DLX-Assembler-Programmierung

Kutil, 2010

Es gibt mehrere Simulatoren für den DLX-Prozessor. Erstens WinDLX, ein altes 16-Bit-Windows-Programm mit GUI und Dokumentation. Zweitens dlxsim, ein Unix-Programm mit Command-Line-Interface, beschrieben weiter hinten in diesem Dokument. Und drittens einen Web-Simulator auf der LV-Seite, basierend auf dlxsim.

Die DLX-Befehle selbst werden im Detail weiter hinten erklärt.

In den Beispielen wird Java-Code und DLX-Assemblercode gegenübergestellt (einmal C-Code). Die korrespondierenden Anweisungen stehen nach Möglichkeit in der gleichen Zeile. Dadurch soll die Funktion der Codeteile erklärt werden und die Beziehung zwischen Assembler und höheren Programmiersprachen verdeutlicht werden.

1 Wie fange ich an?

Der Programmcode muss normalerweise mit einem Editor erzeugt werden und in eine Datei mit Endung `.s` geschrieben werden und im jeweiligen Simulator geöffnet/geladen werden. Für echte Prozessoren würde ein Assembler den Code in Maschinencode in einer Objet-Datei umwandeln, der dann in den Speicher des Computers geladen werden kann.

Wie sieht so ein Programm nun eigentlich aus? Also: Input und Output gibt es nicht. Es gibt nur Daten, die irgendwo im Speicher stehen. Die werden initialisiert (oder auch nicht) und nach Ablauf des Programms sieht man nach, was drin steht.

Ein Programm wird in zwei Teile geteilt. Im ersten Teil werden die Daten definiert. Dieser Teil wird eingeleitet mit `.data`. Im zweiten Teil steht der Assemblercode. Dieser Teil wird eingeleitet mit `.text`.

Java

```
1
2 int var1 = 7;
3 int var2 = 3;
4
5
6 var2 = var1;
7
8
```

DLX Asm

```
1         .data
2 var1:   .word   7
3 var2:   .word   3
4
5         .text
6 main:   lw      r1, var1
7         sw      var2, r1
8         trap    #0
```

Was steht da? Also ganz links stehen die Labels. Mit Hilfe dieser Labels kann man nachher die Adresse im Speicher erreichen, an der das Ding steht, das man rechts vom Label definiert. Das können sowohl Daten als auch Codeteile sein (Sprungadressen).

Mit `.word` wird der Typ eines Datums definiert (ein Wort eben), das an dieser Stelle stehen soll. Dahinter steht der Inhalt. Also: an der Stelle `var1` soll ein Wort mit Inhalt 7 stehen. Ein Wort hat hier 4 Byte, kann also Werte von 0 bis 2^{32} bzw. von -2^{31} bis $2^{31} - 1$ beinhalten, je nachdem ob man mit oder ohne Vorzeichen arbeitet. Das legt man aber erst fest, wenn man die Daten manipuliert. Neben `.word` gibt es noch `.byte`, `.float` und `.double`. Mit `.ascii` kann man Strings definieren, mit `.asciiz` sind diese null-terminiert. Mit `.space` kann man einen (uninitialisierten) Speicherbereich definieren, dessen Größe man hinter `.space` in Bytes angibt.

Hinter `main:` steht der erste Befehl, der ausgeführt wird. Die Befehle haben meist die Form "Befehl Ziel,Quelle" oder "Befehl Ziel,Quelle,Quelle". In diesem Fall ist der erste Befehl `lw`, ausgesprochen "load word". Dieser Befehl nimmt das Datum an der Speicherstelle "Quelle"

und schreibt es in ein Register mit Namen "Ziel". (Mehr zu Registern später.) Die "Quelle" ist in diesem Fall die Variable `var1`, genauer gesagt die Adresse, an der im Speicher diese Variable steht. Das "Ziel" ist das Register `r1`. Der nächste Befehl schreibt nun den Inhalt des selben Registers an die Stelle `var2`. Insgesamt wird also `var1` nach `var2` kopiert, genau so wie im Java-Code.

Der nächste Befehl, `trap #0`, ist eine Verlegenheitslösung und soll nur das Ende des Programms darstellen. Er signalisiert dem DLX-Simulator einen Ausnahmezustand (Exception), was diesen veranlasst, anzuhalten. Ohne diesen Befehl würden die Zufallsdaten, die hinter dem Programm im Speicher stehen, als Maschinenbefehle missinterpretiert werden und das Programm dadurch wirren Unfug treiben (abstürzen oder sich aufhängen).

Die meisten Befehlscodes sind aus folgenden Kürzel zusammengesetzt: l = load, s = store oder set, b = branch, j = jump, eq = equal, ne = not equal, gt = greater than, lt = less than, ge = greater or equal, le = less or equal, r = register, z = zero, w = word, h = half word, b = byte, i = immediate. Damit dürfte sich die Bedeutung der einzelnen Operationen einfach erschließen.

2 Register

Der DLX hat 32 Register, in denen Wörter zwischengespeichert werden können. Sie heißen r0, r1, ..., r31. Sie können fast beliebig als Quelle und Ziel von Befehlen verwendet werden. Folgendes Programm berechnet das Polynom $y = ax^2 + bx + c$.

Java

```
1
2  int a = 3;
3  int b = 4;
4  int c = -5;
5  int x = 2;
6  int y;
7
8
9  y = a
10
11     * x
12     * x +
13     b
14     * x
15
16
17     + c;
```

DLX Asm

```
1          .data
2  a:      .word   3
3  b:      .word   4
4  c:      .word   -5
5  x:      .word   2
6  y:      .word   0
7
8          .text
9  start:  lw      r1, a
10         lw      r2, x
11         mult    r1, r1, r2
12         mult    r1, r1, r2
13         lw      r3, b
14         mult    r3, r3, r2
15         add     r1, r1, r3
16         lw      r4, c
17         add     r1, r1, r4
18         sw      y, r1
19         trap    #0
```

Es werden also Register für Variablen und Zwischenergebnisse verwendet. Es ist sehr hilfreich, sich diese Zuordnung zu notieren, damit man den Überblick nicht verliert. Und das ist durchaus ernst gemeint!

```
r1=y, r2=x, r3=b*x, r4=c
```

Das Register `r0` wurde hier aus gutem Grund nicht verwendet. Es hat eine spezielle Funktion. Es ist immer gleich 0 und kann daher nicht beschrieben werden. Was das für einen Sinn hat? Man kann damit zB Werte von Register zu Register kopieren. Der Befehl `add r4,r5,r0` kopiert

zB `r5` nach `r4`. Für sowas hat der DLX nämlich keinen eigenen Befehl.

3 Verzweigungen

Hurra, in Assembler dürfen Sie GOTOs verwenden, ja Sie müssen sogar! Genießen Sie diese Gelegenheit! Hier heißen die GOTOs Jump oder Branch. Jumps sind die unbedingten Sprünge, d.h. es wird immer gesprungen. Bei Branches wird der Sprung nur durchgeführt, wenn eine Bedingung erfüllt ist (bedingter Sprung).

Folgender Programmteil begrenzt den Wert von `R1` mit -10 nach unten und $+20$ nach oben:

Java

```
1  if (x < -10)
2
3    x = -10;
4
5  else if (x > 20)
6
7    x = 20;
```

DLX Asm

```
1          slti    r2,r1,#-10  ; r2!=0 wenn r1<-10
2          beqz    r2,elsif    ; zu else, wenn r2=0
3          addi    r1,r0,#-10  ; r1=-10
4          j       endif
5  elsif:  sgti    r2,r1,#20   ; r2!=0 wenn r1>20
6          beqz    r2,endif    ; zu endif, wenn r2=0
7          addi    r1,r0,#20   ; r1=20
8  endif:
```

Das hinter den Strichpunkten sind Kommentare. Übrigens Musterbeispiele an bescheuerten Kommentaren, weil sie nur das aussagen, was eh schon aus den Befehlen selbst hervorgeht. Sollten Sie selbst irgendwann mal was kommentieren, finden Sie bessere Kommentare! Sowas wie der Java-Code links zB.

4 Schleifen

Wenn man rückwärts springt, entsteht eine Programmschleife. Dabei ist es wichtig, den Schleifenabbruch richtig hinzukriegen. Vor allem für den Fall, dass – abhängig von den Daten – die Schleife nur einmal oder gar nicht durchlaufen wird.

Als erstes sehen wir hier eine `for`-Schleife:

Java

```
1
2  int n = 16;
3  byte[] data = new byte[n];
4  int i;
5
6
7
8  for (i = 0; i < n; i ++)
9  {
10
11   data[i] *= 2;
12
13
14 }
```

DLX Asm

```
1          .data
2  n:      .word   16
3  data:   .space  16
4
5          .text
6  start:  lw      r1, n           ; r1 = n
7          add     r2, r0, r0      ; i = 0
8  loop:   slt     r3, r2, r1      ; Wenn nicht i < n,
9          beqz    r3, endloop     ; dann Abbruch
10         lb      r4, data(r2)    ; loop body
11         add     r4, r4, r4      ; loop body
12         sb      data(r2), r4    ; loop body
13         addi    r2, r2, #1      ; i++
14         j       loop            ; nächste Iteration
15 endloop:
16         trap    #0
```

Es ist zu beachten, dass die Abbruchbedingung zu Beginn der Schleife geprüft wird. Das mag

anfangs etwas eigentümlich wirken, aber falls `n` gleich 0 ist, darf die Schleife gar nicht durchlaufen werden. Deshalb *muss* der Schleifenabbruch zu Beginn der Schleife stattfinden. Schleifen, die mindestens einmal durchlaufen werden müssen, gibt es zwar auch, die sind aber wirklich nicht häufig.

Die zweite wichtige Schleifenart ist die `while`-Schleife. Hier ein Beispiel, das den abgerundeten 2-er-Logarithmus von `n` berechnet:

Java

```
1
2 int l = 0;
3 while (n > 1)
4 {
5   n /= 2;
6   l ++;
7 }
```

DLX Asm

```
1         lw      r1, n           ; lese n
2         add     r2, r0, r0      ; Log. null
3 loop:   sgt     r3, r1, #1      ; Wenn nicht n > 1
4         beqz    r3, endloop     ; dann Abbruch
5         sra     r1, r1, #1      ; shift right = Div.
6         addi    r2, r2, #1      ; Log. erhöhen
7         j       loop            ; nächste Iteration
8 endloop:
```

5 Pointer

Im Beispiel zur `for`-Schleife haben wir auf eine Array von Bytes zugegriffen mittels `data(r2)` (bzw. `data[i]` in Java). Dabei ist `r2` (bzw. `i`) der Index. Das geht deshalb so einfach, weil die Elemente genau die Größe 1 haben (1 Byte). Daher errechnet sich die Adresse von `data[i]` aus der Adresse `data` plus dem Index (`i` bzw. `r2`). Wenn wir auf ein Array von Wörtern zugreifen, muss man aber den Index mit 4 multiplizieren, weil ein Wort eben vier Bytes lang ist. Also so:

Java

```
1
2 int[] data = new int[16];
3 int i = 3;
4
5
6 data[i] *= 2;
```

DLX Asm

```
1         .data
2 data:   .space  64
3 i:      .word   3
4
5         .text
6 start:  lw      r2, i           ; lade i
7         sla     r3, r2, #2      ; berechne i*4
8         lw      r4, data(r3)    ; lade von data+i*4
9         add     r4, r4, r4      ; *= 2
10        sw      data(r3), r4    ; schreibe data+i*4
```

Wenn man aber jedes mal beim Zugriff auf ein Array die richtige Adresse erst mühsam errechnen muss, ist das umständlich und auch langsam. Eine bessere Methode ist die Verwendung von Pointern. Das sind Adressen, die in einem Register gespeichert werden. Pointer zeigen mitten in ein Array hinein. Pointer werden nach vor und zurück verschoben, indem man die Größe eines Elements addiert bzw. subtrahiert. Das Beispiel mit der `for`-Schleife sieht für ein `word`-Array dann so aus:

C

```
1
2  int n = 16;
3  int data[16];
4  int i;
5
6  int *ptr = data;
7
8  for (i = n; i > 0; i --)
9  {
10   *ptr *= 2;
11
12   ptr ++;
13
14 }
```

DLX Asm

```
1          .data
2  n:      .word   16
3  data:   .space  64
4
5          .text
6  start:  addui   r2, r0, data    ; Pointer laden
7          lw      r1, n           ; r1 = n = i
8  loop:   beqz    r1, endloop     ; Abbruch bei i=0
9          lw      r4, 0(r2)       ; loop body
10         add     r4, r4, r4      ; loop body
11         sw      0(r2), r4       ; loop body
12         addi    r2, r2, #4      ; Pointer erhöhen
13         subi    r1, r1, #1      ; i--
14         j       loop            ; nächste Iteration
15 endloop:
16         trap    #0
```

Man beachte, dass das Array jetzt vier mal so viel Platz braucht (64 statt 16 Bytes). In Zeile 6 wird Register `r2` mit der Adresse von `data` geladen und dient damit als Pointer. In Zeile 9 und 11 wird über den Pointer auf das Array zugegriffen. In Zeile 12 wird der Pointer um ein Wort (4 Bytes) erhöht.

Da `i` jetzt nicht mehr als Index gebraucht wird und nur noch zum Zählen der Schleifeniterationen verwendet wird, kann man `i` von 10 herunterzählen lassen. Das ist ein beliebter Trick, mit dem man sich einen `slt`-Befehl bei der Endebedingung sparen kann (Zeile 8). `i` wird einfach mit `beqz` auf 0 getestet.

6 Unterprogramme

Unterprogramme werden in Assembler aufgerufen, indem man an den Anfang des Unterprogramms springt und am Ende wieder zurückspringt. Wenn das Unterprogramm aber von mehreren Stellen aus aufgerufen wird, weiß es natürlich nicht, wohin es zurückspringen soll. Daher muss man dem Unterprogramm in einem Register die Rücksprungadresse mitteilen. Zu diesem Zweck hat der DLX-Prozessor einen eigenen Befehl: `jal` (Jump and link). Dieser speichert die Adresse des nachfolgenden Befehls im Register `r31`. Der Rücksprung erfolgt demgemäß mit `jr r31`.

Java

```
1
2  static void main ()
3  {
4    e = max (a, b)
5
6
7
8        + max (c, d);
9
10
11 }
12
13 static int max (int p, int q)
14 { if (p < q)
15       p = q;
16   return p;
17 }
```

DLX Asm

```
1          .text
2  main:   lw      r1, a
3          lw      r2, b
4          jal     max
5          add     r3, r0, r1
6          lw      r1, c
7          lw      r2, d
8          jal     max
9          add     r3, r3, r1
10         sw      e, r3
11 end:    trap    #0
12
13 max:    slt     r4, r1, r2  ; r1=p, r2=q
14         beqz    r4, ismax
15         add     r1, r0, r2
16 ismax:  jr      r31         ; r1=return value
```

Sowohl die Argumente (p, q) als auch das Resultat werden in Registern übergeben.

Es muss darauf geachtet werden, dass das Unterprogramm nicht Register überschreibt, die vom aufrufenden Programm verwendet werden. Da dies vor allem bei größeren Programmen problematisch werden kann (Fehlerquelle: man übersieht leicht etwas), gibt es zwei gängige Lösungen:

Callee-Save Das aufgerufene Unterprogramm sichert die Inhalte aller Register, die es verändert, vorher im Speicher und holt sie vor dem Rücksprung wieder zurück.

Caller-Save Das aufrufende Programm sichert alle Register, deren Inhalte nicht verloren gehen dürfen, vor dem Unterprogramm-Aufruf im Speicher und holt sie nachher wieder zurück.

Bei beiden Varianten wird möglicherweise zu viel gesichert. Das gleiche Problem tritt auch für das Register `r31` auf, wenn ein aufgerufenes Unterprogramm ein weiteres Unterprogramm aufruft. Dabei würde die erste Rücksprungadresse verloren gehen. Das Register `r31` kann klarerweise nur nach dem Caller-Save-Prinzip gesichert werden. Hier ist das Programm noch einmal in beiden Save-Varianten:

Caller-Save

```
1         .data
2  r3sv:  .word   0
3  r31sv: .word   0
4
5         .text
6  main:  lw      r1, a
7         lw      r2, b
8         sw      r31sv, r31 ; save
9         jal     max
10        lw      r31, r31sv ; restore
11        add     r3, r0, r1
12        lw      r1, c
13        lw      r2, d
14        sw      r3sv, r3   ; save
15        sw      r31sv, r31 ; save
16        jal     max
17        lw      r3, r3sv   ; restore
18        lw      r31, r31sv ; restore
19        add     r3, r3, r1
20        sw      e, r3
21 end:   trap    #0
22
23 max:   slt     r3, r1, r2 ; jetzt darf
24        beqz    r3, ismax  ; r3 verw.
25        add     r1, r0, r2 ; werden
26 ismax: jr      r31
```

Callee-Save

```
1         .data
2  r3sv:  .word   0
3
4         .text
5  main:  lw      r1, a
6         lw      r2, b
7         jal     max
8         add     r3, r0, r1
9         lw      r1, c
10        lw      r2, d
11        jal     max
12        add     r3, r3, r1
13        sw      e, r3
14 end:   trap    #0
15
16 max:   sw      r3sv, r3   ; save
17        slt     r3, r1, r2 ; jetzt darf
18        beqz    r3, ismax  ; r3 verw.
19        add     r1, r0, r2 ; werden
20 ismax: lw      r3, r3sv   ; restore
21        jr      r31
```

Bei diesem Schema bekommt man noch immer Probleme, falls man ein rekursives Unterprogramm programmieren möchte. Da die Register immer an die selbe Stelle gesichert werden, wird die Sicherung beim ersten Selbstaufruf überschrieben. Als Lösung kann man einen Stack implementieren. Man reserviert einen Speicherbereich und setzt ein dediziertes Register als Stackpointer ein, der auf diesen Speicherbereich zeigt. Nach jeder Register-Sicherung an die Stelle, wo der Stackpointer hinzeigt, wird der Stackpointer erhöht und vor der Wiederherstellung wieder vermindert. Der Stack kann von allen Unterprogrammen gemeinsam benutzt werden. Ordentliche Prozessoren bieten für sowas Unterstützung im Befehlssatz an.

7 Vorsicht, Falle!

7.1 Adressen passen eigentlich nicht in Immediate-Werte

So, jetzt lernen wir, dass alles, was wir bis jetzt gemacht haben, gar nicht funktionieren kann. Naja, nur sehr eingeschränkt halt. Das Problem ist, dass wir an mehreren Stellen Speicheradressen durch Immediate-Werte angesprochen haben. So zB in "`lw r1,a`", "`sw data(r3),r1`" oder "`addui r2,r0,data`". Der ganze Adressraum umfasst 2^{32} Byte, Immediate-Werte fassen aber nur 16 Bit. Falls also eine Variable an einer Speicherstelle über 65535 zu finden ist, sind diese Befehle kaputt. Hier eine Anleitung, wie man es richtig machen müsste:

Kaputte Befehle

```
1        addui   r2, r0, data
2
3
4        lw      r1, a
5
6
7        sw      data(r3), r1
8
9
```

So wär's richtig

```
1        lhi     r2, data >> 16
2        addui   r2, r2, data & 0xffff
3
4        lhi     r2, a >> 16
5        lw      r1, (a & 0xffff)(r2)
6
7        lhi     r2, data >> 16
8        add     r2, r2, r3
9        sw      (data & 0xffff)(r2), r1
```

Tja, blöder Prozessor! Bei Sprungadressen ist dieses Problem zum Glück nicht so akut, weil Sprungadressen relativ sind (zB 23 Befehle nach vor, 15 zurück, etc.). Nur wenn Programmteile weit über den Adressbereich verstreut sind, werden auch Sprünge kritisch.

7.2 Label Alignment

Ein anderes Problem lässt sich leichter beheben: Manchmal definiert man Daten, deren Größe nicht ein Vielfaches von 4 ist (zB Textstrings). Nachfolgende Daten sollten aber womöglich an einer Adresse stehen, die durch 4 teilbar ist. Das nennt man "word aligned". Daher sollten gerade so viele Bytes freigelassen werden, dass das erfüllt ist. Die Anweisung `.word` erledigt das für uns. Das `start:` Label vor dem ersten Code macht da aber oft Probleme. Die Lösung ist die Anweisung "`.align 4`". Diese ist am besten eine Zeile vor `.text` einzufügen.

7.3 Branch Delay Slots

Manche Simulatoren implementieren einen branch delay slot, bei manchen kann man ihn einstellen. WinDLX benutzt ihn (defaultmäßig) nicht. Was ist ein branch delay slot? Nun, da wird einfach der erste Befehl *nach* einem Branch oder Jump *immer* ausgeführt, egal ob gesprungen wird oder nicht. Dadurch können beim Pipelining Stalls verhindert werden, die durch Verzweigungen entstehen.

Um also DLX-Programme zu schreiben, die auf jedem DLX-Simulator richtig laufen, ist es notwendig, nach jedem Branch oder Jump einen Befehl einzubauen, der nichts tut, also den Befehl `nop`. Der Web-Simulator benutzt die branch delay slots, es gibt allerdings die Einstellung "insert nops into branch delay slots", die beim Assemblieren die branch delay slots mit nops füllt.

8 DLX-Befehle

Im folgenden steht r◊ für ein Integer-Register, also z.B. r3, f◊ für ein Float-Register, also z.B. f3, und □ für einen Immediate-Wert, das kann eine Zahl in der Form #xxx sein oder auch ein Label, dessen relative Adresse dann als Immediate-Wert eingesetzt wird.

8.1 Datentransfer-Befehle

LB[U]	r◊, □(r◊)	Load Byte [Unsigned]
LH[U]	r◊, □(r◊)	Load Half word [Unsigned]
LW	r◊, □(r◊)	Load Word
LF	f◊, □(r◊)	Load Float
LD	f◊, □(r◊)	Load Double
SB	□(r◊), r◊	Store Byte
SH	□(r◊), r◊	Store Half word
SW	□(r◊), r◊	Store Word
SF	□(r◊), f◊	Store Float
SD	□(r◊), f◊	Store Double
MOVI2S		Move Integer To Special
MOVS2I		Move Special To Integer
MOVF	f◊, f◊	Move Float
MOVD	f◊, f◊	Move Double
MOVFP2I	r◊, f◊	Move Float To Integer
MOVI2FP	f◊, r◊	Move Integer To Float

Die Befehle, die mit L beginnen laden ein Byte, ein Halbword (2 Byte), ein Wort (4 Byte), ein Float (4 Byte Gleitkommazahl) oder ein Double (8 Byte Gleitkommazahl) aus dem Speicher in ein Register.

Die Speicher-Adresse setzt sich dabei aus einem Immediate-Wert plus einem Register zusammen, die beiden werden einfach addiert. Oft ist der Immediate-Wert die Beginn-Adresse eines Arrays, angegeben durch dessen Label, und der Registerinhalt fungiert als Index im Array. Daher auch die Schreibweise mit der Klammer. Oft macht man es aber genau umgekehrt, wenn das Register die Adresse einer Struktur (oder eines Klassenobjekts) enthält und der Immediate-Wert das jeweilige Element der Struktur auswählt.

Bei LB und LH gibt es noch zwei Varianten, nämlich signed und unsigned. Da der geladene Wert kürzer ist als ein Register, müssen die führenden Bits mit etwas befüllt werden. Bei unsigned wird mit Nullen befüllt, bei signed mit dem Vorzeichenbit des geladenen Werts. So hat dann das Register als ganzes auch bei negativen Werten den korrekten Inhalt.

Die Befehle, die mit S beginnen, machen das gleiche in die andere Richtung, sie speichern also Registerinhalte im Speicher. Die Befehle, die mit M beginnen, bewegen Registerinhalte von einem in ein anderes Register, ohne den Inhalt zu verändern.

8.2 Arithmetisch-logische Befehle

ADD[U][I]	r◊, r◊, r◊/□	Add [Unsigned] [Immediate]
SUB[U][I]	r◊, r◊, r◊/□	Subtract [Unsigned] [Immediate]
MULT[U]	r◊, r◊, r◊	Multiply [Unsigned]
DIV[U]	r◊, r◊, r◊	Divide [Unsigned]
AND[I]	r◊, r◊, r◊/□	Logical And [Immediate]
OR[I]	r◊, r◊, r◊/□	Logical Or [Immediate]
XOR[I]	r◊, r◊, r◊/□	Logical Xor [Immediate]
LHI	r◊, □	Load High Immediate
SLL[I]	r◊, r◊, r◊/□	Shift Left Logical
SRL[I]	r◊, r◊, r◊/□	Shift Right Logical
SRA[I]	r◊, r◊, r◊/□	Shift Right Arithmetic
S{EQ,NE}[U][I]	r◊, r◊, r◊/□	Set {Equal, Not Equal} [Uns.] [Imm.]
S{GT,LT}[U][I]	r◊, r◊, r◊/□	Set {Greater, Less} Than [Uns.] [Imm.]
S{GE,LE}[U][I]	r◊, r◊, r◊/□	Set {Greater, Less} or Equal [Uns.] [Imm.]

Die meisten dieser Befehle akzeptieren als zweiten Operanden entweder ein Register oder einen Immediate-Wert. In letzterem Fall muss an den Befehl ein I angehängt werden, also z.B. ADDI.

MULT und DIV sind eigentlich keine gültigen Befehle, stattdessen müssten MULTI und DIVI (die ansonsten eigentlich MULT und DIV heißen) verwendet werden, die allerdings zwar mit Integer-Werten arbeiten, diese aber in Gleitkomma-Registern erwarten. Die Integer-Befehle MULT und DIV werden daher beim Assemblieren so übersetzt, dass die Quellregister (z.B. r5) mit MOVI2FP in ein entsprechendes Gleitkomma-Register kopiert werden (z.B. f5), dann MULTI oder DIVI eingesetzt wird, und anschließend das Ergebnis mit MOVFP2I wieder in ein Integer-Register zurückkopiert wird.

Der LHI-Befehl wird benötigt, wenn man 32-Bit-Immediate-Werte (z.B. Adressen) in ein Register laden möchte. Mit "ADDI r1,r0,#xxx" geht das nicht, weil Immediate-Werte weniger als 32 Bit beinhalten. LHI lädt 16 Bit in die obere Hälfte des Registers. Die untere Hälfte kann dann mit ADDI oder ORI ergänzt werden.

Für die Shift-Right-Befehle gibt es die Varianten logical und arithmetic. In letzterem Fall werden die frei werdenden führenden Bits mit dem bisher führenden Bit aufgefüllt, das entspricht dann einer Signed-Division durch eine 2er-Potenz, bei logical werden sie mit 0 aufgefüllt.

Die Set-Befehle führen den entsprechenden Vergleich der zwei Quell-Operanden durch und setzen das Ziel-Register auf einen Wert ungleich 0 (meist 1), wenn der Vergleich wahr ist. Dieser Ergebniswert wird dann meistens in Branch-Befehlen weiter verwendet.

8.3 Sprungbefehle

BEQZ	r◊, □	Branch on Equal Zero
BNEZ	r◊, □	Branch on Not Equal Zero
BFPT	□	Branch on Float True
BFPF	□	Branch on Float False
J	□	Jump
JR	r◊	Jump Register
JAL	□	Jump And Link
JALR	r◊	Jump And Link Register
TRAP	□	Trap
RFE		Return From Exception

Die Branch-Befehle springen an die Adresse, die als relative Adresse im Immediate-Wert angegeben wird, falls das angegebene Register (nicht) null enthält, bzw. wenn ein Gleitkomma-Vergleich wahr (falsch) ergeben hat. Ansonsten fährt der Prozessor mit dem nächsten Befehl fort.

Die Jump-Befehle springen dagegen ohne Bedingung. Die Zieladresse kann hier auch als absolute Adresse in einem Register stehen. Bei "And Link" wird die Adresse des nächsten Befehls in das Register r31 geschrieben. So kann man später mit "JR r31" zurückspringen und mit dem ursprünglichen Code fortfahren. Dieser Vorgang entspricht einem Funktionsaufruf inklusive "return".

TRAP löst eine Ausnahmesituation aus. Am häufigsten wird "TRAP #0" verwendet, um den Simulator anzuhalten.

8.4 Gleitkomma-Befehle

ADD{F,D}	f◊, f◊, f◊	Add {Float, Double}
SUB{F,D}	f◊, f◊, f◊	Subtract {Float, Double}
MULT{F,D,[U]I}	f◊, f◊, f◊	Multiply {Float, Double, Integer}
DIV{F,D,[U]I}	f◊, f◊, f◊	Divide {Float, Double, Integer}
CVT{F,D,I}2{F,D,I}	f◊, f◊	Convert {Float, Double, Integer} to {...}
{EQ,NE}{F,D}	f◊, f◊	{Equal, Not Equal} {Float, Double}
{GT,LT}{F,D}	f◊, f◊	{Greater, Less} Than {Float, Double}
{GE,LE}{F,D}	f◊, f◊	{Greater, Less} or Equal {Float, Double}

Diese Befehle arbeiten mit Float-Zahlen in Gleitkomma-Registern bzw. mit Double-Zahlen, die zwei Gleitkomma-Register überspannen können. Bei Double-Befehlen muss als Register immer ein geradzahliges (f0, f2, f4, ...) angegeben werden.

Für Integer-Befehle muss ein Integer-Wert in einem Gleitkomma-Register stehen, also z.B. vorher mit MOVI2FP aus einem Integer-Register kopiert worden sein.

Die Vergleichs-Befehle haben im Gegensatz zu den äquivalenten Integer-Befehlen kein Zielregister. Das Vergleichsergebnis steht in einem eigenen Spezialregister, wird bei jedem Vergleich überschrieben und kann nur mit BFPT und BFPF abgefragt werden.

9 DLX unter Linux/Unix

WinDLX gibt es – wie der Name schon sagt – nur unter Windows. Für Linux/Unix gibt es ein kommandozeilenorientieres Tool namens *dlxsim*, das im Web frei verfügbar und leicht zu ergooglen ist. Es kommt meist als *dlx.tar.gz*. Einfach entpacken und dann "cd dlx/dlxsim; make" sagen. Die Software simuliert leider das Pipelining nicht so schön wie WinDLX. Es gibt auch eine Dokumentation dazu. Diese kann erzeugt werden mit "cd dlx/man; latex report; dvips report; gv report.ps".

Die Bedienung ist etwas gewöhnungsbedürftig, nach einer Weile ist man damit aber sogar schneller als mit WinDLX, weil man nicht ständig blöd herumklicksen muss. Zuerst startet man dlxsim mittels "dlxsim". Dann lädt man das Programm mittels "load programm.s". Mit "go" wird das Programm gestartet und läuft bis zum "trap 0" durch. Mit "step" tracet man das Programm schrittweise durch. Mit dem "get" Befehl kann man alles mögliche anzeigen: Der erste Parameter bestimmt, *was* angezeigt wird, der zweite Parameter *wie* es angezeigt wird. So zeigt z.B. "get r5 d" den Inhalt des Registers r5 als Dezimalzahl, "get name 10c" die 10 Bytes an der Speicherstelle "name" und "get start 8i" die 8 Instruktionen ab dem Programm-Label "start". Mit dem Befehl "stop" können auch Breakpoints gesetzt werden. Aber das ist in der Dokumentation (s.o.) alles viel genauer beschrieben.

Ach ja: "go" und "step" starten immer vom aktuellen program counter weg. Am Anfang ist dieser auf 0 gesetzt. Das Programm beginnt (üblicherweise) aber bei Speicherstelle 256. Wenn man also nur "go" sagt, führt der Prozessor zuerst die Instruktionen zwischen 0 und 256 aus. Da dieser Bereich (üblicherweise) mit Nullen gefüllt ist, werden dabei nur `nop`-Befehle ausgeführt. Trotzdem sollte man bei "go" oder dem ersten "step" besser die Startadresse angeben. ZB so: "go start".

Einen wichtigen Unterschied zwischen dlxsim und WinDLX gibt es noch: In dlxsim gibt es den Befehl "mult" für Integer-Register nicht. Diese müssen zuerst mit "movi2fp" in Float-Register kopiert werden, dort mit "mult" multipliziert und mit "movfp2i" wieder zurück kopiert werden. WinDLX macht das im Hintergrund eigentlich genauso.

Noch einen Unterschied gibt es: dlxsim verwendet einen branch delay slot. Daher immer einen `nop`-Befehl nach jedem Branch und jedem Jump einbauen. Oder eben den branch delay slot zur Performance-Optimierung verwenden. Dann läuft das Programm aber wahrscheinlich in WinDLX nicht mehr.