

# Arithmetik (Zerlegung von Ausdrücken)

**Ziel: Vereinfachung von Ausdrücken:**

- Zerlegung der Ausdrücke entsprechend der Grammatik
- Einführen von temporären Variablen

**Beispiel**  $y = (x + 2z)/(z - \sin x)$  :

t1 = 2 * z;		t1 = 2 * z;
t2 = x + t1;	Optimierung	t1 = x + t1;
t3 = sin(x);	=====>	t2 = sin(x);
t4 = z - t3;		t2 = z - t2;
y = t2 / t4;		y = t1 / t2;

**Optimierung:**

unnötig viele temporäre Variablen vermeiden:

- Anzahl der (schnellen) Register beschränkt
- Speicherzugriffe langsam

# Arithmetik (Zerlegung von Ausdrücken)

**Jeder Ausdruck lässt sich in Teilausdrücke zerlegen, die die Form**

- $y = (x)$
- $y = \text{op } x$  mit  $\text{op} \in \{-, +\}$
- $y = a \text{ op } b$  mit  $\text{op} \in \{+, -, *, /, \text{mod}\}$
- $y = \text{fn}(a_0, a_1, \dots, a_{N-1})$  mit  $\text{fn}()$  beliebige Funktion/Prozedur

**haben.**

**Möglichkeiten:**

- Drei-Adress-CPU's
- Zwei-Adress-CPU's  
(Ziel ist immer gleich dem ersten Operanden)
- Ein-Adress-CPU's  
(Ziel und erster Operand ist immer bestimmtes Register "Akkumulator")

# Arithmetik (Zerlegung von Ausdrücken)

*Umsetzung auf Assembler bietet Hardware-Designer mehrere Möglichkeiten:*

- Drei-Adress-CPUs (z.B. m88100)
- Zwei-Adress-CPUs (z.B. i80x86, m680x0)  
(Ziel ist immer gleich einem Operanden)
- Ein-Adress-CPUs (z.B. z80, 6502, m68x05)  
(Ziel und ein Operand ist immer bestimmtes Register "Akkumulator")

**Beispiel**  $x = y + z$ :

Drei-Adress-CPU	Zwei-Adress-CPU	Ein-Adress-CPU
add y, z, x	mov y, x add z, x	load y # y=>akku add z # akku=akku+z store x # akku=>x

# Arithmetik (Integer, Hinweis)

*Da alle Integer-Variablen im Speicher bzw. in Registern abgelegt werden, können sie nicht beliebig große Werte annehmen (=> siehe OTR I).*

*Ähnliches gilt für die Arithmetik:*

*Da alle Werke der ALU nur für eine bestimmte Anzahl von Bits ausgelegt sind, können auch hier die Werte nicht beliebig groß werden. Gilt generell für alle Rechner!*

**Beispiel:**

mit beliebig vielen Stellen gerechnet:

$$\begin{array}{r} 10110010_{(2)} \\ 178_{(10)} \end{array} + \begin{array}{r} 11001101_{(2)} \\ 205_{(10)} \end{array} = \begin{array}{r} 10111111_{(2)} \\ 383_{(10)} \end{array}$$

nur mit acht Stellen gerechnet:

$$\begin{array}{r} 10110010_{(2)} \\ 178_{(10)} \end{array} + \begin{array}{r} 11001101_{(2)} \\ 205_{(10)} \end{array} \begin{array}{l} ?= \bar{1}0111111_{(2)} \\ ?= 127_{(10)} \end{array}$$

## Arithmetik (Integer)

*Negation:  $x = -y$*

# 8-Bit-Werte	# 16-Bit-Werte	# 32-Bit-Werte
<code>movb y, %al</code>	<code>movw y, %ax</code>	<code>movl y, %eax</code>
<code>negb %al</code>	<code>negw %ax</code>	<code>negl %eax</code>
<code>movb %al, x</code>	<code>movw %ax, x</code>	<code>movl %eax, x</code>

*kürzer:*

# 8-Bit-Werte	# 16-Bit-Werte	# 32-Bit-Werte
<code>movb y, x</code>	<code>movw y, x</code>	<code>movl y, x</code>
<code>negb x</code>	<code>negw x</code>	<code>negl x</code>

*Je nachdem, ob die Variablen  $x, y$  im Speicher oder in Registern liegen, ist die kürzere Version langsamer oder schneller als die längere.*

## Arithmetik (Integer)

*Addition:  $x = y + z$*

# 8-Bit-Werte	# 16-Bit-Werte	# 32-Bit-Werte
<code>movb y, %al</code>	<code>movw y, %ax</code>	<code>movl y, %eax</code>
<code>addb z, %al</code>	<code>addw z, %ax</code>	<code>addl z, %eax</code>
<code>movb %al, x</code>	<code>movw %ax, x</code>	<code>movl %eax, x</code>

*Subtraktion:  $x = y - z$*

# 8-Bit-Werte	# 16-Bit-Werte	# 32-Bit-Werte
<code>movb y, %al</code>	<code>movw y, %ax</code>	<code>movl y, %eax</code>
<code>subb z, %al</code>	<code>subw z, %ax</code>	<code>subl z, %eax</code>
<code>movb %al, x</code>	<code>movw %ax, x</code>	<code>movl %eax, x</code>

# Arithmetik (Integer, Carry-Bit)

**Problem:** Rechnung ausserhalb des „normalen“ Wertebereichs einer Maschine.

**Beispiel aus der Schule:**

678	1234
+ 789	- 453
-----	-----
1110 <= Übertrag	1100 <= Übertrag
1467	781

**Übertrag** entweder '0' oder '1' => „Carry-Bit“ / „Borrow-Bit“

**Beispiel für „long long“-Rechnung:**  $(x_1x_0) = (y_1y_0) + (z_1z_0)$  bzw.  $(x_1x_0) = (y_1y_0) - (z_1z_0)$

<code>movl y0, %eax</code>	<code>movl y0, %eax</code>
<code>addl z0, %eax</code>	<code>subl z0, %eax</code>
<code>movl %eax, x0</code>	<code>movl %eax, x0</code>
<code>movl y1, %eax</code>	<code>movl y1, %eax</code>
<code>adcl z1, %eax</code>	<code>sbb z1, %eax</code>
<code>movl %eax, x1</code>	<code>movl %eax, x1</code>

# Arithmetik (Integer)

**Multiplikation, Division:**

**Beispiel**  $x = y \cdot z$  bzw.  $x = y/z$  (i80x86):

<code>movl y, %eax</code>	<code>movl y, %eax</code>
<code>imull z</code>	<code>movl \$0, %edx</code>
<code>movl %eax, x</code>	<code>idivl z</code>
<code>/* %edx enthält Überlauf */</code>	<code>movl %eax, x</code>
	<code>/* %edx enthält Rest d. Division */</code>

**Besonderheiten / Probleme:**

- Multiplikation kann große Überläufe erzeugen (Carry-Bit reicht **nicht!**)  
=> Extra-Register für Überlauf
- `idiv`-Befehl berechnet gleichzeitig  $y/z$  und  $y \bmod z$ .
- Multiplikation und Division aufwändig in Hardware implementierbar  
(in Mikro-Controllern selten als Maschinen-Befehl implementiert!)

# Arithmetik (Komplexität)

**Wichtig:**

**Komplexität des Assembler-Codes vergleichbar mit Komplexität des zugehörigen Hochsprachen-Programms**

**Beweisidee für i80x86 für Integer-Ausdrücke (rekursiver Beweis über Term-Aufbau):**

**1. Fall: Ausdruck hat die Form var oder const:**

```
movl var, %eax      bzw.   movl $const, %eax
```

**2. Fall: Ausdruck hat die Form +term oder -term:**

**(Annahme: Ergebnis der Berechnung von term liegt bereits im Register %eax vor)**

```
keine Operation notwendig   bzw.   negl %eax
```

**3. Fall: Ausdruck hat die Form term+term, term-term, term\*term, term/term:**

**(Annahme: Operand 1 liegt im Register %eax, Operand 2 liegt im Register %ebx)**

```
addl %ebx, %eax      ähnlich: subl, imull, idivl
```

# Arithmetik (Integer)

**„Ersatz“ für fehlende/langsame Multiplikationsbefehle:**

**Beobachtung:**

Häufig werden nur Multiplikationen mit einem konstanten Multiplikator durchgeführt. Diese können auf sukzessive Additionen zurückgeführt werden.

**Beispiel  $y = 10x$ :**

$$y = 10x = 5(2x) = 4(2x) + 1(2x) = 2(2(2x)) + 2x$$

```
movl x, %eax
addl %eax, %eax
movl %eax, %ebx      # => %ebx = 2*x
addl %eax, %eax
addl %eax, %eax      # => %eax = 8*x
addl %ebx, %eax
movl %eax, y
```

**Nicht 10-mal addieren (Hier: 5 Arithmetikbefehle statt 10)!**

# Arithmetik (Integer)

„Ersatz“ für fehlende/langsame Multiplikationsbefehle:

## 2. Beobachtung:

Häufig werden Multiplikationen mit konstanten 2er-Potenzen durchgeführt. Diese können hardware-mäßig leicht auf sogenannte Shift-Operationen abgebildet werden.

Ähnlich: Divisionen durch 2er-Potenzen.

### Beispiel Multiplikation mit 2er-Potenz:

$$1181_{(10)} \cdot 32_{(10)} = 10010011101_{(2)} \cdot 100000_{(2)} = 1001001110100000_{(2)}$$

(entspricht Anhängen der entsprechenden Anzahl von Nullen)

### Beispiel Division durch 2er-Potenz (mit Abrundung):

$$1181_{(10)} / 32_{(10)} = 10010011101_{(2)} / 100000_{(2)} = 100100_{(2)}$$

(entspricht dem Löschen der entsprechenden Anzahl von Ziffern)

# Arithmetik (Integer)

Shift-Befehle: *sal, shl, sar, shr*

## Beispiele (i80x86):

```
/* %ax enthält 0011.0110.1110.1010 */
salw $3, %ax
/* %ax enthält 1011.0111.0101.0000 */
shr $4, %ax
/* %ax enthält 0000.1011.0111.0101 */
shlw $6, %ax
/* %ax enthält 1101.1101.0100.0000 */
sarw $3, %ax
/* %ax enthält 1111.1011.1010.1000 */
```

**Unterschied *sar(sal)* / *shr(shl)*: Vorzeichenbeachtung**

## Arithmetik (Bit)

*Bit-weises Oder (or), Und (and), Exklusiv-oder (xor) oder Bit-weise Invertierung (not):*

*Es werden N Bits (8, 16, 32 oder 64) gleichzeitig verarbeitet. Beispiele:*

```
movw $0x1234, %ax
andw $0xf61c, %ax
/* %ax hat jetzt den Wert 0x1214 */

movl $0x11111111, %eax
orl $0x12345678, %eax
/* %eax hat jetzt den Wert 0x13355779 */

movb $0x24, %al
xorb $0xff, %al
/* %al hat jetzt den Wert 0xdb */

movl $12345678, %eax
notl %eax
/* %eax hat jetzt den Wert 0xedcba987 */
```

## Arithmetik (Hinweise)

*Zur Beschleunigung / Verkürzung der Programme gibt es für sehr häufig vorkommende Ausdrücke Extra-Maschinenbefehle:*

*Beispiele:*

```
addl $1, x      => incl x
subw $1, y      => decw y
movl $0, %eax   => xorl %eax, %eax
```

# Arithmetik (Gleitpunkt)

*Berechnung von Gleitpunkt-Ausdrücken entsprechen weitgehend der Berechnung von Integer-Ausdrücken:*

- Zerlegen der Ausdrücke gemäß der Grammatik
- Einführen von temporären Variablen

*Assembler-Operationen existieren bei großen Prozessoren (Beispiel: i80x86)*

Integer	Gleitpunkt
negb, negw, negl	fchs
addb, addw, addl	fadds, faddl
subb, subw, subl	fsubs, fsubl
imulb, imulw, imull	fmuls, fmull
idivb, idivw, idivl	fdivs, fdivl

# Arithmetik (Gleitpunkt)

*Bei kleineren Prozessoren existieren keine Floating-Point-Operationen*

- Berechnung über getrennte Behandlung des Vorzeichens, der Mantisse und des Exponenten (=> siehe z.B. OTRS I)

*Bei vielen Prozessoren existiert keine Floating-Point-Division*

- Berechnung von  $1/y$  mit Newton-Raphson-Algorithmus
- Berechnung von  $x/y$  durch  $x * 1/y$

## Arithmetik (Newton-Raphson)

**Berechnung von  $z = 1/y$  ( $y \neq 0$ ):**

$$z = \frac{1}{y} = \frac{1}{m2^e} = \frac{1}{m}2^{-e} = g2^{-e} \text{ mit } 0.5 \leq m < 1$$

**Vorzeichen bleibt**

**Exponent (bzw. Charakteristik) lässt sich einfach berechnen**

**Berechnung von  $g = 1/m$  ( $m \neq 0$ ) durch Iterationsverfahren nach Newton-Raphson:**

- $g_0 = 1$  (besserer Startwert über Tabellen möglich)
- $g_{n+1} = g_n(2 - mg_n)$  (zwei Multiplikationen, eine Subtraktion)

**Anhaltspunkt (Abhängig vom Startwert) für Genauigkeit: 4 Iterationen reichen für 64 Bit; je Iteration verdoppelt sich die Anzahl der gültigen Bits**

## Arithmetik (Taylor-Reihen)

**Problem: wie berechnet man (mit einfachen Mitteln) z.B.  $\sin(x)$ ,  $\exp(x)$ , ...**

**Idee: Taylor-Reihen (siehe z.B. Bronstein / Semendjajew)**

**Ist  $f(x)$  in  $(x_0 - \alpha, x_0 + \alpha)$  ( $\alpha > 0$ )  $n + 1$  mal differenzierbar.**

**Dann gilt für  $x \in (x_0 - \alpha, x_0 + \alpha)$ :**

$$f(x) = \sum_{v=0}^n \frac{f^{(v)}(x_0)}{v!} (x - x_0)^v + R_n(x)$$

**mit**

$$R_n(x) = \frac{f^{(n+1)}(x_0 + \vartheta(x - x_0))}{(n+1)!} (x - x_0)^{n+1}, \vartheta \in (0, 1)$$

## Arithmetik (Taylor-Reihen, Beispiele)

$$e^x = 1 + \frac{x}{1!} + \frac{x^2}{2!} + \frac{x^3}{3!} + \frac{x^4}{4!} + \dots$$

$$\sin x = x - \frac{x^3}{3!} + \frac{x^5}{5!} - \frac{x^7}{7!} + \frac{x^9}{9!} - \dots, \cos x = 1 - \frac{x^2}{2!} + \frac{x^4}{4!} - \frac{x^6}{6!} + \frac{x^8}{8!} - \dots$$

$$a \sin x = x + \frac{x^3}{2 \cdot 3} + \frac{1 \cdot 3 x^5}{2 \cdot 4 \cdot 5} + \frac{1 \cdot 3 \cdot 5 x^7}{2 \cdot 4 \cdot 6 \cdot 7} + \dots$$

...

## Arithmetik

*Verschiedene „Tricks“, um die Konvergenz (Genauigkeit des Ergebnisses bzw. Geschwindigkeit der Berechnung) der Algorithmen zu verbessern:*

$$e^x = (e^{x/2})^2$$

$$\ln(m2^n) = \ln m + n \ln 2, \quad \ln(1-x) = -\left(x + \frac{x^2}{2} + \frac{x^3}{3} + \frac{x^4}{4} + \dots\right) \quad \text{mit } 0.5 \leq x < 1$$

$$\sin x = \sin(x \bmod 2\pi)$$

Horner-Schema (statt Taylor-Schema):  $((a_{n-1}x + a_{n-2}) \dots + \dots)x + a_1)x + a_0$   
schneller und genauer als  $a_{n-1}x^{n-1} + a_{n-2}x^{n-2} + \dots + a_1x + a_0$

=> Mathematik für Ingenieure, Numerik I + II