3.6.22 ROTRI**

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

„Rotate Left Immediate“.

**3.7 Vergleichsinstruktionen****3.7.1 CMP**

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

„Compare“.

$$X \leftarrow \begin{cases} -1 & \text{falls } Y < Z \\ \pm 0 & \text{falls } Y = Z \\ +1 & \text{falls } Y > Z \end{cases}$$

**3.7.2 CMPU**

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

„Compare Unsigned“.

**3.7.3 CMPI**

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

„Compare Immediate“.

### 3.7.4 CMPIU

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

„Compare Immediate Unsigned“.

## 3.8 Sprungbefehle

Alle Sprungbefehle, außer dem [GO](#) Befehl, veranlassen einen relativen Sprung im Programmcode. Relativ im Sinne, dass die Parameter der Sprungbefehle einen Versatz zur aktuellen Programmadresse angeben.

Die Sprungbefehle bauen auf die Vergleichsbefehle auf.

### 3.8.1 BZ

Assemblername	Parameter	Maschinencode	Format
BZ	$X \in \mathcal{R}, N \in \mathbb{Z}$	0xC0	RNN

„Branch if Zero“.

$$X = 0 \Rightarrow IP \leftarrow IP + 4 \cdot N$$

### 3.8.2 BNZ

Assemblername	Parameter	Maschinencode	Format
BNZ	$X \in \mathcal{R}, N \in \mathbb{Z}$	0xC1	RNN

„Branch if Not Zero“.



**3.8.3 BLZ**

Assemblername	Parameter	Maschinencode	Format
BLZ	$X \in \mathcal{R}, N \in \mathbb{Z}$	0xC2	RNN

„Branch if Less than Zero“.

**3.8.4 BLEZ**

Assemblername	Parameter	Maschinencode	Format
BLEZ	$X \in \mathcal{R}, N \in \mathbb{Z}$	0xC3	RNN

„Branch if Less or Equal than Zero“.

**3.8.5 BGZ**

Assemblername	Parameter	Maschinencode	Format
BGZ	$X \in \mathcal{R}, N \in \mathbb{Z}$	0xC4	RNN

„Branch if Greater than Zero“.

**3.8.6 BGEZ**

Assemblername	Parameter	Maschinencode	Format
BGEZ	$X \in \mathcal{R}, N \in \mathbb{Z}$	0xC5	RNN

„Branch if Greater or Equal Zero“.

**3.8.7 BEI**

Assemblername	Parameter	Maschinencode	Format
BEI	$X \in \mathcal{R}, N, M \in \mathbb{Z}$	0xC8	RNN

„Branch if Equal Immediate“.

Fall  $X = N$ , springe  $M$  Instruktionen weiter.

**3.8.8 BNI**

Assemblername	Parameter	Maschinencode	Format
BNI	$X \in \mathcal{R}, N, M \in \mathbb{Z}$	0xC9	RNN

„Branch if Not equal Immediate“.

**3.8.9 BLI**

Assemblername	Parameter	Maschinencode	Format
BLI	$X \in \mathcal{R}, N, M \in \mathbb{Z}$	0xCA	RNN

„Branch if Less then Immediate“.

**3.8.10 BLEI**

Assemblername	Parameter	Maschinencode	Format
BLEI	$X \in \mathcal{R}, N, M \in \mathbb{Z}$	0xCB	RNN

„Branch if Less or Equal then Immediate“.

**3.8.11 BGI**

Assemblername	Parameter	Maschinencode	Format
BGI	$X \in \mathcal{R}, N, M \in \mathbb{Z}$	0xCC	RNN

„Branch if Greater then Immediate“.

**3.8.12 BGEI**

Assemblername	Parameter	Maschinencode	Format
BGEI	$X \in \mathcal{R}, N, M \in \mathbb{Z}$	0xCD	RNN

„Branch if Greater or Equal then Immediate“.

**3.8.13 JMP**

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

**3.8.14 G0**

Assemblername	Parameter	Maschinencode	Format
G0	$N \in \mathbb{N}$	0xDF	NNN

Absolute Adresse.

## 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

2.1	Liste der Spezialregister . . . . .	9
3.1	Verteilung des Befehlsraums . . . . .	22
3.2	Befehlentabelle . . . . .	23

# 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 Maschinenennamen von Registern und Befehlen.

**R****Register**

Eine sich im Prozessor befindende Speichereinheit. Das Register ist dem Programmierer sichtbar und kann mit Werten geladen werden. Siehe Abschnitt [2.2](#), Seite [8](#).

**S****Speicher**

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

# Index

- $\mathcal{R}$ , 8, 24
- 000, 19
- ABS, 49
- ADD, 44
- ADDI, 45
- ADDIU, 45
- ADDU, 44
- Adressierung
  - Indirekte, 11
- Adressierungsarten, 10
- Adressraumverteilung, 15
- ALIV, 28
- Allzweckregister, 8
- AND, 50
- ANDI, 50
- Assemblername, 8
- AUTSM, 26
- Befehlsraum, 21
  - Verteilung, 21
  - Verteilungstabelle, 22
- BEI, 58
- Betriebsmodus, 7, 25
- BGEI, 59
- BGEZ, 57
- BGI, 59
- BGZ, 57
- BLEI, 58
- BLEZ, 57
- BLI, 58
- BLZ, 57
- BNI, 58
- BNZ, 56
- Bussystem, 12
- Byte Order, 18
- BZ, 56
- CALL, 60
- CIN, 10
- CMP, 55
- CMPI, 55
- CMPIU, 56
- CMPU, 55
- COPY, 29
- CRM, 25
- CSM, 26
- Datentypen, 12
- DEC, 50
- DIE, 26
- DIV, 48
- DIVI, 48
- DIVIU, 48
- DIVU, 48
- ERR, 10
- ERRM, 10
- FP, 10
- GO, 59
- HATE, 27
- HI, 10
- INC, 50
- Instruktionen, 18
  - Kategorien, 21
- Instruktionsbreite, 18
- Instruktionsformat, 18
  - 000, 19
  - Liste, 21
  - NNN, 19



R00, 19  
RNN, 20  
RR0, 20  
RRN, 20  
RRR, 21  
Instruktionssatz, 18  
IP, 10  
  
JMP, 59  
  
KILL, 60  
  
LB, 30  
LBI, 32  
LBU, 30  
LBUI, 33  
LH, 31  
LHI, 33  
LHU, 31  
LHUI, 33  
LIN, 10  
LO, 10  
LW, 32  
LWI, 34  
LWU, 32  
LWUI, 34  
  
Maschinenname, 8  
Memory Mapped I/O, 12  
MOD, 49  
MODI, 49  
MOVE, 29  
MUL, 47  
MULI, 47  
MULIU, 48  
MULU, 47  
  
NAND, 52  
NANDI, 52  
NEG, 49  
NIN, 10  
NNN, 19  
NOP, 25  
NOR, 52  
NORI, 52  
  
NOT, 51  
Notation, 24  
NOTI, 51  
Null-Register, 9  
  
OR, 50  
ORI, 51  
  
Peripherie, 12  
    Adressierung, 14  
POP, 43  
POPB, 41  
POPH, 42  
Port, 15  
    Port-Typ, 16  
PUSH, 41  
PUSHB, 40  
PUSHH, 40  
  
R00, 19  
Register, 8  
    Allzweckregister, 8  
    Assemblername, 8  
    Maschinenname, 8  
    Spezialregister, 9  
RET, 60  
RNN, 20  
ROTL, 54  
ROTLI, 54  
ROTR, 54  
ROTRI, 55  
RR0, 20  
RRN, 20  
RRR, 21  
RSR, 26  
RST, 25  
  
SB, 34  
SBI, 37  
SBU, 35  
SBUI, 38  
SET, 28  
SETU, 28  
SH, 35  
SHI, 38

SHL, [52](#)  
SHLI, [53](#)  
SHR, [53](#)  
SHRA, [53](#)  
SHRAI, [54](#)  
SHRI, [53](#)  
SHU, [36](#)  
SHUI, [38](#)  
SOCL, [27](#)  
SP, [10](#)  
Speicher, [12](#)  
Speicherbasierte Kommunikation, [12](#)  
Speichermodell, [10](#)  
Speicherport, [12](#)  
Speicherunterraum, [14](#)  
Spezialregister, [9](#)  
Stack, [12](#)  
SUB, [45](#)  
SUBI, [46](#)  
SUBIU, [46](#)  
SUBU, [46](#)  
SW, [36](#)  
SWI, [39](#)  
SWU, [37](#)  
SWUI, [39](#)  
syscall, [60](#)  
  
TRST, [27](#)  
  
UMach  
    Aufbau, [7](#)  
    Register, [8](#)  
  
WAKE, [60](#)  
  
XOR, [51](#)  
XORI, [51](#)  
  
ZERO, [10](#)  
ZMB, [27](#)