Das zugrundeliegende Programm:

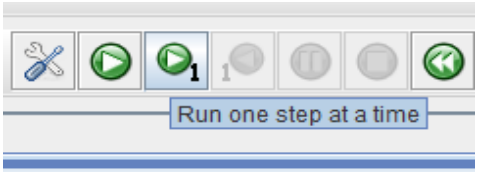
Programm zur Berechnung der ersten n Fibonacci-Zahlen

Wie im HowTo beschrieben, wird *Simulation15.asm* im MARS geöffnet. Das Programm liest eine Ganzzahl *n* ein und berechnet die ersten n Fibonacci-Zahlen.

```
.data
        fibs:    .word 0:25          #'array' für die berechneten Fibonaccizahlen
        size:    .word 25            # Größe dieses 'arrays'
        prompt:  .asciiz "\nBitte geben Sie eine Zahl (n <= 25) an, n = "
```

Im *.data* Teil des Codes wird Platz geschaffen für die maximal 25 Zahlen, die abgelegt werden müssen, sowie ein String hinterlegt, der die Ausgabe begleiten soll.

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, diese hinterlegten Werte in die Register zu laden:

```
.text
main:
        la $s0, fibs                # Adresse des 'arrays' laden
        la $s5, size                # Adresse der size Variable laden
        lw $s5, 0($s5)              # Arraygröße laden
```

Dann wird n eingelesen:

```
input:
# Einlesen und Überprüfen von n
        la $a0, prompt              # Adresse des Strings laden
        li $v0, 4                   # der Wert 4 für den syscall bedeutet print string
        syscall                     # String ausgeben
        li $v0, 5                   # der Wert 5 für den syscall bedeutet: read integer
        syscall                     # n einlesen, gelesener Wert steht in $v0
```

Der Wert *4* für den *syscall* bedeutet *print strin*g, und die Adresse dieses Strings muss dafür in Register *$a0* geladen werden. Nach Ausgabe der Nachricht *prompt* kann dann *n* eingelesen werden, dazu dient der Wert *5: read integer* für den *syscall.* Wird dann eine Zahl eingegeben und mit *Enter* bestätigt, liegt sie in *$v0* vor und kann wie folgt überprüft werden:

```
        bgt $v0, $s5, input         # Falls der eingegebene Wert größer als 25 ist, neu einlesen
        blt $v0, $zero, input       # Falls der eingegebene Wert kleiner Null ist, neu einlesen
```

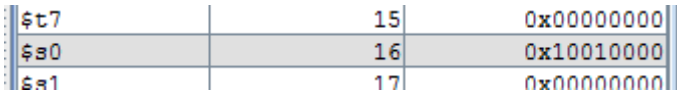
Gilt $0 \leq n \leq 25$, dann wird fortgefahren, ansonsten erneut zur Eingabe einer gültigen Zahl aufgefordert durch Sprung zur Marke *input*. Ein korrekt eingegebener Wert wird dann mithilfe des Additionsbefehls und der konstanten *0* im Register *$zero* in das Register *$s5* geschrieben:

```
add $s5, $zero, $v0# korrekt eingelesenen Wert in $s5 speichern
```

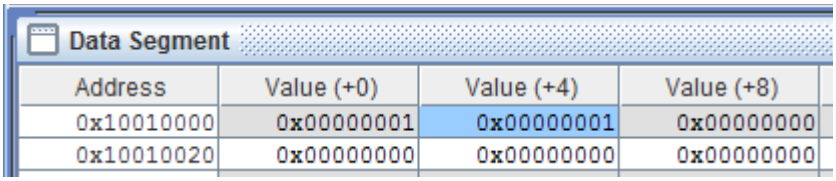
Die ersten beiden Fibonacci-Zahlen sind 1, deshalb können die direkt in das 'array' dessen Startadresse in $s0 liegt, geschrieben werden:

```
# Da die ersten beiden Fibonacci-Zahlen 1 sind, wird 1 direkt gespeichert
        li $s2, 1
        sw $s2, 0($s0)              # F[0]=1
        sw $s2, 4($s0)              # F[1]=1
        addi $s1, $s5, -2           # dient als Counter für die Schleife, läuft n-2 mal
```

Das kann man schön im *Text Segment* des *Execute* Fensters nachverfolgen. Zuerst schauen wir rechts in der Register-Übersicht, welchen Wert *$s0* enthält:



Unser 'array' *fibs* beginnt also bei der Adresse *0x10010000* und nach Ausführen der Befehle *sw $s2, 0($s0)* und *sw $s2, 4($s0)* finden wir dort 1en vor:



Folgende Schleife *loop* wird solange wiederholt, bis der Counter in *$s1* 0 erreicht hat:

```
loop:
# Schleife zur Berechnung der Fibonaccizahlen
        lw $s3, 0($s0)              # Wert aus F[x-2] holen
        lw $s4, 4($s0)              # Wert aus F[x-1] holen
        add $s2, $s3, $s4           # F[x] = F[x-2] + F[x-1]
        sw $s2, 8($s0)              # F[x] speichern
        addi $s0, $s0, 4            # Inkrementieren der Adresse
        addi $s1, $s1, -1           # Dekrementieren des Counters
        bgtz $s1, loop              # wiederholen der Schleife loop, solange $s1>0
```

Danach liegen die ersten *n* (hier im Beispiel gilt *n=10*) Fibonacci-Zahlen vor ….

Data Segment

Address	Value (+0)	Value (+4)	Value (+8)	Value (+c)	Value (+10)	Value (+14)	Value (+18)	Value (+1c)
0x10010000	0x00000001	0x00000001	0x00000002	0x00000003	0x00000005	0x00000008	0x0000000d	0x00000015
0x10010020	0x00000022	0x00000037	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000
0x10010040	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000

… und es wird mit folgendem Codefragment fortgefahren:

```
# Nach der Schleife liegen die Fibonacci-Zahlen vor und müssen noch ausgegeben werden
        la $a0, fibs            # Erster Parameter für die print Funktion ist das gefüllte array
        add $a1, $zero, $s5     # Zweiter Parameter ist n (gespeichert in $s5)
        jal print               # Aufruf des Unterprogramms print
```

Es wird das Unterprogramm *print* aufgerufen, dem über die Register *$a0* und $a1 die berechneten Zahlen sowie *n* übergeben werden. Im Unterprogramm *print* findet sich wieder ein *.data* und ein *.text* Teil, im .data Teil werden Strings hinterlegt, die zur Ausgabe benötigt werden:

```
.data
        space:          .asciiz " "# Leerstelle, die zwischen den Zahlen eingefügt wird
        message:        .asciiz "\nDie Fibonacci-Zahlen sind: \n"
```

Im *.text* Teil werden Die Startadresse des 'arrays' für die Ausgabe, sowie *n* als counter geladen, dann der String *message* ausgegeben:

```
.text
print:
        add $t0, $zero, $a0     # Startadresse des arrays für Ausgabe
        add $t1, $zero, $a1     # counter
        la $a0, message         # Adresse des Strings 'message' laden
        li $v0, 4               # Der Wert 4 für den syscall bedeutet print string
        syscall                 # Ausgabe des Strings
```

Danach wird die Schleife *loop2* durchlaufen, die der Reihe nach die berechneten Zahlen, getrennt durch eine Leerstelle, ausgibt:

```
loop2:
# Schleife zur Ausgabe der Zahlen
        lw $a0, 0($t0)          # Laden der aktuellen Fibonaccizahl
        li $v0, 1               # der Wert 1 für den syscall bedeutet: print integer
        syscall                 # Ausgabe der Zahl
        la $a0, space           # Laden der Adresse des Leerstellenstrings
        li $v0, 4               # Der Wert 4 für den syscall bedeutet: print string
        syscall                 # Leerstelle ausgeben

        addi $t0, $t0, 4        # Inkrementieren der Adresse der auszugebenden Daten
        addi $t1, $t1, -1       # Dekrementieren des counters
        bgtz $t1, loop2         # wiederholen der Schleife loop2, solange $t1>0
```

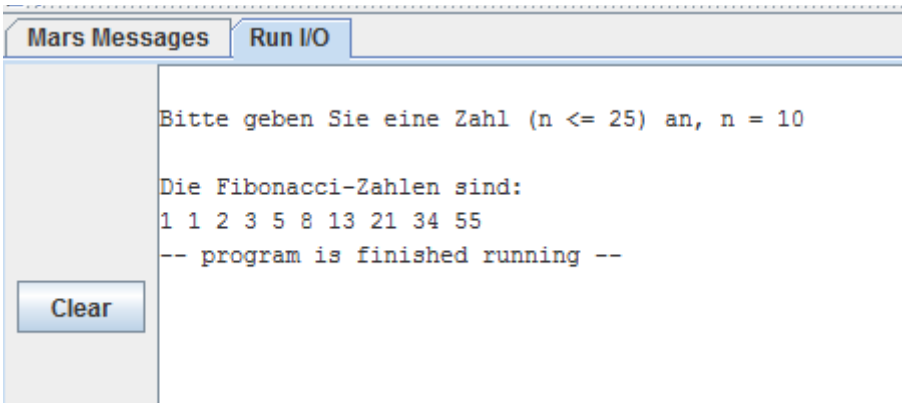
Diese Schleife läuft, bis der counter in *$t1* herunter gezählt ist, also alle *n* Fibonacci-Zahlen ausgegeben sind, dann erfolgt der Rücksprung zum Hauptprogramm *main*:

```
        jr $ra                  # Rücksprung zu main
```

Dort 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 ist eine leicht modifizierte Version der Datei *fibonacci.asm*, die von den Entwicklern des *MARS* auf dessen Homepage zum Download bereitgestellt wird.