

1.4. Calling-Conventions

Ein C-Programm möchte eine Assembler-Routine einbinden und dabei Parameter übergeben und einen Rückgabewert zurück bekommen.

Der Compiler der Hoch-Sprache (hier C) nutzt bestimmte Calling Conventions für Funktionsaufrufe, die die Assembler-Routine dann beachten muss.

Auch für den Aufruf von (Unter-) Routinen innerhalb eines Assembler-Programms muss der Programmierer (seine eigenen) Calling Conventions einhalten (z.B. bei Rekursion).

1. Beispiel: call und ret

```
get_int:  call read_int
          mov  [rbx], rax
          ret
```

ret entnimmt die **Rücksprungadresse** vom Stack (incl. pop)

```
mov  rbx, input1
call get_int
mov  rbx, input2
call get_int
```

call springt an die angegebene Stelle, hat die **Adresse des nächsten Befehls** auf den Stack gelegt (push).

Die **Übergabe von Parametern und Rückgabewerten** kann (beliebig) über Register und/oder den Stack realisiert werden.



Erläuterung des Beispiels

1. Beispiel: call und ret

```
SECTION .data
```

```
input1:    dd 0
input2:    dd 0
           dw 0
           db 0
```

```
SECTION .text
```

```
get_int:   call read_int
           mov  [rbx], rax
           ret
```

```
...
```

```
mov  rbx, input1
call get_int
mov  rbx, input2
call get_int
```

```
...
```

Textteil des Assembler-Programms,
das eigentliche Programm selbst.

Datenteil des Assembler-Programms,
z.B. Konstanten, globale Variablen, ...
db / .byte reserviert 1 Byte
dw / .word reserviert 2 Byte
dd / .double word reserviert 4 Byte
dq / .quad word reserviert 8 Byte

get_int ruft Funktion read_int auf,
read_int liefert ein Ergebnis in Register rax

get_int soll das Ergebnis in die Speicherstelle
schreiben, die in Register rbx steht

Vor Aufruf der Funktion get_int muss die
gewünschte Speicheradresse in rbx geladen
werden.
Hier: Adresse input1, dann Adresse input2
(mit unmittelbarer/„immediate“ Adressierung)



Mehr zum Datenteil

1. Beispiel: call und ret

NASM documentation, section 3.2

<https://nasm.us/doc/nasmdoc3.html#section-3.2>

```
SECTION .data
```

```
input1:    dd 0
input2:    dd 0
           dw 0
           db 0
```

```
SECTION .text
```

```
get_int:   call read_int
           mov  [rbx], rax
           ret
```

```
...
```

```
mov  rbx, input1
call get_int
mov  rbx, input2
call get_int
```

```
...
```

Datenteil des Assembler-Programms:

3.2 Pseudo-Instructions

Pseudo-instructions are things which, though not real x86 machine instructions, are used in the instruction field anyway because that's the most convenient place to put them. The current pseudo-instructions are DB, DW, DD, DQ, DT, DO, DY and DZ; their uninitialized counterparts RESB, RESW, RESD, RESQ, REST, RESO, RESY and RESZ; the INCBIN command, the EQU command, and the TIMES prefix.

In this documentation, the notation "Dx" and "RESx" is used to indicate all the DB and RESB type directives, respectively.

3.2.1 Dx: Declaring Initialized Data

DB, DW, DD, DQ, DT, DO, DY and DZ (collectively "Dx" in this documentation) are used, much as in MASM, to declare initialized data in the output file.

```
db  0x55          ; just the byte 0x55
db  0x55,0x56,0x57 ; three bytes in succession
db  'a',0x55       ; character constants are OK
db  'hello',13,10,'$' ; so are string constants
dw  0x1234         ; 0x34 0x12
dw  'a'           ; 0x61 0x00 (it's just a number)
dw  'ab'          ; 0x61 0x62 (character constant)
dw  'abc'         ; 0x61 0x62 0x63 0x00 (string)
dd  0x12345678     ; 0x78 0x56 0x34 0x12
dd  1.234567e20    ; floating-point constant
dq  0x123456789abcdef0 ; eight byte constant
dq  1.234567e20    ; double-precision float
dt  1.234567e20    ; extended-precision float
```

DT, DO, DY and DZ do not accept integer numeric constants as operands.

```
db  0x55
db  0x55,0x56,0x57
db  'a',0x55
db  'hello',13,10,'$'
dw  0x1234
dw  'a'
dw  'ab'
dw  'abc'
dd  0x12345678
dd  1.234567e20
dq  0x123456789abcdef0
dq  1.234567e20
dt  1.234567e20
```



Calling Conventions (allgemein)

Contents [hide]

- 1 Historical background
- 2 Caller clean-up
 - 2.1 cdecl
 - 2.2 syscall
 - 2.3 optlink
- 3 Callee clean-up
 - 3.1 pascal
 - 3.2 stdcall
 - 3.3 Microsoft fastcall
 - 3.4 Microsoft vectorcall
 - 3.5 Borland register
 - 3.6 Watcom register
 - 3.7 TopSpeed / Clarion / JPI
 - 3.8 safecall
- 4 Either caller or callee clean-up
 - 4.1 thiscall
- 5 Register preservation
 - 5.1 Caller-saved (volatile) registers
 - 5.2 Callee-saved (non-volatile) registers
- 6 x86-64 calling conventions
 - 6.1 Microsoft x64 calling convention
 - 6.2 System V AMD64 ABI
- 7 List of x86 calling conventions
- 8 References
 - 8.1 Footnotes
 - 8.2 Other sources
- 9 Further reading

Übersicht über Aufrufkonventionen – Calling Conventions

https://en.wikipedia.org/wiki/X86_calling_conventions

Die so genannte **cdecl-Aufrufkonvention** (C Declaration) wird von vielen C- und C++-Compilern verwendet, die auf der x86-Architektur laufen.

Die **AMD64 ABI-Aufrufkonvention** ist der de facto-Standard für x86-64-Systeme.



cdecl Calling Convention (32 bit) – C Declaration

Ziel: sicheren (reentranten) Aufruf einer Unter-Routine

- Parameterübergabe über den Stack (d.h. Aufrufer packt auf den Stack)
- Unter-Routine: Inhalt des Registers ebp (Basepointer) auf den Stack sichern
- sowie Inhalt des Stackpointers esp in Basepointer schreiben

Beispiel:

```
# void stars(int low, int high)
stars:
    # set up stack frame
    push ebp
    mov  ebp, esp
```

| | |
|---------|----------|
| | ebp + 12 |
| esp + 8 | ebp + 8 |
| esp + 4 | ebp + 4 |
| esp | ebp |

| |
|-------------------|
| ... |
| Parameter high |
| Parameter low |
| Rücksprungadresse |
| gerettetes ebp |

Der Stack wächst „von oben nach unten“,
in Richtung kleiner werdender Adressen

cdecl Calling Convention: Rückkehr

Vor der Rückkehr aus der Unter-Routine muss der ursprüngliche Zustand wieder hergestellt werden:

Beispiel:

```
# void stars(int low, int high)
stars:
    # set up stack frame
    push ebp
    mov  ebp, esp

...    # rest of function stars

    # reconstruct stack- and base-pointer
    leave

    # return
    ret
```

Die Instruktion leave bringt sowohl Stackpointer esp als auch Basepointer ebp in ihren ursprünglichen Zustand.



cdecl Calling Convention: Aufrufer

Der Aufrufer muss die Parameter auf den Stack legen und nach der Rückkehr die Parameter wieder entfernen:

So genanntes „Caller clean-up“

Beispiel:

```
# void stars(int low, int high)
```

```
# call stars(3,12)
```

```
push    12
```

```
push    3
```

```
call    stars
```

```
# adjust stack
```

```
add     esp, 8
```

Die Parameter werden **in umgekehrter Reihenfolge** auf den Stack gelegt.

Hier: zuerst <high>
 dann <low>

Der Aufruf der Assembler-Routine <stars> aus einem C-Programm (unter Linux mit gcc) funktioniert genau in dieser Weise!



cdecl Calling Convention: Lokale Variablen

Beispiel:

```
# void stars(int low, int high)
stars:
    # set up stack frame
    push ebp
    mov  ebp, esp
    # reserve space for local variable i
    sub  esp, 4

    ... # rest of function stars

    # reconstruct stack- and base-pointer
    leave

    # return
    ret
```

Ziel: Die Assembler-Routine möchte lokale Variablen verwenden. Diese sollen auch bei rekursiven Aufrufen lokal pro Aufruf existieren!

Lösung: Die lokalen Variablen werden auf dem Stack abgelegt!

| | |
|---------|----------|
| | ebp + 12 |
| | ebp + 8 |
| esp + 8 | ebp + 4 |
| esp + 4 | ebp |
| esp | ebp - 4 |

| |
|-------------------|
| ... |
| Parameter high |
| Parameter low |
| Rücksprungadresse |
| gerettetes ebp |
| lokale Variable i |



cdecl Calling Convention: Retten von Registern

Innerhalb der Routine wird der Basepointer ebp nicht mehr verändert.

- Zugriff auf Parameter
ebp + <Index>
- Zugriff auf lokale Variablen
ebp - <Index>

| | |
|---------|----------|
| | ebp + 12 |
| | ebp + 8 |
| esp + 8 | ebp + 4 |
| esp + 4 | ebp |
| esp | ebp - 4 |

| |
|-------------------|
| ... |
| Parameter high |
| Parameter low |
| Rücksprungadresse |
| gerettetes ebp |
| lokale Variable i |

Der Stackpointer kann sich noch durch weitere push-Instruktionen verändern!
(in entsprechender Anzahl muss pop verwendet werden)

Retten von Registern:

Die cdecl Calling Conventions nehmen an, dass die Register ebx, esi, edi, ebp, cs, ds, ss, es nicht in der Routine verändert werden.

Nicht dabei: eax, ecx, edx
(d.h. diese kann man bedenkenlos nutzen)

Werden diese Register benötigt, so müssen ihre Werte auf den Stack gesichert (push) und vor Verlassen der Routine wieder hergestellt werden (pop).

cdecl Calling Convention: Rückgabewert

- Sofern möglich, wird der Rückgabewert von C-Funktionen im **eax-Register** (32 Bit) übergeben (char, int, enum, Pointer, ...). Floating-Point Werte werden z.B. über ein Co-Processor-Register bzw. Spezialregister übergeben.
- Dies impliziert, dass der Aufrufer einer Unter-Routine keine wichtigen Werte im eax-Register gespeichert hat.

Name der Unter-Routine und Linker-Label:

- Der Name der Funktion im C-Programm ist identisch mit dem Label der Unter-Routine im Assembler-Programm. Beim Linken werden diese entsprechend miteinander verbunden (siehe auch Beispiele in 1.6.).

Nicht dabei: eax, ecx, edx (vgl. vorige Folie)
(d.h. diese kann man bedenkenlos nutzen)

Warum diese 3?



System V AMD64 ABI Calling Convention (64 bit)

Unterschiede zu cdecl:

- Parameterübergabe über Register `rdi`, `rsi`, `rdx`, `rcx`, `r8`, `r9`
 - wenn weitere Parameter benötigt werden, werden diese in umgekehrter Reihenfolge über den Stack übergeben
- Calling Convention nimmt an, dass Register `rbx`, `rsp`, `rbp`, `r12-r15` nicht von der aufgerufenen Funktion verändert werden
 - Inhalte müssen gegebenenfalls auf dem Stack gesichert (push) und am Ende der Unteroutine wiederhergestellt werden (pop)

Nicht dabei: `rax`, `rcx`, `rdx`, `rsi`, `rdi`, `r8-r11`
(d.h. diese kann man bedenkenlos nutzen)

Beispiel:

```
# void stars(int low, int high)
```

```
# call stars(3,12)
```

```
mov    rdi, 3
mov    rsi, 12
call   stars
```



Vergleich der Eigenschaften der Calling Conventions

| | cdecl (32 bit) | AMD64 ABI |
|--|---|---|
| Parameterübergabe | Alle Par. über den Stack (umgekehrte Reihenfolge) | Erste 6 Par.: in Registern rdi, rsi, rdx, rcx, r8, r9 7. und weitere Par. über den Stack |
| Zu rettende Register (bedenkenlos verwendbare Register) „Eselsbrücke“ | ebx, esi, edi, ebp (cs, ds, ss, es) eax, ecx, edx <i>“Registers EAX, ECX, and EDX are caller-saved, and the rest are callee-saved.”</i> | rbx, rsp, rbp, r12 – r15 rax, rcx, rdx, rsi, rdi, r8 – r11 rax, all six parameter registers and r10, r11 are caller-saved, and the rest are callee-saved. |
| Lokale Variablen | Auf dem Stack | Auf dem Stack |
| Rückgabewert | eax (Integer und Pointer) | rax (64 bit integer), rax + rdx (128 bit integer) |
| Stack Clean-Up | Caller Clean-Up (Aufrufer entfernt Par. vom Stack) | Caller Clean-Up (Aufrufer entfernt 7. und weitere Par. vom Stack) |



Besonderheiten der Calling Conventions

cdecl (32 bit)

AMD64 ABI

https://wiki.osdev.org/System_V_ABI

Details und weitere Quellen

Calling Convention

This is a short overview of the important [calling convention](#) details for the major System V ABI architectures. This is an incomplete account and you should consult the relevant processor supplement document for the details. Additionally, you can use the `-S` compiler option to stop the compilation process before the assembler is invoked, which lets you study how the compiler translates code to assembly following the relevant calling convention.

i386

This is a 32-bit platform. The stack grows downwards. Parameters to functions are passed on the stack in reverse order such that the first parameter is the last value pushed to the stack, which will then be the lowest value on the stack. Parameters passed on the stack may be modified by the called function. Functions are called using the `call` instruction that pushes the address of the next instruction to the stack and jumps to the operand. Functions return to the caller using the `ret` instruction that pops a value from the stack and jump to it. The stack is 4-byte aligned all the time, on older systems and those honouring the SYSV psABI. On some newer systems, the stack is additionally 16-byte aligned just before the call instruction is called (usually those that want to support SSE instructions); consult your manual (GNU/Linux on i386 has recently become such a system, but code mixing with 4-byte stack alignment-assuming code is possible).

Functions preserve the registers `ebx`, `esi`, `edi`, `ebp`, and `esp`; while `eax`, `ecx`, `edx` are scratch registers. The return value is stored in the `eax` register, or if it is a 64-bit value, then the higher 32-bits go in `edx`. Functions push `ebp` such that the `caller-return-eip` is 4 bytes above it, and set `ebp` to the address of the saved `ebp`. This allows iterating through the existing stack frames. This can be eliminated by specifying the `-fomit-frame-pointer` GCC option.

As a special exception, GCC assumes the stack is not properly aligned and realigns it when entering `main` or if the attribute `((force_align_arg_pointer))` is set on the function.

x86-64

This is a 64-bit platform. The stack grows downwards. Parameters to functions are passed in the registers `rdi`, `rsi`, `rdx`, `rcx`, `r8`, `r9`, and further values are passed on the stack in reverse order. Parameters passed on the stack may be modified by the called function. Functions are called using the `call` instruction that pushes the address of the next instruction to the stack and jumps to the operand. Functions return to the caller using the `ret` instruction that pops a value from the stack and jump to it. The stack is 16-byte aligned just before the call instruction is called.

Functions preserve the registers `rbx`, `rsp`, `rbp`, `r12`, `r13`, `r14`, and `r15`; while `rax`, `rdi`, `rsi`, `rdx`, `rcx`, `r8`, `r9`, `r10`, `r11` are scratch registers. The return value is stored in the `rax` register, or if it is a 128-bit value, then the higher 64-bits go in `rdx`. Optionally, functions push `rbp` such that the `caller-return-rip` is 8 bytes above it, and set `rbp` to the address of the saved `rbp`. This allows iterating through the existing stack frames. This can be eliminated by specifying the `-fomit-frame-pointer` GCC option.

Signal handlers are executed on the same stack, but 128 bytes known as the red zone is subtracted from the stack before anything is pushed to the stack. This allows small leaf functions to use 128 bytes of stack space without reserving stack space by subtracting from the stack pointer. The red zone is well-known to cause problems for x86-64 kernel developers, as the CPU itself doesn't respect the red zone when calling interrupt handlers. This leads to a subtle kernel breakage as the ABI contradicts the CPU behavior. The solution is to build all kernel code with `-mno-red-zone` or by handling interrupts in kernel mode on another stack than the current (and thus implementing the ABI).

Stack alignment !!!

= 32 bit

The stack is **4-byte aligned all the time**, on older systems and those honouring the SYSV psABI. On some newer systems, the stack is ... (!!!)

= 128 bit !

The stack is **16-byte aligned just before the call instruction is called**.

1.5. Bezug zur 2-Adressmaschine

In Systemnaher Informatik wurden die möglichen Addressformate und entsprechenden Adressmaschinen besprochen.

Folgendes Beispiel wurde dort mit der 3-, 2-, 1- sowie 0-Adressmaschine gelöst:

- Gegeben:** Werte **A, B, C** in den Speicherzellen **a, b und c** $\in \mathbb{N}$.
- Ziel:** Berechnung von **$D = (A - B \cdot C) \cdot (A + B \cdot C)$** und **Ablage in Zelle d.**
- Sonstiges:** Die Zellen **h_1, h_2, \dots** dürfen zur Abspeicherung von **Zwischenergebnissen** genutzt werden.
Die Inhalte der Zellen **a, b und c** dürfen **nicht verändert** werden.

(1.3.4.1.) Eine, zwei oder drei Adressen ?

Systemn. Inf.
Prof. Martini
SS 2023

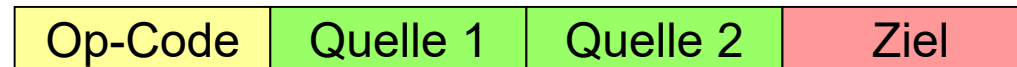
Prinzipiell kann ein Befehl **sehr viele Adressen** von Operanden beinhalten.

In der Praxis werden Befehle mit **sehr wenigen Adressen** verwendet, weil

- die **Länge der Adressen** in direktem Zusammenhang zur **Größe des** adressierbaren **Speicherbereiches** steht,
- **sehr lange Befehle keine effiziente Bearbeitung** zulassen.

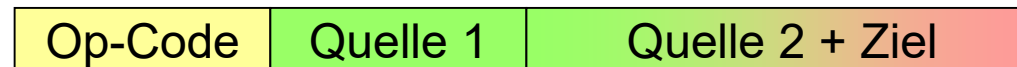
Die häufigsten Befehlsformen sind:

3-Adress-Format:



Gibt es ***NICHT***
beim **80x86 Assembl.**

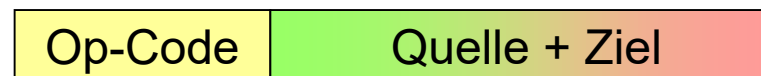
2-Adress-Format:



(„Überdeckung“: Der zweite Operand wird zerstört)

Gibt es beim
80x86 Assembler
z.B. add, sub, ...

1-Adress-Format:



(„Überdeckung“: Der Operand wird zerstört;
ggf. zusätzlich „Implizierung“)

Gibt es beim
80x86 Assembler
z.B. inc, dec, not,
...



(1.3.4.1.) Eine, zwei oder drei Adressen ?

Systemn. Inf.
Prof. Martini
SS 2023

- **Prinzipiell** kann ein Befehl **sehr viele Adressen** von Operanden beinhalten.
- **In der Praxis** werden Befehle mit **sehr wenigen Adressen** verwendet, weil
 - die **Länge der Adressen** in direktem Zusammenhang zur **Größe des** adressierbaren **Speicherbereiches** steht,
 - **sehr lange Befehle keine effiziente Bearbeitung** zulassen.

Die häufigsten Befehlsformen sind:

3-Adress-Format:



Gibt es ***NICHT***
beim 80x86 Assembl.

Doch!
Seit 80186 drei Varianten von „imul“ ...

Im WS08/09 zufällig gefunden ...

https://de.wikibooks.org/wiki/Assembler-Programmierung_f%C3%BCr_x86-Prozessoren/Rechnen_mit_dem_Assembler
<http://andremueller.gmxhome.de/befehle.html>

... oder: große Suchmaschine mit „assembler imul“ füttern ...

```
int 21h
section .data
zahl    DW 750h
ergebnis DD 0h
```

Wie Sie sehen, besitzt der Zeiger `zahl` das Präfix `word`. Dies benötigt der Assembler, um zu wissen, ob es sich um einen 8 Bit oder einen 16 Bit Operand handelt. Wenn der Zielloperand dagegen 16 Bit groß, muss der Assembler den Befehl in den Operandenmodus ändern. Ist der höherwertige Anteil des Ergebnisses 0, so werden Carryflag und Overflowflag und Parity Flag werden nicht verändert.

In der Literatur wird erstaunlicherweise oft "vergessen", dass der `imul` Befehl im 80186 CPU eingeführt wurde (und stellt daher eine der vielen Erweiterungen dar). Die erste Variante des `imul` Befehls entspricht der Syntax des `mul` Befehls:

```
imul Quelloperand
```

Eine Speicherstelle oder ein allgemeines Register wird entweder mit dem `AX` oder dem `EAX` Register multipliziert. Die zweite Variante des `imul` Befehls besitzt zwei Operanden und hat die folgende Syntax:

```
imul Zielloperand, Quelloperand
```

Der Zielloperand wird mit dem Quelloperand multipliziert und das Ergebnis im Zielloperand gespeichert. Der Zielloperand kann entweder ein Wert, ein allgemeines Register oder eine Speicherstelle sein.

Die dritte Variante des `imul` Befehls besitzt drei (!) Operanden. Damit widerlegt der Befehl die häufig geäußerte Aussage, dass die Befehle der Intel 80x86 Plattform entweder einen oder zwei Operanden besitzen können:

```
imul Zielloperand, Quelloperand1, Quelloperand2
```

Bei dieser Variante wird der erste Quelloperand mit dem zweiten Quelloperanden multipliziert und das Ergebnis im Zielloperanden gespeichert. Der Zielloperand und der erste Quelloperand müssen entweder ein allgemeines Register oder eine Speicherstelle sein, der zweite Quelloperand dagegen muss ein Wert sein.

Bitte beachten Sie, dass der DOS Debugger nichts mit dem Opcode der zweiten und dritten Variante anfangen kann, da er nur Befehle des 8086/88er Prozessors versteht. Weiterhin sollten Sie beachten, dass diese Varianten nur für den `imul` Befehl existieren - also auch nicht für `idiv` Befehl.

Die Division

[Bearbeiten]

Vorzeichenbehaftete Multiplikation

```
imul Faktor
imul Ziel, Faktor1, Faktor2      ;ab 80186
imul Ziel, Faktor                ;ab 80186 (ab 80386 erweitert)
```

IMUL führt eine Integer-Multiplikation unter Berücksichtigung des Vorzeichens durch.

Wie Sie sehen können, gibt es drei verschiedene Formen. Bei den Formen mit mehr als einem Operanden müssen die Operanden die gleiche Größe haben. Es sind Byte-, Word- und DWord-Operanden erlaubt.

Wird nur ein Operand verwendet (die erste Variante), dann wird implizit abhängig von der Operandengröße das Register `AL`, `AX` oder `EAX` verwendet. Bei der Variante mit drei Operanden werden Operand 2 und 3 miteinander multipliziert und das Ergebnis in den ersten Operanden geschrieben.

Die folgende Tabelle zeigt, wie bei den unterschiedlichen Varianten die Ablage des Ergebnisses erfolgt und welche Operandenkombinationen möglich sind. Da als Operanden sowohl Register als auch Konstanten sowie Speicherstellen möglich sind, sind an den Stellen, an denen für einen Operanden verschiedene Typen möglich sind, die Typen durch einen Schrägstrich getrennt aufgeführt.

| Befehl | IMUL-Varianten | Ergebnis |
|------------------------------|--|----------|
| IMUL reg8/mem8 | implizite Verwendung von <code>AL</code> , Ergebnis in <code>AX</code> | |
| IMUL reg16/mem16 | implizite Verwendung von <code>AX</code> , Ergebnis in <code>DX:AX</code> | |
| IMUL reg32/mem32 | implizite Verwendung von <code>EAX</code> , Ergebnis in <code>EDX:EAX</code> | |
| IMUL reg16,reg16/mem16 | Ergebnis im ersten Operanden | |
| IMUL reg32,reg32/mem32 | Ergebnis im ersten Operanden | |
| IMUL reg16,reg16/mem16,con8 | Ergebnis im ersten Operanden | |
| IMUL reg32,reg32/mem32,con8 | Ergebnis im ersten Operanden | |
| IMUL reg16,con8 | Ergebnis im ersten Operanden | |
| IMUL reg32,con8 | Ergebnis im ersten Operanden | |
| IMUL reg16,reg16/mem16,con16 | Ergebnis im ersten Operanden | |
| IMUL reg32,reg32/mem32,con32 | Ergebnis im ersten Operanden | |
| IMUL reg16,con16 | Ergebnis im ersten Operanden | |
| IMUL reg32,con32 | Ergebnis im ersten Operanden | |

Ist das Ergebnis von `IMUL` für das Ziel zu groß, dann werden das Overflow- und das Carry-Flag gesetzt, ansonsten werden beide gelöscht.

imul

Hervorheben Groß-/Kleinschreibung

Beispiel: Problemlösung mit 2-Adressmaschine

Systemn. Inf.
Prof. Martini
SS 2023

Bei einer 2-Adressmaschine gibt es für die Speicherung des Ergebnisses zweistelliger Rechenoperationen **drei Möglichkeiten**:

$\alpha := \rho(x) \text{ op } \rho(y)$

$\rho(x) := \rho(x) \text{ op } \rho(y)$

$\rho(y) := \rho(x) \text{ op } \rho(y)$

Implizierung

Überdeckung

Gibt es ***NICHT***
beim 80x86 Assembl.

Gibt es beim
80x86 Assembler
teilw. mit Register A + D

Lösung mittels 2-Adressmaschine

i: $\alpha := \rho(b) \cdot \rho(c);$

i+1: $\rho(h_1) := \alpha;$

i+2: $\alpha := \rho(a) - \rho(h_1);$

i+3: $\rho(d) := \alpha;$

i+4: $\rho(h_1) := \rho(a) + \rho(h_1);$

i+5: $\rho(d) := \rho(h_1) \cdot \rho(d);$

speichert $B \cdot C$ in den Akkumulator

sichert den Akkumulator nach Zelle h_1

speichert $A - B \cdot C$ in den Akkumulator

sichert den Akkumulator nach Zelle d

speichert $A + B \cdot C$ nach h_1

speichert D nach d

Anmerkung:

Wir gehen hier davon aus, dass bei Operationen mit zwei Speicherzellen entweder die zweite Speicherzelle oder der Akkumulator überschrieben wird.

Lösung mittels 80x86 Assembler

Lösung mittels 2-Adressmaschine

i: $\alpha := \rho(b) \cdot \rho(c);$

i+1: $\rho(h_1) := \alpha;$

i+2: $\alpha := \rho(a) - \rho(h_1);$

i+3: $\rho(d) := \alpha;$

i+4: $\rho(h_1) := \rho(a) + \rho(h_1);$

i+5: $\rho(d) := \rho(h_1) \cdot \rho(d);$

Achtung:
Dieses Beispiel berücksichtigt nicht korrekte Wertebereiche bei der Multiplikation!

Übertragung in Assembler:

```
; bei mul muss Acc. rax vorkommen
mov  rax, [b]      ; b in Acc.
mul  [c]           ; Acc. = Acc.*c
mov  [h1], rax     ; Acc. in h1
```

```
; Implizierung geht nicht!
mov  rax, [a]      ; a in Acc.
sub  rax, [h1]     ; a-b*c in Acc.
mov  [d], rax      ; Acc. in d
```

```
; Op. auf 2 Speicherzellen nicht möglich
mov  rax, [a]      ; a in Acc.
add  [h1], rax     ; a+b*c in h1
```

```
; bei mul muss Acc. rax vorkommen
mov  rax, [d]      ; d in Acc.
mul  [h1]          ; Acc. = Acc.*h1
mov  [d], rax      ; Endergebnis in d
```



Besondere arithmetische Operationen – add, sub, mul

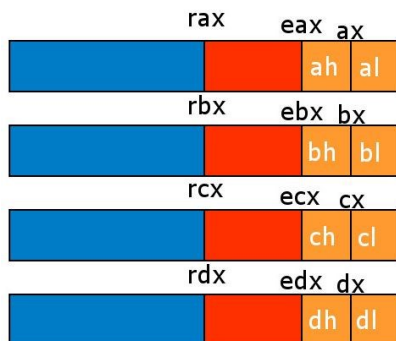
Manche Befehle sind mit beliebigen Operatoren ausführbar:

| ARITHMETIC | | Code | Operation | Flags | | | | | | | | |
|------------|----------------------|-----------------|------------------------|-------|---|---|---|---|---|---|---|---|
| Name | Comment | | | O | D | I | T | S | Z | A | P | C |
| ADD | Add | ADD Dest,Source | Dest:=Dest+Source | ± | | | | ± | ± | ± | ± | ± |
| ADC | Add with Carry | ADC Dest,Source | Dest:=Dest+Source+CF | ± | | | | ± | ± | ± | ± | ± |
| SUB | Subtract | SUB Dest,Source | Dest:=Dest-Source | ± | | | | ± | ± | ± | ± | ± |
| SBB | Subtract with borrow | SBB Dest,Source | Dest:=Dest-(Source+CF) | ± | | | | ± | ± | ± | ± | ± |

Operatoren von add und sub können Register, Speicher oder Konstanten sein (**jedoch nur maximal ein Operand kann auf Speicher zugreifen**). Das Ergebnis der Operation beeinflusst die Status-Flags.

Manche Befehle beziehen spezielle Register fest ein:

| | | | | | | | | | | | | | |
|---------|---------------------|--------|----------------------------|------------|---|--|--|--|---|---|---|---|---|
| MUL | Multiply (unsigned) | MUL Op | Op=byte: AX:=AL*Op | if AH=0 ♦ | ± | | | | ? | ? | ? | ? | ± |
| MUL | Multiply (unsigned) | MUL Op | Op=word: DX:AX:=AX*Op | if DX=0 ♦ | ± | | | | ? | ? | ? | ? | ± |
| MUL 386 | Multiply (unsigned) | MUL Op | Op=double: EDX:EAX:=EAX*Op | if EDX=0 ♦ | ± | | | | ? | ? | ? | ? | ± |



Bei der Multiplikation kann das Ergebnis den Wertebereich des Zielregisters (Accu A, byte/word/long) deutlich überschreiten.

Bei Operation der Länge byte wird ah als Overflowregister genutzt, sonst wird Datenregister D (word dx, long edx, quad rdx) als Overflow benutzt. Statusflags O und C zeigen Overflow an.



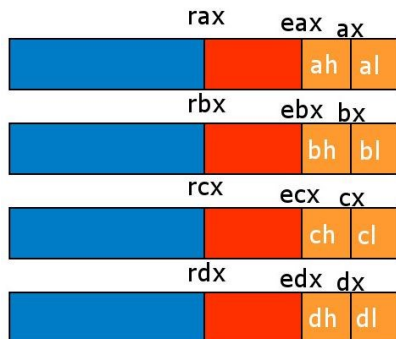
Besonderheiten Multiplikation

Manche Befehle beziehen **spezielle Register** fest ein:

| | | | | | | | | | | | | |
|---------------|-------------------------|---------|--|---|--|--|--|---|---|---|---|---|
| MUL | Multiply (unsigned) | MUL Op | Op=byte: AX:=AL*Op if AH=0 ♦ | ± | | | | ? | ? | ? | ? | ± |
| MUL | Multiply (unsigned) | MUL Op | Op=word: DX:AX:=AX*Op if DX=0 ♦ | ± | | | | ? | ? | ? | ? | ± |
| MUL 386 | Multiply (unsigned) | MUL Op | Op=double: EDX:EAX:=EAX*Op if EDX=0 ♦ | ± | | | | ? | ? | ? | ? | ± |
| IMUL <i>i</i> | Signed Integer Multiply | IMUL Op | Op=byte: AX:=AL*Op if AL sufficient ♦ | ± | | | | ? | ? | ? | ? | ± |
| IMUL | Signed Integer Multiply | IMUL Op | Op=word: DX:AX:=AX*Op if AX sufficient ♦ | ± | | | | ? | ? | ? | ? | ± |
| IMUL 386 | Signed Integer Multiply | IMUL Op | Op=double: EDX:EAX:=EAX*Op if EAX sufficient ♦ | ± | | | | ? | ? | ? | ? | ± |

♦ then CF:=0, OF:=0 else CF:=1, OF:=1

64 bit: Op=quad: RDX:RAX:=RAX*Op



Bei der Multiplikation kann das **Ergebnis den Wertebereich** des Zielregisters (Accu A, byte/word/long) deutlich **überschreiten**.

Bei Operation der Länge byte wird ah als **Overflowregister** genutzt, sonst wird Datenregister D (word dx, long edx, quad rdx) als Overflow benutzt. Statusflags O und C zeigen Overflow an.



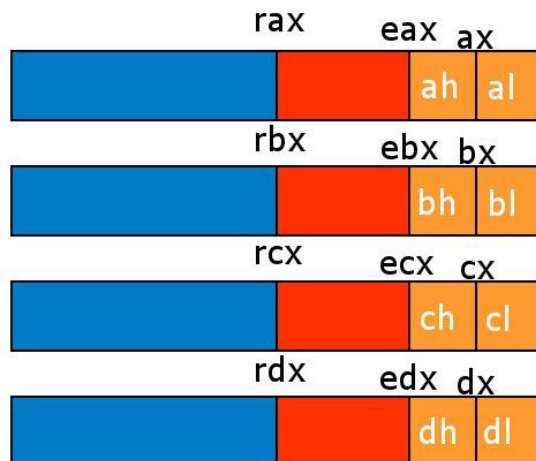
Besondere arithmetische Operationen (2) – div

Manche Befehle beziehen **spezielle Register fest** ein:

| ARITHMETIC | | Code | Operation | Flags | | | | | | | | |
|------------|-------------------|--------|--|-------|---|---|---|---|---|---|---|---|
| Name | Comment | | | O | D | I | T | S | Z | A | P | C |
| DIV | Divide (unsigned) | DIV Op | Op=byte: AL:=AX / Op AH:=Rest | ? | | | | ? | ? | ? | ? | ? |
| DIV | Divide (unsigned) | DIV Op | Op=word: AX:=DX:AX / Op DX:=Rest | ? | | | | ? | ? | ? | ? | ? |
| DIV 386 | Divide (unsigned) | DIV Op | Op=doublew.: EAX:=EDX:EAX / Op EDX:=Rest | ? | | | | ? | ? | ? | ? | ? |

Bei der Division kann der **Dividend einen größeren Wertebereich** aufweisen als das Ergebnis betragen kann.

Entsprechend kann der Dividend nicht nur allein aus dem Accu A gelesen werden, sondern:



- byte: Dividend ax (= ah:al), Ergebnis al
- word: Dividend dx:ax, Ergebnis ax
- long: Dividend edx:eax, Ergebnis eax
- quad: Dividend rdx:rax, Ergebnis rax

Die Division wird als **ganzzahlige Division** ausgeführt. Zusätzlich wird der **Rest im Hilfsregister** gespeichert (ah, dx, edx, rdx)

Besonderheiten Division

Manche Befehle beziehen **spezielle Register** fest ein:

| ARITHMETIC | | Code | Operation | Flags | | | | | | | | |
|------------|-------------------|--------|----------------------------------|-------|---|---|---|---|---|---|---|---|
| Name | Comment | | | O | D | I | T | S | Z | A | P | C |
| DIV | Divide (unsigned) | DIV Op | Op=byte: AL:=AX / Op AH:=Rest | ? | | | | ? | ? | ? | ? | ? |
| DIV | Divide (unsigned) | DIV Op | Op=word: AX:=DX:AX / Op DX:=Rest | ? | | | | ? | ? | ? | ? | ? |
| DIV 386 | | | | | | | | | | | | |

Es kann passieren, dass das Ergebnis der Division „zu groß“ für das Zielregister ist.

Bei d
als d

Beispiel: word: Divisor und Ergebnis 16 bit

Entsp
sond

Dividend dx:ax ; ist eine große Zahl

Divisor cx ; ist eine kleine Zahl

div cx ; dx:ax / cx

Ergebnis ax ; kann größer als 16 Bit sein

https://de.wikibooks.org/wiki/Assembler-Programmierung_f%C3%BCr_x86-Prozessoren/_Rechnen_mit_dem_Assembler#Die_Division

„Wenn die CPU auf einen solchen Überlauf stößt, löst sie eine *Divide Error Exception* aus. Dies führt dazu, dass der Interrupt 0 aufgerufen und eine Routine des Betriebssystems ausgeführt wird. Diese gibt die Fehlermeldung „Überlauf bei Division“ aus und beendet das Programm. Auch die Division durch 0 führt zum Aufruf der *Divide Error Exception*.

Wenn der Prozessor bei einem Divisionsüberlauf und bei einer Division durch 0 eine Exception auslöst, **welche Bedeutung haben dann die Statusflags?** Die Antwort lautet, dass sie bei der Division tatsächlich **keine Rolle** spielen, da sie keinen definierten Zustand annehmen.“

1.6. Ein (weiteres) Assembler-Programmbeispiel

Beispiel 2: Programm „stars“ komplett in C

```
#include <stdlib.h>
#include <stdio.h>
#include <unistd.h>
```

```
void stars(int low, int high);
void printl(int n);
```

Deklaration von Funktionen
<stars> und <printl>

```
int main(int argc, char **argv)
{
    int low;
    int high;

    if (argc != 3) {
        fprintf(stderr, „\nUsage: %s low high\n\n“, argv[0]);
        exit(1);
    }

    low = atoi(argv[1]);
    high = atoi(argv[2]);

    stars(low, high);

    return 0;
}
```



Beispiel 2: Programm „stars“ komplett in C

```
void stars(int low, int high)
```

```
{
```

```
    int i = low;
```

```
    do {
```

```
        printf(i);
```

```
        i++;
```

```
    }
```

```
    while (i <= high);
```

```
}
```

```
void printf(int n)
```

```
{
```

```
    int i = n;
```

```
    do {
```

```
        write(1, "*", 1);
```

```
        i--;
```

```
    }
```

```
    while (i > 0);
```

```
    write(1, "\n", 1);
```

```
}
```

Definition von Funktionen

<stars> und <printf>

; Funktion <stars> nutzt Funktion <printf>, Schleife läuft aufwärts

; Schleife in printf läuft abwärts



Beispiel 2: Programm „stars“ in C + Assembler (C-Teil)

```
#include <stdlib.h>
#include <stdio.h>

#include "stars.h"           ; <stars> einbinden

int main(int argc, char **argv)
{
    int low;
    int high;

    if (argc < 3) {
        fprintf(stderr, „\nUsage: %s low high\n\n“, argv[0]);
        exit(1);
    }

    low = atoi(argv[1]);
    high = atoi(argv[2]);

    stars(low, high);       ; Aufruf von Funktion <stars>

    return 0;
}
```



Beispiel 2: Programm „stars“ in C + Assembler (stars.h)

```
#ifndef _STARS_H
#define _STARS_H

void stars(int low, int high);

#endif
```

Beispiel 2: Programm „stars“ in C + Assem. (stars.asm)

Teil 1

```

;,,write“ macro
%macro write 3
    mov rax, 1
    mov rdi, %1
    mov rsi, %2
    mov rdx, %3
    syscall
%endmacro

; stars global verfügbar machen

global stars

SECTION .data

; Start des Datensegments

starChar:    db "*"
starCharLen: equ $ - starChar
newLineChar: db 10
newLineCharLen: equ $ - newLineChar

```

; Definition von Makro write
; (Makro in NASM Syntax)

; Zeichen * und /n definieren und labeln



Beispiel 2: Programm „stars“ in C + Assembler (Ass.-Teil)

Teil 2

SECTION .text

; Start des Textsegments

stars:

push rbp

; Set up stack frame

mov rbp, rsp

mov rcx, rdi

; Hole 2 Parameter

mov rbx, rsi

outerWhile:

mov rdi, rcx

; Parameter für printf in rdi, rette rcx

push rcx

call printf

; Aufruf printf (mit korrekten Calling Conventions!)

pop rcx

add rcx, 1

cmp rbx, rcx

; Schleife läuft aufwärts (wie in C-Version)

jae outerWhile

leave

ret



Beispiel 2: Programm „stars“ in C + Assembler (Ass.-Teil)

Teil 3

```

printl:
    push rbp                ; Set up stack frame
    mov rbp, rsp

    mov rcx, rdi            ; Hole einzigen Parameter
    innerWhile:             ; Rette (wieder) rcx, auch bei Macro (wg. syscall)
        push rcx
        write 1, starChar, starCharLen
        pop rcx             ; Schleife läuft abwärts (loop), wie in C-Version
        loop innerWhile
    write 1, newLineChar, newLineCharLen

    leave
    ret





```

Was macht eigentlich das Programm bzw. die Routine stars(low, high)?

(selbst) ausprobieren!

Ein .ZIP mit den Source-Files liegt auf der SysProg Webseite!

Name

-  main.c
-  stars.asm
-  stars.h
-  stars_komplett.c

Programm “stars” – Compilieren und Starten

Beispiel „stars“:

Ein C-Programm möchte eine Assembler-Routine einbinden.

| Aufruf | Erläuterung |
|---|---|
| <code>nasm -f elf64 -o stars.o stars.asm</code> | Aufruf des Assemblers Assembler-Teil in Datei <stars.asm> (oder auch <stars.S>) |
| <code>clang -o main stars.o main.c</code> | Aufruf des C-Compilers clang mit gleichzeitigem Linken von stars.o C-Programm-Teil in Datei <main.c> |
| <code>./main</code> | Aufruf des C-Programms inklusive Assembler-Routine |

1.7. Socket-/Netzwerkprogrammierung in Assembler

Auch aus den sehr maschinennahen Assembler-Programmen ist die **Nutzung von Systemfunktionen zur Netzwerkprogrammierung** möglich.

Die Netzwerkprogr./Assembler wollen wir uns im Rahmen von SysProg ersparen ;-)

Prinzipiell können aber **beliebige Betriebssystem-Funktionen** aus Assembler-Programmen aufgerufen werden (wie bereits gesehen):

```
# „write“ macro
%macro write 3
    mov     rax, 1
    mov     rdi, %1
    mov     rsi, %2
    mov     rdx, %3
    syscall
%endmacro
```

Funktion write: rax=1
rdi, rsi, rdx Parameter fd, buf, len

Linux System Call Table for x86

Refer to the syscall numbers in [arch/x86/entry/syscalls/syscall_64.tbl](http://arch.x86/entry/syscalls/syscall_64.tbl) to determine if the table below is out of date

By the way, the system call numbers are different for 32-bit x86. A system call table for i386 (32-bit) can be found at http://docs.cs.up.ac.za/programming/asm/derick_tut/syscalls.htm

Information on the order of registers can be found on page 124 of the x86_64 ABI paper at <http://www.x86-64.org/documentation/abi.pdf>

| %rax | System call | %rdi | %rsi | %rdx | %r10 | %r8 |
|------|--------------------|-----------------------|-----------------------------|------------------------|---------------------|------------------|
| 0 | sys_read | unsigned int fd | char *buf | size_t count | | |
| 1 | sys_write | unsigned int fd | const char *buf | size_t count | | |
| 2 | sys_open | const char *filename | int flags | int mode | | |
| 3 | sys_close | unsigned int fd | | | | |
| 4 | sys_stat | const char *filename | struct stat *statbuf | | | |
| 5 | sys_fstat | unsigned int fd | struct stat *statbuf | | | |
| 6 | sys_lstat | const char *filename | struct stat *statbuf | | | |
| 7 | sys_poll | struct poll_fd *ufds | unsigned int nfds | long timeout_msecs | | |
| 8 | sys_lseek | unsigned int fd | off_t offset | unsigned int origin | | |
| 9 | sys_mmap | unsigned long addr | unsigned long len | unsigned long prot | unsigned long flags | unsigned long fd |
| 10 | sys_mprotect | unsigned long start | size_t len | unsigned long prot | | |
| 11 | sys_munmap | unsigned long addr | size_t len | | | |
| 12 | sys_brk | unsigned long brk | | | | |
| 13 | sys_rt_sigaction | int sig | const struct sigaction *act | struct sigaction *oact | size_t sigsetsize | |
| 14 | sys_rt_sigprocmask | int how | sigset_t *nset | sigset_t *oset | size_t sigsetsize | |
| 15 | sys_rt_sigreturn | unsigned long _unused | | | | |
| 16 | sys_ioctl | unsigned int fd | unsigned int cmd | unsigned long arg | | |
| 17 | sys_pread64 | unsigned long fd | char *buf | size_t count | loff_t pos | |
| 18 | sys_pwrite64 | unsigned int fd | const char *buf | size_t count | loff_t pos | |

Weitere Informationen unter :

https://blog.rchapman.org/posts/Linux_System_Call_Table_for_x86_64/



1.8. ARM Assembly

Der erste ARM-Prozessor wurde 1985 auf den Markt gebracht, 2013 erschienen dann die ersten 64-Bit-Prozessoren. Die meisten Smartphones nutzen ARM-Prozessoren.

Unterschiede zur x86-64-Assembly:

- RISC statt CISC:
 - ARM ist ein **R**educed-**I**nstruction-**S**et-**C**omputing-Prozessor. Das bedeutet, dass deutlich weniger Befehle, dafür aber mehr allgemein verfügbare Register zur Verfügung stehen als beim **C**omplex **I**nstruction **S**et **C**omputing.
 - Dadurch können einzelne Befehle schneller ausgeführt werden.
- ARM-Befehle arbeiten nur auf Registern und nutzen Load/Store-Befehle für den Speicherzugriff.
- Der ARM-Befehlssatz nutzt 31 allgemein verfügbare 64-bit-Register X0-X30.

„Hello World“ in ARM-Assembly:

```
.text
.global _start
_start:  mov r0, #1
         ldr r1, =message
         ldr r2, =len
         mov r7, #4
         swi 0

         mov r7, #1
         swi 0

.data
message: .asciz "hello world\n"
len = .-message
```

1.9. MIPS Assembly

Wie bei ARM handelt es sich bei der MIPS-Architektur um einen RISC-Prozessor.

- Es gibt 3 Befehlsformate:
 - **R-Befehle:** `instruction rd, rs, rt`
 - rs und rt sind Quell-Register
 - rd ist das Zielregister
 - **I-Befehle:** `instruction rt, rs, imm`
 - rs ist das Quellregister
 - rt ist das Zielregister
 - imm ist ein 16-bit „immediate“-Wert
 - **J-Befehle:** `instruction addr`
 - springt an die angegebene Adresse
 - Adresse ist absolut und liegt in einem Bereich von 256MB

„Hello World“ in MIPS-Assembly:

```
.data
message: .asciiz „\nHello World!\n“

.text
main:    li $v0, 4
         la $a0, message
         syscall

         li $v0, 10
         syscall
```

1.10. Zusammenfassung

Was haben wir gelernt?

- Kurzes Assembler-Kapitel, jedoch **viel Information** zu Assembler bereits aus der Vorlesung „Systemnahe Informatik“ bekannt
- **Assembler-Sprache** als „Hoch“-Sprache **zur Maschinenprogrammierung**
- Maschinenprogrammierung in Assembler ist **spezifisch für bestimmten Prozessor**
- Der Prozessor besitzt **diverse Register**, auf die mit Assembler-Befehlen zugegriffen wird
- Die Ausführung von Assembler-Befehlen **berücksichtigt und beeinflusst Status-Flags**
- (Leider) gibt es für 80x86 die Intel- und die **AT&T-Syntax**
- Es gibt **diverse Adressierungsarten**, die ***NICHT*** beliebig miteinander kombinierbar sind (z.B. nicht 2 Operanden mit Register- bzw. indirekter Adressierung)
- Wir legen den Schwerpunkt nicht auf eigene Assembler-Programme, sondern auf den **Aufruf von Assembler-Programmen aus Hochsprachen**, z.B. C unter Linux
- Hierfür sind (strenge) **Calling Conventions** einzuhalten (z.B. **cdecl Convention**, **AMD64 ABI**)
 - Stack Frame, Parameterübergabe, lokale Variablen, Registerrettung, Rückgabewert