

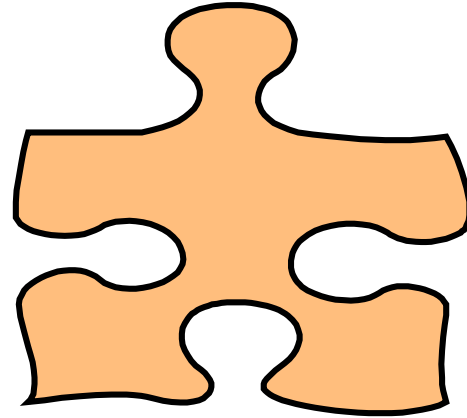
# „Systemnahe Programmierung“ (BA-INF 034) Wintersemester 2023/2024

Dr. Matthias Frank, Dr. Matthias Wübbeling

Institut für Informatik 4  
Universität Bonn

E-Mail: [{matthew, matthias.wuebbeling}@cs.uni-bonn.de](mailto:{matthew, matthias.wuebbeling}@cs.uni-bonn.de)  
Sprechstunde: nach der Vorlesung bzw. nach Vereinbarung

# 1. Maschinenprogrammierung in Assembler



1. Teil: Maschinenprogrammierung in Assembler  
(nahe Anlehnung an Inhalte der Systemnahen Informatik BA-INF 023)

„Wir werden hier keine Assembler-Profis, aber Assembler-Routinen werden bei Bedarf in Programme anderer (Hoch-) Sprachen eingebunden.“

[1.1. Motivation](#)

[1.2. Bezug zur Systemnahen Informatik \(SS 2023\)](#)

[1.3. Allgemeines zu 80x86 Assembler-Programmierung](#)

[1.4. Calling Conventions \(aus C unter Linux\)](#)

[1.5. Bezug zur 2-Adressmaschine \(Systemnahe Informatik, SS 2023\)](#)

[1.6. Assembler-Programmbeispiele](#)

[1.7. Socket-/Netzwerkprogrammierung in Assembler](#)

[1.8. ARM-Assembly](#)

[1.9. MIPS-Assembly](#)

[1.10. Zusammenfassung](#)

# 1.1. Motivation

## 2.2. Übersetzung höherer Programmiersprachen

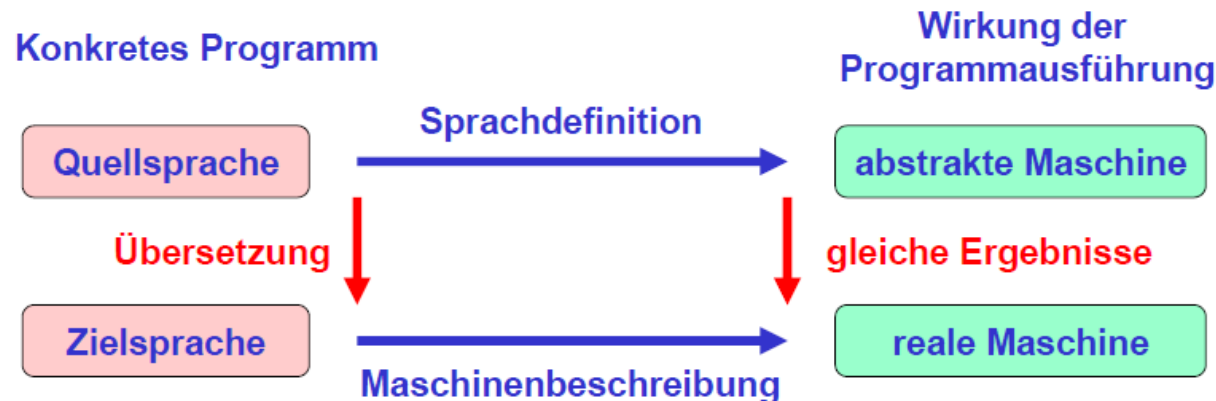
Ein **Übersetzer** erzeugt

- zu einem **Programm in einer Quellsprache**
- ein **äquivalentes Programm in einer Zielsprache**.

**Quellsprache A** → **Zielsprache B**

Ist die **Quellsprache mächtiger als die Zielsprache**, dann sprechen wir auch von einem „**Compiler**“ (to compile: zusammentragen).

Der wichtigste Fall ist die Übersetzung von höherer Programmiersprache in Maschinensprache:



Ein Compiler erzeugt eine Code-Datei für eine spezifische Rechnerplattform (PC, Sun, MAC, ...)

(aus Kapitel „2. Vom Programm zum lauffähigen Code“, Systemnahe Informatik (BA-INF 023, SS 2023, Prof. Dr. Peter Martini)



# Kenntnisse zu Maschinenprogrammierung / Assembler

- Selbst programmieren in Maschinensprache/Assembler
  - Programmierung von sog. **Embedded** Devices (Digitalkamera, PDA, Mobile Phone, ...)
  - **hoch optimierter Code** bzw. Teilprogramme (kommt eher selten vor ...)
- Programme/Fragmente in Maschinensprache/Assembler verstehen/analysieren
  - Verständnis
    - **was passiert intern** in Programmen einer Hochsprache?  
(C, C++, PASCAL, ...)
    - warum **verhält sich ein Programm so** wie es ist?
  - Analyse
    - Analyse von sog. **Malware**
    - Binary Audit/**Reverse Code Engineering**
    - Suche von Bugs (3rd Party Software)
    - **Patchen** (oder cracken) von Software

```
void rt_update_table (peer *p)
{
    int bucketpos, firstpos, bucketno;
    rt_entry *bucketentry;
    char *nullhash =
        „\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00“
    ;

    if (!p)
        return;
    if (memcmp(p->hash, self.hash, 0x10) == 0)
        return;
    if (memcmp(p->hash, nullhash, 0x10) == 0)
        return;
    if ((bucketno = xor_dist(p->hash, self.hash)) == -1)
        return;
}
```



# Historische Entwicklung der Programmiersprachen

- Die ersten Programmierer mussten ihre **Algorithmen in „Maschinensprache“** darstellen. Hierzu war es erforderlich, umfassendes Wissen über den **Aufbau** und den spezifischen **Befehlsvorrat der jeweiligen Maschine** zu haben.
- In zunehmendem Umfang gelang es aber, **mächtigere Programmiersprachen** zu entwickeln, die ein **Denken und Arbeiten mit Begriffen und Strukturen des Anwendungsbereiches** ermöglichen.

Problemlösung in einer Umgebung, in der der **Mensch den Charakteristika des Rechners gerecht** werden muß.



...



Problemlösung in einer Umgebung, in der der **Rechner den Charakteristika des Menschen gerecht** werden muß.



1.1.1. Programmierung in Maschinensprache

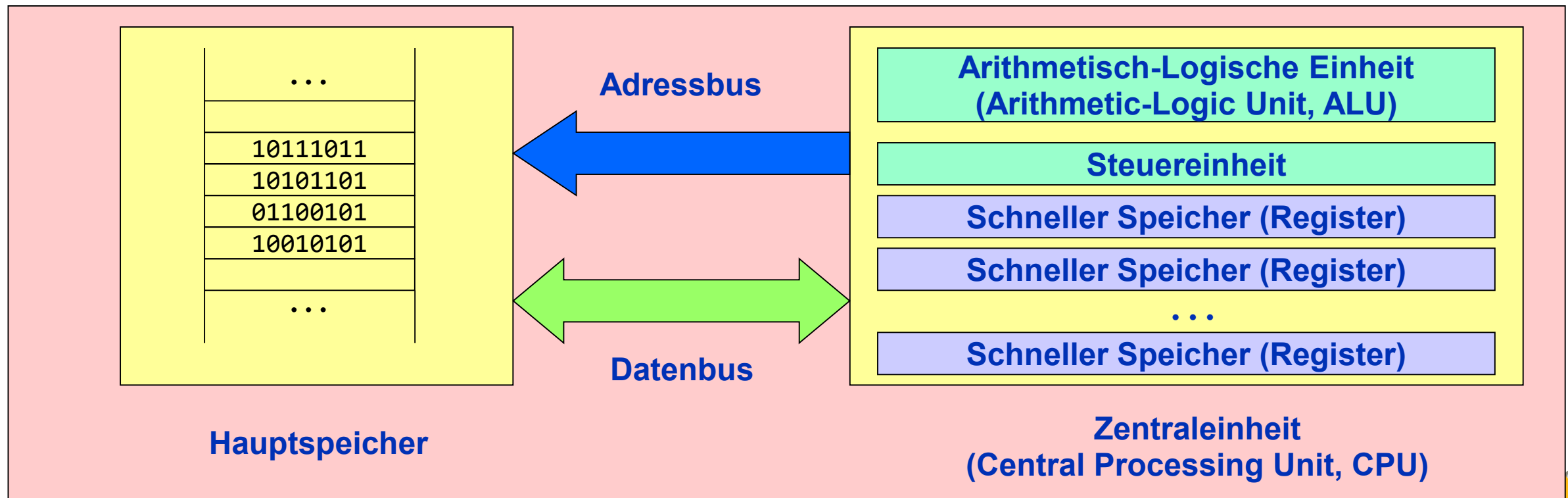
1.1.2. Programmierung in Assembly-Sprachen

1.1.3. Maschinenunabhängige Programmierung

*Ein Teil dieser Folien stammt aus der damaligen Informatik-I-Vorlesung von Prof. Martini WS 99/00!*

# 1.1.1. Programmierung in Maschinensprache

- Rechner arbeiten **intern** mit **sehr elementaren Befehlen**:
  - **Zugriff auf Speicherzellen** des Hauptspeichers (Daten bzw. Befehle holen / abspeichern)
  - **Zugriff auf** (einige wenige) **Register** (schnelle Speicher) innerhalb der Zentraleinheit
  - **Arithmetische** oder **logische Operationen** auf Daten in Registern
  - **Sprünge zu** anderen **Befehlen**, die nachfolgend ausgeführt werden



# Befehlsdarstellung in Maschinencode

- Jede **CPU** hat einen **spezifischen Befehlsvorrat**. Intern: **binäre Befehlsdarstellung!**
- Zur Darstellung von **Algorithmen in „Maschinencode“** würde der Mensch sicher eine Hexadezimaldarstellung vorziehen.

## Beispiel: Addition zweier ganzer Zahlen

<b>Algorithmus:</b>	<b>1. Schritt:</b>	<b>Hole</b> die <b>erste Zahl</b> aus dem Speicher und speichere sie <b>in ein „Register“*</b> (schneller Speicher).
	<b>2. Schritt:</b>	<b>Hole</b> die <b>zweite Zahl</b> aus dem Speicher und speichere sie <b>in ein anderes Register</b> .
	<b>3. Schritt:</b>	Aktiviere eine <b>„Additionsschaltung“</b> , welche die <b>Registerinhalte</b> mittels Addition <b>verknüpft</b> und das <b>Ergebnis</b> in einem dritten Register <b>abspeichert</b> .
	<b>4. Schritt:</b>	<b>Lege</b> das <b>Ergebnis</b> im Speicher <b>ab</b> .

\* Näheres zu Registern: Sommersemester

<b>Maschinencode:</b>	<b>1. Schritt:</b>	0001010101101100	<b>156C</b>
	<b>2. Schritt:</b>	0001011001101101	<b>166D</b>
	<b>3. Schritt:</b>	0101000001010110	<b>5056</b>
	<b>4. Schritt:</b>	0011000001101110	<b>306E</b>



# 1.1.2. Programmierung in Assembler-Sprachen

Offenbar ist auch Hex-Darstellung für Menschen **nicht zumutbar**.

Daher wurden schon früh sog. „**Assembler-Sprachen**“ entwickelt, die eine **Darstellung mit leichter verständlichen mnemonischen\* Symbolen** zulassen.

<b>Assemblercode:</b>	<b>LD R5, PREIS</b>	für	<b>156C</b>
	<b>LD R6, STEUER</b>	für	<b>166D</b>
	<b>ADDI R0, R5, R6</b>	für	<b>5056</b>
	<b>ST R0, TOTAL</b>	für	<b>306E</b>

156C	1: Befehl „Load“	5: Nummer des Registers	6C: Adresse der gewünschten Speicherzelle
166D	1: Befehl „Load“	6: Nummer des Registers	6D: Adresse der gewünschten Speicherzelle
5056	5: Befehl „ADDI“	0: Nummer des Registers für das Ergebnis	5: Nummer des Registers des einen Summanden 6: Nummer des Registers mit dem anderen Summanden
306E	3: Befehl „Store“	0: Nummer des Registers	6E: Adresse der gewünschten Speicherzelle

\* Mnemonik = Mnemotechnik: Die Kunst, das Einprägen von Gedächtnisstoff durch Lernhilfen zu erleichtern.



# Vor- und Nachteile der Assembler-Programmierung

Informatik-I  
Prof. Martini  
WS 99/00

## Vorteile der Programmierung in Assembler-Sprachen:

- **Angenehmere Programmierung** als in Maschinencode
- **Ermöglichung geschickter Nutzung des Befehlsvorrats** der jeweiligen Maschine
  - Dadurch: maximale Effizienz (Laufzeit, Speicherplatzbedarf) erzielbar
- **Leichte automatische Umsetzung** von Assemblercode in Maschinencode
  - (durch den sog. „Assembler“)

## Nachteil der Programmierung in Assembler-Sprachen:

- **Abhängigkeit** der Programme **von einer bestimmten Maschine**
  - Beim Wechsel auf einen anderen Maschinentyp müssen die Programme komplett neu geschrieben werden: Andere Befehle, andere Registerstruktur.
- **Programmentwicklung in sehr kleinen Schritten**
  - Der Programmierer muss **komplexe Lösungen aus winzigen Teilen konstruieren**.
- **Analogie:** Entwurf eines **Hauses** auf der Basis **von Brettern, Nägeln, Scheiben, ...**

# 1.1.3. Maschinenunabhängige Programmierung

Informatik-I  
Prof. Martini  
WS 99/00

**Idee:** Definiere „mächtigere“ Befehle, die

- **automatisch in** Befehle von **Assembler-Sprachen umsetzbar** und
- **unabhängig vom spezifischen Befehlsvorrat** bestimmter Rechner sind.
  - („Laden“, „Springen“, ..., gibt es fast überall, auch wenn es anders heißt)

Programme in den sog. „**höheren**“ = „**problemorientierten**“ **Programmiersprachen** machen (im Gegensatz zu den maschinenorientierten Programmiersprachen) **keinen Gebrauch von spezifischen Charakteristika spezieller Maschinen.**

**Theoretisch:** sind sie damit **auf allen Rechnern einsetzbar.**

**Praktisch:**

- gibt es auch bei standardisierten Sprachen **häufig Erweiterungen**, die leider auch genutzt werden.
- Derartige Erweiterungen werden aber **nicht überall unterstützt.**

**Beispiel:**

**assign** total **the value** preis + steuer



```
LD R5, PREIS  
LD R6, STEUER  
ADDI R0, R5, R6  
ST R0, TOTAL
```



# 1.2. Bezug zur Systemnahen Informatik (SS 2023)

Systemn. Inf.  
Prof. Martini  
SS 2023

- Mit den folgenden Folien dieses Unterkapitels 1.2. soll auf die wichtigen vorangegangenen Inhalte der **Vorlesung Systemnahe Informatik** (BA-INF 023) aus den Bereichen der maschinennahen Programmierung hingewiesen werden.
- Die (eingeklammerten) Kapitelnummerierungen in der Titelzeile sind die aus der Vorlesung Systemnahe Informatik (des SS 2023) mit besonderem Augenmerk auf den folgenden (dortigen) Unterkapiteln:

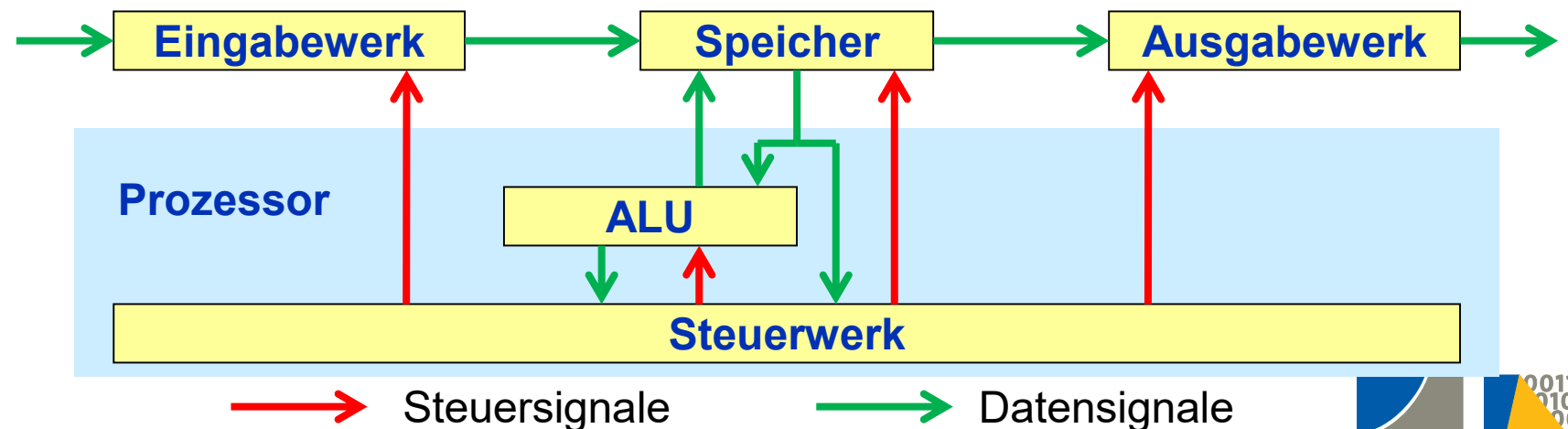
1.2.1 Der Von-Neumann-Rechner  
1.2.2.2 Interne Struktur von BORIS  
1.2.2.3 Ein einfaches Programm und dessen Bearbeitung  
1.3.2 Prinzipielle Gestalt von Maschinenbefehlen  
1.3.3 Assembler-Sprachen  
Pseudoassembler (a-Notation)  
1.3.6 Vom Assemblerprogramm zum Maschinen-Code

# (1.2.1.) Der Von-Neumann-Rechner

**Der Von-Neumann-Rechner besteht aus fünf Funktionseinheiten.**

- **Steuerwerk** (control unit)
  - **Laden** und **Decodieren** der Programmbefehle, **Koordinieren** der Befehlsausführung.
- **Arithmetisch-logische Einheit** (arithmetic logical unit, ALU)
  - Ausführung arithmetischer und logischer Operationen **unter Kontrolle des Steuerwerks**; Bereitstellung der Operanden durch das Steuerwerk.
- **Speicher** (memory)
  - Einteilung des Speichers in **fortlaufend nummerierte, gleich große Zellen**
  - Zugriff (Lesen/Schreiben) auf Zellinhalte über ihre Nummer (**Adresse**)

- **Eingabewerk**
- **Ausgabewerk**



Die Kontrolleinheit hat mehr Verbindungen zu den Komponenten des Mikroprozessors, als in der Graphik gezeigt.



# (1.2.2.3.) Ein einfaches Programm und dessen Bearbeitung

BORIS soll nun ein einfaches Programm ausführen, das **ab Position 100 im Speicher** liegt und die folgenden Befehle umfasst:

<b>LOAD</b>	<b>A, [10]</b>
<b>ADD</b>	<b>A, [11]</b>
<b>STORE</b>	<b>A, [12]</b>

**LOAD A, [10]**

Lade den Akkumulator mit dem Inhalt der Speicherzelle 10.

**ADD A, [11]**

Addiere den Inhalt der Speicherzelle 11 zum Inhalt des Akkumulators und lege das Ergebnis im Akkumulator ab.

**STORE A, [12]**

Lege den Inhalt des Akkumulators in Speicherzelle 12 ab.

Die Zahl  $2_{10}$   
Die Zahl  $3_{10}$   
reserviert

**LOAD A, [10]**  
**ADD A, [11]**  
**STORE A, [12]**

...	Adressen
xxxxxxxx	8
xxxxxxxx	9
00000010	10
00000011	11
xxxxxxxx	12
xxxxxxxx	13
xxxxxxxx	14
⋮	
xxxxxxxx	98
xxxxxxxx	99
10011010	100
11001011	101
11101100	102
xxxxxxxx	103
xxxxxxxx	104
...	

Systemn. Inf.  
Prof. Martini  
SS 2023



## (1.3.2.) Prinzipielle Gestalt von Maschinenbefehlen

Maschinenbefehle werden (wie Daten) in Worten gespeichert, manchmal auch in Halbworten, Doppelworten oder Mehrfachworten.

Ein Befehl hat prinzipiell die folgende Gestalt:

Format	Op-Code	Daten 1	Daten 2	...	Daten k	Ziel	Folge
--------	---------	---------	---------	-----	---------	------	-------

**Format** (kann entfallen, falls Op-Code eindeutig):

- Angabe der **Länge** und der **Positionen** der einzelnen **Felder**.

**Op-Code:**

- Angabe der auszuführenden **Operation**

**Daten** (Wo sind die Operanden?):

- **unmittelbar** („immediate“, Angabe von Konstanten im Maschinencode selbst),
- **implizit** (Akkumulator oder Stack, Art der Bereitstellung folgt aus Op-Code),
- **direkt** (Angabe der Adressdarstellung im Befehl) oder
- **indirekt** (Angabe der Adresse der Speicherzelle\*, die den Operanden enthält)

**Ziel** (Wo soll das Ergebnis gespeichert werden?):

- ggf. „Überdeckung“ (eine Quelle ist auch Ziel)
- ggf. „Implizierung“ (Akkumulator oder Stack als Ziel)

**Folge:**

- **Adresse des nächsten Befehls** („nächster“ Befehl oder Sprung),
- Angabe **entfällt bei sequentieller Abarbeitung**.

\* ggf. auch Angabe der Adresse der Speicherzelle, welche die Adresse enthält.

## (1.3.3.) Assembler-Sprachen

Systemn. Inf.  
Prof. Martini  
SS 2023

- Die sog. „Assembler\*-Sprachen“ sind **maschinenorientierte Programmiersprachen**.
- Im Gegensatz zu reinen Maschinensprachen gestatten Sie es dem Menschen, **statt** der **binär dargestellten Befehle** sehr viel leichter verständliche **mnemotechnische\*\* Symbole** einzusetzen.

Ein **Befehl einer Assembler-Sprache umfasst**

- **immer** die **Bezeichnung** der durchzuführenden **Operation**,
- **meist** maschinenspezifische **Angaben zu den Operanden**,
- **häufig** eine „**Marke**“ (label, symbolische Adresse) zur Kennzeichnung der Programmzeile.
  - Marken können u.a. als Operanden in Sprungbefehlen verwendet werden.

Vor der Ausführung muss ein Assembler-Programm in ein entsprechendes Maschinenprogramm übersetzt werden. Der hierzu erforderliche „**Assembler**“ muss

- **Befehle und Operanden in Binärcode umsetzen**,
- **Marken in Adressen umrechnen** und
- bestimmte „**Pseudobefehle**“ **bearbeiten** (z.B. Reservierung von Speicherplatz).

\* to assemble: zusammenbringen, -tragen, -setzen, -bauen, montieren.

\*\*Mnemonik = Mnemotechnik: Die Kunst, das Einprägen von Gedächtnisstoff durch Lernhilfen zu erleichtern.



## (1.3.3.) Pseudoassembler ( $\alpha$ -Notation)

- Ein Assembler ist stark maschinenabhängig. Gleiches gilt für die in der zugehörigen Sprache geschriebenen Programme.
- Im Rahmen der **Vorlesung** benutzen wir die mnemotechnischen Symbole der sog.  **$\alpha$ -Notation**.

Systemn. Inf.  
Prof. Martini  
SS 2023

Später lassen wir z.T. auch mehrere Akkumulatoren zu

$\alpha$       Akkumulator bzw. dessen Inhalt  
 $\rho(i)$     Inhalt der Speicherzelle mit Adresse  $i$   
 $+, -, \cdot, /$  Erlaubte Operationen

- Bei der  $\alpha$ -Notation ist die Verwendung von **Marken** zulässig
- sowohl zur Kennzeichnung von **Zellen des Datenspeichers**
- als auch zur Kennzeichnung von **Zellen des Programmspeichers**.

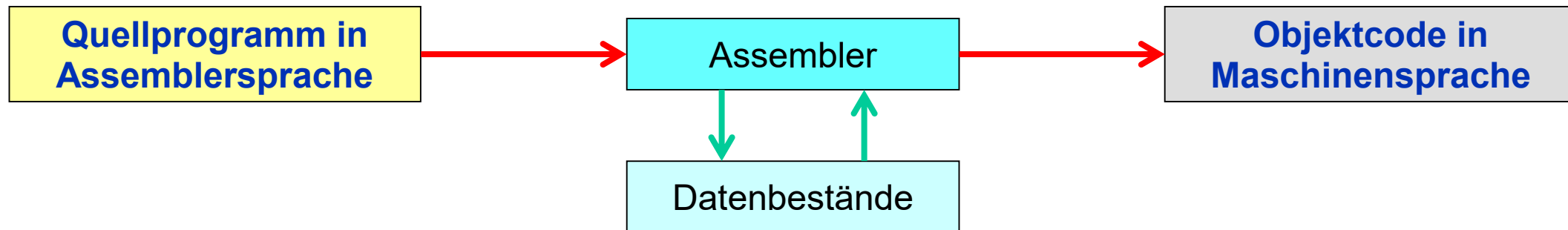
In diesem Unterkapitel verschieden Varianten der Adressierung, 1-Adress-Maschine, ....



## (1.3.6.) Vom Assemblerprogramm zum Maschinen-Code

Systemn. Inf.  
Prof. Martini  
SS 2023

- Programme in Assemblersprache sind
  - zwar **maschinennah**,
  - aber **nicht unmittelbar lauffähig** (da Assemblersprachen zu „luxuriös“ sind).
- Die Übersetzung in ein Objektprogramm (d.h. in Maschinsprache) erfolgt durch ein spezielles Programm: „**Assembler**“ (**Assembler**).



- Neben der Verarbeitung von „Pseudo-Operationen“ (z.B. Auflösung externer Referenzen) übernimmt der Assembler i.w. die folgenden Aufgaben:
  - Umsetzung** der mnemotechnischen Darstellung **in Binärcode**,
  - Umsetzung** symbolischer Adressen („Label“) **in Speicheradressen**,
  - Erzeugung von Daten** (Umwandlung von Literalen in Binärdarstellung der Daten).

# Ein Durchgang ist nicht genug

Systemn. Inf.  
Prof. Martini  
SS 2023

Der typischer Assembler benötigt mindestens zwei Durchgänge:

## 1. Durchgang:

- Bestimmung der **Länge** von **Maschineninstruktionen**
- **Verwaltung** eines **Adresszählers** (Befehle und Daten)
- **Zuordnung symbolischer Adressen**
- **Zuordnung** von **Literalen**
- Verarbeitung einiger Assembler-Instruktionen

## 2. Durchgang:

- **Heranziehung** der **Symbolwerte** (= Speicheradressen)
- **Erzeugung** von **Maschineninstruktionen**
- **Erzeugung** von **Daten** (= Konstanten)
- Verarbeitung der restlichen Assembler-Instruktionen

## Benötigte Tabellen:

- Tabelle der **Maschineninstruktionen** (statisch)
- Tabelle der **Assemblerinstruktionen** (statisch)
- **Symboltabelle** (dynamisch)
- ggf. **weitere Tabellen** (z.B. Basisregistertabelle)

# Makros

Assembler-Programmierung macht häufig die Wiederholung von Code-Blöcken erforderlich. Hier können „Makros“ zum Einsatz kommen:

Ein **Makro** fasst **mehrere Befehle oder Deklarationen** zu einer Einheit zusammen.

Einem Makro wird bei seiner Definition ein eindeutiger **Bezeichner zugeordnet**.

Wo immer dieser Bezeichner im Programmtext auftaucht, wird er vom Assembler bei der „**Makroexpansion**“ durch den zugehörigen Text ersetzt, bevor die Umwandlung in Maschinencode vorgenommen wird.

Es ist auch möglich, **ähnliche Makrobefehle zu parametrisieren**: Die Parameter werden dann jeweils durch die aktuellen Parameter ersetzt.

## Beispiel: Definition eines Makros

```
MACRO;  
Potenzieren &Bas, &Exp  
 $\alpha_3 := \rho ( \&Bas ) ;$   
 $\alpha_2 := \rho ( \&Exp ) ;$   
call UP ;  
MEND ;
```

Pseudo-Operation  
Übergabe symbolischer Adressen

} Setzen der Akkumulatoren und  
Aufruf des Unterprogramms

Pseudo-Operation



# 1.3. Allgemeines zu 80x86 Assembler-Programmierung

Online-Literaturhinweise:

- Informationen zu Assembler unter Linux:
  - <http://asm.sourceforge.net/>
- Sehr ausführliches Tutorial/PDF-Buch zu PC-Assembler: (in **Intel-Syntax**, für **NASM**)  
Paul Carter, incl. Historie von x86 Programmierung
  - <http://pacman128.github.io/pcasm/>
- Linux Assembly HOWTO
  - <http://www.tldp.org/HOWTO/Assembly-HOWTO/>
- Assembler-Crash-Kurs der Uni Magdeburg
  - <http://www-ivs.cs.uni-magdeburg.de/bs/lehre/sose99/bs1/seminare/assembler.shtml>  
(2014: Link nicht mehr aktiv; siehe aber <https://archive.org/web/>)  
(<https://web.archive.org/web/20130517095415/http://www-ivs.cs.uni-magdeburg.de/bs/lehre/sose99/bs1/seminare/assembler.shtml>)



# Literaturhinweise – Online-Bücher

---

- Webster Online-Buch
  - Randall Hyde „The Art of Assembly Language Programming (AoA)
  - <http://plantation-productions.com/Webster/www.artofasm.com/Linux/>
  
- Online-Buch „Programming from the Ground Up“, Jonathan Bartlett 2004 GNU Free Documentation License
  - <http://savannah.nongnu.org/projects/pgubook/>

# Literaturhinweise – NASM Documentation

## NASM Documentation

<https://nasm.us/doc/nasmdoc0.html>



### NASM - The Netwide Assembler

*version 2.15.05*

This manual documents NASM, the Netwide Assembler: an assembler targetting the Intel x86 series of processors, with portable source.

#### Chapter 1: Introduction

##### Section 1.1: What Is NASM?

Section 1.1.1: License

#### Chapter 2: Running NASM

##### Section 2.1: NASM Command-Line Syntax

Section 2.1.1: The `-o` Option: Specifying the Output File Name

Section 2.1.2: The `-f` Option: Specifying the Output File Format

Section 2.1.3: The `-l` Option: Generating a Listing File

Section 2.1.4: The `-L` Option: Additional or Modified Listing Info

Section 2.1.5: The `-M` Option: Generate Makefile Dependencies

Section 2.1.6: The `-MG` Option: Generate Makefile Dependencies

Section 2.1.7: The `-MF` Option: Set Makefile Dependency File

Section 2.1.8: The `-MD` Option: Assemble and Generate Dependencies

Section 2.1.9: The `-MT` Option: Dependency Target Name

Section 2.1.10: The `-MQ` Option: Dependency Target Name (Quoted)

Section 2.1.11: The `-MP` Option: Emit phony targets

Section 2.1.12: The `-MW` Option: Watcom Make quoting style

Section 2.1.13: The `-F` Option: Selecting a Debug Information Format

Section 2.1.14: The `-g` Option: Enabling Debug Information.



# X86-64-Architektur

- Wir arbeiten hier mit dem **(64-Bit-)x86-Instruktionssatz** im sog. Protected Mode
- x86-Prozessoren adressieren den Speicher byteweise
- Instruktionen können auf einem oder mehreren Bytes im Arbeitsspeicher arbeiten
- Je nach Anzahl der Bytes verwendet man bei x86-Prozessoren die folgenden Begriffe:
  - word = 2 Byte
  - double word = dword = 4 Byte
  - quad word = 8 Byte
  - paragraph = 16 Byte
- Die **Bytereihenfolge** der x86-Architektur ist Little Endian (d.h. anders als Network Byte Order!)
  - mehr dazu folgt in Kapitel 3 Netzwerkprogrammierung
- Inzwischen sind **64-Bit-Erweiterungen** (ursprünglich von AMD entworfen) weit verbreitet
- wir gehen auch nochmal kurz auf die 32-Bit-Instruktionen ein
- bei den **Übungsaufgaben** Hinweise zu 32-bit vs. 64-bit



# Intel Assembler 80x86 CodeTable - Befehlsübersicht

Intel Assembler CodeTable 80x86 - Übersicht der Befehle - Netscape

File Edit View Go Bookmarks Tools Window Help

<http://www.jegerlehner.ch/intel/index.html>

Search

Intel Assembler CodeTable 80x86 - ...

Home About me Assembler Tips Gallery

## Intel Assembler 80x86 CodeTable Übersicht der Befehle


English  
Deutsch  
Español

Als ich meine ersten Schritte mit der Programmiersprache *Assembler Intel x86* machte, suchte ich eine kompakte Übersicht aller Befehle. Da ich keine finden konnte, stellte ich meine eigene zusammen:

- ▶ enthält die **meisten** Instruktionen (Transfer, Arithmetik, Logik, Sprünge, ...)
- ▶ enthält ein Diagramm der Register (EAX, EDX, ECX, EBX) und Flags
- ▶ enthält Quellcode eines Beispiel-Programms
- ▶ ist handlich und passt auf ein einziges Blatt (zweiseitig A4)
- ▶ ist kostenlos, aber ich erwarte dein **Feedback**

### Intel Assembler 80186 and higher CodeTable

Name	Comment	Code	Operation
MOV	Move (copy)	MOV Dest,Source	Dest:=Source
XCHG	Exchange	XCHG Op1,Op2	Op1:=Op2, Op2:=Op1
STC	Set Carry	STC	CF:=1
CLC	Clear Carry	CLC	CF:=0
CMC	Complement Carry	CMC	CF:= ~CF
STD	Set Direction	STD	DF:=1 (string op's)
CLD	Clear Direction	CLD	DF:=0 (string op's)
STI	Set Interrupt	STI	IF:=1
CLI	Clear Interrupt	CLI	IF:=0

 **Als PDF Datei herunterladen** (148 kB) und beidseitig auf ein Blatt drucken.

Benötigt [Acrobat Reader](#). Alternative Vorschau: [Seite1.gif](#), [Seite2.gif](#)  
Version 2.3, Sept 2003 [History/Changes](#). Über 1200 Downloads pro Monat (von verschiedenen IPs)!

© 2023/2024

BONN

# Intel Assembler 80x86 CodeTable - Befehlsübersicht

Intel Assembler 80186 and higher

CodeTable 1/2

© 1996-2003 by Roger Jegerlehner, Switzerland  
V 2.3 English. Also available in Spanish

TRANSFER		Code	Operation	Flags									
Name	Comment			O	D	I	T	S	Z	A	P	C	
MOV	Move (copy)	MOV Dest,Source	Dest:=Source										
XCHG	Exchange	XCHG Op1,Op2	Op1:=Op2, Op2:=Op1										
STC	Set Carry	STC	CF=1									1	
CLC	Clear Carry	CLC	CF=0									0	
CMC	Complement Carry	CMC	CF:=~CF										
STD	Set Direction	STD	DF=1 (string op's downwards)		1								
CLD	Clear Direction	CLD	DF=0 (string op's upwards)		0								
STI	Set Interrupt	STI	IF=1			1							
CLI	Clear Interrupt	CLI	IF=0			0							
PUSH	Push onto stack	PUSH Source	DEC SP, [SP]:=Source										
PUSHF	Push flags	PUSHF	O, D, I, T, S, Z, A, P, C 286+; also NT, IOPL										
PUSHA	Push all general registers	PUSHA	AX, CX, DX, BX, SP, BP, SI, DI										
POP	Pop from stack	POP Dest	Dest:=[SP], INC SP										
POPF	Pop flags	POPF	O, D, I, T, S, Z, A, P, C 286+; also NT, IOPL										
POPA	Pop all general registers	POPA	DI, SI, BP, SP, BX, DX, CX, AX										
CBW	Convert byte to word	CBW	AX:=AL (signed)										
CWD	Convert word to double	CWD	DX:AX:=AX (signed)										
CWDE	Conv word extended double	CWDE	EAX:=AX (signed)										
IN	Input	IN Dest, Port	AL/AX/EAX := byte/word/double of specified port										
OUT	Output	OUT Port, Source	Byte/word/double of specified port := AL/AX/EAX										

/ for more information see instruction specifications Flags: ±=affected by this instruction ?=undefined after this instruction

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										
DIV	Divide (unsigned)	DIV Op	Op:=byte: AL:=AX / Op Op:=word: AX:=DX:AX / Op Op:=double: EAX:=EDX:EAX / Op										
IDIV	Signed Integer Divide	IDIV Op	Op:=byte: AL:=AX / Op Op:=word: AX:=DX:AX / Op Op:=double: EAX:=EDX:EAX / Op										
MUL	Multiply (unsigned)	MUL Op	Op:=byte: AX:=AL*Op Op:=word: DX:AX:=AX*Op Op:=double: EDX:EAX:=EAX*Op										
IMUL	Signed Integer Multiply	IMUL Op	Op:=byte: AX:=AL*Op Op:=word: DX:AX:=AX*Op Op:=double: EDX:EAX:=EAX*Op										
INC	Increment	INC Op	Op:=Op+1 (Carry not affected!)										
DEC	Decrement	DEC Op	Op:=Op-1 (Carry not affected!)										
CMP	Compare	CMP Op1,Op2	Op1-Op2										
SAL	Shift arithmetic left (= SHL)	SAL Op,Quantity											
SAR	Shift arithmetic right	SAR Op,Quantity											
RCL	Rotate left through Carry	RCL Op,Quantity											
RCR	Rotate right through Carry	RCR Op,Quantity											
ROL	Rotate left	ROL Op,Quantity											
ROR	Rotate right	ROR Op,Quantity											

/ for more information see instruction specifications ± then CF=0, OF=0 else CF=1, OF=1

LOGIC		Code	Operation	Flags									
Name	Comment			O	D	I	T	S	Z	A	P	C	
NEG	Negate (two-complement)	NEG Op	Op:=0-Op (if Op=0 then CF=0 else CF=1)										
NOT	Invert each bit	NOT Op	Op:=~Op (invert each bit)										
AND	Logical and	AND Dest,Source	Dest:=Dest&Source										
OR	Logical or	OR Dest,Source	Dest:=Dest Source										
XOR	Logical exclusive or	XOR Dest,Source	Dest:=Dest^Source										
SHL	Shift logical left (= SAL)	SHL Op,Quantity											
SHR	Shift logical right	SHR Op,Quantity											

Intel Assembler 80186 and higher

CodeTable 2/2

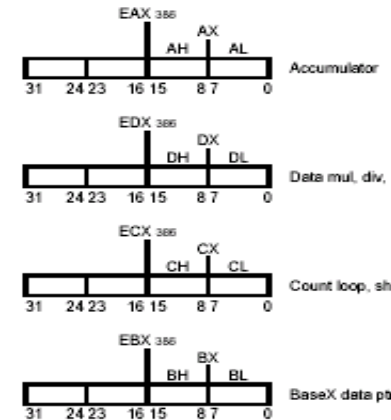
© 1996-2003 by Roger Jegerlehner, Switzerland  
V 2.3 English. Also available in Spanish

MISC		Code	Operation	Flags									
Name	Comment			O	D	I	T	S	Z	A	P	C	
NOP	No operation	NOP	No operation										
LEA	Load effective address	LEA Dest,Source	Dest := address of Source										
INT	Interrupt	INT Nr	Interrupts current program, runs spec. int-program			0	0						

JUMPS (flags remain unchanged)		Code	Operation	Name		Code	Operation
Name	Comment						
CALL	Call subroutine	CALL Proc		RET	Return from subroutine	RET	
JMP	Jump	JMP Dest					
JE	Jump if Equal	JE Dest (= JZ)		JNE	Jump if not Equal	JNE Dest (= JNZ)	
JZ	Jump if Zero	JZ Dest (= JE)		JNZ	Jump if not Zero	JNZ Dest (= JNE)	
JCXZ	Jump if CX Zero	JCXZ Dest		JECXZ	Jump if ECX Zero	JECXZ Dest	
JP	Jump if Parity (Parity Even)	JP Dest (= JPE)		JNP	Jump if no Parity (Parity Odd)	JNP Dest (= JPO)	
JPE	Jump if Parity Even	JPE Dest (= JP)		JPO	Jump if Parity Odd	JPO Dest (= JNP)	

JUMPS Unsigned (Cardinal)		Code	Operation	Name		Code	Operation
Name	Comment						
JA	Jump if Above	JA Dest (= JNBE)		JG	Jump if Greater	JG Dest (= JNLE)	
JAE	Jump if Above or Equal	JAE Dest (= JNB = JNC)		JGE	Jump if Greater or Equal	JGE Dest (= JNL)	
JB	Jump if Below	JB Dest (= JNAE = JC)		JL	Jump if Less	JL Dest (= JNGE)	
JBE	Jump if Below or Equal	JBE Dest (= JNA)		JLE	Jump if Less or Equal	JLE Dest (= JNG)	
JNA	Jump if not Above	JNA Dest (= JBE)		JNG	Jump if not Greater	JNG Dest (= JLE)	
JNAE	Jump if not Above or Equal	JNAE Dest (= JB = JC)		JNGE	Jump if not Greater or Equal	JNGE Dest (= JL)	
JNB	Jump if not Below	JNB Dest (= JAE = JNC)		JNL	Jump if not Less	JNL Dest (= JGE)	
JNBE	Jump if not Below or Equal	JNBE Dest (= JA)		JNLE	Jump if not Less or Equal	JNLE Dest (= JG)	
JC	Jump if Carry	JC Dest		JO	Jump if Overflow	JO Dest	
JNC	Jump if no Carry	JNC Dest		JNO	Jump if no Overflow	JNO Dest	
				JS	Jump if Sign (= negative)	JS Dest	
				JNS	Jump if no Sign (= positive)	JNS Dest	

General Registers:



Flags: - - - - - O D I T S Z A P C

Control Flags (how instructions are carried out):  
D: Direction 1 = string op's process down from high to low address  
I: Interrupt whether interrupts can occur. 1= enabled  
T: Trap single step for debugging

Example:

```

DOSSEG ; Demo program
.MODEL SMALL
.STACK 1024

Two EQU 2 ; Const
.DATA
VarB DB ? ; define Byte, any value
VarW DW 1010b ; define Word, binary
VarW2 DW 257 ; define Word, decimal
VarD DD 0AFFFFh ; define Doubleword, hex
S DB "Hello!", 0 ; define String
.CODE
main: MOV AX,DGROUP ; resolved by linker
MOV DS,AX ; init datasegment reg
MOV [VarB],42 ; init VarB
MOV [VarD],-7 ; set VarD
MOV BX,Offset[S] ; addr of "H" of "Hello!"
MOV AX,[VarW] ; get value into accumulator
ADD AX,[VarW2] ; add VarW2 to AX
MOV [VarW2],AX ; store AX in VarW2
MOV AX,4C00h ; back to system
INT 21h
END main

```

Status Flags (result of operations):  
C: Carry result of unsigned op. is too large or below zero. 1 = carry/borrow  
O: Overflow result of signed op. is too large or small. 1 = overflow/underflow  
S: Sign sign of result. Reasonable for Integer only. 1 = neg. / 0 = pos.  
Z: Zero result of operation is zero. 1 = zero  
A: Aux. carry similar to Carry but restricted to the low nibble only  
P: Parity 1 = result has even number of set bitsDownload latest version free of charge from [www.jegerlehner.ch/intel/](http://www.jegerlehner.ch/intel/) This page may be freely distributed without cost provided it is not changed. All rights reserved

# Register – ab Intel Prozessor Intel 64 und aufwärts (64 bit)

Erweiterung auf  
32-bit-Register  
ab 80386

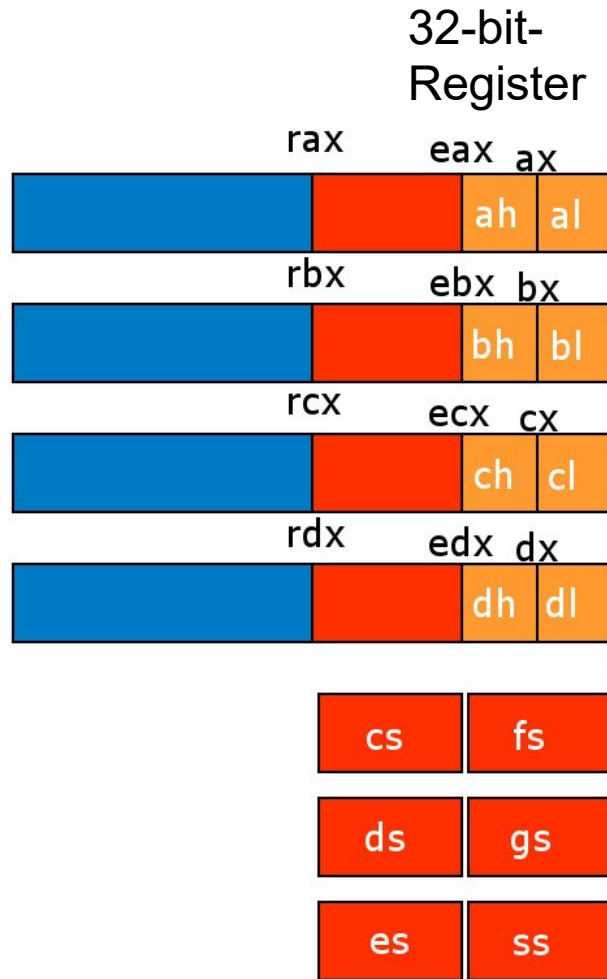
**Accumulator**  
(vgl. Sys.Inf.)

Allg. verw.  
Register

Counter für  
Schleifen

Datenregister  
mul, div, IO

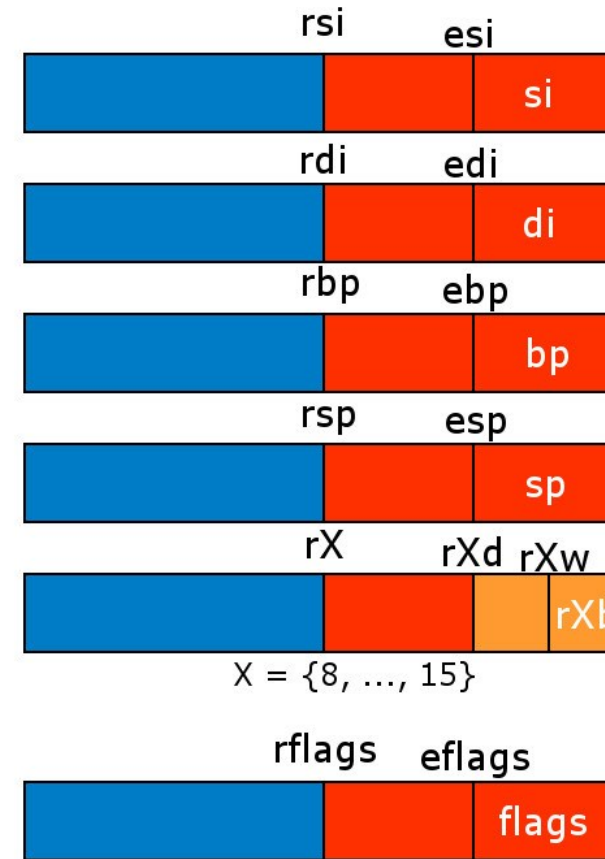
Codesegm.  
Datasegm.  
belieb. Segm.



**Segmentregister** (16 Bit)

belieb. Segm.  
belieb. Segm.  
Stacksegm.

32-bit-Register



Source für  
Stringoperationen

Destination für  
Stringoperationen

Basepointer

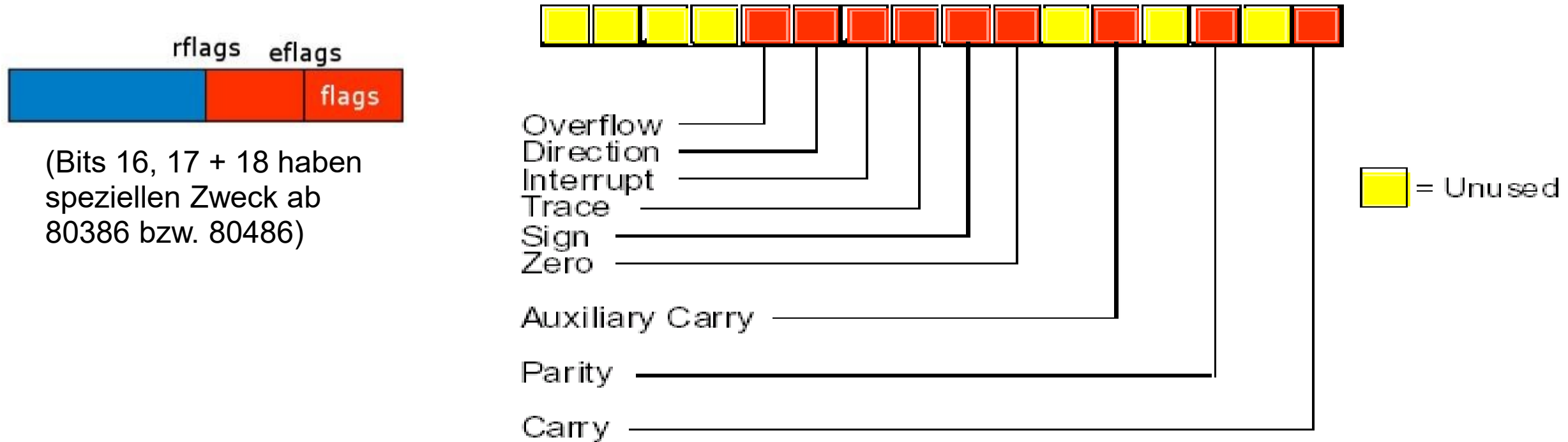
Stackpointer

Allg. verw.  
Register

**Flags bzw.  
Prozessor-  
Statusregister**  
(siehe Folie 28)



# Flags im Prozessor-Statusregister (64 bit)



(Bits 16, 17 + 18 haben speziellen Zweck ab 80386 bzw. 80486)

## Control Flags:

(Einfluss auf Ausführung von Instruktionen)

D: Direction, String Operation

1 = von hohen nach niedrigen Adressen

I: Interrupt, 1 = Interrupt enabled

T: Trap/Trace, Debugging

Single Step/Trace Modus

## Status Flags:

(Ergebnis der Ausführung von Instruktionen)

C: Carry, Unsigned Op. zu groß oder <0

O: Overflow, Signed Op. zu groß oder zu klein

S: Sign, Vorzeichen 1 = neg. / 0 = pos.

Z: Zero, 1 = Ergebnis ist null

A: Aux. Carry, für BCD-Operationen

P: Parity, 1 = Ergebnis hat gerade Anzahl

an Einsen



# Welcher Befehl beeinflusst welche Flags?

Siehe rechte Spalten der CodeTable/Befehlsübersicht:

Beispiele:

STC, Set Carry, CF := 1

ADC, Add with Carry,  
Dest:=Dest+Source+CF

liest C-Flag  
beeinflusst Flags O, S, Z, A, P, C







beeinflusst C-Flag (C := 1)

Intel Assembler 80186 and higher

CodeTable 1/2

© 1996-2003 by Roger Jegerlehner, Switzerland  
V 2.3 English. Also available in Spanish

TRANSFER				Flags								
Name	Comment	Code	Operation	O	D	I	T	S	Z	A	P	C
MOV	Move (copy)	MOV Dest,Source	Dest:=Source									
XCHG	Exchange	XCHG Op1,Op2	Op1=Op2 , Op2=Op1									
STC	Set Carry	STC	CF:=1									1
CLC	Clear Carry	CLC	CF:=0									0
CMC	Complement Carry	CMC	CF:= ~CF									±
STD	Set Direction	STD	DF:=1 (string op's downwards)		1							
CLD	Clear Direction	CLD	DF:=0 (string op's upwards)		0							
STI	Set Interrupt	STI	IF:=1			1						
CLI	Clear Interrupt	CLI	IF:=0			0						
PUSH	Push onto stack	PUSH Source	DEC SP, [SP]=Source									
PUSHF	Push flags	PUSHF	O, D, I, T, S, Z, A, P, C 286+: also NT, IOPL									
PUSHA	Push all general registers	PUSHA	AX, CX, DX, BX, SP, BP, SI, DI									
POP	Pop from stack	POP Dest	Dest:=[SP], INC SP									
POPF	Pop flags	POPF	O, D, I, T, S, Z, A, P, C 286+: also NT, IOPL	±	±	±	±	±	±	±	±	±
POPA	Pop all general registers	POPA	DI, SI, BP, SP, BX, DX, CX, AX									
CBW	Convert byte to word	CBW	AX:=AL (signed)									
CWD	Convert word to double	CWD	DX:AX:=AX (signed)	±					±	±	±	±
CWDE	Conv word extended double	CWDE 386	EAX:=AX (signed)									
IN	Input	IN Dest, Port	AL/AX/EAX := byte/word/double of specified port									
OUT	Output	OUT Port, Source	Byte/word/double of specified port := AL/AX/EAX									

/ for more information see instruction specifications				Flags: ±=affected by this instruction ?=undefined after this instruction								
ARITHMETIC			Flags									
Name	Comment	Code	Operation	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)	±					±	±	±	±
DIV	Divide (unsigned)	DIV Op	Op=byte: AL:=AX / Op AH:=dest ? ? ? ? ? ?						?	?	?	?
DIV	Divide (unsigned)	DIV Op	Op=word: AX:=DX:AX / Op DX:=dest ? ? ? ? ? ?						?	?	?	?
DIV 386	Divide (unsigned)	DIV Op	Op=doublew: EAX:=EDX:EAX / Op EDX:=dest ? ? ? ? ? ?						?	?	?	?
IDIV	Signed Integer Divide	IDIV Op	Op=byte: AL:=AX / Op AH:=dest ? ? ? ? ? ?						?	?	?	?
IDIV	Signed Integer Divide	IDIV Op	Op=word: AX:=DX:AX / Op DX:=dest ? ? ? ? ? ?						?	?	?	?
IDIV 386	Signed Integer Divide	IDIV Op	Op=doublew: EAX:=EDX:EAX / Op EDX:=dest ? ? ? ? ? ?						?	?	?	?
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	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 ± ± ± ± ± ±						?	?	?	±
INC	Increment	INC Op	Op:=Op+1 (Carry not affected !)						±	±	±	±
DEC	Decrement	DEC Op	Op:=Op-1 (Carry not affected !)						±	±	±	±
CMP	Compare	CMP Op1,Op2	Op1-Op2						±	±	±	±
SAL	Shift arithmetic left (= SHL)	SAL Op,Quantity		/					±	±	?	±
SAR	Shift arithmetic right	SAR Op,Quantity		/					±	±	?	±
RCL	Rotate left through Carry	RCL Op,Quantity		/								±
RCR	Rotate right through Carry	RCR Op,Quantity		/								±
ROL	Rotate left	ROL Op,Quantity		/								±
ROR	Rotate right	ROR Op,Quantity		/								±

/ for more information see instruction specifications				• then CF:=0, OF:=0 else CF:=1, OF:=1															
LOGIC			Code	Operation	Flags														
Name	Comment	O			D	I	T	S	Z	A	P	C							
NEG	Negate (two-complement)	NEG Op	Op:=0-Op	if Op=0 then CF:=0 else CF:=1	1	±						±	±	±	±				
NOT	Invert each bit	NOT Op	Op:=~Op (invert each bit)									±	±	±	±				
AND	Logical and	AND Dest,Source	Dest:=Dest&Source		0							±	±	?	±				
OR	Logical or	OR Dest,Source	Dest:=Dest Source		0							±	±	?	±				
XOR	Logical exclusive or	XOR Dest,Source	Dest:=Dest (xor) Source		0							±	±	?	±				
SHL	Shift logical left (= SAL)	SHL Op,Quantity			/							±	±	?	±				
SHR	Shift logical right	SHR Op,Quantity			/							±	±	?	±				

# Erster „Gehversuch“ in Assembler

Der Klassiker: „Hello World!“	Erläuterung
<pre> ; syscall „write“ %macro write 3     mov rax, 1     mov rdi, %1     mov rsi, %2     mov rdx, %3     syscall %endmacro  SECTION .data hello: db „Hello World!\n“ helloLen: equ \$ - hello  global _start  SECTION .text _start:     write 1, hello, helloLen      mov rax, 60     mov rdi, 0     syscall </pre>	<p>Makro: An der aufrufenden Stelle wird Textersetzung durch die Instruktionen des Makros vorgenommen</p> <p>hier: Aufruf der Systemfunktion write</p> <p>Datenteil des Assembler-Programms, z.B. String-Konstanten, globale Variablen, ... Eigene Adresse – Adresse Label „hello“</p> <p>Für den Linker sichtbares Label.</p> <p>Textteil des Assembler-Programms, das eigentliche Programm selbst. _start ist Label für Start des Hauptprogramms.</p>

# Hello World Bsp. – Compilieren + Aufruf unter 64-bit Linux

Aufruf	Erläuterung
<code>nasm -f elf64 -g -F dwarf -o helloworld.o helloworld.S</code>	Programmtext in Datei <helloworld.S> speichern Aufruf des Assemblers mit Erzeugung von Debug-Infos (Debug format „dwarf“ für elf64)
<code>ld -o helloworld helloworld.o</code>	Aufruf des Linkers für aufrufbare Datei
<code>clang -o helloworld helloworld.o</code>	clang Linker
<code>./helloworld</code>	Aufruf des Programms
<code>strace ./helloworld &gt; /dev/null</code>	Aufruf des Programms mit Trace der aufgerufenen Systemfunktionen (Unterdrückung von STDOUT)
<code>gdb ./helloworld</code>	Aufruf des Programms im Debugger, dort Setzen von Breakpoints, Step-Modus, Register-Inhalt, ...

# Debuggen von Programmen, insbes. Assembler

- gdb, einfachster (GNU) Debugger in Commandline
- gdb -tui, oder gdbtui, Debugger mit strukturiertem UI im Terminal

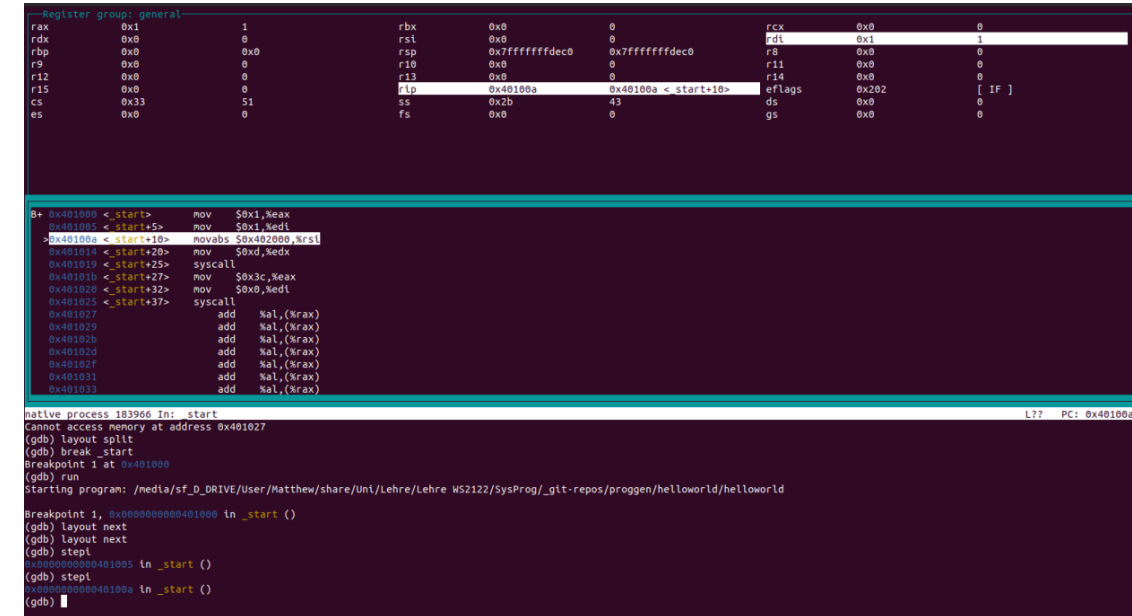
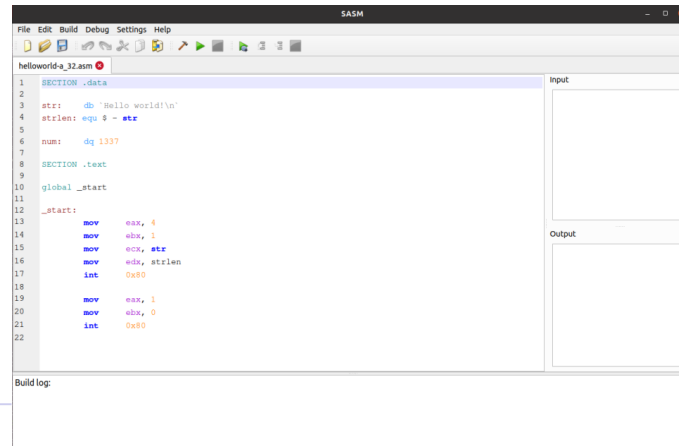
(Einführung z.B.

[https://www.youtube.com/watch?v=mm0b\\_H0KIRw](https://www.youtube.com/watch?v=mm0b_H0KIRw) )

- SASM, crossplatform IDE for NASM, ...

<https://dman95.github.io/SASM/english.html>

- IDE mit Qt GUI, auch als Ubuntu Paket verfügbar



Mehr dazu in den Übungen ...



# Gruppen/Klassen von Operationen

## 1. Data movement instructions – **Verschieben von Daten in und von Registern**

mov, lea, les, push, pop, pushf, popf

## 2. Conversions – **Konvertierungen Byte, Word, Double, Extended Double**

cbw, cwd, cwde

## 3. Arithmetic instructions – **Arithmetik**

add, inc, sub, dec, cmp, neg, mul, imul, div, idiv

## 4. Logical, shift, rotate and bit instructions – **Logik, Bit-Operationen**

and, or, xor, not, shl, shr, rcl, rcr

## 5. I/O instructions – **Input/Output**

in, out

## 6. String instructions – **String Verarbeitung**

movs, stos, lods

## 7. Program flow control instructions – **Programmfluss, Sprungbefehle**

jmp, call, ret, (conditional jumps)

## 8. Miscellaneous instructions – **Verschiedenes**

cld, stc, cmc

# Überblick über die wichtigsten Befehle (1)

Name	Comment	Syntax
MOV	Move (Copy)	MOV Dest, Source
XCHG	Exchange	XCHG Op1, Op2
STC	Set Carry	STC
CLC	Clear Carry	CLC
CMC	Complement Carry	CMC
STD	Set Direction	STD
CLD	Clear Direction	CLD
STI	Set Interrupt	STI
CLI	Clear Interrupt	CLI
PUSH	Push onto stack	PUSH Source
PUSHF	Push flags	PUSHF
PUSHA	Push all general registers	PUSHA
POP	Pop from stack	POP Dest
POPF	Pop flags	POPF
POPA	Pop all general registers	POPA
CBW	Convert byte to word	CBW
CWD	Convert word to double	CWD
CWDE	Convert word to extended double	CWDE
IN	Input	IN Dest, Port
OUT	Output	OUT Port, Source

Verschieben  
von Daten  
in und von  
Registern

+

Verschiedene

Konvertierung

Input/Output



# Überblick über die wichtigsten Befehle (2)

Name	Comment	Syntax
ADD	Add	ADD Dest, Source
ADC	Add with Carry	ADC Dest, Source
SUB	Subtract	SUB Dest, Source
SBB	Subtract with Borrow	SBB Dest, Source
DIV	Divide (unsigned)	DIV Op
IDIV	Signed Integer Divide	IDIV Op
MUL	Multiply (unsigned)	MUL Op
IMUL	Signed Integer Multiply	IMUL Op
INC	Increment	INC Op
DEC	Decrement	DEC Op
CMP	Compare	Comp Op1, Op2
SAL	Shift arithmetic left	SAL Op, Quantity
SAR	Shift arithmetic right	SAR Op, Quantity
RCL	Rotate left through Carry	RCL Op, Quantity
RCR	Rotate right through Carry	RCR Op, Quantity
ROL	Rotate left	ROL Op, Quantity
ROR	Rotate right	ROR Op, Quantity

Arithmetik

Bit-Operationen

# Überblick über die wichtigsten Befehle (3)

Name	Comment	Syntax
NEG	Negate (two-complement)	NEG Op
NOT	Invert each bit	NOT Op
AND	Logical and	AND Dest, Source
OR	Logical or	OR Dest, Source
XOR	Logical exclusive or	XOR Dest, Source
SHL	Shift logical left	SHL Op, Quantity
SHR	Shift logical right	SHR Op, Quantity
NOP	No operation	NOP
LEA	Load effective address	LEA Dest, Source
INT	Interrupt	INT Nr
CALL	Call subroutine	CALL Proc
RET	Return from subroutine	RET
JMP	Jump	JMP Dest
JE	Jump if Equal	JE Dest
JZ	Jump if Zero	JZ Dest
JCXZ	Jump if CX Zero	JCXZ Dest

Logik

Verschiedenes

Sprünge  
(allgemein)



# Überblick über die wichtigsten Befehle (4)

Name	Comment	Syntax
JP	Jump if Parity (Parity Even)	JP Dest
JPE	Jump if Parity Even	JPE Dest
JNE	Jump if not Equal	JNE Dest
JNZ	Jump if not Zero	JNZ Dest
JECXZ	Jump if ECX Zero	JECXZ Dest
JNP	Jump if not Parity (Parity Odd)	JNP Dest
JPO	Jump if Parity Odd	JPO Dest
JA	Jump if Above	JA Dest
JAE	Jump if Above or Equal	JAE Dest
JB	Jump if Below	JB Dest
JBE	Jump if Below or Equal	JBE Dest
JNA	Jump if not Above	JNA Dest
JNAE	Jump if not Above or Equal	JNAE Dest
JNB	Jump if not Below	JNB Dest
JNBE	Jump if not Below or Equal	JNBE Dest
JC	Jump if Carry	JC Dest
JNC	Jump if no Carry	JNC Dest

Sprünge  
(allgemein)

Sprünge  
(unsigned)

# Überblick über die wichtigsten Befehle (5)

Name	Comment	Syntax
JG	Jump if Greater	JG Dest
JGE	Jump if Greater or Equal	JGE Dest
JL	Jump if Less	JL Dest
JLE	Jump if Less or Equal	JLE Dest
JNG	Jump if not Greater	JNG Dest
JNGE	Jump if not Greater or Equal	JNGE Dest
JNL	Jump if not Less	JNL Dest
JNLE	Jump if not Less or Equal	JNLE Dest
JO	Jump if Overflow	JO Dest
JNO	Jump if not Overflow	JNO Dest
JS	Jump if Sign (= negative)	JS Dest
JNS	Jump if no Sign (= positive)	JNS Dest

Sprünge  
(signed)  
Integer

# Assembler-Syntax: Intel vs. AT&T

- Für die Assembler-Programmierung unter 80x86 gibt es (leider) zwei verschiedene Syntax-Formen, die sog. **AT&T Syntax**, sowie die **Intel-Syntax**.
- **Geschmackssache**, wem welche Syntax besser gefällt
- Früher haben wir den GNU-Assembler genutzt. Dieser war für die AT&T-Syntax ausgelegt. Nun sind wir auf NASM gewechselt. **Intel-Syntax!**

## CREDITS - DANKSAGUNG

- **Für den Assembler-Foliensatz** (für WS 2015/2016) hat unser (damaliger) Tutor Lennart Buhl dieses Kapitel an die (mittlerweile weiter verbreitete) **Intel-Syntax angepasst**.
- NASM unterstützt (, soweit uns bekannt,) nicht die AT&T-Syntax. Für die Übungsaufgaben sowie die (möglichen) Assembler-Aufgaben in der Klausur **akzeptieren wir (eigentlich nicht) beide Syntax-Formen\***.

\* ... und wesentlich wichtiger sind gute, erklärende Kommentare!

# Assembler-Syntax: Intel vs. AT&T

Für Details zum Unterschied zwischen Intel und AT&T Syntax siehe:

- <http://asm.sourceforge.net/articles/linasm.html>

## Präfixe vor Konstanten/Registern:

Intel Syntax	AT&T Syntax	Besondere Präfixe in AT&T Syntax
<code>mov rax, 1</code>	<code>movq \$1, %rax</code>	Konstanten / „Immediate“ \$
<code>mov rbx, 0xffff</code>	<code>movq \$0xffff, %rbx</code>	Register %
<code>int 0x80</code> alternativ: <code>80h</code>	<code>int \$0x80</code>	Hexadezimal 0x

## Reihenfolge der Operanden:

Intel Syntax	AT&T Syntax	Besondere Präfixe in AT&T Syntax
<code>cmd dest, src</code>	<code>cmd src, dest</code>	Hauptunterschied: Operandenreihenfolge
<code>mov rax, rcx</code> (quasi <code>rax:=rcx</code> )	<code>movq %rcx, %rax</code> (quasi <code>rcx-&gt;rax</code> )	Häufigster Befehl: <code>mov</code>



# Assembler-Syntax: Intel vs. AT&T

## Speicheroperanden + Indizierung:

Intel Syntax	AT&T Syntax	Besonderheit AT&T Syntax
<code>mov rax, [rbx]</code>	<code>movq (%rbx), %rax</code>	Register-indirekte Adressierung mit (..)
<code>mov rax, [rbx+3]</code>	<code>movq 3(%rbx), %rax</code>	Indizierung mit Basisregister innerhalb (...), Offset außerhalb

## Grundstruktur:

Intel Syntax	AT&T Syntax
intuitiv lesbar: <code>segreg:[base+index*scale+disp]</code>	(leider) etwas obskur: <code>%segreg:disp(base, index, scale)</code>

(segreg: Segment Register,  
base: Basis,  
disp: Displacement,  
Index,  
Scale)

Intel Syntax	AT&T Syntax
<code>cmd foo, [base+index*scale+disp]</code>	<code>cmd disp(base, index, scale), foo</code>
<code>mov rax, [rbx+20h]</code>	<code>movq 0x20(%rbx), %rax</code>
<code>lea rax, [rbx+rcx]</code>	<code>leaq (%rbx, %rcx), %rax</code>
<code>add rax, [rbx+rcx*2h]</code>	<code>addq (%rbx, %rcx, 0x2), %rax</code>
<code>sub rax, [rbx+rcx*4h-20h]</code>	<code>subq -0x20(%rbx, %rcx, 0x4), %rax</code>

# Assembler-Syntax: Intel vs. AT&T

## Suffixes für Größen der Operatoren hinter Befehl:

Intel Syntax	AT&T Syntax	Suffixes bei AT&T Syntax
<code>mov al, bl</code>	<code>movb %bl, %al</code>	b = Byte (8 Bit)
<code>mov ax, bx</code>	<code>movw %bx, %ax</code>	w = Word (16 Bit)
<code>mov eax, ebx</code>	<code>movl %ebx, %eax</code>	l = Long (32 Bit)
<code>mov rax, rbx</code>	<code>movq %rbx, %rax</code>	q = Quad (64 Bit)
<del><code>mov rax, qword ptr [rbx]</code></del>  (bei Speicheroperanden mit Intel in ursprünglicher Intel-Syntax: <code>byte ptr, word ptr, dword ptr, qword ptr</code> )	<code>movq (%rbx), %rax</code>	

Bei unserem NASM: ohne `ptr` („warning: is not a NASM keyword“ o.ä.)

`byte, word, dword, qword`

Bzw. überflüssig/redundant, wenn durch zweiten Operanden klar.  
 ggf. Fehlermeldung von NASM, z.B. `error: mismatch in operand sizes`

# Adressierungsarten

- Die meisten Befehle können ihre Operanden aus Registern, aus dem Speicher oder unmittelbar („immediate“) einer Konstante entnehmen.

- Registeradressierung:** `mov rbx, rdi`

Wert eines Registers in ein anderes Register (rbx)

- Unmittelbare Adressierung:** `mov rbx, 1000`

Konstante in Register rbx

- Direkte Adressierung:** `mov rbx, [1000]`

Wert der angegebenen Speicherstelle in Register rbx

- Register-indirekte Adressierung:** `mov rbx, [rax]`

Wert der in Register rax stehenden Speicherstelle in Register rbx

- Basis-Register Adressierung:** `mov rax, [rsi+10]`

Wert der Speicherstelle (Summe Konstante + Inhalt rsi) in Register rax

- Allgemeine Indizierung**

- `mov register, segreg:[base+index*scale+disp]`

- (segreg: Segment Register, base: Basis, disp: Displacement, Index, Scale)

# Adressierungsarten, Indizierung, Segmentregister

## Allgemeine Indizierung

`mov register, segreg:[base+index*scale+disp]`

disp (Displacement): beliebige Konstante

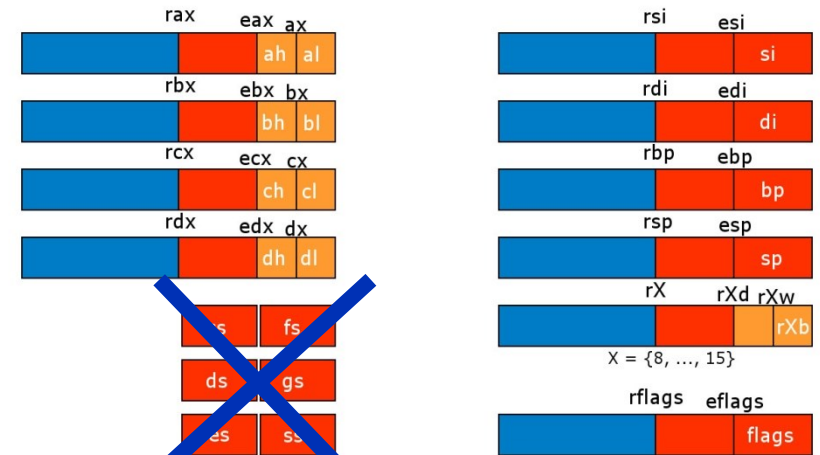
base, index: beliebiges General-Purpose Register

scale: Konstante 1, 2, 4 oder 8  
(z.B. Indizierung von Word, Long)

segreg: Segmentregister

Im sog. „**Real Mode**“ müssen die Segmentregister und ggf. Offset korrekt benutzt werden. Es gibt keinen (automatischen) Speicher-Segment-Schutz. (benutzt z.B. vom Betriebssystem/Systemfunktionen)

Im sog. „**Protected Mode**“ muss sich der Programmierer nicht um die Segmentregister kümmern. Diese werden automatisch gesetzt und es gibt einen (automatischen) Speicher-Segment-Schutz.



Im Protected Mode  
keine Segmentregister  
zu benutzen!

# Erste Assembler-Beispiele: Zuweisung + Summe

C-Anweisung für Summenbildung:

```
summe = a + b + c + d;
```

Zu kompliziert für Assembler, deshalb:

```
summe = a;  
summe = summe + b;  
summe = summe + c;  
summe = summe + d;
```

Übertragung in Assembler:

```
mov rax, [a]  
add rax, [b]  
add rax, [c]  
add rax, [d]
```

a, b, c, d sind Speicherstellen  
im Daten-Segment

Ergebnis im Register rax

# Erste Assembler-Beispiele: if – then – else

C-Anweisung für if-then-else:

```
if (a == 4711)
{
    ...
}
else
{
    ...
}
```

Zu kompliziert für Assembler, deshalb Label und Sprünge verwenden:

```
        if (a != 4711)
            goto ungleich

gleich:  ...
        goto weiter

ungleich: ...

weiter:  ...
```

Übertragung in Assembler:

```
        cmp rax, 4711
        jne ungleich

gleich:  ...
        jmp weiter

ungleich: ...

weiter:  ...
```

# Erste Assembler-Beispiele: Zählschleife for i = 0 ... 99

C-Anweisung für einfache Zählschleife:

```
for (i=0; i<100; i++)  
{  
    summe = summe + a;  
}
```

Dies ist nicht ganz so kompliziert  
in Assembler, denn es gibt das  
spezielle rcx Register als Loop-Count

Übertragung in Assembler:

```
                mov    rcx, 100  
schleife: add    rax, [a]  
                loop   schleife
```

a ist Speicherstelle  
im Daten-Segment

loop dekrementiert rcx und  
springt wenn rcx ungleich 0

# Weiteres Beispiel, indirekte Adressierung

## Allgemeine Indizierung

`mov register, segreg:[base+index*scale+disp]`

disp (Displacement):	beliebige Konstante
base, index:	beliebiges General-Purpose Register
scale:	Konstante 1, 2, 4 oder 8 (z.B. Indizierung von Word, Long)
segreg:	Segmentregister

Bilde Summe von 100 beliebigen Zahlen (Array):

<code>mov rcx, 100</code>	<code>; Initialisiere Index mit 100</code>
<code>mov rax, 0</code>	<code>; Initialisiere Summe mit 0</code>
<code>mov rbx, array</code>	<code>; Arrayadresse in Basis</code>
<code>schleife: add ax, word [rbx+rcx*2-2]</code>	<code>; Array-Durchlauf in 2er Schritten (rückwärts)</code>
<code>loop schleife</code>	<code>; Schleifenende</code>

array ist Speicherstelle  
im Daten-Segment, ab der  
100 Zahlen abgelegt sind  
(jeweils 2 Byte lang)

loop dekrementiert rcx und  
springt wenn rcx ungleich 0

**Achtung:** (`rcx * scale`) hat in der Schleife  
die Werte 200 ... 2  
=> Displacement „-2“ => Index 198 ... 0





# Beispiel – jetzt ohne “loop”-Befehl

array ist Speicherstelle  
im Daten-Segment, ab der  
100 Zahlen abgelegt sind  
(jeweils 2 Byte lang)

Jetzt:

- nicht „loop“-Befehl verwenden
- Index soll aufwärts von 0 bis 99 laufen

Bilde Summe von 100 beliebigen Zahlen (Array):

<code>mov rcx, 0</code>	<code>; Initialisiere Index mit 0</code>
<code>mov rax, 0</code>	<code>; Initialisiere Summe mit 0</code>
<code>mov rbx, array</code>	<code>; Arrayadresse in Basis</code>
<code>schleife: add ax, word [rbx+rcx*2]</code>	<code>; Array-Durchlauf in 2er Schritten (aufwärts)</code>
<code>inc rcx</code>	<code>; erhöhe Index</code>
<code>cmp 100, rcx</code>	<code>; Ende: Index == 99 + 1 ?</code>
<code>jne schleife</code>	<code>; wenn gleich, Schleifenende</code>

Hinweis:

Als **General Purpose Register für Basis und Index** bei indirekter Adressierung können die Register rax, rbx, rcx, rdx, rsi, rdi sowie r8-r15 verwendet werden.

