

DS-Systeme

Kapitel 6

**Datentypen
und Datenzugriff**



Inhaltsverzeichnis

Thema	Seite
Der GNU-Assembler GAS	3
"Hello World!" als Assembler-Programm	
- Unterschiede in der Vorbereitung für as und gcc	5
- unter Verwendung von as	7
- unter Verwendung von gcc	8
Allgemeine Bemerkungen:	9
- Spezialzeichen	
- Reihenfolge der Operanden bei Befehlen	
Suffixe bei Instruktionen	10
GAS – Definition von Daten	11
.section .bss – mit 0 initialisierte Daten	13
Zugriff auf Daten	15
- Aufgabe	16
Der Befehl - mov	17
- movz	19
- movs	20
Pointer	22
Stackbefehle	23
Aufgabe	25

GNU-Assembler- GAS

- Der Befehl **as** wird verwendet, um den GNU-Assembler aufzurufen, der für die Übersetzung von Assembler-Code in ausführbare Programme verantwortlich ist.

Konkret nimmt er den Quellcode in Assemblersprache als Eingabe und erzeugt daraus eine Objektdaten im ELF-Format (Executable and Linkable Format). Diese Objektdaten können dann von einem Linker (z.B. dem GNU-Linker **ld**) verwendet werden, um ein ausführbares Programm zu erstellen.

- **as** kann mit verschiedenen Optionen und Dateinamen aufgerufen werden. Die Reihenfolge der Dateinamen ist signifikant. Optionen können in beliebiger Reihenfolge erscheinen (vor, nach und zwischen Dateinamen).
- Jede Option ändert das Verhalten von **as**.
- Keine Option ändert die Funktionsweise einer anderen.
- Eine Option besteht aus einem Bindestrich, gefolgt von einem oder mehreren Buchstaben (case sensitive).
- Eine besondere Option ist der doppelte Bindestrich "--" ohne anschließenden Buchstaben. Er repräsentiert die Standardeingabe (Tastatur). D.h. es wird das assembliert, was über die Tastatur eingegeben wird. Ende der Eingabe mit <strg> <D>.
- Auf manche Optionen muss genau ein Dateiname folgen
z.B.: **as source.s -o objectfile.o**

GNU-Assembler- GAS

- Ein Aufruf von **as** mit einer **.s**-Datei erzeugt eine **Objekt-Datei** namens **a.out** (**Achtung:** nicht zu verwechseln mit der ausführbaren Datei **a.out**). Deshalb möglichst immer die Option **-o** verwenden, um einen eigenen Namen anzugeben.
- Die Objektdatei ist für die Eingabe in den Linker **ld** gedacht. Sie enthält zusammengesetzten Programmcode, Informationen, die dabei helfen, das zusammengesetzte Programm in eine ausführbare Datei zu integrieren, und (optional) symbolische Informationen wie Funktions- und Variablennamen.
- Wenn Sie diese Informationen zum Debuggen nutzen möchten, benötigen Sie noch die Option **-g**.

Hello World-Programm **Unterschiede** in der Vorbereitung für **as** und **gcc**

```
# data section (uninitialized / init with zero)
.section .bss

# data section (read/write)
.section .data

# data section read only
.section .rodata
# Example:
outstring:          # label
.asciz "Hello, World!\n" # value at label (here outstring:)

#code section
.section .text

#.globl _start
#.type _start, @function
#_start:            # start if using gas

.globl main
.type main, @function
main:               # start if using gcc directly
```

cont. -->

Hello World-Programm

--> continued

```
pushq %rbp                # saves base pointer of stack
movq %rsp, %rbp           # sets the current value of the stack pointer
                           # (RSP) into the base pointer (RBP) to establish
                           # a new stack frame.

## hand over parameters via reg (order RDI, RSI, RDX, RCX, R8, R9; XMM0-7)

movq $outstring, %rdi     # $-operator=gets address of label

# call of function
call puts
# free stack (allocated parameters, here none)

# exit main using gcc directly
movq $0, %rax             # return 0 (success)
popq %rbp
ret

# exit main using gas
# movq $60, %rax          # sys_exit
# movq $0, %rdi           # return 0 (success)

# popq %rbp              # restores base pointer of stack
# syscall                 # execute Command no. 60
```

Hello Word-Programm - unter Verwendung von **as**

```

.section .rodata
outstring:
.asciz "Hello World! AS version\n"

.section .text
.type _start, @function
.globl _start

_start:
    pushq %rbp
    movq %rsp, %rbp
    movq $1, %rax           # system call 1 is write
    movq $1, %rdi           # file handler 1 is stdout
    movq $outstring, %rsi   # address of string to output
    movq $26, %rdx          # number of bytes
    syscall

    # exit(0)
    movq $60, %rax          # system call 60 is exit
    xorq %rdi, %rdi         # we want to return code 0
    popq %rbp
    syscall

```

Ausgabe:

Hello World! AS version

```

Assemblieren:      $ as hello.s -o hello.o
Linken:            $ ld hello.o -o hello
Assemblieren und  $ as hello.s -o hello.o && ld hello.o -o hello
Ausführen:        $ ./hello

```

Hello Word-Programm - unter Verwendung von gcc

```
# gcc-Version of HelloWorld

.section .rodata
outstring:
.asciz "Hello World! - GCC version\n"

.section .text
.globl main
.type main, @function

main:
    movq $outstring, %rdi
    call puts

    movq $0, %rax
    ret
```

Ausgabe:

Hello World! - GCC version

```
main:
    pushq %rbp
    movq %rsp, %rbp

    movq $outstring, %rdi
    call puts

    movq $0, %rax
    popq %rbp
    ret
```

```
Kompilieren:      $ gcc -c hello.s -o hello.o
Linken:           $ gcc hello.o -o hello -no-pie

Kompilieren und linken: $ gcc hello.s -o hello -no-pie

Ausführen:       $ ./hello
```



GAS - Allgemeine Bemerkungen (Spezialzeichen und Reihenfolge von Operanden)

- # oder ; - einzeilige Kommentare beginnen mit einem Rautenzeichen '#' (engl. hash) oder einem Semikolon ';'
- /* ... */ - mehrzeilige (nicht schachtelbare) Kommentare
- .text - **Pseudo-Op's** (Direktiven) sind spezielle Anweisungen für den Assembler bzw. Linker und beginnen mit einem Punkt (z.B. .text, .data, .global main, u.s.w.)
- \$ - Literale (engl. immediates) beginnen mit einem Dollarzeichen '\$' (z.B. \$-50, \$'A', \$0x33, u.s.w.)
- % - Register beginnen mit einem Prozentzeichen '%' (z.B. %rax)
- Label: - Label (Bezeichner/Markierung) enden mit einem Doppelpunkt (z.B. start_:)

Reihenfolge der Operanden bei Befehlen (nicht Funktionen):

instruction source, target

- **source** (Quelle) ist der erste und **target** (bzw. **dest**) (Ziel) der jeweils zweite Operand eines Befehls
z.B.: **movq \$outstring, %rdi**


instruction source target

GAS - Allgemeine BemerkungenSuffixe bei Instruktionen

Instruktionen besitzen die in der Tabelle aufgeführten Suffixe (Endungen) und symbolisieren die Größe der Operanden. Die Größe der Register muss zu dem jeweiligen Befehl passen.

Beispiele:

q für 64-bit (quad word)
l für 32-bit (double word)
w für 16-bit (word)
b für 8-bit (byte)




C declaration	Intel data type	Assembly-code suffix	Size (bytes)
<code>char</code>	Byte	<code>b</code>	1
<code>short</code>	Word	<code>w</code>	2
<code>int</code>	Double word	<code>l</code>	4
<code>long *</code>	Quad word	<code>q</code>	8
<code>char *</code> **	Quad word	<code>q</code>	8

Beispiel: `movq $8, %rax`

* Hier ist **long** mit 8 Byte unter **Linux** gemeint (long/Windows 4 Byte).
 ** Pointer (hier **char ***) werden grundsätzlich in 64-Bit gespeichert.

GAS - Definition von Daten

Variablen und Konstanten werden in der `.section .data` bzw. `.section .rodata` (`read only`) durch spezielle Direktiven definiert.

Directive	Data Type
<code>.ascii</code>	Text string
<code>.asciz</code>	Null-terminated text string
<code>.byte</code>	Byte value
<code>.double</code>	Double-precision floating-point number
<code>.float</code> 	Single-precision floating-point number
<code>.int</code>	32-bit integer number
<code>.long</code> 	32-bit integer number (same as <code>.int</code>)
<code>.octa</code>	16-byte integer number
<code>.quad</code>	8-byte integer number
<code>.short</code>	16-bit integer number
<code>.single</code> 	Single-precision floating-point number (same as <code>.float</code>)

Achtung: Die Direktive `.long` wird hier mit 4 Byte definiert.
`.float` und `.single` sind gleichbedeutend.

GAS – Definition von DatenBeispiel:

```

.section .data
height:
    .quad 5
length:
    .quad 6, 7, 8    # int64 array (usage see later)

.section .rodata
fstring:
    .asciz "The result is %d\n"

.section .text
.globl main
.type main, @function

main:
    pushq %rbp
    movq %rsp, %rbp
    movq height, %rax
    addq length, %rax

    #exit main using gcc directly
    movq $0, %rax
    popq %rbp
    ret

```

Syntax:

```

labelname:
    .data type value [, value, ... ]

```

In diesem Programm noch kein Aufruf für die Ausgabe des Formatstrings – mehr dazu später.

main und andere Funktionen befinden sich immer in der **.section .text**.

Hinweis:

Jeder **.section**-Name darf nur einmal vorkommen. Weitere Vorkommen werden von der ersten **section** mit dem gleichen Namen überschrieben.

GAS - Definition von Daten - nicht initialisiert bzw. zero-initialised

Der Speicherbereich **.bss** (**B**lock **S**tarted by **S**ymbol) eignet sich z. B. für Felder (Arrays), die nicht mit vordefinierten Werten initialisiert sind. Der Lader (läd das Programm in den Arbeitsspeicher) wertet dann diese Information aus und fordert einen entsprechend großen Speicherbereich vom Betriebssystem an, wobei er sicherstellt, dass der Speicherbereich mit Nullwerten initialisiert wird.

In der Objektdati werden üblicherweise nicht die Nullwerte gespeichert, sondern nur die Größe des **.bss**-Bereichs. Im Segment **.section .bss** sind die Pseudo-Op's **.comm** bzw **.lcomm** definiert.

Vorteil:

Im Gegensatz zu **.data** sind **.bss**-Daten nicht Teil der ausführbaren Datei, also wird der Speicherverbrauch für den Code dadurch reduziert.

Syntax: **.comm** <symbol>, <length>

Directive	Description
.comm	Declares a common memory area for data that is not initialized
.lcomm	Declares a local common memory area for data that is not initialized

GAS – Definition von Daten – nicht initialisiert bzw. zero-initialisedBeispiel:

```
.section .bss
.comm int64var, 8

.section .data
height:
    .quad 5

.section .text

.globl main
.type main, @function

main:
    pushq %rbp
    movq %rsp, %rbp

    movq height, %rax
    movq %rax, int64var

    movq $0, %rax
    popq %rbp
    ret
```

Der Name ist frei gewählt.
Die Größe wird in Byte angegeben.

Unterschied **.comm** und **.lcomm**:

Besteht das Programm aus mehreren Modulen (Übersetzungseinheiten), sind die **.comm**-Daten in allen (global), die **.lcomm**-Daten nur in dem Modul sichtbar in dem sie deklariert wurden (static).

GAS - Zugriff auf Daten - Verschiedene Arten von Operanden

Type	Form	Operandenwert
Immediate	\$Imm	Imm
Register	r_a	$R[r_a]$
Memory (absolut)	Imm	$M[Imm]$
Memory (indirect)	(r_b)	$M[R[r_b]]$
Memory (indexed)	$Imm(r_b)$	$M[Imm + R[r_b]]$
Memory (indexed)	(r_b, r_i)	$M[R[r_b] + R[r_i]]$

Allgemeine Formel der indizierten Adressierung (memory indexed):

$Imm(r_b, r_i, s)$	$M[Imm + R[r_b] + R[r_i] * s]$
--------------------	--------------------------------

Immediate: konstante Werte mit Präfix \$ - z.B.: \$-577 oder \$0x1F
z.B. `movq $-577, %rax`

Register: Register mit Präfix %
z.B.: `movq %rax, %rbx`

Memory: Memory Access; die allgemeinste Form ist $Imm(r_b, r_i, s)$
 r_b : Basisregister; r_i : Indexregister;
 Imm: intermediate offset; $s \in \{1, 2, 4, 8\}$

Anwendung: z.B. Zugriff auf Arrays

GAS - Zugriff auf Daten - Aufgabe

Gegeben sind folgende Werte in Registern und Speicheradressen.

Address	Value	Register	Value
0x100	0xFF	%rax	0x100
0x104	0xAB	%rcx	0x1
0x108	0x13	%rdx	0x3
0x10C	0x11		

Füllen Sie für folgende Operatoren die Tabelle aus:

Operand	Wert (Wert //Operand typ und bei memory typ auch Adresse)
%rax	
0x104	
\$0x104	
(%rax)	0xFF //Memory (indirect) Address 0x100
4(%rax)	
9(%rax,%rdx)	
0xFC(,%rcx,4)	
(%rax,%rdx,4)	

GAS – Zugriff auf Daten – Der Befehl `mov`

Instruction		Effect	Description
MOV	<i>S, D</i>	$D \leftarrow S$	Move (Copy)
<code>movb</code>			Move byte
<code>movw</code>			Move word
<code>movl</code>			Move double word
<code>movq</code>			Move quad word
<code>movabsq</code>	<i>I, R</i>	$R \leftarrow I$	Move absolute quad word

Source und Destination müssen zu der Größe des Befehls passen. Die Größe wird durch das Suffix hinter **mov** bestimmt.

Mit **movabsq** kann man besonders große (8 Bytes) vorzeichenlose bzgl. Ganzzahlen verschieben.

mov (engl. to move – bewegen)

- Kopiert ein Datum von der Quelle (source) in das Ziel (destination).
- **Nur ein Operand darf eine Speicheradresse repräsentieren.**
- Ein Immediate-Operand wird immer als vorzeichenbehaftet (signed) interpretiert.
- Nur der Teil des Registers, der durch die Größe des Operanden beschrieben wird, wird beeinflusst.
 - Ausnahme: 'l' für 32 Bit: Hier werden die oberen 32 Bit als Nullen und die unteren wie gegeben behandelt (hat historischen, also keinen techn. Hintergrund).

GAS – Zugriff auf Daten – Der Befehl [mov](#)Beispiele:

von --> nach

```
movl  $0x4050, %eax    # Immediate --> Register, 4 Byte
movw  %bp, %sp         # Register --> Register, 2 Byte
movb  (%rdi, %rcx), %al # Memory* --> Register, 1 Byte
movb  $-17, (%rsp)     # Immediate --> Memory**, 1 Byte
movq  %rax, -12(%rbp)  # Register --> Memory*, 8 Byte
```

* Memory (indexed)

** Memory (indirect)

GAS - Zugriff auf Daten - Der Befehl `movz`

`movz` = `mov` mit `z` für zero extension;

Im Suffix: Erster Buchstabe für **source size**, zweiter für **destination size**.

Instruction	Effect	Description
<code>MOVZ S, R</code>	$R \leftarrow \text{ZeroExtend}(S)$	Move with zero extension
<code>movzbw</code>		Move zero-extended byte to word
<code>movzbl</code>		Move zero-extended byte to double word
<code>movzwl</code>		Move zero-extended word to double word
<code>movzbq</code>		Move zero-extended byte to quad word
<code>movzwq</code>		Move zero-extended word to quad word

Die Nullen werden jeweils **vorne** angefügt.

* * Wegen des speziellen Verhaltens bei Befehlen mit der Endung 'l', werden die oberen 4 Bytes des Zielregisters mit 0en überschrieben.

Anmerkung:

Es gibt kein `movzlq`, da `movl` die Erweiterung automatisch hinzufügt.
(s. Ausnahme vorletzte Folie)

GAS - Zugriff auf Daten - Der Befehl `movs`

movs = **mov** mit **s** für **signed extension** (Vorzeichenerweiterung);

Im Suffix: Erster Buchstabe für **source size** und zweiter für **destination size**.

Instruction	Effect	Description
<code>MOVS S,R</code>	$R \leftarrow \text{SignExtend}(S)$	Move with sign extension
<code>movsbw</code>		Move sign-extended byte to word
<code>movsbl</code>		Move sign-extended byte to double word
<code>movswl</code>		Move sign-extended word to double word
<code>movsbq</code>		Move sign-extended byte to quad word
<code>movswq</code>		Move sign-extended word to quad word
<code>movslq</code>		Move sign-extended double word to quad word
<code>cltq</code>	$\%rax \leftarrow \text{SignExtend}(\%eax)$	Sign-extend <code>%eax</code> to <code>%rax</code>

Befehle wie **movz** und **movs** werden benötigt, weil in **GAS** für Datentypen die Qualifizierer **signed** bzw. **unsigned** nicht verfügbar sind.

← *

← *

Anmerkung:

cltq
(**c**onvert **l**ong **t**o **q**uad)
besitzt keine Operanden und benutzt `%eax` bzw. `%rax` implizit.

GAS - Zugriff auf Daten - Der Befehl [movs](#)Beispiele:

```

movabsq $0x0011223344556677, %rax # %rax = 0011223344556677
movb    $0xAA, %dl                # %dl  = AA
movb    %dl, %al                  # %rax = 00112233445566AA
movsbq  %dl, %rax                 # %rax = FFFFFFFF000000AA
movsbl  %dl, %eax                 # %rax = 00000000FFFFFFAA
cltq    ← nur %eax --> %rax      # %rax = FFFFFFFF000000AA
          keine anderen Register
movzbq  %dl, %rax                 # %rax = 00000000000000AA

```

← *

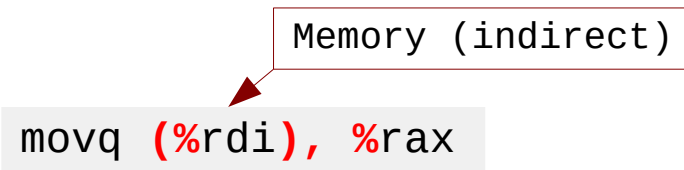
GAS - Zugriff auf Daten - Pointer

Pointer "zeigen" auf eine Adresse.

Sie werden dereferenziert, indem man sie (also die Adresse) in ein Register lädt und einen Memory Access (Speicherzugriff) ausführt.

Beispiel:

Das Register **%rdi** enthält eine Adresse. Der Wert auf den diese Adresse "zeigt", wird in das Register **%rax** kopiert.



```
movq (%rdi), %rax
```

Memory (indirect)

Anmerkung:

Die Adresse einer **Variablen** erhält man z.B. mit dem Befehl **leaq** – mehr dazu später.

GAS - Zugriff auf Daten - Stack-Befehle

Instruction	Effect	Description
<code>pushq S</code>	$R[\%rsp] \leftarrow R[\%rsp] - 8;$ $M[R[\%rsp]] \leftarrow S$	Push quad word
<code>popq D</code>	$D \leftarrow M[R[\%rsp]];$ $R[\%rsp] \leftarrow R[\%rsp] + 8$	Pop quad word

Der Stack vergrößert sich mit jeder Aufnahme eines neuen Datums in Richtung der niedrigeren Adressen. Deshalb zeigt der Stackpointer **rsp** immer auf das untere, also der Adresse 0 nähere Ende des Stacks.

Anmerkungen:

`pushq %rbp`

ist äquivalent zu

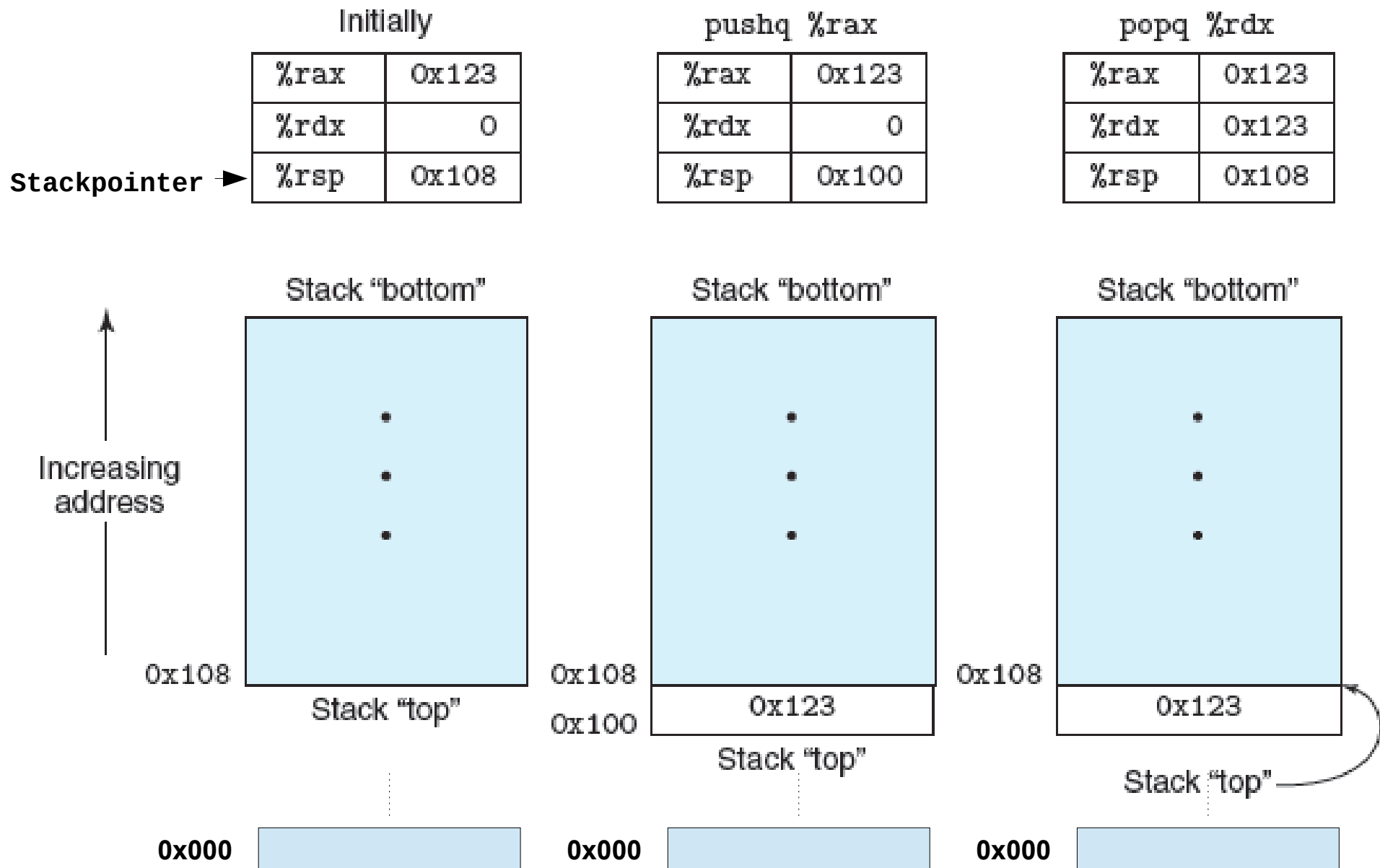
```
subq $8, %rsp
movq %rbp, (%rsp)
```

`popq %rax`

ist äquivalent zu

```
movq (%rsp), %rax
addq $8, %rsp
```

GAS - Zugriff auf Daten - Stack-Befehle



GAS - Zugriff auf Daten - Aufgabe

Gegeben ist folgender Assembler-Code, der die C-Funktion **decode1()** darstellt:

```
#void decode1(long *px, long *py, long *pz)
#px in %rdi, py in %rsi, pz in %rdx

decode1:
    movq (%rdi), %r8    # x in r8
    movq (%rsi), %rcx   # y in rcx
    movq (%rdx), %rax   # z in rax
    movq %r8, (%rsi)
    movq %rcx, (%rdx)
    movq %rax, (%rdi)

    ret #return;
    #in this case
    #return nothing
```

- a) Schreiben Sie die zugehörige C-Funktion entsprechend dem in der ersten Zeile angegebenen Prototyp. Verwenden Sie dabei die in der zweiten Zeile angegebenen Variablennamen und für **r8** **x**, für **rcx** **y** und für **rax** **z**.
- b) Angenommen vor dem Funktionsaufruf würden in `main()` die Werte `long x = 100l`, `long y = 20l` und `long z = 3l` gesetzt und mittels `printf("x = %ld, y = %ld, z = %ld\n", x, y, z);` ausgegeben. Wie sähe die Ausgabe dann aus?