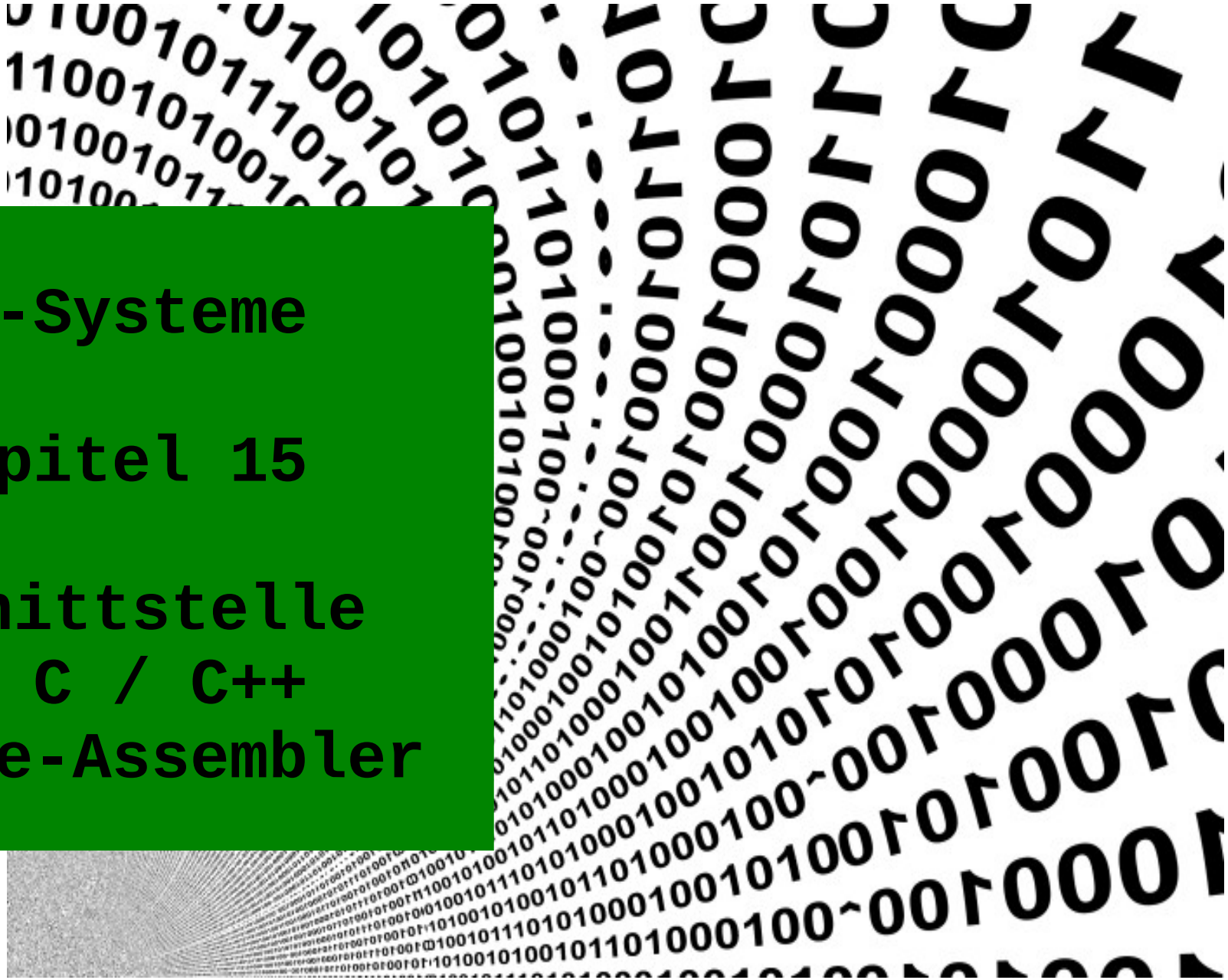

DS-Systeme

Kapitel 15

Schnittstelle zu C / C++ Inline-Assembler

Inhaltsverzeichnis

Thema	Seite
Schnittstelle zu C/C++ - Inline-Assembler	3
- Vor- und Nachteile der Verwendung	4
- Basisversion	5
-- Beispiel	6
Schnittstelle zu C/C++ - Inline-Assembler - erweiterte Version	
- Allgemine Syntax	7
- Beispiel	8
- **output-list**	9
- **input-list**	15
- **overwrite-list**	16
- Limitierungen	17
- Aufgabe	18
- Lösung	19
Inline-Assembler – Verpackt in eine C-Funktion (C-Stub)	20
- Aufgabe	22
- Lösung	23

Schnittstelle zu C/C++ - Inline-Assembler

Es gibt mehrere Möglichkeiten, Assembler mit Hochsprachen wie C oder C++ zu verbinden:

1. Aufruf von C-Bibliotheksfunktionen von Assembler aus
   z.B.: **printf()**

2. Aufruf von Assemblerfunktionen von C / C++ aus

3. **Verwendung von Inline-Assembler**
   (auch **integrierter Assemblercode** genannt)

Vor- und Nachteile zur Verwendung von Inline-Assembler

Vorteile:

- Wenn der meiste Code in C, aber nur wenig Code in Assembler geschrieben werden muss, ist es vorteilhaft **inline-Assembler** zu verwenden.
- Verringert die Menge an Assemblercode, die geschrieben werden muss.
- Reduziert den Overhead, der durch Funktionaufrufe entsteht.

Nachteile:

- Abhängigkeit vom Compiler
  (Punkte 1 und 2 der letzten Folie sind unabhängig vom Compiler.)

Schnittstelle zu C/C++ - Inline-Assembler - Basisversion

Allgemeine Syntax:

- **__asm__ __volatile__ (<codestring>);**

  Die Verwendung von **__volatile__** verhindert, dass der Compiler den Code wegoptimiert. Wenn die Optimierung erwünscht ist, kann man es weglassen.
  Alternative zu **__asm__** ist **asm**, dann aber nicht zwingend ANSI-C-konform.

Limitierungen:

- Nur Zugriff auf globale C-Variablen möglich.
- Der Compiler prüft den Code nicht auf Fehler (gilt nicht nur für die Basisversion).
  Lösung: Verwenden von **gcc -S** um sich anzusehen, wie der Compiler den Code eingebaut hat und ob das so gewünscht ist.
- Der eingebaute Code findet sich in der Ausgabedatei zwischen den Kommentaren **#APP** und **#NO_APP** wieder.

Inline-Assembler - Basisversion

Beispiel:

C-Code:

```
#include <stdio.h>

int res = 0;
int a = 11;

int main()
{
  __asm__ __volatile__(
    "movl a, %eax\n\t"
    "addl $3, %eax\n\t"
    "movl %eax, res\n\t"
  );
  printf("the result is %d\n", res);
  return 0;
}
```

nur globale Variablen möglich.

Assembler-Code (Ausschnitt):

```
main:
# ...
#APP
  movl a, %eax
  addl $3, %eax
  movl %eax, res
#NO_APP
  movl res(%rip), %esi
  leaq .LC0(%rip), %rdi
  movq $0, %rax
  call printf
# ...
  movq $0, %rax
  popq %rbp
  ret
```

indirekter Speicher-zugriff

Mit **gcc -S** erzeugter Assemblercode zwischen **#APP** und **#NO_APP**.

Inline-Assembler – Erweiterte Version

Allgemeiner Syntax:

- **__asm__ __volatile__ (<codestrings>**
  [: <output-list>
  [: <input-list>
  [: <overwrite-list>]]]
  );

  Die Verwendung von **__volatile__** verhindert, dass der Compiler den Code wegoptimiert.

<codestrings>:

- Jede Assembler Codezeile (string line) muss in doppelte Anführungszeichen ("") gesetzt werden.
- Jede Codezeile (string line) (außer der letzten) muss mit den Steuerzeichen <RETURN> (**\n**) und <TAB> (**\t**) beendet werden.
- Jede **output-**, **input-** oder **owerwrite-list** wird mit einem Doppelpunkt (**:**) eingeleitet.

Inline-Assembler – Erweiterte Version

Beispiel:

```
__asm__ __volatile__(
    "movl a, %eax\n\t"
    "addl $3, %eax\n\t"
    "movl %eax, res");
```

Code aus Basisversion

C-Code:

```
#include <stdio.h>

int main(){

    int res = 0;
    int a = 11;

  __asm__ __volatile__(
    "movl %[a], %%eax\n\t"
    "addl $3, %%eax\n\t"
    "movl %%eax, %[res]\n\t"
    : [res] "=r" (res)   // output
    : [a] "r" (a)        // input
    : "%eax"             // override
  );
  printf("The result is %d\n", res);
  return 0;
}
```

lokale und globale Variablen möglich

Assembler-Code (Ausschnitt):

```
main:
  pushq   %rbp
  movq %rsp, %rbp
  movq    %rsp, %rbp
  subq    $16, %rsp
  movl    $0, -8(%rbp)
  movl    $11, -4(%rbp)
  movl    -4(%rbp), %edx
#APP
  movl %edx, %eax
  addl $3, %eax
  movl %eax, %edx
#NO_APP
  movl  %edx, %esi
  leaq .LC0(%rip), %rdi
  movl $0, %rax
  call printf

  movl $0, %eax
  leave # replaces "movq %rbp, %rsp"
        # and      "popq %rbp"
  ret
```

Mit **gcc -S** erzeugter Assemblercode zwischen **#APP** und **#NO_APP**.

Inline-Assembler – Erweiterte Version - output-list

Die **output-list** ist eine kommaseparierte Liste mit Elementen der Struktur "**[name] tag (expr)**"

output-list: `:[name] tag (expr)`

z.B.: `: [res] "=r" (res)  // output`

name:

- Die Namen (z.B. Variablen aus dem C-Programm) werden mit der Notation **%[name]** in die **string lines** des Assemblercodes geschrieben

z.B.: `"movl %%eax, %[res]\n\t"` ← **string line**

- Registernamen werden in den **string lines** mit einem zusätzlichen Prozentzeichen (**%%**) angegeben.

- In der **output-, input-** und **overwrite-list** entfällt das zusätzliche führende Prozentzeichen (für Variablen- und Registernamen).

z.B.:
```
:
: [a] "r" (a)  // input
: "%eax"       // override
```

Inline-Assembler – Erweiterte Version - output-list

Die **output-list** ist eine kommaseparierte Liste mit Elementen der Struktur "**[name] tag (expr)**"

output-list: **:[name] tag (expr)**

tag / constraint (Rahmenbedingung):

Constraint	Meaning
"=r"	Update value stored in a register
"+r"	Read and update value stored in a register
"=m"	Update value stored in memory
"+m"	Read and update value stored in memory
"=rm"	Update value stored in a register or in memory
"+rm"	Read and update value stored in a register or in memory

r = Register
m = Memory

update = veränderbar (also Schreibrechte)
read = Leserechte

Inline-Assembler – Erweiterte Version - output-list

Die **output-list** ist eine kommaseparierte Liste mit Elementen der Struktur "**[name] tag (expr)**"

output-list: **:[name] tag (expr)**

tag–Details – Teil 1:

Output Modifier	Description
+	The operand can be both read from and written to.
=	The operand can only be written to.

Wird nur angegeben, wenn die Möglichkeiten auch tatsächlich genutzt werden.
Wird weder + noch = angegeben, bestehen nur Leserechte.

Inline-Assembler – Erweiterte Version - output-list

Die **output-list** ist eine kommaseparierte Liste mit Elementen der Struktur "**[name] tag (expr)**"

output-list: **:[name] tag (expr)**

tag–Details – Teil 2:

Constraint	Description
a	Use the %eax, %ax, or %al registers.
b	Use the %ebx, %bx, or %bl registers.
c	Use the %ecx, %cx, or %cl registers.
d	Use the %edx, %dx, or $dl registers.
S	Use the %esi or %si registers.
D	Use the %edi or %di registers.
r	Use any available general-purpose register.
q	Use either the %eax, %ebx, %ecx, or %edx register.
A	Use the %eax and the %edx registers for a 64-bit value.

Inline-Assembler – Erweiterte Version - output-list

Die **output-list** ist eine kommaseparierte Liste mit Elementen der Struktur "**[name] tag (expr)**"

output-list: **:[name] tag (expr)**

tag–Details – Teil 3:

f	Use a floating-point register.
t	Use the first (top) floating-point register.
u	Use the second floating-point register.
m	Use the variable's memory location.
o	Use an offset memory location.
V	Use only a direct memory location.
i	Use an immediate integer value.
n	Use an immediate integer value with a known value.
g	Use any register or memory location available.

Für **xmm**, **ymm** bzw. **zmm** wird jeweils **x** verwendet, z.B **"=x"**.

Inline-Assembler – Erweiterte Version - output-list

Die **output-list** ist eine kommaseparierte Liste mit Elementen der Struktur "**[name] tag (expr)**"

output-list: **:[name] tag (expr)**

expr:

- Die **expr** ist typischerweise ein Variablennamen im C-Code.
  Die **expr** kann jedoch jeder in C zuweisbare Wert sein.

Beispiel:

```
#include <stdio.h>
int main() {
    int a = 10;
    int result;
    __asm__ (
        "movl %[input], %[output]\n\t"
        : [output] "=r" (result)
        : [input] "r" (a + 42)
    );
    printf("Das Ergebnis ist: %d\n", result);
    return 0;
}
```

Inline-Assembler – Erweiterte Version - input-list

Die **input-list** hat die gleiche Struktur wie die **output-list** mit folgenden Abweichungen:

input list - **:[name] tag (expr)**

- Die Elemente können nur die **tags** mit **Leserechten** besitzen (z.B. **"r"**, **"m"** or **"rm"**).

Schnittstelle zu C/C++ - Inline-Assembler – Erweiterte Version

Die **overwrite-list** wird manchmal auch **clobber-list** genannt.

Sie wird verwendet um dem Compiler mitzuteilen, wenn der Inline-Assembler-Code zusätzliche Register (oder zusätzlichen Speicher) nutzt, die **nicht** als **input** oder **output** **angegeben sind**.

Hinweis:

- Wenn das Conditioncode-Register (RFlags) verändert würde, muss man **"cc"** zur **overwrite-list** hinzufügen.
- **"memory"** teilt dem Compiler mit, dass der Assembler-Code Speicherlese- oder -schreibvorgänge an anderen Elementen als den in den Eingabe- und Ausgabeoperanden aufgeführten ausführt (z. B. Zugriff auf den Speicher, auf den einer der Eingabeparameter zeigt).

Inline-Assembler – Erweiterte Version - Limitierungen

Sprungziele dürfen nur im vorliegenden eigenen Assemblerblock liegen.

- Labelnamen müssen eindeutig sein
  -> Empfehlung: nur lokale Labels verwenden.
  Das sind Labels die nur aus Zahlen bestehen (z.B. **0:** oder **1:**)

- Der Compiler prüft den Code nicht auf Fehler
  -> Verwenden von **gcc -S** um sich anzusehen, wie der Compiler den Code eingebaut hat und ob das so gewünscht ist.

- Der eingebaute Code befindet sich in der Ausgabedatei zwischen den Kommentaren **#APP** und **#NO_APP**.

Inline-Assembler – Erweiterte Version

Aufgabe:

- Schreiben Sie Inline-Assembler-Code, der im nebenstehenden C-Code den Wert der Variable **para1** quadriert und in der Variablen **res** wieder zurückgibt.
- Halten Sie die Variablen **para1** und **res** in Registern.

```
#include <stdio.h>

int main()
{
  long res = 0;
  long para1 = -2;

  __asm__ __volatile__(
      _____________________
      _____________________
      _____________________
      : _____________________//output
      : _____________________//input
      : _____________________//override
    );

  printf("the result is %ld\n", res);
  return 0;
}
```

Lösung:

```c
#include <stdio.h>

int main() {
    long res = 0, para1 = 2;

    __asm__ __volatile__(
        "movq %[para1], %%rax\n\t"
        "imulq %[para1]\n\t"
        "movq %%rax, %[res]\n\t"
        : [res] "=r" (res)
        : [para1] "r" (para1)
        : "%rax", "%rdx"
    );

    /* %rdx is part of the multiplication
     * it does not stand for %[para], which
     * is selected by the assembler itself */

    printf("Result = %ld\n", res);

    return 0;
}
```

Original-Assemblercode

```
#APP
  movq %rcx, %rax
  imulq %rcx
  movq %rax, %rcx
#NO_APP
```

Inline-Assembler – Verpackt in eine C-Funktion (C-Stub)

Engl.: Wrapped in a C-function (**C-stub**)

- Ein **C-Stub** wird verwendet, wenn der Code mehrfach gebraucht wird
  -> Kopieren und einfügen des Inline-Assembler-Codes wird vermieden.

- Verwendung:

  C-Stub in einer Header-Datei ablegen und dort inkludieren, wo es benötigt wird und anschließend aufrufen.

Inline-Assembler – Verpackt in eine C-Funktion (C-Stub)

Beispiel:

C-Code:

x2.h

```
#ifndef X2_H
#define X2_H
int cadd3(int a) {
  __asm__ __volatile__(
    "addl $3, %[a]"
    : [a] "+r" (a) //output
    :              //input
    :              //override
  );
  return a;
}
#endif
```

main.c

```
#include <stdio.h>
#include "x2.h"

int main(){
  int res = 0;
  res = cadd3(11);
  printf("the result is %d\n", res);
  return 0;
}
```

Assembler-Code:

```
cadd3:
  movl %edi, %eax
#APP
  addl $3, %eax
#NO_APP
  ret
  .text
  .globl main
  .type main, @function
main:
  pushq %rbp
  movq %rsp, %rbp
  movl $11, %edi
  call cadd3
  movl %eax, %esi
  movl $.LC0, %edi
  movl $0, %eax
  call printf
  movq $0, %rax
  popq %rbp
  ret
```

Mit **gcc -S** erzeugter Assemblercode zwischen **#APP** und **#NO_APP**.

Inline-Assembler – Verpackt in eine C-Funktion (C-Stub)

Aufgabe:

- Schreiben Sie Inline-Assembler-Code, der im nebenstehenden C-Code den Wert der Variablen **para1** mit 3 potenziert und in der Variablen **res** wieder zurück gibt, damit sie in **main()** mit **printf()** ausgegeben werden kann.
- Halten Sie die Variablen **para1** und **res** in Registern
- Packen Sie den Inline-Assembler-Code in eine Funktion mit dem Namen **cube()**, die Sie in **main()** aufrufen.

```
#include <stdio.h>

____ cube(_________){
  ___________
  __asm__ __volatile__(
      ____________________
      ____________________
      ____________________
      ____________________
      :____________________ //output
      :____________________ //input
      :____________________ //override
    );
  return ____;
}

int main()
{
  long res = 0;
  long para1 = 2;
  res = _______________;
  printf("The result is %ld\n", res);
  return 0;
}
```

Lösung:

```c
#include <stdio.h>

long cube(long para1){
  long res = 0;
  __asm__ __volatile__(
    "movq %[para1], %%rax\n\t"
    "imulq %[para1]\n\t"
    "imulq %[para1]\n\t"
    "movq %%rax, %[res]"
    :[res] "=r" (res)
    : [para1] "r" (para1)
    : "%rax", "%rdx"
  );
  return res;
}
    /* %rdx is part of the multiplication
     * it does not stand for %[para], which
     * is selected by the assembler itself */

int main(){
  long para1 = 2, res = 0;
  res = cube(para1);
  printf("Result = %ld\n", res);
  return 0;
}
```

Original-Assemblercode

```
#APP
  movq %rcx, %rax
  imulq %rcx
  imulq %rcx
  movq %rax, %rcx
#NO_APP
```

Inline-Assembler – Verpackt in eine C-Funktion (C-Stub)

Präprozessor

Allgemein: **#<directive> <token>**

Wird vor dem Kompilieren ausgeführt und erzeugt eine neue Textdatei.

gcc –E <infile> > <outfile>

Beispiel:

```
//define constants
#define MAXEXP 10
//define macro function
#define LIMITCHECK(x,y) ( (x) > (y) ? (y) : (x) )

int power (double base, unsigned int exp, double* res)
{
  exp = LIMITCHECK(exp, MAXEXP);
  //... content of function
}
```

Inline-Assembler – Verpackt in eine C-Funktion (C-Stub)

- Das Verpacken (wrapping) des Inline-Assembler-Codes wird dann verwendet, wenn man auf den Overhead eines Funktionsaufrufs verzichten will / muss.
- Nachteil:
  - Bei häufigen Aufrufen vergrößert sich die ausführbare Datei.
- Verwendung:
  - Makro in einer Header-Datei ablegen und dort inkludieren, wo es benötigt wird und anschließend aufrufen.

Inline-Assembler – Verpackt in eine C-Funktion (C-Stub)

Beispiel:

C-Code:

```
#include <stdio.h>

#define cadd3(a) ({ \
  __asm__ __volatile__ ( \
     "addl $3, %[a]" \
    : [a] "+r" (a) \
    : \
    : \
  ); \
  a; })

int main(){
  int res = 0;
  int a = 11;
  res = cadd3(a);
  printf("the result is %d\n", res);
  return 0;
}
```

a; - damit das Resultat in main() wie bei einem normalen Funktionsaufruf übergeben werden kann.

Assembler-Code:

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

  subq $8, %rsp
  movl $11, %edx
#APP
  addl $3, %esi
#NO_APP
  movl $.LC0, %edi
  movl $0, %eax
  call printf
  movl $0, %eax
  addq $8, %rsp

  movq $0, %rax
  ret
```

Mit **gcc -S** erzeugter Assemblercode zwischen **#APP** und **#NO_APP**.