Kurzdokumentation zum D·CORE-Assembler und Emulator

1 Kurzdokumentation der D·CORE-Assemblersprache

Die folgenden Ausführungen setzen die Kenntnis des Befehlssatzes des D·CORE-Prozessors voraus. Hierfür sei insbesondere auf die Aufgabenglätter des RSB-Praktikums verwiesen (siehe hierzu: https://tams.informatik.uni-hamburg.de/lectures/) .

Den hier vorgestellten Assembler mit integriertem Emulator gibt es als ausführbares Binary für Linux, Windows und für MAC und kann von der Website des Praktikums heruntergeladen werden (https://tams.informatik.uni-hamburg.de/):

```
t3asm-x86_64-linux.zip --- Zip-Archiv des Binary (t3asm) für Linux (Ubuntu)
t3asm-win32.zip        --- Zip-Archiv des Binary (t3asm.exe) für Windows
T3asm.dmg              --- komprimiertes Disk Image für MAC-OS;
                           (starten mit ctrl+click oder Gatekeeper "bypass-dialog")
```

Der Assembler/Emulator ermöglicht, ein Assembler-Programm zu editieren, zu assemblieren und im integrierten Emulator zu testen. Der erzeugte Maschinencode kann auch in den Speicher des HADES-Modells geladen und hier simuliert werden. Unterstützt wird der gesamte in der Praktikumsdokumentation angegebene Befehlssatz und einige zusätzliche Pseudobefehle, auf die später eingegangen wird.

Bitte beachten Sie folgende Eigenschaften:

a) Der Assembler unterscheidet nicht zwischen Groß- und Kleinschreibung.

b) Ein symbolisches Label ist eine Zeichenkette, die mit einem Buchstaben beginnt, und mit einem Doppelpunkt abgeschlossen wird.

c) Die sechzehn Register des D·CORE tragen 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, 0x0`**.

1.1 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 kopieren
            ADDU        R0, r2          ; R2 zu R0 addieren
```

1.2 Immediate-Befehle

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

Allgemeine Syntax

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

Beispiel:

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

1.3 Speicher-Operationen

Es sind dies die beiden Befehle: **LDW** und **STW**.

Bitte beachten Sie, dass sie beide Befehle die Gleiche 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 entfallen. 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
```

1.4 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           ; Unterprogrammaufruf UNTER
            SUBI        R8, 1
            CMPE        r8, r0          ; R8 = 0 ?
            BF          LOOP            ; Schleife auf LOOP
            ..........
  UNTER:
            ..........
```

1.5 JMP-Befehl

Allgemeine Syntax

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

Beispiel:

```
            JSR         UNTER           ; Ein Unterprogrammaufruf
            ..........
  UNTER:
            ..........
            JMP         r15             ; Rücksprung (jsr speichert Rücksprungadresse
                                                per Konvention in r15)
```

1.6 sonstige Befehle

Es sind dies die Befehle: **RFI**, **EEPC** und **HALT**.

Zum Austesten sollte jedes Programm als letzten Befehl einen HALT-Befehl haben.

Allgemeine Syntax

```
  Label:    opcode                      ; Kommentar
```

Beispiel:

```
            RFI                         ; Rücksprung aus einem Interrupt
            HALT                        ; Anhalten
```

1.7 Trap-Befehl

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

Allgemeine Syntax

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

1.8 Pseudo-Befehle

Außer den bereits beschriebenen Assemblerbefehlen, die die Maschinenbefehlen des D·CORE-Prozessors abbilden, werden normalerweise noch weitere Befehle, so genannte Pseudo-Befehle, für ein sinnvolles Assembler-Programm benötigt. Die Pseudo-Befehle beginnen alle mit einem Punkt (.), um sie klar von den Maschinenbefehlen des Prozessors abzusetzen.

1.8.1 .ORG

Manchmal ist es nötig, genau festzulegen, wo ein bestimmter Teil des Programms im Speicher abgelegt wird. Beim D·CORE muss z.B. die Service-Routine eines Interrupts auf einer festgelegten, vordefinierte Adresse liegen ($100).
Dasselbe Problem tritt auf, wenn Daten vom Assembler im RAM ablegt werden sollen. Dazu dient der .ORG-Befehl. Dieser (Pseudo-)Befehl setzt den Adresszähler des Assemblers auf den angegebenen Wert; die Wirkung dieses Befehles ist im Machinenprogramm daran zu erkennen, dass die auf den .ORG-Befehl folgenden Maschinenbefehle ab der angegebenen Adresse im Speicher liegen.
Per Default ist die Anfangsadresse des Assemblers auf 0x0 eingestellt.

Allgemeine Syntax

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

Beispiel:

```
           .org       0         ; Adresszähler auf 0 setzen (nicht nötig; s.o.!)
           movi       r0, 0     ; Befehl steht unter Adresse 0x0000 im Speicher
           BSETI      r0, 15    ; Befehl steht unter Adresse 0x0002 im Speicher
           JMP        r0        ; Befehl steht unter Adresse 0x0004 im Speicher
           .org       $8000     ; Adresszähler auf $8000 setzen
           MOVI       r1, 1     ; Befehl steht unter Adresse 0x8000 im Speicher
           ..........
```

1.8.2 .defw (define word)

Dieser Befehl dient dazu, ein Speicherwort mit einem bestimmten Wert zu belegen. Die Adresse des Speicherwortes wird durch den aktuellen Wert des Adresszählers des Assemblers bestimmt.

Allgemeine Syntax

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

Beispiel:

```
           .defw      $FFFF     ; Legt den Wert -1 auf der aktuellen Adresse ab
           .org       $8000     ; Adresszähler auf $8000 setzen
           .defw      $7000     ; Legt den Wert $7000 im Speicher unter
                                ; der Adresse $8000 ab
```

1.8.3 .assho, .ascii

Diese Befehle dienen dazu, einen String wortweise im Speicher abzulegen. Man beachte dabei, dass wirklich nur der String abgelegt wird, aber nicht das terminierende Null-Wort, wie es z. B. bei nullterminierten Strings in der Programmiersparache C erforderlich 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 beginnend mit der
                                  ; aktuellen Adresse (des Adresszählers) in
                                  ; den Speicher ab
            .defw       0         ; Null-Wort terminiert den String
```

1.8.4 .defs (define storage)

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

Allgemeine Syntax

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

`Zahl`: Anzahl Speicherworte

Beispiel:

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

1.8.5 .equ (equate)

Dieser Befehl dient dazu, einen konstanten Wert (assemblerintern) zu definieren. Es wird also 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 möglich
            ..........
            BR          RAM       ; ebenfalls möglich, aber Achtung:
                                  ; Branch-Offset nur 12 Bit breit
```

1.8.6 .end

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

Allgemeine Syntax

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

Beispiel:

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

1.8.7 .stack, .push, .pop

.stack legt das Register fest, das als Stackpointer verwendet wird und welches somit 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 wird. Das Default-Register ist **R0**. Es Aufgabe des Programmierers, dieses Register mit einem Wert zu initialisieren, der im Adressbereich des RAMs liegt.
Der **.stack**-Befehl erzeugt keinen ausführbaren Code.

Beispiel:

```
            .stack      R14         ; R14 als Stackpointer verwenden
            movi        r14, 0      ; Initialisieren R14= 0x8080
            bseti       r14, 15     ;
            bseti       r14, 7      ;
            ..........
            .push       r2          ; R2 auf Stack sichern
            jsr         SUB         ; Unterprogrammaufruf
            .pop        r2          ; R2 zurücksichern
```

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)  ; Adresse Summand 1
            ldw         r11, 4(r4)  ; Adresse Summand 2
            ldw         r12, 6(r4)  ; Adresse Summe
            movi        r2, 5       ; Schleifenzaehler
loop:       ldw         r0, (r10)   ; lade Summand 1
            ldw         r1, (r11)   ; lade Summand 1
            addu        r1, r0      ; berechne Summe
            stw         r1, (r12)   ; speichere Summe
            subi        r2, 1       ; dekrementiere Schleifenzähler
            cmpe        r2, r4      ; prüfe Abbruchbedingung
            bt          ende
            addi        r10, 2      ; erhöhe Adresse für Summand 1
            addi        r11, 2      ; erhöhe Adresse für Summand 2
            addi        r12, 2      ; erhöhe Adresse für Summe
            br          loop
ende:       halt
;================
            .org        $8000
feld3:      .defs       5           ; fuenf Worte fuer das Ergebnis reservieren
feld2:      .defw       0
            .defw       10
            .defw       20
            .defw       30
            .defw       40
feld1:      .defw       40
            .defw       30
            .defw       20
            .defw       10
            .defw       0
            .end
```

2 Der Assembler

Das Hauptfenster des D·CORE-Assemblers ist denkbar einfach: eine Menue-Leiste mit den üblichen Datei- und Editpunkten und einem Werkzeug- und Hilfebutton. Darunter eine Schnellzugriffsleiste für häufig benutzte Funktioen, wie Datei öffen bzw. neu anlegen, über sichern bis zu assemblieren und schließlich Starten des Emulators. Dominiert wird das Hauptfenster durch das Editorfenster, zum Bearbeiten bzw. Erstellen des Assemblerprogramms (siehe Abb: 1). Wird das Assemblieren des Quelltextes ohne Fehlermeldung abgeschlossen kann der Emulator gestartet werden. Sollten während der Assemblierung Fehler auftreten, werden diese in der Statusleiste angezeigt und die verantwortliche Programmzeile zum Korrigieren farblich hervorgehoben.

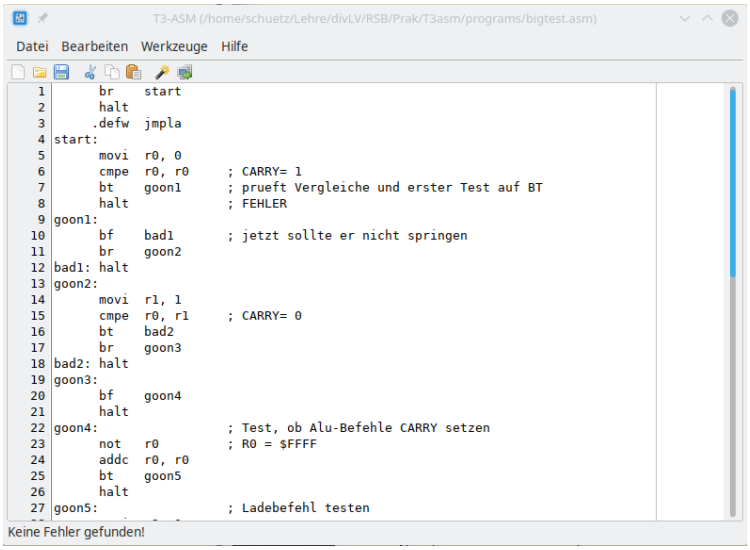

Abbildung 1: Fenster des D·CORE-Assemblers

2.1 Optionen der Assemblierung bzw. des Emulators

Über den Menuepunkt **`Optionen`** (*Bearbeiten* → *Optionen*) lassen sich verschiedene Parameter auswählen:

2.1.1 `RAM/ROM-Files`

Es werden Dateien erzeugt, die in das ROM bzw. das RAM des HADES-Modells des D·CORE-Prozessors geladen werden können. Die Dateien erhalten die Extension ROM und RAM (⟨*name*⟩.ROM und ⟨*name*⟩.RAM).

2.1.2 `Listing`

Es wird ein vollständiges Listing des Programms mit Zeilennummer, Adresse und Opcode unter dem Namen ⟨*name*⟩.LST erzeugt.

2.1.3 Altera

Es wird eine ROM-Datei für den Altera RAM/ROM-Generator erzeugt. Dieses Format ist veraltet.

2.1.4 Byte-Offset

Die Byteadressierung wird standardmäßig verwendet und sollte angeschaltet bleiben.

2.1.5 Steps?

Bei aktivierter Steps?-Option wird die Anzahl der nach der Betätigung der Schaltfläche GO ausgeführten Instruktionen angezeigt. Diese Option wird nicht mehr unterstützt.

2.1.6 $-Style

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

2.1.7 Trap?

Nach Auswahl kann der Trap-Befehl verwendet werden. Dabei ist zu beachten, dass das HADES-Modell diese Anweisung **nicht** unterstützt. Weitere Einzelheiten sind in den Ausführungen zur ISR-Adresse zu finden.

2.1.8 SimuRAM

Ohne Funktion

2.1.9 MMU

Das D·CORE-System verfügt über eine einfache Speicherverwaltungseinheit (Memory Management Unit, kurz: MMU). Soll diese verwendet werden, ist diese Option zu aktivieren. Um die erforderliche Speicherverwaltung auf ein Minimum zu reduzieren, wird der Haupspeicher in 16 Seiten á 4 KByte Größe aufgeteilt und darüberhinaus auf jegliche Statusinformationen verzichten. Zur Umsetzung der virtuellen Adressen in physikalische Adressen genügt daher ein einstufiges Adressumsetzungsschema (siehe 2) und damit lediglich ein kleines RAM mit 16 Worten für die Basisadresse der Seite (die obersten vier Adressbits der virtuellen Adresse), während die unteren 12 Bit der Adresse als Seitenoffset direkt durchgereicht werden.
Die Seitentabelle wird unter den Adressen 0x7010 .. 0x702E in den Adressraum eingeblendet. Die Abbildung 3 zeigt mögliche Belegungen der Seitentabelle.

2.1.10 Interr.?

Diese Option schaltet die Interruptbehandlung ein. Im Emulator kann durch die Betätigung der Schaltfläche *Interrupt* ein Interrupt ausgelöst werden. Nach Abarbeitung des aktuellen Befehles wird die beginnend an der Adresse 0x0100 abgelegte Interrupt-Service-Routine angesprungen und nach deren Abarbeitung wieder in das unterbrochene Programm zurüchgesprungen (siehe ISR-Adresse).

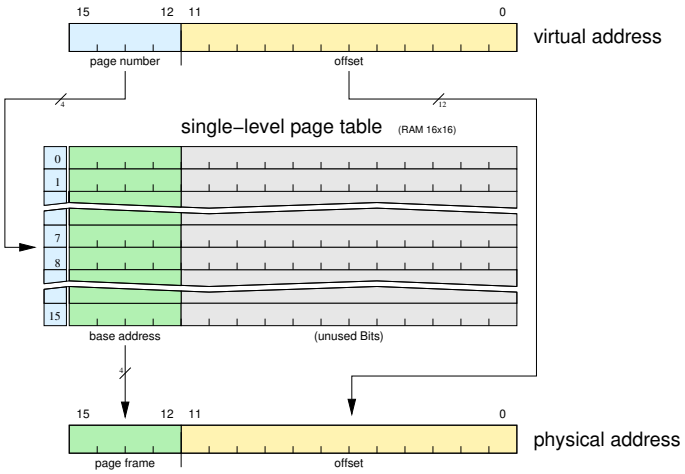

Abbildung 2: Einfache einstufige Adressumsetzung ohne Statusbits. Da aus Gründen der Einheitlichkeit anstelle eines 16x4 Bit RAM ein 16x16 Bit RAM verwendet wurde, sind die niederwertigen 12 Bit don't care.

2.1.11 ISR-Adresse

Gibt die Adresse der Interrupt-Service-Routine (TRAP nicht aktiviert) bzw. der Interrupt-Vektor-Tabelle (TRAP aktiviert) an. Voreingestellt ist ist die Adresse $100, wie im HADES-Modell. Im Folgenden soll kurz der Unterschied erläutert werden, wobei das HADES-Modell nur die Interrupt-Service-Routine unterstützt. Die ISR-Adresse ist dabei die Adresse, unter der die Routine steht, die ausgeführt wird, wenn ein Interrupt erfolgt. Beispielsweise könnte es 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 ISR-Adresse ein Zeiger auf eine Liste von Unterprogramm-Adressen, die mit Hilfe des Trap-Befehls angesprungen werden können. Insbesondere liegt auf der ISR-Adresse die Adresse der normalen Interrupt-Service-Routine. Obiges Beispiel könnte man dann auch wie folgt formulieren:

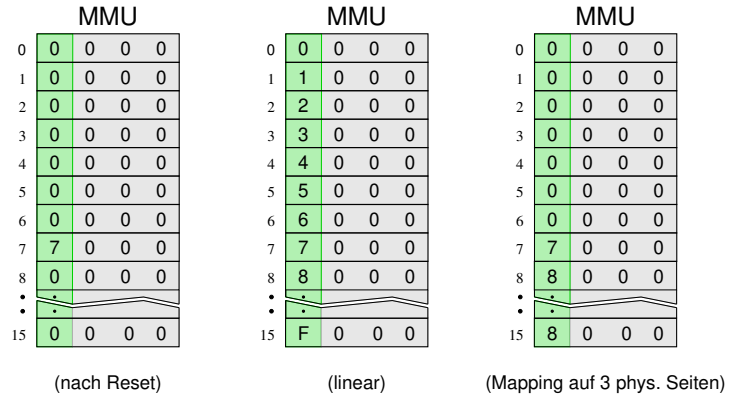

Abbildung 3: Beispiele für die Adressumsetzung: links die Seitentabelle nach Reset mit Abbildung des nahezu gesamten Adressraumes auf die erste Seite des ROMs, wobei lediglich der Index 7 als Page Table Entry (PTE) die 7 enthält. In diesem Adressbereich liegen beispielsweise die Memory Mapped I/O Adressen und die Seitentabelle selbst (deshalb ist deser Eintrag auch zwingend erforderlich). Die mittlere Tabelle illustriert ein lineares Mapping, während die letzte Abbildung ein Mapping zeigt, wie es beim D·CORE mit einem Prozess und Zugriff auf ROM und RAM möglich wäre. Es kann hier letztendlich neben dem eingeblendeten Adressbereich nur auf zwei weitere physikalische Seiten (erste Seite im ROM und erste Seite im RAM) zugegriffen werden.

```
            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, welchen Vorteil dieses Konzept mit sich bringt. Ein Vorteil für den D·CORE-Prozessor ist, dass die Routine, die angesprungen werden soll, irgendwo im Speicher liegen kann und nicht nur innerhalb eines 12 Bit-Offsets zum aktuellen Wert des Programmzählers wie beim JSR-Befehl.
Eine weiterer Vorteil liegt viel allgemeiner eher auf der Betriebssystemebene, auf der es Basisrou-

tinen geben muss, die von allen Anwendungsprogrammen genutzt werden (etwa die Ausgabe eines Zeichens auf den Bildschirm oder das 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 die Anwenderprogramme neu kompiliert/assembliert/gebunden werden müssen, was inakzeptabel wäre. Eine mögliche Lösung dieses Problems wäre, für diese Basisroutinen keine klassischen 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 Anwenderprogramm nicht berührt. Sollte jemand noch das etwas zweifelhafte Vergnügen Assembler-Programme für DOS zu schreiben gehabt haben, so kennt derjenige wahrscheinlich die INT 24H und INT 10H Aufrufe, die letztlich genau dieses Konzept realisieren.

3 Der Emulator

Der Emulator kann aus dem Assembler-Fenster heraus über das Menü (*Werkzeuge* → *Emulator Starten*) oder über das Schnellzugriff (*Monitor-Ikon*) gestartet werden. Voraussetzung hierfür ist, dass ein Programm unter Anwahl der Option Byteadressierung fehlerfrei übersetzt worden ist. Es erscheint dann das in Abb: 4 gezeigte Fenster.

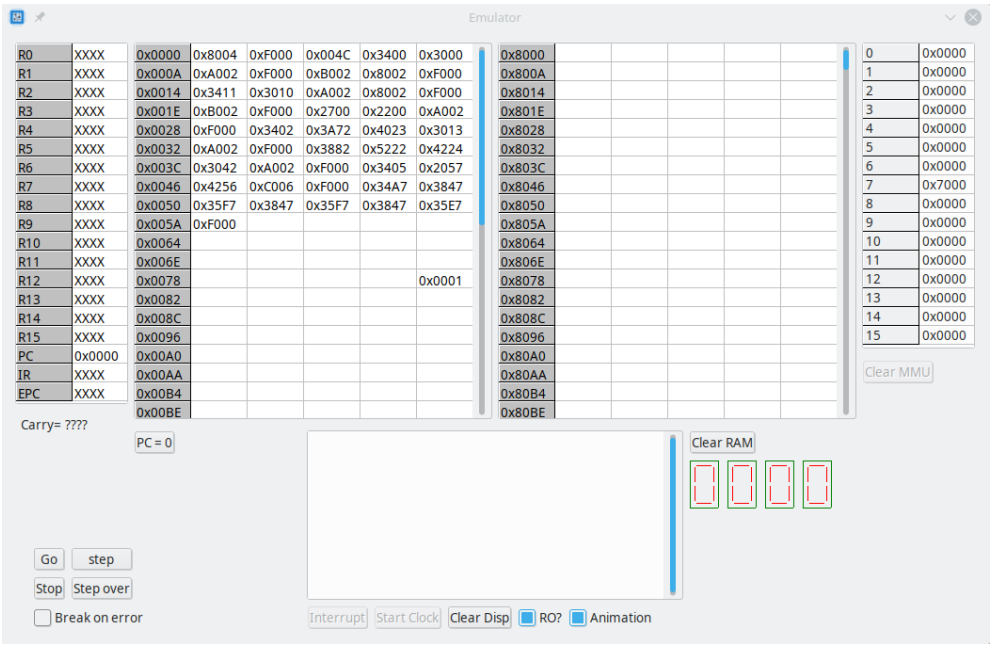

Abbildung 4: Fenster des D·CORE-Emulators

3.1 Anzeige- und Steuerelemente des Emulators

Der folgende Abschnitt erläutert die Anzeigeelemente mit den ggf. dazugehörigen Steuerelementen und die grundsätzlichen Schaltflächen zur Steuerung des Emulators (siehe Abb. 4).

3.1.1 Register, RAM und ROM des D·CORE

Am linken Rand nach unten auslaufend sind die 16 Register des D·CORE nebst PC, IR und EPC zu sehen. Rechts daneben wird das ROM angezeigt und daneben wiederum das RAM. Sowohl RAM und ROM sind jeweils rechts mit einem Slider versehen, um weitere Speicherzellen anzuzeigen. Sowohl Registerinhalte als auch Inhalte der Speicherzellen können bei pausierter Emulation verändert werden.
Über die Schaltfläche *PC=0* lässt sich der *PC* komfortabel auf `0x0` setzen und über *Clear RAM* das RAM löschen (alle Zellen auf `0x0` setzen).

3.1.2 Die MMU

Am rechten Rand des Emulator-Fensters werden die Register der MMU (die Seitentabelle) angezeigt, die allerdings nur relevant sind, wenn die entsprechende Option (siehe 2.1.9) ausgewählt ist. Bei nicht gesetzter Option wird die Schaltfläche `Clear MMU` ausgegraut, mit der ggf. die Register der MMU, die Seitentabelle, gelöscht werden kann (auf `0x0` setzen).

3.1.3 Das alphanumerische Display

Im unteren Bereich des Emulatorfensters befindet sich mittig ein Feld, das als Display dienen kann. Der Assembler kennt hierfür drei spezielle Pseudobefehle, um etwas auf diesem Display des Emulators ausgeben zu können und zwar (siehe Versuchsbogen Bogen 4 des RSB-Praktikums, Aufgabe 4.9):

- **`.prdez`** ⟨***reg***⟩ gibt den Inhalt des Registers ⟨*reg*⟩ als Dezimalzahl aus
- **`.prnewline`** gibt einen Zeilenumbruch aus
- **`.prstr`** ⟨***reg***⟩ gibt den im ROM/RAM liegenden String aus, dessen Startadresse im Register ⟨*reg*⟩ steht

Dieses Display reagiert aber nicht auf die VT100-Escape-Sequenzen wie das Display des HADES-Modells.

3.1.4 Sieben-Segment-Anzeige

Rechts neben dem Display befinden sich die vier Sieben-Segment-Anzeigen, die Memory Mapped über die Adresse *0x7002* angesprochen werden können. Bei einem auszugebenden 16 Bit Wert ergeben nur die Kodierungen einer vierstelligen BCD-Zahl sinnvolle Anzeigen. Kodierungen im Bereich der Pseudotetraden der jeweiligen BCD-Ziffern ergeben z.T. unsinnige Darstellungen.

3.1.5 Steuerungselemente unterhalb des Displays

Interrupt – bei gesetzter `Interr.?`-Option (siehe Abschnitt 2.1.10) kann hier ein Interrupt ausgelöst werden

Start Clock – wenn die Option `Interr.?` (siehe Abschnitt 2.1.10) gesetzt ist, kann durch Betätigung von `Start Clock` neben der manuellen Interruptauslösung (s.o.) automatisch wiederkehrend entsprechend der im Optionenfeld `Clock` gesetzten Anzahl Takte ein Interrupt ausgelöst werden

Clear Disp – löscht das alphanumerische Display

RO – `RO` steht für read-only. Bei gesetztem `RO` ist das Terminal aus Sicht des Benutzers read-only, und somit aus Sicht des Prozessors write only, also ein Ausgabegerät. Bei nicht gesetztem `RO` kann das Terminal auch als Eingabegerät verwendet werden.

Animation – Bei Entfernen dieser Option werden Animationen der GUI, z.B. das Hervorheben, an welche ROM Adresse gelesen wird, abgeschaltet. Dadurch läuft die Emulation geringfügig schneller.

3.1.6 Steuerung des Emulators

Links neben dem Textdisplay befinden sich fünf Schaltflächen zur elementaren Steuerung des Emulationslaufes:

Go – startet das Programm; die Emulation wird entweder durch `Stop` angehalten oder durch Ausführen eines Halt-Befehls beendet

Stop – stoppt die Programmausführung

Step – führt genau einen Befehl aus und färbt die entsprechende Zeile im Assemblercode ein

Step over – Unterprogramm-Aufruf wird übersprungen

Break on error – wird zur Laufzeit ein Fehler festgestellt, z.B. Schreibzugriff ins ROM, wird die Ausführung gestoppt und die auslösende Anweisung im Assemblercode hervorgehoben