Das zugrundeliegende Programm:

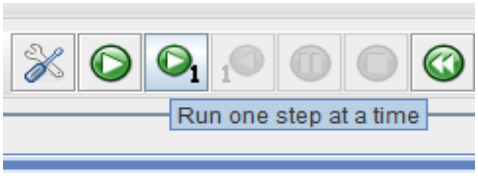
Programm zur Veranschaulichung von Speicherzugriffen und Nutzung des Stacks

Wie im HowTo beschrieben, wird *Simulation17.asm* im MARS geöffnet. Das Programm wandelt für ein Feld von Bytes jeweils das obere Nibble und anschließend das untere Nibble in die entsprechende ASCII-Darstellung um und speichert diese auf dem Stack.

```
.data
        in:             .byte 0x5d 0x18 0x2a 0x34 0x00
        mask:           .word 0xF
```

Im *.data* Teil des Codes werden die Bytes sowie eine Maske hinterlegt.

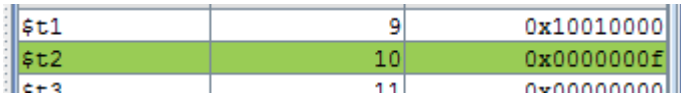
Empfehlenswert ist es, nach dem Assemblieren Schritt für Schritt durch das Programm zu gehen, das geschieht durch Klicks auf den „*Run one step at a time*“ Button:



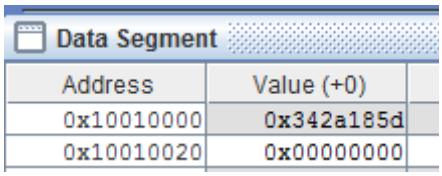
Der *.text* Teil beginnt damit, die Adresse des soeben erzeugten Byte-Arrays in das Register *$t1* und die Maske *mask* in das Register *$t2* zu schreiben:

```
.text
        la $t1, in              # lädt die Adresse des byte-Arrays in $t1
        lw $t2, mask            # lädt die Maske mask in $t2
```

Wichtig: In *$t1* liegt nun die Adresse von *in*, nicht die enthaltenen Bytes selbst, zu sehen an folgenden Screenshots:



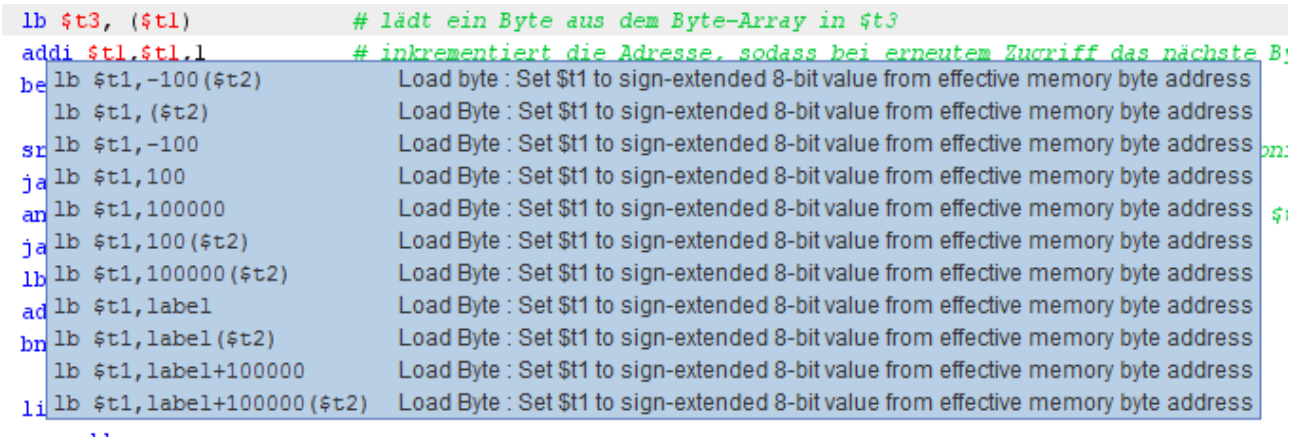
unter der in *$t1* gespeicherten Adresse *0x10010000* finden wir im Data Segment des *Execute* Fensters unsere Bytes:



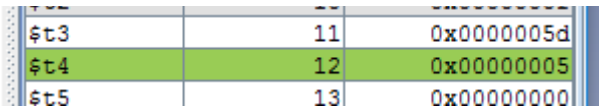
Dann wird das erste dieser Bytes in *$t3* geladen, die Zugriffsadresse inkrementiert und zu Marke *m2* gesprungen, falls das geladene Byte *0x00* ist, denn dann sind wir am Ende des Arrays angekommen:

```
        lb $t3, ($t1)          # lädt ein Byte aus dem Byte-Array in $t3
        addi $t1,$t1,1         # inkrementiert die Adresse, sodass bei erneutem Zugriff das nächste Byte geladen wird
        beqz $t3, m2           # Beendet das Programm, wenn das geladene Byte = 0 ist
```

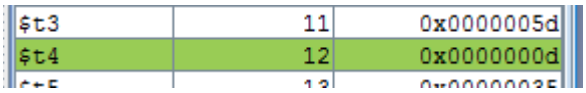
Wesentlich hier: der Befehl *lb $t3, ($t1)* lädt ein Byte des Eintrags, der unter der in *$t1* gespeicherten Adresse liegt, in diesem Fall also das Byte *0x5d*. Es wird in *$t3* geschrieben und dabei um das Vorzeichenbit erweitert.



Solange *$t3* nicht gleich *0* ist, wird nicht zu *m2* gesprungen, sondern mit Marke *m1* fortgefahren.
In *m1* wird zuerst das obere Nibble des geladenen Bytes in die Marke *m3* geschickt, dann das untere. Das obere wird ausgewählt, indem der Inhalt von *$t3* um *4* nach rechts geschoben und in *$t4* geschrieben wird:



Das untere wird ausgewählt, indem der Inhalt von $t3 mit der Maske in $t2 UND-verknüpft wird:




```
m1:     srl $t4, $t3, 0x4      # Schiebt den Inhalt von $t3 um 4 nach rechts und speichert das Ergebnis in $t4
        jal m3                  # Sprung zu m3 (Pc+4 wird in Register 31 = $ra gespeichert)
        and $t4, $t3, $t2      # bitweise UND-Verknüpfung von $t3 und der Maske in $t2, Ergebnis in $t4
        jal m3                  # Sprung zu m3 (PC+4 wird in Register 31 = $ra gespeichert)
        lb $t3, ($t1)           # Nächstes Byte aus dem Array laden
        addi $t1, $t1, 1       # Adresse inkrementieren
        bnez $t3, m1            # Sprung zu m1, falls das geladene Byte ungleich 0 ist
```

Die Marke *m3* addiert erst 30_{16} zum Nibble, denn ab hier sind in der ASCII-Tabelle die Zeichen für 0-9 Stelle 48_{10}-57_{10}) hinterlegt, speichert diese Summe in *$t5* und fragt dann ab, ob das Nibble > *10* ist. Falls ja, müssen erneut *7* addiert werden, um auf die Zeichen 'A' bis 'F' (Stelle 65_{10}-70_{10}) zu verweisen:

```
m3:     addi $t5,$t4,0x30       # 0x30 wird addiert, hier sind die ASCII-Zeichen ab '0' hinterlegt
        slti $t6,$t4,0xA        # falls das aktuelle Nibble > 10 ist, wird 0x7 addiert, um auf die Zeichen
        bnez $t6,m4              # 'A' bis 'F' zu verweisen, sonst weiter mit Marke m4
        addi $t5,$t5,0x7
```

Es folgt die Marke *m4*, in der der Wert, der in *m3* ermittelt und in *$t5* geschrieben wurde, auf dem Stack gesichert wird:

```
m4:     addi $sp, $sp, -4       # Prädekrement des Stackpointers
        sb $t5, ($sp)           # Sichern von $t5 auf dem Stack
        jr $ra                  # Rücksprung
```

Sind alle Bytes geladen und ihre ASCII-Darstellung auf dem Stack gespeichert, sieht das folgendermaßen aus:

Data Segment

Address	Value (+0)	Value (+4)	Value (+8)	Value (+c)	Value (+10)	Value (+14)	Value (+18)	Value (+1c)
0x7fffefc0	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000034
0x7fffefe0	0x00000033	0x00000041	0x00000032	0x00000038	0x00000031	0x00000044	0x00000035	0x00000000
0x7ffff000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000

Als Beispiel, hier rot markiert, die $32_{16} = 50_{10}$. In der ASCII-Tabelle steht an Stelle *50* eben die *2*, das obere Nibble des dritten Bytes *0x2a* unseres Arrays.

Was nun noch bleibt, ist wie bei den anderen Simulationen dieser Reihe auch, das Beenden des Programms durch den Wert *10 (terminate execution)* für den *syscall*. Erreicht wird dieses Codefragment durch den Sprung in die Marke *m2*, nachdem *0x00* in *$t3* geladen wurde:

```
m2:     li $v0, 10              # exit
        syscall
```

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