DS-Systeme Kapitel 8

Kontroll-strukturen Teil 2

Inhaltsverzeichnis

Kontrollstrukturen – Schleifen – do while

C- bzw. C++-Code:

```
do
  // body-statements
while (test-expr);
```

Assembler-Code (dargestellt wie C-Code):

```
loop:
  body-statements
  t = test-expr;
  if (t)
    goto loop;
```

Eine **do while**-Schleife wird immer mindestens einmal durchlaufen.

Deshalb steht das **loop**–Label direkt am Beginn und der Test-Ausdruck **if (t)** mit anschließendem bedingten Sprung **goto loop;** erst am Ende.

Kontrollstrukturen – Schleifen – do while

Beispiel für **do while-Schleife:**

C- bzw. C++-Code:

```
// Calculation of factorial

long fact_do(long n)
{
  long result = 1;
  do {
    result *= n;
    n = n - 1;
  } while (n > 1);
  return result;
}
```

Assembler-Code:

```
.section .text
  .globl fact_do
  .type fact_do, @function

# long fact_do(long n)
# n in rdi

fact_do:
  movq $1, %rax
.L2:
  imulq %rdi
  subq $1, %rdi
  cmpq $1, %rdi
  jg .L2
  rep; ret
```

Rückgabewert muss immer in **rax** stehen.

true-Label

true-statements (body)

Schleifenbedingung wird NICHT negiert.

Kontrollstrukturen – Schleifen – do while

Beispiel für do while-Schleife:

C- bzw. C++-Code:

```
// Calculation of factorial

long fact_do(long n)
{
  long result = 1;
  do {
    result *= n;
    n = n - 1;
  } while (n > 1);
  return result;
}
```

Assembler-Code kommentiert:

```
.section .text
  .globl fact_do
  .type fact_do, @function
# long fact_do(long n)
# n in rdi
fact_do:
  movq $1, %rax      # Set result = 1
.L2:                 # loop:
  imulq %rdi         # Compute result *= n
  subq $1, %rdi      # Decrement n
  cmpq $1, %rdi      # Compare n:1
  jg .L2             # If >, goto loop
  rep; ret           # Return
```

Kontrollstrukturen – Schleifen – while

C- bzw. C++-Code:

```
while (test-expr)
  // body-statement
```

Assembler-Code (dargestellt wie C-Code):

```
 goto test;
loop:
  body-statement
test:
  t = test-expr;
  if (t)
    goto loop;
```

Eine **while**-Schleife wird nicht immer zwingend mindestens einmal durchlaufen. Deshalb wird das **loop**–Label mit body-statement erst einmal mittels goto **test**; zum Test-Ausdruck if **(**t**)** übersprungen.

Der Sprung goto **loop**; befindet sich am Ende, so dass das Programm bei Auswertung zu **false** direkt zur nächsten Zeile weitergehen kann.

Kontrollstrukturen – Schleifen – while

Beispiel für while-Schleife:

C- bzw. C++-Code:

```
// Calculation of factorial

long fact_while(long n)
{
  long result = 1;
  while (n > 1) {
    result *= n;
    n = n - 1;
  }
  return result;
}
```

Assembler-Code:

```
  .globl fact_while
  .type fact_while, @function

# long fact_while(long n)
# n in rdi

fact_while:
  movq $1, %rax
  jmp .L5
.L6:
  imulq %rdi
  subq $1, %rdi
.L5:
  cmpq $1, %rdi
  jg .L6
  rep; ret
```

Rückgabewert muss immer in **rax** stehen.

true-Label

true-statements (body)

test-Label

Schleifenbedingung wird NICHT negiert.

Kontrollstrukturen – Schleifen – while

Beispiel für while-Schleife:

C- bzw. C++-Code:

```
// Calculation of factorial

long fact_while(long n)
{
  long result = 1;
  while (n > 1) {
    result *= n;
    n = n - 1;
  }
  return result;
}
```

Assembler-Code kommentiert:

```
  .globl fact_while
  .type fact_while, @function
# long fact_while(long n)
# n in rdi
fact_while:
  movq $1, %rax     # Set result = 1
  jmp .L5           # Goto test
.L6:                # loop:
  imulq %rdi        # Compute result *= n
  subq $1, %rdi     # Decrement n
.L5:                # test:
  cmpq $1, %rdi     # Compare n:1
  jg .L6            # If >, goto loop
  rep; ret          # Return
```

Aufgabe

Vervollständigen Sie den C-Code:

C- bzw. C++-Code:

```
long loop_while(long a, long b)
{
  long result= ___;

  while (_____)
  {
    result= _______________;
    a = ___;
  }

  return result;
}
```

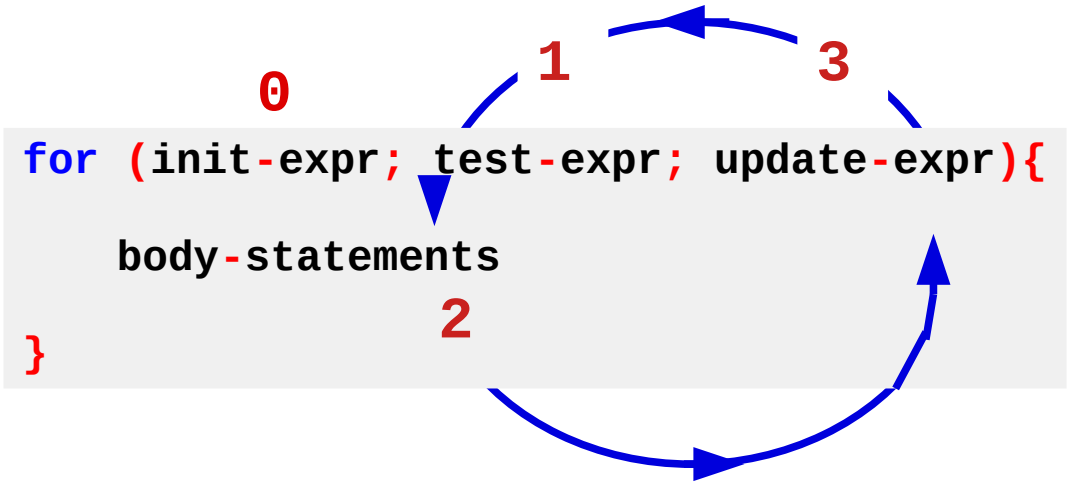
Assembler-Code:

```
#long loop_while(long a, long b)
#a in %rdi, b in %rsi

loop_while:
  movq  $1, %rax
  jmp  .L2
.L3:
  leaq (%rdi, %rsi), %rdx
  imulq %rdx, %rax
  addq  $1, %rdi
.L2:
  cmpq %rsi, %rdi
  jl .L3
  rep; ret
```

Kontrollstrukturen – Schleifen – for

C- bzw. C++-Code:



Assembler-Code
(dargestellt wie C-Code):

```
0     init-expr;

      goto test;

loop:
2     body-statement

3     update-expr;

1   test:
      t = test-expr;
      if (t)
        goto loop;
```

Kontrollstrukturen – Schleifen – for

Beispiel für for-Schleife:

C- bzw. C++-Code:

```
// Calculation of factorial

long fact_for(long n)
{
    long i;
    long result = 1;

    for (i = 2; i <= n; i++){
            result = result * i;
    }

    return result;
}
```

Assembler-Code:

```
 .globl fact_for
  .type fact_for, @function

# long fact_for(long n)
# n in rdi

fact_for:
  movq $1, %rax

  movq $2, %rdx
  jmp .L8

.L9:
  imulq %rdx, %rax
  addq $1, %rdx

.L8:
  cmpq %rdi, %rdx
  jle .L9
  rep; ret
```

init 0

body 2

i++ 3

test 1

Schleifenbedingung wird NICHT negiert.

Kontrollstrukturen – Schleifen – for

Beispiel für for-Schleife:

C- bzw. C++-Code:

```
// Calculation of factorial

long fact_for(long n)
{
    long i;
    long result = 1;

    for (i = 2; i <= n; i++){
            result = result * i;
    }

    return result;
}
```

Assembler-Code kommentiert:

```
  .globl fact_for
   .type fact_for, @function
# long fact_for(long n)
# n in rdi
fact_for:
   movq $1, %rax      # Set result = 1
   movq $2, %rdx      # Set i = 2
   jmp .L8            # Goto test
.L9:                  # loop:
   imulq %rdx, %rax   # Compute result *= i
   addq $1, %rdx      # Increment i
.L8:                  # test:
   cmpq %rdi, %rdx    # Compare i:n
   jle .L9            # If <=, goto loop
   rep; ret           # Return
```

Aufgabe

Vervollständigen Sie den C-Code:

C- bzw. C++-Code:

```
long func (unsigned long x) {

  long val = ___;
  long i;

  for (___; _____; ___)
    val = ___;

  return val;
}
```

Assembler-Code:

```
#long func (unsigned long x)
#x in %rdi

func:

movq    $1, %rax
movq    %rdi, %rcx

jmp .L2

.L3:
imulq   %rcx, %rax
subq    $1, %rcx

.L2:
testq   %rcx, %rcx # cmpq $0, %rcx
jnz .L3
rep; ret
```

Kontrollstrukturen – Bedingte Verzweigung mit bedingtem mov

Die bedingte Verzweigung unter Verwendung von **conditional mov** (**cmov**) ist

einsetzbar, wenn

a) eine einzelne Verzweigung nur ein **mov** oder **add** ist.

b) die Berechnung der unterschiedlichen Zweige schneller ist als der negative Einfluss des Sprungbefehls.

nicht einzusetzen, wenn

a) Einer der Zweige / Pfade zu einem Fehler führen würde (z.B. Dereferenzierung des Nullpointers)

b) ein Zweig umfangreiche Berechnungen benötigt.

Kontrollstrukturen – Bedingte Verzweigung mit bedingtem mov

Assembler-Code mit conditional **jump** (dargestellt wie C-Code):

```
t = test-expr;
if (!t) goto false;
  then-statement
  goto done;
false:
  else-statement
done:
```

Assembler-Code mit conditional **mov** (dargestellt wie C-Code):

```
te = then-expr;
ee = else-expr;
t = test-expr;
if (!t) te = ee;
```

Kontrollstrukturen – Bedingte Verzweigung mit bedingtem mov

Beispiel:

$$absdiff(x, y) = \begin{cases} x - y, \text{ if } x < y \\ y - x, \text{ if } x >= y \end{cases}$$

C-Code:

```
long absdiff(long x, long y)
{
  long result;

  if (x < y)

    result = y - x;

  else

    result = x - y;

  return result;
}
```

Assembler-Code:

```
.section .text
  .globl absdiff
  .type absdiff, @function
# long absdiff(long x, long y)
# x in rdi, y in rsi
absdiff:
  movq %rsi, %rax
  subq %rdi, %rax
  movq %rdi, %rdx
  subq %rsi, %rdx
  cmpq %rsi, %rdi
  cmovge %rdx, %rax
  ret
```

Kontrollstrukturen – Bedingte Verzweigung mit bedingtem mov

Assembler-Code kommentiert:

```
  .text
   .globl absdiff
   .type absdiff, @function
# long absdiff(long x, long y)
# x in rdi, y in rsi
absdiff:
   movq %rsi, %rax
   subq %rdi, %rax    # then-val  = y-x
   movq %rdi, %rdx
   subq %rsi, %rdx    # felse-val = x-y
   cmpq %rsi, %rdi    # Compare x:y
   cmovge %rdx, %rax  # If >=, then-val = else-val
   ret                # Return then_val
```

Instruction		Synonym	Move condition	Description
`cmove`	*S*, *R*	`cmovz`	`ZF`	Equal / zero
`cmovne`	*S*, *R*	`cmovnz`	`~ZF`	Not equal / not zero
`cmovs`	*S*, *R*		`SF`	Negative
`cmovns`	*S*, *R*		`~SF`	Nonnegative
`cmovg`	*S*, *R*	`cmovnle`	`~(SF ^ OF) & ~ZF`	Greater (signed >)
`cmovge`	*S*, *R*	`cmovnl`	`~(SF ^ OF)`	Greater or equal (signed >=)
`cmovl`	*S*, *R*	`cmovnge`	`SF ^ OF`	Less (signed <)
`cmovle`	*S*, *R*	`cmovng`	`(SF ^ OF) | ZF`	Less or equal (signed <=)
`cmova`	*S*, *R*	`cmovnbe`	`~CF & ~ZF`	Above (unsigned >)
`cmovae`	*S*, *R*	`cmovnb`	`~CF`	Above or equal (Unsigned >=)
`cmovb`	*S*, *R*	`cmovnae`	`CF`	Below (unsigned <)
`cmovbe`	*S*, *R*	`cmovna`	`CF | ZF`	Below or equal (unsigned <=)

Aufgabe

Übersetzen Sie den C-Code in Assembler-Code:

C-Code:

```c
long max(long a, long b)
{
  if (a > b)
    return a;
  else
    return b;
}
```

Assembler-Code:

```
#long max(long a, long b)
#a in %rdi, b in %rsi

max:

# your solution here
```