UMach Spezifikation

27. Juli 2012

Inhaltsverzeichnis

Tabellenverzeichnis

Abbildungsverzeichnis

1 Einführung

UMach ist eine einfache programmierbare virtuelle Maschine (VM), die einen definierten Instruktionssatz und eine definierte Architektur hat. UMach orientiert sich dabei an Prinzipien von RISC Architekturen: feste Instruktionslänge, kleine Anzahl von einfachen Befehlen, Speicherzugriff durch Load- und Store-Befehlen, usw. Die UMach Maschine ist Register-basiert. Der genaue Aufbau dieser Rechenmaschine ist im Abschnitt 2.1 ab der Seite 8 beschrieben.

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

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

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

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

```
ADDI R1 R2 5 # Maschinencode 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 eine einfache Konstruktion, die 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 auch die Abbildung 2.1 auf der Seite 9, die die UMach Maschine darstellt.

Wenn die Maschine startet, sucht sie nach einem Program im Speicher, holt jede Instruktion nach einander in den Kern und führt sie aus. Danach bleibt sie im Wartezustand bis sie ausdrücklich ausgeschaltet wird. Alle Eingabe- und Ausgabeoperationen werden an die I/O-Einheit weitergeleitet (siehe auch die Abschnitte 2.5 und 3.10).

2.1.1 Betriebsmodi

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

1. Normalmodus
2. Einzelschrittmodus

Der standard-Modus ist der Normalmodus. Der Modus wird vor dem Hochfahren der Maschine festgelegt.

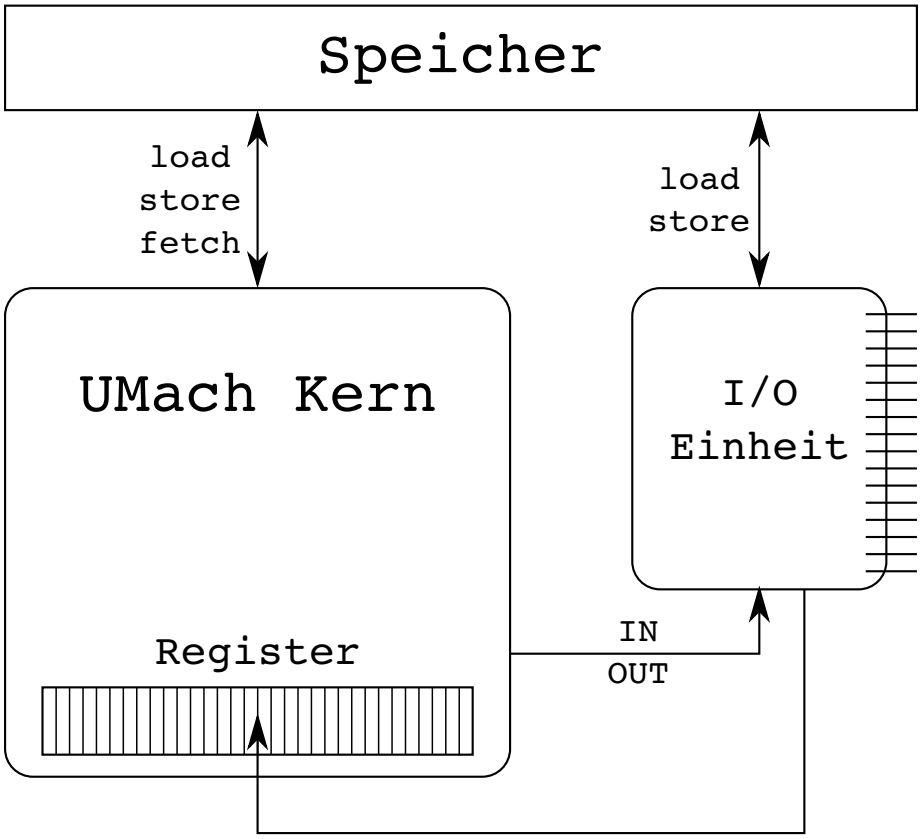

Abbildung 2.1: Aufbau der UMach Maschine

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

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

2.1.2 Von Neumann Zyklus

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

In einem dritten Schritt Execute wird die Instruktion ausgeführt. Der theoretisch 4. Schritt, das Inkrementieren des Programmcounters PC, wird parallel zu den Fetch und Decode Vorgängen ausgeführt. Diese Tatsache ist bei Instruktionen, welche den Inhalt des PC manipulieren, zu berücksichtigen. Siehe auch Abbildung 2.2 auf der Seite 11.

Auflistung der Schritte:

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

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

3. Execute – Auführen des Befehles in der ALU.

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

2.2 Kern

Der Kern der Maschine besteht aus den folgenden Komponenten:

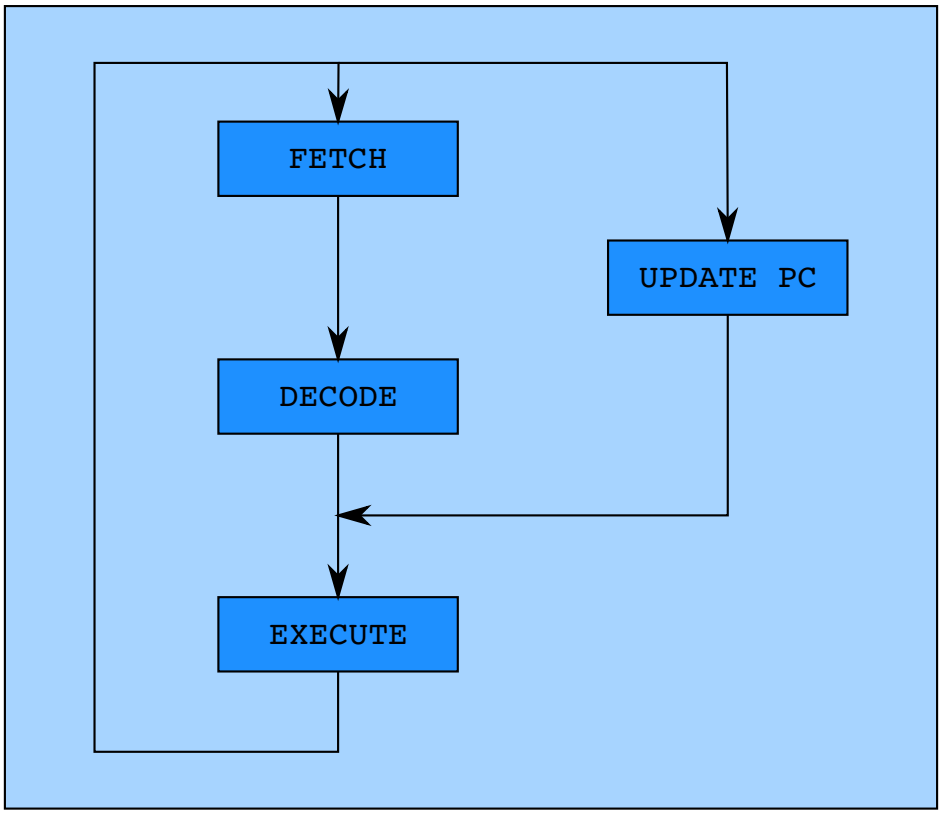

Abbildung 2.2: Von Neumann Zyklus

1. Befehlsabruf Einheit (Instruction Fetch Unit), die die Programminstruktionen der Reihe nach aus dem Speicher holt und der Decodierungseinheit übergibt.

2. Decodierungseinheit, die dafür zuständig ist, eine Instruktion zu decodieren, bzw. in ihren Komponenten zu zerteilen.

3. Recheneinheit, die für die tatsächliche Ausführung der Instruktionen zuständig ist.

4. Register, kleine Speichereinheiten, die sich im Kern befinden.

Der Kern besitzt kein Pipeline.

2.3 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 natürliche Nummer, die innerhalb der Maschine das Register identifiziert. Diese Nummer heißt Registernummer und wird von einer Instruktion auf Maschinencode-Ebene verwendet, wenn sie das Register anspricht – mit anderen Worten, auf Maschinecode-Ebene wird das Register durch seine Nummer angegeben.

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.

2.3.1 Allzweckregister

Es gibt 32 Allzweckregister, die dem Programmierer für allgemeine Zwecke zur Verfügung stehen. Diese 32 Register werden beim Hochfahren der Maschine mit dem Wert Null (`0x00`) initialisiert. 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 die Registernummer des Registers. Auf Assembler-Ebene, werden sie mit den Namen $R1, R2 \ldots$ bis $R32$ angesprochen (Assemblername). Die Zahl nach dem Buchstaben R ist im Dezimalsystem angegeben und ist fester Bestandteil des Registernamens.

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

Das Register mit Nummer Null, oder das Null-Register, ist ein Spezialregister, dass immer den Wert Null hat und nicht beschreibbar ist . Es wird mit dem Namen `ZERO` angesprochen (siehe auch den Abschnitt 2.3.2 auf Seite 13).

Beispiel für die Verwendung von Registernamen:

Assembler Maschinencode	`ADD` `0x50`	`R1` `0x01`	`R2` `0x02`	`R3` `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 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 Registernummern der Spezialregister setzen die Nummerierung der Allzweckregister zwar fort, die Assemblernamen aber nicht: es gibt kein Register $R33$. 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 steht der Maschinenname im Dezimalsystem, so wie er im Maschinencode steht. 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 Null gesetzt, wenn die Maschine hochfährt. Wird nach dem Abfangen einer Instruktion in das Register `IR` automatisch inkrementiert. Schreibgeschützt.
`SP`	34	„Stack Pointer“. Enthält die Speicheradresse des höchsten Eintrag 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 2.4.1, Seite 18.
`FP`	35	„Frame Pointer“. Enthält die Startadresse des Stack Frames einer Subroutine und unterstützt die Implementierung von Funktionen.
`IR`	36	„Instruction Register“. Enthält die gerade ausgeführte Instruktion. Schreibgeschützt.
`STAT`	37	Enthält Status-Informationen. Diese Informationen sind in den einzelnen Bits dieses Register gespeichert. Welche Informationen vorhanden sind liegt an der jeweiligen Anwendung.
`ERR`	38	„Error“. Fehlerregister. Die einzelnen Bits dises Registers geben Auskunft über Fehler, die mit der Ausführung des Programms verbunden sind. Liegt kein Fehler vor, so ist der Inhalt dieses Registers gleich Null. 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 15.
`HI`	39	„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`	40	„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	41	„Comparison Result“. Enthält das Ergebnis eines Vergleichs. Siehe auch Abschnitt 3.6, Seite 48.
ZERO	00	Enthält die Zahl Null. Schreibgeschützt.

Das ERR Register

Zu jedem Fehler gehört ein Bit im Register. 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 den folgenden 4 Gruppen eingeteilt:

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

Die Tabelle 2.2 listet alle Fehler auf, die in dem ERR Register signalisiert werden können.

Tabelle 2.2: Bedeutung der einzelnen Bits im ERR Register

0	Division durch Null
1	Arithmetischer Überlauf
5	Ungültiger Befehl
6	Ungültige Registernummer
8	Ungültige Speicheradresse
9	Stack Fehler
10	Stack Überlauf
11	Speicher unzureichend
16	I/O Port existiert nicht

Einige der genannten Fehler dürften näher erläutert werden.

Stack Fehler Ein Stack Fehler liegt vor, wenn eine ungültige Stack-Operation durchgeführt werden sollte. Zum Beispiel, wenn ein `POP`-Befehl angegeben wird, ohne vorher entsprechende Daten mittels `PUSH` auf den Stack gepusht zu haben.

Stack Überlauf Ein Stack Überlauf liegt vor, wenn der Stack Pointer (Register `SP`) auf einen Speicherbereich gesetzt wird, der nicht mehr zum Stack-Bereicht des Speichers gehört. Das passiert, wenn der Stack Pointer im Program-Bereich oder auf noch kleinere Adressen gesetzt wird.

Die UMach Maschine sollte solche kleine Werte nicht zulassen.

2.4 Der Speicher

Der Speicher wird in seiner Gesamtheit beim Hochfahren der Maschine mit dem Wert Null initialisiert. Seine Größe ist nicht festgelegt.

2.4.1 Speicherstruktur

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

1. Interrupttabelle
2. Programm und Daten
3. Stack

Siehe auch die Abbildung 2.3 auf der Seite 16.

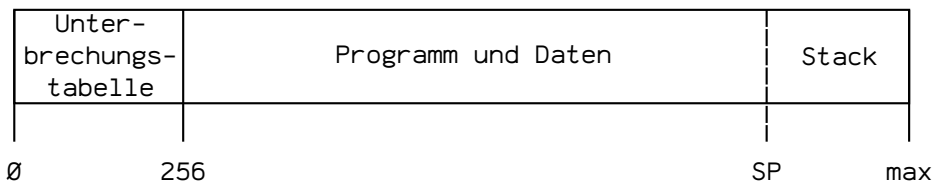

Abbildung 2.3: Speicherstruktur zur Laufzeit

Interrupttabelle

Die Interrupttabelle besteht aus einer Reihe von 32-Bit langen Sprungadressen zum ausführbaren Code, bzw. zu sogenannten Interruptroutinen, oder „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 als Index in dieser Tabelle verwendet.

Die Interrupttabelle wird von der Maschine nicht gefüllt, sondern es ist Sache der Software, die entsprechenden Stellen im Speicher zu füllen und entsprechende Funktionalität zur Verfügung zu stellen. Wird eine solche Adresse nicht gesetzt, d.h. ist der entsprechende Tabelleneintrag auf Null gesetzt, so reagiert die Maschine auf die Ausnahmesituation mit seiner Standardfunktion: die Maschine hält an. Für weitere Informationen bzgl. der Fehlerbehandlung siehe den Abschnitt 2.6, ab der Seite 20.

Die Interrupttabelle besteht aus 64 Einträgen, die 64 mögliche 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 die Abbildung 2.3 auf der Seite 16.

Die Tabelle 2.3 gibt alle definierten Interruptnummer und deren Bedeutung wieder. Undefinierte Interruptnummer werden nicht explizit angegeben.

Tabelle 2.3: Interruptnummer

Nummer	Beschreibung
0	Standardinterrupt: Maschine ausschalten
1	Division durch Null
2	Arithmetischer Überlauf
8	Ungültiger Befehl
9	Ungültige Registernummer
16	Ungültige Speicheradresse
24	Stack Fehler
26	Stack Überlauf
31	Speicher unzureichend
32	I/O Port existiert nicht

Programm und Daten

Nach der Interrupttabelle, die 256 Bytes groß ist (64 mal 4), folgt das eigentliche Program, das von der Maschine ausgeführt werden soll. Dieses Programm wird also ab der Adresse 256 gelesen und ausgeführt.

Insbesondere ist dieses Programm dafür zuständig, die Interrupttabelle zu füllen, falls (bestimmte) Interrupts behandelt werden sollten.

Der Stack

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. Wird versucht, den Inhalt von `SP` kleiner Null oder größer als die maximale Speicheradresse zu setzen, so wird dies von der Maschine verweigert und als Fehler im Register `ERR` signalisiert.

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

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]. Obwohl diese Architektur deutliche Einschränkungen hat, wird sie wegen ihrer konzeptionellen Einfachheit benutzt.

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 19). Die I/O-Einheit besitzt zu diesem Zweck einen direkten Zugriff auf den Speicher und kann dort unabhängig vom Kern lesen und schreiben (Direct

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

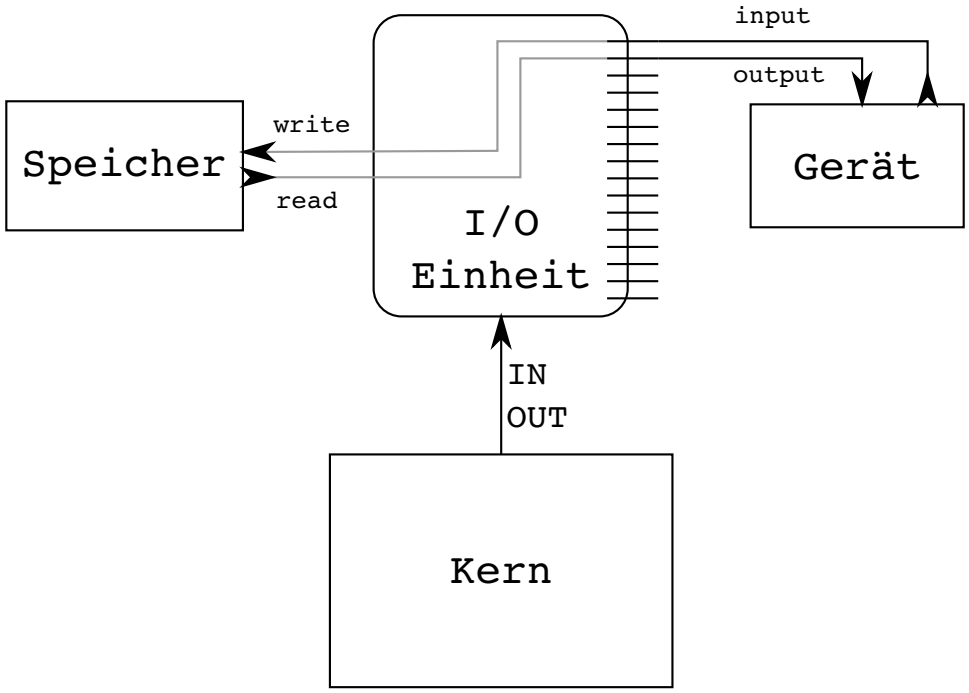

Abbildung 2.4: I/O wird von der I/O Einheit erledigt

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 den Abschnitt 3.10 auf Seite 54). Es werden also keine I/O Operationen ausgeführt, die nicht explizit durch Maschinenbefehle angefordert werden. Insbesondere werden keine I/O-Operationen aufgrund von Unterbrechungsanfoderungen (interrupts) initialisiert.

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 fertig mit dem Transfer zwischen Speicher und Peripherie ist, fährt der Kern die Ausführung 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, die die entsprechenden Daten generieren bzw. verarbeiten können. Siehe dazu auch die Abbildung 2.1 auf der 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 54 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. Eine zukünftige Version dieser Maschine könnte die Kontrolle der peripherischen Geräte hinzufügen (wie z.B. ein- und ausschalten der Geräte).

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 64 annimmt. Die Interruptnummer wird als Index in der Interrupttabelle verwendet, um die Speicheradresse einer Interruptroutine zu finden (siehe den Abschnitt 2.4.1, besonders die Tabelle 2.3 auf der Seite 17). 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. Jede solche Interruptroutine sollte folgendes beachten:

- Bevor die Routine aufgerufen wird, sichert die Maschine auf den Stack nur das Register `PC`. Andere Register wie `SP` müssen von der Routine selbst gesichert und wiederhergestellt werden.

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 2.3 auf der Seite 17). 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.

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). Instruktionen, die für ihren Informationsgehalt weniger als 32 Bit brauchen, wie z.B. `NOP`, werden mit Nullbits gefüllt. Alle Daten und Informationen, die mit einer Instruktion übergeben werden, müssen in diesen 32 Bit untergebracht werden.

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

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

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

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

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

000

erstes Byte	zweites Byte	drittes Byte	viertes Byte
Befehl	nicht verwendet		

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

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.

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.

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`.

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.

RRN

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

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

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

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

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.

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

3.1.2 Verteilung des Befehlsraums

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

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

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

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

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

5. Vergleichsinstruktionen, die einen Vergleich zwischen Registerinhalten angeben.

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

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

8. Systeminstruktionen, die die Unterstützung eines Betriebssystem ermöglichen.

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 27 angegeben.

Die Tabelle 3.3 auf der Seite 56 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 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.

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.

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 22 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 selbst.

Die Tabelle 3.2 auf der Seite 27 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: $\ldots, -2, -1, 0, 1, 2, \ldots$

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

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

3.2 Kontrollinstruktionen

3.2.1 `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.

3.2.2 `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

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);
```

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.

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
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 Speicherstelle Y. X und Y sind dabei Register.

Entspricht dem algebraischen Ausdruck

$$X \rightarrow mem_1(Y)$$

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

```
SET R1 128      # R1 = Speicheradresse 128
SET R2 513      # R2 = 0x0201
SB  R2 R1       # 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 $Y+3$.

$$X \rightarrow mem_4(Y)$$

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

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
```

3.3.8 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	$2^{31} \leftrightarrow 2^{24}$	$2^{23} \leftrightarrow 2^{16}$	$2^{15} \leftrightarrow 2^{8}$	$2^{7} \leftrightarrow 2^{0}$
	$\uparrow$	$\uparrow$	$\uparrow$	$\uparrow$
Stack-Bereich	`mem[SP + 0]`	`mem[SP + 1]`	`mem[SP + 2]`	`mem[SP + 3]`

Diese Instruktion kann algebraisch so ausgedrückt werden:

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

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
```

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

$$X \leftarrow Y + Z$$

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

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

3.4.3 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 + (−2) = 1
ADDI  R2 R3 R4   # Fehler! R4 kein n ∈ Z
```

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.

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.

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).

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
```

3.4.7 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` und die 32 niedrigstwertigen Bits des Ergebnisses in das Register `LO` 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
COPY  R2 LO    # R1^2 mod 2^32
```

3.4.8 `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.

3.4.9 `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`.

3.4.10 `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.

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
```

3.4.11 `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.

3.4.12 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 $[-2^{15}, 2^{15} - 1] \setminus 0$. Nach der Durchführung der Division gilt:

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

Beispiel Der folgende Code demonstriert die Verwendung von DIVI.

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

3.4.13 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:

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

3.4.14 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:

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

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

3.4.15 `INC`

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

„Increment“. Inkrementiert den Inhalt des Registers X.

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

3.4.16 `DEC`

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

„Decrement“. Dekrementiert den Inhalt des Registers X.

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

3.4.17 `MOD`

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

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

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

Algebraische Schreibweise:

$$X \leftarrow Y \bmod Z$$

oder

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

3.4.18 MODI

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

$$X \leftarrow Y \bmod N$$

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 Zahl aus dem Bereich $[0, 255]$. Wird eine größere Zahl angegeben, so wird sie modulo 256 berechnet – mit anderen Worten, nur das letzte Byte zählt. Andere Schreibweise:

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

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

Beispiel für die Verwendung von `ANDI`:

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

3.5.3 `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.

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

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.

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

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.

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

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.

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

```
NOT R1 5 # R1 ← 5̄
```

3.5.9 NAND

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

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

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

oder

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

3.5.10 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. $N \in [0, 255]$.

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

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“.

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

```
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 in Z muss positiv sein.

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.

3.5.15 SHR

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

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

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

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

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

3.5.16 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.

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 so viele Stellen nach links „rotiert“ wie der Inhalt des Registers Z und das Ergebnis in das Register X gespeichert.

$$X \leftarrow Y \circlearrowleft Z$$

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

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

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

3.5.20 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 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 `CMP` gespeichert. Dieses Ergebnis ist -1, 0 oder $+1$ und wird folgenderweise berechnet: werden zwei Werte x und y vergliechen, so wird das Register `CMP` wie folgt gesetzt:

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

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).

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 $[-2^{15}, 2^{15} - 1]$, entsprechend dem Datentyp „Half“.

3.7 Übersprungsbefehle

Alle Sprungbefehle, außer dem GO Befehl, veranlassen einen relativen Übersprung im Programmcode – relativ im Sinne, dass die Parameter der Übersprungbefehle einen ganzzahligen Versatz zur aktuellen Programmadresse angeben. Dabei bedeutet der Versatz, wieviele Instruktionen müssen bis zur Zielinstruktion übersprungen werden. Z.B. die Instruktion

```
JMP 2
```

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

```
BE 5
```

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

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

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

3.7.1 BE

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

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

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

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

Bemerkung Die UMach Maschine erhöht den Programmcounter (Register PC) nach der Ausführung jeder Instruktion (siehe 2.1.2, Seite 10). Die Modifizierung des PC-Registers durch den BE Befehl wirkt sich nicht störend auf die automatische Inkrementierung des Programmcounters.

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

```
SET   R1 100     # R1 ← 100
LBUI  R2 R1  0   # Lade Byte von Adresse 100 nach R2
LBUI  R3 R1  1   # Lade Byte von Adresse 100+1 nach R3
CMPU  R2 R3      # Vergleiche Inhalt von R2 und R3
                 # Ergebnis geht ins CMP
#BE    equal     # Asm Schreibweise
BE    3          # Wenn CMP gleich 0 ist, überspringe
                 # die folgenden 3 Instruktionen
                 # (gehe zum label equal)
                 # N ist in diesem Fall 3
```

```
  ADDU  R2 R2 R3  # Addiere Inhalt von R2 und R3
  SBUI  R2 R1 0   # Speichere R2 nach Adresse 100
  #JMP   finish   # Asm Schreibweise
  JMP   2         # Überspringe die folgenden 2 Befehle,
                  # N von JMP ist 2.
equal:
  MULIU R2 2      # Multipliziere Inhalt von R2 mit 2
  SBUI  LO R1 0   # Speichere Inhalt von LO nach Adresse in R1
finish:
```

3.7.2 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 Übersprung ausgeführt wird, wenn CMP nicht 0 ist.

3.7.3 BL

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

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

3.7.4 BLE

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

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

3.7.5 BG

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

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

3.7.6 BGE

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

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

3.7.7 JMP

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

„Jump“. Überspringt N Instruktionen.

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.

3.8.2 `CALL`

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

3.8.3 `RET`

Assemblername	Parameter	Maschinencode	Format
`RET`	keine	`0x92`	000

3.9 Systeminstruktionen

3.9.1 `INT`

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

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

3.9.2 `IRET`

Assemblername	Parameter	Maschinencode	Format
`IRET`	$N \in \mathbb{N}$	`0xA1`	NNN

„Interrupt Return“. Rücksprung aus einer Hardware-Unterbrechung. Setzt HIR zusätzlich auf Null und Setzt den Status so, dass Hardware-Unterbrechungen wieder erlaubt sind.

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 Register und es ist hier deren Inhalt gemeint.

$$P \rightarrow mem_N(A)$$

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.

3.10.2 `OUT`

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

Schreibt N Bytes Daten aus dem Speicher mit 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.

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

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)
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 56 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		
	MUL	MULU	MULI	DIV	DIVU	DIVI		
4	ABS	NEG	INC	DEC				
	MOD	MODI						
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	IRET						
B	IN							
	OUT							
C								
D								
E								
F								
	8	9	A	B	C	D	E	F

4 Debugging

Die UMach Maschine stellt eine Debugging-Schnittstelle zur Verfügung. Um diese Schnittstelle zu aktivieren, muss man die Maschine im Einzelschrittmodus starten.

Glossar

A | B | D | I | R

A

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.

Byte
: Eine Reihe oder Gruppe von 8 Bit.

D

DMA
: Direct Memory Access

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 3.1.1.

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

R

Register
Eine sich im Prozessor befindende Speichereinheit. Das Register ist dem Programmierer sichbar und kann mit Werten geladen werden. Siehe Abschnitt 2.3, Seite 12.

Registernummer
Eine eindeutige Zahl, die ein Register identifiziert. Eine Instruktion verwendet diese Zahl, wenn sie ein Register angibt.

Index