Das zugrundeliegende Programm:

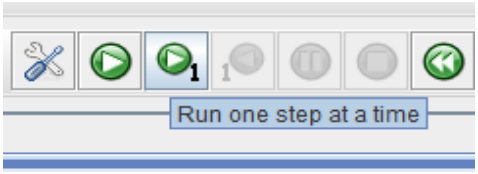
Programm zur Berechnung der Fakultät einer einzulesenden Ganzzahl n

Wie im HowTo beschrieben, wird *Simulation14.asm* im MARS geöffnet. Das Programm liest eine Ganzzahl *n* ein und berechnet rekursiv deren Fakultät *n!*. Zeigt den Gebrauch des Stacks.

```
.data
        prompt:   .asciiz "\nBitte geben Sie die Zahl ein, deren Fakultät berechnet werden soll: n = "
        message: .asciiz "\n Die Fakultät von n ist:\n n! = "
```

Im *.data* Teil des Codes werden Strings hinterlegt, die einerseits zur Eingabe der Ganzzahl 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 Zahl *n*, einzugeben:

```
.text
main:
# n einlesen und in $a0 ablegen
        la $a0, prompt                  # Laden der Adresse des Strings prompt
        li $v0, 4                       # Der Wert 4 für den syscall bedeutet print string
        syscall                         # Ausgabe des Strings prompt
        li $v0, 5                       # der Wert 5 für den syscall bedeutet read integer
        syscall                         # n in $v0 einlesen
        move $a0, $v0                   # n in $a0 ablegen
```

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

$v1	3	0x00000000
$a0	4	0x00000006
$a1	5	0x00000000

Wie man sieht, liegt hier als Beispiel *n = 6* in *$a0* vor.

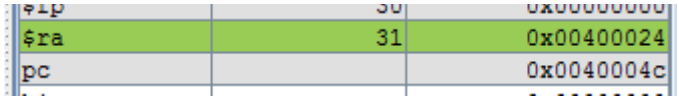
Da wir hier ein Programm vorliegen haben, das Rekursion nutzt, brauchen wir auch eine Abbruchbedingung für eben diese rekursiven Aufrufe:

```
# Wert für die Abbruchbedingung
        li $s1, 1
```

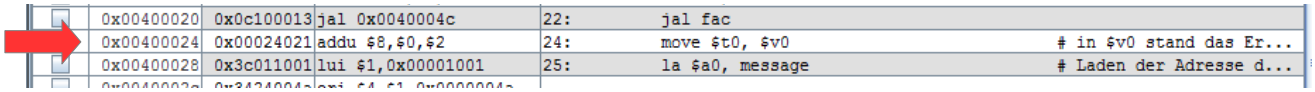
Danach kann das Unterprogramm *fac* aufgerufen werden, in dem die eigentliche Berechnung stattfindet:

```
# Aufruf des Unterprogramms fac
        jal fac
```

Der Aufruf des Unterprogramms geschieht wieder, wie wir das schon in Simulation 13 dieser Reihe gesehen haben, über den Befehl *jal: jump and link*, der *PC+4* im Register *$ra* speichert, damit das Unterprogramm eine Rücksprungadresse hat.



Beim ersten Aufruf von *fac*, aus *main* heraus, wird die Rücksprungadresse *$0x00400024* in *$ra* gespeichert, was genau dem Befehl entspricht, der nach *jal fac* im Hauptprogramm *main* steht, wie man schön im Text Segment des *Execute* Fenster sehen kann:

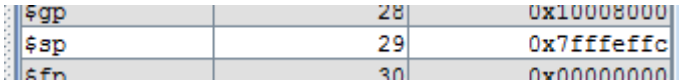


Was aber passiert mit den rekursiven Aufrufen von *fac* innerhalb von *fac*? Wenn jeder Aufruf mit *jal* die Rücksprungadresse in *$ra* sichert, dann werden die vorherigen überschrieben und das Programm kann nie zurückkehren in das Hauptprogramm, von dem aus es aufgerufen wurde. Was ist also zu tun? Die jeweilige Rücksprungadresse muss gesichert werden, bevor durch den erneuten (rekursiven) Aufruf die Adresse unwiederbringlich überschrieben und damit verloren ist. Ebenso muss der Inhalt des Registers $s0, in dem jeweils das Argument eingetragen wird, gesichert werden. Folgende Codezeilen realisieren diese Anforderungen, indem zuerst 8 Byte Speicherplatz auf dem Stack allokiert werden, um dann zwei 32-Bit-Worte dort zu sichern:

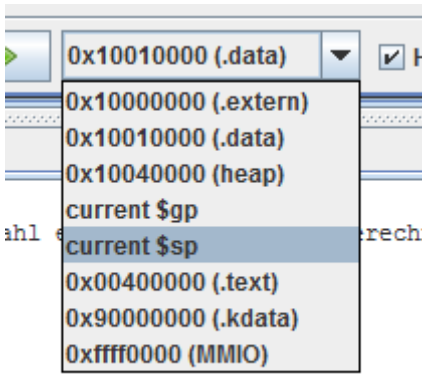
```
##################### Unterprogramm fac ######################
fac:
        addi $sp, $sp, -8               # Auf dem Stack Platz für 2 Einträge schaffen
        sw $s0, 0($sp)                  # Sichern des Registers $s0
        sw $ra, 4($sp)                  # Sichern der Rücksprungadresse
```

Im Folgenden zeigen Screenshots, wie das beim ersten Aufruf von *fac*, also aus dem *main* Programm heraus, aussieht:

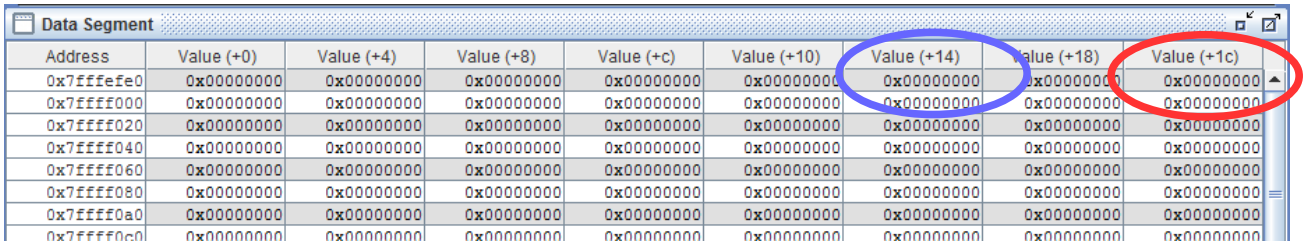
Nach dem Aufruf von *fac* steht im Register $sp = $0x7FFFEFFC, der Stackpointer zeigt also auf diese Adresse.



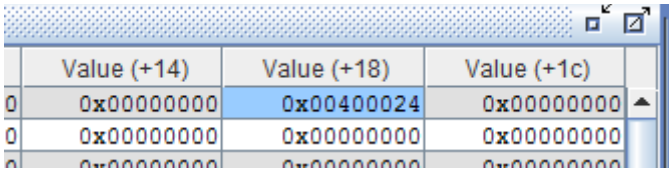
Im unteren Bereich des Data Segments des Execute Fensters kann man die Ansicht des Stacks mithilfe des Dropdown Menus auswählen:



Zu Beginn ist der Stack leer, rot markiert ist der Bereich, auf den der Stackpointer gerade zeigt:



Nach Ausführen des Befehls *addi $sp, $sp, -8* ist *$sp = 0x7FFFEFF4* zeigt also in obiger Abbildung auf die erste Zeile der dritten Spalte von rechts (blau markiert). Dort wird dann mit *sw $s0, 0($sp)* der Inhalt von *$s0* eingetragen (im Moment noch 0) und mit *sw $ra, 4($sp)* wird die Rücksprungadresse in der Zelle rechts davon eingetragen:



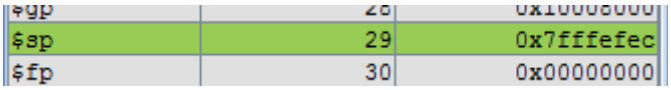
Dies ist eben genau die Rücksprungadresse aus *fac* heraus, zurück ins *main* Programm, wo wir den Befehl finden, der im Programm direkt hinter dem *jal fac* steht, wie wir das oben bereits in der Abbildung gesehen haben.

Nun wird das Argument in *$s0* geschrieben, auf die Abbruchbedingung geprüft und *$a0* dekrementiert, sodass daraufhin *fac* mit *n-1* aufgerufen werden kann:

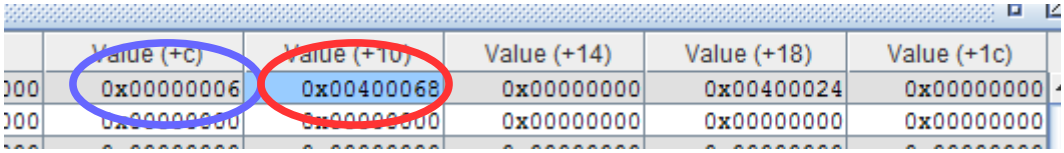
```
        add $s0,$zero,$a0       # lädt das Argument in $s0
        beq $s0, $s1, done      # wenn das Argument = 1 ist, springe zur Marke done, Abbruchbedingung
        addi $a0, $a0, -1       # dekrementiere das Argument (n-1)

# rekursiver Aufruf mit n-1
        jal fac
```

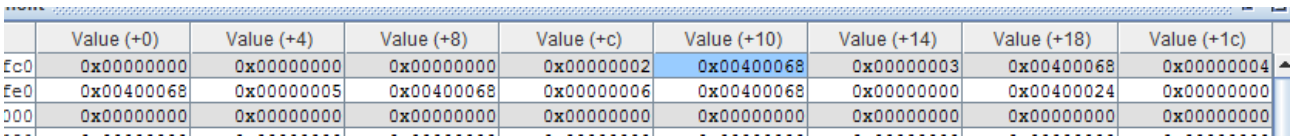
Exkurs: Was passiert derweil auf dem Stack?



$sp = 0x7FFFEFEC, wieder 2 Spalten weiter links in der Übersicht des Stacks im *Execute* Fenster. *sw $s0, 0($sp)* schreibt also den Inhalt von *$s0* in diese Zelle (blau) und *sw $ra, 4($sp)* sichert die Rücksprungadresse in die Zelle rechts daneben (rot):



Schön zu sehen: der Stack wächst (nach unten)! Mit jedem rekursiven Aufruf von *fac* wandert der Zeiger 2 Zellen nach links, trägt das Argument und die Rücksprungadresse ein, bis das Argument in *$a0=1* ist:



Damit liegen die Werte, die für die Berechnung von *6!* (in unserem Beispiel) benötigt werden, im Stack vor. *6*5*4*3*2*1=720*, wobei die *1* nicht auf den Stack gelegt werden muss, weil sie in der Marke *done* in das Rückgaberegister *$v0* geschrieben wird:

```
done:   # diese Zeile wird nur einmal ausgeführt, wenn fac mit a0=1 aufgerufen wurde
        li $v0, 1               # ist das Argument = 1, wird 1 zurückgegeben
```

Zurück zum Programmablauf:

Sobald *fac* mit *$a0=1* aufgerufen wurde, ist die Abbruchbedingung in Zeile 43 erreicht (denn dort ist dann *$s0=1*), sodass zur Marke *done* gesprungen wird, wo eine *1* als Rückgabewert von *fac* in das Register *$v0* geschrieben wird. Vom Stack zurückgelesen werden der zuletzt gesicherte *$s0* -Inhalt sowie die zuletzt gesicherte Rücksprungadresse, dann wird der Stackpointer wieder auf den Wert vor dem letzten Unterprogrammaufruf gesetzt (Zeiger wandert 2 Zellen nach rechts!)

```
end:
        lw $s0, 0($sp)                  # Zurücklesen des Registers $s0 vom Stack
        lw $ra, 4($sp)                  # Zurücklesen der Rücksprungsadresse vom Stack
        addi $sp, $sp, 8                # Stackpointer wieder auf den Wert vor Aufruf des Unterprogramms setzen
        jr $ra                          # Rücksprung

################## Ende des Unterprogramms fac ##################
```

Es erfolgt der Rücksprung (Zeile 61), woraufhin das Produkt der Inhalte von *$s0* und *$v0* berechnet und in *$v0* abgelegt wird, um dann wieder zu Zeile 57 in die Marke *end* zu springen. Das wiederholt sich solange, bis die Rücksprungadresse auf dem Stack erreicht ist, die aus *fac* hinaus wieder in das Hauptprogramm *main* führt.

Sobald der Rücksprung *fac* verlässt, wird das Programm an Zeile 24 fortgesetzt. Das Ergebnis der Berechnung steht in *$v0* und wird zusammen mit dem zugehörigen String *message* ausgegeben:

```
        move $t0, $v0                   # in $v0 stand das Ergebnis, ablegen in $t0
        la $a0, message                 # Laden der Adresse des Strings message
        li $v0, 4                       # der Wert 4 für den syscall bedeutet print string
        syscall                         # Ausgabe des Strings message
        move $a0, $t0                   # Laden des Ergebnisses in $a0
        li $v0, 1                       # der Wert 1 für den syscall bedeutet print integer
        syscall                         # Ausgabe des Ergebnisses
```

Danach bleibt nur noch, das Programm zu beenden, was wie in den anderen Codebeispielen auch, über den syscall mit dem Wert 10 (terminate execution) 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: