Das zugrundeliegende Programm:

Demonstration der Lade-/Speicher-Operationen lb, lbu, sb

Wie im HowTo beschrieben, wird *Simulation05.asm* im MARS geöffnet. Der Code zeigt die Verwendung der Lade- und Speicheroperationen:

- ***lb: load byte,***
- ***lbu: load byte unsigned und***
- ***sb: store byte***

Im Folgenden werden einzelne Codeabschnitte näher erläutert.

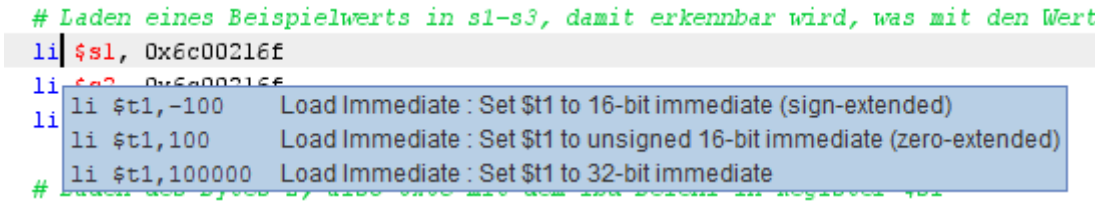
```
.data
        src:    .word 0xf78c4203
        lbu:    .asciiz " Nach dem Ladebefehl lbu $s1, 2($s0) steht in Register $s1 nun das mit 0en erweiterte Byte 2 des
                        Eingabewortes: \n"
        lb:     .asciiz "\n Nach dem Ladebefehl lb $s2, 2(s0) steht in Register $s2 nun das vorzeichenerweiterte Byte 2 des
                        Eingabewortes: \n"
        sb:     .asciiz "\n Mit dem Speicherbefehl sb $s3, 3(s0) wird das Byte 3 in $s0 durch das least significant Byte\n aus
                        $s3 ersetzt, die anderen Bytes aus $s3 wurden ignoriert: \n"
```

Die willkürlich gewählte Beispielzahl *src = 0xF78C 4203*, sowie einige für den Programmablauf zwar nicht wesentliche, aber für die Übersichtlichkeit der Ausgabe dienliche Strings werden hinterlegt.

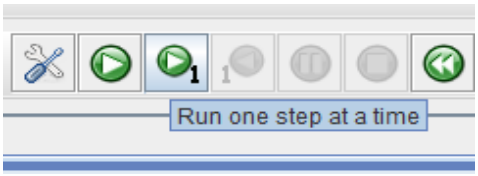
```
.text
        # Laden der Adresse von src in s0
        la $s0, src

        # Laden eines Beispielwerts in s1-s3, damit erkennbar wird, was mit den Werten in den Zielregistern passiert
        li $s1, 0x6c00216f
        li $s2, 0x6c00216f
        li $s3, 0x6c00216f
```

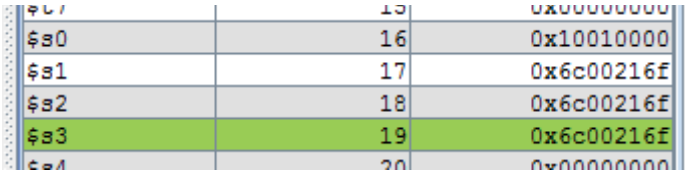
Nun wird die Adresse der Konstanten *src* in Register s0 geladen und dann mit dem *li* Befehl (load immediate) eine ebenfalls willkürlich gewählte Beispielzahl in die Register s1-s3 geschrieben. Hier wird der *li* Befehl benutzt, um seine Verwendung zu zeigen - die für die Register vorgesehene Beispielzahl hätte auch im .data Bereich schon hinterlegt werden können.



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

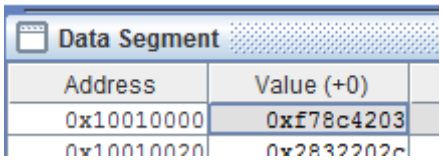


Im folgenden Bildausschnitt sieht man, dass in s0 nach Ausführung der obigen Befehle nun nicht der Wert *src*, sondern seine Adresse steht, und in den Registern s1-s3 liegt die Beispielzahl *$0x6C00 216F* bereit:



$s0	16	0x10010000
$s1	17	0x6c00216f
$s2	18	0x6c00216f
$s3	19	0x6c00216f

Im Data Segment auf der linken Seite des MARS findet man die Adresse, die in s0 steht, nämlich *$0x10010000,* sowie den Wert, der unter dieser Adresse zu finden ist, eben unsere Beispielzahl *src=$0xF78C 4203*:



Nun können die eigentlichen Operationen beginnen, zuerst wird ***lbu*** demonstriert:

```
# Laden des Bytes 2, also 0x8c mit dem lbu Befehl in Register $s1
        lbu $s1, 2($s0)
```

Es wird also das Byte 2 des Wertes, der unter der Adresse, die in s0 liegt, gefunden wird, in das Register s1 geladen:

```
# Laden des Bytes 2, also 0x8c mit dem lbu Befehl in Register $s1
lbu $s1, 2($s0)
```

lbu $t1,-100($t2)	Load byte unsigned : Set $t1 to zero-extended 8-bit value from effective memory byte address
lbu $t1,($t2)	Load Byte Unsigned : Set $t1 to zero-extended 8-bit value from effective memory byte address
lbu $t1,-100	Load Byte Unsigned : Set $t1 to zero-extended 8-bit value from effective memory byte address
lbu $t1,100	Load Byte Unsigned : Set $t1 to zero-extended 8-bit value from effective memory byte address
lbu $t1,100000	Load Byte Unsigned : Set $t1 to zero-extended 8-bit value from effective memory byte address
lbu $t1,100($t2)	Load Byte Unsigned : Set $t1 to zero-extended 8-bit value from effective memory byte address
lbu $t1,100000($t2)	Load Byte Unsigned : Set $t1 to zero-extended 8-bit value from effective memory byte address
lbu $t1,label	Load Byte Unsigned : Set $t1 to zero-extended 8-bit value from effective memory byte address
lbu $t1,label($t2)	Load Byte Unsigned : Set $t1 to zero-extended 8-bit value from effective memory byte address
lbu $t1,label+100000	Load Byte Unsigned : Set $t1 to zero-extended 8-bit value from effective memory byte address
lbu $t1,label+100000($t2)	Load Byte Unsigned : Set $t1 to zero-extended 8-bit value from effective memory byte address

In s0 liegt die Adresse unseres Beispielwertes *src = 0xF78C 4203,* das Byte 2 ist 0x8C. Was steht also nun nach Ausführung des *lbu* Befehls in Register s1?

$s0	16	0x10010000
$s1	17	0x0000008c
$s2	18	0x6c00216f

Das Byte *0x8C* wurde in s1 geschrieben und die übrigen Bits in s1 wurden mit 0en überschrieben, die 3 oberen Bytes des ursprünglichen Eintrags gehen also verloren.

Es folgt die Demonstration des ***lb*** – Befehls:

```
# Laden des Bytes 2, also 0x8c mit dem lb Befehl in Register $s2
        lb $s2, 2($s0)
```

Es wird also, ebenso wie bei ***lbu***, das adressierte Byte in das Zielregister geladen. Wo liegt nun der Unterschied zum ***lbu***?

```
# Laden des Bytes 2, also 0x8c mit dem lb Befehl in Register $s2
lb $s2, 2($s0)
```

lb $t1,-100($t2)	Load byte : Set $t1 to sign-extended 8-bit value from effective memory byte address
lb $t1,($t2)	Load Byte : Set $t1 to sign-extended 8-bit value from effective memory byte address
lb $t1,-100	Load Byte : Set $t1 to sign-extended 8-bit value from effective memory byte address
lb $t1,100	Load Byte : Set $t1 to sign-extended 8-bit value from effective memory byte address
lb $t1,100000	Load Byte : Set $t1 to sign-extended 8-bit value from effective memory byte address
lb $t1,100($t2)	Load Byte : Set $t1 to sign-extended 8-bit value from effective memory byte address
lb $t1,100000($t2)	Load Byte : Set $t1 to sign-extended 8-bit value from effective memory byte address
lb $t1,label	Load Byte : Set $t1 to sign-extended 8-bit value from effective memory byte address
lb $t1,label($t2)	Load Byte : Set $t1 to sign-extended 8-bit value from effective memory byte address
lb $t1,label+100000	Load Byte : Set $t1 to sign-extended 8-bit value from effective memory byte address
lb $t1,label+100000($t2)	Load Byte : Set $t1 to sign-extended 8-bit value from effective memory byte address

Die durch MARS bereitgestellten Erläuterungen zeigen den Unterschied schon auf: sign-extended bei ***lb*** und zero-extended bei ***lbu***. Der ***lb***-Befehl lädt also das gewählte Byte und füllt das Register nach oben mit dem Vorzeichenbit auf, während der ***lbu***-Befehl mit 0en auffüllt, unabhängig vom Vorzeichenbit des gewählten Bytes. Was steht nun als folgerichtig in s2 nach Ausführen des lb-Befehls?

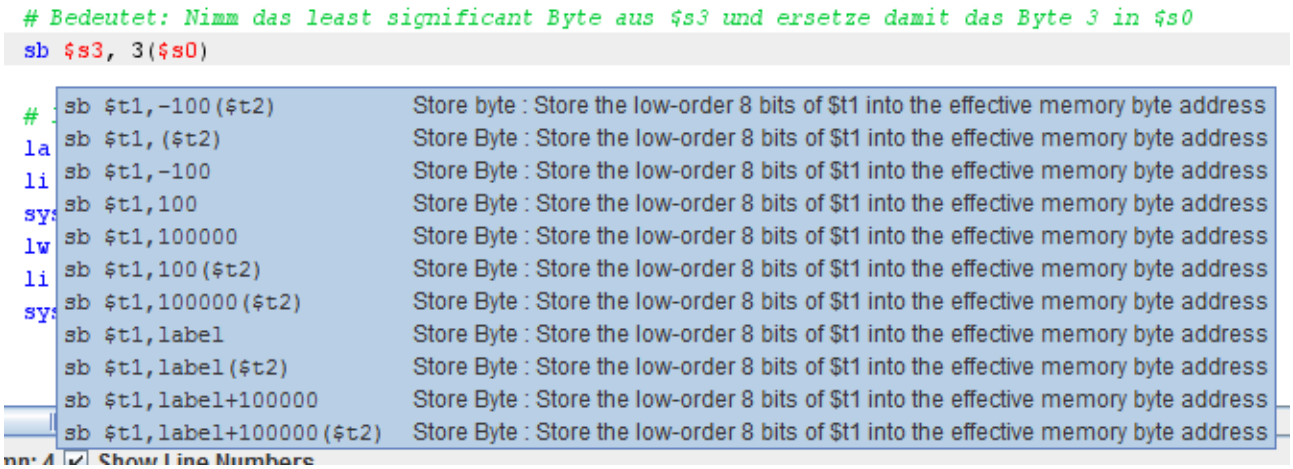
$t7	15	0x00000000
$s0	16	0x10010000
$s1	17	0x0000008c
$s2	18	0xffffff8c
$s3	19	0x6c00216f
$s4	20	0x00000000

Die oberen 3 Byte des ursprünglichen Eintrags in s2 sind verloren gegangen, überschrieben durch das Vorzeichenbit des geladenen Bytes *0x8C*, hier also durch 1en.

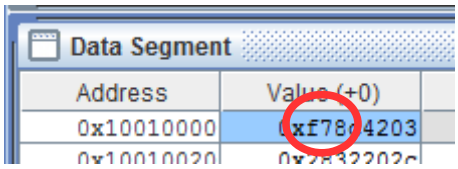
Es folgt die Demonstration des ***sb***-Befehls:

```
# Bedeutet: Nimm das least significant Byte aus $s3 und ersetze damit das Byte 3 in $s0
        sb $s3, 3($s0)
```

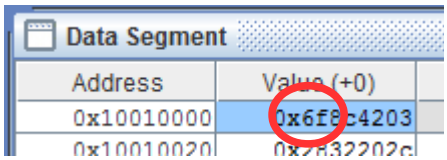
Wesentlicher Unterschied zu den Ladebefehlen ist hier die Reihenfolge der Register im Befehl: Während bei den Ladebefehlen das Zielregister als erstes genannt wird, ist dies beim Speicherbefehl ***sb*** das Quellregister. Also wird das least significant Byte aus dem erstgenannten Register, hier s3, gespeichert. Das Ziel wird wieder mit Anfangsadresse + Offset angegeben, hier ist dies also das Byte 3 des Eintrags, der unter der Adresse zu finden ist, die in s0 steht, also unserer Beispielzahl *src = 0xF78C 4203*. Das Byte 3 ist *0xF7*.Was passiert mit diesem Byte und insbesondere mit den drei anderen Bytes?



Nach Ausführung des ***sb***-Befehls müssen wir im Data Segment unter der Adresse *0x10010000* nachsehen, was mit dem Eintrag dort passiert ist:



vorher



nachher

Es wurde also das Byte 3 ersetzt (durch das least significant Byte, also die unteren 8 Bit des Eintrags in s3) und die anderen 3 Byte in s0 bleiben von der Ausführung unverändert.

Was bleibt ist das Beenden der Programmausführung:

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

Der Wert 10 für den syscall bedeutet „*terminate execution*“ und beendet das Programm.

Diese drei Codefragmente dienen der Ausgabe der hinterlegten Strings und der Registerbelegungen nach Ausführung der jeweiligen Lade-/Speicherbefehle:

```
# In s1 steht nun das mit 0 erweiterte Byte, Ausgabe:
            la $a0, lbu
            li $v0, 4
            syscall
            move $a0, $s1
            li $v0, 34
            syscall
...
...
...

# In s2 steht nun das vorzeichenerweiterte Byte, Ausgabe:
            la $a0, lb
            li $v0, 4
            syscall
            move $a0, $s2
            li $v0, 34
            syscall
…
…
…

# In s0 wurde durch den sb Befehl das Byte 3 durch das Byte 0 aus $s3 ersetzt
            la $a0, sb
            li $v0, 4
            syscall
            lw $a0, ($s0)
            li $v0, 34
            syscall
```

- Zuerst wird die Adresse des Strings, der ausgegeben werden soll, in a0 geladen, dann der Wert 4 (print string) in $v0 geschrieben, sodass der syscall für die Ausgabe des Strings sorgt.
- Als Nächstes wird der Wert, der ausgegeben werden soll, in a0 geschrieben und mit dem syscall Wert 34 erfolgt die Ausgabe dieses Wertes in hexadezimaler Darstellung:

```
 Nach dem Ladebefehl lbu $s1, 2($s0) steht in Register $s1 nun das mit 0en erweiterte Byte 2 des Eingabewortes:
0x0000008c
 Nach dem Ladebefehl lb $s2, 2(s0) steht in Register $s2 nun das vorzeichenerweiterte Byte 2 des Eingabewortes:
0xffffff8c
 Mit dem Speicherbefehl sb $s3, 3(s0) wird das Byte 3 in $s0 durch das least significant Byte
 aus $s3 ersetzt, die anderen Bytes aus $s3 wurden ignoriert:
0x6f8c4203
-- program is finished running --
```

Relevanter Unterschied beim Code für die Ausgabe:
Um einen Registerwert auszugeben schreibt man folgende Codezeile: *move $a0, $s2*
die den Inhalt von s2 nach a0 kopiert, als Parameter für den syscall. Um den Wert auszugeben, der unter der Adresse zu finden ist, die in einem Register steht: *lw a0, ($s0),* diese Zeile lädt das Wort, das unter der Adresse in s0 zu finden ist, in a0 als Parameter für den syscall.