Das zugrundeliegende Programm:

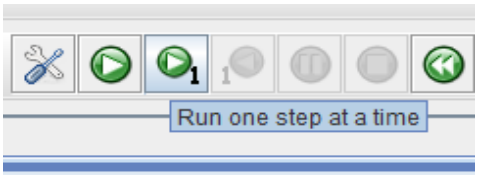
Programm zur Umwandlung einer vorzeichenbehafteten Ganzzahl in das IEEE 754 Format

Wie im HowTo beschrieben, wird *Simulation16.asm* im MARS geöffnet. Das Programm wandelt eine vorzeichenbehaftete 32-Bit-Integer in das IEEE 754 Format, legt das Ergebnis an der Speicherstelle *dst* ab und gibt es zusätzlich aus.

```
.data
        src:        .word 0xFFFFFF85
        dst:        .word 0xDEADBEEF
        mask1:      .word 0x80000000
        mask2:      .word 0xFFFFFFFF
        mask3:      .word 0x7FFFFF

        message1: .asciiz "\nDie 32-Bit-IEEE-754-Darstellung ist binär: "
        message2: .asciiz " und hexadezimal: "
```

Im *.data* Teil des Codes werden die zu wandelnde Zahl *src=0xFFFFFF85*, einige benötigte Masken und 2 Strings, die die Ausgabe begleiten sollen, hinterlegt. Zusätzlich wird die Speicherstelle *dst* erschaffen.

Empfehlenswert ist es, nach dem Assemblieren Schritt für Schritt durch das Programm zu gehen, das geschieht durch Klicks auf den „*Run one step at a time*“ Button:



Der *.text* Teil beginnt damit, die hinterlegten Werte in die Register zu laden:

```
.text
# Laden der zu wandelnden Zahl und der Masken in die Register
        lw $t1, src
        lw $t2, mask1
        lw $t3, mask2
        lw $t4, mask3
```

Nach Ausführen dieses Codefragments liegen *src* und die Masken 1-3 in den Registern *$t1-$t4* vor:

$t1	9	0xffffff85
$t2	10	0x80000000
$t3	11	0xffffffff
$t4	12	0x007fffff
$t5	13	0x00000000

Als nächstes wird der Wert in *src* mit der Maske *mask1* bitweise UND-verknüpft. Das entspricht einem Vorzeichentest: Wenn in *src* ein positiver Wert liegt, ist das erste Bit, also das Vorzeichenbit, eine *0*, wenn die Zahl negativ ist, eine *1*. In *mask1* liegt eine *1* an erster Stelle, die UND-Verknüpfung speichert also das Vorzeichenbit unserer zu wandelnden Zahl in *$t5*:

```
# Bitweise UND-Verknüpfung mit Maske 1, Vorzeichentest, wenn Zahl positiv ist, springe zu m1
        and $t5, $t1 , $t2
        beqz $t5, m1
```

Wenn in *$t5* nun eine *0* liegt, ist *src* positiv und es wird zu Marke *m1* gesprungen mit dem *beqz: **b**ranch if **eq**ual **z**ero* Befehl:

```
# Bitweise UND-Verknüpfung mit Maske 1, Vorzeichentest, wenn Zahl positiv ist, springe zu m1
and $t5, $t1 , $t2
beqz $t5, m1

# Di  beqz $t1,label   Branch if EQual Zero : Branch to statement at label if $t1 is equal to zero egativ ist:
```

In unserem Fall (*src=0xFFFFFF85)* gilt nun *$t5=80000000* und entsprechend wird der Sprung nach *m1* nicht ausgeführt, sondern mit den folgenden Codezeilen fortgefahren:

```
# Die beiden folgenden Zeilen werden nur ausgeführt, wenn die Ursprungszahl negativ ist:
# Bitweise Negation durch die Verwendung von Maske 2, danach Addition von 1,
# berechnet also das Zweierkomplement
        xor $t1, $t1, $t3
        addi $t1, $t1, 0x1
```

Die XOR-Verknüpfung der Zahl *src* in *$t1* mit der Maske *mask2* in $t3 entspricht der bitweisen Negation. Die anschließende Addition von 1 vollendet die Berechnung des Zweierkomplements der Zahl in *$t1*, wo nun also Folgendes steht:

In Marke *m1* wird die Maske *mask1* so verschoben, dass sie nun das Bit *30* (statt *31*) extrahiert, in *$t6* wird ein Zähler eingerichtet und mit *0* initialisiert:

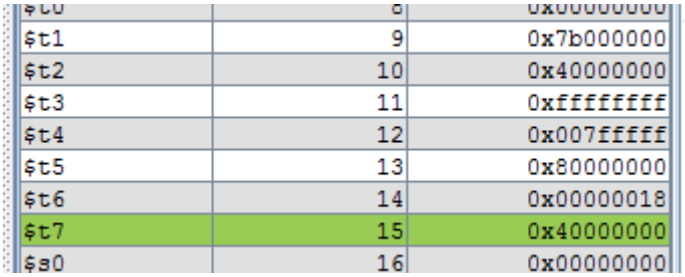
```
m1:     srl $t2, $t2, 1
        and $t6, $zero, $zero
```

Es folgt die Schleife *m2*, in der der Inhalt von *$t1* (also unsere in das Zweierkomplement gewandelte Zahl *src*) so lange links geschoben wird, bis an Bit *30* eine *1* vorliegt. Die Anzahl dieser Verschiebungen wird im Register *$t6* gezählt.

Sobald ein 1-Bit an Stelle 30 entdeckt wird, wird zur Marke m3 gesprungen.

```
m2:     and $t7, $t1, $t2
        bnez $t7, m3
        addi $t6, $t6, 1
        sll $t1, $t1, 1
        j m2
```

Das ist hier der Fall:



$t0	8	0x00000000
$t1	9	0x7b000000
$t2	10	0x40000000
$t3	11	0xffffffff
$t4	12	0x007fffff
$t5	13	0x80000000
$t6	14	0x00000018
$t7	15	0x40000000
$s0	16	0x00000000

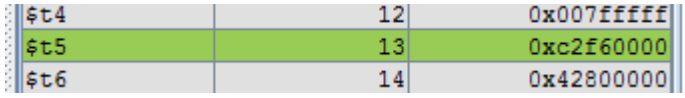
In *$t1* wurde geschoben, bis an Stelle *30* ein 1-Bit steht, die Anzahl der Verschiebungen (18_{16} = 24_{10}) steht in *$t6*. Im m3 werden nun erst Charakteristik, dann Mantisse berechnet:

```
m3:     # Berechnung der Charakteristik und Verknüpfung mit $t5
        addi $t7, $zero, 30
        sub $t6, $t7, $t6
        addi $t6, $t6, 127
        sll $t6, $t6, 0x17
        or $t5, $t5, $t6
```

Hier wird der ermittelte Wert in $t6 von 30 subtrahiert, dann 127 addiert, um die Charakteristik zu erzeugen. Die wird dann an die korrekte Stelle geschoben und mit *$t5* kombiniert, das schon das Vorzeichen enthält.

Für die Berechnung der Mantisse werden *$t1* und *$t4* UND-verknüpft, sodass die unteren *23* Bits erhalten bleiben, und die oberen auf *0* gesetzt werden, dann erfolgt die ODER-Verknüpfung mit $t5, wo dann unser Ergebnis steht:

```
# Berechnung der Mantisse und Verknüpfung mit $t5
        srl $t1, $t1, 0x7
        and $t1, $t1, $t4
        or $t5, $t5, $t1
```



$t4	12	0x007fffff
$t5	13	0xc2f60000
$t6	14	0x42800000

Die Berechnung ist abgeschlossen, es folgt die Ausgabe. Zunächst wird der String *message1* ausgegeben:

```
# Ausgabe des Strings message1
        la $a0, message1
        li $v0, 4
        syscall
```

Der Wert *4* für den *syscall* bedeutet *print string*, und die Adresse dieses Strings muss dafür in Register *$a0* geladen werden. Die Ausgabe des Ergebnisses in Binärdarstellung erfolgt durch folgende Codezeilen:

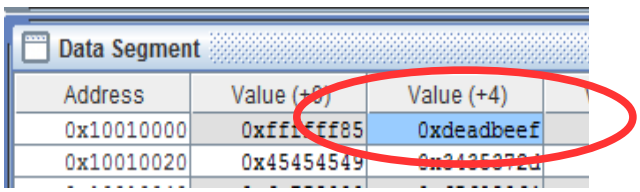
```
# Ausgabe des Ergebnisses binär
        move $a0, $t5
        li $v0, 35
        syscall
```

Das Ergebnis in *$t5* wird in *$a0* geschrieben, dann der Wert *35* (*print integer in binary*) in *$v0* geladen, der *syscall* gibt die Zahl aus. Es folgt die Ausgabe des Strings *message2* und des Ergebnisses in Hexadezimaldarstellung, der Wert *34* für den *syscall* bedeutet *print integer in hexadecimal:*

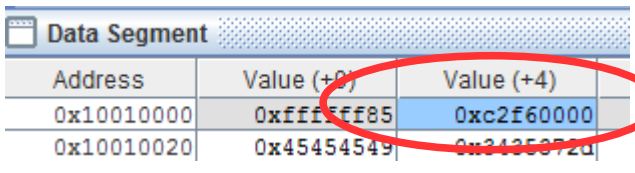
```
# Ausgabe des Strings message2
        la $a0, message2
        li $v0, 4
        syscall

# Ausgabe des Ergebnisses hexadezimal
        move $a0, $t5
        li $v0, 34
        syscall
```

Die ursprüngliche Version der Aufgabenstellung forderte statt einer Ausgabe, dass das Ergebnis an der Speicherstelle *dst* abgelegt werden sollte, dies erfolgt hier nun:

```
# Ablage des Ergebnisses an der Speicherstelle dst
        sw $t5, dst
```



vorher

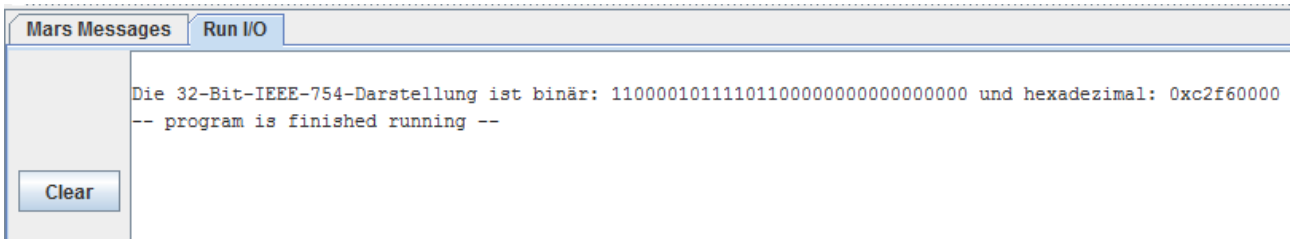


nachher

Dann ist das Programm nur noch zu beenden, was wie bei den anderen Simulationen dieser Reihe auch, über den Wert 10 für den syscall passiert:

```
# exit
        li $v0, 10          # der Wert 10 für den syscall bedeutet: exit (terminate execution)
        syscall
```

Die Ausgabe ist im Fenster *Run I/O* unterhalb des *Data Segments* im *execute* Fenster zu finden:



Dieses Programm war ursprünglich in DLX-Assembler verfasst und diente erst als Klausuraufgabe, später als Einsendeaufgabe. Es wurde übersetzt und leicht modifiziert, um dieser Reihe *Simulationen mit dem MARS Simulator* als Beispielaufgabe anzugehören.