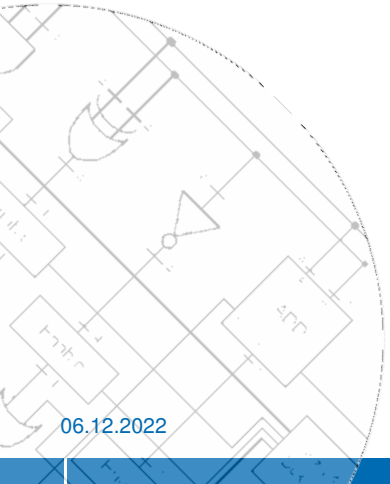
Rechnerarchitektur

Vorlesung 14:x86 Architektur, Programmaufbau und einfache Instruktionen

Prof. Dr. Martin Mauve



06.12.2022

Haben Sie noch Fragen zur letzten Vorlesung?

Thema: Caching. Details; Core i7, Einführung Assembler

Fahrplan

Speicher

- x86-Prozessoren adressieren den Arbeitsspeicher 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 = 4 byte
  - quad word = 8 byte
  - paragraph = 16 byte
- Die Bytereihenfolge der x86-Architektur ist Little Endian.

  (niederwertigstes Byte wird zuerst, also an der niedrigsten Adresse gespeichert)

Speichermodelle

- Real Mode, Flat Model: ab Intel 8080
  - 64K für alles (Programm und Daten)
  - Betriebssysteme: CP/M-80, später DOS
- Real Mode, Segmented Model: ab Intel 8086
  - für die Kompatibilität mit 8080: Einteilung des Speichers in 64k Blöcke
  - Man muss manuell festlegen, welchen Block man gerade verwendet
  - sehr aufwändig und fehleranfällig
  - Betriebssysteme: DOS
- Protected Mode: ab Intel 80386
  - 4GB virtueller Speicher für alles (Programm und Daten)
  - Betriebssysteme: Windows, Linux
- Für die Vorlesung:
  - Protected Mode für Linux
  - Real Mode, Flat Model für DOS

Prozessor

- Der Prozessor stellt den Instruktionssatz zur Verfügung.
- Dieser Instruktionssatz unterscheidet sich i.d.R. von Prozessor zu Prozessor.
- Instruktionssatz wird durch die Mikroarchitektur/-programme implementiert.
- Wir verwenden hier den Instruktionssatz der Intel-x86-Familie:
  - wird auch IA-32 (Intel Architecture 32) genannt
  - ab 80386, auch entsprechende AMD-Prozessoren
  - Erweiterungen MMX, SSE,. . .
  - 32 Bit ist (hier) die natürliche Größe für Operanden und Adressen
  - inzwischen für alle aktuellen CPUs: 64-Bit-Erweiterungen (verwenden wir hier nicht)

Register

- Innerhalb des Prozessors gibt es so genannte Register:
  - dies sind Speicherzellen
  - auf die besonders schnell zugegriffen werden kann
  - die teilweise besondere Bedeutung bei verschiedenen Instruktionen haben
- Viele Operationen verwenden Register als Operanden oder zum Speichern des Ergebnisses.

Register der IA-32–Architektur I

- Entwicklungsgeschichte (8086 von 1978!) kann aus dem Registersatz abgelesen werden
  - E = Extended = 32 Bit
- EAX, EBX, ECX, EDX
  - allgemeine 32-Bit-Register, aber:
  - EAX: Berechnungen
  - EBX: Zeiger (Pointer)
  - ECX: Schleifen
  - EDX: Mult/Div, 64 Bit mit EAX
  - 16- und 8-Bit-Teile (286, 8088)

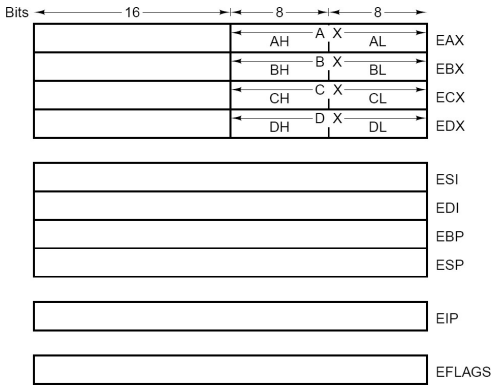


Bild: Structured Computer Organization, 5th Edition, Fig. 5-3

Register der IA-32–Architektur II

- ESI, EDI: String-Operationen
- EBP, ESP: Base/Stack-Pointer
- CS-GS: Speicher-Segment-Zugriff (nicht dargestellt)
- EIP: Instruction Counter (=PC)
- EFLAGS: Prozessorstatuswort

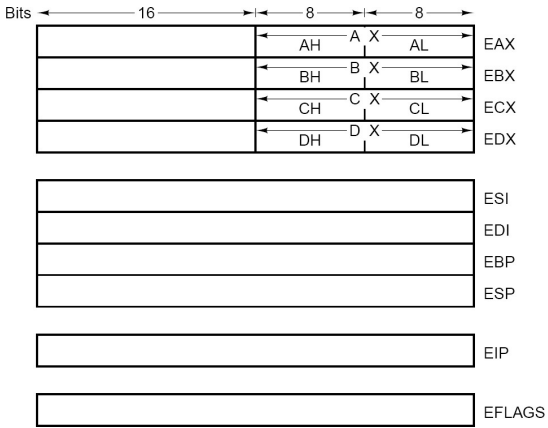


Bild: Structured Computer Organization, 5th Edition, Fig. 5-3

EFLAGS-Register

Das EFLAGS-Register hat eine besondere Bedeutung:

- Es gibt den Status des Prozessors an.
- Es erteilt Informationen über die letzte Berechnung.
- Jedes Bit in diesem Register hat eine eigene Bedeutung.
- Beispiele:
  - Sign Flag (S):
    Im Ergebnis der letzten Operation war das höchstwertige Bit gesetzt.
  - Zero Flag (Z):
    Das Ergebnis der letzten Operation war 0.
  - Carry Flag (C):
    Bei der letzten Operation gab es einen Übertrag.
  - Overflow Flag (O)
    Bei der letzten Operation gab es einen Überlauf.
- Die Flags des EFLAGS-Registers können mit speziellen Instruktionen abgefragt werden.
  - realisieren bedingter Sprünge

Fahrplan

Programmaufbau bei Assembler (NASM)

- Ein Programm kann aus mehreren Übersetzungseinheiten bestehen.
- Eine Übersetzungseinheit wird Modul genannt.
- Ein Modul kann die folgenden Elemente beinhalten:
  - Segmentidentifikation
  - Assembler-Instruktionen
  - Pseudo-Instruktionen
  - Präprozessor-Anweisungen
  - Direktiven an den Assembler (beinhaltet Segmentidentifikation)
- Dies ist assemblerspezifisch (also hier für NASM), von den Grundzügen her aber typisch für x86-Assembler.

Segmentidentifikation

- Eine Übersetzungseinheit kann aus mehreren (Speicher-)Bereichen bestehen.
- Beispiele:
  - `.text`: In diesem Bereich befinden sich die ausführbaren Instruktionen.
  - `.data`: In diesem Bereich befinden sich „Variablen“ mit vordefinierten Werten.
  - `.bss` (block start symbol): In diesem Bereich befinden sich „Variablen“, die keinen vordefinierten Wert besitzen.
- Diese Bereiche nennt man Segmente oder Sektionen.
- Der Beginn eines Segmentes wird mit der Assembler-Direktive `SEGMENT` (oder wahlweise `SECTION`) gekennzeichnet.
- Beispiel:

```
%include "asm_io.inc"
segment .text
        global  asm_main
asm_main:
        enter   0,0               ; Funktionseintritt
        ...
```

Assembler-Instruktion I

- Assembler-Instruktionen werden vom Assembler in Instruktionen des Befehlssatzes des Prozessors übersetzt.
- In NASM haben Assembler-Instruktionen folgendes Format:
  `<label>: <instruction> <operands> ; <comment>`
- Beispiel:
  `loop1: mov ebx, 1 ; Start of Loop, initialize ebx`
- label
  - Damit kann diese Zeile identifiziert werden, ohne dass man mühsam die Speicheradresse dieser Zeile bestimmen muss:
    - Das Bestimmen der Speicheradresse übernimmt der Assembler für uns.
  - Ein Label kann anstelle einer Adresse geschrieben werden, z.B.
    - um den Ort zu identifizieren, an dem Daten abgelegt sind
    - um Ziele von Verzweigungen oder Sprüngen zu identifizieren

Assembler-Instruktion II

- `instruction`
  - die eigentliche Assembler-Instruktion
- `operands`
  - Je nach Instruktion können Operanden folgen.
  - Operanden können
    - direkt angegeben werden (als Konstante)
    - in einem Register stehen
    - im Speicher stehen
- `comment`
  - alles nach dem `;` wird ignoriert
- Zeilenverlängerung: Eine Assembler-Instruktion muss in einer Zeile stehen.
  - Verlängerung einer Zeile durch `\` als abschließendes Zeichen möglich.
  - Dann wird der Zeilenumbruch ignoriert.

Spezifikation von Instruktionen

- Auszug aus dem NASM-Handbuch:

```
MOV mem,reg8                    [8086]
MOV reg16,reg16                 [8086]
MOV mem,reg32                    [386]
MOV reg8,imm                    [8086]

...
```

- Dies gibt an, welche Operanden erlaubt sind:
  - `regX` = Register der Größe X
  - `mem` = Speicher
  - `imm` = direkte Angabe des Operanden
  - ...

  wobei hier 1. Operand das Ziel und 2. Operand die Quelle ist
- Dann folgt der Prozessor, ab dem die Instruktion vorhanden ist.

Erlaubte Adressierung von Operanden

- Immediate: Operand als Konstante angeben
  - nur für Operanden, die kein Ziel für eine Operation sind
  - `mov eax, 0`
- Register: Register angeben
  - für Operanden, die Ziel oder Quelle für eine Operation sind
  - `mov eax, ebx`
- Speicher: Adresse der Speicherzelle in `[]` angeben
  - nur ein Operand darf direkt auf diese Art adressiert werden
  - `mov eax, [esp]`

Typische Addressierungsfehler

- `mov 17, 1`
  nur ein Operand darf Immediate adressiert werden
- `mov 17, bx`
  Ziel darf nicht Immediate adressiert werden
- `mov cx, dh`
  beide Operanden müssen die gleiche Größe haben
- `mov [ax],[bx]`
  nur ein Operand darf direkt im Speicher adressiert werden

Speicherzugriff

- Wenn ein Operand im Speicher steht, wird die Adresse des Operanden so angegeben: `[Adresse]`
- Beispiele ohne Speicherzugriff:
  - `mov eax, 1 ; lädt eine 1 in das Register eax`
  - Sei `loop1` ein label, dann
    `mov eax, loop1 ; lädt die Adresse (!) von loop1 in eax`
- Beispiele mit Speicherzugriff:
  - Sei `buffer` ein label, dann
    ```
    mov eax, [buffer] ; lädt 4 Bytes aus dem Speicher beginnend \
                        mit dem Byte an der Speicherstelle buffer
    ```
  - ```
    mov eax, [ecx+ebx*4+0x800] ; lädt 4 Bytes aus dem Speicher \
                                 beginnend mit dem Byte an der \
                                 Speicherstelle ecx+ebx*4+0x800
    ```
    - Dieses Konstrukt benötigt man z.B. um effizient über ein Array iterieren zu können.
    ```

Speicheradressierung

erlaubte Adressierung: `[BASE+(INDEX*SCALE)+DISP]`

- BASE: EAX, EBX, ECX, EDX + deren verkürzte Varianten
  - z.B.: Adresse des ersten Wertes in einem Array
- INDEX: EAX, EBX, ECX, EDX + deren verkürzte Varianten
  - z.B.: Index eines Array Elementes
- SCALE: 1, 2, 4 oder 8
  - z.B.: Größe eines Array Elementes
- DISP: 32-Bit Konstante
  - z.B. um einen konstanten Offset in eine Tabelle zu addieren
- Alle Elemente sind optional.

Datentypen

- In Assembler ist der Datentyp eines Operanden seine Länge in Bytes.
- Hat eine Instruktion mehr als einen Operanden, dann muss die Länge der Operanden meist identisch sein:
  - `mov eax, ebx` ist korrekt
  - `mov eax, bx` ist nicht korrekt
- Die Länge eines Operanden kann implizit bekannt sein:
  - z.B. bei `mov eax, 0x10` wird `0x10` als 32-Bit Zahl interpretiert
- Eine explizite Angabe der Länge eines Operanden ist erforderlich, wenn sie implizit nicht bestimmt werden kann:
  - z.B. bei `neg byte [wert]`
  - erlaubte Typangaben: `byte, word, dword`

Pseudoinstruktionen

- Pseudoinstruktionen werden nicht direkt in Instruktionen des Befehlssatzes übersetzt
- Sie haben aber das gleiche Format wie Assembler-Instruktionen
- Man verwendet Pseudoinstruktionen zum Beispiel für
  - das Bereitstellen von initialisiertem Speicherplatz (meist im `.data` Segment):
    - `db 0x55 ; Speicherplatz der Größe 1 Byte belegt mit 0x55`
    - `db 0x55, 0x56, 0x57 ; drei aufeinander folgende Bytes`
    - `db 'hello',13,10,'$' ; 8 aufeinander folgende Bytes`
    - `dw 0x1234 ; 0x34 0x12` (little endian)
    - `dw 'abc' ;0x61 0x62 0x63 0x00` (Auffüllen auf Wort Grenzen)
    - `dd 0x12345678 ; 0x78 0x56 0x34 0x12` (little endian)
  - das Bereitstellen von uninitialisiertem Speicherplatz (meist im `.bss` Segment):
    - `buffer: resb 64 ; 64 * 1 Byte reserviert`

Konstanten I

- In NASM gibt es 4 verschiedene Arten von Konstanten:
  - numeric, character, string, (floating-point)
- numeric (überall, wo die direkte Angabe eines Operanden erlaubt ist):
  - standard: dezimal (`mov ax, 100`)
  - hexadezimal durch
    - den Präfix `0x` (Beispiel: `mov ax, 0xa0`)
    - den Suffix `h` (Beispiel: `mov ax, 0a0h`)
      Achtung! die Zahl muss mit einer Ziffer beginnen, daher die '0'
    - den Präfix `$` (Beispiel: `mov ax, $0a0`)
      Achtung! die Zahl muss mit einer Ziffer beginnen, daher die '0'
  - oktal durch `q` oder `o` als Suffix
    - `mov ax, 777q`
    - `mov ax, 777o`
  - binär durch `b` als Suffix (`mov ax, 10010011b`)

Konstanten II

- character (überall, wo die direkte Angabe eines Operanden erlaubt ist):
  - bis zu 4 Zeichen in einfachen oder doppelten Anführungszeichen
  - Little Endian wird berücksichtigt:

```
mov eax, 'abcd' ; danach steht 0x64636261 in eax
```

- string (nur bei den Pseudoinstruktionen `DB/DW/DD` und `INCBIN`):
  - beliebig lange Kette von Zeichen in einfachen oder doppelten Anführungszeichen
  - es wird immer auf die „richtige“ Größe aufgefüllt
    - `DB`=ein Byte, `DW`=zwei Byte, `DD`=vier Byte
  - die Reihenfolge im Speicher ist so wie angegeben
    - `db 'hello' ;` ist äquivalent zu
    - `db 'h','e','l','l','o'`
    - `dd 'ninechars' ;` ist äquivalent zu
    - `dd 'ninechars',0,0,0`

Beispiel: Addieren von zwei Zahlen

```
%include "asm_io.inc"

segment .data
wert1:   dd  0x20
wert2:   dd  0x40
segment .bss
resultat:        resd 1

segment .text
        global  asm_main
asm_main:
        enter   0,0             ; Funktion initialisieren
        pusha

        mov eax, [wert1]        ; Ersten Operanden laden
        add eax, [wert2]        ; Zweiten Operanden hinzuaddieren
        mov [resultat], eax     ; Ergebnis speichern
        dump_regs 1             ; Ausgabe der Registerwerte

        popa                    ; zu C zurückkehren
        mov     eax, 0
        leave
        ret
```

Addieren von zwei Zahlen – Resultat

```
Register Dump # 1
EAX = 00000060 EBX = 40155B90 ECX = 00000001 EDX = 401570C0
ESI = 40014020 EDI = BFFFEBB4 EBP = BFFFEB58 ESP = BFFFEB38
EIP = 08048425 FLAGS = 0206
```

Der Präprozessor

- Analog zu C unterstützt auch NASM einen Präprozessor
- Dieser wird vor dem eigentlichen Übersetzen ausgeführt
- Anweisungen an den Präprozessor sind mit `%` gekennzeichnet
- Beispiel: `%include "asm_io.inc"`
  - fügt den Inhalt der Datei *asm_io.inc* an diese Stelle ein
- sehr mächtig:
  - genaueres im NASM-Handbuch (Kapitel 4)