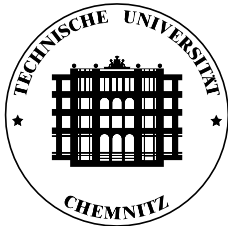


2. Übung
Rechnerorganisation/Rechnerarchitektur

LC-2: Einführung und Programmierung in Maschinensprache

Lehrveranstaltung: Prof. Dr. W. Rehm
Bearbeiter: Friedrich Seifert, 13.11.2001

1 Ziel

Anhand des LC–2 Simulators (Little Computer 2) sollen grundlegende Kenntnisse über die maschinennahe Programmierung von Rechnersystemen erlernt werden. Gegenstand des vorliegenden Versuchs ist die Architektur des LC–2 und die Programmierung in Maschinensprache.

2 Literatur

[1] *Folien zur Rechnerorganisation/Rechnerarchitektur–Vorlesung*
`https://www.tu-chemnitz.de/informatik/RA/educ/roa/download.html`

[2] Yale N. Patt and Sanjay J. Patel: *Introduction to Computing Systems: from bits and gates to C and beyond*, McGraw-Hill, 2001.
`http://www.crhc.uiuc.edu/patt_patel/`

[3] Matt Postiff: *LC–2 Programmer's Reference and User Guide*, University of Michigan EECS 100.
`http://www.eecs.umich.edu/~postiffm/lc2/lc2.pdf`

[4] Kathy Buchheit: *Guide to Using the Unix version of the LC–2 Simulator* und *Guide to Using the Windows version of the LC–2 Simulator and LC2Edit*, The University of Texas at Austin.
`http://www.crhc.uiuc.edu/patt_patel/manual/`

[5] Wolfgang Rehm: *Rechnerarchitektur, In: Informatik für Ingenieure kompakt*,
Vieweg Verlag, 2001, ISBN 3-528-03918-3.

3 Der LC–2 Computer

Vorbemerkung

Wir verwenden folgende Konventionen bei der Notation:
Ein Wort ist eine 16–Bit Zahl, wobei die Nummer der Bitstelle von rechts nach links ansteigt. Das ganz linke Bit ist Bit 15, das ganz rechte Bit 0. Bit 15 ist das *höchstwertige* Bit, wohingegen Bit 0 das *niederwertigste* Bit darstellt:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

Notation	Bedeutung
0xZahl, xZahl, $Zahl	Die Zahl ist hexadezimal angegeben.
#Zahl	Die Zahl ist dezimal angegeben.
A<l:r>	Bezeichnet einen Teil des angegebenen Wertes, und zwar genau die Bits von Position l bis Position r. Wenn z.B. das PC Register den Wert (hex) 0x301A = (binär) 0011 0000 0001 1010 enthält, dann bezeichnet PC<15:9> die 7 Bits 0011 000.
DR	Destination Register: Zielregister einer Operation (R0...R7)
SR, SR1, SR2	Source Register: Quellregister einer Operation (R0...R7)
MBZ	Must Be Zero – Die entsprechenden Bits müssen null sein
MB1	Must Be 1 – Die entsprechenden Bits müssen eins sein
mem[Adresse]	Bezeichnet den Inhalt des Speichers an der angegebenen Adresse

3.1 Einführung

Der Little Computer 2 (LC–2) wurde an der Universität von Michigan (USA) für die Ausbildung entwickelt, um Studenten ohne weitere Vorkenntnisse in Rechnerarchitektur und Logikentwurf mit Aufbau und Funktion von Universalrechnern vertraut zu machen. Die Funktionsweise des LC–2 ist durch seine Architektur und seinen Befehlsatz definiert. Obwohl es einige Realisierungen in Hardware gibt, benutzen wir einen Simulator, der unter Unix und Windows läuft und die volle Funktionalität des LC–2 nachbildet.

3.2 Die Architektur

Der LC–2 ist im Gegensatz zu realen Prozessoren sehr einfach gestaltet und damit gut überschaubar. Es handelt sich dabei um einen 16–Bit Prozessor. Er besitzt 8 Allzweckregister von je 16 Bit Breite. Ein besonderes Register ist der Program Counter PC. Er enthält die Speicheradresse des nächsten Befehls, d.h. Programme werden im Speicher abgelegt. Die Arithmetik–Logik–Einheit (ALU) verarbeitet ebenfalls 16–Bit Worte.
Der LC–2 verfügt neben den Registern R0 bis R7 über drei Flipflops, die in Abhängigkeit vom Ausgang eines Befehls bestimmte Bedingungscodes (condition codes) enthalten. Diese werden auch als *Flags* bezeichnet. Es ist zu jedem Zeitpunkt immer genau ein Flag aktiv, d.h. gleich eins:

- **N** Ergebnis der letzten Operation war negativ.
- **Z** Ergebnis der letzten Operation war null.
- **P** Ergebnis der letzten Operation war positiv.

Die Adressbreite beträgt 16 Bit, so dass die Maschine über 2^{16} (65536 oder 64K) Worte bzw. 128 KB Speicher verfügt. Man kann den Speicher des LC–2 als in $2^7 = 128$ Seiten zu je $2^9 = 512$ Worten unterteilt betrachten. Die oberen 7 Bit der PC Registers bestimmen dabei die Seite. Ein– und Ausgabegeräte erscheinen im Speicher an festgelegten Adressen. Tabelle 1 zeigt die Speicherbelegung des LC–2. Der Bereich von 0x3000 bis 0xCFFF steht für Programme zur Verfügung. Wie wir später sehen werden, beginnen unsere Programme in der Regel bei Adresse 0x3000.

Adressbereich	Verwendung
0x0000-0x0FFF	Vektortabelle/Betriebssystem
0x1000-0x1FFF	Reserviert
0x2000-0x2FFF	Reserviert
0x3000-0x3FFF	Verfügbar
0x4000-0x4FFF	Verfügbar
0x5000-0x5FFF	Verfügbar
0x6000-0x6FFF	Verfügbar
0x7000-0x7FFF	Verfügbar
0x8000-0x8FFF	Verfügbar
0x9000-0x9FFF	Verfügbar
0xA000-0xAFFF	Verfügbar
0xB000-0xBFFF	Verfügbar
0xC000-0xCFFF	Verfügbar
0xD000-0xDFFF	Reserviert
0xE000-0xEFFF	Reserviert
0xF000-0xFFFF	Video RAM, Tastatur RAM, System RAM, Boot ROM, Ein/Ausgabe

Tabelle 1: Speicherbelegung des LC–2

3.3 Der Befehlssatz

Der Befehlssatz des LC–2 umfasst 16 grundlegende Befehle. Sie lassen sich folgendermaßen gruppieren:

Arithmetisch/Logische Operationen: ADD, AND und NOT

Datentransferbefehle: LD, LDI, LDR, ST, STI, STR, LEA

Ablaufsteuerung: BR, JSR, JMP, JSRR, RET

Steuerbefehle: RTI, TRAP

Im weiteren Verlauf dieses Abschnitts wird näher auf die einzelnen Befehle eingegangen.

3.3.1 Adressierungsarten

Zunächst sollen jedoch die fünf verschiedenen Adressierungsarten vorgestellt werden. Diese sind in Tabelle 2 zusammengefasst. Abbildung 1 illustriert die Adressierungsarten anhand der gegebenen Beispiele.

Name der Adressieringsart	Bedeutung	Beispiel
Register	Das Datum befindet sich im angegebenen Register	add r1, r2, r3
Immediate (unmittelbar)	Das Datum ist Teil des Befehls selbst. Im Beispiel ist #5 das “immediate” Datum.	add r1, r2, #5 lea r1, LABEL
Direkt	Das Datum befindet sich an der angegebenen Adresse auf der aktuellen Seite.	ld r1, LABEL
Indirekt	Die angegebene Adresse auf der aktuellen Seite enthält die Adresse des Datums.	ldi r1, LABEL
Basis+Index	Das Datum befindet sich an der Adresse, die sich aus der Summe des Basisregisterinhalts und des null–erweiterten[a] Indexes ergibt.	ldr r1, r2, #3

Tabelle 2: Adressierungsarten

[a]Die oberen Bitstellen (hier 15 bis 6) werden mit Nullen aufgefüllt

3.3.2 Sprungbefehlsarten

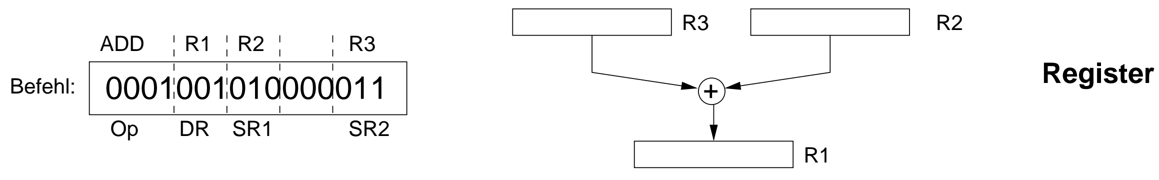
Als nächstes betrachten wir die verschiedenen Sprung- und Verzweigungsbefehle. Verzweigungen (branches) sind eine besondere Form der Sprungbefehle, deren tatsächliche Ausführung von bestimmten Bedingungen abhängt. Unbedingte Sprünge werden in der Regel nur “Sprünge” (jumps) genannt.
Es gibt zwei grundlegende Arten zur Bestimmung der Zieladresse:

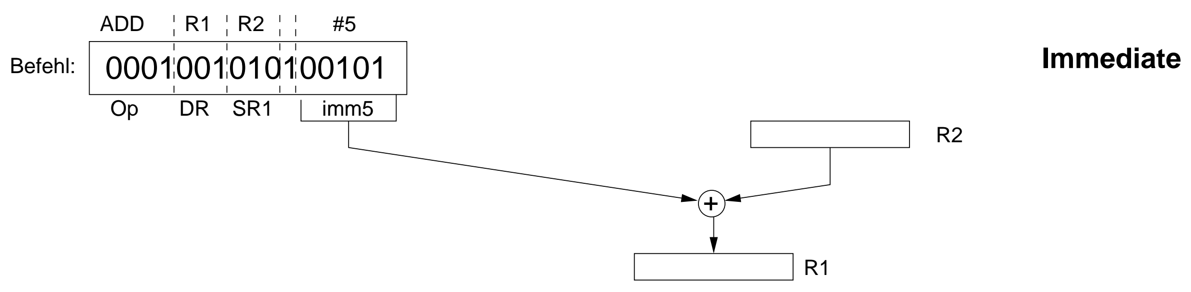
Beispiel	Bedeutung
BR LABEL JSR LABEL	Das Ziel der Verzweigung oder des Sprungs ist die abgegebene Adresse auf der aktuellen Seite.
JSRR BaseR, index6	Das Sprungziel ergibt sich aus der Summe des Inhalts von Register BaseR und dem null-erweiterten 6–Bit Index aus dem Befehlswort.

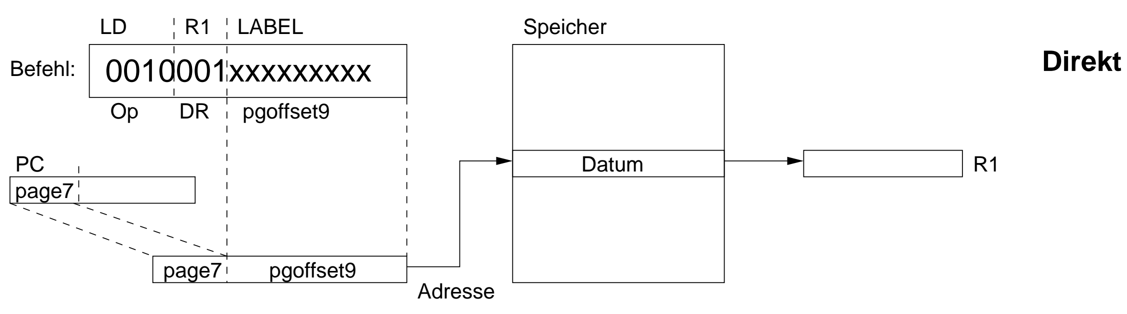
Abbildung 2 illustriert die Funktionsweise.

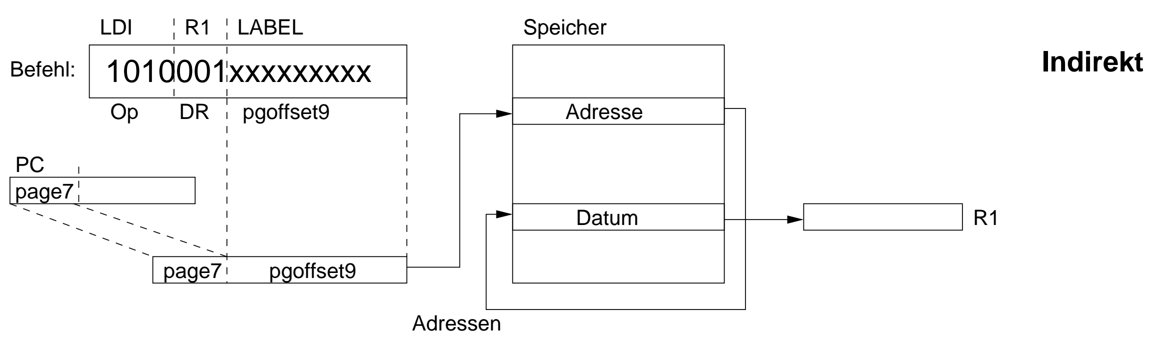
3.3.3 Systemrufe

Der LC–2 verfügt über eine Reihe von Betriebssystemrufen, die den Programmierer bei der Ein/Ausgabe unterstützen und die Maschine am Programmende anhalten. Die Funktionen sind mittels sog. Traps realisiert. Tabelle 3 beschreibt alle vorhandenen Funktionen.









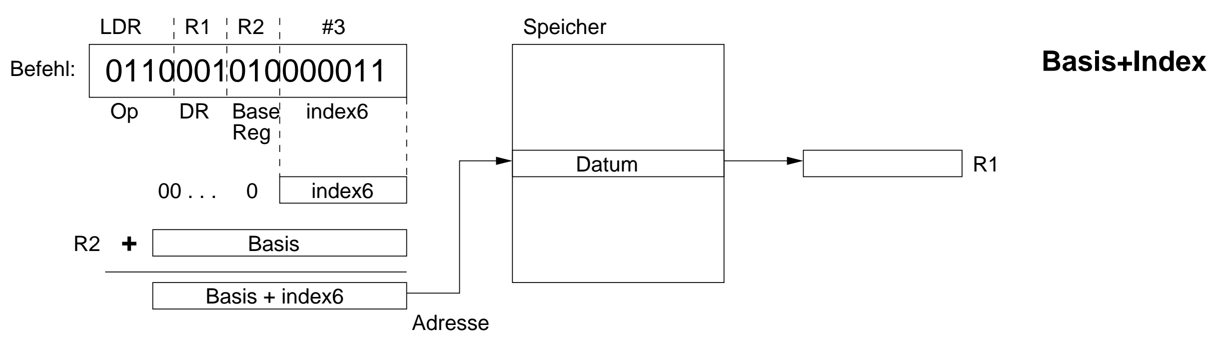


Abbildung 1: Illustration der Adressierungsarten des LC–2

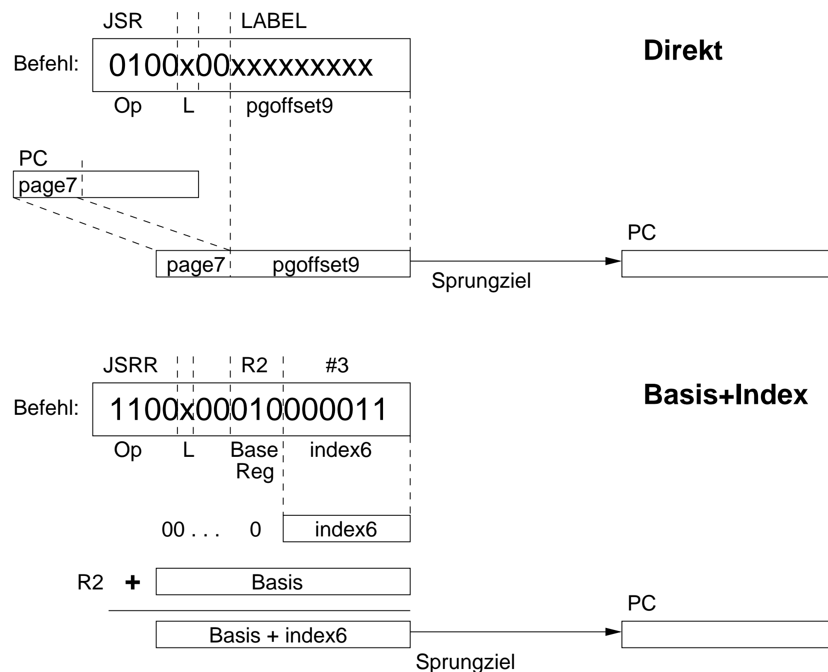


Abbildung 2: Illustration der Sprungarten des LC–2

Trap Nr.	Assembler Name	Beschreibung
$20	GETC	Liest ein einzelnes Zeichen von der Tastatur. Das Zeichen wird nicht auf der Konsole dargestellt. Der ASCII Code des Zeichens wird nach R0 kopiert, wobei die oberen 8 Bit auf null gesetzt werden.
$21	OUT	Schreibt ein Zeichen aus R0<7:0> auf die Konsole.
$22	PUTS	Schreibt die Zeichenkette, auf die R0 zeigt, auf die Konsole.
$23	IN	Gibt eine Eingabeaufforderung auf dem Bildschirm aus und liest ein einzelnes Zeichen von der Tastatur, wobei das Zeichen angezeigt wird. Sein ASCII Code wird nach R0 geschrieben. Die oberen 8 Bit von R0 werden auf null gesetzt.
$24	PUTSP	Schreibt eine gepackte Zeichenkette, auf die R0 zeigt, auf die Konsole.
$25	HALT	Schreibt eine Meldung auf die Konsole und hält die Befehlsausführung an.
Andere		Alle anderen Trapvektoren zeigen auf eine Routine, die eine Meldung auf der Konsole ausgibt, dass ein Trap ausgeführt wurde, für den keine Funktion definiert ist.

Tabelle 3: Systemrufe des LC–2

3.3.4 Die wichtigsten Befehle im Einzelnen

ADD — Addition

Mögliche Formen:

ADD DR, SR1, SR2 (Register)
ADD DR, SR1, imm5 (Immediate)

Maschinencode:

<table>
<tr><td>15</td><td>14</td><td>13</td><td>12</td><td>11</td><td>10</td><td>9</td><td>8</td><td>7</td><td>6</td><td>5</td><td>4</td><td>3</td><td>2</td><td>1</td><td>0</td></tr>
<tr><td colspan="4">0001</td><td colspan="3">DR</td><td colspan="3">SR1</td><td>0</td><td colspan="2">MBZ</td><td colspan="3">SR2</td></tr>
<tr><td colspan="4">0001</td><td colspan="3">DR</td><td colspan="3">SR1</td><td>1</td><td colspan="5">imm5</td></tr>
</table>

Beschreibung: Wenn Bit 5 gleich 0 ist, werden die Inhalte der Quellregister SR1 und SR2 addiert. Wenn Bit 5 gleich 1 ist, werden der Inhalt von SR1 und der vorzeichenerweiterte[1] immediate–Wert imm5 addiert. Im beiden Fällen wird das Ergebnis in DR abgelegt und die Bedingungscodes entsprechend dem Ergebnis gesetzt.

Beispiele: ADD R2, R3, R4 ; R2 = R3 + R4
ADD R2, R3, #7 ; R2 = R3 + 7

AND — Bitweises logisches UND

Mögliche Formen: AND DR, SR1, SR2 (Register)
AND DR, SR1, imm5 (Immediate)

Maschinencode:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0101				DR			SR1			0	MBZ		SR2		
0101				DR			SR1			1	imm5				

Beschreibung: Wenn Bit 5 gleich 0 ist, werden die Inhalte der Quellregister SR1 und SR2 bitweise UND–verknüpft. Wenn Bit 5 gleich 1 ist, werden der Inhalt von SR1 und der vorzeichenerweiterte immediate–Wert imm5 UND–verknüpft. Im beiden Fällen wird das Ergebnis in DR abgelegt und die Bedingungscodes entsprechend dem Ergebnis gesetzt.

Beispiele: AND R2, R3, R4 ; R2 = R3 UND R4
AND R2, R3, #7 ; R2 = R3 UND 7

NOT — Bitweise Negation (Invertieren, Einerkomplement)

Form: NOT DR, SR

Maschinencode:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1001				DR			SR			MB1					

Beschreibung: Führt eine bitweise Negation des Inhalts von SR aus und schreibt das Ergebnis nach DR. Die Bedingungscodes werden entsprechend gesetzt.

Beispiel: NOT R2, R3 ; R2 = NICHT(R3)

LD — Laden direkt vom Speicher in Register

Form: LD DR, LABEL

Maschinencode:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0010				DR			pgoffset9								

Beschreibung: Lädt ein Register mit dem Inhalt des Speichers an der angegebenen Position auf der aktuellen Seite. Die oberen 7 Bits des Programmzählers (PC) ergeben zusammen mit dem pgoffset9–Feld eine 16–Bit Speicheradresse. Der Inhalt dieser Speicherzelle wird nach DR geschrieben. Dabei werden die Bedingungscodes entsprechend dem geladenen Wert gesetzt. Siehe auch Abbildung 1.

Beispiel: LD R2, ZAEHLER ; R2 = mem[ZAEHLER]

LDI — Laden indirekt vom Speicher in Register

Form: LDI DR, LABEL

Maschinencode:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1010				DR			pgoffset9								

[1]Der Wert wird als Zweierkomplement interpretiert und in der Weise auf 16 Bit erweitert, dass das höchste Bit (Vorzeichen) in die zu erweiternden Bitstellen übertragen wird.

Beschreibung: Lädt ein Register indirekt von der angegebenen Position. Die oberen 7 Bits des Programmzählers (PC) ergeben zusammen mit dem pgoffset9–Feld eine 16–Bit Speicheradresse. Der Inhalt dieser Speicherzelle wird wiederum als Adresse benutzt. Der Speicherinhalt an dieser Stelle wird nach DR geschrieben. Dabei werden die Bedingungscodes entsprechend dem geladenen Wert gesetzt. Siehe auch Abbildung 1.

Beispiel: LDI R2, ZEIGER ; R2 = mem[mem[ZEIGER]]

LDR — Laden von mem[Basis + Index] in Register

Form: LDR DR, BaseR, index6

Maschinencode:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0110				DR			BaseR			index6					

Beschreibung: Lädt ein Register mittels Basisregister und Index. Der 6–Bit Index (index6) wird null–erweitert und zum Inhalt von BaseR addiert. Der resultierende Wert dient als Speicheradresse. Der Speicherinhalt an dieser Stelle wird nach DR geladen. Zu beachten ist, dass index6 immer einen positiven Offset darstellt. Die Bedingungscodes werden entsprechend gesetzt. Siehe auch Abbildung 1.

Beispiel: LDR R2, R4, #10 ; R2 = mem[R4 + 10]

LEA — Laden der effektiven Adresse

Form: LEA DR, LABEL

Maschinencode:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1110				DR			pgoffset9								

Beschreibung: Die oberen 7 Bits des Programmzählers (PC) ergeben zusammen mit dem pgoffset9–Feld eine 16–Bit Speicheradresse, die in das Register DR geladen wird. Dabei werden die Bedingungscodes entsprechend dem geladenen Wert gesetzt. Das Register, das die Adresse enthält, kann nun z.B. für LDR Befehle verwendet werden.

Beispiel: LEA R2, FOO ; R2 = Adresse von FOO

ST — Speichern direkt von Register in Speicher

Form: ST SR, LABEL

Maschinencode:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0011				SR			pgoffset9								

Beschreibung: Schreibt den Inhalt eines Registers an die angegebene Speicherstelle auf der aktuellen Seite. Die oberen 7 Bits des Programmzählers (PC) ergeben zusammen mit dem pgoffset9–Feld eine 16–Bit Speicheradresse. Der Inhalt von SR wird in diese Speicherzelle kopiert. Dabei werden die Bedingungscodes nicht verändert.

Beispiel: ST R2, ZAEHLER ; mem[ZAEHLER] = R2

STI — Speichern indirekt von Register in Speicher

Form: STI SR, LABEL

Maschinencode:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1011				SR			pgoffset9								

Beschreibung: Speichert den Inhalt eines Registers indirekt an der angegebenen Position im Speicher. Die oberen 7 Bits des Programmzählers (PC) ergeben zusammen mit dem pgoffset9–Feld eine 16–Bit Speicheradresse. Der Inhalt dieser Speicherzelle wird wiederum als Adresse benutzt. Der Inhalt von SR wird an diese Stelle im Speicher kopiert. Dabei werden die Bedingungscodes nicht verändert.

Beispiel: STI R2, ZEIGER ; mem[mem[ZEIGER]] = R2

STR — Speichern von Register nach mem[Basis + Index]

Form: STR SR, BaseR, index6

Maschinencode:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0111				SR			BaseR			index6					

Beschreibung: Speichert den Inhalt eines Registers mittels Basisregister und Index. Der 6–Bit Index (index6) wird null–erweitert und zum Inhalt von BaseR addiert. Der resultierende Wert dient als Speicheradresse. Der Inhalt von SR wird an diese Stelle geschrieben. Zu beachten ist, dass index6 immer einen positiven Offset darstellt. Die Bedingungscodes werden nicht verändert.

Beispiel: STR R4, R2, #10 ; mem[R2 + 10] = R4

BR — Verzweigung zur angegebenen Stelle auf der aktuellen Seite

Mögliche Formen:

	Äquivalente Form
BRn LABEL	BRlt LABEL
BRz LABEL	BReq LABEL
BRp LABEL	BRgt LABEL
BRnz LABEL	BRle LABEL
BRnp LABEL	BRne LABEL
BRzp LABEL	BRge LABEL
BRnzp LABEL	BR LABEL
BRNOP	NOP

Maschinencode:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0000				n	z	p	pgoffset9								

Beschreibung: Verzweigt zur angegebenen Stelle auf der aktuellen Seite, wenn die Bedingung erfüllt ist. Die oberen 7 Bits des Programmzählers (PC) ergeben zusammen mit dem pgoffset9–Feld den neuen 16–Bit PC Wert. Dieser wird nur dann nach PC geschrieben, wenn die Bedingung wahr ist. Die Bedingungscodes werden dabei nicht verändert. Die nzp Bits im Befehlswort spezifizieren, welche Bedingungscodes ausgewertet werden sollen. Die folgende Tabelle erläutert den Zusammenhang zwischen den nzp Bits des Befehls und den Bedingungscodes, die gesetzt sein müssen, um diese Bedingung zu erfüllen.

nzp	gesetztze Bedingungs-codes	Bedingung	nzp	gesetztze Bedingungs-codes	Bedingung
000	—	Verzweige unter keiner Bedingung	100	N	Negativ Kleiner als
001	P	Positiv Größer als	101	N oder P	Negativ oder Positiv Ungleich
010	Z	Null Gleich	110	N oder Z	Negativ oder null Kleiner als oder gleich
011	Z oder P	Null oder positiv Größer als oder gleich	111	N, Z oder P	Negativ, null oder positiv Unbedingt

Beispiel: BRzp SCHLEIFE ; Verzweigt zur Marke SCHLEIFE, wenn das letzte Ergebnis null oder positiv war

JSR/JMP — Sprung zu Unterprogramm/Sprung

Mögliche Formen: JSR LABEL (L=1)
JMP LABEL (L=0)

Maschinencode:

15–12	11	10–9	8–0
0100	L	MBZ	pgoffset9

Beschreibung: Springt unbedingt an die angegebene Stelle auf der aktuellen Seite. Die oberen 7 Bits des Programmzählers (PC) ergeben zusammen mit dem pgoffset9–Feld den neuen 16–Bit PC Wert. Die Bedingungscodes werden dabei nicht verändert. Wenn das Link–Bit L gesetzt ist, wird der Inhalt des Programmzählers PC in das Register R7 kopiert, um eine spätere Rückkehr zu dieser Stelle zu ermöglichen. Siehe auch Abbildung 2.

Beispiele:

```
JSR  FOO   ; Springt zur Marke FOO, Rücksprungadresse wird in R7 gespeichert
JMP  BAR   ; Springt zur Marke BAR
```

JSRR/JMPR — Sprung zu Unterprogramm über Register/Sprung über Register

Mögliche Formen:

```
JSRR  BaseR, index6   (L=1)
JMPR  BaseR, index6   (L=0)
```

Maschinencode:

15–12	11	10–9	8–6	5–0
1100	L	MBZ	BaseR	index6

Beschreibung: Der 6–Bit Index index6 wird null–erweitert und zum Inhalt von BaseR addiert, um die Sprungzieladresse zu erhalten. Wenn das Link–Bit L gesetzt ist, wird vor dem Sprung PC nach R7 kopiert. Die Bedingungscodes werden dabei nicht verändert. Siehe auch Abbildung 2.

Beispiele:

```
JSRR  R3, #11   ; Springt zu R3 + 11, Rücksprungadresse wird in R7 gespeichert
JMPR  R3, #11   ; Springt zu R3 + 11
```

RET — Rückkehr von Unterprogramm

Form: RET

Maschinencode:

15–12	11–0
1101	MBZ

Beschreibung: Lädt den PC mit dem Wert aus R7 und erlaubt damit die Rückkehr von einem vorangegangenen JSR oder JSRR Befehl. Die Bedingungscodes werden dabei nicht verändert.

Beispiele:

```
RET   ; PC = R7
```

3.3.5 Befehlsübersicht

<table>
<tr><th>Mnemonic</th><th>15</th><th>14</th><th>13</th><th>12</th><th>11</th><th>10</th><th>9</th><th>8</th><th>7</th><th>6</th><th>5</th><th>4</th><th>3</th><th>2</th><th>1</th><th>0</th></tr>
<tr><td>*ADD</td><td colspan="4">0001</td><td colspan="3">DR</td><td colspan="3">SR1</td><td>0</td><td colspan="2">MBZ</td><td colspan="3">SR2</td></tr>
<tr><td>*ADD</td><td colspan="4">0001</td><td colspan="3">DR</td><td colspan="3">SR1</td><td>1</td><td colspan="5">imm5</td></tr>
<tr><td>*AND</td><td colspan="4">0101</td><td colspan="3">DR</td><td colspan="3">SR1</td><td>0</td><td colspan="2">MBZ</td><td colspan="3">SR2</td></tr>
<tr><td>*AND</td><td colspan="4">0101</td><td colspan="3">DR</td><td colspan="3">SR1</td><td>1</td><td colspan="5">imm5</td></tr>
<tr><td>BR</td><td colspan="4">0000</td><td>n</td><td>z</td><td>p</td><td colspan="9">pgoffset9</td></tr>
<tr><td>JSR</td><td colspan="4">0100</td><td>L</td><td colspan="2">MBZ</td><td colspan="9">pgoffset9</td></tr>
<tr><td>JSRR</td><td colspan="4">1100</td><td>L</td><td colspan="2">MBZ</td><td colspan="3">BaseR</td><td colspan="6">index6</td></tr>
<tr><td>*LD</td><td colspan="4">0010</td><td colspan="3">DR</td><td colspan="9">pgoffset9</td></tr>
<tr><td>*LDI</td><td colspan="4">1010</td><td colspan="3">DR</td><td colspan="9">pgoffset9</td></tr>
<tr><td>*LDR</td><td colspan="4">0110</td><td colspan="3">DR</td><td colspan="3">BaseR</td><td colspan="6">index6</td></tr>
<tr><td>*LEA</td><td colspan="4">1110</td><td colspan="3">DR</td><td colspan="9">pgoffset9</td></tr>
<tr><td>*NOT</td><td colspan="4">1001</td><td colspan="3">DR</td><td colspan="3">SR</td><td colspan="6">MB1</td></tr>
<tr><td>RET</td><td colspan="4">1101</td><td colspan="12">MBZ</td></tr>
<tr><td>RTI</td><td colspan="4">1000</td><td colspan="12">MBZ</td></tr>
<tr><td>ST</td><td colspan="4">0011</td><td colspan="3">SR</td><td colspan="9">pgoffset9</td></tr>
<tr><td>STI</td><td colspan="4">1011</td><td colspan="3">SR</td><td colspan="9">pgoffset9</td></tr>
<tr><td>STR</td><td colspan="4">0111</td><td colspan="3">SR</td><td colspan="3">BaseR</td><td colspan="6">index6</td></tr>
<tr><td>TRAP</td><td colspan="4">1111</td><td colspan="4">MBZ</td><td colspan="8">trapvect8</td></tr>
</table>

* bedeutet, dass der Befehl die Flags modifiziert

4 Bedienung des LC–2 Simulators

Wir behandeln hier nur die Bedienung der Windows–Version des Simulators. Die UNIX–Variante ist ähnlich zu bedienen, aber nicht ganz so komfortabel. [4] gibt eine ausführliche Erläuterung beider Versionen.
Abbildung 3 zeigt das Hauptfenster des LC–2 Simulators unter Windows.
Im oberen Bereich wird der Inhalt der Register R0 bis R7 hexadezimal und dezimal dargestellt. Rechts daneben befinden sich der Programmzähler PC, das Befehlsregister IR, das das aktuelle Befehlswort enthält, und die Bedingungscodes CC (Flags). Der größte Teil des Fensters dient der zeilenweisen Anzeige des Speicherinhalts nach folgendem Schema:

	Adresse (hex)	Inhalt (binär)	Inhalt (hex)	Marke	Assemblermnemonic (bei Befehlen)
Beispiel:	0x3001	0001001000100000	0x1220	LOOP	ADD R1, R0, #0000

Ein Pfeil vor der Adresse macht die Stellung des PC deutlich.

4.1 Erzeugen und Laden einer Objektdatei

Damit der LC–2 Simulator ein Programm abarbeiten kann, muss es zunächst in der Speicher gebracht werden. Eine Möglichkeit bestünde darin, den Maschinencode Wort für Wort mit Hilfe des “Simulate → Set Value” (F4) Dialogs einzugeben. Praktischer ist es, das Programm vorher in einer sog. Objektdatei abzulegen. Diese kann dann über “File → Load Program” in den Speicher geladen werden.
Objektdateien können nicht direkt mit einem gewöhnlichen Editor erstellt oder angezeigt werden, da sie Binärdaten enthalten. Stattdessen verwenden wir ein Programm namens LC2Edit, das es erlaubt, ein Programm in Textform einzugeben und es in eine für den LC–2 lesbare Binärdatei umzuwandeln. Die Eingabe kann auf drei Arten erfolgen:

Binär Die Daten werden zeilenweise in Binärdarstellung (16 Bit) angebenen. Eine Zeile entspricht genau einem Speicherwort. Beispiel: 0001010010100001, zur besseren Übersichtlichkeit können Leerzeichen zwischen Bitgruppen gelassen werden: 0001 0100 1010 0001.

Hexadezimal Die Daten werden zeilenweise in Hexadezimalschreibweise angegeben. Beispiel: 14A1.

Assembler Die Befehle werden in einer symbolischen Schreibeweise angegeben. Dies wird jedoch erst Gegenstand der nächsten Übung sein.

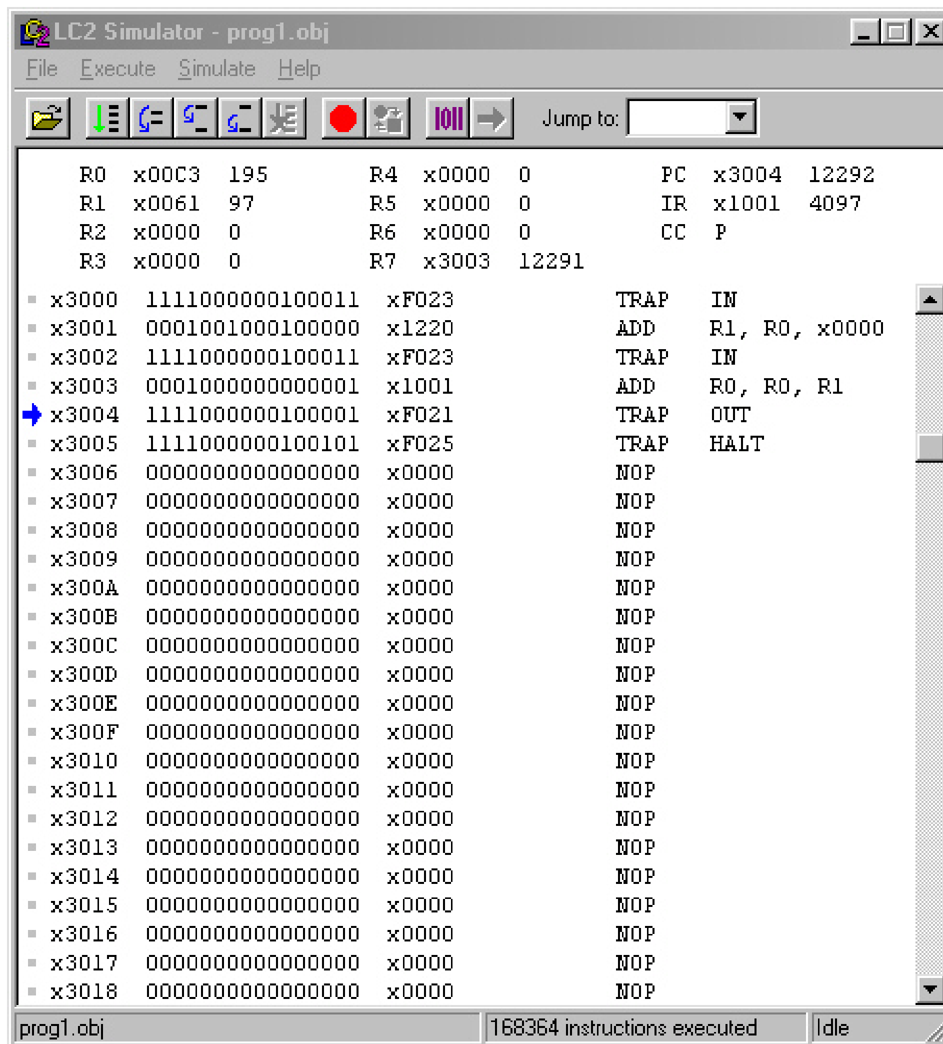


Abbildung 3: Screenshot des LC–2 Simulator für Windows

Mit Hilfe der Funktionen “Translate → Convert Base 2/Convert Base 16/Assemble” kann die Erzeugung der Objektdatei angestoßen werden. Jede Zeile kann einen Kommentar zur Erläuterung enthalten. Dazu wird alles, was nach einem Semikolon steht, bei der Umwandlung ignoriert.
Mit Objektdateien können sowohl Programme als auch Daten in der Speicher geladen werden. Die erste Zeile (erstes Wort) bestimmt dabei immer die Adresse, an welche die Daten geschrieben werden sollen. Programme beginnen in der Regel bei Adresse 0x3000, so dass es in Hexadezimaldarstellung immer mit 3000 beginnt. Beispiel:

```
3000
1220
F023
...
```

4.2 Ausführen eines Programms

Nachdem das Programm erstellt und geladen wurde, kann die Ausführung beginnen. Die Funktion “Execute → Run” startet die Programmausführung von der aktuellen Position des PC aus. Nach der Initialisierung steht dieser auf 0x3000. Mit dem “Jump to” Feld kann die Speicheranzeige auf eine bestimmte Adresse gesetzt werden. Mit “Simulate → Set PC to selection location” lässt sich der Programmzähler auf die gerade ausgewählte Adresse setzen. Die Ausführung der Programms wird solange fortgesetzt, bis entweder der HALT Systemruf oder ein Unterbrechungspunkt (breakpoint) erreicht wird. Mit “Execute → Stop” kann der Prozessor ebenfalls angehalten werden. Ein– und Ausgaben erfolgen über das sog. Konsolenfenster.

4.3 Abarbeitung im Einzelschrittmodus

Sowohl zum debuggen (Fehler suchen und entfernen) von Programmen als auch zum Verstehen des LC–2 und seiner Maschinensprache ist es hilfreich, ein Programm Befehl für Befehl durchgehen zu können und dabei die Veränderungen in den Registern und im Speicher zu beobachten. Dies ist im *Einzelschrittbetrieb* (single step) möglich. Der Prozessor hält nach jedem Befehl an und setzt die Abarbeitung erst auf Kommando wieder fort. Dabei werden bei jedem Schritt die Inhalte der Register und des Sepichers aktualisiert.
Mit "Execute → Step Into" führt der LC–Simulator eine einzelne Instruktion aus. Im Fall eines Unterprogrammaufrufs (JSR, JSRR oder TRAP) wird in die Subroutine "hineingegangen", d.h. der nächste Schritt führt den ersten Befehl des Unterprogramms aus.
Mit "Execute → Step Over" kann die Ausführung eines Unterprogramms in einem Schritt erfolgen. Der Simulator hält vor der Ausführung des ersten Befehls nach dem Sprungbefehl an.
Mit "Execute → Step Out" kann man in einem Schritt an das Ende eines einmal betretenen Unterprogramms gelangen, ohne die restlichen Anweisungen schrittweise durchgehen zu müssen.

5 Programmierung des LC–2 in Maschinensprache

Nachdem wir die grundlegende Funktionsweise des LC–2 Computers und den Umgang mit dem Simulator kennen gelernt haben, beschäftigen wir uns nun mit der untersten Ebene, auf der ein Rechner programmiert werden kann: der Maschinensprache.
Die Maschinensprache besteht nur aus (Binär–)Zahlen und wird vom Prozessor direkt verstanden. Allerdings ist sie für den Programmierer schwierig zu lesen. Um ein Programm in Maschinensprache zu schreiben, muss man wissen, wie die Befehlsworte der einzelnen Befehle aussehen. Diese Informationen wurden in Abschnitt 3.3 gegeben. Besonders hilfreich ist die Befehlsübersicht von Seite 10.

Unser erstes LC–2 Programm Unser erstes Programm soll zwei Zahlen, die an den Adressen 0x3010 und 0x3011 im Speicher stehen addieren und das Ergebnis in R3 liefern. Da der LC–2 Additionsoperationen nur zwischen Registern erlaubt, müssen wir zunächst die Operanden in Register laden, sagen wir nach R1 und R2. Dazu bedienen wir uns des LD (Lade direkt) Befehls. Wie oben erläutert, hat er das folgende Format:

15–12	11–9	8–0
0010 (LD)	DR	pgoffset9

Im ersten Fall ist DR 001 (für R1), im zweiten 010 (R2). Die aktuelle Speicherseite ist 0x3000, so dass wir die Werte 0x10 bzw 0x11 für pgoffset9 verwenden müssen. Damit lauten die zwei Ladebefehle in Maschinensprache:

15–12	11–9	8–0
0010	001	000010000

und

15–12	11–9	8–0
0010	010	000010001

Nun wollen wir die Inhalte von R1 und R2 addieren. Dazu verwenden wird den Additionsbefehl mit folgendem allgemeinen Format:

15–12	11–9	8–6	5	4–3	2–0
0001 (ADD)	DR	SR1	0	MBZ	SR2

Mit SR1 = 001 (R1), SR2 = 010 (R2) und DR = 011 (R3) erhalten wir:

15–12	11–9	8–6	5	4–3	2–0
0001	011	001	0	00	010

Dies entspricht der Operation R3 = R1 + R2. Damit haben wir gleichzeitig die Bedingung erfüllt, dass das Ergebnis in R3 stehen soll. Schauen wir uns nun das vollständige Maschinenprogramm an (Binärformat):

```
0011 0000 0000 0000 ; Erste Zeile: Programm wird an Adresse 0x3000 geladen
0010 0010 0001 0000 ; Lade Inhalt von Adresse 0x3010 nach R1
0010 0100 0001 0001 ; Lade Inhalt von Adresse 0x3011 nach R2
0001 0110 0100 0010 ; R3 = R1 + R2
1111 0000 0010 0101 ; HALT Systemruf
```

Ein komplexeres Programm mit Schleife und Basisregisteradressierung Ein zweites Beispiel soll die Verwendung der Verzweigungsbefehle und der Adressierung über ein Basisregister verdeutlichen. Dazu wollen wir ein Programm schreiben, das die Länge einer Zeichenkette (String) im Speicher ermittelt. Eine Zeichenkette der Länge l besteht aus l Worten, deren Bits¡7:0¿ jeweils den ASCII Code des Zeichens enthalten. Die oberen Bits¡15:8¿ sind null. Das Ende der Kette wird durch den Wert 0x0000 gekennzeichnet. Der String "LC–2" sieht damit im Speicher so aus (hexadezimal):

```
004C
0043
002D
0032
0000
```

Die zu untersuchende Zeichenkette beginne bei Adresse 0x3010. R2 sei der Längenzähler. Wir gehen folgerdermaßen vor: mittels der Basisregisteradressierung wird der String Zeichen für Zeichen eingelesen und auf Null getestet. Solange das Ende nicht erreicht ist, wird der Zähler R2 um eins erhöht.
Zunächst wird R2 gelöscht, indem sein Inhalt mit 0x0000 UND–verknüpft und wieder nach R2 geschrieben wird:

15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0

0101 (AND)	DR	SR1	1	imm5
0101	010	010	1	00000

R1 soll immer die Adresse des zu betrachtenden Zeichens enthalten, es fungiert als *Zeiger*. Dazu muss es mit der Startadresse der Zeichenkette geladen werden, was wir mit dem LEA (Lade effektive Adresse) bewerkstelligen. Der Seitenoffset ist dabei 0x10.

15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0

1110 (LEA)	DR	pgoffset9
1110	001	000010000

Der dritte Befehl unseres Programms soll nun das erste Zeichen des Strings in Register R0 laden. Dazu benutzen wir den LDR Befehl. Die benötigte Basisadresse befindet sich bereits in R1. Wie wir später sehen werden, müssen wir uns die Adresse dieses Befehls merken. Sie ist 0x3002, da es der dritte Befehl ist und das Programm bei 0x3000 beginnt.

15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0

0110 (LDR)	DR	BaseR	index6
0110	000	010	000000

Da beim Laden die Bedingungscodes gesetzt wurden, können wir testen, ob wir das Ende des Strings (0x0000) erreicht haben. in diesem Fall ist das z–Flag gesetzt, und genau dann verzweigen wir mittels BRz an das Ende des Programms. Wie wir später sehen werden, liegt dies an Adresse 0x3007, d.h. der Offset ist 0x7:

15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0

0000 (BR)	n	z	p	pgoffset9
0000	0	1	0	000000111

Betrachten wir nun weiter den Fall, in dem das Ende des Strings noch nicht erreicht wurde. Dann erhöhen wir unseren Zeichenzähler R2 und das Basisregister R1, so dass es auf das nächste Zeichen zeigt:

15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0

0001 (ADD)	DR	SR1	1	imm5
0001	010	010	1	00001
0001	001	001	1	00001

Um mit dem nächsten Zeichen fortzufahren, springen wir (unbedingt) zu dem Befehl, der das aktuelle Zeichen nach R0 lädt, d.h. zu Adresse 0x3002:

15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0

0100 (JMP)	L	MBZ	pgoffset9
0100	0	00	00000010

Den letzten Befehl unseres Programms bildet der HALT Systemruf an Adresse 0x3007, der angesprungen wird, wenn das Ende der Zeichenkette erreicht ist.
Damit sieht unser Maschinenprogramm im Binär– und Hexadezimalformat so aus:

0011 0000 0000 0000	3000
0101 0100 1010 0000	54A0
1110 0010 0001 0000	E210
0110 0000 0100 0000	6040
0000 0100 0000 0111	0407
0001 0100 1010 0001	14A1
0001 0010 0110 0001	1261
0100 0000 0000 0010	4002
1111 0000 0010 0101	F025

6 Aufgabenstellung

1. Wir betrachten noch einmal das erste Beispielprogramm zur Addition zweier Zahlen im Speicher aus Abschnitt 5.
   a) Erzeugen Sie eine Objektdatei, die die zu addierenden Zahlen an die richtigen Adressen im Speicher lädt. Unmittelbar nach dem zweiten Operanden soll Platz für das Ergebnis gehalten werden. Geben Sie die Datei im Hexadezimalformat an.
   b) Modifizieren Sie das Additionsprogramm so, dass das Ergebnis der Addition am Ende an Adresse x3012 im Speicher steht.
2. Schreiben Sie ein Programm, das den Inhalt von Adresse 0x3011 vom Inhalt von Adresse 0x3010 subtrahiert und das Ergebnis nach Adresse 0x3012 schreibt.
3. Analysieren Sie folgendes Programm!

binär	*hexadezimal*
0011 0000 0000 0000	3000
0101 010 010 1 00000	54A0
0001 010 010 0 00 100	1484
0001 101 101 1 11111	1B7F
0000 001 000000001	0201
1111 0000 00100101	F025

R4 habe den Wert #3, R5 den Wert #4. (Im Simulator können Register mit Hilfe von “Simulate → Set Value” mit einem bestimmten Wert belegt werden.) Welchen Wert hat R2 nach Ablauf des Programms? Was bedeuten die einzelnen Befehle und welche Funktion hat das Programm insgesamt? Sie können dazu auch andere Ausgangszahlen probieren.