Technische Informatik I
x86 Maschinensprache + Assembler

Thorsten Thormählen

14. Januar 2025

Teil 10, Kapitel 1

Dies ist die Druck-Ansicht.

Aktiviere Präsentationsansicht

Steuerungstasten

→ nächste Folie (auch Enter oder Spacebar).
← vorherige Folie
d schaltet das Zeichnen auf Folien ein/aus
p wechselt zwischen Druck- und Präsentationsansicht
CTRL + vergrößert die Folien
CTRL - verkleinert die Folien
CTRL 0 setzt die Größenänderung zurück

Notation

Typ	Schriftart	Beispiele
Variablen (Skalare)	kursiv	a, b, x, y
Funktionen	aufrecht	$\mathrm{f}, \mathrm{g}(x), \max(x)$
Vektoren	fett, Elemente zeilenweise	$\mathbf{a}, \mathbf{b} = \begin{pmatrix} x \\ y \end{pmatrix} = (x, y)^\top$, $\mathbf{B} = (x, y, z)^\top$
Matrizen	Schreibmaschine	$\mathtt{A}, \mathtt{B} = \begin{bmatrix} a & b \\ c & d \end{bmatrix}$
Mengen	kalligrafisch	$\mathcal{A}, B = \{a, b\}, b \in \mathcal{B}$
Zahlenbereiche, Koordinatenräume	doppelt gestrichen	$\mathbb{N}, \mathbb{Z}, \mathbb{R}^2, \mathbb{R}^3$

Inhalt

Einführung in die x86 Maschinen- und Assemblersprache

Register der x86 Familie

Assemblerbefehle der x86 Familie

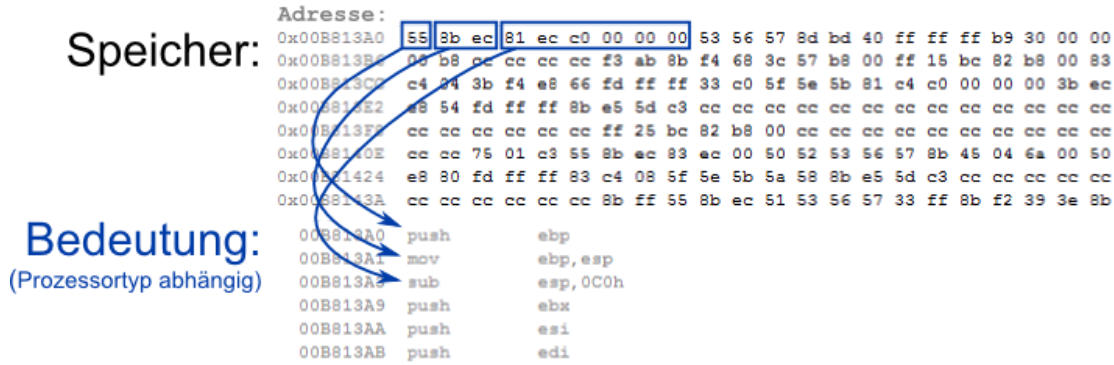
Maschinensprache

Computer werden durch Programme gesteuert, die aus Befehlen in Maschinensprache bestehen (wie aus dem vorangegangenen Kapitel bekannt)

Diese Befehle liegen im Hauptspeicher und werden nacheinander abgearbeitet

Die Maschinensprache wird vom Hersteller einer CPU definiert und kann vom Programmierer nicht verändert werden

Der Hersteller legt fest, welcher Zahlencode im Speicher welchem Maschinensprachebefehl entspricht



Damit ist ein Programm in Maschinensprache immer nur auf einer bestimmten CPU bzw. auf den untereinander kompatiblen CPUs einer Prozessorfamilie lauffähig

Maschinensprache / Assembler

Eine Maschinensprache ist die Menge von Befehlen, die einem Programmierer für eine konkrete CPU zur Verfügung steht

Diese Befehle werden normalerweise in der Praxis nicht direkt durch ihren Zahlencode eingegeben, sondern in einer Assemblersprache (auch "Assembler" genannt)

Eine Assemblersprache ist einer lesbare Art, Programme in Maschinensprache zu formulieren

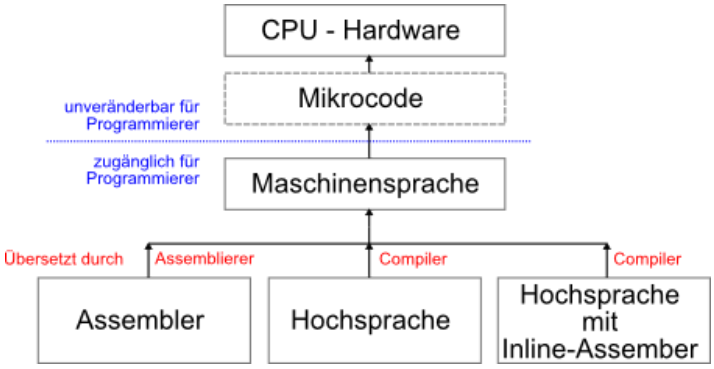
Die lesbare Variante eines Maschinenbefehls in Assemblersprache wird Mnemonik genannt

So wird z.B. für den Sprungbefehl die Mnemonik `jmp` verwendet (für engl. "jump");

Es gibt somit eine eineindeutige Abbildung zwischen Maschinensprache und Assemblersprache

- Ein Assemblierer ist ein Übersetzer von Assemblersprache in Maschinensprache
- Ein Disassemblierer ist ein Übersetzer von Maschinensprache in Assemblersprache

Erzeugung von Maschinensprache mittels Compiler



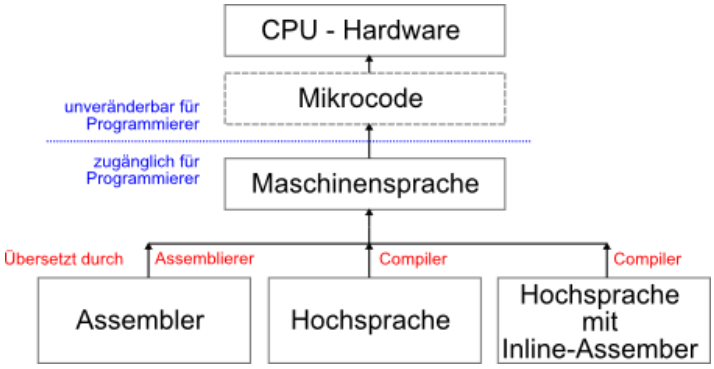
Softwareentwicklung findet heutzutage typischerweise in einer höheren Programmiersprache statt (z.B. C, C++, Fortran, Pascal, ...)

Um ein in einer höheren Programmiersprache geschriebenes Programm ausführen zu können, muss dies jedoch letztendlich durch einen Compiler in eine äquivalente Folge von Maschinenbefehlen übersetzt werden

Beim Kompilieren kann angegeben werden für welchen CPU Typ der Maschinencode erzeugt werden soll

Daher kann ein Programm in Hochsprache im Gegensatz zu einem Assemblerprogramm leicht auf andere Hardware angepasst werden

Inline-Assembler



Früher wurden viele Programme in Assembler erstellt

Heute ist mit der schnelleren Hardware die Bedeutung von Maschinensprache zurückgedrängt worden

Softwareentwicklung findet typischerweise in einer höheren Programmiersprache statt

Bei sehr zeitkritischen Funktionen innerhalb eines Programms wird jedoch weiterhin Maschinensprache in Form von Inline-Assembler verwendet

Der Compiler optimiert nur den Teil, der in der Hochsprache geschrieben ist, der Assemblerteil wird direkt in die Maschinensprache übertragen

x86 Maschinensprache

In dieser Vorlesung wird aufgrund der starken Verbreitung und Praxisrelevanz, im Folgenden die Maschinensprache der x86 Prozessorfamilie vorgestellt

Diese Maschinensprache ist auf allen 80x86 und x86-64 Prozessoren der Firmen Intel und AMD ausführbar, d.h. auf fast jedem Desktop-PC oder Laptop (Ausnahmen sind z.B. PowerPC-basierte Apple Computer)

Hörer der Vorlesung können zur Maschinensprache-Programmierung einen Rechner in den PC-Pools des Fachbereichs oder ihren privaten Rechner verwenden

Werkzeuge zur Erstellung von x86 Maschinensprache

Zur Erstellung vom x86 Assembler stehen verschiedene Werkzeuge zur Auswahl:

Microsoft Macro Assembler (MASM) ist ein verbreiteter Assembler für MS-DOS und Windows

MASM32 ist eine freie Variante von MASM

Turbo Assembler (TASM) ist ein Assembler von Borland für das Betriebssystem MS-DOS

NASM ("Netwide Assembler") ist ein beliebter Assembler, der für verschiedene Betriebssysteme (DOS, Windows, Unix) verwendet werden kann. Es gibt sogar Online-Dienste bei denen ohne Installation auf dem eigenen Rechner mit NASM experimentiert werden kann:
https://www.jdoodle.com , https://onecompiler.com ,

Diese Werkzeuge unterscheiden sich natürlich nicht in den Maschinenbefehlen (diese sind durch die x86 Prozessorfamilie vorgegeben) sondern lediglich in der Art der Darstellung und durch unterschiedliche Makros, die die Arbeit erleichtern sollen

Inline-Assembler in Microsoft Visual C/C++

In dieser Vorlesung werden die gezeigten Maschinensprache-Beispiele in Form von Inline-Assembler innerhalb eines C/C++-Programms vorgestellt

Als Entwicklungsumgebung wird Microsoft Visual Studio verwendet

Vorteile:

- Inline-Assembler wird in der Praxis gerne verwendet, da niemand größere Projekte komplett in Assembler entwickeln möchte
- Auf Symbole und Variablen des C/C++-Programms kann direkt innerhalb des Assembler-Codes zugegriffen werden
- Die Syntax ist verhältnismäßig einfach
- Microsoft Visual Studio hat einen integrierten Debugger
- Auch Registerinhalte, Speicher etc. lassen sich komfortabel anzeigen
- Visual Studio ist auf jedem Windows-Rechner im Fachbereich installiert
- Microsoft Visual Studio (Community Version) steht kostenfrei zur Verfügung: Download

Nachteil:

- Plattformabhängigkeit: Für die Verwendung mit anderen Assemblern muss die Syntax entsprechend angepasst werden

Inline-Assembler in Microsoft Visual C/C++

Beispiel:

```
#include <stdio.h> // includes "printf" command ;

int add(int varA, int varB ) {
   // inline assembler starts here
   __asm {
      mov eax, varA   ; // move first argument to EAX register
      mov ecx, varB   ; // move second argument to ECX register
      add eax, ecx    ; // EAX = EAX + ECX
   }
}

int main() {
  int result = add(1, 2);    // call C-function "add"
  printf( "1 + 2 = %d\n", result); // output result to console
  return 0;
}
```

Quelldatei: main.cpp

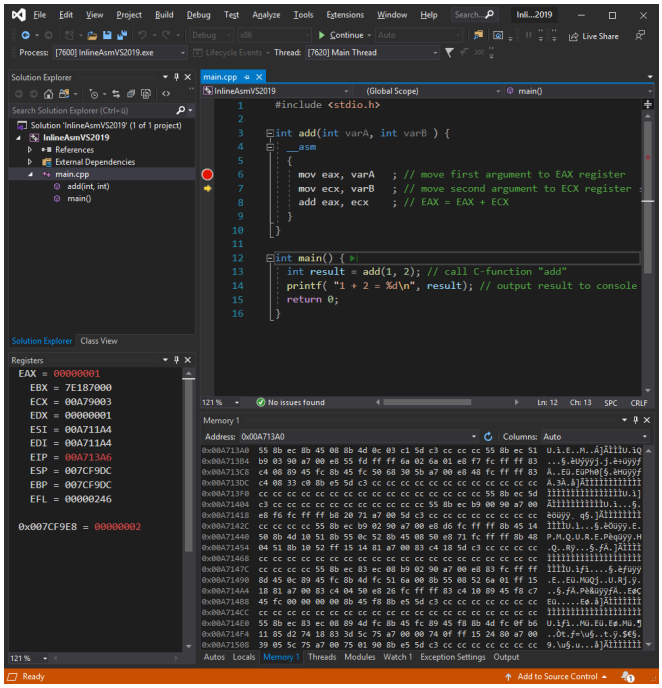
Projekt für MS Visual Studio: VS2012 , VS2013 VS2015 VS2017 VS2019 VS2022

Diese Projektdateien können für alle folgenden Beispiele wiederverwendet werden, indem jeweils die Quelldatei "main.cpp" ersetzt wird

Inline-Assembler in Microsoft Visual C/C++

Verwendung vom Microsoft Visual Studio:

- Erstellen des Projekts: F7
- Ausführen: CTRL+F5
- Debuggen starten: F5
- Debuggen Einzelschritt: F11
- Debuggen Prozedurschritt: F10
- Breakpoint setzen/entfernen: in die Leiste neben dem Quellcode klicken
- Anzeigen von Registern: Menü → Debuggen→ Fenster→ Register
- Anzeigen von Arbeitsspeicher: Menü→ Debuggen→ Fenster→ Arbeitsspeicher



Alternative für Linux-Nutzer

Unter Linux ist die Verwendung von Microsoft Visual Studio nicht möglich. Als Alternative zum Kompilieren der bereitgestellen Beispiele kann der Intel C and C++ Compiler mit der Option `-use-msasm` verwendet werden

In einer Shell:

```
> icc -use-msasm -o my_program main.cpp
```

Der Intel Compiler steht Studenten kostenfrei zur Verfügung: Download

Eine weitere Möglichkeit sind Online-Dienste, bei denen ohne Installation auf dem eigenen Rechner mit dem Microsoft Compiler experimentiert werden kann:

https://gcc.godbolt.org

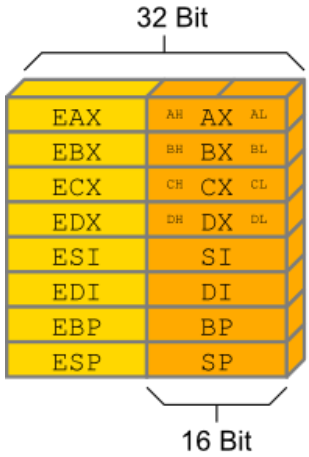
Als Compiler "x86 msvc (WINE)" auswählen

Diese Alternativen haben jedoch keinen integrierten Debugger und werden daher nur bedingt empfohlen

Microsoft Visual Studio ist auf jedem Windows-Rechner im Fachbereich installiert

Register der x86-Familie

Allzweckregister



Sonderfunktion:
Akkumulator
Base
Counter
Data
Source Index
Destination Index
Base Pointer
Stack Pointer

Segmentregister

Funktion:
Code Segment
Data Segment
Stack Segment
Extra Segment

Befehlszähler

Flagregister

Register der x86-Familie

Die ersten x86-Prozessoren (8088, 8086 und 80286) waren 16-Bit Prozessoren

Sie hatten 14 Register (orange gekennzeichnet in Abb.)

- 8 Allzweckregister, die jedoch jeweils auch noch eine Sonderfunktion hatten
- 4 Segment Register
- 1 Befehlszähler
- 1 Flagregister

Das niederwertige Byte (Low Byte) bzw. höherwertige Byte (High Byte) der Register AX, BX, CX und DX sind als 8-Bit Register gesondert ansprechbar (z.B. AX besteht aus AL und AH)

Seit dem 80386 wird mit 32-Bit breiten Registern gearbeitet

Die bestehenden Register wurden erweitert (gelb gekennzeichnet) und können als EAX, EBX, usw. angesprochen werden

Die 16-Bit Teile der Register sind weiterhin ansprechbar. Neuerungen in der x86-Architektur waren bisher immer abwärts-kompatibel, d.h. neuere Prozessoren können ältere Programme immer noch ausführen

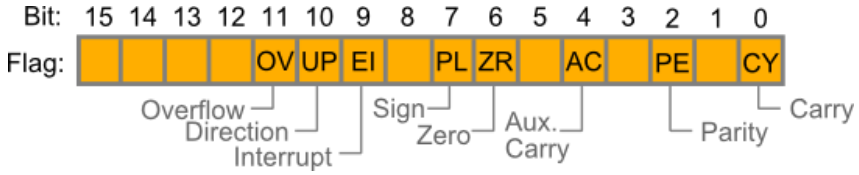
Spätere x86-Prozessoren haben weitere Register eingeführt (MMX-Register, SSD-Register, usw.) diese werden zunächst hier nicht weiter betrachtet

Allzweckregister

Die Allzweckregister haben jeweils noch eine Sonderfunktion:

Register	Sonderfunktion	Erklärung
AX	Akkumulator	Ziel für Rechenoperationen
BX	Base	Zeiger für Zugriffe auf Speicher
CX	Counter	Zähler in Schleifen
DX	Data	Datenregister
SI	Source Index	Quellindex für Verarbeitung von Zeichenketten
DI	Destination Index	Zielindex für Verarbeitung von Zeichenketten
BP	Base Pointer	Anfangsadresse des Stapelsegments
SP	Stack Pointer	Stapelzeiger

Flagregister



Das Flagregister ändert sich nach arithmetischen Operationen

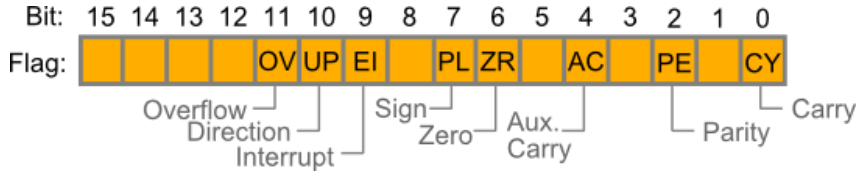
Es dient dazu, die Situationen nach der Durchführung einer ALU-Operation anzuzeigen

Es ist eigentlich ein Ausgaberegister, dennoch kann es auf dem Umweg über den später zu besprechenden Stack gezielt verändert werden

Von den 16 bzw. 32 Bits des Flag-Registers sind nur 8 für uns interessant:

- Die Flags CY, AC, OV, PL, ZR, PE beziehen sich immer auf das Ergebnis einer gerade durchgeführten Operation
- Die Flags UP, EI dienen als Schalter. Sie bleiben unverändert, bis man sie durch Spezialbefehle verändert

Flagregister



Flag	Bedeutung	Erklärung
CY	Carry	Bereichsüberschreitung für vorzeichenlose Zahlen
AC	Aux. Carry	Bereichsüberschreitung für vorzeichenlose 4-Bit Zahlen
OV	Overflow	Bereichsüberschreitung bei arithmetischer Operation auf Zahlen mit Vorzeichen
PL	Sign	Ergebnis war negativ
ZR	Zero	Ergebnis war null
PE	Parity	Ergebnis hat eine gerade Anzahl von Einsen (im niederwertigsten Byte)
UP	Direction	Legt die Richtung von String-Befehlen fest
EI	Interrupt	Bestimmt, ob Interrupts zugelassen werden

Segmentregister

Die Segmentregister werden später bei der Adressierung des Speichers eine Rolle spielen:

Register	Funktion	Erklärung
CS	Code Segment	Anfangsadresse des Code-Segments
DS	Data Segment	Anfangsadresse des Daten-Segments
SS	Stack Segment	Anfangsadresse des Stack-Segments
ES	Extra Segment	keine besondere Funktion
FS		keine besondere Funktion
GS		keine besondere Funktion

Format von Assemblerbefehlen

In Assembler wird je ein Befehl pro Zeile geschrieben

Dabei hat ein Assemblerbefehl die Form:

```
[Label] [Operation] [Operanden] [;Kommentar]
```

Die eckigen Klammern sollen andeuten, dass diese Komponente optional ist, d.h. es ist auch eine leere Zeile möglich

Zwischen Groß- und Kleinschreibung wird nicht unterschieden (außer wenn im Inline-Assembler auf C/C++ Variablen zugegriffen wird)

Leerzeichen zwischen den Komponenten werden ignoriert

Bei einem Semikolon beginnt ein Kommentar, der sich bis zum Zeilenende erstreckt

Alternativ kann ein Kommentar in MS Visual Studio auch mit den in C/C++ üblichen doppelten Schrägstrichen "//" gekennzeichnet werden

Format von Assemblerbefehlen

Viele Assemblerbefehle haben die Form:

```
op ziel, quelle
```

- Dies bedeutet, dass die Operanden Ziel und Quelle mit der Operation `op` verknüpft werden
- Das Ergebnis wird in Ziel gespeichert
- In Java bzw. C/C++ entspräche dies demnach: `Ziel = Ziel op Quelle;`
- Ziel kann ein Register oder eine Speicheradresse sein
- Quelle kann ein Register, eine Speicheradresse oder ein konstanter Zahlenwert sein
- Die verknüpften Operanden müssen in ihrer Bitbreite zueinander passen

Beispiele:

```
mov eax, ecx    ; eax = ecx
sub ax, cx      ; ax = ax - cx
add cx, ax      ; cx = cx + ax
```

Der Assemblerbefehl MOV

Einer der meist verwendeten Befehle ist der Move-Befehl:

```
mov ziel, quelle
```

Dabei werden die Daten aus der Quelle zum Ziel kopiert

Ziel kann ein Register oder eine Speicheradresse sein

Quelle kann ein Register, eine Speicheradresse oder eine Kontante sein

Einschränkungen:

- Es können nicht beide Operanden Speicheradressen sein
- Es können nicht beide Operanden Segmentregister sein
- Konstanten können nicht direkt in Segmentregister geschrieben werden

Der Assemblerbefehl MOV

Beispiele für den Move-Befehl:

```
int main() {
  int varA = 6;
  int varB = 7;
  __asm {
    mov eax, 10h        ; // move the hex value 10 to EAX
    mov eax, 10         ; // move the decimal value 10 to EAX
    mov eax, 010        ; // move the octal value 10 to EAX
    mov ecx, eax        ; // move EAX register to ECX register
    //mov  fs, ds       ; // does not work
    //mov  ax,  ds      ; // this works instead: (but segment registers
    //mov  fs,  ax      ; // should not be modified)
    mov eax, 11111111h  ; // EAX = 11111111h;
    mov ax, 3333h       ; // EAX = 11113333h;
    //mov  varA, varB   ; // does not work
    mov  eax, varB      ; // this works instead:
    mov  varA, eax
  }
  return 0;
}
```

Quelldatei: main.cpp

Der Assemblerbefehl ADD

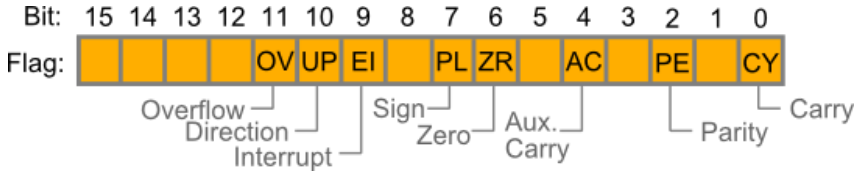
Der Befehl `add` führt eine Addition aus:

```
add ziel, quelle
```

Beispiele für den Add-Befehl (Quelldatei: main.cpp):

```
#include <stdio.h>
int main() {
  int varA = 6, varB = 7;
  __asm {
    //add varA, varB      ; // does not work
    mov eax, varB         ; // this works instead:
    add varA, eax         ; //
  }
  printf("varA = %d\n", varA);
  __asm {
    mov eax, 21
    mov edx, 24
    add eax, edx
    mov varA, eax
  }
  printf("varA = %d\n", varA);
  return 0;
}
```

Arithmetische Operationen und das Flagregister



Arithmetische Operationen, wie z.B. der Befehl `add`, verändern das Flagregister

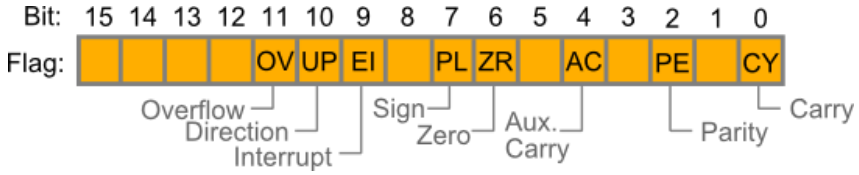
Der Prozessor weiß nicht, ob der Inhalte eines Registers als vorzeichenlose Zahl (unsigned) oder als vorzeichenbehaftete ganze Zahl in Zweierkomplement-Darstellung interpretiert werden muss

Die Operation wird einfach ausgeführt und die Flags gesetzt. Die Interpretation der Flags ist Sache des Programmierers

Nicht alle Befehle beeinflussen alle Flags, so dass auch später noch durch eine frühere Operation erzeugte Flags ablesbar sind

Der Befehl `add` beeinflusst: `OV`,`PL`,`ZR`,`AC`, `PE` und `CY`

Arithmetische Operationen und das Flagregister



Beispiele (Quelldatei: main.cpp):

```
int main() {
  __asm
  {
    mov eax, 00000000h
    mov ecx, 00000000h
    add eax, ecx    ; // EAX=0;        Flags: OV=0 PL=0 ZR=1 AC=0 PE=1 CY=0
    mov eax, 00000000h
    mov ecx, 00000001h
    add eax, ecx    ; // EAX=1;        Flags: OV=0 PL=0 ZR=0 AC=0 PE=0 CY=0
    mov eax, 00000001h
    mov ecx, 0000000Fh
    add eax, ecx    ; // EAX=10;       Flags: OV=0 PL=0 ZR=0 AC=1 PE=0 CY=0
    mov eax, 00000000h
    mov ecx, 7FFFFFFFh
    add eax, ecx    ; // EAX=7FFFFFFF; Flags: OV=0 PL=0 ZR=0 AC=0 PE=1 CY=0
    mov eax, 00000001h
```

Quiz

Frage: Was ist das Ergebnis folgender Assemblerbefehle?

```
mov eax, 00000002h
mov ecx, 7FFFFFFEh
add eax, ecx
```

Antwort 1: **EAX=FFFFFFFF; Flags: OV=0 PL=1 ZR=0 AC=1 PE=0 CY=0**

Antwort 2: **EAX=80000000; Flags: OV=1 PL=1 ZR=0 AC=1 PE=1 CY=0**

Antwort 3: **EAX=80000000; Flags: OV=0 PL=0 ZR=1 AC=1 PE=0 CY=1**

Am Online-Quiz teilnehmen durch Besuch der Webseite:
www.onlineclicker.org

Arithmetische Operationen und das Flagregister

```
    mov al, -3   ; // AL=FDh;
    mov cl, -1   ; // CL=FFh;
    add al, cl   ; // AL=FCh; Flags: OV=0 PL=1 ZR=0 AC=1 PE=1 CY=1
    mov al, 253  ; // AL=FDh;
    mov cl, 255  ; // CL=FFh;
    add al, cl   ; // AL=FCh; Flags: OV=0 PL=1 ZR=0 AC=1 PE=1 CY=1
```

In diesem Beispiel (Quelldatei: main.cpp) werden zweimal exakt die gleichen Operationen ausgeführt

Der Assembler übersetzt -3 und 253 in das gleiche Bitmuster: 11111101 = FDh

Der Assembler übersetzt -1 und 255 in das gleiche Bitmuster: 11111111 = FFh

Nach der Addition ist in beiden Fällen das Ergebnis FCh=252 und Overflow=0 Carry=1

Interpretation als Addition von vorzeichenlosen Zahlen:

- Das gesetzte Carry zeigt, dass das Ergebnis 252 falsch ist: 253+255=508

Interpretation als Addition von vorzeichenbehafteten Zahlen im Zweierkomplement:

- Das nicht gesetzte Overflow zeigt, dass das Ergebnis FCh=-4 richtig ist: -3 + -1 =-4

Erkenntnis: Für die richtige Interpretation sind die Flags wichtig

Der Assemblerbefehl ADC

Um die Addition 253+255=508 doch richtig auszuführen, gibt es zwei Möglichkeiten

Ein größeres Register verwenden

```
mov eax, 253
mov ecx, 255
add eax, ecx    ; // EAX=000001FC; Flags: OV=0 PL=0 ZR=0 AC=1 PE=1 CY=0
```

Den Befehl `adc` verwenden, der genau wie der `add` Befehl arbeitet, jedoch zusätzlich das Carry-Bit hinzuaddiert

```
mov al, 253     ; // AL=FDh;
mov cl, 255     ; // CL=FFh;
mov ah, 0       ; // AH=0h;
add al, cl      ; // AL=FCh; AH=0h; Flags: OV=0 PL=1 ZR=0 AC=1 PE=1 CY=1
adc ah, 0       ; // AL=FCh; AH=1h; Flags: OV=0 PL=0 ZR=0 AC=0 PE=0 CY=0
// AX=01FCh
```

Der Assemblerbefehl SUB

Der Befehl `sub` führt eine Subtraktion aus:

```
sub ziel, quelle
```

Der Befehl beeinflusst die gleichen Register, wie der Add-Befehl: `OV`,`PL`,`ZR`,`AC`, `PE` und `CY`

Bei x86 wird das Carry `CY` gesetzt, wenn $(\texttt{ziel} < \texttt{quelle})$ (bei Interpretation als vorzeichenlose Zahl). Das Carry erfüllt dann die Funktion eines Borrow.

Bei vorzeichenbehafteten Zahlen ist wieder nur das Overflow-Flag `OV` relevant

Eine Subtraktion kann hardwaretechnische durch eine Addtionen von `quelle` und dem Zweierkomplement von `ziel` implementiert werden, also $a - b = a + (-b)$ (siehe Kapitel Arithmetik Schaltungen)

- Dies funktioniert sofort perfekt bei Interpretation der Operanden als vorzeichenbehaftete Zahl
- Bei der Interpretation als vorzeichenlose Zahl wird allerdings das Carry von der gezeigten Harwareschaltung gesetzt, wenn $(\texttt{ziel} \geq \texttt{quelle})$. Für den x86-`sub` Befehl, müsst das Carry der Harwareschaltung demnach invertiert werden

Der Assemblerbefehl SUB

Beispiel für den sub-Befehl (Quelldatei: main.cpp):

```
int main() {
  __asm {
    mov al, 2       ; // AL=02h;
    mov bl, -1      ; // BL=FFh;
    sub al, bl      ; // AL=03h; Flags: OV=0 PL=0 ZR=0 AC=1 PE=1 CY=1
    mov al, 2       ; // AL=02h;
    sub al, 255     ; // AL=03h; Flags: OV=0 PL=0 ZR=0 AC=1 PE=1 CY=1
  }
  return 0;
}
```

Kontrollrechnung mit dem Zweierkomplement (Carry CY wird invertiert):

$$\begin{array}{rr} 00000010 & (2) \\ -11111111 & (-1) \text{ bzw. } (255) \\ \downarrow & \\ 00000010 & (2) \\ +00000001 & \text{Zweierkomplement} \\ \hline 00000011 & (3) \end{array}$$

Der Assemblerbefehl SBB

Der Befehl `sbb` steht für engl. "subtract with borrow", d.h. es wird eine Subtraktion ausgeführt und zusätzlich das Carry subtrahiert

Beispiel für den sbb-Befehl (Quelldatei: main.cpp):

```
int main() {
  __asm {
    //
    mov ax, 800     ; // AX=320h; AL=20h
    mov cl, 34      ; // CL=22h;
    sub al, cl      ; // AL=FEh=-2; Flags: OV=0 PL=1 ZR=0 AC=1 PE=0 CY=1
    sbb ah, 0       ; // AH=02h;    Flags: OV=0 PL=0 ZR=0 AC=0 PE=0 CY=0
    // AX=2FE=766
  }
  return 0;
}
```

Größenvergleich mittels SUB Befehl

Der Befehl `sub` kann verwendet werden, um zwei Operanden zu vergleichen

```
sub operandA, operandB
```

Ist es so einfach?

- Zero-Flag=0, Sign-Flag=1 bedeutet operandA < operandB
- Zero-Flag=1, Sign-Flag=0 bedeutet operandA = operandB
- Zero-Flag=0, Sign-Flag=0 bedeutet operandA > operandB

Nein, es kommt darauf an, ob es sich um eine vorzeichenlose oder vorzeichenbehaftete Zahl handelt. Beispiel:

- Ist der Hex-Wert 2h kleiner oder großer als FFh?
- Wenn vorzeichenlos: 2h=2, FFh=255 → Ergebnis: kleiner
- Wenn vorzeichenbehaftet: 2h=2, FFh=-1 → Ergebnis: größer

Daher werden zwei verschiedene Größenvergleiche eingeführt:

- Wenn vorzeichenlos: "above" und "below"
- Wenn vorzeichenbehaftet: "greater" und "less"

Größenvergleich mittels SUB Befehl

```
sub operandA, operandB
```

Vorzeichenlose Zahlen: "above" und "below"

Aussage	Bedingung
operandA below operandB	CY=1
operandA equal operandB	ZR=1
operandA above operandB	CY=0

Vorzeichenbehaftete Zahlen: "less" und "greater"

Aussage	Bedingung
operandA less operandB	PL ≠ OV
operandA equal operandB	ZR=1
operandA greater operandB	PL = OV

Größenvergleich mittels SUB Befehl

```
sub operandA, operandB
```

Vorzeichenlose Zahlen: "above" und "below"

Aussage	Bedingung
operandA below operandB	CY=1
operandA equal operandB	ZR=1
operandA above operandB	CY=0

Erklärung:

- Beim `sub` Befehl wird das Carry als Borrow verwendet
- Ein Borrow wird nur benötigt, wenn `operandA` kleiner als `operandB` ist

Größenvergleich mittels SUB Befehl

```
sub operandA, operandB
```

Vorzeichenbehaftete Zahlen: "less" und "greater"

Aussage	Bedingung
operandA less operandB	PL ≠ OV
operandA equal operandB	ZR=1
operandA greater operandB	PL = OV

Zur Erklärung betrachten wir den Fall, dass `operandA` größer als `operandB` ist:

PL = OV = 0, kein Overflow ist aufgetreten (d.h. keine unerwarteter Vorzeichenwechsel, Ergebnis stimmt) und Ergebnis ist positiv

PL = OV = 1, Overflow ist aufgetreten (d.h. unerwarteter Vorzeichenwechsel, Ergebnis falsch) und das Ergebnis ist negativ ("negativer Overflow").
Der negative Overflow bedeutet, dass das Ergebnis eigentlich positiv sein sollte und daher `operandA` größer als `operandB` ist.

Obwohl das Berechnungsergebnis falsch ist (angezeigt durch den Overflow), kann trotzdem ein Größenvergleich durchgeführt werden

Größenvergleich mittels SUB Befehl

Beispiel für den Größenvergleich von vorzeichenbehafteten Zahlen:

```
void main() {
  __asm {
    mov eax, -1
    mov ecx, -2
    sub eax, ecx          ; // OV = 0, PL = 0, ZR = 0 -> eax > ecx
    mov eax, -2
    mov ecx, -1
    sub eax, ecx          ; // OV = 0, PL = 1, ZR = 0 -> eax < ecx
    mov eax, 7FFFFFFFh    ; // largest positive signed integer
    mov ecx, -1
    sub eax, ecx          ; // OV = 1, PL = 1, ZR = 0 -> eax > ecx
    mov eax, 80000000h    ; // largest negative signed integer
    mov ecx, 1
    sub eax, ecx          ; // OV = 1, PL = 0, ZR = 0 -> eax < ecx
    mov eax, 1
    mov ecx, 1
    sub eax, ecx          ; // OV = 0, PL = 0, ZR = 1 -> eax == ecx
  }
}
```

Quelldatei: main.cpp

Der Assemblerbefehl CMP

Da für einen Vergleich nur die Flags nach der Subtraktion eine Rolle spielen, nicht aber das Ergebnis, gibt es folgenden Maschinenbefehl:

```
cmp operandA, operandB
```

Dieser Befehl setzt genau die Flags, die ein analoger Sub-Befehl setzen würde

Der wesentliche Unterschied ist, dass der operandA unverändert bleibt

Die Auswertung der gesetzten Flags erfolgt meist durch einen direkt nachfolgenden bedingten Sprung-Befehl.

Sprungbefehle und Sprungmarken

Assemblerbefehle werden in der Reihenfolge ausgeführt, in der sie im Text erscheinen, es sei denn, es handelt sich um einen Sprungbefehl

Ein solcher bewirkt die Fortsetzung des Programms an einer beliebigen anderen Stelle

Das Argument eines Sprungbefehls ist das Ziel des Sprungs, das über eine Adresse oder eine Sprungmarke (engl. "label") angegeben wird

Die Sprungmarke wird vom Assembler in eine Adresse umgesetzt

Eine Sprungmarke, die als Sprungziel dienen soll, muss mit einem Doppelpunkt abgeschlossen werden

Als Sprungmarke dürfen natürlich keine Wörter verwendet werden, die bereits in der Assemblersprache anderweitig genutzt werden

Die Sprungbefehle realisieren einen Sprung, in dem der Befehlszähler `EIP` auf die Zieladresse gesetzt wird

Der Assemblerbefehl JMP

Für den unbedingten Sprung gibt es den JMP-Befehl (engl. "jump")

```
jmp sprungziel
```

Sprungziel kann eine Sprungmarke oder eine Adresse sein

Beispiel:

```
int main() {
  __asm {
            mov eax, 3
            jmp Important
            mov eax, 33
  Important: mov ecx, 6
            add eax, ecx  ; // eax = ?
  }
  return 0;
}
```

Quelldatei: main.cpp

Assemblerbefehle für bedingte Sprünge

Die bedingten Sprünge werten die Flags aus, die durch vorangegangene Befehle gesetzt wurden

Wenn die entsprechende Bedingung für den Sprung erfüllt ist, wird der Sprung ausgeführt, ansonsten der nächste Befehl

Die Bedingung ist in die Mnemonik kodiert. Es gibt mehrere Mnemoniks mit der gleichen Funktion (nicht alle sind hier aufgeführt)

Befehle für bedingte Sprünge, die das Zero-Flag auswerten sind:

Befehl	Bedeutung	Bedingung für Flags
JZ	Spring, wenn null ("jump if zero")	ZR=1
JE	Spring, wenn gleich ("jump if equal")	ZR=1
JNZ	Spring, wenn nicht null ("jump if not zero")	ZR=0
JNE	Spring, wenn nicht gleich ("jump if not equal")	ZR=0

Assemblerbefehle für bedingte Sprünge

Bedingte Sprünge nach dem Vergleich von vorzeichenlosen Zahlen:

Befehl	Bedeutung	Bedingung
JB	Spring, wenn niedriger ("jump if below")	CY=1
JNAE	Spring, wenn nicht höher oder gleich ("jump if not above or equal")	CY=1
JAE	Spring, wenn höher oder gleich ("jump if above or equal")	CY=0
JNB	Spring, wenn nicht niedriger ("jump if not below")	CY=0
JBE	Spring, wenn niedr. gleich ("jump if below equal")	CY=1 oder ZR=1
JNA	Spring, wenn nicht höher ("jump if not above")	CY=1 oder ZR=1
JA	Spring, wenn höher ("jump if above")	CY=0 und ZR=0
JNBE	Spring, wenn nicht niedriger oder gleich ("jump if not below or equal")	CY=0 und ZR=0

Assemblerbefehle für bedingte Sprünge

Bedingte Sprünge nach dem Vergleich von vorzeichenbehafteten Zahlen:

Befehl	Bedeutung	Bedingung für Flags
JG	Spring, wenn größer ("jump if greater")	ZR=0 und PL = OV
JNLE	Spring, wenn nicht kleiner oder gleich ("jump if not less or equal")	ZR=0 und PL = OV
JLE	Spring, wenn kleiner oder gleich ("jump if less or equal")	ZR=1 oder PL ≠ OV
JNG	Spring, wenn nicht größer ("jump if not greater")	ZR=1 oder PL ≠ OV
JL	Spring, wenn kleiner ("jump if less")	PL ≠ OV
JNGE	Spring, wenn nicht größer oder gleich ("jump if not greater or equal")	PL ≠ OV
JGE	Spring, wenn größer oder gleich ("jump if greater or equal")	PL = OV
JNL	Spring, wenn nicht kleiner ("jump if not less")	PL = OV

Assemblerbefehle für bedingte Sprünge

Bedingte Sprünge basierend auf bestimmten Flags:

Befehl	Bedeutung	Bedingung
JC	Spring, wenn Carry gesetzt ("jump if carry")	CY=1
JNC	Spring, wenn Carry nicht gesetzt ("jump if not carry")	CY=0
JO	Spring, wenn Overflow gesetzt ("jump if overflow")	OV=1
JNO	Spring, wenn Overflow nicht gesetzt ("jump if not overflow")	OV=0
JS	Spring, wenn Sign gesetzt ("jump if sign")	PL=1
JNS	Spring, wenn Sign nicht gesetzt ("jump if not sign")	PL=0

Assemblerbefehle für bedingte Sprünge

Bedingte Sprünge basierend auf dem (E)CX-Register:

Befehl	Bedeutung	Bedingung
JCXZ	Spring, wenn CX-Register null ist ("jump if register CX is zero")	CX=0
JECXZ	Spring, wenn ECX-Register null ist ("jump if register ECX is zero")	ECX=0
LOOP	Dekrementiere (E)CX; springe, wenn (E)CX nicht null ("decrement (E)CX; jump if (E)CX not zero")	(E)CX≠0

Assemblerbefehle für bedingte Sprünge

Beispiel: JNA ("jump if not above")

```
int main() {
  __asm {
                    mov eax, 2
                    mov ecx, 3
                    cmp eax, ecx
                    jna ThisIsMyLabel
                    mov eax, 33
    ThisIsMyLabel:  mov ecx, 6
                    add eax, ecx; // eax = ?
  }
  return 0;
}
```

Quelldatei: main.cpp

Assemblerbefehle für bedingte Sprünge

Beispiel: Summe 1+2+3+ .... + n :

```
#include <stdio.h>
unsigned int smallGauss(unsigned int varA) { // version 1
  __asm
  {
                mov eax, 0
                mov ecx, varA
                cmp ecx, 0
    ExecuteLoop: jz EndLoop
                add eax, ecx
                sub ecx, 1
                jmp ExecuteLoop;
    EndLoop:
  }
}

int main() {
  unsigned int n = 5;
  unsigned int result = smallGauss(n); // call C-function "smallGauss"
  printf( "smallGauss(%d) = %d\n", n, result); // output result to console
  return 0;
}
```

Quelldatei: main.cpp

Die Assemblerbefehle INC und DEC

Die arithmetischen Operationen `inc` und `dec` dienen zum Inkrementieren bzw. Dekrementieren eines Speicher- oder Registerinhaltes um 1

Aufgrund ihrer Geschwindigkeit und Lesbarkeit sind sie einer Addition von 1 mit dem Add-Befehl bzw. Subtraktion von 1 mit dem Sub-Befehl vorzuziehen

Wird der Code aus der vorherigen Implementierung der Funktion `smallGauss` angepasst, ergibt sich:

```
unsigned int smallGauss(unsigned int varA) { // version 2
  __asm
  {
                mov eax, 0
                mov ecx, varA
                cmp ecx, 0
    ExecuteLoop: jz EndLoop
                add eax, ecx
                dec ecx
                jmp ExecuteLoop;
    EndLoop:
  }
}
```

Quelldatei: main.cpp

Die Assemblerbefehle INC und DEC

Die Befehle `inc` und `dec` beeinflussen die Flags: `OV`,`PL`,`ZR`,`AC` und `PE`

Interessant ist, dass das Carry-Flag nicht gesetzt wird

Dies ist nicht nötig, da das Zero-Flag und das Carry-Flag bei Addition von 1 immer den gleichen Wert haben. D.h. der Befehl

```
add ziel, 1
```

setzt genau dann das Carry-Flag, wenn das Ergebnis null ist und dementsprechend das Zero-Flag gesetzt wird (z.B. bei einem 8-Bit-Register für ziel=FFh)

Bei Subtrahieren von 1 mit

```
sub ziel, 1
```

wird genau dann das Carry-Flag gesetzt, wenn vorher ziel=0 war. Auch dies ist leicht zu prüfen, so dass auf das Carry-Flag verzichtet werden kann

Der Assemblerbefehl LOOP

Der Befehl `loop` wurde bereits bei den bedingten Sprüngen erwähnt

```
loop ziel
```

Ziel kann eine Sprungmarke oder eine Adresse sein

Beim Aufruf des Befehls wird das (E)CX-Register dekrementiert und der Sprung ausgeführt, wenn (E)CX ungleich null ist

Ist `ziel` eine 16-Bit Adresse, wird das CX- und bei 32-Bit das ECX-Register betrachtet

Die Funktion `smallGauss` kann somit nochmal verbessert werden: (main.cpp)

```
unsigned int smallGauss(unsigned int varA) { // version 3
    __asm
    {
                    mov eax, 0
                    mov ecx, varA
                    jecxz EndLoop
     ExecuteLoop:   add eax, ecx
                    loop ExecuteLoop
     EndLoop:
    }
}
```

Der Assemblerbefehl MUL

Der Befehl `mul` führt eine Multiplikation für vorzeichenlose Zahlen aus

Hierbei wird von dem üblichen Schema:

```
op ziel, quelle
```

abgewichen, da das Ergebnis im Allgemeinen nicht in ein Register passt

Stattdessen wird das Ziel, welches mehrere Register umfassen kann, fest vorgegeben und der Mul-Befehl benötigt nur die Quelle als Parameter:

```
mul quelle
```

Welche Register als Ziel verwendet werden hängt von der Bitbreite von `quelle` ab:

Bitbreite	Ziel	Operation
8	`AX`	`AX = AL * quelle`
16	`DX:AX`	`DX:AX = AX * quelle`
32	`EDX:EAX`	`EDX:EAX = EAX * quelle`

Überschreitet das Ergebnis die Bitbreite von `quelle` werden Carry- und Overflow-Flag gesetzt

Der Assemblerbefehl MUL

Beispiel:

```
int main() {
  __asm {
    mov cl, 16          ; // CL=10h (8-bit register)
    mov al, 8           ; // AL=08h
    mul cl              ; // AX=0080h=128                Flags: OV=0, CY=0
    mov cx, 4000h       ; // CX=4000h (16-bit register)
    mov ax, 5000h       ; // AX=5000h
    mul cx              ; // DX:AX= 1400:0000            Flags: OV=1, CY=1
    mov ecx, 7FFFFFFFh ; // ECX=7FFFFFFFh (32-bit register)
    mov eax, 4          ; // EAX=00000004h
    mul ecx             ; // EDX:EAX=00000001:FFFFFFFC   Flags: OV=1, CY=1
  }
  return 0;
}
```

Quelldatei: main.cpp

Der Assemblerbefehl DIV

Der Assemblerbefehl `div` führt eine Division aus

```
div quelle
```

Das Ergebnis der Division ist ein ganzzahliger Anteil und ein Rest

Der Divisor wird mit `quelle` angegeben

Der Dividend muss in fest vorgegebenen Registern abgelegt werden. Welche dies sind, ist abhängig von der Bitbreite von `quelle`:

Bitbreite	Dividend	Ganzzahliges Ergebnis	Rest
8	`AX`	`AL = AX / quelle`	`AH = AX mod quelle`
16	`DX:AX`	`AX = DX:AX / quelle`	`DX = DX:AX mod quelle`
32	`EDX:EAX`	`EAX = EDX:EAX / quelle`	`EDX = EDX:EAX mod quelle`

Die Division ist demnach so implementiert, dass sie die exakte Umkehrfunktion einer Multiplikation realisiert

Flags werden nicht verändert

Der Assemblerbefehl DIV

Beispiel:

```
int main() {
  __asm {
    mov cx, 4000h      ; // CX=4000h (16-bit register)
    mov ax, 5000h      ; // AX=5000h
    mul cx             ; // DX:AX=1400:0000            Flags: OV=1, CY=1
    div cx             ; // AX=5000h DX=0000h
    mov dx, 0000h      ;
    mov ax, 0002h      ; // DX:AX=0000:0002h
    mov cx, 10h        ; // CX=10h
    div cx             ; // AX=0000h DX=0002h
    mov dx, 7FFFh      ;
    mov ax, 0000h      ; // DX:AX=7FFF:0000h
    mov cx, 2h         ; // CX=2h
    div cx             ; // result does not fit in AX, throws divide error exception
  }
  return 0;
}
```

Quelldatei: main.cpp

Der Assemblerbefehl DIV

Beispiel: Größter gemeinsamer Teiler (ggT) mit dem Euklidischen Algorithmus:

```
unsigned int ggT_C(unsigned int a, unsigned int b) {
  unsigned int res = 0;
  while(b != 0) { // while b not zero
    res = (a % b); // res = (a mod b)
    a = b;
    b = res;
  }
  return a;
}
```

Z.B. Größter gemeinsamer Teiler von 525 und 399:

$$525 = 1 \cdot 399 + 126$$
$$399 = 3 \cdot 126 + 21$$
$$126 = 6 \cdot 21 + 0$$

Ergebnis: 21

Der Assemblerbefehl DIV

Größter gemeinsamer Teiler (ggT) mit dem Euklidischen Algorithmus in Assembler

```
unsigned int ggT_Asm(unsigned int a, unsigned int b) {
  __asm {
              mov edx, 0    ;
              mov eax, a    ; // EDX:EAX = a
              mov ecx, b    ; // ECX = b
    WhileLoop: cmp ecx, 0    ;
              jz EndLoop    ; // jump to "done" if ECX == 0
              div ecx       ;
              mov eax, ecx  ; // EAX = ECX
              mov ecx, edx  ; // ECX = (a mod b)
              mov edx, 0    ; // EDX = 0 for next "div" command
              jmp WhileLoop ;
    EndLoop:
  }
}
```

Quelldatei: main.cpp

Der Assemblerbefehl IMUL

Der Befehl `IMUL` führt eine Multiplikation von vorzeichenbehafteten Zahlen aus

Es gibt drei verschiedene Varianten:

Die Syntax entspricht dem MUL-Befehl

```
imul quelle
```

Das Ergebnis der Multiplikation von `quelle` und `ziel` wird in `ziel` gespeichert

```
imul ziel quelle
```

Das Ergebnis der Multiplikation von `operandA` und `konstante` wird in `ziel` gespeichert

```
imul ziel operandA konstante
```

Bei der 2. und 3. Variante werden die niederwertigen Bits des Ergebnisses abgeschnitten falls es nicht in `ziel` passen sollten. Carry- und Overflow-Flag werden in diesem Fall gesetzt

Quelldatei für das Beispiel auf der nächsten Folie: main.cpp

Der Assemblerbefehl IMUL

```
#include <stdio.h>
int main() {
  int result = 0;
  __asm {
    mov eax, -2
    mov ecx, 800
    imul ecx
    mov result, eax   // EDX is ignored here
  }
  printf("result=%d \n", result);
  __asm {
    mov eax, -2
    mov ecx, 800
    imul eax, ecx
    mov result, eax
  }
  printf("result=%d \n", result);
  __asm {
    mov ecx, 800
    imul eax, ecx, -2
    mov result, eax
  }
  printf("result=%d \n", result);
  return 0;
}
```

Der Assemblerbefehl IDIV

Der Befehl `IDIV` führt eine Division von vorzeichenbehafteten Zahlen aus

Der Befehl wird äquivalent zum Befehl `DIV` verwendet

```
#include <stdio.h>
int main() {
  int result = 0;
  int remainder = 0;
  __asm {
    mov edx, 0
    mov eax, 800
    mov ecx, -3
    idiv ecx
    mov result, eax
    mov remainder, edx
  }
  printf("result=%d \n", result);
  printf("remainder=%d \n", remainder);
  return 0;
}
```

Quelldatei: main.cpp

Der Assemblerbefehl NEG

Der Assemblerbefehl neg ändert das Vorzeichen einer vorzeichenbehafteten Zahl

Beispiel:

```
int main() {
  __asm {
    mov eax, -2      ; // EAX=FFFFFFFEh=-2
    neg eax          ; // EAX=00000002h= 2
    mov al, 254      ; // AL=FEh=-2
    neg al           ; // AL=02h= 2
  }
}
```

Quelldatei: main.cpp

Die Assemblerbefehle AND, OR, NOT

Die Assemblerbefehle `and`, `or`, und `not` führen elementare logische Grundoperationen aus, wie diese aus der booleschen Algebra bekannt sind

Konjunktion (AND): $y = a \wedge b$

a	b	$y = a \wedge b$
0	0	0
0	1	0
1	0	0
1	1	1

Disjunktion (OR): $y = a \vee b$

a	b	$y = a \vee b$
0	0	0
0	1	1
1	0	1
1	1	1

Negation (NOT): $y = \neg a$

a	$y = \neg a$
0	1
1	0

Die Assemblerbefehle AND, OR, NOT

Dabei wird jedes Bit einzeln betrachtet

Das Carry-Flag CY und Overflow-Flag OF werden auf null gesetzt. Die Flags PL, ZR und PE werden gemäß des Ergebnisses gesetzt. Das Aux. Carry AC wird nicht beeinflusst.

Beispiel:

```
int main() {
  unsigned int a = 0x000078F1;
  unsigned int b = 0x00007F8F;
  __asm {
    mov eax, a    ; // EAX=000078F1h
    not eax       ; // EAX=FFFF870Eh
    and eax, b    ; // EAX=0000070Eh
    mov edx, b    ; // EDX=0000070Eh
    not edx       ; // EDX=FFFF8070h
    and edx, a    ; // EDX=00000070h
    or eax, edx   ; // EAX=0000077Eh
  }
  return 0;
}
```

Frage: Was wurde hier gerade umständlich implementiert?

Der Assemblerbefehl XOR

Antwort: XOR

```
int main() {
  unsigned int a = 0x000078F1;
  unsigned int b = 0x00007F8F;
  __asm {
    mov eax, a    ; // EAX=000078F1h
    not eax       ; // EAX=FFFF870Eh
    and eax, b    ; // EAX=0000070Eh
    mov edx, b    ; // EDX=0000070Eh
    not edx       ; // EDX=FFFF8070h
    and edx, a    ; // EDX=00000070h
    or eax, edx   ; // EAX=0000077Eh
  }

  // simple alternative
  __asm {
    mov eax, a     ; // EAX=000078F1h
    xor eax, b     ; // EAX=0000077Eh
  }
  return 0;
}
```

Quelldatei: main.cpp

Die Assemblerbefehle SHL und SHR

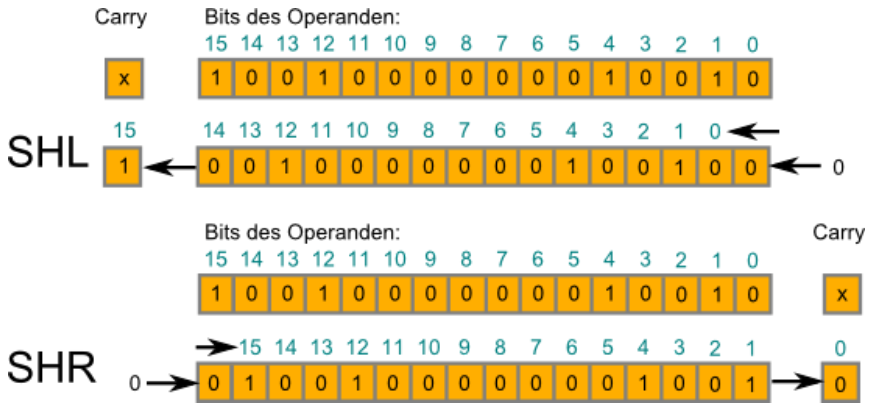
Der Befehl `shl` ("Shift logical left") und `shr` ("Shift logical right") schieben das Ziel um die angegeben Anzahl an Bits nach links bzw. rechts

```
shl ziel, anzahl
shr ziel, anzahl
```

Eine Verschiebung nach links um n Bits entspricht einer Multiplikation mit 2^n

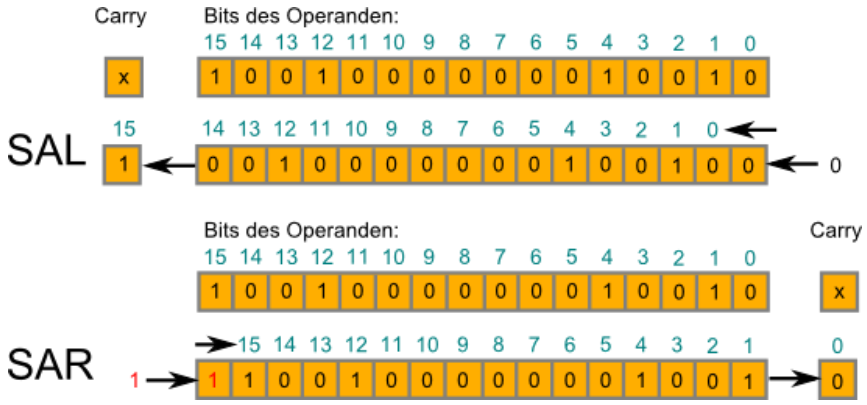
Eine Verschiebung nach rechts entspricht einer ganzzahligen Division mit 2^n

Das jeweils letzte herausgeschobene Bit landet im Carry:



Die Assemblerbefehle SAL und SAR

Der Befehle `sal` ("Shift arithmetic left") und `sar` ("Shift arithmetic right") entsprechen den Befehlen `shl` und `shr`. Der einzige Unterschied ist, dass bei `sar` statt einer 0 eine 1 hineingeschoben wird



Der Assemblerbefehl XCHG

Der Befehl `xchg` ("exchange") tauscht den Inhalt der beiden Operanden aus

Beispiel:

```
int main() {
  __asm {
    mov eax, 1    ; // EAX=1
    mov edx, 2    ; // EDX=2
    xchg eax, edx ; // EDX=1, EAX=2
  }
  return 0;
}
```

Quelldatei: main.cpp

Die Assemblerbefehle BT, BTS, BTR, BTC

Die "Bit Test"-Befehle, `bt`, `bts`, `btr` und `btc`, kopieren das durch `bitIndex` indizierte Bit ins Carry-Flag und modifizieren dieses Bit in `ziel` entsprechend der Semantik der Mnemonik

```
bt ziel, bitIndex
```

Befehl	Wirkung auf das Carry cy	Wirkung auf das indizierte Bit
`bt` ("bit test")	`cy=ziel[bitIndex]`	`ziel[bitIndex]` unbeeinflusst
`bts` ("bit test and set")	`cy=ziel[bitIndex]`	`ziel[bitIndex]=1`
`btr` ("bit test and reset")	`cy=ziel[bitIndex]`	`ziel[bitIndex]=0`
`btc` ("bit test and complement)")	`cy=ziel[bitIndex]`	`ziel[bitIndex] = not(ziel[bitIndex])`

Die Assemblerbefehle CLC und STC

Die Assemblerbefehle `clc` ("clear carry") und `stc` ("set carry") löschen bzw. setzen das Carry-Flag

Beispiel:

```
int main() {
  __asm {
    stc             ; // CY=1
    clc             ; // CY=0
    mov ax, 00FFh   ; // AX=00FFh
    bt  ax, 0       ; // CY=1
    bt  ax, 7       ; // CY=1
    bt  ax, 8       ; // CY=0
    bts ax, 8       ; // CY=0, AX=01FFh
  }
  return 0;
}
```

Quelldatei: main.cpp

Der Assemblerbefehl NOP

Der Assemblerbefehl `nop` ("no operation") führt keine Operation aus

Er dient z.B. dazu, die Ablaufgeschwindigkeit von Maschinespracheprogrammen zu beeinflussen, da ein NOP-Befehl jeweils eine definierte Verzögerung erzeugt

Weitere Assemblerbefehle der x86 Prozessorfamilie

Die hier beispielhaft gezeigten Befehle sind natürlich nur ein Auszug aus den insgesamt ca. 200 Mehrzweckbefehlen und weiteren 300 Spezialbefehlen

Für viele einfache Assemblerprogramme sind die hier gezeigten Befehle jedoch bereits ausreichend

Einige weitere Befehle werden in den folgenden Kapiteln vorgestellt

Die komplette Dokumentation der x86 Architektur steht auf den Webseiten von Intel zur Verfügung:

Intel 64 and IA-32 Architectures Software Developer's Manual

- Volume 1: Basic Architecture
- Volume 2: Instruction Set Reference, A-Z
- Volume 3: System Programming Guide

Die x86-Befehlsreferenz

Die Dokumentation eines Assembler-Befehls besteht jeweils aus eine großen Tabelle, die zeigt für welche Operanden der Befehl vorhanden ist

Hier z.B. die Tabelle für den Befehl `add` (Seite 3-31 der Befehlsreferenz):

ADD—Add

Opcode	Instruction	Op/ En	64-bit Mode	Compat/ Leg Mode	Description
04 *ib*	ADD AL, *imm8*	I	Valid	Valid	Add *imm8* to AL.
05 *iw*	ADD AX, *imm16*	I	Valid	Valid	Add *imm16* to AX.
05 *id*	ADD EAX, *imm32*	I	Valid	Valid	Add *imm32* to EAX.
REX.W + 05 *id*	ADD RAX, *imm32*	I	Valid	N.E.	Add *imm32* sign-extended to 64-bits to RAX.
80 /0 *ib*	ADD *r/m8*, *imm8*	MI	Valid	Valid	Add *imm8* to *r/m8*.
REX + 80 /0 *ib*	ADD *r/m8**, *imm8*	MI	Valid	N.E.	Add sign-extended *imm8* to *r/m64*.
81 /0 *iw*	ADD *r/m16*, *imm16*	MI	Valid	Valid	Add *imm16* to *r/m16*.
81 /0 *id*	ADD *r/m32*, *imm32*	MI	Valid	Valid	Add *imm32* to *r/m32*.
REX.W + 81 /0 *id*	ADD *r/m64*, *imm32*	MI	Valid	N.E.	Add *imm32* sign-extended to 64-bits to *r/m64*.
83 /0 *ib*	ADD *r/m16*, *imm8*	MI	Valid	Valid	Add sign-extended *imm8* to *r/m16*.
83 /0 *ib*	ADD *r/m32*, *imm8*	MI	Valid	Valid	Add sign-extended *imm8* to *r/m32*.
REX.W + 83 /0 *ib*	ADD *r/m64*, *imm8*	MI	Valid	N.E.	Add sign-extended *imm8* to *r/m64*.
00 /r	ADD *r/m8*, *r8*	MR	Valid	Valid	Add *r8* to *r/m8*.
REX + 00 /r	ADD *r/m8**, *r8**	MR	Valid	N.E.	Add *r8* to *r/m8*.
01 /r	ADD *r/m16*, *r16*	MR	Valid	Valid	Add *r16* to *r/m16*.
01 /r	ADD *r/m32*, *r32*	MR	Valid	Valid	Add *r32* to *r/m32*.
REX.W + 01 /r	ADD *r/m64*, *r64*	MR	Valid	N.E.	Add *r64* to *r/m64*.
02 /r	ADD *r8*, *r/m8*	RM	Valid	Valid	Add *r/m8* to *r8*.
REX + 02 /r	ADD *r8**, *r/m8**	RM	Valid	N.E.	Add *r/m8* to *r8*.
03 /r	ADD *r16*, *r/m16*	RM	Valid	Valid	Add *r/m16* to *r16*.
03 /r	ADD *r32*, *r/m32*	RM	Valid	Valid	Add *r/m32* to *r32*.
REX.W + 03 /r	ADD *r64*, *r/m64*	RM	Valid	N.E.	Add *r/m64* to *r64*.

Assembler

```
add eax, 0A0B0C0Dh
```

Maschinensprache

```
05 0D 0C 0B 0A
```

Opcode imm32 (Little Endian)

Die x86-Befehlsreferenz

Es fällt auf, dass je nach Art der angegebenen Operanden ein anderer Opcode verwendet wird

D.h. der verwendeten Maschinenbefehl (Opcode) bestimmt sich zum Einen aus der Mnemonik, ist aber auch abhängig von den Operanden

Abkürzungen:

`r8`, `r16`, `r32` Operand ist eine 8-, 16-, bzw. 32-Bit-Mehrzweckregister

`m8`, `m16`, `m32` Operand ist eine Speicheradresse (engl. "memory")

`m/r8`, `m/r16`, `m/r32` Operand ist ein Register oder eine Speicheradresse

`imm8`, `imm16`, `imm32` Operand ist eine konstante vorzeichenbehaftete Zahl (engl. "immediate value")

Eine komplette Liste der Abkürzungen ist in Kapitel 3.1.1.3 (Seite 3-5) der Befehlsreferenz angegeben

Neben der Tabelle und der Beschreibung eines Befehls ist für einen Programmierer besonders der Abschnitt über die veränderten Flags ("Flags Affected") des jeweiligen Befehls wichtig

Verkürzte Befehlsreferenz für die Klausur (1 von 2)

Assemblerbefehl	Beschreibung
`ADC ziel, quelle`	Addiere mit Übertrag aus vorangegangener Addition
`ADD ziel, quelle`	`ziel = ziel + quelle`
`AND ziel, quelle`	Logische Und-Verknüpfung von `ziel` und `quelle`
`CMP x, y`	Vergleiche `x` und `y`
`DEC x`	Dekrementiere `x`
`DIV op`	Teile durch `op`
`IDIV op`	Vorzeichenbehaftete, ganzzahlige Division
`IMUL op`	Vorzeichenbehaftete, ganzzahlige Multiplikation
`INC x`	Inkrementiere `x`
`JA sprungziel`	Springe, wenn größer
`JAE sprungziel`	Springe, wenn größer oder gleich
`JB sprungziel`	Springe, wenn kleiner
`JBE sprungziel`	Springe, wenn kleiner oder gleich
`JECXZ sprungziel`	Springe, wenn `ECX` gleich 0
`JE sprungziel`	Springe, wenn gleich
`JMP sprungziel`	Unbedingter Sprung

Verkürzte Befehlsreferenz für die Klausur (2 von 2)

Assemblerbefehl	Beschreibung
`JO sprungziel`	Springe, wenn Überlauf
`JZ sprungziel`	Springe, wenn gleich 0
`LOOP sprungziel`	Dekrementiere `ECX`, springe falls `ECX` ungleich 0
`MOV ziel, quelle`	Kopiere `quelle` nach `ziel`
`MUL op`	Multipliziere mit `op`
`NEG ziel`	Negiere Wert in `ziel` mit Zweierkomplement
`NOT ziel`	Logische Negation von `ziel`
`OR ziel, quelle`	Logische Oder-Verknüpfung von `ziel` und `quelle`
`POP ziel`	Hole obersten Wert vom Stack
`PUSH quelle`	Lege Wert aus `quelle` auf Stack
`SAL ziel, schrittzahl`	Arithmetische, bitweise Verschiebung nach links
`SAR ziel, schrittzahl`	Arithmetische, bitweise Verschiebung nach rechts
`SHL ziel, schrittzahl`	Logische, bitweise Verschiebung nach links
`SHR ziel, schrittzahl`	Logische, bitweise Verschiebung nach rechts
`SUB ziel, quelle`	`ziel = ziel - quelle`
`XOR ziel, quelle`	Logische XOR-Verknüpfung von `ziel` und `quelle`

Gibt es Fragen?

Anregungen oder Verbesserungsvorschläge können auch gerne per E-mail an mich gesendet werden: Kontakt

Weitere Vorlesungsfolien