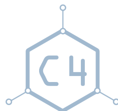
# U23 Assembler Workshop

Ike

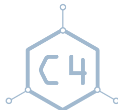
Chaos Computer Club Cologne e.V.
http://koeln.ccc.de

2016-11-05

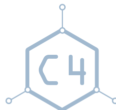


# Überblick

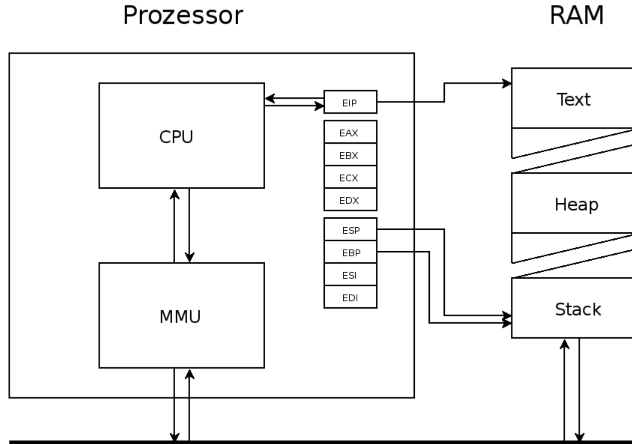

1. CPU, Assembler
   - Überblick
   - x86
   - x86 Assembler
   - Aufgaben
2. RAM, Stack, Calling Conventions
   - Stack
   - Calling Conventions
   - Stackframes
   - Aufgaben
3. Branches
   - Jumps
   - Vergleiche
   - Aufgaben
4. Speicher, C-Interface
   - C-Typen
   - main
   - Aufgaben
5. Intel Syntax
   - AT&T vs Intel

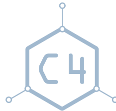



1. CPU, Assembler
   - Überblick
   - x86
   - x86 Assembler
   - Aufgaben
2. RAM, Stack, Calling Conventions
   - Stack
   - Calling Conventions
   - Stackframes
   - Aufgaben
3. Branches
   - Jumps
   - Vergleiche
   - Aufgaben
4. Speicher, C-Interface
   - C-Typen
   - main
   - Aufgaben
5. Intel Syntax
   - AT&T vs Intel



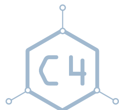
# CPU



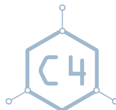
# Assembler

- Opcodes optimiert für Implementierung (CPU)
- Umsetzen von "menschenlesbaren" Befehlen zu Opcodes
- Anweisungen für vorhandene Hardware
- Nah an der Hardware
- Arbeiten auf Registern
- Feste Auswahl an Registern und Operationen
- Immer abhängig von der Hardware
- Bezeichnet sowohl den Compiler als auch die Sprache



# x86 Architektur (32-Bit)

- Gewachsener “Standard”
- Definierter Befehlssatz (Instruction Set)
- Definierte Register
- Mit zusätzlichen Erweiterungen
  - x87
  - Vektorregister (SSE, AVX, …)
  - …
  - AMD64



# Register

- Allgemeine Register

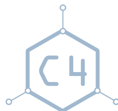
  eax Akkumulator
  ebx Basisregister
  ecx Counterregister
  edx Datenregister

- Pointerregister

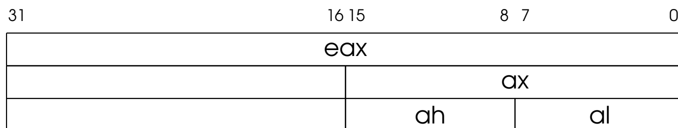
  esp Stackpointer
  ebp Stackbasispointer
  eip Instruktionspointer

- Indexregister

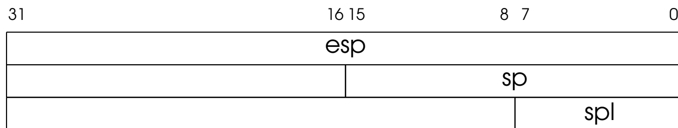
  esi Quellindex
  edi Zielindex



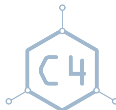
# Register




Genauso für `%ebx`, `%ecx`, `%edx`.



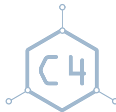

Genauso für `%ebp`, `%esi`, `%edi`.



# ASM Syntax (AT&T)

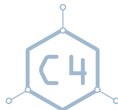
```
1  # Kommentare
2  func:              # Label (Funktionsname)
3  mov %edx, %eax     # Kopiere Wert (move)
4  mov $5, %eax       # Lade Konstante
5  add %ecx, %eax     # Addiere Werte
6  add $5, %eax       # Addiere Konstante
7  xchg %eax, %ebx    # Tausche Werte
8  ret                # return
```

- Ein Befehl pro Zeile
- Name zuerst (Mnemonic)
- Reihenfolge: Quelle, Ziel
- Register mit Prefix '%'
- Konstanten mit Prefix '$'



# Instruction Set Reference

- `http://www.intel.com/content/dam/www/public/us/en/documents/manuals/64-ia-32-architectures-software-developer-instruction-set-reference-manual-325383.pdf`
- Dokumentation aller Instruktionen
- Auch Parameter und Seiteneffekte



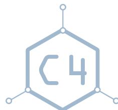
# Aufgaben

stub.c:

```c
#include <stdio.h>

int func(uint32_t, uint32_t)
  __attribute__((fastcall));

int main(void)
{
        printf("%i\n", func(5, 42));
        return 0;
}
```

asm.S:

```asm
.global func

func:
mov %edx, %eax
add %ecx, %eax
add $5, %eax
ret
```

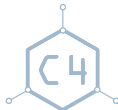


# Aufgaben

asm.S:

```
1  .global func
2
3  func:
4  mov %edx, %eax
5  add %ecx, %eax
6  add $5, %eax
7  ret
```

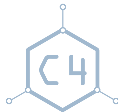
- 1. Parameter in `%ecx`
- 2. Parameter in `%edx`
- Rückgabewert in `%eax`



# Aufgaben

Führe das Beispielprogramm aus

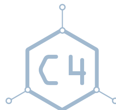
```
1 gcc -Wall -m32 -c stub.c
2 gcc -Wall -m32 -c asm.S
3 gcc -m32 -o main stub.o asm.o
```



# Aufgaben

Implementiere

```
uint32_t func(uint32_t p1, uint32_t p2) {
  • return 7;
  • return p1;
  • return p1 - p2;
  • return (p1 << 16) | (p2 & 0xffff);
  • return (p1 + p2) ^ (p1 & p2);
}
```

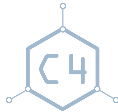




Ike <ike@koeln.ccc.de>

# Fragen

- Wofür sind die Register da?
- Was macht eip?
- Warum braucht man assembler?



1 CPU, Assembler
- Überblick
- x86
- x86 Assembler
- Aufgaben

2 RAM, Stack, Calling Conventions
- Stack
- Calling Conventions
- Stackframes
- Aufgaben

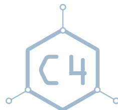
3 Branches
- Jumps
- Vergleiche
- Aufgaben

4 Speicher, C-Interface
- C-Typen
- main
- Aufgaben

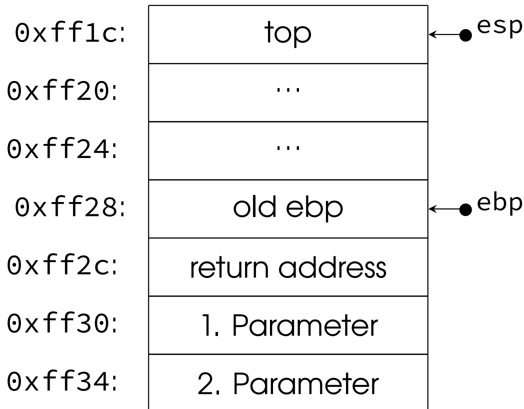
5 Intel Syntax
- AT&T vs Intel

# RAM

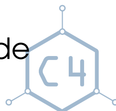
```
mov (%ebx), %eax    # Zugriff über Pointer
mov 4(%ebx), %eax   # Zugriff über Pointer mit Displacement

# Displacement ist  8 * ecx + 4
mov 4(%ebx, %ecx, 8), %eax

# Und in die andere Richtung
mov %eax, (%ebx)
mov %eax, 8(%ebx)

# RAM zu RAM geht nicht
# mov (%ebx), (%eax)
```



# Stack




- Verwendet für
  - Lokale Variablen
  - Funktionsparameter
  - Rücksprungadressen
- Dedizierter Speicherbereich
- Wächst von höheren zu niedrigeren Adressen
- `%esp` zeigt auf das obere Ende des Stack (Stack Pointer)



# Push / Pop

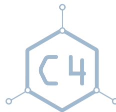
- Werte auf den Stack schieben / vom Stack holen

**Push** Verringert Stack Pointer, schreibt Wert nach (*%esp*)

**Pop** Lädt Wert von (*%esp*), erhöht Stack Pointer

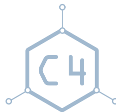
```
push %eax

pop %eax
```

```
sub $4, %esp
mov %eax, (%esp)

mov (%esp), %eax
add $4, %esp
```



# Calling Conventions

- Funktionsaufrufe geben Kontrolle ab
- Aufgerufene Funktion arbeitet auf derselben Hardware
- Keine Absicherung / Trennung von Variablen / Werten (Scope)
- Registerwerte können verändert werden
- Deshalb definierte Konvention
    - Welche Register dürfen verändert werden?
    - Wie werden Parameter übergeben?
    - Wie wird der Rückgabewert übergeben?
    - Wer räumt die Parameter vom Stack?

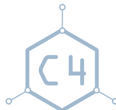




Ike <ike@koeln.ccc.de>

# cdecl Calling Convention

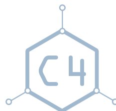
- `%eax`, `%ecx`, `%edx` caller-saved (dürfen von aufgerufenen Funktionen überschrieben werden)
- Alle anderen Register sind callee-saved (die aufgerufene Funktion muss die Werte speichern und wiederherstellen)
- Parameter werden, von rechts nach links, auf dem Stack übergeben
- Rückgabewert in `%eax` für Integer und Pointer
- Parameter werden von der aufrufenden Funktion aufgeräumt
- Normalerweise: Stack ist 16-Byte aligned
- Zusätzliche Regeln für z.B. Floats



# call Instruktion

```
func2:
# ...
push $0x1
# rufe func1(1) auf
call func1
# ...
push $0x2
# Indirekter Funktionsaufruf
call (%eax)
# ...
```

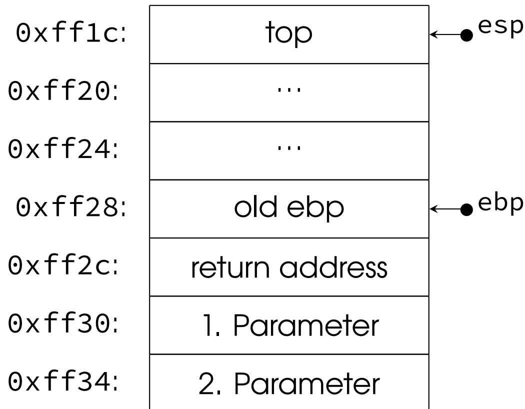
- Legt Rücksprungadresse auf den Stack
- Gibt Kontrolle an eine andere Funktion ab
- `ret` holt Rücksprungadresse vom Stack und gibt Kontrolle an diese Adresse zurück



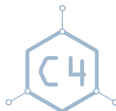


Ike <ike@koeln.ccc.de>

# Stackframe



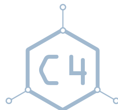

- Der Platz auf dem Stack, den eine Funktion benutzt
- Begrenzt durch `%esp` und (`%ebp`+4)
- Muss selbst verwaltet werden
- Am Anfang der Funktion `%ebp` aufsetzen
- Am Ende `%ebp` und `%esp` zurücksetzen



# Stackframe

```
func:
enter $0, $0
# ...
leave
ret

# enspricht:
func:
push %ebp
mov %esp, %ebp
# ...
mov %ebp, %esp
pop %ebp
ret
```

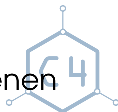
- Der Platz auf dem Stack, den eine Funktion benutzt
- Begrenzt durch `%ebp` und `%esp`
- Muss selbst verwaltet werden
- Am Anfang der Funktion `%ebp` aufsetzen
- Am Ende `%ebp` und `%esp` zurücksetzen



# Lokale Variablen

```
func:
push %ebp
mov %esp, %ebp
# Platz für 4 ints
sub $0x10, %esp

# ...

mov %eax, -4(%ebp)
mov -0xc(%ebp), %edx

# ...

leave  # Setzt %esp zurück
```

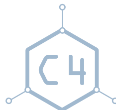
- Zuerst Platz für lokale Variablen schaffen (*%esp* verringern)
- Dann lokale Variablen und Parameter relativ zu (*%ebp*) benutzen
- Damit unabhängig von *%esp*
- *%ebp* muss nicht unbedingt benutzt werden
- Hilfreich für selbstgeschriebenen assembler



# Static Variablen

```
1   .data   # Was folgt sind Daten
2   mein_int:
3   .long 0x02
4   mein_string:
5   .string "foobar"
6
7   .text   # Jetzt wieder Code
8   meine_funktion:
9   enter $0, $0
10  push $mein_string   # Adresse
11  call printf
12  mov mein_int, %eax
13  leave
14  ret
```

- Label stehen für Adressen
- Von Funktionen (Code) und Variablen / Daten





Ike <ike@koeln.ccc.de>

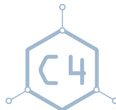
# Aufgaben

Implementiere

```
uint32_t func(uint32_t p1, uint32_t p2) {
```

- `return 7;`
- `return p1;`
- `return p1 - p2;`
- `return (p1 << 16) | (p2 & 0xffff);`
- `return (p1 + p2) ^ (p1 & p2);`

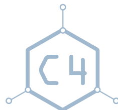
```
}
```



# Aufgaben

Implementiere

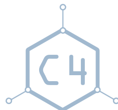
```
uint32_t func(uint32_t p1, ..., uint32_t p12) {
        return p1 + p2 + ... + p12;
}

void func(void) {
        printf("Hello World!");
}
```



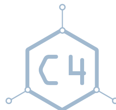
# Fragen

- Wofür braucht man den Stack?
- Was sind / wofür braucht man calling conventions?
- Was machen `enter`, `leave`, `call`, `ret`?
- Warum funktionieren

```
1 int main(void);
2 int main(int argc, char *argv[]);
3 int main(int argc, char *argv[], char *envp[]);
```



1. CPU, Assembler
   - Überblick
   - x86
   - x86 Assembler
   - Aufgaben
2. RAM, Stack, Calling Conventions
   - Stack
   - Calling Conventions
   - Stackframes
   - Aufgaben
3. Branches
   - Jumps
   - Vergleiche
   - Aufgaben
4. Speicher, C-Interface
   - C-Typen
   - main
   - Aufgaben
5. Intel Syntax
   - AT&T vs Intel

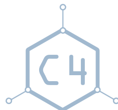



# Jump

Springe zu Labeln

```
func:
enter $0, $0
jmp end

# Wird nicht ausgeführt
add %ebx, %eax

end:
leave
ret
```

```
func:
enter $0, $0

# Endlosschleife
loop:
add %ebx, %eax
jmp loop

# Wird nie ausgeführt
leave
ret
```

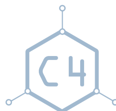


# Flags

- FLAGS Register
- Vordefinierte Flags
- Anhand derer können Entscheidungen getroffen werden

| | |
|---|---|
| Carry | Hardware-Überlauf (Unsigned) |
| Zero | Ergebnis == 0 |
| Sign | Sign-Bit |
| Overflow | Arithmetischer Signed-Überlauf |

- Werden von logischen / arithmetischen Operationen gesetzt
- Zum Beispiel add, sub, shifts, ...

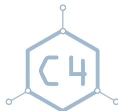


# Conditional Jump

```
1   # if ((eax + ebx) == 0) ecx = 23;
2   # else                  ecx = 42;
3
4   add %eax, %ebx
5   jz zero_case
6
7   mov $42, %ecx
8   jmp end
9
10  zero_case:
11  mov $23, %ecx
12
13  end:
14  # ...
```

- Springe nur falls Flag gesetzt ist

`jz` Sprung falls Zero Flag gesetzt

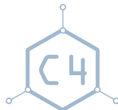


# Schleifen

```
1  mov $42, %ecx
2  loop:
3  # wird 42 Mal ausgeführt
4  add %eax, %ebx
5  dec %ecx
6  jnz loop
7
8  mov $42, %ecx
9  loop2:
10 # wird 42 Mal ausgeführt
11 add %eax, %ebx
12 loop loop2
```

- Springe solange `%ecx` ungleich 0

`loop` Kombiniert herunterzählen und bedingten Sprung





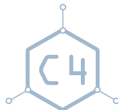
Ike <ike@koeln.ccc.de>

# cmp Instruktion

```
cmp %eax, %ebx
# Sprung falls eax == ebx
# 'jump if equal'
je some_label
```

`cmp` führt Subtraktion aus

- Speichert das Ergebnis nicht
- Setzt aber die gleichen Flags





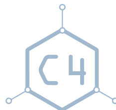
Ike <ike@koeln.ccc.de>

# test Instruktion

```
test %eax, %ebx
jz some_label

test %eax, %eax
# Sprung falls eax == 0
jz some_label
```

`test` führt and aus

- Speichert das Ergebnis nicht
- Setzt aber die gleichen Flags

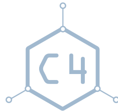



Ike <ike@koeln.ccc.de>

# Aufgaben

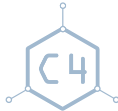
Implementiere

- Fibonacci
- FizzBuzz

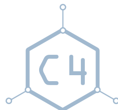


# Fragen

- Was ist der Unterschied zwischen `je` und `jz`?



1. CPU, Assembler
   - Überblick
   - x86
   - x86 Assembler
   - Aufgaben
2. RAM, Stack, Calling Conventions
   - Stack
   - Calling Conventions
   - Stackframes
   - Aufgaben
3. Branches
   - Jumps
   - Vergleiche
   - Aufgaben
4. Speicher, C-Interface
   - C-Typen
   - main
   - Aufgaben
5. Intel Syntax
   - AT&T vs Intel

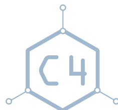


# Arrays

```
uint32_t a[6];
```

| | |
|---|---|
| 0xab00: | a[0] |
| 0xab04: | a[1] |
| 0xab08: | a[2] |
| 0xab0c: | a[3] |
| 0xab10: | a[4] |
| 0xab14: | a[5] |

- Elemente liegen nacheinander im Speicher





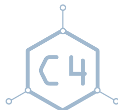
Ike <ike@koeln.ccc.de>

# Arrays

```
uint16_t a[12];
```

| | | |
|---|---|---|
| 0xab00: | a[0] | a[1] |
| 0xab04: | a[2] | a[3] |
| 0xab08: | a[4] | a[5] |
| 0xab0c: | a[6] | a[7] |
| 0xab10: | a[8] | a[9] |
| 0xab14: | a[10] | a[11] |

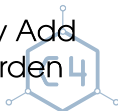
- Elemente liegen nacheinander im Speicher



# lea Instruktion

```
# ebx = eax * 4 - 8
lea -8(%eax, 4), %ebx

# ebx = edx + eax * 4 - 8
lea -8(%edx, %eax, 4), %ebx
```

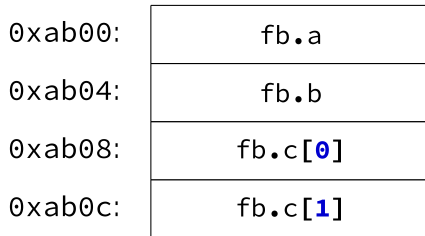
- Load Effective Address
- Berechnet Adresse mit Base Address, Displacement und Multiplier
- Die Adresse, auf die bei einer `mov` Instruktion zugegriffen würde
- Nicht gebunden an Semantik
- Kann auch als Fused Multiply Add (`a = b + c * d`) benutzt werden

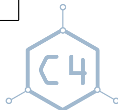


# C-Structs

Elemente liegen nacheinander im Speicher

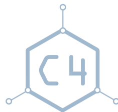
```
1 struct foobar {
2         uint32_t a;
3         float b;
4         uint32_t c[2];
5 };
6
7 struct foobar fb;
```



# Padding

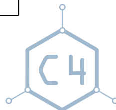
- C-Typen haben bestimmtes (minimales) Alignment
- Müssen gepaddet werden um das zu erfüllen
- Das Alignement eines Structs ist das maximale Alignment der Member



# Padding

```
1  struct foobar {
2          uint32_t a;
3          uint8_t b;
4          uint32_t c;
5          uint16_t d;
6  };
7
8  struct foobar fb;
```

| | | |
|---|---|---|
| 0xab00: | fb.a | |
| 0xab04: | fb.b | Padding |
| 0xab08: | fb.c | |
| 0xab0c: | fb.d | Padding |

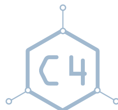


# C-Strings

- Arrays von **char**
- Nullterminiert

```c
char mein_string[] =
              "Hallo Welt!";
```

| | | | | |
|---|---|---|---|---|
| 0xab00: | 'H' | 'a' | 'l' | 'l' |
| 0xab04: | 'o' | ' ' | 'W' | 'e' |
| 0xab08: | 'l' | 't' | '!' | '\0' |



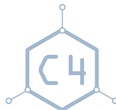


Ike <ike@koeln.ccc.de>

# main Funktion

```
1  main:
2  push %ebp
3  mov %esp, %ebp
4  mov 0xc(%ebp), %ecx
5  mov 0x4(%ecx), %eax
6  push %eax
7  call puts
8  leave
9  ret
```

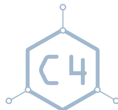
- Entspricht `main` in C
- Die gleichen Parameter



# Aufgaben

Implementiere

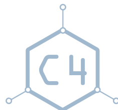
```
uint32_t sum_array(size_t size, uint32_t a[]) {
        uint32_t t = 0;
        for (size_t i=0; i<size; i++)
                t += a[i];
        return t;
}
```



# Aufgaben

Schreibe ein Programm in Assembler, dass

- die `argv[1]`-te Fibonaccizahl berechnet.
- die `1` Bits in `argv[1]` zählt.

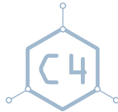




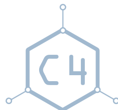
Ike <ike@koeln.ccc.de>

# Fragen

- Warum padded man structs am ende?

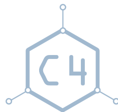


1. CPU, Assembler
   - Überblick
   - x86
   - x86 Assembler
   - Aufgaben
2. RAM, Stack, Calling Conventions
   - Stack
   - Calling Conventions
   - Stackframes
   - Aufgaben
3. Branches
   - Jumps
   - Vergleiche
   - Aufgaben
4. Speicher, C-Interface
   - C-Typen
   - main
   - Aufgaben
5. **Intel Syntax**
   - AT&T vs Intel

# Intel Asm

- Alternative Assembler Syntax
- Destination Operand zuerst, Source als zweites
- Register ohne `%`
- Konstante Werte ohne `$`
- Zugriff auf RAM mit eckigen Klammern
- Adresse von Labeln mit `offset(label)`
- Displacement ist irgendwie hübscher notiert
- Mit GAS: `.intel_syntax noprefix` am Anfang



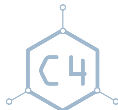
# Syntaxvergleich

## GAS (AT&T)

```
1  # Kommentare
2  enter $0, $0
3  mov 8(%ebp), %ecx
4  mov $0, %eax
5  mov $1, %ebx
6  loop1:
7  add %ebx, %eax
8  xchg %ebx, %eax
9  loop loop1
10 leave
11 ret
```

## NASM

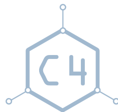
```
1  ; Kommentare
2  enter 0, 0
3  mov ecx, [ebp+8]
4  mov eax, 0
5  mov ebx, 1
6  loop1:
7  add eax, ebx
8  xchg eax, ebx
9  loop loop1
10 leave
11 ret
```



# Syntaxvergleich

```
1  .data
2  mein_int:
3  .long 0x02
4  mein_string:
5  .string "foobar"
6
7  .text
8  meine_funktion:
9  enter $0, $0
10 push $mein_string
11 call printf
12 mov mein_int, %eax
13 leave
14 ret
```

```
1  .data
2  mein_int:
3  .long 0x02
4  mein_string:
5  .string "foobar"
6
7  .text
8  meine_funktion:
9  enter 0, 0
10 push offset(mein_string)
11 call printf
12 mov eax, [mein_int]
13 leave
14 ret
```



# Syntaxvergleich

GAS (AT&T)

```
mov 8(%ebp, %eax, 4), %ecx
```

NASM

```
mov ecx, [ebp+eax*4+8]
```