# Praktikum Rechnerstrukturen

# Bogen 3

Mikroprogammierung II – Assemblerebene

Department Informatik, AB TAMS
MIN Fakultät, Universität Hamburg
Vogt-Kölln-Str. 30
D22527 Hamburg

# 1 Mikroprogrammierung II

In den Versuchen dieses Bogens werden zunächst die Mikroprogramme für die noch fehlenden Befehle unseres Prozessors implementiert und dann die *Register-Transfer-Ebene* verlassen, um die darüberliegen *Assembler-Ebene* zu betrachten.

**Aufgabe 1.1: Bedingte Sprünge** Programmverzweigungen erfordern bedingte Sprungbefehle. Im D·CORE dienen dazu die beiden Befehle `BT` (*branch if true*) und `BF` (*branch if false*), wobei der Wert des `C`-Registers für die Bedingung ausgewertet wird. Dazu ist der Ausgang des `C`-Registers einfach an den Select-Eingang des 2:1-Multiplexers im Steuerwerk geschaltet, was im Mikroprogramm die Auswahl von `uROM.nextA` (C= **0**) oder `uROM.nextB` (C= **1**) als Folgeadresse erlaubt. Dies ist auch in Abbildung 1 aus Bogen 1 oder 2 skizziert, und Sie haben diese Umschaltung bereits in Aufgabe 5.1 im ersten Aufgabenblatt benutzt. Im Grunde müssen Sie also zur Implementation der Befehle `BF` und `BT` nur noch den 4:1-Multiplexer richtig ansteuern sowie `uROM.nextA` und `uROM.nextB` geeignet setzen.

Auf der Assembler-Ebene muss vor einem bedingten Sprung natürlich das `C`-Register z.B. durch einen Vergleichsbefehl entsprechend gesetzt werden. Da das `C`-Register anders als bei den meisten älteren Architekturen aber nicht von allen ALU-Befehlen beeinflusst wird, muss der Vergleichsbefehl nicht unbedingt direkt vor dem Sprungbefehl stehen.

Realisieren Sie die beiden `BT`- und `BF`-Befehle im Mikroprogramm:

| addr | nextA | nextB | µPCmux.s1 | µPCmux.s0 | RXBUF | AX=15 | IR | ADDRBUF | C | ALUBUF | REGS.nWE | PCBUF | PC | PCMUX.s1 | PCMUX.s0 | MRRBUF | MRR | MDR | MDRBUF | MAR | nWE | nOE | name |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | 00 | 00 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | |
| | 00 | 00 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | |



| addr | nextA | nextB | µPCmux.s1 | µPCmux.s0 | RXBUF | AX=15 | IR | ADDRBUF | C | ALUBUF | REGS.nWE | PCBUF | PC | PCMUX.s1 | PCMUX.s0 | MRRBUF | MRR | MDR | MDRBUF | MAR | nWE | nOE | name |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | 00 | 00 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | |
| | 00 | 00 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | |



Mit den eben implementierten `BT`- bzw. `BF`-Befehlen lassen sich jetzt auf dem D·CORE auch Schleifen mit einer Abbruchbedingung ausführen. Das folgende Programm demonstriert neben der Umsetzung einer `while`-Schleife auch noch die indizierte Adressierung für den Zugriff auf Arrays.

**Aufgabe 1.1: While-Schleife** Schreiben Sie ein Programm, um ein Array (Feld) mit *n* Elementen auf die Werte 0 .. n-1 vorzubesetzen. Das Feld soll ab der Adresse `base` im Speicher liegen. Hier ein C-Pseudocode für das Programm:

```
int length = 5;
int base[] = 0x8010;  // Startadresse

int i = 0;
do {
  base[i] = i;
  i++;
} while( i < length );
```

| Label | Adresse | Befehlscode | Mnemonic | Kommentar |
|---|---|---|---|---|
| test_array_init: | 0000 | 0x3480 | movi R0, 8 | R[0] = 8 |
| | 0002 | | | |
| | 0004 | | | |
| | 0006 | | | |
| | 0008 | | | |
| | 000A | | | |
| | 000C | | | |
| | 000E | | | |
| | 0010 | | | |
| | 0012 | | | |
| | 0014 | | | |
| | 0016 | | | |
| | 0018 | | | |
| | 001A | | | |
| | 001C | | | |
| | 001E | | | |

**Aufgabe 1.1: JSR-Befehl** Die Abkürzung `JSR` steht für *Jump to Subroutine*. Der eigentliche Sprung erfolgt genau wie beim `BR`-Befehl; allerdings wird der aktuelle Wert des `PC` vorher im Register `R15` abgespeichert. Für diesen ersten Schritt des `JSR`-Befehls ist ein wenig zusätzliche Logik im Prozessor erforderlich, da die Schreib-Adresse der Registerbank für alle anderen Befehle direkt aus dem Befehlsregister kommt, hier aber fest auf den Wert 15 gesetzt werden muss. Dies erledigt ein kleiner Block von OR-Gattern (Komponente `AX-or-15`), der zwischen Befehlsdekoder und die Schreibadresse `AZ` der Registerbank gesetzt ist, und über die Steuerleitung `ax=15` aus dem Mikroprogramm aktiviert wird. Da das Abspeichern des `PC` erfolgt, nachdem dieser in der Decode-Phase bereits um 2 inkrementiert wurde, zeigt Register `R15` nach einem `JSR` direkt auf den nach einem Rücksprung auszuführenden Befehl.

Erweitern Sie Ihr Mikroprogramm um den letzten noch fehlenden Befehl `JSR`:

| addr | nextA | nextB | µPCmux.s1 | µPCmux.s0 | RXBUF | AX=15 | IR | ADDRBUF | C | ALUBUF | REGS.nWE | PCBUF | PC | PCMUX.s1 | PCMUX.s0 | MRRBUF | MRR | MDR | MDRBUF | MAR | nWE | nOE | name |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | 00 | 00 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | |
| | 00 | 00 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | |

**Aufgabe 1.2: Return** Begründen Sie, warum die D·CORE-Architektur keinen expliziten Return-Befehl bereitstellt:

Laden Sie jetzt die vorgegebene Datei `bigtest.rom` in das ROM und starten Sie den Prozessor. Das Testprogramm überprüft alle bisher vorhandenen Befehle (ALU, Immediate, Compare, Load, Store, Jump, Branch, Jump to Subroutine, Halt). Wenn alles funktioniert, schreibt das Programm den Wert 0xaffe in das Register `R7`.

### Aufgabe 1.3: Unterprogramme

Schreiben Sie ein Unterprogramm zur Berechnung der Funktion

$$f(n) = \begin{cases} n/2 & \text{wenn } n \text{ gerade} \\ 3*n+1 & \text{wenn } n \text{ ungerade} \end{cases}$$

und ein Hauptprogramm, das dieses Unterprogramm aufruft und f(5) in R0 ablegt, f(6) in R1 und f(7) in R2.

| Label | Adresse | Befehlscode | Mnemonic | Kommentar |
|---|---|---|---|---|
| | 0000 | | | |
| | 0002 | | | |
| | 0004 | | | |
| | 0006 | | | |
| | 0008 | | | |
| | 000A | | | |
| | 000C | | | |
| | 000E | | | |
| | 0010 | | | |
| | 0012 | | | |
| | 0014 | | | |
| | 0016 | | | |
| | 0018 | | | |
| | 001A | | | |
| | 001C | | | |
| | 001E | | | |

# 2 Zusammenfassung

Machen Sie sich noch einmal die folgenden Punkte klar:

- Das Modell aufeinander aufbauender, zunehmend abstrakterer Schichten zur Beschreibung (und zum Verständnis) eines Computersystems — von der Algorithmenebene hinunter zur logischen Ebene und physikalischen Ebene.
- Den grundlegenden Aufbau eines von-Neumann-Rechners mit Steuerwerk, Operationswerk mit Registern und ALU, dem Speicher und I/O-Komponenten.
- Den Befehlszyklus mit den Phasen *fetch*, *decode* und *execute*.
- Alle Rechenwerke des System sind jederzeit aktiv und berechnen ununterbrochen Ausgangswerte. Aber von all diesen Werten werden nur die für den aktuellen Befehl benötigten Ergebnisse mit der nächsten Taktflanke abgespeichert.
- Mikroprogrammierung als direkte Umsetzung von endlichen Automaten in Hardware.
- Die Trennung zwischen Befehlsarchitektur (z.B. x86), die für den Programmierer sichtbar ist, und der Struktur des Rechners (z.B. Pentium-II als RISC-Registermaschine).
- Speicherzugriffe und I/O sind langsame Operationen. Cache-Speicher dienen dazu, die Zugriffszeiten zu verstecken.
- Die Adressierung mit Basisadresse und Offset als effiziente Möglichkeit zum Zugriff auf zusammengesetzte Datentypen.

# 3 Die Assemblerebene

Inhalt dieses Versuchs ist die Programmierung auf der Assemblerebene. Schon beim Erstellen der winzigen Maschinenprogramme dürfte klargeworden sein, dass die Programmierung in der reinen Maschinensprache sowohl extrem (zeit-) aufwändig als auch fehleranfällig ist. Ohne weitere Unterstützung lassen sich auf diese Weise keine Programme mit mehr als einigen hundert Befehlen erstellen.

Andererseits haben Sie beim Erstellen der Maschinenprogramme bereits alle Funktionen kennen gelernt, die ein Assembler automatisiert — vom Zusammensetzen von Opcode und Registerangaben zu vollständigen Befehlsworten bis zur Berechnung von Sprungadressen.

Merkmal einer *Assemblersprache* ist die 1:1-Abbildung jedes Assemblerbefehls auf einen Maschinenbefehl. Wegen dieser direkten Zuordnung zu Maschinenbefehlen sind Assembler und ihre Eingabesprachen normalerweise auf eine bestimmte Architektur zugeschnitten. Es gibt aber auch universelle Assembler (wie den GNU-Assembler oder TASM), die für eine Reihe von verschiedenen Architekturen und Prozessoren benutzt werden können. Wichtige gemeinsame Merkmale aller Assemblersprachen sind die folgenden:

- Verwendung von einprägsamen Namen für die einzelnen Befehle (*Mnemonics*)
- Einfache und reguläre Syntax für Befehlsargumente wie Register oder Speicheradressen
- Definition von symbolischen Namen für Konstanten und Sprungmarken
- Unterstützung von Kommentaren und freie Formatierung
- Umrechnung der symbolischen Programmadressen in die wirklichen physikalischen Adressen
- Erstellen von Hilfsdateien, etwa eine Liste aller verwendeten Namen, aller Sprungmarken, usw.
- Voller Zugriff auf alle Befehle und Register des benutzten Prozessors
- Evtl. Unterstützung fortgeschrittener Techniken, etwa das Einbinden mehrerer Quelldateien mittels `include` oder Makrofähigkeit
- Häufig wird der eigentliche Assembler um weitere Tools wie Debugger und Disassembler ergänzt. Damit können Details völlig vom Benutzer ferngehalten werden (etwa die Umrechnung zwischen Byte- und Wortadressen)

Obwohl ein Assemblerprogramm weiterhin auf der Ebene einzelner Befehle geschrieben wird, ist der Produktivitätsgewinn gegenüber der Maschinensprache beträchtlich. Auf der anderen Seite ist die Assemblerprogrammierung natürlich immer noch sehr viel aufwändiger als die Programmierung in Hochsprachen (wie Java oder C usw.). Trotzdem gibt es eine Reihe von guten Gründen, in Assembler zu programmieren:

- es steht (noch) kein geeigneter Compiler für eine Hochsprache zur Verfügung
- kritische Programmanteile erfordern maximale Performance
- Zugriff auf Spezialregister und privilegierte Register, etwa für Gerätetreiber
- eingeschränkte Resourcen an Programm- und Datenspeicher — viele 8-bit Mikrocontroller enthalten weniger als 1KByte RAM

## 3.1 Format der Assemblersprache

Obwohl jede Assemblersprache auf die Struktur der Befehle der zugrundeliegenden Architektur zugeschnitten ist, ähneln sich die Assemblersprachen für verschiedene Prozessoren doch sehr stark. Fast immer werden die Programme mit genau einer Assemberanweisung pro Zeile geschrieben, und jede Zeile wiederum beginnt mit einer optionalen Marke, gefolgt vom Befehl (Opcode), den Operanden, und einem optionalen Kommentar. Der Assembler `winasm.exe` für den D·CORE verwendet das folgende Format für die Eingabedateien:

```
; strtoint.asm
; Unwandeln eines nullterminierten Strings in eine Zahl
; der String steht ab Adresse 0x8000 im Speicher
; Das Ergebnis im Register R12

Start:
          movi   r10, 8          ; R10 = 8
          lsli   r10, 12         ; R10 = 0x8000 Stringadresse
          movi   r0, 0           ; zum Vergleich
          movi   r12, 0          ; Zahl initialisieren
Schleife:
          ldw    r1, 0(r10)      ; Character laden
          cmpe   r1, r0          ; = 0?  (Ende des Strings)
          bt     ende
          andi   r1, 0xf         ; Character -> Zahl
          addu   r12, r12        ; 2*r12_alt
          mov    r2, r12         ; Sichern
          lsli   r12, 2          ; 4*r12= (8* r12_alt)
          addu   r12, r2         ; 10*R12_alt
          addu   r12, r1         ; + Zahl
          addi   r10, 2          ; Adresse erhoehen
          br     Schleife
ende:
          halt

          .org    0x8000         ; Adresszaehler auf 0x8000
          .ascii  "1324"
          .defw  0               ; Null-Wort als String-Ende
          .end                   ; Kann auch weggelassen werden
```

Die Details finden Sie in der ausführlichen, separaten Beschreibung `t3asm.pdf` für den Assembler. Zusammengefasst gelten die folgenden Regeln für das Eingabeformat:

- *Kommentare* beginnen mit ; und reichen bis zum Zeilenende
- *Label-Definitionen* sind Strings, die in der ersten Spalte der Datei beginnen und mit einem Doppelpunkt abgeschlossen werden.
- *Hex-Konstanten* werden in der Schreibweise `0xCAFE` erwartet.

- Die *.org-Direktive* sorgt dafür, dass die nachfolgenden Befehle oder Konstanten ab der angegebenen Adresse `<addr>` im ROM/RAM abgelegt werden.
- Die *.defw-Direktive* dient dazu, ein bestimmtes Datenwort in die jeweilige Speicherstelle zu schreiben.
- Die *.defs-Direktive* reserviert die angegebene Anzahl von Speicherworten.
- Die *.ascii-Direktive* erlaubt es, Zeichenketten im ROM/RAM abzulegen, mit jeweils einem ASCII-Zeichen pro Speicherwort.

# 4 Maschinenarithmetik

Die nächsten Aufgaben dienen dazu, noch einmal einige Aspekte der Zahldarstellung und Integerarithmetik aufzufrischen.

**Achtung:** Die folgenden Aufgaben bauen aufeinander auf. Denken Sie deshalb daran, alle Programme abzuspeichern und ausreichend zu kommentieren (z.B. die benutzten Register), damit Sie sie wiederverwenden können! Bitte schreiben Sie zusätzlich einen Kommentar-Header, der neben Programmnamen und -funktion auch ihre Namen und Matrikelnummern enthält.

**Aufgabe 4.1: Absolutwert (Betrag)** Schreiben Sie ein möglichst kurzes Unterprogramm, um den Absolutwert (Betrag) des Inhalts des Registers `R14` zu berechnen und in `R13` zurückzuliefern. Verwenden Sie möglichst wenig Zwischenregister. Wie behandeln Sie den Eingabewert `0x8000`?

| Label | Adresse | Befehlscode | Mnemonic | Kommentar |
|---|---|---|---|---|
| | 0000 | | | |
| | 0002 | | | |
| | 0004 | | | |
| | 0006 | | | |
| | 0008 | | | |
| | 000A | | | |
| | 000C | | | |
| | 000E | | | |
| | 0010 | | | |
| | 0012 | | | |
| | 0014 | | | |
| | 0016 | | | |
| | 0018 | | | |
| | 001A | | | |
| | 001C | | | |
| | 001E | | | |

**Aufgabe 4.2: Betrag und Vorzeichen** Erweitern Sie das Unterprogramm aus der letzten Aufgabe um den Wert aus Register `R14` von der Zweierkomplementdarstellung in die Betrag-und-Vorzeichen-Darstellung umzuwandeln, wobei das höchste Bit als Vorzeichen interpretiert wird. Die positiven Zahlen bleiben dabei natürlich unverändert. Zum Beispiel soll die negative Zahl -1 von der Zweierkomplementdarstellung `0xffff` in die Betrag-und-Vorzeichen-Darstellung `0x8001` umgewandelt werden, `0xfffe` in `0x8002` usw.

| Label | Adresse | Befehlscode | Mnemonic | Kommentar |
|---|---|---|---|---|
| | 0000 | | | |
| | 0002 | | | |
| | 0004 | | | |
| | 0006 | | | |
| | 0008 | | | |
| | 000A | | | |
| | 000C | | | |
| | 000E | | | |
| | 0010 | | | |
| | 0012 | | | |
| | 0014 | | | |
| | 0016 | | | |
| | 0018 | | | |
| | 001A | | | |
| | 001C | | | |
| | 001E | | | |

**Aufgabe 4.3: 32-Bit Addition** Schreiben Sie ein Unterprogramm zur Addition von 32-Bit Zahlen. Die Argumente werden in (`R8`,`R9`) und (`R10`,`R11`) übergeben, das Resultat soll in (`R12`,`R13`) sowie dem `C`-Flag abgelegt werden (jeweils MSW,LSW). Wie viele Maschinenbefehle werden benötigt?

| Label | Adresse | Befehlscode | Mnemonic | Kommentar |
|---|---|---|---|---|
| | 0000 | | | |
| | 0002 | | | |
| | 0004 | | | |
| | 0006 | | | |
| | 0008 | | | |
| | 000A | | | |
| | 000C | | | |
| | 000E | | | |
| | 0010 | | | |
| | 0012 | | | |
| | 0014 | | | |
| | 0016 | | | |
| | 0018 | | | |
| | 001A | | | |
| | 001C | | | |
| | 001E | | | |

# 5 Indizierte Adressierung und Zeichenketten

Machen Sie sich aus Ihren Vorlesungsunterlagen noch einmal dem Begriff der *Adressierungsarten* vertraut. Beschreiben Sie die Funktion der folgenden Adressierungsarten: unmittelbare Adressierung, direkte Adressierung, Registeradressierung, indirekte Registeradressierung, indizierte Adressierung, Stapeladressierung.

**Aufgabe 5.1: Adressierungsarten** Welche der Adressierungsarten werden beim D·CORE verwendet:
für die arithmetischen Befehle:
für Speicherzugriffe:
für den jmp-Befehl:
für die Branch-Befehle:

Eine besonders wichtige Anwendung von Arrays und indizierter Adressierung sind Zeichenketten (Strings). In C und verwandte Sprachen wird eine Zeichenkette nur durch ihre Speicheradresse spezifiziert. Alle nachfolgenden Bytes bis zum ersten Null-Byte `0x00` (einschließlich) stellen die Zeichenkette dar. Einige andere Sprachen benutzen statt dessen eine zusammengesetzte Datenstruktur mit einem Integer für die Anzahl der Zeichen und einem separaten Array von Zeichen.

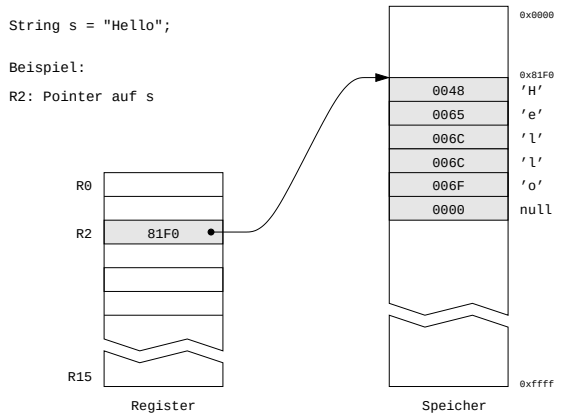


Abbildung 1: Null-terminierter String im Speicher, ein Zeichen pro Wort

**Aufgabe 5.2: strlen()** Erstellen Sie ein Assemblerunterprogramm zusammen mit einem aufrufenden Hauptprogramm für die Funktion `strlen()`, die die Länge einer Zeichenkette zurückliefert. Die Startadresse des Strings stehe in `R10`, das Resultat soll in `R11` zurückgeliefert werden. Verwenden Sie die C-Konvention mit null-terminierten Strings und 16-Bit pro Zeichen, wie in Abbildung 1 illustriert.

Einen String bekommen Sie dabei mit der Befehlsfolge

```
.org   0x8000          ; Adresse
.ascii "Ein String"    ; der String
.defw  0               ; terminierende Null
```

ab der Adresse 0x8000 in der Speicher. Setzen Sie Sie diese Anweisungen bitte immer ganz an das

Ende ihres Programms!

**Aufgabe 5.3: strcpy()** Schreiben Sie jetzt das analoge Assemblerunterprogramm für die Funktion `strcpy()`, um eine Zeichenkette zu kopieren. Die Startadresse für Original und Kopie stehe in `R10` und `R11`.

Testen Sie Ihre Funktion, indem Sie einen nicht zu langen String, der ab der Adresse 0x8000 im Speicher steht, in einen Bereich ab der Adresse 0x80A0 kopieren.