

Das zugrundeliegende Programm:

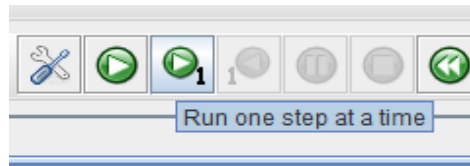
Multiplikation zweier Integer mit Eingabe der Zahlen und Ausgabe des Produkts als double

Wie im HowTo beschrieben, wird *Simulation11.asm* im MARS geöffnet. Es sollen zwei Integerzahlen eingegeben werden, deren Produkt dann als Gleitpunktkommazahl ausgegeben wird.

```
.data
    prompt1: .asciiz "\nBitte die erste Zahl eingeben, x = "
    prompt2: .asciiz "\nBitte die zweite Zahl eingeben, y = "
    message: .asciiz "\nDas Ergebnis der Multiplikation als double ist x * y = "
```

Im *.data* Teil des Codes werden Strings hinterlegt, die einerseits zur Eingabe der Summanden auffordern und andererseits die Ausgabe begleiten sollen.

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 mit der Aufforderung, die erste Zahl, also *x*, einzugeben:

```
.text
    # Ausgabe der ersten Nachricht: prompt1
    li $v0, 4 # der Wert 4 für den syscall bedeutet: print string
    la $a0, prompt1 # lädt die Adresse des ersten Strings in $a0
    syscall

    # erste Zahl einlesen und im temporären Register $t0 ablegen
    li $v0, 5 # der Wert 5 für den syscall bedeutet: read integer
    syscall
    move $t0, $v0
```

Der Wert 4 für den *syscall* bedeutet *print string*, und die Adresse dieses Strings muss dafür in Register *\$a0* geladen werden. Nach Ausgabe der Nachricht *prompt1* kann dann *x* eingelesen werden, dazu dient der Wert 5: *read integer* für den *syscall*. Wird dann eine Zahl eingegeben und mit *Enter* bestätigt, liegt sie in *\$v0* vor und wird hier, zur weiteren Verwendung, in das temporäre Register *\$t0* kopiert.

Ebenso wird für y verfahren, sodass nach Ausführen dieser Codezeilen x und y in den Registern $\$t0$ und $\$t1$ vorliegen und mit dem Multiplikationsbefehl fortgefahren werden kann.
 Hier zu sehen mit der beispielhaften Eingabe $x = 5$ und $y = 6$:

$\$t0$	8	0x00000005
$\$t1$	9	0x00000006
$\$t2$	10	0x00000000

Es folgt der Multiplikationsbefehl, der die unteren 32 Bit des Produkts der Inhalte der Register $\$t0$ und $\$t1$ in das Register $\$t2$ schreibt und die oberen 32 Bit im Spezialregister HI ablegt. Nach dem Multiplikationsbefehl wird dann der Inhalt von HI in das Register $\$t3$ geschrieben, hierzu nutzt man den Befehl *mfhi*:

```
# Multiplikation
mul $t2, $t0, $t1
mfhi $t3
# Transport des Integerprodukts zum Coproc1 und Konvertieren zu double
mtc1 mfhi $t1 Move from HI register : Set $t1 to contents of HI (see multiply and divide operations)
```

```
# Multiplikation
mul $t2, $t0, $t1
mfhi $t3
#Transport des Integerprodukts zum Coproc1 und Konvertieren zu double
mtc1.d $t2, $f12
cvt.d.w $f12, $f12
```

Dann wird das Produkt mit dem *mtc1.d* Befehl zum Coprozessor eins, der *Floating-Point Unit FPU*, transportiert und dort mithilfe des Befehls *cvt.d.w* zu double konvertiert.

```
# Transport des Integerprodukts zum Coproc1 und Konvertieren zu double
mtc1.d $t2, $f12
cvt.d.w $f12, $f12
mtc1.d $t1, $f2 Move To Coprocessor 1 Double : Set $f2 to contents of $t1, set next higher register from $f2 to contents of next higher register from $t1
cvt.d.w $f12, $f12
# Ausgabe
cvt.d.w $f2, $f1 Convert from word to double precision : Set $f2 to double precision equivalent of 32-bit integer value in $f1
li $v0, 4
```

Warum nun wird der Inhalt von HI in das Register $\$t3$ gerettet?

Der *Move to Coprocessor 1* Befehl schreibt hier nicht nur den Inhalt von $\$t2$ in $\$f12$, sondern auch den Inhalt des nächsthöheren, also $\$t3$ in das nächsthöhere FPU Register, hier also $\$f13$.

Das lässt sich wieder schön im *MARS* verfolgen, rechts in der Übersicht über die Register öffnet man durch Klick auf den Reiter *Coproc 1* die Ansicht der FPU Register:



Nach dem Verschieben des Produkts sieht das (hexadezimal) folgendermaßen aus:

\$f11	0x00000000	
\$f12	0x00000001e	0x0000000000000001e
\$f13	0x00000000	
\$f14	0x00000000	0x0000000000000000

Nach dem Konvertieren dann so:

\$f11	0x00000000	
\$f12	0x00000000	0x403e000000000000
\$f13	0x403e0000	
\$f14	0x00000000	0x0000000000000000

Nun muss noch das Ergebnis ausgegeben werden, die Ausgabe wird vom String *message* begleitet:

```
# Ausgabe der dritten Nachricht: message
li $v0, 4
la $a0, message
syscall
```

Wie schon bei der Ausgabe der Aufforderungen *prompt1* und *prompt2* steht hier der Wert 4 für den *syscall* für *print string* und die Adresse des auszugebenden Strings muss in *\$a0* geladen werden.

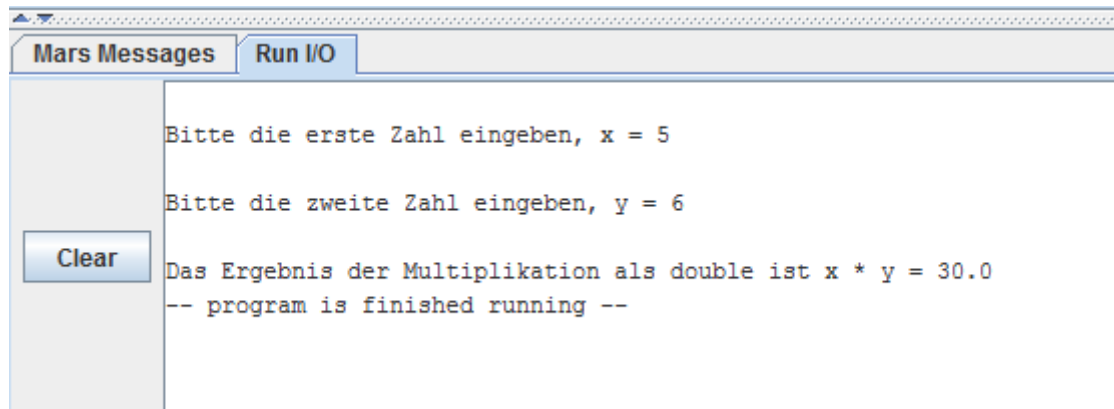
```
# Ausgabe des Produkts als double
li $v0, 3 # der Wert 3 für den syscall bedeutet: $f12 = double to print
syscall
```

Der Wert 3 für den *syscall* bedeutet, dass in *\$f12* ein double steht, der ausgegeben werden soll.

Nun bleibt nur noch das Beenden des Programms, was wie in den anderen Simulationen dieser Reihe *Simulationen mit dem MARS Simulator* auch, über den Wert 10: *terminate execution* 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:



Abschließend hierzu ist anzumerken, dass der Code in dieser Form nur für die Multiplikation von Integers geeignet ist, deren Produkt maximal eine 32-Bit Zahl ist. Zwar sind im *MIPS* 64-Bit Gleitkommaoperationen möglich, aber das Vorgehen wie hier, wo der Befehl *cvt.d.w*: *convert integer to double* genutzt wird, funktioniert bei Zahlen, die länger als 32 Bit sind nicht. Der *cvt.d.w* konvertiert 32-Bit Zahlen zu double, die Codezeile *cvt.d.w \$f12, \$f12* beachtet also nur die 32 Bit, die in *\$f12* liegen. Für 64-Bit-Zahlen jedoch verwendet *MIPS* 2 Register, der *cvt* Befehl müsste hier *\$f13* ebenfalls berücksichtigen, kann er aber nicht, sodass die Konvertierung von Zahlen, die länger als 32 Bit sind, nicht korrekt ausgeführt wird. Um das Problem zu lösen, könnte man die Faktoren erst wandeln und dann multiplizieren, statt umgekehrt, aber die Aufgabenstellung war eben anders.

Ich möchte Sie nachdrücklich auffordern, dies zu testen, ein wenig damit herumzuspielen, man begreift so viel eher, was da tatsächlich passiert.