

Einführung in (Intel) 80x86 Assembler

Die folgenden Folien geben einen Überblick über die weit verbreitet x86 Architektur die in PCs benutzt wird. Die Einführung ist aus der Sicht des Compilerbauers:

- Schwerpunkt liegt auf Instruktionssatz und Performance
- Wenig Beachtung der Hardwareebene und Peripherie

Kapitel 9: Einführung in (Intel) 80x86 Assembler

- 1 Geschichte
- 2 Programmiermodell
- 3 Befehlssatz
- 4 Konventionen
- 5 Beispiele
- 6 SSE
- 7 Literatur



Abbildung: Intel 8086

1978 **8086/8088 Prozessor**

16-Bit Prozessoren, 1-MByte Adressraum; Der 8088 besitzt im Gegensatz zum 8086 einen 16-Bit breiten Datenbus.

1982 **80286 Prozessor (286er)**

MMU Unit bringt Speicherschutz („Protected Mode“);
24 Bit Segmente \Rightarrow 16 MByte Arbeitsspeicher.

1985 **80386 Prozessor (386er)**

32-Bit Adressbus \Rightarrow 4 GBytes Arbeitsspeicher; Segmentiertes und Lineares Speichermodell; Virtuelle Speicher Verwaltung (Paging); Neuer 32-Bit Befehlssatz (IA-32); Virtual-8086 Mode

Geschichte

- 1989 **80486 Prozessor (486er)**
Einführung eines Level1-Caches für Instruktionen; Integrierte Fließkommaeinheit (x87); Detailverbesserungen
- 1993 **Pentium Prozessor, K5, K6**
Schnellerer Virtual-8086 Modes; Spätere Versionen enthalten MMX-Befehlssatz zur Beschleunigung von Multimedianewendungen (SIMD - single instruction multiple data)
- 1995 **P6 Prozessoren, Athlon**
Einführung des SSE Befehlserweiterungen mit 128 bit Registern)
- ab 2000 **Pentium 4, Pentium M, Core, Opteron, Phenom**
Einführung der SSE2, SSE3 Erweiterungen. 64Bit Modus („AMD64“, „EM64T“)

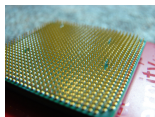


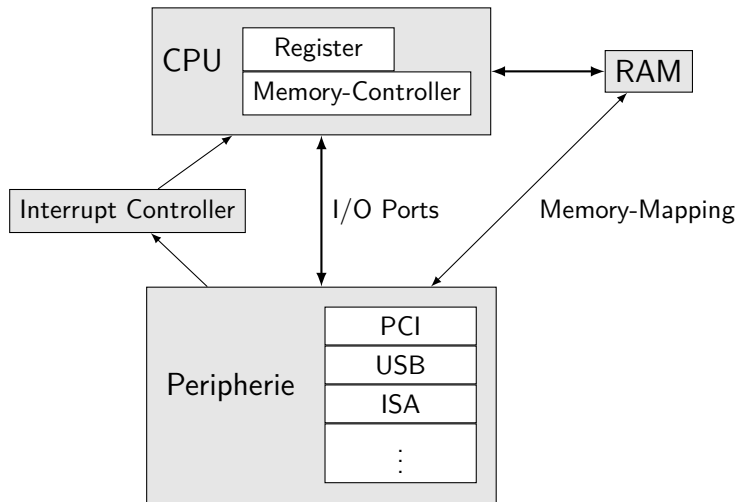
Abbildung: AMD Athlon 64

Kapitel 9: Einführung in (Intel) 80x86 Assembler

- 1 Geschichte
- 2 Programmiermodell**
- 3 Befehlssatz
- 4 Konventionen
- 5 Beispiele
- 6 SSE
- 7 Literatur

Verbindung mit der Aussenwelt

Programmiermodell:



Bestandteile:

- 8 General Purpose Register: A, B, C, D, SI, DI, BP, SP
 - in 32-Bit breiter Form: EAX, EBX, ECX, EDX, ESI, EDI, EBP, ESP
 - in 16 Bit Breiter Form: AX, BX, CX, DX, SI, DI, BP, SP
 - in 8 Bit breiter Form: AH/AL, BH/BL, CH/CL, DH/DL
- Flagsregister (Vergleichsergebnisse, Überläufe, ...)
- Segmentregister: CS, DS, SS, ES, FS, GS
- Instruction Pointer (EIP)
- x87 Register: ST0-ST7 als Stack organisiert, x87 Status- und Control-Words
- weitere Register (Control, Debug Registers, Performance Counters, ...)

Flagsregister

0	2	4	6	7	8	9	10	11	12-15	16	17	18	19	20	21	22-31
CF	PF	AF	ZF	SF	TP	IF	DF	OF		RF	VM	AC	VIF	VIP	ID	

Arithmetische Flags:

Bit	Name	Beschreibung
CF	Carry	Carry oder Borrow nach höchstwertigstem Bit
PF	Parity	gibt an ob unterstes Byte gerade/ungerade Zahl bits enthält
AF	Adjust	Carry nach 4 untersten Bits (für BCD Arithmetik)
ZF	Zero	Ergebnis ist 0
SF	Sign	höchstwertigstes Bit im Ergebnis ist gesetzt (negativer Wert)
OF	Overflow	Carry oder Borrow nach zweithöchsten Bit (für vorzeichenbehaftete Zahlen)

Vergleiche

Nach CMP oder SUB Befehl:

Vergleich	Unsigned		Signed	
	Name	Bits	Name	Bits
<	B	CF	L	genau SF oder OF
≤	BE	CF oder ZF	LE	ZF oder genau SF or OF
=	E	ZF	E	ZF
≠	NE	¬ZF	NE	¬ZF
>	A	¬CF und ¬ZF	G	¬ZF und weder SF noch OF
≥	AE	ZF und ¬CF	GE	ZF oder weder SF noch OF

Kapitel 9: Einführung in (Intel) 80x86 Assembler

- 1 Geschichte
- 2 Programmiermodell
- 3 Befehlssatz**
- 4 Konventionen
- 5 Beispiele
- 6 SSE
- 7 Literatur

CISC vs. RISC

x86 ist eine CISC (complex instruction set computing) Architektur.

- Reichhaltiger Befehlssatz, insbesondere viele verschiedene Adressierungsmodi für Daten.
- Programme sind kompakt.
- 2-Address-Code. Ziel einer Operation muss gleich einem der Quelloperanden sein: Befehle müssen $A = A + B$ Form haben.
- Prozessoren übersetzen CISC-Befehle intern in Microcode mit RISC Eigenschaften (um Pipelining zu ermöglichen).

Adressierungsmodi

Die meisten Befehle erlauben verschiedene Varianten um ihre Operanden zu erhalten:

- Konstante Werte (immediates)
- Register
- „Address-Mode“: Wert aus Speicher laden

Mögliche Adressberechnungen

$$\text{addr} = \text{Const} + \text{Base} + \text{Index} * \text{Scale}$$

- **Const** — 8-, 16- oder 32-Bit Konstante die im Befehl kodiert wird.
- **Base** — beliebiges Registers
- **Index** — beliebiges Registers ausser ESP
- **Scale** — 1, 2, 4 oder 8
- Komponenten sind Optional, Mindestens Const oder Base muss gegeben sein.

Assembler Syntax (AT&T)

Befehle

- Register: %eax, %esp, ...
- Konstanten: \$5, \$0x32, symbol, ...
- Address-Mode: Const, Const(Base), Const(Base, Index, Scale)
- Befehle bekommen ein Suffix um ihre Breite zu signalisieren: b, w, l, q für 8-, 16-, 32- oder 64-bit Breite Operationen.
- Bei mehreren Operanden wird erst der Quelloperand, dann der Zieloperand angegeben. `addl $4, %eax`

Beispiele

```
xorl %eax, %eax
subl $4, %esp
movl array+20(%eax,%ecx,4), %eax
incl (%esp)
```

Assembler Syntax (AT&T)

Assembler Direktiven

- Label: `name: —` Namensvergabe für Programmstellen
- Export/Import: `.globl name` — Linker löst Namen auf.
- Daten/Code-Segment: `.data, .text`
- Datenwerte: `.byte, .word, .long`

Beispiel

Globale Variable **int** `var = 42;`

```
.data  
.globl var  
var:  
.long 42
```

Grundlegende Befehle

mov	Daten kopieren
add	Addition
sub	Subtraktion
neg	Negation
inc	Addition von 1
dec	Subtraktion von 1
imul	Multiplikation
mul	(unsigned) Multiplikation, Ergebnis in EAX:EDX
imul	mit einem Operand wie mul aber signed statt unsigned
div	Division. Dividend stets in EAX:EDX, Divisor wählbar
and	Bitweises Und
or	Bitweises Oder
xor	Bitweises exklusives Oder
not	Bitweises invertieren
shl	Linksshift
shr	Rechtsshift
sar	(signed) Rechtsshift

Grundlegende Befehle

jmp	unbedingter Sprung
cmp	Werte vergleichen
jCC	bedingter Sprung
setCC	Register abhängig von Testergebnis setzen
call	Unterfunktion Aufrufen
ret	Aus Funktion zurückkehren
push	Wert auf den Stack legen und ESP vermindern
pop	Wert vom Stack legen und ESP erhöhen
int	„interrup“ Routine Aufrufen (nötig für Systemaufrufe)
lea	Führt Adressrechnung durch, schreibt Ergebnis in Register „3-Adressmode Addition“.

Kapitel 9: Einführung in (Intel) 80x86 Assembler

1 Geschichte

2 Programmiermodell

3 Befehlssatz

4 Konventionen

5 Beispiele

6 SSE

7 Literatur

Funktionsaufrufe

call Befehl: Rücksprungadresse (Adresse nach dem call) wird auf den Stack gelegt, danach wird die Zieladresse angesprungen. Parameter- und Ergebnisübergabe hängen von Aufrufkonventionen ab. Verbreitete Konventionen:

- **C** — Parameter werden auf den Stack gelegt, Ergebnis in EAX/ST0. Aufrufer räumt Parameter vom Stack.
- **Pascal** — Parameter werden auf den Stack gelegt, Ergebnis in EAX/ST0. Aufgerufene Funktion räumt Parameter vom Stack.
- **fastcall** — Die Ersten 2 Parameter in ECX, EDX, der Rest auf dem Stack. Ergebnis in EAX/ST0. Aufrufer räumt Parameter vom Stack. *nicht standardisiert, leichte Unterschiede zwischen Compilern*

Frame Zeiger

Der Framezeiger zeigt auf den Activation Record der aktuellen Funktion.

Standard Prolog:

```
push %ebp
movl %esp, %ebp
subl $XX, %esp # XX bytes fuer activation record allozieren
```

Standard Epilog:

```
popl %ebp
ret
```

Falls Stackgröße zur Übersetzungszeit bekannt und nicht debuggt werden soll, kann der Framezeiger weggelassen werden und EBP anderweitig genutzt werden. (-fomit-frame-pointer Option bei gcc)

Kapitel 9: Einführung in (Intel) 80x86 Assembler

- 1 Geschichte
- 2 Programmiermodell
- 3 Befehlssatz
- 4 Konventionen
- 5 Beispiele**
- 6 SSE
- 7 Literatur

Beispiel - printf aufrufen

```
.data
.STR0:
.string "Hello\n"

.text
.globl main
main:
    # Argument auf den Stack legen
    pushl $.STR0
    # Funktion aufrufen
    call printf
    # Argument vom Stack entfernen
    addl $4, %esp

    # Ergebnis fuer "main" setzen
    movl $0, %eax
    # Zurueckkehren
    ret
```

Beispiel - Funktion die 2 Zahlen addiert

```
.text
.globl add
add:
    # Frame Zeiger sichern und neu setzen
    push %ebp
    movl %esp, %ebp
    # Argumente laden
    movl 4(%ebp), %eax
    movl 8(%ebp), %edx
    # Addieren ( $\$eax = \$eax + \$edx$ )
    addl $edx, $eax
    # (alten) Frame Zeiger wiederherstellen und return
    popl %ebp
    ret
```

Fibonacci Funktion

```
.globl fib
    .p2align 4,,15
fib:
# Argument laden
    movl 4(%esp), %edx
# Faelle n==0 und n==1 behandeln
    xorl %eax, %eax
    cmpl $0, %edx
    je .return
    movl $1, %eax
    cmpl $1, %edx
    je .return
# Callee-saves sichern
    pushl %ebp
    pushl %edi
```


Fibonacci Funktion (Fortsetzung)

```
# fib(n-1) aufrufen
    movl %edx, %ebp
    decl %edx
    pushl %edx
    call fib
    movl %eax, %edi
# fib(n-2) aufrufen
    leal -2(%ebp), %edx
    pushl %edx
    call fib
# fib-Argumente vom Stack nehmen
    addl $8, %esp
# Ergebnis berechnen
    addl %edi, %eax
# Callee-saves wiederherstellen
    popl %edi
    popl %ebp
.return:
    ret
```

Kapitel 9: Einführung in (Intel) 80x86 Assembler

- 1 Geschichte
- 2 Programmiermodell
- 3 Befehlssatz
- 4 Konventionen
- 5 Beispiele
- 6 SSE**
- 7 Literatur

SSE (Streaming SIMD Extensions)

Eingeführt mit Pentium III (1999). Zusätzliche Befehle für Multimedia nach dem SIMD (Single Instruction Multiple Data) Prinzip. Bei angepasstem Code oft deutliche Geschwindigkeitssteigerungen.

- 8 zusätzliche 128-Bit Register (für 4 float/int oder 2 double Werte)
- Neue Befehle (Arithmetic, Comparison, Logical, Shuffle, Conversion, ...)

Nachteile

- Programmierung nur manuell oder mit speziellen Bibliotheken.
- Code läuft nur auf modernen CPUs.

Skalarprodukt in ANSI C

```
float scalar_product(float *xs, float *ys, int k) {  
    float result = 0.0;  
  
    for (int i = 0; i < k; ++i)  
        result += xs[i] * ys[i];  
  
    return result;  
}
```

Skalarprodukt SSE (gcc mit Builtins)

```
float scalar_product_sse(float *xs, float *ys, int k) {  
    /* Datentyp fuer SSE Werte */  
    typedef float v4sf __attribute__((vector_size(16)));  
    /* Immer 4 Werte auf einmal berechnen */  
    v4sf result = {0, 0, 0, 0};  
    assert(k % 4 == 0);  
    for (int i = 0; i < k; i += 4) {  
        /* Werte in SSE Register laden, multiplizieren, addieren */  
        v4sf X = __builtin_ia32_loadups(&xs[i]);  
        v4sf Y = __builtin_ia32_loadups(&ys[i]);  
        v4sf mul = __builtin_ia32_mulps(X, Y);  
        result = __builtin_ia32_addps(result, mul);  
    }  
    /* Werte zurueck in Normale Variable, Addieren */  
    float temp[4]; __builtin_ia32_storeups(temp, result);  
    return temp[0] + temp[1] + temp[2] + temp[3];  
}
```

⇒ Faktor 3 schneller auf Core 2 Duo.

Kapitel 9: Einführung in (Intel) 80x86 Assembler

- 1 Geschichte
- 2 Programmiermodell
- 3 Befehlssatz
- 4 Konventionen
- 5 Beispiele
- 6 SSE
- 7 Literatur**

- Ausführliche Dokumentation:
`http://www.intel.com/products/processor/manuals/`
- Knappe Übersicht:
`http://www.posix.nl/linuxassembly/nasmdohtml/nasmdoca.html`
- Gut organisierte Sammlung von Dokumenten zu x86:
`http://www.sandpile.org`
- Aufrufkonventionen und Optimierungstechniken:
`http://www.agner.org/optimize/`