Kurzdokumentation zum D·CORE-Assembler und Emulator

Die folgenden Ausführungen setzen eine Kenntnis der Praktikumsunterlagen, insbesondere des Befehlssatzes des D·CORE-Prozessors, voraus. Darauf wird also nur im Einzelfällen eingegangen.

Den hier vorgestellten Assembler mit integriertem Emulator gibt es in zwei Version für Windows (in DELPHI) und in JAVA, die sich nur in unwesentlichen Einzelheiten unterscheiden sollten. Er ermöglicht es, ein Assembler-Programm zu editieren, zu assemblieren und dann auch zu testen. Der erzeugte Maschinencode kann dann in den Speicher des HADES-Modells geladen werden. Unterstützt wird der gesamte in der Praktikumsdokumentation angegebene Befehlssatz und einige Pseudobefehle, auf die weiter unten eingegangen wird.

Man beachte im Folgenden:
a) Der Assembler unterscheidet nicht zwischen Groß- und Kleinschreibung.
b) Ein symbolisches Label ist eine Zeichenkette, die mit einem Buchstaben beginnt, gefolgt von einem Doppelpunkt.
c) Die sechzehn Register des D·CORE haben die Namen R0, . . . , R15.
d) Kommentare beginnen mit einem “;” und enden am Ende der Zeile.
e) Zahlen können sowohl dezimal eingegeben werden als auch hexadezimal. Im zweiten Fall beginnt die Zahl wahlweise mit **0x** oder einem **$**; z.B. **$FFFF, 0xFFF, $10**.

a) ALU-Befehle

Es sind dies die Befehle **mov, addu, addc, subu, and, or, xor, not, lsl, lsr, asr, lslc, lsrc, asrc, cmpe, cmpne, cmpgt** und **cmplt**.

Allgemeine Syntax

```
Label:    Opcode   Rx,  Ry    ; Kommentar
```

Beispiel:

```
MOV    R0, R1    ; Inhalt von R1 nach R0 bringen
ADDU   R0, r2    ; R2 zu R0 addieren
```

b) Immediate-Befehle

Es sind dies die Befehle **movi, addi, subi, andi, lsli, lsri, bseti** und **bclri**.

Allgemeine Syntax

```
Label:    Opcode   Rx, Zahl    ; Kommentar
```

Beispiel:

```
MOVI   R10, 0     ; Inhalt von R10 auf 0 setzen
ADDI   R3, $F     ; 15 zu R3 addieren
```

c) Speicher-Operationen

Es sind dies die beiden Befehle **LDW** und **STW**. Man beachte, dass sie beide dieselbe Syntax haben, obwohl einmal das Ziel und einmal die Quelle links steht. Der Offset wird dabei in **Bytes** gezählt. Erlaubt sind also Werte im Bereich von 0 bis 31, wobei ungerade Werte keinen Sinn ergeben, aber nicht als Fehler betrachtet werden. Ein Offset von 0 kann auch weggelassen werden. Negative Offsets sind nicht möglich.

Allgemeine Syntax

```
Label:    Opcode   Rx, Offset(Ry)     ; Kommentar
```

Beispiel:

```
LDW  R10, (r2)    ; Lade R10 mit dem Wert, dessen
                  ; Adresse in R2 steht
STW  R10, 2(r2)   ; Speichere den Wert in die Stelle mit
                  ; der Adresse (Inhalt von R2) + 2
```

d) Branch-Operationen

Es sind dies die Befehle **BR, JSR, BT** und **BF**.

Allgemeine Syntax

```
Label:    Opcode   Adresse     ; Kommentar
```

Beispiel:

```
       MOVI   R0, 0
LOOP:
       JSR    unter    ; Ein Unterprogrammaufruf
       SUBI   R8, 1
       CMPE   r8, r0   ; R8 = 0 ?
       BF     LOOP     ; Eine Schleife
          ..........
UNTER:
          ..........
```

e) JMP-Befehl

Allgemeine Syntax

```
  Label:     JMP    Rx      ; Kommentar
```

Beispiel:

```
       JSR    unter    ; Ein Unterprogrammaufruf
          ..........
UNTER:
          ..........
       JMP    r15      ; der R"ucksprung
```

f) sonstige Befehle

Es sind dies die Befehle **RFI**, **EEPC** und **HALT**. Man beachte, dass zum Austesten jedes Programm als letzten einen HALT-Befehl haben sollte.

Allgemeine Syntax

```
  Label:     opcode      ; Kommentar
```

Beispiel:

```
       RFI            ; R"ucksprung aus einem Interrupt
       HALT           ; Anhalten
```

g) Trap-Befehl

Dieser Befehl wird vom HADES-Hardwaremodell noch nicht unterstützt, wohl aber vom Assembler, wenn man die entsprechende Option anwählt. Weiter unten wird genauer darauf eingegangen.

Allgemeine Syntax

```
  Label:     TRAP   offset       ; Kommentar
```

Außer den eben beschriebenen, eigentlichen Maschinenbefehlen des D·CORE-Prozessors braucht man normalerweise noch weitere, so genannte Pseudo-Befehle, um ein sinnvolles Assembler-Programm schreiben zu können. Implementiert sind hier die folgenden, die alle mit einem . beginnen, um deutlich zu machen, dass es sich nicht eigentliche Befehle des Prozessors handelt.

a) .ORG

Manchmal ist es nötig, genau sagen zu können, wohin man einen bestimmten Teil seines Programms in den Speicher gelegt haben möchte. Beim D·CORE muss man z.B. die Service-Routine für einen Interrupt auf eine bestimmte vordefinierte Adresse legen ($100). Dasselbe Problem tritt auf, wenn man Daten im RAM ablegen möchte. Dazu dient der .ORG-Befehl. Man beachte, dass per Default eine Anfangsadresse von 0 eingestellt ist.

Allgemeine Syntax

```
Label:     .ORG   adresse      ; Kommentar
```

Beispiel:

```
.org    0        ; Adresse 0 setzen (nicht n"otig!)
movi    r0, 0    ; Landet auf Adresse 0
BSETI   r0, 15   ; Landet auf Adresse 2
JMP     r0       ; Landet auf Adresse 4
                 ; Sprung nach Adresse $8000
.org    $8000
MOVI   r1, 1     ; Landet auf Adresse $8000
  ..........
```

b) .defw (define word)

Dieser Befehl dient dazu, ein Speicherwort mit einem bestimmten Wert zu belegen.

Allgemeine Syntax

```
Label:     .defw   wert      ; Kommentar
```

Beispiel:

```
.defw   $FFFF    ; Legt den Wert -1 auf der
                 ; aktuellen Adresse ab
.defw   $7000    ; Legt den Wert $7000 im
                 ; Speicher ab
```

c) .assho, .ascii

Diese Befehle dient dazu, einen String wortweise im Speicher abzulegen. Man beachte dabei, dass wirklich nur der String abgelegt wird, aber nicht das terminierende Null-Wort, das für die Aufgaben aus Bogen 3 nötig ist. **.ascii** ist dabei eine obsolete Form des **.assho**-Befehls, der sich nur auf ein mögliches Listing auswirkt.

Allgemeine Syntax

```
  Label:    .assho   "String"      ; Kommentar
```

Beispiel:

```
      .assho "12345"  ; Legt den String 12345 von der
                      ; aktuellen Adresse ab in den
                      ; Speicher
      .defw   0       ; Null-Wort als Ende
```

d) .defs (define storage)

Dieser Befehl dient dazu, Speicherplatz, z.B. für ein Array, zu reservieren. Der Inhalt ist dabei undefiniert.

Allgemeine Syntax

```
  Label:    .defs   Zahl      ; Kommentar
```

Beispiel:

```
      .org    $200
null: .defw   0       ; 0 in Adresse $200
ARRY: .defs   4       ; Reserviert 4 Speicherworte
                      ; beginnend bei Adresse $202
STRI: .ascii  "??"    ; Ein String; Anfangsadresse $20A
```

e) .equ (equate)

Dieser Befehl dient dazu, einen konstanten Wert zu definieren. Es wird dabei aber kein Speicherplatz belegt.

Allgemeine Syntax

```
Label:    .equ   Zahl      ; Kommentar
```

Beispiel:

```
RAM:  .equ    $8000  ; Weist RAM den Wert $8000 zu
      .org    RAM    ; Adresse setzen
      .defw   RAM    ; Auch das geht
        ........
      BR      RAM    ; und das auch, wenn der Offset sich
                     ; in 12 Bit darstellen l"asst
```

f) .end

Dieser Befehl dient dazu, das Assemblieren zu beenden. Dieser Befehl kann auch ganz fehlen.

Allgemeine Syntax

```
Label:    .end       ; Kommentar
```

Beispiel:

```
      BR       weiter
      .end
weiter:             ; Wird nicht mehr assembliert und darum
      .........     ; ergibt der BR-Befehl einen Fehler, weil
                    ; das Label WEITER nicht in die
                    ; Symboltabelle aufgenommen worden ist.
```

g) .stack, .push, .pop

.stack legt das Register fest, das von den Pseudobefehlen (Makros) **.push** und **.pop** implizit verwendet wird.

Allgemeine Syntax

```
Label:    .stack    Rx        ; Kommentar
Label:    .push     Ry        ; Kommentar
Label:    .pop      Ry        ; Kommentar
```

.push Ry erzeugt dabei die Befehlsfolge

```
          subi      Rx, 2
          stw       Ry, (Rx)
```

.pop Ry dagegen die Befehlsfolge

```
         ldw        Ry, (Rx)
         addi       rx, 2
```

wobei **Rx** das Register ist, das mit dem Pseudobefehl **.stack Rx** als Stackpoiter festgelegt worden ist. Default ist das Register **R0**. Man beachte, dass es Aufgabe des Programmierers ist, dieses Register mit einem Wert zu initialisieren, der im RAM liegt. Der **.stack**-Befehl erzeugt keinen ausführbaren Code.

Beispiel:

```
.stack    R14    ; R14 soll als Stackpointer
                 ; verwendet werden
movi      r14, 0 ; Initialisieren R14= 0x8080
bseti     r14, 15
bseti     r14, 7
...........
.push     r2     ; R2 auf den Stack retten
jsr       SUB
.pop      r2     ; R2 zurueckholen
```

Zum Abschluss dieses Teils hier noch ein Beispielprogramm, das die Zahlen in zwei Feldern addiert und das Ergebnis in einem dritten Feld ablegt.

```
      br    start
      .defw feld1     ; Adresse Feld1
      .defw feld2     ; Adresse Feld2
      .defw feld3     ; Adresse Feld3
start:
      movi  r4, 0      ; Basis-Adresse
      ldw   r10, 2(r4)
      ldw   r11, 4(r4)
      ldw   r12, 6(r4)
      movi  r2, 5      ; Schleifenzaehler
loop:
      ldw   r0, (r10)
      ldw   r1, (r11)
      addu  r1, r0
      stw   r1, (r12)
      subi  r2, 1
      cmpe  r2, r4
      bt    ende
      addi  r10, 2
      addi  r11, 2
      addi  r12, 2
      br    loop
ende:
      halt
;=============
      .org  $8000
feld3: .defs  5      ; fuenf Worte fuer das Ergebnis
feld2: .defw  0
       .defw  10
       .defw  20
       .defw  30
       .defw  40
```

```
feld1: .defw  40
       .defw  30
       .defw  20
       .defw  10
       .defw  0
       .end
```

Das eigentliche Programm

Das Hauptfenster des D·CORE-Assemblers ist denkbar einfach: eine Menue-Leiste mit den üblichen Datei- und Editpunkten, auf die hier nicht eingegangen werden muss, und zwei weiteren Punkten, die unten näher betrachtet werden; dann ein Editorfenster, in das man sein Assemblerprogramm eingeben bzw. laden kann und einen Button zum Assemblieren des Quelltextes. Wenn kein Fehler gefunden wird, werden auf Wunsch (einstellbar in Menue **Optionen**) zwei Dateien mit der Extension *.ROM und *.RAM erzeugt, die in die beiden Speicher des HADES-Modells geladen werden können. Sollten Fehler auftreten, kann man sie sich anzeigen lassen und korrigieren. Man beachte dabei, dass man in der momentanen Programmversion möglichst keine Zeilen einfügen oder löschen sollte, weil es sonst geschehen kann, dass die falsche Zeile als fehlerhaft rot eingefärbt wird. Bei den kleinen Programmen, die für das Praktikum zu entwickeln sind, sollte das keine wesentliche Einschränkung darstellen.

Jetzt zum Menuepunkt **Optionen**, über den sich verschiedene Parameter auswählen lassen:

Byteadressierung
Ist inzwischen überflüssig und sollte angeschaltet bleiben.

$-Style
Gibt an, wie der Emulator Hexadezimalzahlen anzeigen soll. Default ist die Form **0x????**. Nach Auswahl dieser Option werden die Zahlen in der Form **$????** angezeigt. Dies betrifft nicht die Eingabe; hier sind immer beide Formen möglich.

Listing
Durch Auswahl dieses Punktes erzeugt der Assembler ein Listing mit Zeilennummer, Adresse und Opcode in einer Datei *.LST.

Trap?
Nach Auswahl kann der Trap-Befehl verwendet werden. Man beachte dabei aber, dass er durch das HADES-Modell **nicht** unterstützt wird. Zu Einzelheiten beachte man die Ausführungen zu IP-Adresse.

MMU
Muss angeschaltet werden, wenn man die MMU emulieren will. Es werden dann die Adressen wie in den Versuchsunterlagen beschrieben umgesetzt.

IP-Adresse
Gibt die Adresse der Interrupt-Service-Routine (TRAP nicht aktiviert) bzw. der Interrupt-Vektor-Tabelle (TRAP aktiviert) an. Voreingestellt ist wie in HADES-Modell $100. Im Folgenden soll kurz der Unterschied erläutert werden, wobei das HADES-Modell nur den ersten Fall unterstützt, der in Bogen 4 der Praktikumsunterlagen beschrieben ist. Die IP-Adresse ist dabei einfach die Adresse, unter der die Routine steht, die ausgeführt wird, wenn ein Interrupt erfolgt. Konkret könnte das wie folgt aussehen:

```
        jsr   Eingabe
        movi  r0, 0
warten:              ; Wartet auf einen Interrupt
        br    warten
Eingabe:
        ..............
        jmp   r15    ; Ruecksprung
;============================
        .org  $100
        addi  r0, 1    ; Wenn Interrupt, dann R0= R0 + 1
        jsr   Ausgabe  ; R0 ausgeben oder etwas anderes
        rfi            ; Ruecksprung
```

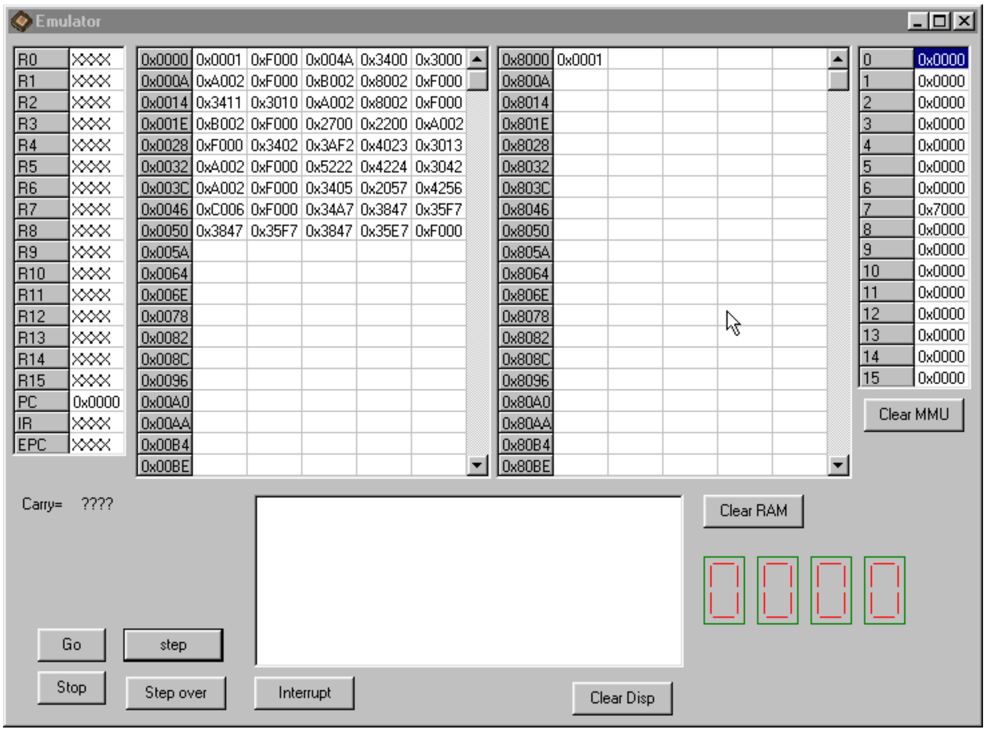
Wenn die Trap?-Option aktiviert ist, ist die IP-Adresse ein Zeiger auf eine Liste von Unterprogramm-Adressen, die mit Hilfe des Trap-Befehls angesprungen werden können. Insbesondere liegt auf der IP-Adresse die Adresse der normalen Interrupt-Service-Routine. Obiges Beispiel könnte man dann auch wie folgt formulieren:

```
        TRAP  1
        movi  r0, 0
warten:                   ; Wartet auf einen Interrupt
        br    warten
Eingabe:
        ................
        rfi               ; Ruecksprung
;============================
        .org  $100
        .defw ISR         ; Zeiger auf die Routine, die den
                          ; Interrupt behandelt
        .defw Eingabe
;============================
```

```
        .org  $200
ISR:    addi  r0, 1       ; Wenn Interrupt, dann R0= R0 + 1
        jsr   Ausgabe     ; R0 ausgeben oder etwas anderes
        rfi               ; Ruecksprung
```

Es stellt sich natürlich die Frage, was dieses Konzept überhaupt soll. Für den D·CORE-Prozessor ist eine Antwort, dass die Routine, die man anspringen möchte, irgendwo im Speicher liegen kann und nicht nur innerhalb eines Offsets von zwölf Bit zum aktuellen Wert des Programmzählers wie beim JSR-Befehl. Eine andere liegt viel allgemeiner eher auf der Betriebssystemebene, auf der es Basisroutinen geben muss, die von allen Anwendungsprogrammen genutzt werden (etwa Ausgabe eines Zeichens auf den Bildschirm oder Lesen eines Zeichens von der Tastatur). Bei einer neuen Betriebssystemversion ist dann zu befürchten, dass sich auch die Einsprungstellen dieser Unterprogramme ändern, was heißt, dass man sein Programm neu kompilieren/assemblieren/binden muss, was natürlich ein kaum haltbarer Zustand ist. Eine mögliche Lösung dieses Problems ist es, für diese Basisroutinen keine echten Unterprogramm-Aufrufe (beim D·CORE also JSR) zu verwenden, sondern Trap-Befehle, bei denen das Betriebssystem seine geänderten Einsprungstellen nur noch in einer modifizierten Tabelle abzulegen hat, was das Anwendungsprogramm nicht berührt. Wer noch das etwas zweifelhafte Vergnügen gehabt hat, Assembler-Programme für DOS zu schreiben, kennt wahrscheinlich die INT 24H und INT 10H Aufrufe, die letztlich genau dieses Konzept realisieren.

Als letzter Punkt bleibt noch das Emulieren zu besprechen. Dieser Menuepunkt kann nur angewählt werden, wenn man ein Programm unter Anwahl der Option Byteadressierung fehlerfrei übersetzt hat. Es erscheint dann folgendes Fenster:



Links oben sind die 16 Register des D·CORE nebst PC, IR und EPC zu sehen, in der Mitte bzw. rechts der untere Bereich des ROM und des RAM (für unsere Zwecke genügt das, obwohl es natürlich prinzipiell auch möglich ist, das Programm so zu erweitern, dass es mit einem vollständigen Speicher arbeitet). Ganz rechts befinden sich die Register der MMU, die nur angezeigt werden, wenn man die entsprechende Option ausgewählt hat. Darunter befindet sich ein Feld, das als Display dienen kann (Versuchsbogen 3), aber nicht auf die VT100-Escape-Sequenzen reagiert wie das Display aus dem HADES-Modell. Daneben befinden sich die vier Sieben-Segment-Anzeigen, die ebenfalls benötigt werden.

Links daneben befinden sich vier Button. **GO** startet das Programm und **STOP** hält es wieder an. **STEP** führt genau einen Befehl aus und färbt die Zeile im Hauptfenster ein. **Step over** dient dazu, Unterprogramm-Aufrufe zu überspringen.