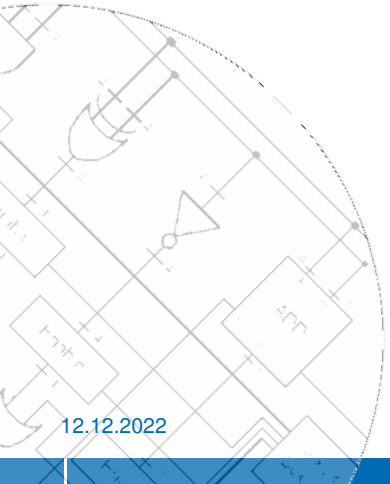
Rechnerarchitektur

Vorlesung 15: Instruktionen, Sprünge, Schleifen und der Stack

Prof. Dr. Martin Mauve



12.12.2022

Haben Sie noch Fragen zur letzten Vorlesung?

Thema: Programmaufbau und einfache Instruktionen

Fahrplan

Arithmetische Integer-Instruktionen

- Addieren von ganzen Zahlen:
  - `add op1, op2` (ohne Berücksichtigung des Carry-Flags)
    - op1 $=$ op1 $+$ op2
  - `adc op1, op2` (mit Berücksichtigung des Carry-Flags)
    - op1 $=$ op1 $+$ op2 $+ C$
- Subtrahieren von ganzen Zahlen:
  - `sub op1, op2` (ohne Berücksichtigung des Carry-Flags)
    - op1 $=$ op1 $-$ op2
  - `sbb op1, op2` (mit Berücksichtigung des Carry/Borrow-Flags)
    - op1 $=$ op1 $-$ op2 $- C$

Beispiel: Addition (mit Carry)

```
segment .data

wert1:    dd  0x00000000, 0xFFFFFFFF
wert2:    dd  0x00000000, 0x00000001

segment .bss
resultat:       resd 2

segment .text
        global  asm_main
asm_main:
    ...
    mov eax, [wert1+4]      ; Operanden laden
    mov ebx, [wert1]
    mov ecx, [wert2+4]
    mov edx, [wert2]
    dump_regs 1             ; Ausgabe der Register
    add eax, ecx            ; Addition
    dump_regs 2             ; Ausgabe der Register
    adc ebx, edx            ; Addition mit Carry
    dump_regs 3             ; Ausgabe der Register
    mov [resultat], eax     ; Ergebnis speichern
    mov [resultat+4], ebx   ; (als 64-Bit Little Endian)
    ...
```

Beispiel: Addition (mit Carry) – Ausgabe I

```
...
 mov eax, [wert1+4]      ; Operanden laden
 mov ebx, [wert1]
 mov ecx, [wert2+4]
 mov edx, [wert2]
 dump_regs 1             ; Ausgabe der Register
```

Ausgabe:

```
Register Dump # 1
EAX = FFFFFFFF EBX = 00000000 ECX = 00000001 EDX = 00000000
ESI = 40014580 EDI = BFFFF544 EBP = BFFFF4E8 ESP = BFFFF4C8
EIP = 0804844C FLAGS = 200286       SF       PF
```

Beispiel: Addition (mit Carry) – Ausgabe II

```
    dump_regs 1              ; Ausgabe der Register
    add eax, ecx             ; Addition
    dump_regs 2              ; Ausgabe der Register
    adc ebx, edx             ; Addition mit Carry
    dump_regs 3              ; Ausgabe der Register
    mov [resultat], eax      ; Ergebnis speichern
    mov [resultat+4], ebx    ; (als 64-Bit Little Endian)
   ...
```

Ausgabe:

```
Register Dump # 2
EAX = 00000000 EBX = 00000000 ECX = 00000001 EDX = 00000000
ESI = 40014580 EDI = BFFFF544 EBP = BFFFF4E8 ESP = BFFFF4C8
EIP = 08048458 FLAGS = 200257          ZF AF PF CF
Register Dump # 3
EAX = 00000000 EBX = 00000001 ECX = 00000001 EDX = 00000000
ESI = 40014580 EDI = BFFFF544 EBP = BFFFF4E8 ESP = BFFFF4C8
EIP = 08048464 FLAGS = 200202
```

Arithmetische Integer-Instruktionen

- Multiplikation von ganzen Zahlen
  - `mul op1` (Multiplikation von vorzeichenlosen Zahlen)
    - $ax = al \cdot op1$ (bei 8 Bit op1)
    - $dx : ax = ax \cdot op1$ (bei 16 Bit op1)
    - $edx : eax = eax \cdot op1$ (bei 32 Bit op1)
  - `imul op1` (Multiplikation von vorzeichenbehafteten Zahlen)
    - analog zu `mul`
    - weitere Formate (hier nicht besprochen)
- Division von ganzen Zahlen
  - `div op1` (Division von vorzeichenlosen Zahlen)
    - $al = ax \text{ div } op1$ und $ah = ax \mod op1$ (bei 8 Bit op1)
    - $ax = dx : ax \text{ div } op1$ und $dx = dx : ax \mod op1$ (bei 16 Bit op1)
    - $eax = edx : eax \text{ div } op1$ und $edx = edx : eax \mod op1$ (bei 32 Bit op1)
  - `idiv op1` (Division von vorzeichenbehafteten Zahlen)
    - analog zu `div`

Beispiel: Multiplikation

```
mov eax, 0xFFFFFFFF     ; Operanden laden
mov ecx, 0x00000010
dump_regs 1             ; Registerwerte ausgeben
mul ecx                 ; Multiplizieren
dump_regs 2             ; Registerwerte ausgeben
```

Ausgabe:

```
Register Dump # 1
EAX = FFFFFFFF EBX = 401579A8 ECX = 00000010 EDX = 40158E90
ESI = 40014580 EDI = BFFFF544 EBP = BFFFF4E8 ESP = BFFFF4C8
EIP = 0804843F FLAGS = 200286       SF       PF
Register Dump # 2
EAX = FFFFFFF0 EBX = 401579A8 ECX = 00000010 EDX = 0000000F
ESI = 40014580 EDI = BFFFF544 EBP = BFFFF4E8 ESP = BFFFF4C8
EIP = 0804844B FLAGS = 200A87 OF    SF       PF CF
```

Logische Instruktionen

arbeiten i.d.R. bitweise:

- `and op1, op2`
  - op1 = op1 AND op2
- `or op1, op2`
  - op1 = op1 OR op2
- `xor op1, op2`
  - op1 = op1 XOR op2
- `not op1`
  - op1 = Einerkomplement von op1 (=alle Bits invertieren)
- `neg op1`
  - op1 = Zweierkomplement von op1 (= (NOT op1) + 1)

Fahrplan

Verschiebe-Instruktionen

- `shr op1, op2` (Shift Right)
  - `op1` wird um `op2` Stellen nach rechts verschoben
  - von links werden 0en nachgeschoben
- `shl op1, op2` (Shift Left)
  - `op1` wird um `op2` Stellen nach links verschoben
  - von rechts werden 0en nachgeschoben
- `sar op1, op2` (Shift Arithmetic Right)
  - `op1` wird um `op2` Stellen nach recht verschoben
  - von links wird die Ziffer nachgeschoben, die vorher im höchstwertigen Bit stand
  - Sign Extension!
- `sal op1, op2` (Shift Arithmetic Left) Synonym für `shl`

Rotationsinstruktionen

- `ror op1, op2` (Rotate Right)
  - `op1` wird um `op2` Stellen nach rechts rotiert
  - von links wird das Bit übernommen, welches rechts herausrotiert wurde
- `rol op1, op2` (Rotate Left)
  - `op1` wird um `op2` Stellen nach links rotiert (analog zu `ror`)
- `rcr op1, op2` (Rotate Carry Right)
  - `C:op1` wird um `op2` Stellen nach rechts rotiert
  - von links wird das Bit übernommen, welches im Carry-Flag stand
  - rechts wird in das Carry-Bit hineinrotiert
- `rcl op1, op2` (Rotate Carry Left)
  - `C:op1` wird um `op2` Stellen nach links rotiert (analog zu `rcr`)

Fahrplan

Bedingte Sprünge

- bisher: lineare Ausführung von Instruktionen
- `jmp ende`
  - unbedingter Sprung zur Instruktion mit dem Label `ende`
- bedingte Sprünge:
  - verzweigen, wenn Flags in EFLAGS-Register gesetzt oder nicht gesetzt sind
- Beispiel für einen bedingten Sprung: `jo fehler`
  - falls das Overflow (O) Flag gesetzt ist, wird gesprungen;
  - wenn nicht, wird die nächste Instruktion ausgeführt.
- Sprungbefehle (Auszug):
  - `jo, jno, jz, jnz, jc, jnc`

Beispiel: bedingte Sprünge I

```
; based on skel.asm
%include "asm_io.inc"

;
; initialized data is put in the .data segment
;
segment .data
;
; These labels refer to strings used for output
;

input_prompt db     "Bitte geben Sie eine Zahl ein ", 0

segment .text
        global  asm_main
```

Beispiel: bedingte Sprünge II

```
asm_main:
        enter   0,0                 ; setup routine
        pusha

        ; clear eax and ebx
        xor     eax, eax
        xor     ebx, ebx

        ; Ask user for input
        mov     eax, input_prompt
        call    print_string
        call    read_int
        dump_regs 1

        ; calc and jump
        mov     ebx, 0x7FFFFFFD
```

Beispiel: bedingte Sprünge III

```
        add     eax, ebx
        dump_regs 2

        jno     ende
        mov     eax, 0x0

ende:
        dump_regs 3

        popa
        mov     eax, 0              ; return back to C
        leave
        ret
```

Beispiel: bedingte Sprünge IV

Ausgabe:

```
Register Dump # 1
EAX = 00000080 EBX = 40157901 ECX = 401579A8 EDX = 40158E90
ESI = 40014580 EDI = BFFFF544 EBP = BFFFF4E8 ESP = BFFFF4C8
EIP = 0804843B FLAGS = 200A92 OF    SF    AF
Register Dump # 2
EAX = 00000000 EBX = 40157901 ECX = 401579A8 EDX = 40158E90
ESI = 40014580 EDI = BFFFF544 EBP = BFFFF4E8 ESP = BFFFF4C8
EIP = 08048449 FLAGS = 200A92 OF    SF    AF
```

Vergleiche I

- Springen auf Grund von Vergleichen
- `cmp op1, op2`
  - berechnet op1 − op2
  - speichert kein Ergebnis
  - setzt aber die entsprechenden Flags im EFLAGS-Register
  - bei vorzeichenlosen Zahlen:
    - op1 = op2 wenn Z-Flag gesetzt ist
    - op1 < op2 wenn Z-Flag nicht gesetzt ist und C-Flag gesetzt ist
    - op1 > op2 wenn Z-und C-Flag nicht gesetzt sind
  - bei vorzeichenbehafteten Zahlen:
    - ähnlich, aber etwas komplizierter
- prinzipiell reicht das!
  - Es ist aber nicht besonders praktisch, da mehrere Instruktionen für eine Auswahl benötigt werden.
  - daher: spezielle bedingte Sprünge für die wichtigsten Fälle

Vergleiche II

- beim Vergleich vorzeichenbehafteter Zahlen:
  - `JE` (op1 $=$ op2), `JNE` (op1 $\neq$ op2)
  - `JL` und `JNGE` (op1 $<$ op2), `JLE` und `JNG` (op1 $\leqslant$ op2)
  - `JG` und `JNLE` (op1 $>$ op2), `JGE` und `JNL` (op1 $\geqslant$ op2)
- beim Vergleich vorzeichenloser Zahlen
  - `JE` (op1 $=$ op2), `JNE` (op1 $\neq$ op2) (wie oben!)
  - `JB` und `JNAE` (op1 $<$ op2), `JBE` und `JNA` (op1 $\leqslant$ op2)
  - `JA` und `JNBE` (op1 $>$ op2), `JAE` und `JNB` (op1 $\geqslant$ op2)

Beispiel: Vergleiche

```
      mov eax, 0x10     ; Operanden laden
      mov ebx, 0x20
      cmp eax, ebx      ; Der Vergleich
      dump_regs 1       ; Registerwerte ausgeben
      jl bgr
      mov ecx, eax
      jmp ende
bgr:  mov ecx, ebx
ende: dump_regs 2       ; Registerwerte ausgeben
```

Ausgabe:

```
Register Dump # 1
EAX = 00000010 EBX = 00000020 ECX = 401579A8 EDX = 40158E90
ESI = 40014580 EDI = BFFFF544 EBP = BFFFF4E8 ESP = BFFFF4C8
EIP = 08048441 FLAGS = 200287       SF       PF CF
Register Dump # 2
EAX = 00000010 EBX = 00000020 ECX = 00000020 EDX = 40158E90
ESI = 40014580 EDI = BFFFF544 EBP = BFFFF4E8 ESP = BFFFF4C8
EIP = 08048456 FLAGS = 200287       SF       PF CF
```

Schleifen

Für die elegante Unterstützung von for-Schleifen gibt es folgende Instruktionen:

- `loop marke`
  - dekrementiert ecx (d.h. `ecx := ecx - 1`);
  - springt zum Label `marke`, wenn danach ecx $\neq 0$.
- `loope marke` (`loopz marke`)
  - dekrementiert ecx (ohne EFLAGS zu modifizieren);
  - springt zum Label `marke`, wenn danach ecx $\neq 0$ und das Z Flag gesetzt ist.
- `loopne marke` (`loopnz marke`)
  - dekrementiert ecx (ohne EFLAGS zu modifizieren);
  - springt zum Label `marke`, wenn danach ecx $\neq 0$ und das Z Flag nicht gesetzt ist.

Beispiel: Schleifen

```
        mov eax, 0      ; eax enthält die Summe
        mov ecx, 10     ; ecx enthält die Laufvariable
start:  add eax, ecx    ; Aufsummieren
        loop start      ; ecx dekrementieren und zurückspringen
        dump_regs 1     ; Registerinhalte ausgeben
```

Ausgabe:

```
Register Dump # 1
EAX = 00000037 EBX = 40155B90 ECX = 00000000 EDX = 401570C0
ESI = 40014020 EDI = BFFFF2B4 EBP = BFFFF258 ESP = BFFFF234
EIP = 08048403 FLAGS = 0202
```

Setzen und Löschen von Flags

- Das Ausführen von Instruktionen kann den Status der Flags im EFLAGS-Register verändern.
  - Im Anhang des Buches ist dies für die wichtigsten Befehle angegeben.
- Man kann insbesondere das Carry-Flag gezielt setzen und löschen:
  - `CLC` zum Löschen des Carry-Flags
  - `STC` zum Setzen des Carry-Flags
  - `CMC` zum Invertieren des Carry-Flags

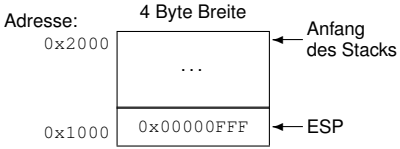
Fahrplan

Der Stack

- Bisher:
  - alle Daten entweder in einem Register
  - . . . oder in einem festen Speicherplatz
  - schwierig, wenn Funktionen realisiert werden sollen
  - . . . die auch rekursiv aufrufbar sein sollen
- Der Stack:
  - allgemeine Datenstruktur!
  - hier: Ein Speicherbereich, in dem dynamisch Daten abgelegt werden können
  - der Stack beginnt bei einer hohen Speicheradresse und wächst nach unten
  - elementare Instruktionen sind `push` (Ablegen auf Stack) und `pop` (Herunternehmen vom Stack)
  - weitere Stack-Instruktionen für die Unterstützung von Funktionen

Beispiel: elementare Stack-Instruktionen - I

- Stackanfang: Adresse 0x2000 (hier)
- Stackbreite: 4 Byte.
- ESP: zeigt auf Top of Stack



- Nächste Aktion:

```
push dword 1
```

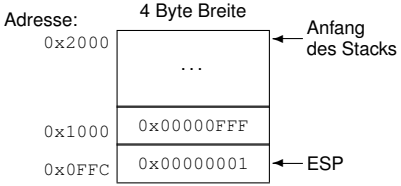
- Bedeutung: 1 als ein Doppelwort (4 Byte) auf den Stack legen

Beispiel: elementare Stack-Instruktionen - II

- Aktuelle Aktion:

  ```
  push dword 1
  ```

- ESP wird um 4 verringert.
- Dann wird in den Speicher, auf den ESP zeigt, die 1 als ein Doppelwort (4 Byte) geschrieben.



- Nächste Aktion:

  ```
  push dword 2
  ```

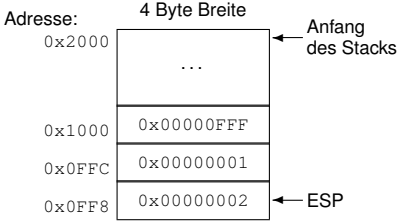
- Bedeutung: 2 als ein Doppelwort (4 Byte) auf den Stack legen

Beispiel: elementare Stack-Instruktionen - III

- Aktuelle Aktion:

```
push dword 2
```

- ESP wird um 4 verringert.
- Dann wird in den Speicher, auf den ESP zeigt, die 2 als Doppelwort (4 Byte) geschrieben



- Nächste Aktion:

```
pop eax
```

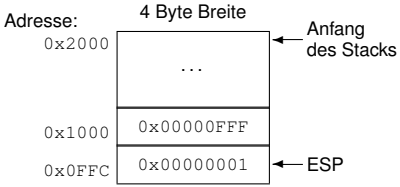
- Bedeutung: das oberste Doppelwort (4 Byte) vom Stack in das Register eax laden

Beispiel: elementare Stack-Instruktionen - IV

- Aktuelle Aktion:

  ```
  pop eax
  ```

- Der Inhalt, auf den ESP zeigt, wird in das Register eax geschrieben.
- ESP wird um 4 erhöht.



- Nächste Aktion:

  ```
  pop ebx
  ```

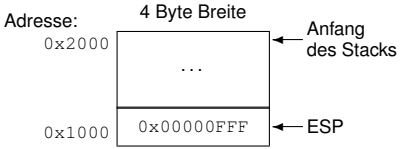
- Bedeutung: das oberste Doppelwort (4 Byte) vom Stack in das Register ebx laden

Beispiel: elementare Stack-Instruktionen - V

- Aktuelle Aktion:

  ```
  pop ebx
  ```

- Der Inhalt, auf den ESP zeigt, wird in das Register ebx geschrieben.
- ESP wird um 4 erhöht.



Vertiefungsübung

Was?	Von Bytecode zu Java und wieder zurück: hinter den Kulissen von javac
Wann?	Donnerstag, 08:30 Uhr
Wo?	2522.U1.55