Speicherzugriffe in Assembler

Bisher

- `not eax`
- `cmp cx, dx`
- `add rax, 1`

Jetzt: Instruktionen die auf dem Speicher arbeiten

Speicherzugriffe in Assembler

```
add dword ptr [rax], 1
```

- `[rax]` - Speicheroperand; `rax` ist Adresse/Pointer
- `dword ptr` - Pointer Typ, bzw. Größe des Werts im Speicher; hier 32-bit.

- Register/Immediates häufig durch Speicheroperanden austauschbar
- Aber: max. ein (expliziter) Speicheroperand pro Instruktion
- `mov` zum laden/schreiben von Werten aus/in den Speicher

Speicheroperanden

`mov rax, [`*`Base + Index*Scale + Displacement`*`]`

Komponente	Wert	Bedeutung
`Base`	64-bit Reg.	(Basis-)Adresse eines Pointers/Arrays
`Index*`	64-bit Reg.	Index eines Array Elements
`Scale`	1,2,4,8	bestimmter Größe
`Displacement`	32-bit Imm.	Statischer Offset

- Alle Komponenten optional
- Default *`Base`*, *`Index*Scale`* und *`Displacement`* ist 0

Quiz: Speicheroperanden (1)

Wie kann man den 32-bit Wert an der Adresse `rdi`
auf den 32-bit Wert an der Adresse `rsi` addieren?

☐ `add [rsi], [rdi]`

☐
```
mov eax, [rdi]
add [rsi], eax
```

Quiz: Speicheroperanden (2)

Angenommen `rdi` ist die Basis-Adresse eines Arrays mit 32-bit großen Elementen. Wie kann man das Element am Index `ecx` von `eax` subtrahieren?

☐
```
sub eax, [rdi+4*ecx]
```

☐
```
sub eax, [rdi+4*rcx]
```

☐
```
mov ecx, ecx
sub eax, [rdi+4*rcx]
```

☐
```
sub eax, [edi+4*ecx]
```

Quiz: Speicheroperanden (3)

Wie kann man `eax` auf den 32-bit Wert an der Adresse `0x7ffe01234567` addieren?

☐
```
add [0x7ffe01234567], eax
```

☐
```
mov rdi, 0x7ffe01234567
add [rdi], eax
```

`rip`-relative Adressierung

`mov rax, [rip + *Displacement*]`

- Basis-Register `rip`
- Kein (skalierter) Index
- 32-bit Displacement
- Displacement häufig Label
  z.B. `mov rax, [rip + my_number]`
- Adressberechnung dann relativ zu `rip`
- Zum position-independent Laden von hardgecodeten Werten

Quiz: `rip`-relative Adressierung

Angenommen an dem Label `array` befindet sich ein statisches Array mit 4 32-bit Elementen. Wie können wir den Wert am Index 2 position-independent in `eax` laden?

☐
```
mov rdi, array
mov eax, [rdi + 8]
```

☐
```
mov eax, [array + 8]
```

☐
```
mov eax, [rip + array + 8]
```

☐
```
lea rdi, [rip+array]
mov eax, [rdi+8]
```

Wichtige Zugriffsgrößen

```
mov size ptr [rax], 1
```

- Gibt Größe des Werts im Speicher an
- Nötig wenn Größe nicht implizit ersichtlich ist
  z.B. `mov byte ptr [rax], 42` (Immediate "42" ist 1 Byte groß)
- Optional wenn Größe implizit ersichtlich ist
  z.B. `mov [rax], cx` (`cx` ist 2 Byte groß)
  z.B. `mov word ptr [rax], cx`
- **Wichtig:** `ptr` nicht vergessen! Sonst wird *size* als Displacement interpretiert.
- Einige Pointer Typen:

size	byte	word	dword	qword
Bytes	1	2	4	8

Quiz: Wichtige Zugriffsgrößen (1)

Wie kann man ein NULL-Byte an die Adresse `rdi` schreiben?

- ☐
```
xor eax, eax
mov [rdi], al
```
- ☐
```
xor eax, eax
mov byte [rdi], al
```
- ☐
```
mov [rdi], 0
```
- ☐
```
mov byte ptr [rdi], 0
```

Quiz: Wichtige Zugriffsgrößen (2)

Wie kann man den 32-bit Wert an der Adresse 0x12345678 inkrementieren?

☐
```
add [0x12345678], 0x00000001
```

☐
```
mov eax, 1
add [0x12345678], eax
```

☐
```
mov rax, 1
add dword ptr [0x12345678], rax
```

☐
```
mov eax, 1
add dword ptr [0x12345678], eax
```

Die Instruktion `lea`

```
lea rax, [rdi + 2*rsi + 1]
```

- `lea` greift *nicht* auf den Speicher zu
- Schreibt Ergebnis der Berechnung des Speicheroperanden in Zielregister
- Zum Berechnen von Adressen (**l**oad **e**ffective **a**ddress)
- oder zum effizienten Berechnen einiger arithmetischer Ausdrücke

Quiz: Arithmetik mit `lea` (1)

Wie kann man `eax ← edx + 4*ecx - 2` berechnen?

☐
```
imul eax, ecx, 4
add eax, edx
add eax, -2
```

☐
```
lea eax, [edx + 4*ecx - 2]
```

Quiz: Arithmetik mit `lea` (2)

Wie kann man `eax ← 3*edx + 4*ecx - 5` berechnen?

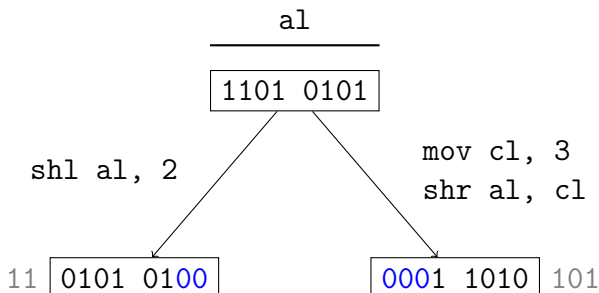
☐
```
lea eax, [3*edx + 4*ecx - 5]
```

☐
```
imul eax, edx, 3
imul ecx, ecx, 4
add eax, ecx
sub eax, 5
```

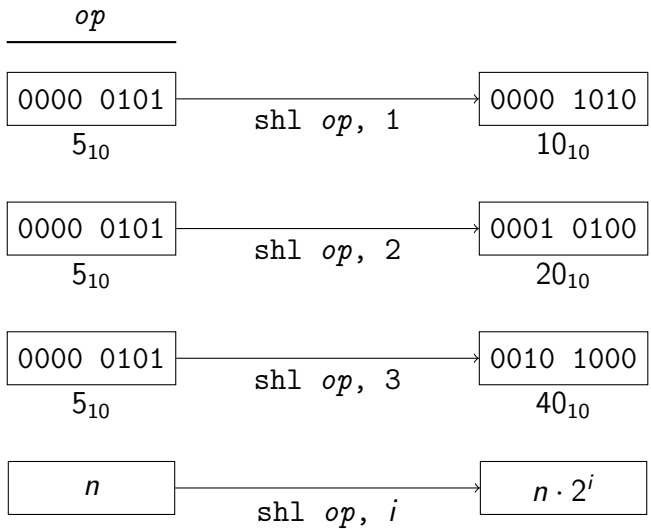
☐
```
lea eax, [rdx + 2*rcx]
lea eax, [rdx + 2*rax - 5]
```

Shifts

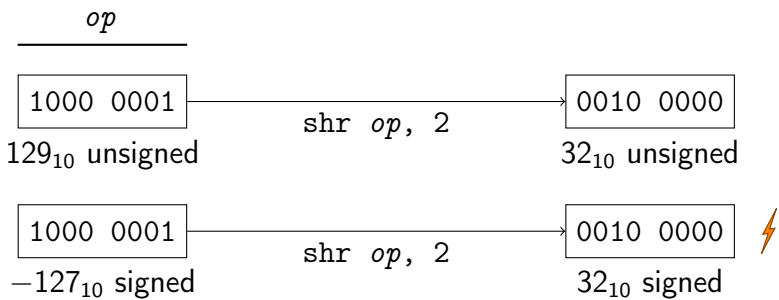
- Verschieben Bits nach links oder rechts → Links-/Rechtsshift
- `shl/shr` *reg/mem*`,` *imm8*`/cl`



Arithmetische Bedeutung von Linksshifts



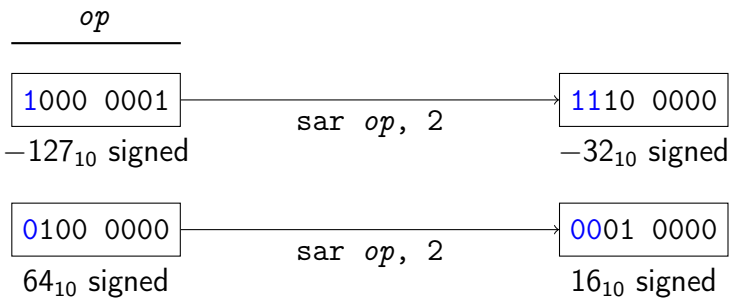
Arithmetische Bedeutung von Rechtsshifts



Arithmetischer Rechtsshift

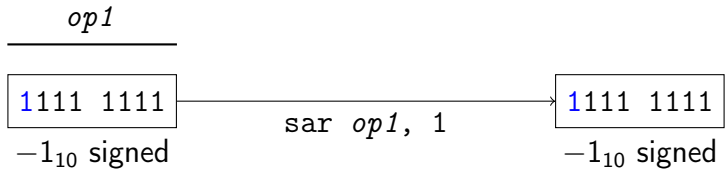
`sar` *reg/mem*, *imm8*/`cl`

- `sar` - Arithmetischer Rechsshift
- `shr` - Logischer Rechtsshift



Weiteres zu arithmetischen Shifts

- `sar` rundet auch negative Zahlen ab (Richtung $-\infty$)



- `sar` für unsigned Zahlen i.A. falsch→ hier `shr` verwenden
- `sal` - Arithmetischer Linksshift
- `shl` - Logischer Linksshift
- `sal` funktional identisch zu `shl`

Quiz: Arithmetische Bedeutung von Shifts (1)

Wie können wir $\texttt{rax} \leftarrow \texttt{rax} \cdot 2^{\texttt{dl}}$ berechnen?

☐ `sal rax, dl`

☐
```
mov cl, dl
shl rax, cl
```

Quiz: Arithmetische Bedeutung von Shifts (2)

Wie kann man $\texttt{rax} \leftarrow \lfloor \texttt{rax}/2 \rfloor$
mit nur einer Instruktion berechnen?

- ☐ `shr rax, 1`
- ☐ `sar rax, 1`
- ☐ Man kann es nicht ohne weiteres berechnen

Quiz: Arithmetische Bedeutung von Shifts (3)

Wie kann man $\texttt{rax} \leftarrow \lfloor \texttt{rax}/4 \rfloor$ berechnen wenn `rax` unsigned ist?

☐ `shr rax, 2`

☐ `sar rax, 2`

Quiz: Arithmetische Bedeutung von Shifts (4)

Wie kann man $\texttt{rax} \leftarrow \lfloor \texttt{rax}/4 \rfloor$ berechnen wenn `rax` signed ist?

☐ `shr rax, 2`

☐ `sar rax, 2`