UMach Spezifikation

18. Januar 2013

Inhaltsverzeichnis

Tabellenverzeichnis

Abbildungsverzeichnis

1 Einführung

UMach ist eine möglichst einfach konzipierte und programmierbare virtuelle Maschine (VM). Um diese Anforderung zu erfüllen wurde auf komplexere Funktionalitäten bewusst verzichtet. Sie besitzt einen fest definierten Instruktionssatz und Architektur. 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 8 beschrieben.

Für die Verwendung der virtuellen Maschine wird zuerst eine Assembler-Sprache zur Verfügung gestellt. In dieser Sprache werden Programme geschrieben und anschließend assembliert. Die assemblierte Datei (Maschinen-Code) kann dann von der virtuellen Maschine ausgeführt werden.

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). Bei der Implementierung eines Assemblers steht es frei, zusätzliche Befehle, Instruktionsformate, Aliase und sprachliche Konstrukte auf der Assembler-Ebene einzuführen. So sind z.B. die folgenden Befehle

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

auf Maschinencode-Ebene ungültig, denn hier verlangt die Formatdefinition der `ADD` und `SUB` Befehle die Angabe von drei Registern. Ein Assembler kann jedoch diese zusätzliche Formate definieren, solange er daraus gültigen UMach Maschinencode erzeugt. Gültiger Maschinencode für die genannten Beispiele wäre

```
ADDI R1 R2 5 # Maschinencode 0x32 0x01 0x02 0x05
SUBI R2 R2 4 # Maschinencode 0x35 0x02 0x02 0x04
```

2 Organisation der UMach VM

2.1 Aufbau

Die virtuelle UMach Maschine hat einen einfachen Aufbau, der im wesentlichen aus den folgenden Komponenten besteht:

1. Kern (Abschnitt 2.2)
2. Speicher (Abschnitt 2.4)
3. I/O Einheit (Abschnitt 2.5)

Siehe dazu Abbildung 2.1 auf Seite 9.

Nach dem Start der Maschine, setzt diese das Register `PC` („Programm Counter“) auf die Adresse 256. Danach fängt sie an, alle Instruktionen ab dieser Adresse der Reihe nach in den Kern zu holen und auszuführen. Nach Abarbeiten aller vorhandenen Instruktionen, schaltet sich die Maschine aus. Das Ende eines Programms wird entweder vom Programm selbst (Befehl `EOP`), oder durch erreichen des Datensegments signalisiert. Siehe auch den Abschnitt 2.4.3 Programm und Daten, auf der Seite 18.

Alle Eingabe- und Ausgabeoperationen werden an die I/O-Einheit weitergeleitet und von dieser ausgeführt (siehe auch die Abschnitte 2.5 und 3.10).

2.1.1 Betriebsmodi

Die Umach VM kann in zwei verschiedenen Betriebsmodi oder auch Betriebsarten betrieben werden:

1. Normalmodus
2. Debugmodus

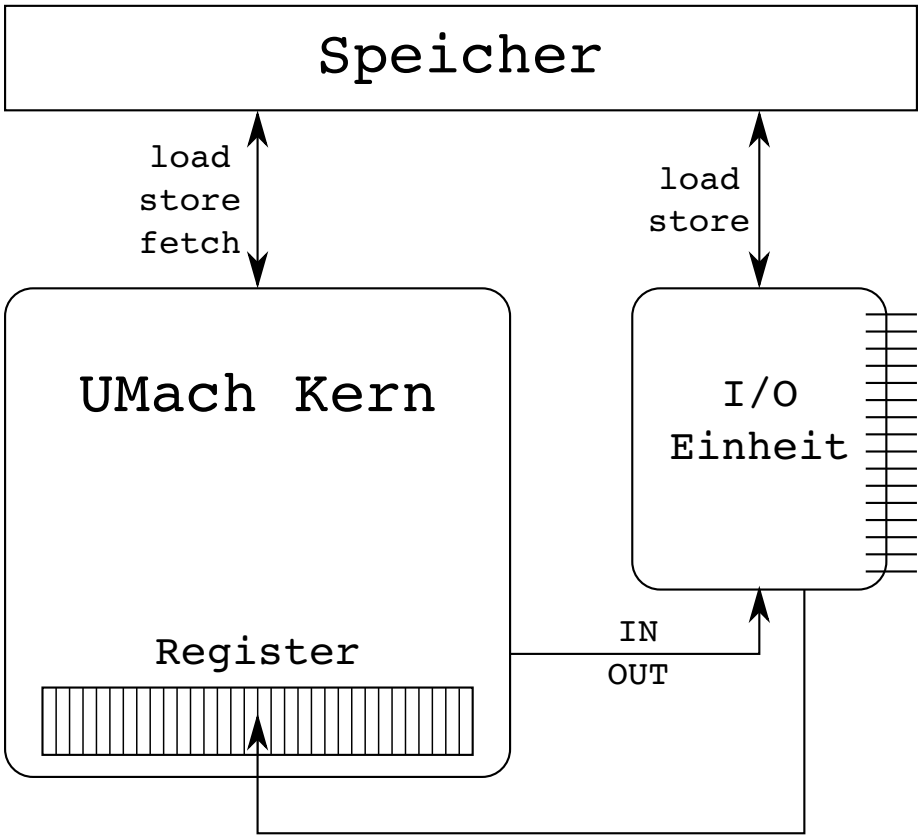

Abbildung 2.1: Aufbau der UMach Maschine. Die I/O Einheit wird durch den Kern bei der Ausführung der I/O-Befehle angestoßen. Sie schreibt in die Kern-Register die Anzahl der tatsächlich gelesenen oder ausgegebenen Bytes zurück.

Normalmodus Die virtuelle Maschine führt alle Instruktionen ohne Unterbrechung aus. Nach der Ausführung schaltet sich die Maschine aus.

Debugmodus Die virtuelle Maschine führt immer nur eine einzige Instruktion aus und bleibt dannach stehen. Um mit der folgenden Instruktion fortzufahren wird immer ein externen Signal benötigt. Dieser Modus soll dem Entwickler erlauben, ein Programm schrittweise zu debuggen.

Der Modus kann vor dem Hochfahren der Maschine festgelegt, jedoch im Betrieb nicht mehr geändert werden. Wird kein Modus angegeben, startet die Maschine im Normalmodus.

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 Instruktionscode (Opcode) 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 2.2 auf Seite 11.

Auflistung der Schritte:

1. Fetch – Holen der Instruktion aus dem Speicher an der im `PC` vorliegenden Adresse.
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.

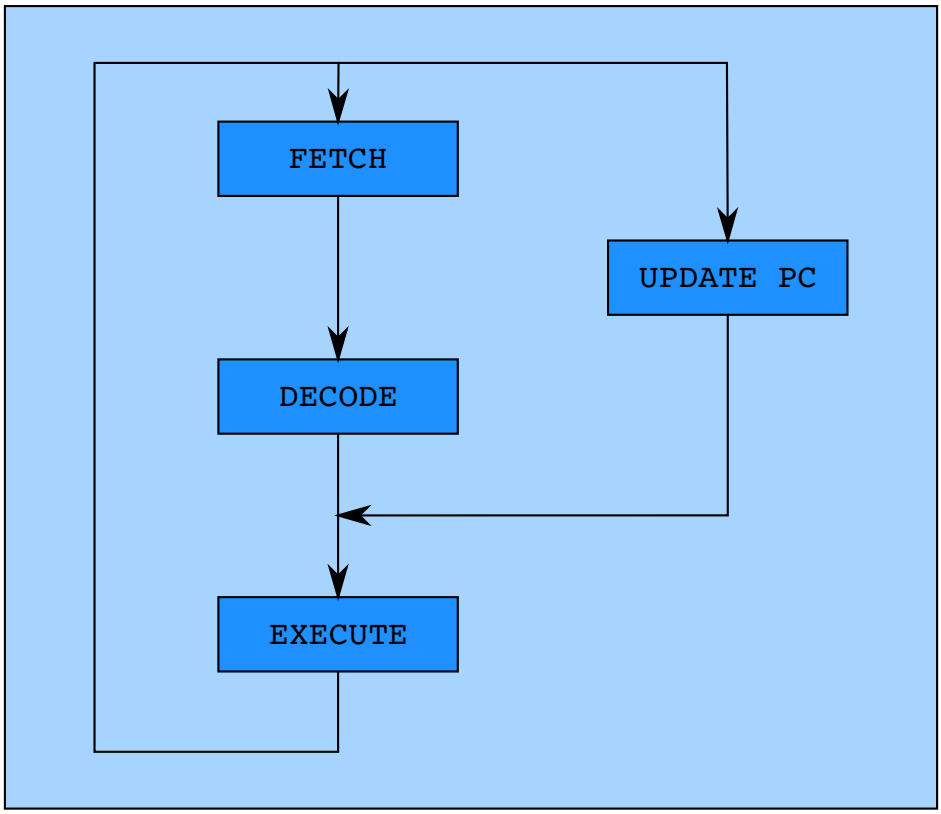

Abbildung 2.2: Von Neumann Zyklus

2.2 Kern

Der Kern der Maschine besteht aus den folgenden Komponenten:

1. Steuerwerk – Das Steuerwerk besteht aus einer "Instruction Fetch Unit" und einer Decodierungseinheit. Aufgabe der Instruction Fetch Unit ist es, die Programminstruktionen entsprechend dem `PC` aus dem Speicher zu holen und der Decodierungseinheit zu übergeben. Diese decodiert die Instruktion und stellt die Recheneinheit entsrpechend ein.

2. Recheneinheit – Zuständig für die tatsächliche Ausführung der Instruktionen.

3. Register – Kleine Speichereinheiten, die entweder jene Daten halten auf der die Recheneinheit operiert, oder Ergebnisse einer Instruktion aufnehmen.

Der Kern besitzt keine Pipeline.

2.3 Register

Die Register sind Speichereinheiten im Prozessorkern. Die meisten Instruktionen operieren auf diesen Registern.

Für alle Register gilt:

1. Das Register ist ein Element aus der Menge $\mathcal{R}$ aller Register. Die Notation $x \in \mathcal{R}$ bedeutet somit, dass x ein Register ist.

2. Die Speicherkapazität beträgt 32 Bit.

3. Es gibt eine eindeutige natürliche Nummer, die Registernummer. Diese dient zur Identifikation des Registers in der Maschine und auf Maschinencode-Ebene.

4. Auf Assembler Ebene git es einen eindeutigen Namen für das Register.

Die UMach Maschine besitzt zwei Arten von Registern: die Allzweck- und Spezialregister.

2.3.1 Allzweckregister

Es gibt 32 Allzweckregister, welche dem Programmierer für generelle Zwecke zur Verfügung stehen. Alle Allzweckregister werden beim Hochfahren der Maschine mit dem Wert Null (`0x00`) initialisiert. Außer dieser Initialisierung verändert die Maschine den Inhalt der Allzweckregister nur in Folge einer expliziten Instruktion.

Die Allzweckregister werden auf Maschinencode-Ebene fortlaufend mit den Registernummern 1 bis 32 gekennzeichnet (`0x01` bis einschliesslich `0x20` im Hexadezimalsystem). Die Assemblernamen sind entsprechend $R1, R2 \ldots$ bis $R32$. Die Zahl nach dem Buchstaben R ist im Dezimalsystem angegeben und ist fester Bestandteil des Registernamens.

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

Das Register mit der Registernummer Null, oder auch Null-Register, ist ein Spezialregister, welches immer den Wert Null hat und nicht beschreibbar ist. Der Assemblername ist `ZERO`. (siehe auch Abschnitt 2.3.2 auf Seite 13).

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	$R_1 \leftarrow R_2 + R_3$			

2.3.2 Spezialregister

Die Spezialregister werden von der UMach Maschine für gesonderte 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 Registernummer der Spezialregister setzen die Registernummerierung der Allzweckregister zwar bei 33 fort, die Assemblernamen sind aber zweckspezifisch: es gibt kein Spezialregister mit dem Namen $R33$. Die Tabelle 2.1 auf Seite 14 enthält die Liste aller

Spezialregister. In der ersten Spalte steht der Assemblername, so wie er vom Programmierer verwendet wird. In der zweiten Spalte ist der Maschinenname bzw. die Registernummer im Dezimalsystem angegeben. Die dritte Spalte enthält eine kurze Beschreibung und Bemerkungen. Falls die Beschreibung nicht anders spezifiziert, ist das Register nicht schreibgeschützt.

Tabelle 2.1: Liste der Spezialregister

`PC`	33	„Instruction Pointer“, oder „Program Counter“. Enthält zu jeder Zeit die Adresse der nächsten Instruktion. Wird auf 256 gesetzt, wenn die Maschine hochfährt. Diese Adresse ist die Startadresse des Programms im Speicher (siehe auch Abschnitt 2.4.1, Seite 17). Wird nach dem Abfangen einer Instruktion in das Register `IR` automatisch inkrementiert. Schreibgeschützt.
`DS`	34	„Data Segment“, Datensegment. Enthält diejenige Adresse im Speicher, wo das Datensegment anfängt. Ab dieser Adresse werden im Speicher die eingebetten Programmdaten abgelegt. Die interne Logik der UMach-Maschie geht davon aus, dass ab dieser Adresse keine Instruktionen mehr vorhanden sind. Schreibgeschützt.
`HS`	35	„Heap Segment“, „Heap Start“, oder Freispeicher Segment. Enthält diejenige Speicheradresse, ab wo das Programm neuen Speicher belegen kann. Dieser Speicherbereich folgt unmittelbar nach dem Datensegment und wird rechts vom Stack begrenzt. Schreibgeschützt.
`HE`	36	„Heap End“, oder Ende des Heaps. Markiert das Ende des Heap-Bereichs, bzw. die Adresse des letzen Bytes, das noch zum Heap gehört.
`SP`	37	„Stack Pointer“. Enthält die Speicheradresse des höchsten Eintrags auf dem Stack. Wird beim Hochfahren der Maschine auf die maximal erreichbare Speicheradresse plus 1 gesetzt. Die 1, die zur maximalen Adresse addiert wird, setzt den Stack Pointer auf eine ungültige Adresse und sorgt dafür, dass keine Werte mittels `POP` gelesen werden, bevor Werte auf den Stack gepusht wurden. Siehe auch 2.4.4, Seite 19.
`FP`	38	„Frame Pointer“. Enthält die Startadresse des Stack Frames einer Subroutine und unterstützt die Implementierung von Funktionen.
`IR`	39	„Instruction Register“. Enthält die gerade ausgeführte Instruktion. Schreibgeschützt.
`STAT`	40	Enthält Status-Informationen. Diese Informationen sind in den einzelnen Bits dieses Register gespeichert. Welche Informationen vorhanden sind liegt an der jeweiligen Anwendung.

ERR	41	„Error“. Fehlerregister. Die einzelnen Bits dises Registers geben Auskunft über Fehler, die mit der Ausführung eines Befehls verbunden sind. Ist ein bestimmtes Bit gesetzt, so wird dadurch der entsprechende Fehler signalisiert. Für eine Liste der verwendeten Bits un deren Bedeutung, siehe Tabelle 2.2 auf der Seite 16.
HI	42	„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	43	„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.
CMPR	44	„Comparison Result“. Enthält das Ergebnis eines Vergleichs. Siehe auch Abschnitt 3.6, Seite 55.
ZERO	00	Enthält die Zahl Null. Schreibgeschützt.

2.3.3 Das ERR Register

Jedes Bit im Register signalisiert das auftreten eines spezifischen Fehlers. Die Bitstellen werden dabei entsprechend deren Stelligkeiten durchnummeriert: Bit mit Stelligkeit 2^0 hat Position 0, Bit mit Stelligkeit 2^{31} hat die Position 31.

Die Bits im ERR Register werden in die folgenden 4 Gruppen unterteilt:

0 - 7	Kern Fehler
8 - 15	Speicherfehler
16 - 23	I/O Fehler
24 - 31	Systemfehler

Tabelle 2.2 listet alle Fehler auf, die in dem ERR Register signalisiert werden können. Alle nicht spezifizierten Bitnummer können frei verwendet werden – das bedeutet, sie werden von der Maschine selbst nicht berücksichtigt.

Tabelle 2.2: Bedeutung der einzelnen Bits im ERR Register

0	Division durch Null
1	Arithmetischer Überlauf
5	Ungültiger Befehlscode
6	Ungültige Registernummer
8	Ungültige Speicheradresse
9	Zugriffsverletzung (segmentation fault)
11	Stack Überlauf
16	I/O Port existiert nicht

Siehe auch den Abschnitt 2.4.5 Speicher Fehler, Seite 19.

2.3.4 Das STAT Register

Das `STAT`-Register wird zur Speicherung verschiedener Status-Informationen verwendet. Dabei sind diese Informationen als boolsche Werte (gesetzt oder nicht gesetzt) in den einzelnen Bits des Register gespeichert. Ein Überblick über die verwendeten Bits wird in der Tabelle 2.3 angeboten. Dort wird zu jeder Bitnummer seine Funktion angegeben.

Zum Beispiel, ist das Bit mit Nummer 0 (Wertigkeit 2^0) in diesem Register gesetzt, so bedeutet das, dass Interrupts nicht ausgewertet werden. Das wird z.B. intern vom Befehl `INT` verwendet, um sich selber zu blockieren, wenn der Befehl den Speicher anspricht, was wiederum einen Interrupt generieren kann.

Tabelle 2.3: Bedeutung der einzelnen Bits im STAT Register

0	Interrupts sind blockiert (werden nicht ausgeführt)

Die nicht spezifizierten Bitnummer können frei verwendet werden.

2.4 Der Speicher

Beim Hochfahren der Maschine wird der komplette Speicher mit dem Wert Null initialisiert und anschließend das angegebene Maschinenprogramm in den Speicher geladen. Beim Start der Maschine kann die gewünschte Größe des Speichers angegeben werden. Wird

kein Wert vorgeschlagen, so nimmt der Speicher eine Größe von 2048 Bytes ein (genug für die Interrupttabelle und Programm). Nach dem Start bleibt die Größe des Speichers fest und kann nicht mehr geändert werden.

2.4.1 Speicherstruktur

Der Speicher der UMach Maschine hat zur Laufzeit eine bestimmte Struktur, bzw. wird in bestimmte Bereiche, oder Segmente, unterteilt. Diese Segmente sind

1. Interrupttabelle
2. Programmsegment (Code-Segment)
3. Datensegment
4. Heap
5. Stack

Diese Segmente, oder Speicherbereiche, werden in den folgenden Abschnitten getrennt vorgestellt. Für einen Überblick siehe auch die Abbildung 2.3.

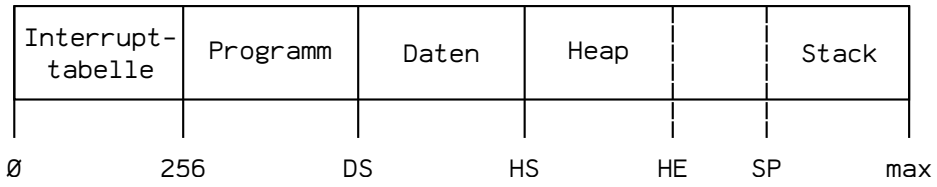

Abbildung 2.3: Speicherstruktur zur Laufzeit. Die Begrenzungen zwischen den Segmenten werden durch konstante Zahlen oder Registerinhalte angegeben. Von diesen sind lediglich die Register `SP` und `HE` veränderbar, alle anderen sind, nach dem Laden des Programms in den Speicher, fest.

2.4.2 Interrupttabelle

Die Interrupttabelle besteht aus einer Reihe von 32-Bit langen Sprungadressen zum ausführbaren Code, bzw. zu sogenannten Interruptroutinen, oder auch „Interrupt Handlers“, die in Ausnahmefällen ausgeführt werden sollen. Jedes mal, wenn eine Ausnahmesituation auftritt (meistens eine Fehlersituation), wird intern ein Interruptsignal erzeugt, das mit einer Kennnummer (Interruptnummer) versehen ist. Die Interruptnummer wird zur Berechnung eines Index in dieser Tabelle verwendet.

Die Interrupttabelle wird nicht von der Maschine gefüllt, sondern es ist Aufgabe des Maschinenprogramms bei Bedarf entsprechende Einträge der Interrupttabelle zu füllen und die zugehörige Funktionalität bereit zu stellen. Ist eine Interrupadresse nicht gesetzt, d.h. ist der entsprechende Tabelleneintrag Null, so reagiert die Maschine auf die Ausnahmesituation indem sie hält. Für weitere Informationen bzgl. der Fehlerbehandlung siehe den Abschnitt 2.6, ab der Seite 22.

Die Interrupttabelle besteht aus 64 Einträgen, die 64 möglichen Interrupts entsprechen. Jeder Eintrag beträgt wie ein Register 32 Bit, oder 4 Byte. Da es 64 Einträge gibt, ist die Interrupttabelle $64 \cdot 4 = 256$ Bytes groß. Die Interrupttabelle fängt an der Adresse Null an. Siehe auch Abbildung 2.3 auf Seite 17.

Die Berechnung der Adresse eines Interrupt-Handelers aus der Interruptnummer erfolgt durch Multiplizieren mit 4 der Interruptnummer. Also für sine Interruptnummer n wird die Adresse a wie folgt berechnet:

$$a = 4 \cdot n$$

Als Beispiel, aus der Interruptnummer $n = 16$ wird die Adresse $a = 4 \cdot 16 = 64$ berechnet. An dieser Adresse wird die Adresse einer Subroutine gesucht, die als Interrupt-Handler gestartet wird – bzw. dort gesprungen wird.

Tabelle 2.4 auf Seite 23 listet alle definierten Interruptnummern und deren Bedeutung auf.

2.4.3 Programm und Daten

Nach der 256 Bytes (64 mal 4) großen Interrupttabelle folgt das eigentliche Programm, das von der Maschine ausgeführt werden soll. Insbesondere ist dieses Programm dafür zuständig, die Interrupttabelle zu füllen, falls (bestimmte) Interrupts behandelt werden sollen.

Das Programmsegment fängt an der Adresse 256 an und ist schreibgeschützt. Es enthält eine Kopie der ausführbaren Instruktionen aus der Programmdatei.

Das Datensegment folgt nach dem Programmsegment. Seine Anfangsadresse wird im Register *DS* gespeichert. Seine Größe ist fest und wird durch die Anzahl der Datenbytes bestimmt. Dieses Segment enthält eine Kopie der Daten, die in der Programmdatei eingebettet sind. Das Datensegment ist nicht schreibgeschützt.

2.4.4 Heap und Stack

Nach dem Datensegment folgt das Heap-Segment. Seine Anfangsadresse wird im Register `HS` gespeichert. Dieses Segment, oder Speicherbereich, steht dem Programmierer frei. Das Ende des Heap-Bereichs wird durch den Inhalt des Registers `HE` markiert. Der Heap belegt also alle Speicheradressen aus dem Interval $[\$HS, \$HE]$, wobei $\$HS$ und $\$HE$ die Inhalte der Register `HS` und `HE` bezeichnen.

Der Stack ist ein spezieller Bereich im Speicher. Dieser Bereich fängt am Ende des Speichers mit der größten Adresse an und erstreckt sich bis zur derjenigen 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.

Beim Hochfahren der Maschine, wird das Register `SP` auf die maximal erreichbare Speicheradresse plus Eins gesetzt. Damit können keine Werte gelesen werden, bevor Werte geschrieben wurden.

2.4.5 Speicher Fehler

Bei der Speicheraddressierung können während der Programmausführung verschiedene Fehler auftreten.

Ungültige Speicheradresse Falls das Programm eine nicht-existierende Speicheradresse anspricht, wird von der Maschine der entsprechende Interrupt erzeugt. Folgende Fälle zählen dazu:

1. Schereiben oder Lesen an Adresse kleiner Null.

2. Schereiben oder Lesen an Adresse größer als die maximal erreichbare Adresse.

Zugriffsverletzung Eine Zugriffsverletzung (ähnlich einer „segmentation fault“, oder „segfault“) liegt vor, wenn Schreibrechte eines Segments verletzt werden, oder wenn die Grenzen eines Segments überschritten werden. Konkret sind die folgenden Fälle eine Zugriffsverletzung:

1. Schreiben im Programmsegment.

2. Register `HE` auf einer Adresse größer oder gleich der Adresse in `SP` setzen (Heap Überlauf).

3. Register `SP` auf eine Adresse größer als die maximal erreichbare Adresse setzen (auch durch den `POP` Befehl).

Stack Überlauf Ein Stack Überlauf liegt vor, wenn der Stack Pointer (Register `SP`) auf eine Speicheradresse kleiner gleich dem Wert des Registers `HE` gesetzt wird. Mit anderen Worten, wenn der Stack das Daten-, Programm- oder Heap-Segment überlappen würde.

2.4.6 Speicheradressierung

Um die Komplexität der UMach Maschine gering zu halten, kennt die vorliegende Version lediglich absolute Adressen. Konzepte wie virtuelle Adressen, Prozesse und getrennte Addressräume für Prozesse werden nicht unterstützt. Wird versucht, den Inhalt einer Speicheradresse zu lesen oder zu schreiben, so wird die angegebene Adresse nicht übersetzt oder modifiziert, sondern direkt verwendet.

2.5 I/O Einheit

Die I/O-Architektur der UMach Maschine ist am sogenannten „Port-Mapped I/O“, oder „Port I/O“ angelehnt[1]. In dieser Architektur, verfügt eine Maschine über Ein- und Ausgabe „Ports“, die einen eigenen Adressraum bilden.

Alle I/O Operationen finden zwischen dem Speicher und den Peripherie-Geräten statt. Dabei wird der gesamte Datentransfer von der I/O-Einheit vermittelt und gesteuert (siehe Abbildung 2.4 auf Seite 21). Die I/O-Einheit besitzt zu diesem Zweck einen direkten Zugriff auf den Speicher und kann dort unabhängig vom Kern lesen und schreiben (Direct Memory Access, DMA). Es ist zu bemerken, dass der direkte Speicherzugriff von der I/O-Einheit unternommen wird und nicht von den peripherischen Geräten selbst.

Alle I/O Operationen werden vom Kern unter Programmkontrolle angestoßen (siehe auch Abschnitt 3.10 auf Seite 62). Es werden also keine I/O Operationen ausgeführt, die

[1] Als Gegenstück von „Memory Mapped I/O“.

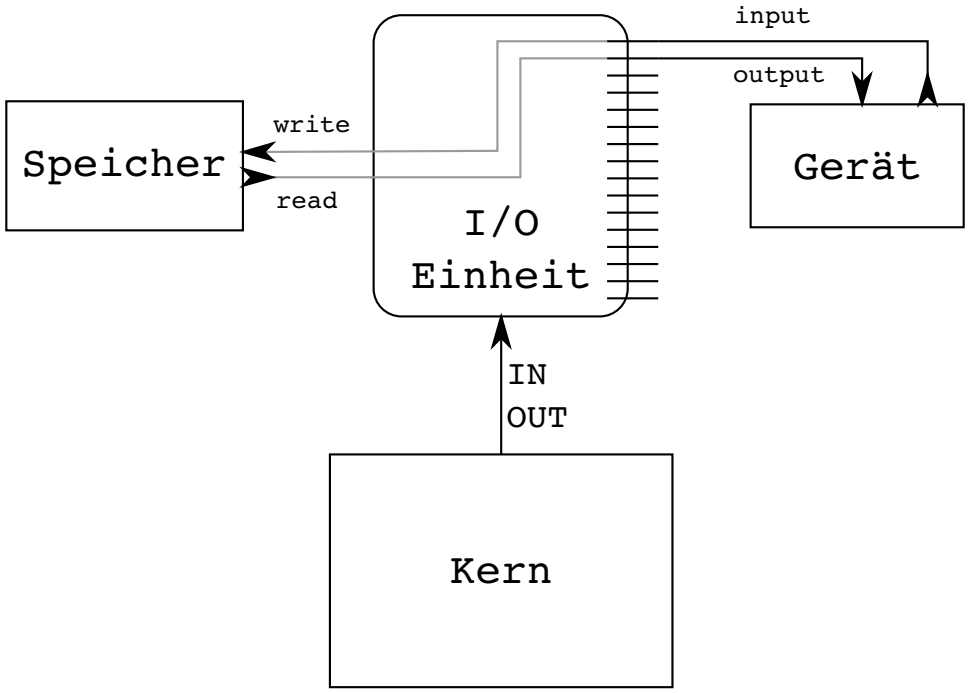

Abbildung 2.4: I/O wird von der I/O Einheit erledigt

nicht explizit durch Maschinenbefehle angefordert werden. Insbesondere werden keine I/O-Operationen aufgrund von Unterbrechungsanfoderungen (interrupts) initiiert.

Wenn ein I/O-Befehl ausgeführt wird, delegiert der Kern die Ausführung an die I/O-Einheit und wartet, bis diese fertig ist. Erst wenn die I/O-Einheit mit dem Transfer zwischen Speicher und Peripherie fertig ist, fährt der Kern mit dem Programmablauf fort. Es werden parallel keine Instruktionen aus dem Speicher geholt oder ausgeführt.

Die I/O-Einheit der UMach Maschine besteht aus einer Reihe von jeweils 8 Eingangs- und Ausgangschnittstellen, auch Ports genannt. An diesen Ports können verschiedene physikalische Geräte angeschlossen werden, welche die entsprechenden Daten generieren bzw. verarbeiten können. (Siehe Abbildung 2.1 auf Seite 9.)

Die I/O Ports sind in zwei Kategorien unterteilt: 8 Eingabeports und 8 Ausgabeports. Von der Bauart und Struktur her, gibt es innerhalb der jeweiligen Kategorie keine Unterschiede zwischen Ports. Sie werden lediglich anhand deren Nummern identifiziert. Die Nummerierung der Ports fängt bei 0 an und wird bis einschließlich Port 7 fortgeführt.

Die Eingabe- und Ausgabefunktionen der I/O-Einheit werden durch I/O-Instruktionen angefordert. Diese Instruktionen werden im Abschnitt 3.10, ab der Seite 62 beschrieben.

Bemerkung Zu diesem Entwicklungspunkt sind für die peripherischen Geräte, bzw. für die entsprechenden Ports keine Kontrolle- oder Statusregister vorgesehen. Es gibt auch keine entsprechende Befehle für die Kontrolle und Statusabfrage der peripherischen Geräte. Es wird lediglich in den Speicher eingelesen und aus dem Speicher ausgegeben. Kontrolle und Statusabfrage sind als Software-Protokoll gedacht und nicht als Hardware-Vorgang.

2.6 Interrupts

Ein Interrupt ist eine Unterbrechung der Programmausführung unter bestimmten Umständen. Die UMach Maschine verwendet die folgenden Arten von Interrupts:

1. Automatische Interrupts, oder Hardware-Interrupts, die aufgrund von Fehlern geschehen und eine rudimentäre Fehlerbehandlung bei schwerwiegenden Fehlern, wie die Nulldivision, ermöglichen sollen.

2. Software-Interrupts, die vom der Software angefordert werden können.

Jeder Interrupt hat eine bestimmte Nummer, die ganzzahlige Werte von 0 bis 63 annimmt. Die Interruptnummer wird zur Berechnung eines Index in der Interrupttabelle verwendet, um die Speicheradresse einer Interruptroutine zu finden (siehe den Abschnitt 2.4.2, besonders die Tabelle 2.4 auf der Seite 23). Ist diese Adresse ungleich Null, so wird das `PC` Register auf diese Adresse gesetzt und die Interruptroutine wird ausgeführt. Wird keine solche Routine gefunden, so wird die gesamte Programmausführung unterbrochen und die Maschine hält an.

Die Interruptroutinen müssen vom Hauptprogramm als Sprungadressen in der Interrupttabelle gesetzt werden. Dabei ist die Struktur der Interrupttabelle zu beachten, so wie sie im Abschnitt 2.4.2 auf Seite 17 beschrieben wird.

2.6.1 Automatische Interrupts

Im Normalfall wird eine Instruktion ohne Weiteres ausgeführt. In Ausnahmefällen – wenn die Instruktion nicht ausgeführt werden kann, meistens wegen ungültigen Parametern – wird eine Ausnahmesituation durch eine Interruptnummer signalisiert und der Programmfluss unterbrochen.

Beispiel Zum Beispiel, die `DIV` Instruktion (Division) erzeugt den Interrupt mit Nummer 1, falls ihr zweites Argument gleich Null ist (siehe auch die Tabelle 2.4 auf der Seite 23). Wenn der Interrupt bearbeitet wird, schaut die Maschine in der Interrupttabelle an der Position 1 nach. Ist der Eintrag ungleich Null, so wird der Eintrag als Adresse einer Routine behandelt und dorthin gesprungen: das Register `PC` wird zuerst auf den Stack gesichert und dannach gleich dem Tabelleneintrag gesetzt. Ist der Eintrag dagegen Null, so hält die Maschine an.

2.6.2 Software-Interrupts

Eine Unterbrechung der Programmausführung kann auch absichtlich erzeugt werden. Dafür verwendet man den Befehl `INT`. Wenn ein Software-Interrupt generiert wird, verhält sich die Maschine wie im Falle eines automatischen Interrupts.

2.6.3 Interruptnummer

Tabelle 2.4: Interruptnummer

Nummer	Beschreibung
0	Standardinterrupt
1	Division durch Null
2	Arithmetischer Überlauf
8	Ungültiger Befehlscode
9	Ungültige Registernummer
10	Ungültiges Argument
16	Ungültige Speicheradresse
17	Zugriffsverletzung (segfault)
26	Stack Überlauf
32	I/O Port existiert nicht
56	Interner Fehler
63	Syscall, Aufruf an das Betriebssystem

Eine Bemerkung zum Interrupt mit Nummer 10: dieser Interrupt wird dann erzeugt, wenn zwar eine Registernummer oder Adresse in Ordnung ist, dafür aber der Inhalt nicht. Dies passiert z.B. wenn negativen Längen für den Befehl `OUT` verwendet werden.

Siehe auch den Abschnitt 2.4.5 Speicher Fehler, Seite 19.

3 Instruktionssatz

In diesem Kapitel werden alle Instruktionen der UMach VM vorgestellt.

3.1 Allgemeines

3.1.1 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). 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 Byte 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 Befehlscode (Opcode), bzw. der Operation, die von der UMach virtuellen Maschine ausgeführt werden soll. Dieser 8-Bit-Befehlscode belegt also die 8 höchstwertigen Bits einer 32-Bit-Instruktion. Die übrigen 24 Bits, falls 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	Befehlscode	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.

000

erstes Byte	zweites Byte	drittes Byte	viertes Byte
Befehlscode	nicht verwendet		

Eine Instruktion, die das Format 000 hat, besteht lediglich aus einem Befehlscode ohne Argumenten. Die letzen drei Bytes werden von der Maschine nicht ausgewertet.

NNN

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

Die Instruktion im Format NNN besteht aus einem Befehlscode 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.

R00

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

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

RNN

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

Eine Instruktion im Format RNN besteht aus einem Befehlscode, gefolgt von einer Registernummer 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` („big endian“ Interpretation der Zahl N).

RR0

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

Eine Instruktion im Format RR0 besteht aus einem Befehlscode im ersten Byte, gefolgt von der Angabe zweier Registernummer in den folgenden 2 Bytes. Das dritte Byte wird nicht verwendet, bzw. wird ignoriert.

RRN

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

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

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

wird wie folgt interpretiert: 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`.

RRR

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

Eine Instruktion im Format RRR besteht aus der Angabe eines Befehlscodes im ersten Byte, gefolgt von der Angabe dreier Registernummer R_1, R_2 und R_3 in den jeweiligen folgenden drei Bytes.

Zusammenfassung

Im folgenden werden die Instruktionsformate tabellarisch zusammengefasst.

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

3.1.2 Verteilung des Befehlsraums

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

Maschinencodes	Kategorie
00 - 0F	Kontrollbefehle
10 - 2F	Lade-/Speicherbefehle
30 - 4F	Arithmetische Befehle
50 - 6F	Logische Befehle
70 - 7F	Vergleichsbefehle
80 - 8F	Sprungbefehle
90 - 9F	Unterprogrambefehle
A0 - AF	Systembefehle
B0 - BF	IO Befehle

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

1. Kontrollinstruktionen, die die Maschine in ihrer gesamten Funktionalität betreffen.

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

9. IO Instruktionen

Die oben angegebenen Instruktionskategorien unterteilen den Befehlsraum in 9 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 28 angegeben.

Die Tabelle 3.3 auf der Seite 65 enthält eine Übersicht aller Befehle und deren Maschinencodes (Opcodes). Diese Tabelle wird folgenderweise gelesen: in der am weitesten linken Spalte wird die erste hexadezimale Ziffer eines Befehls angegeben (ein Befehlscode ist zweistellig im Hexadezimalsystem). Jede solche Ziffer hat rechts Zwei Zeilen, die von links nach rechts gelesen werden: eine Zeile für die Ziffern von 0 bis 7, die anderen für die übrigen Ziffern 8 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 und das Instruktionsformat, das die Typen der Parameter definiert, formal angegeben. Zudem werden Anwendungsbeispiele angegeben. Die Instruktionsformate können im Abschnitt 3.1.1 ab der Seite 24 nachgeschlagen werden.

3.1.3 Notationen

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.

Wenn X ein Register bezeichnet, dann bezeichnet $\$X$ den Inhalt des Registers X. Diese Notation wird verwendet, um in Aussagen wie „$\$X + \$Y$“ klar zu machen, dass es sich um die Registerinhalte von X und Y handelt, nicht um die Register, bzw. um die Registernummer selbst.

Die Tabelle 3.2 auf der Seite 29 wiedergibt alle Notationen, die in diesem Kapitel verwendet werden.

Tabelle 3.2: Verwendete Notationen

Notation	Bedeutung
$\mathbb{N}$	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 $\mathbb{N}$
$\mathcal{R}$	Menge aller Register
$X \in \mathcal{R}$	X ist ein Register

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

Notation	Bedeutung
$\$X$	Inhalt des Registers X
$a \leftarrow b$	a wird auf b gesetzt
$a \rightarrow b$	b wird auf a gesetzt
$X \leftarrow \$Y$	X wird auf den Inhalt des Registers Y gesetzt
$mem(X)$	Speicherinhalt an der Adresse X (1 Byte)
$mem_n(X)$	n-Bytes-Block im Speicher ab Adresse X
$a \rightarrow mem_n(X)$	n-Bytes-Block im Speicher ab Adresse X werden mit a belegt
`mem[n]`	äquivalent zu $mem(n)$
$lsb_n(X)$	„least significant n bits“ von X
$msb_n(X)$	„most significant n bits“ von X

3.2 Kontrollinstruktionen

3.2.1 `NOP`

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

Diese Instruktion („No Operation“) bewirkt nichts.

3.2.2 `EOP`

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

„End Of Program“. Die Maschine ausschalten.

3.3 Lade- und Speicherbefehle

3.3.1 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
```

3.3.2 CP

Assemblername	Parameter	Maschinencode	Format
CP	$X, Y \in \mathcal{R}$	0x11	RR0

„Copy“. 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
CP    R2 R1  # R2 ← 5
```

3.3.3 LB

Assemblername	Parameter	Maschinencode	Format
LB	$X, Y \in \mathcal{R}$	0x12	RR0

Lade ein Byte aus dem Speicher mit Adresse $\$Y$ in das niedrigstwertige Byte des Registers X. Die anderen Bytes von X werden von diesem Befehl nicht geändert. Insbesondere, werden sie nicht auf Null gesetzt.

Äquivalenter C Code:

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

Entspricht

$$lsb_8(X) \leftarrow mem(\$Y)$$

(siehe Tabelle 3.2 auf Seite 29 für die Notationen).

Fehler Falls der Inhalt des Registers Y eine ungültige Speicheradresse ist, wird der Interrupt mit Nummer 16 (ungültige Speicheradresse) generiert und das Bit mit Nummer 8 im Register ERR gesetzt.

Beispiel Angenommen, der Speicher an den Adressen 100 und 101 hat den Wert 5, bzw. 6. Dann können diese zwei nacheinander folgenden Bytes auf die folgende Weise in $R2$ gelesen werden. $R1$ wird dabei als Zeiger im Speicher verwendet.

```
SET  R1     100   # Speicheradresse R1 = 100
SET  R2       0
LB   R2  R1       # R2 = 5 (mem(100))
SHLI R2  R2   8   # shift left 8 Bit, R2 = 1280
INC  R1           # R1++, R1 = 101
LB   R2  R1       # R2 = 1286 (R2 + mem(101))
```

3.3.4 LW

Assemblername	Parameter	Maschinencode	Format
LW	$X, Y \in \mathcal{R}$	0x13	RR0

„Load Word“. Lade ein Wort (4 Byte) aus dem Speicher mit Adresse $\$Y$ 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 $\$Y + 0$, $\$Y + 1$, $\$Y + 2$ und $\$Y + 3$ zu einem 4-Byte Wort zusammengesetzt und so in X ablegt.

Entspricht

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

Fehler Wie bei der Instruktion `LB`.

Bemerkung Die UMach Maschine verwendet stets die „big-endian“ Byte-Reihenfolge.

Beispiel Angenommen, 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
LW  R2 R1      # R2 ← mem4(R1) = 66051
```

3.3.5 SB

Assemblername	Parameter	Maschinencode	Format
`SB`	$X, Y \in \mathcal{R}$	`0x14`	RR0

„Store Byte“. Speichert den Inhalt des niedrigstwertigen Byte von X an der Speicheradresse $\$Y$. X und Y sind dabei Register.

Entspricht

$$\$X \rightarrow mem_1(\$Y)$$

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

Fehler Falls der Inhalt des Registers Y eine ungültige Speicheradresse ist, wird der Interrupt mit Nummer 16 (ungültige Speicheradresse) generiert und das Bit mit Nummer 8 im Register `ERR` gesetzt. Falls die Speicheradresse $\$Y$ schreibgeschützt ist, wird der Interrupt mit Nummer 17 (Zugriffsverletzung) erzeugt und das Bit mit Nummer 9 im Register `ERR` gesetzt..

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

3.3.6 SW

Assemblername	Parameter	Maschinencode	Format
SW	$X, Y \in \mathcal{R}$	0x15	RR0

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

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

Fehler Wie bei SB, nur auf den ganzen Speicherbereich $\$Y$ bis $\$Y + 3$ bezogen.

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           # mem[128] = 0x01
                    # mem[129] = 0x02
                    # mem[130] = 0x03
                    # mem[131] = 0x04
```

3.3.7 PUSH

Assemblername	Parameter	Maschinencode	Format
PUSH	$X \in \mathcal{R}$	0x18	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]`

Entspricht

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

Fehler Falls die Verringerung $\$SP - 4$ eine Speicheradresse außerhalb des Stackbereichs ergibt, d.h. wenn $(\$SP - 4) \leq \$HE$, wird der Interrupt mit Nummer 26 (Stack Überlauf) generiert und das Bit mit Nummer 11 im Register `ERR` gesetzt. Dabei beinhaltet das Register HE die Adresse des letzten Bytes im Heap-Segment. Siehe auch den Abschnitt 2.4.4 auf der Seite 19, sowie den Abschnitt 2.4.5 auf der Seite 19.

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
                   # $SP ← $SP − 4
```

3.3.8 `POP`

Assemblername	Parameter	Maschinencode	Format
`POP`	$X \in \mathcal{R}$	`0x19`	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]`

Entspricht

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

Fehler Falls das Register `SP` zu einer Adresse außerhalb des Stackbereichs zeigt, wird der Interrupt mit Nummer 16 (ungültige Speicheradresse) generiert und das Bit mit Nummer 8 im Register `ERR` gesetzt.

Beispiel Angenommen, die ersten 4 Bytes auf dem Stack (d.h. ab der Adresse, die im Register `SP` angegeben ist) sind `0xAA 0xBB 0xCC 0xDD`.

```
POPH R1       # R1 = 0xAABBCCDD
              # $SP ← $SP + 4
```

3.4 Arithmetische Instruktionen

3.4.1 `ADD`

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

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

$$X \leftarrow \$Y + \$Z$$

Fehler Falls das Ergebnis der Addition sich nicht mit 32 Bit darstellen lässt wird der Interrupt mit Nummer 2 generiert (arithmetischer Überlauf) und das Bit mit Nummer 1 im Register `ERR` gesetzt.

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.

3.4.2 `ADDU`

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

„Add Unsigned“. Vorzeichenlose Addition der Registerinhalte $\$Y$ 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
```

Fehler Wie bei `ADD`.

3.4.3 `ADDI`

Assemblername	Parameter	Maschinencode	Format
`ADDI`	$X, Y \in \mathcal{R}$, $N \in \mathbb{Z}$	`0x32`	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 + (−3) = 0
ADDI  R2 R3 R4  # Fehler! R4 kein n ∈ ℤ
```

Fehler Wie bei ADD.

3.4.4 SUB

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

Fehler Wie bei ADD.

3.4.5 SUBU

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

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

Fehler Wie bei ADD.

3.4.6 SUBI

Assemblername	Parameter	Maschinencode	Format
SUBI	$X, Y \in \mathcal{R}$, $N \in \mathbb{Z}$	0x35	RRN

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

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

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
```

Fehler Wie bei ADD.

3.4.7 SUBI2

Assemblername	Parameter	Maschinencode	Format
SUBI2	$X, Y \in \mathcal{R}$, $N \in \mathbb{Z}$	0x36	RRN

„Subtract Immediate“. Funktioniert wie SUBI mit dem Unterschied, dass bei der Subtraktion die Reihenfolge der Operanden $\$Y$ und N umgekehrt ist. Entspricht also

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

Beispiel Folgendes Beispiel demonstriert die Verwendung von SUBI2.

```
SET   R1 50      # R1 ← 50
SUBI2 R2 R1 30   # R2 ← (30 − $R1) = −20
```

Fehler Wie bei ADD.

3.4.8 MUL

Assemblername	Parameter	Maschinencode	Format
MUL	$X, Y \in \mathcal{R}$	0x38	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 (high) und die 32 niedrigstwertigen Bits des Ergebnisses in das Register LO (low) gespeichert. Siehe auch die Tabelle 2.1 auf der Seite 14.

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
CP    R2 LO    # R2 ← $R1^2 mod 2^32
```

Der Befehl CP kopiert den Inhalt des Registers LO in das Register $R2$.

3.4.9 MULU

Assemblername	Parameter	Maschinencode	Format
MULU	$X, Y \in \mathcal{R}$	0x39	RR0

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

3.4.10 `MULI`

Assemblername	Parameter	Maschinencode	Format
`MULI`	$X \in \mathcal{R}$, $N \in \mathbb{Z}$	`0x3A`	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`.

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

Das N mit 16 Bit (2 Byte gemäß Format RNN) dargestellt wird, kann es nur Werte zwischen -2^{15} und $2^{15} - 1$ annehmen (Zweierkomplement).

3.4.11 `DIV`

Assemblername	Parameter	Maschinencode	Format
`DIV`	$X, Y \in \mathcal{R}$	`0x3B`	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 (integer division).

Fehler Falls der Wert im Register Y gleich Null ist, wird der Interrupt mit Nummer 1 erzeugt (Division durch Null) und das Bit mit Nummer 0 im Register `ERR` gesetzt.

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
```

3.4.12 `DIVU`

Assemblername	Parameter	Maschinencode	Format
`DIVU`	$X, Y \in \mathcal{R}$	`0x3C`	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. Mit anderen Worten:

$$\$X, \$Y, \$LO, \$HI \in \mathbb{N} \mod 2^{32}$$

Fehler Wie bei `DIV`.

3.4.13 `DIVI`

Assemblername	Parameter	Maschinencode	Format
`DIVI`	$X \in \mathcal{R}$, $N \in \mathbb{Z}_{\setminus 0}$	`0x3D`	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 $[-2^{15}, 2^{15} - 1] \setminus \{0\}$. Entspricht

$$(HI, LO) \leftarrow \left(\frac{\$X}{N}\right)$$

Nach der Durchführung der Division gilt:

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

beziehungsweise

$$HI \leftarrow \left\lfloor \frac{\$X}{N} \right\rfloor, \qquad LO \leftarrow \$X \bmod N$$

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

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

Fehler Wie `DIV`.

3.4.14 `DIVI2`

Assemblername	Parameter	Maschinencode	Format
`DIVI2`	$X \in \mathcal{R}$, $N \in \mathbb{Z}_{\setminus 0}$	`0x3E`	RNN

„Divide Immediate“. Funktioniert wie `DIVI` mit dem Unterschied, dass bei der Division werden die Operanden $\$X$ und N vertauscht. Entspricht also

$$(HI, LO) \leftarrow \left(\frac{N}{\$X}\right)$$

beziehungsweise

$$HI \leftarrow \left\lfloor \frac{N}{\$X} \right\rfloor, \qquad LO \leftarrow N \bmod \$X$$

3.4.15 `ABS`

Assemblername	Parameter	Maschinencode	Format
`ABS`	$X, Y \in \mathcal{R}$	`0x40`	RR0

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

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

oder

$$X \leftarrow \left(|\$Y| \bmod 2^{31} \right)$$

3.4.16 NEG

Assemblername	Parameter	Maschinencode	Format
NEG	$X, Y \in \mathcal{R}$	0x41	RR0

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

$$X \leftarrow \left((-1) \cdot \$Y \right)$$

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

3.4.17 INC

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

„Increment“. Inkrementiert den Inhalt des Registers X.

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

Fehler Wie bei ADD, als würde man mit Eins addieren.

3.4.18 DEC

Assemblername	Parameter	Maschinencode	Format
DEC	$X \in \mathcal{R}$	0x43	R00

„Decrement“. Dekrementiert den Inhalt des Registers X.

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

Fehler Wie bei `SUB`, als würde man Eins subtrahieren.

3.4.19 `MOD`

Assemblername	Parameter	Maschinencode	Format
`MOD`	$X, Y, Z \in \mathcal{R}$	`0x48`	RRR

Modulo Operation. Dieser Befehl berechnet den Rest der Division $\$Y/\$Z$ und speichert den Rest in das Register X. Nach Definition der Modulo-Funktion

$$\mathrm{mod} : \mathbb{Z} \times (\mathbb{Z} \setminus \{0\}) \to \mathbb{Z}$$

kann das Ergebnis der Modulo-Funktion positiv oder negativ sein (Ergebnis liegt im $\mathbb{Z}$).

Algebraische Schreibweise:

$$X \leftarrow \$Y \bmod \$Z, \qquad \$Z \neq 0$$

Fehler Falls der Inhalt des Registers Z gleich Null ist, wird der Interrupt mit Nummer 1 (Division durch Null) erzeugt.

3.4.20 `MODI`

Assemblername	Parameter	Maschinencode	Format
`MODI`	$X, Y \in \mathcal{R},\ N \in \mathbb{Z}_{\setminus 0}$	`0x49`	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, die Werte aus dem Intervall $[1, 255]$ nimmt.

$$X \leftarrow \$Y \bmod N, \qquad N \in [1, 255]$$

Wird $N = 0$ angegeben, so wird der Interrupt mit Nummer 1 (Division durch Null) erzeugt.

3.4.21 MODI2

Assemblername	Parameter	Maschinencode	Format
MODI2	$X, Y \in \mathcal{R}$, $N \in \mathbb{Z}_{\setminus 0}$	0x4A	RRN

„Modulo Immediate“. Funktioniert und hat die selbe Syntax wie der Befehl MODI mit dem Unterschied, dass bei der Modulo-Bildung (Division) werden die Operanden $\$Y$ und N vertauscht.

$$X \leftarrow N \bmod \$Y, \qquad N \in [1, 255]$$

Fehler Wie bei MODI.

3.5 Logische Instruktionen

3.5.1 AND

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

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

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

3.5.2 ANDI

Assemblername	Parameter	Maschinencode	Format
ANDI	$X, Y \in \mathcal{R}$, $N \in \mathbb{N}$	0x51	RRN

„And Immediate“. Berechnet bitweise $\$Y \wedge N$ (und-Verknüpfung) und speichert das Ergebnis in das Register X. N ist dabei eine feste natürliche 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:

$$X \leftarrow (\$Y \wedge (N \bmod 256)), \qquad N \in [0, 255]$$

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. Da diese Bits auf Null gesetzt sind, wird das Ergebnis immer kleiner als $2^8 = 256$ sein.

Beispiel für die Verwendung von `ANDI`:

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

3.5.3 `OR`

Assemblername	Parameter	Maschinencode	Format
`OR`	$X, Y, Z \in \mathcal{R}$	`0x52`	RRR

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

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

3.5.4 `ORI`

Assemblername	Parameter	Maschinencode	Format
`ORI`	$X, Y \in \mathcal{R}$, $N \in \mathbb{N}$	`0x53`	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.

$$X \leftarrow (\$Y \vee (N \bmod 256)), \qquad N \in [0, 255]$$

3.5.5 XOR

Assemblername	Parameter	Maschinencode	Format
XOR	$X, Y, Z \in \mathcal{R}$	0x54	RRR

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

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

3.5.6 XORI

Assemblername	Parameter	Maschinencode	Format
XORI	$X, Y \in \mathcal{R}$, $N \in \mathbb{N}$	0x55	RRN

„Xor Immediate“. Analog zur Instruktion XOR mit dem Unterschied, dass N eine direkt angegebene Zahl aus dem Intervall $[0, 255]$ ist. Wird eine Zahl angegeben, die größer als 255 ist, die also mehr als 8 Bit braucht, wird sie auf 8 Bit reduziert.

Die Zahl N wird links auf die Länge von 32 Bit mit Nullen aufgefüllt.

$$X \leftarrow (\$Y \oplus (N \bmod 256)), \qquad N \in [0, 255]$$

3.5.7 NOT

Assemblername	Parameter	Maschinencode	Format
NOT	$X, Y \in \mathcal{R}$	0x56	RR0

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

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

oder

$$X \leftarrow \neg\$Y$$

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

```
NOT R1 R2 # R1 ← ¬R2
```

3.5.8 NOTI

Assemblername	Parameter	Maschinencode	Format
NOTI	$X \in \mathcal{R}$, $N \in \mathbb{N}$	0x57	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.

$$X \leftarrow \overline{(N \bmod 256)}, \qquad N \in [0, 255]$$

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

```
NOTI R1 5 # R1 ← ¬5
```

3.5.9 NAND

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

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

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

oder

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

3.5.10 NANDI

Assemblername	Parameter	Maschinencode	Format
NANDI	$X, Y \in \mathcal{R}$, $N \in \mathbb{N}$	0x59	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. $N \in [0, 255]$.

$$X \leftarrow \Big(\$Y \mathbin{\overline{\wedge}} (N \bmod 256)\Big), \qquad N \in [0, 255]$$

Die Zahl N wird auf die Länge von 32 Bit links mit Nullen aufgefüllt.

3.5.11 NOR

Assemblername	Parameter	Maschinencode	Format
NOR	$X, Y, Z \in \mathcal{R}$	0x5A	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“.

$$X \leftarrow (\$Y \mathbin{\overline{\vee}} \$Z)$$

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

3.5.12 NORI

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

3.5.13 SHL

Assemblername	Parameter	Maschinencode	Format
SHL	$X, Y, Z \in \mathcal{R}$	0x60	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.

Die Stellenangabe im Register Z muss positiv sein, bzw. wird als eine vorzeichenlose Zahl interpretiert. $\$Z \in \mathbb{N}$.

Andere Schreibweise:

$$X \leftarrow \$Y << \$Z$$

oder

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

3.5.14 SHLI

Assemblername	Parameter	Maschinencode	Format
SHLI	$X, Y \in \mathcal{R}$, $N \in \mathbb{N}$	0x61	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.

N muss positiv sein und es wird als solches interpretiert: $N \in \mathbb{N}$.

3.5.15 SHR

Assemblername	Parameter	Maschinencode	Format
SHR	$X, Y, Z \in \mathcal{R}$	0x62	RRR

„Shift Right logical“. 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: $\$X, \$Y, \$Z \in \mathbb{N}$.

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

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
```

3.5.16 `SHRI`

Assemblername	Parameter	Maschinencode	Format
`SHRI`	$X, Y \in \mathcal{R}$, $N \in \mathbb{N}$	`0x63`	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`.

$$X \leftarrow (\$Y >> N)$$

3.5.17 SHRA

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

3.5.18 SHRAI

Assemblername	Parameter	Maschinencode	Format
SHRAI	$X, Y \in \mathcal{R}$, $N \in \mathbb{N}$	0x65	RRN

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

3.5.19 ROTL

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

„Rotate Left“. Die Bits aus dem Register Y werden $\$Z$ Stellen nach links „rotiert“ und das Ergebnis in das Register X gespeichert.

$$X \leftarrow \$Y \circlearrowleft \$Z$$

Der Registerinhalt $\$Z$ wird als vorzeichenlose Zahl interpretiert: $\$Z \in \mathbb{N}$.

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
```

3.5.20 ROTLI

Assemblername	Parameter	Maschinencode	Format
ROTLI	$X, Y \in \mathcal{R}$, $N \in \mathbb{N}$	0x69	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 um 6 Stellen
ROTL R4 R1 -6 # Fehler! da −6 ∉ ℕ
```

3.5.21 ROTR

Assemblername	Parameter	Maschinencode	Format
ROTR	$X, Y, Z \in \mathcal{R}$	0x6A	RRR

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

$$X \leftarrow \$Y \circlearrowright \$Z$$

3.5.22 ROTRI

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

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

3.6 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 CMPR gespeichert (siehe auch Tabelle 2.1 auf der Seite 14). Dieses Ergebnis ist -1, 0 oder $+1$ und wird folgenderweise berechnet: werden zwei Werte x und y vergliechen, so wird das Register CMPR wie folgt gesetzt:

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

Alle diese Befehle generieren den Interrupt mit Nummer 9 (ungültige Registernummer), falls die angegebene Registernummern ungültig sind – das heißt, falls es die Register nicht gibt.

3.6.1 CMP

Assemblername	Parameter	Maschinencode	Format
CMP	$X, Y \in \mathcal{R}$	0x70	RR0

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

3.6.2 CMPU

Assemblername	Parameter	Maschinencode	Format
CMPU	$X, Y \in \mathcal{R}$	0x71	RR0

„Compare Unsigned“. Vergleicht analog zu CMP – aber vorzeichenlos – $\$X$ mit $\$Y$ (die Inhalte der Register X und Y werden als vorzeichenlose Werte ausgewertet, $\$X, \$Y \in \mathbb{N}$).

3.6.3 CMPI

Assemblername	Parameter	Maschinencode	Format
CMPI	$X \in \mathcal{R}$, $N \in \mathbb{Z}$	0x72	RNN

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

3.7 Sprungbefehle

Alle Sprungbefehle, außer dem GO Befehl, veranlassen einen relativen Sprung im Programmcode – relativ im Sinne, dass die Parameter der Sprungbefehle einen ganzzahligen (negativen oder positiven) Versatz zur aktuellen Programmadresse angeben. Dabei bedeutet der Versatz, oder Offset, die wievielte Instruktionen muss als nächste ausgeführt werden. Zum Beispiel die Instruktion

```
JMP 2
```

bedeutet „Die zweite Instruktion nach dieser soll ausgeführt werden“. Also es wird eine Instruktion übersprungen, und mit der zweiten fortgefahren. Die Instruktion

```
BE -5
```

bedeutet „Falls das Register CMPR den Wert 0 hat, überspringe 4 Instruktionen zurück (da negativ) und fahre mit der 5. fort“, also die 5. Instruktion zurück.

In dem Programm

```
SET R1 1
SET R2 2
JMP -2
```

bewirkt der JMP-Befehl einen Sprung an die erste Instruktion zurück, also an die zweite zurück.

Der Sprungversatz (Offset) wird stets nicht in Bytes angegeben, sondern in Instruktionen – wobei eine Instruktion der UMach Maschine 4 Bytes beträgt. Die Maschine multipliziert diese Anzahl mit 4, um die richtige Programmadresse zu berechnen.

Die Sprungbefehle bauen auf die Vergleichsbefehle auf (Abschnitt 3.6): jeder Sprungbefehl (außer `JMP` und `GO`) untersuchen das Spezialregister `CMPR` und verzweigen die Programmausführung anhand seines Wertes.

Bemerkung Es ist zu bemerken, dass die hier verwendeten nummerischen Versatzwerte sich auf die interne Funktionsweise der UMach Maschine beziehen. Ein Assembler könnte solche nummerische Angaben für ungültig erklären und statt dessen, textuelle Sprungmarken erwarten (Labels).

3.7.1 `BE`

Assemblername	Parameter	Maschinencode	Format
`BE`	$N \in \mathbb{Z}$	`0x80`	NNN

„Branch Equal“. Wenn das Register `CMP` den Wert 0 hat, wird zur N-ten Instruktion 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:

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

Die Multiplikation mit 4 wird deshalbt ausgeführt, weil ein Befehl immer aus 4 Bytes besteht, sodass der Adressoffset zwischen zwei Befehlen immer 4 ist.

Die Subtraktion von 4 wird deshalbt ausgeführt, weil die Maschine nach jeder Instruktion das Register PC automatisch um 4 inkrementiert, so dass eine Korrektur gebraucht wird.

Fehler Falls das Sprungziel eine Speicheradresse ist, die außerhalb des Code-Segments liegt, wird der Interrupt mit Nummer 17 generiert (Zugriffsverletzung, segfault) und das Bit mit Nummer 9 im Register `ERR` gesetzt (siehe auch Tabelle 2.2 auf Seite 16).

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 möglicher arithmetischer Überlauf wird nicht berücksichtigt.

```
  SET   R1 100    # R1 = 100
  LB    R2 R1     # Lade Byte von Adresse 100 nach R2
  INC   R1        # R1++, R1 = 101
  LB    R3 R1     # Lade Byte von Adresse 101 nach R3
  CMP   R2 R3     # Vergleiche Inhalt von R2 und R3
                  # Ergebnis geht ins CMPR
  #BE    equal    # Asm Schreibweise
  BE    5         # Wenn CMPR gleich 0 ist, springe
                  # zur 5. Instruktion (MULI)
                  # (gehe zum label equal)
  ADD   R2 R2 R3  # Addiere Inhalt von R2 und R3
  DEC   R1        # R1--, R1 = 100
  SB    R2 R1     # Speichere R2 nach Adresse 100
  #JMP   finish   # Asm Schreibweise
  JMP   3         # Gehe zur 3. Instruktion (label finish)

equal:
  MULI  R2 2      # Multipliziere Inhalt von R2 mit 2
  SB    LO R1     # Speichere Inhalt von LO nach Adresse in R1
finish:
  EOP
```

3.7.2 BNE

Assemblername	Parameter	Maschinencode	Format
BNE	$N \in \mathbb{Z}$	0x81	NNN

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

Fehler Wie bei der Instruktion BE.

3.7.3 BL

Assemblername	Parameter	Maschinencode	Format
BL	$N \in \mathbb{Z}$	0x82	NNN

„Branch Less“. Springt zur N-ten Instruktion, wenn der Inhalt von CMPR kleiner 0 ist.

Fehler Wie bei der Instruktion BE.

3.7.4 BLE

Assemblername	Parameter	Maschinencode	Format
BLE	$N \in \mathbb{Z}$	0x83	NNN

„Branch Less Equal“. Springt zur N-ten Instruktion, wenn der Inhalt von CMPR kleiner oder gleich 0 ist.

Fehler Wie bei der Instruktion BE.

3.7.5 BG

Assemblername	Parameter	Maschinencode	Format
BG	$N \in \mathbb{Z}$	0x84	NNN

„Branch Greater“. Springt zur N-ten Instruktion, wenn der Inhalt von CMPR größer als 0 ist.

Fehler Wie bei der Instruktion BE.

3.7.6 BGE

Assemblername	Parameter	Maschinencode	Format
BGE	$N \in \mathbb{Z}$	0x85	NNN

„Branch Greater Equal“. Springt zur N-ten Instruktion, wenn der Inhalt von CMPR größer oder gleich 0 ist.

Fehler Wie bei der Instruktion BE.

3.7.7 JMP

Assemblername	Parameter	Maschinencode	Format
JMP	$N \in \mathbb{N}$	0x88	NNN

„Jump“. Springt zur N-ten Instruktion, unabhängig vom Wert des Registers $CMPR$.

Fehler Wie bei der Instruktion BE.

3.8 Unterprogramminstruktionen

3.8.1 GO

Assemblername	Parameter	Maschinencode	Format
GO	$X \in \mathcal{R}$	0x90	R00

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 (die Maschine prüft nicht, ob die angegebene Adresse $\$X$ ein Vielfaches von 4 ist).

$$PC \leftarrow \$X$$

Fehler Falls das Sprungziel $\$X$ eine Speicheradresse ist, die außerhalb des Code-Segments liegt, wir der Interrupt mit Nummer 17 generiert (Zugriffsverletzung, segfault) und das Bit mit Nummer 9 im Register ERR gesetzt (siehe auch Tabelle 2.2 auf Seite 16).

3.8.2 CALL

Assemblername	Parameter	Maschinencode	Format
CALL	$N \in \mathbb{N}$	0x91	NNN

Funktionert wie der Befehl JMP mit dem Unterschied, dass bevor PC neugesetzt wird, wird es auf den Stack gepusht. Enspricht

```
PUSH PC
JMP N
```

Beispiel Dieser Befehl dient zur Implementierug und Verwendung von Funktionen (function call). Verwendet man diesen Befehl zum Sprung zu einem anderen Programmteil (Aufruf einer Subroutine), so kann aus diesem Teil zurückgekehrt werden, indem man den Befehl RET verwendet. Dies ermöglicht ein Code wie der folgende:

```
CALL routine
CP   R1 R32
EOP

routine:
  SET R32 66
  RET
```

Hier wird nach der Ausführung des ersten Befehls (ein CALL) mit der 4. Instruktion weitergemacht (nach dem Label routine). Der letzte Befehl (RET) bewirkt ein Sprung nach dem CALL-Befehl, also zum Befehl CP. Der letzte Befehl EOP wird verwendet, um das Ende des Programms zu signalisieren, sonst würde die Maschine wieder die Routine ausführen, wobei der Befehl RET einen Fehler erzeugen würde (da dieser Befehl die Return-Adresse aus dem Stack holt und dort steht möglicherweise keine Adresse).

Fehler Wie bei dem Befehl GO.

3.8.3 RET

Assemblername	Parameter	Maschinencode	Format
RET	keine	0x92	000

POPt eine Adresse vom Stack in das Register PC. Dieser Befehl entspricht einem Rückkehr aus einer Subroutine, analog der Anweisung return aus den höheren Programmiersprachen.

```
POP PC
```

Dieser Befehl wird in Zusammenspiel mit den Befehlen CALL und INT verwendet.

Fehler Wie bei der `POP` und `GO` Operation.

3.9 Systeminstruktionen

3.9.1 `INT`

Assemblername	Parameter	Maschinencode	Format
`INT`	$N \in \mathbb{N}$	`0xA0`	NNN

„Interrupt“. Generiert ein Interrupt in der Programmausführung. Die Zahl N ist 3 Byte groß und wird als Interruptnummer interpretiert (siehe dazu auch den Abschnitt 2.4.2 auf der Seite 17). Falls die Interruptnummer N keine von der Maschine erkannte Interruptnummer ist, wird der Interrupt mit Nummer 0 ausgeführt. Für mehr Informationen betreffend das Interrupt-System der UMach Maschine siehe den Abschnitt 2.6 auf der Seite 22.

Der Befehl `INT` speichert, wie der Befehle `CALL` auch, den aktuellen Wert des Registers `PC` auf den Stack. Dies erlaubt einem Interrupt-Handler, zur normalen Ausführung des Programms zurückzukehren, indem er den Befehl `RET` verwendet. Ein Interrupt-Handler ist somit wie eine normale Subroutine, die entweder direkt mittels `CALL`, oder indirekt mit dem Befehl `INT` aufgerufen werden kann.

3.10 IO Instruktionen

3.10.1 `IN`

Assemblername	Parameter	Maschinencode	Format
`IN`	$A, N, P \in \mathcal{R}$	`0xB0`	RRR

Veranlasst die I/O-Einheit, $\$N$ Bytes Daten aus dem Gerät am Port $\$P$, zum Speicher ab der Adresse $\$A$ zu übertragen. A, N und P sind Registernummer.

$$read(\$P) \rightarrow mem_{\$N}(\$A)$$

Dieser Befehl blockiert den Kern solange, bis entweder $\$N$ Bytes gelesen wurden, oder bis das Gerät das Ende der Übertragung signalisiert. Die tatsächliche Anzahl der übetragenen Bytes kann kleiner als $\$N$ sein und wird in das Register N gespeichert.

Fehler Falls die Längenangabe $\$N$ oder die Speicheradresse $\$A$ negativ sind, wird der Interrupt mit Nummer 10 ausgelöst (ungültiges Argument). Falls $\$P$ keine gültige Portnummer ist, wird der Interrupt mit Nummer 32 (nicht existentes I/O Port) generiert und das Bit mit Nummer 16 im Register `ERR` gesetzt. Falls die Speicheradressen $\$A$ bis $\$A + \$N$ keine gültigen Speicheradressen sind, d.h. falls es sie gar nicht gibt, wird der Interrupt mit Nummer 16 (ungültige Speicheradresse) generiert und das Bit mit Nummer 8 im Register `ERR` gesetzt. Falls der Speicherbereich $\$A$ bis $\$A + \$N$ schreibgeschützt ist, wird der Interrupt mit Nummer 17 (segfault) erzeugt und das Bit mit Nummer 9 im Register `ERR` gesetzt.

Siehe auch die Tabelle 2.4 auf der Seite 23.

3.10.2 `OUT`

Assemblername	Parameter	Maschinencode	Format
`OUT`	$A, N, P \in \mathcal{R}$	`0xB8`	RRR

Schreibt $\$N$ Bytes Daten aus dem Speicher ab der Adresse $\$A$ an den Port mit Nummer $\$P$. Die Anzahl der tatsächlich geschriebenen Bytes wird zurück in das Register N geschrieben und beträgt 0 bis $\$N$. Eine niedrigere Zahl as $\$N$ wird z.B. dann erreicht, wenn das Ausgabegerät die Ausgabe aus technischen Gründen verweigert (ausgeschaltet) oder nicht in der gewünschten Anzahl ausführen kann.

Dieser Befehl blockiert den Kern solange, bis die I/O-Einheit fertig mit dem Schreiben ist.

Fehler Wie bei dem Befehl `IN`.

Beispiel Der folgende Code demonstriert die Benutzung der `IN` und `OUT` Befehle in einem „Hello User“ Programm. Der Benutzer wird nach seinem Namen gefragt und dann wird er namentlich begrüßt.

```
SET R1 prompt   # Adresse der ersten Ausgabe
SET R2 10       # Länge der Ausgabe
SET R3 0        # Portnummer 0
OUT R1 R2 R3    # Ausgeben "Your name: "

SET R1 name     # Speicheradresse
SET R2 16       # wieviel input maximal
SET R3 0        # Port 0 (Tastatur)
                # Port 0 für IN und OUT sind getrennt
IN  R1 R2 R3    # oder auch IN R1 R2 ZERO
### blockiert bis input fertig ###
### in R2 steht wieviel tatsächlich gelesen ###

SET R4 hello
SET R5 7        # strlen(hello)
OUT R4 R5 R3    # output R5 bytes auf Port 0
OUT R1 R2 ZERO  # "Hello, User"

SET R4 nl
SET R5 1
OUT R4 R5 ZERO  # '\n'

.data
array 16 name
string promtp "Your name: "
string hello  "Hello, "
string nl     "0x10"
```

3.11 Befehlentabelle

Die Tabelle 3.3 auf der Seite 65 gibt eine Übersicht aller Befehlen, die von der UMach Maschine erkannt und ausgeführt werden.

Tabelle 3.3: Befehlentabelle

	0	1	2	3	4	5	6	7
0	NOP				EOP			
1	SET	CP	LB	LW	SB	SW		
	PUSH	POP						
2								
3	ADD	ADDU	ADDI	SUB	SUBU	SUBI	SUBI2	
	MUL	MULU	MULI	DIV	DIVU	DIVI	DIVI2	
4	ABS	NEG	INC	DEC				
	MOD	MODI	MODI2					
5	AND	ANDI	OR	ORI	XOR	XORI	NOT	NOTI
	NAND	NANDI	NOR	NORI				
6	SHL	SHLI	SHR	SHRI	SHRA	SHRAI		
	ROTL	ROTLI	ROTR	ROTRI				
7	CMP	CMPU	CMPI					
8	BE	BNE	BL	BLE	BG	BGE		
	JMP							
9	GO	CALL	RET					
A	INT							
B	IN							
	OUT							
C								
D								
E								
F								
	8	9	A	B	C	D	E	F

Index