DS-Systeme

Kapitel 11

Funktionen Teil 1

Inhaltsverzeichnis

Funktionen - Einführung

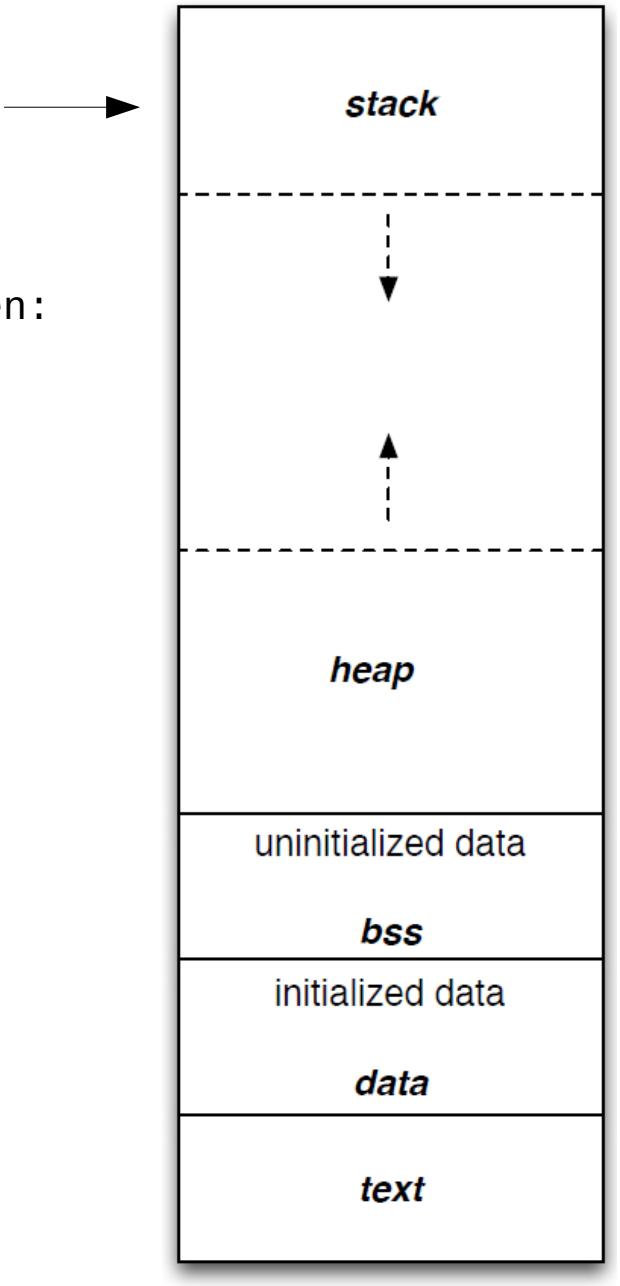
Hochsprachen unterstützen Funktionen (functions / subroutines) um die Wiederverwendung von Code zu ermöglichen und um Programme besser zu strukturieren

Der Aufruf einer Funktion benötigt folgende Mechanismen:

1. P**assing control**
   Übergeben der Kontrolle an die aufgerufene Funktion und wieder zurück.

2. **Passing data**
   Übergeben von Daten an die aufgerufene Funktion.

3. **Allocating and deallocating memory**
   Allokieren und Deallokieren von Speicher für lokale Variablen.



Leaf procedures

... sind Funktionen, die

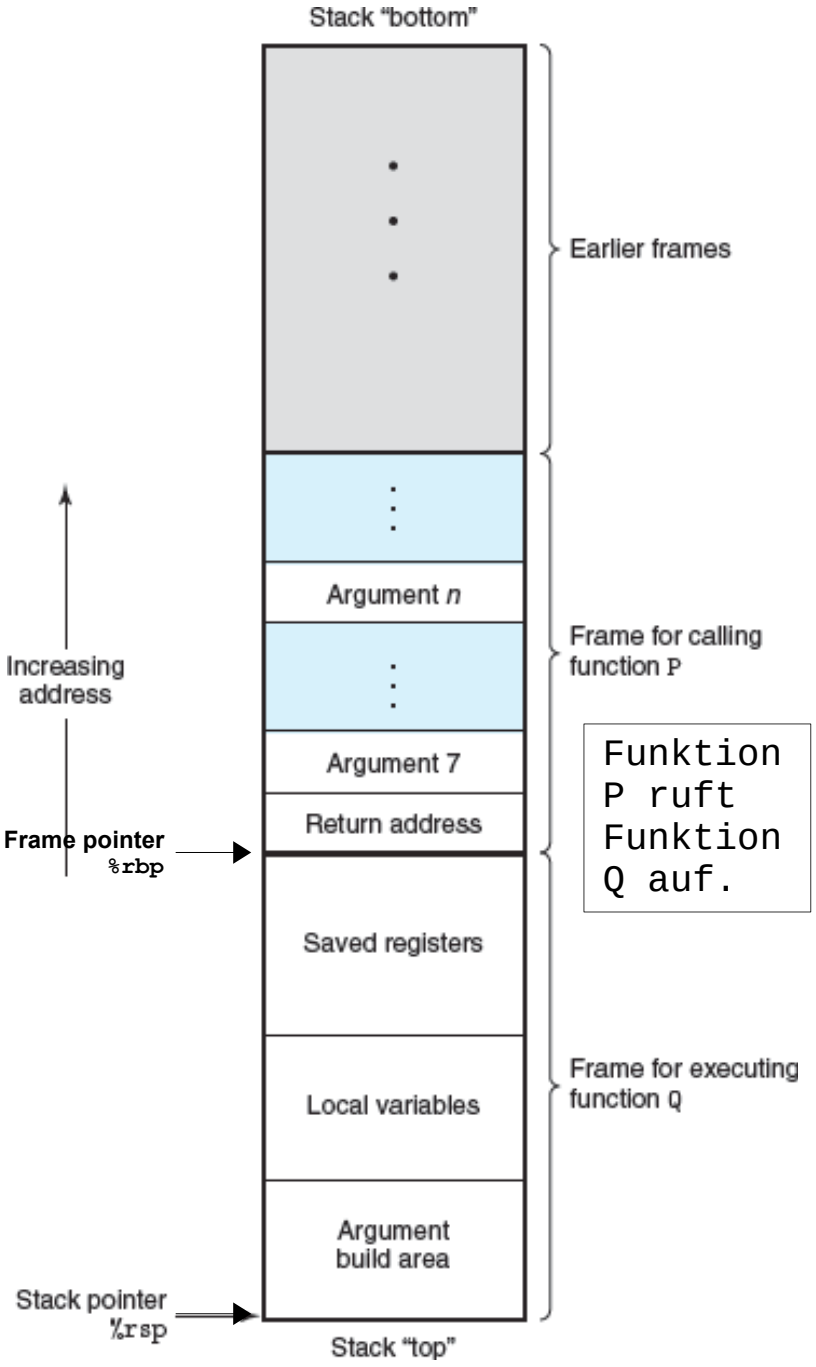
- weniger als sechs Argumente besitzen
- selbst keine anderen Funktionen aufrufen

Sie werden nicht auf dem Stack abgelegt.

Der Runtime-Stack

Einer der wichtigsten Bestandteile für den korrekten Ablauf von Funktionen ist der **Runtime-Stack**.

- Der Stack wächst von höheren Adressen zu niedrigeren Adressen.
- Der Stapelzeiger (stackpointer) **%rsp** zeigt auf das oberste Element (top element) des Stacks (also auf die **niedriegste** Adresse).
- Der Basiszeiger (base- oder bottompointer) **%rbp** zeigt auf den Beginn des aktuellen Stackframes.
- Die Befehle zum Speichern und wieder Abholen lauten: **push** und **pop**
- Wenn eine Funktion mehr Speicher benötigt, als in den zur Verfügung stehenden Registern gehalten werden kann, wird Speicher auf dem Stack alloziert.
- Alle zu einer Funktion gehörenden Teile der Funktion werden als **Stackframe** bezeichnet.



Passing control

Caller – Aufrufer - Funktion, die eine andere Funktion aufruft
(z.B. **main()** ruft **printf()** auf)

Callee – Aufgerufener - Funktion, die von einer anderen Funktion aufgerufen wird
(z.B. **printf()** wird von **main()** aufgerufen)

- Um zum **Caller** zurückgehen zu können, muss die Rücksprungadresse (**return address**) (nächster Befehl nach **call**) auf dem Stack gespeichert werden.
- Das Übergeben der Kontrolle an eine Funktion (z.B. Funktion P ruft Funktion Q auf) wird durch das Setzen des Befehlszeigers (**instruction pointer) %rip** auf die Startadresse des Codes der Funktion bewerkstelligt.
  - Dies erledigt der Befehl **call** implizit!
- Wenn zum Caller zurückgegangen wird, wird die Rücksprungadresse wieder vom Stack geholt und **%rip** auf diese Adresse gesetzt.
  - Dies erledigt der Befehl **ret** implizit!

Passing control

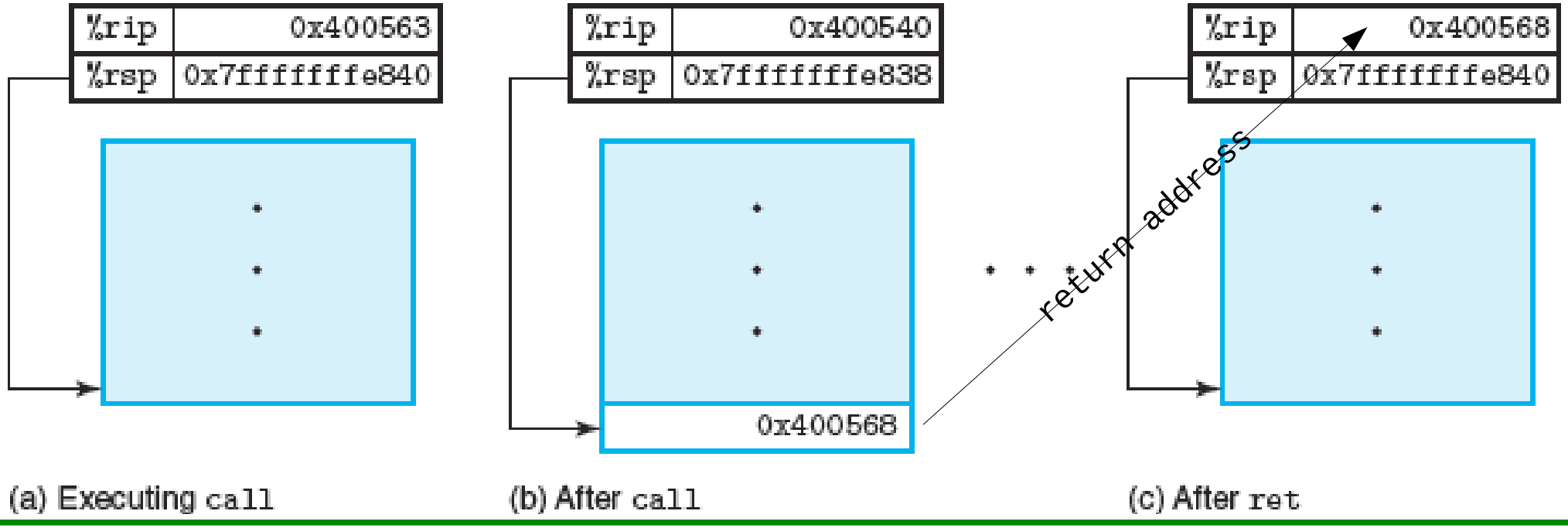
C/C++ - Code:

```
void myfn(...){
  ...
}

int main() {
  ...
  myfn(...);
  ...
}
```

Assemblierter Code:

```
Beginning of function myfn
400540 <myfn>:
...
Return from function myfn
40054d: c3                    ret

Call of myfn from main
400563: e8 d8 ff ff ff        call 400540 <myfn>
400568: 48 8b 54 24 08        ...
```

.section .text



(a) Executing call (b) After call (c) After ret

Passing data (Argumente und Rückgabe)

- Prinzipell werden die meisten Funktionsargumente (max. 6) mittels Registern an Funktionen übergeben.
- Wenn die Register nicht mehr ausreichen, wird der Stack verwendet.
- Der Rückgabewert muss vom **Callee** immer in **%rax** (bzw. **%eax**, u.s.w.) gespeichert werden.

Passing data (Argumente und Rückgabe)

Beispiel **calc01:** Noch nicht korrekt!

C/C++ - Code:

```c
#include <stdio.h>

int calc(int a, int b,
         int c, int d){
  int res;
  res = (a - b) + (c - d);
  return res;
}

int main() {
  int res;
  res = calc(3, 2, 6, 4);
  return 0;
}
```

Assembler Code:

```
.section .text
.globl calc
.type calc, @function
calc:
  subl  %ebx, %eax # a - b
  subl  %edx, %ecx # c - d
  addl  %ecx, %eax # (a – b) + (c - d)
  ret
.globl main
.type main, @function
main:
  pushq %rbp
  movq %rsp, %rbp
  movl  $4, %edx # d
  movl  $6, %ecx # c
  movl  $2, %ebx # b
  movl  $3, %eax # a
  # call of function calc
  call calc
  # exit main
  movq $0, %rax
  popq %rbp
  ret
```

Passing data (Argumente und Rückgabe)

Beispiel **calc01**: Noch nicht korrekt!

Assember Code:

```
.section .text
.globl calc
.type calc, @function
calc:
  subl  %ebx, %eax # a - b
  subl  %edx, %ecx # c - d
  addl  %ecx, %eax # (a – b) + (c - d)
  ret
.globl main
.type main, @function
main:
  pushq %rbp
  movq %rsp, %rbp
  movl  $4, %edx # d
  movl  $6, %ecx # c
  movl  $2, %ebx # b
  movl  $3, %eax # a
  # call of function calc
  call calc
  # exit main
  movq $0, %rax
  popq %rbp
  ret
```

Eine der Aufgaben des Stacks ist es, die Register, die eine Funktion verwendet, zu sichern und wiederherzustellen.

Dies ist in dem Beispiel **calc01** nicht der Fall:
%eax, **%ebx**, **%ecx** und **%edx** werden hier kaputt geschrieben!

Um dies zu lösen, sichert die Funktion die von ihr benutzten Register zu Beginn der Funktion auf den Stack, und stellt die Register am Ende der Funktion wieder her.

Vorgehensweise:

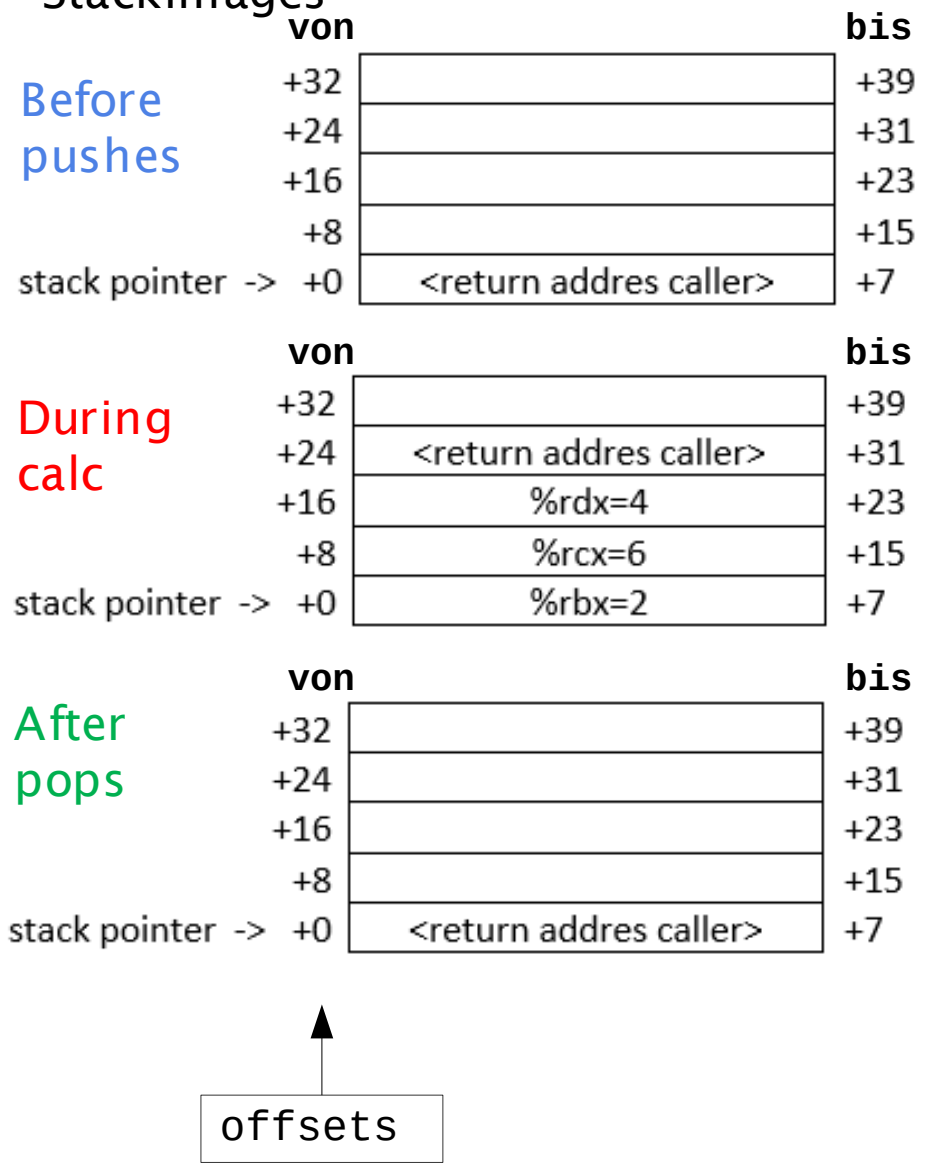
- Register auf den Stack sichern
- Funktion ausführen
- Originalwerte der Register wiederherstellen

Passing data (Argumente und Rückgabe)

Version **calc02**:

```
.section .text
.globl calc
.type calc, @function
calc:
  #push in reverse order
  #(LIFO!!!)
  pushq %rdx
  pushq %rcx
  pushq %rbx
  #pushq %rax
  subl %ebx, %eax #a-b
  subl %edx, %ecx #c-d
  addl %ecx, %eax #(a-b)+(c-d)
  #popq %rax
  popq %rbx
  popq %rcx
  popq %rdx
  ret
```

- Stackimages



Passing data (Argumente und Rückgabe)

Version **calc02**:

```
.section .text
.globl calc
.type calc, @function
calc:
  # push in reverse order
  # LIFO
  pushq %rdx
  pushq %rcx
  pushq %rbx
  # pushq %rax
  subl %ebx, %eax # a-b
  subl %edx, %ecx # c-d
  addl %ecx, %eax # (a-b)+(c-d)
  # popq %rax
  popq %rbx
  popq %rcx
  popq %rdx
  ret
```

Alle Register sichern (**rax**, **rbx**, **rcx**, **rdx**) und nach Verwendung wieder zurück-speichern.

push/pushq und **pop/popq** funktionieren auf 64-Bit-Plattformen nur mit kompletten 64-Bit-Registern.

Bei der Ablage von Daten mit push auf Stack - umgekehrte Reihenfolge beachten wegen LIFO = Last In – First Out.

Passing data (Argumente und Rückgabe)

Version **calc02**:

```
.section .text
.globl calc
.type calc, @function
calc:
  ##push in reverse order
  ##(LIFO!!!)
  pushq %rdx
  pushq %rcx
  pushq %rbx
  #pushq %rax
  subl %ebx, %eax # a-b
  subl %edx, %ecx # c-d
  addl %ecx, %eax # (a-b)+(c-d)
  #popq %rax
  popq %rbx
  popq %rcx
  popq %rdx
  ret
```

Callee save und **Caller save** Register

Wenn der **Caller** die gesicherten Register nicht verwendet, ist der ganze Aufwand des Sicherns und Wiederherstellens Verschwendung. Daher sind die Register in **Callee save** und **Caller save** eingeteilt (x86-64 Linux 64bit (System V AMD64 ABI)):

Callee save register sind **rbx**, **rbp**, **rsp** und **r12 - r15**

Caller save register sind **rax**, **rcx**. **rdx**, **%rsi**, **rdi**, **r8**, **r9**, **r10 und r11**

Der **Callee** darf die **Caller save** Register frei benutzen. Allerdings muss er die **Callee save** Register, die er verwenden möchte zunächst sichern (save) und anschließend vor dem Verlassen der Funktion (Befehl **ret**) wiederherstellen (restore).

Der **Caller** muss die **Caller save** Register vor dem **call** sichern und nach dem **call** wiederherstellen. Beides ist aber nur nötig, wenn diese von ihm nach dem Funktionsaufruf noch verwendet werden sollen.

Passing data (Argumente und Rückgabe)

Callee save Register und **Caller save** Register
Version **calc04**:

```
.section .text
.globl calc
.type calc, @function
calc:
  pushq %rbx
  subl %ebx, %eax # a-b
  subl %edx, %ecx # c-d
  addl %ecx, %eax # (a – b) + (c - d)
  popq %rbx
  ret
.globl main
.type main, @function
main:
  pushq %rbp
  movq %rsp, %rbp
  movl  $4, %edx # d
  movl  $6, %ecx # c
  movl  $2, %ebx # b
  movl  $3, %eax # a
  # call of calc
  call calc
  # program exit
  movq $0, %rax
  popq %rbp
  ret
```

Der **Callee** verwendet das 4 Byte-Register **ebx**, also muss er vorher das 8 Byte-Register **rbx** sichern und anschließend wieder herstellen.

Die Register **eax**, **ecx** und **edx** werden auch verwendet, aber für die Sicherung ist der **Caller** vorher verantwortlich, falls er die alten Inhalte von vor dem Aufruf des **Callee** noch nutzen möchte.

Das bedeutet, dass er in diesem Beispiel keins der Register sichern muss.

Calling Conventions

Die Übergabe von Funktionsargumenten an Funktionen ist in den sog. **Calling Conventions** definiert. Sie sind abhängig von der Architektur und dem Betriebssystem.

Hier wird x86-64 Linux 64-Bit (System V AMD64 ABI) verwendet.

- Die meisten Daten- bzw. Funktionsargumente (bis zu sechs) werden mittels Registern an Funktionen übergeben.
- Bei mehr als sechs Funktionsargumenten wird der Stack verwendet.
- Die Rückgabe von Werten erfolgt (bei ganzzahligen Datentypen und Pointern) in **rax** oder einem seiner Teilregister (**eax, ax, al**).
- Falls der Stack verwendet wird, müssen die Argumente dort in **umgekehrter Reihenfolge** abgelegt werden.
- Die Calling-Conventions müssen zwingend eingehalten werden.

Calling Conventions

Operand size (bits)	Argument number					
	1	2	3	4	5	6
64	%rdi	%rsi	%rdx	%rcx	%r8	%r9
32	%edi	%esi	%edx	%ecx	%r8d	%r9d
16	%di	%si	%dx	%cx	%r8w	%r9w
8	%dil	%sil	%dl	%cl	%r8b	%r9b

Calling Conventions

C/C++ - Code:

```c
#include <stdio.h>

int calc(int a, int b,
         int c, int d){
  int res;
  res = (a - b) + (c - d);
  return res;
}

int main() {
  int res;
  res = calc(3, 2, 6, 4);
  return 0;
}
```

Version **calc05**:

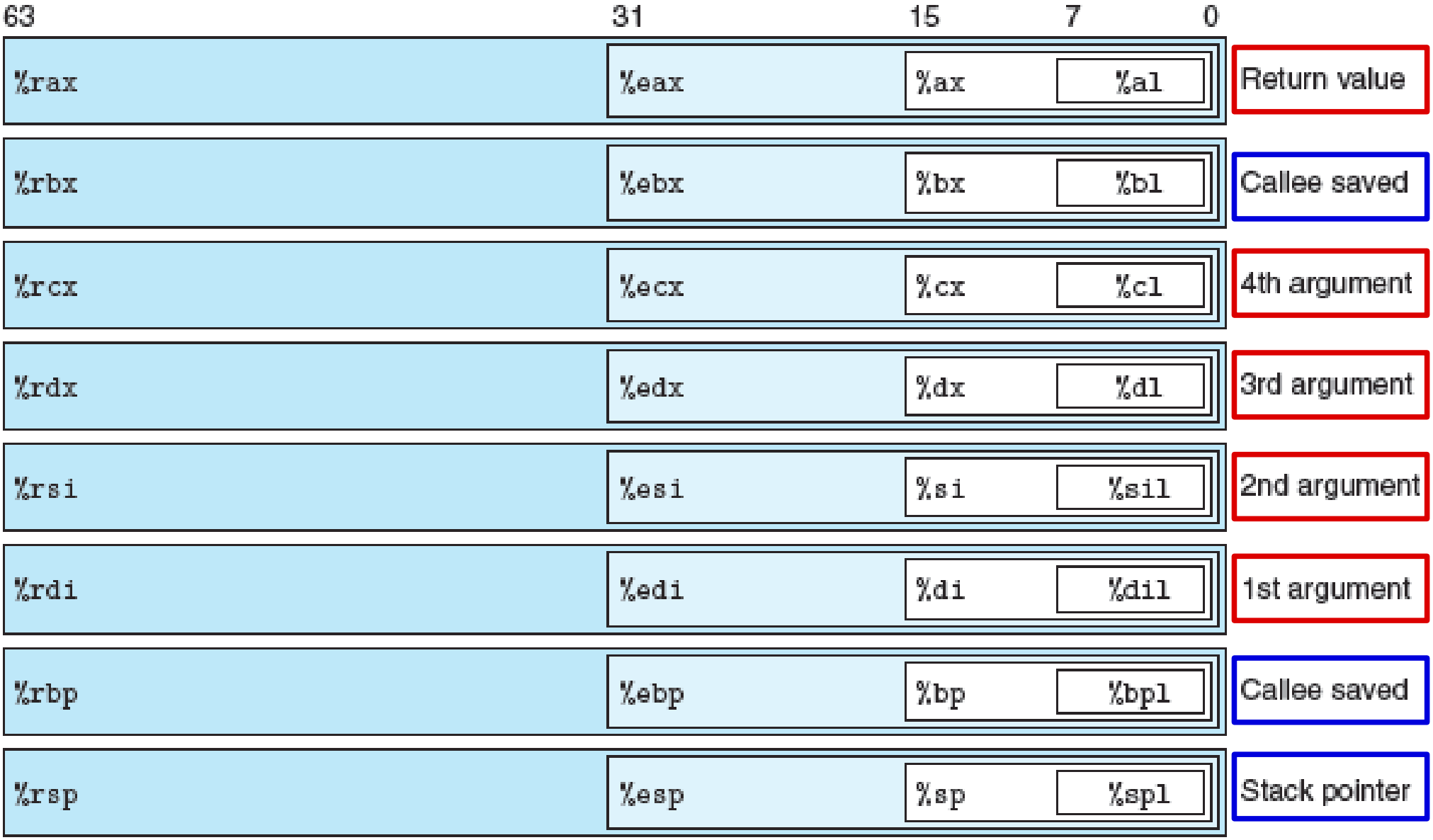
Assember Code:

```
.section .text
.globl calc
.type calc, @function
calc:
  subl  %esi, %edi # a - b
  subl  %ecx, %edx # c - d
  addl  %edx, %edi # (a – b) + (c - d)
  movl  %edi, %eax
  ret
.globl main
.type main, @function
main:
  pushq %rbp
  movq %rsp, %rbp
  # hand over parameters via regs
  # order RDI, RSI, RDX, RCX, R8, R9
  movl  $4, %ecx # d
  movl  $6, %edx # c
  movl  $2, %esi # b
  movl  $3, %edi # a
  # call of function calc
  call calc
  #program exit
  movq $0, %rax
  popq %rbp
  ret
```

Jetzt korrekte Register verwendet!

Calling Conventions

Zusammenfassung: **Caller save** und **Callee save** Register



Calling Conventions

Zusammenfassung: **Caller save** und **Callee save** Register

%r8	%r8d	%r8w	%r8b	5th argument
%r9	%r9d	%r9w	%r9b	6th argument
%r10	%r10d	%r10w	%r10b	Caller saved
%r11	%r11d	%r11w	%r11b	Caller saved
%r12	%r12d	%r12w	%r12b	Callee saved
%r13	%r13d	%r13w	%r13b	Callee saved
%r14	%r14d	%r14w	%r14b	Callee saved
%r15	%r15d	%r15w	%r15b	Callee saved

Calling Conventions - Aufgabe

C/C++ - Code:

```
int myfn(int a, int b, int c){
  return a + b + c;
}

int main() {
  int a = 1;
  int b = 2;
  int c = 3;
  int res = 0;
  res = myfn(a, b, c);
  return res;
}
```

Aufgabe:

Schreiben Sie zu dem Programm den entsprechenden Assembler-Code und halten sich dabei an die Calling Conventions.

Übergabe von mehr als sechs Argumenten

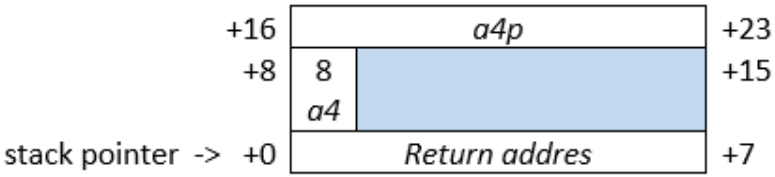
Beispiel:

C/C++ - Code:

```
void proc(
    long  a1, long  *a1p,
    int   a2, int   *a2p,
    short a3, short *a3p,
    char  a4, char  *a4p)
{
  *a1p += a1;
  *a2p += a2;
  *a3p += a3;
  *a4p += a4;
}
```

Nur die beiden letzten Argumente werden auf dem Stack abgelegt.

Stack von proc() **nach** call von proc() mit Stackpointer als Referenz



Assembler - Code:

```
.section .text
  .globl proc
  .type proc, @function
# void proc(a1, a1p, a2, a2p, a3, a3p, a4, a4p)
# Arguments passed as follows:
#   a1  in %rdi    (64 bits)
#   a1p in %rsi    (64 bits)
#   a2  in %edx    (32 bits)
#   a2p in %rcx    (64 bits)
#   a3  in %r8w    (16 bits)
#   a3p in %r9     (64 bits)
#   a4  at %rsp+8  ( 8 bits)
#   a4p at %rsp+16 (64 bits)
proc:
  addq %rdi, (%rsi)    # *a1p += a1
  addl %edx, (%rcx)    # *a2p += a2
  addw %r8w, (%r9)     # *a3p += a3
  movq 16(%rsp), %rax # rax = a4p
  movb 8(%rsp), %dl    # dl = a4 (Teil von %rdx)
  addb %dl, (%rax)     # *a4p = dl
  ret
```

Übergabe von mehr als sechs Argumenten

Assembler – Code kommentiert:

```
.section .text
  .globl proc
  .type proc, @function
# void proc(a1, a1p, a2, a2p, a3, a3p, a4, a4p)
# Arguments passed as follows:
#   a1  in %rdi    (64 bits)
#   a1p in %rsi    (64 bits)
#   a2  in %edx    (32 bits)
#   a2p in %rcx    (64 bits)
#   a3  in %r8w    (16 bits)
#   a3p in %r9     (64 bits)
#   a4  at %rsp+8  ( 8 bits)
#   a4p at %rsp+16 (64 bits)

proc:
  addq %rdi, (%rsi)    # *a1p += a1  (64 bits)
  addl %edx, (%rcx)    # *a2p += a2  (32 bits)
  addw %r8w, (%r9)     # *a3p += a3  (16 bits)
  movq 16(%rsp), %rax  # Fetch a4p   (64 bits)
  movb 8(%rsp), %dl    # Fetch a4    ( 8 bits)
  addb %dl, (%rax)     # *a4p += a4  ( 8 bits)
  ret                  # Return
```

rsp-Adressen hier um 8 höher, weil die return-Adresse auch noch 8 Byte verbraucht.