64-041 Übung Rechnerstrukturen

Aufgabenblatt 12 Ausgabe: 10.01., Abgabe: 17.01. 12:00

Gruppe	
Name(n)	Matrikelnummer(n)

Aufgabe 12.1 (Punkte 5+5+5+5+5)

x86-Assembler: Angenommen, die folgenden Werte sind in den angegebenen Registern bzw. Speicheradressen gespeichert:

Register	Wert
%eax	0x00000104
%ecx	0x0000000C
%edx	0x00000004

Adresse	Wert
0x100	0x0000CAFE
0x104	0x000000CB
0x108	0x00012300
0x10C	0x00000042

Überlegen Sie sich, welche Speicheradressen bzw. Register als Ziel der folgenden Befehle ausgewählt werden und welche Resultatwerte sich aus den Befehlen ergeben:

(a) `addl %edx, (%eax)`

(b) `subl %ecx, 0xfffffffc(%eax)`

(c) `imull $32, (%eax,%edx,2)`

(d) `incl 4(%eax)`

(e) `decl %ecx`

(f) `subl %edx, %eax`

Zur Erinnerung: für den gnu-Assembler gilt

- der Zieloperand steht rechts
- Registerzugriffe werden direkt ausgedrückt
- eine runde Klammer um ein Register bedeutet einen Speicherzugriff, ggf. mit Immediate-Offset und Index: $\langle imm \rangle$`(`$\langle Rb \rangle$`,`$\langle Ri \rangle$`,`$\langle s \rangle$`)` → `MEM[`$\langle Rb \rangle$`+`$\langle s \rangle$`*`$\langle Ri \rangle$`+`$\langle imm \rangle$`]`

⇒ zum Beispiel bewirkt der Befehl: `addl %edx, 8(%eax)`
die Operation: `MEM[0x0000010C] = 0x00000046`

Sie können die Befehle natürlich gerne auch im Assembler und Debugger direkt ausprobieren. Mit einigen Befehlen lassen sich die oben angegebenen Werte in den Speicher schreiben,

und die Resultate lassen sich dann direkt ablesen. Geben Sie in diesem Fall Ihr Assemblerprogramm bitte mit ab.

Aufgabe 12.2 (Punkte 10)

Register auf Null setzen: Wie kann man den Inhalt eines Registers auf Null setzen, wenn dafür kein separater Befehl zur Verfügung steht? Geben Sie x86-Beispielcode für zwei Möglichkeiten zur Lösung dieses Problems an, der *ohne Immediate-Operand* auskommt.

Aufgabe 12.3 (Punkte 20)

Arithmetische Operationen: Eine klassische Aufgabe zur Demonstration einfacher numerischer Operationen ist die Umrechnung zwischen Grad Fahrenheit F und Grad Celsius C nach der Formel $F = 9 * C/5 + 32$.

Da im bisher eingeführten x86-Befehlssatz noch kein Befehl für die Division enthalten ist, nähern wie den Umrechnungsfaktor $9/5$ durch den Wert $9/5 \approx 459/256$ an, der sich zum Beispiel mit Multiplikation (`imull` ⟨*src*⟩`,`⟨*dest*⟩) und Rechtsschieben (`sarl` bzw. `shrl` für arithmetisches und logisches Schieben) effizient umsetzen lässt.

Schreiben Sie x86-Assemblercode für eine Funktion `int c2f(int c)`, die ihr Argument (Grad Celsius), wie in der Vorlesung erläutert, auf dem Stack übergeben bekommt und ihren Rückgabewert entsprechend der Konvention im Register `%eax` hinterlässt.

Nach Ausführung der Funktion sollen die relevanten Datenregister wieder ihren vorherigen Wert enthalten. Bedenken Sie dabei, dass laut Konvention die Register `%eax`, `%ebx`, `%ecx` und `%edx` als „Caller-Save“ klassifiziert sind. Daraus ergibt sich, dass Inhalte der für die Berechnung benötigten Register von der Funktion möglicherweise ebenfalls auf den Stack gerettet und am Ende wiederhergestellt werden müssen.

Aufgabe 12.4 (Punkte 10+10+10+10)

x86-Assembler entschlüsseln:
In manchen Anwendungen ist es notwendig, mit großen ganzen Zahlen, sog. Langzahlen, zu rechnen, für die 32 Bit nicht mehr ausreichen. Im Folgenden werden wir mit 128 Bit breiten Langzahlen arbeiten. Eine Erweiterung auf noch mehr Bits (z.B. 256 oder sogar 1024) sollte dann relativ leicht möglich sein.
Zunächst brauchen wir eine geeignete Datenstruktur für unsere 128 Bit breiten Langzahlen. Dabei bietet sich ein Array *FELD* von vier normalen Integer zu 32 Bit an, auf das wir unsere 128 Bit aufteilen. In *FELD[0]* sollen dabei die niederwertigen Bits der Langzahl stehen, in *FELD[3]* die hochwertigen. Sei konkret `0x00000001` `00000002` `00000003` `ffffffff` eine Langzahl, dann wäre die Aufteilung:

$$\begin{aligned} \text{FELD}[0] &= \texttt{0xffffffff} \\ \text{FELD}[1] &= \texttt{0x00000003} \\ \text{FELD}[2] &= \texttt{0x00000002} \\ \text{FELD}[3] &= \texttt{0x00000001} \end{aligned}$$

Mit solchen Langzahlen möchte man natürlich auch rechnen können. Wir werden beschränken uns hier auf die Addition. In einer Hochsprache wie C oder JAVA ist es nicht völlig trivial, die Summe von zwei Langzahlen zu berechnen, weil es nicht genügt, die einzelnen Felder der beiden Arrays zu addieren, sondern auch mögliche Überträge berücksichtigt werden müssen. Betrachten wir ein Beispiel: Seien *FELD1*=`0x00000000 00000000 00000001 ffffffff` und *FELD2*= `0x00000000 00000000 00000002 00000001` zwei Langzahlen. Dann würde `0xffffffff` + `0x00000001` einen Übertrag ergeben, den man eigentlich bei der Addition von `0x00000001` und `0x00000002` berücksichtigen müsste, an den man aber einer Hochsprache im Allgemeinen nicht herankommt. Die folgende Assemblerroutine *long_add(fe1, fe2, fe3)* schafft hier Abhilfe. *fe1, fe2* und *fe3* sind dabei Langzahlen, oder richtiger die Adressen dieser Arrays, in der oben angegebenen Datenstruktur. Berechnet wird *fe3= fe1 + fe2*.

```
long_add:
        pushl  %ebp
        movl   %esp,%ebp
        pushl  %edi
        movl   16(%ebp),  %edi
        movl   12(%ebp),  %edx
        movl   8(%ebp),   %ebx

        movl  (%ebx), %eax
        addl  (%edx), %eax
        movl  %eax, (%edi)

        movl  4(%ebx), %eax
        adcl  4(%edx), %eax
        movl  %eax, 4(%edi)

        movl  8(%ebx), %eax
        adcl  8(%edx), %eax
        movl  %eax, 8(%edi)

        movl  12(%ebx), %eax
        adcl  12(%edx), %eax
        movl  %eax, 12(%edi)

        popl  %edi
        movl  %ebp,%esp
        popl  %ebp
        ret
```

(a) Kommentieren Sie die einzelnen Assemblerbefehle. `adcl` ist dabei ein Befehl, der die beiden Operanden addiert. Wenn das Carry-Flag gesetzt ist, wird das Ergebnis aus der Addition noch inkrementiert (um 1 erhöht). Weiter zur Erinnerung:
C legt die Parameter beim Aufruf *von rechts nach links* auf den Stack.
Alle arithmetischen Befehle setzen die Flags, ein Befehl wie `movl` dagegen nicht.

(b) Nehmen wir an, dass beim Einsprung in das Unterprogramm das Register `%esp` den Wert `0xffffa664` hat. Wie sieht dann die Belegung des Stacks aus, bevor der Befehl `popl %edi` ausgeführt worden ist? Genauer, was liegt auf welcher Adresse. Angaben wie *Parameter fe1* genügen dabei.

Diese Assemblerroutine ist natürlich etwas unschön, weil nicht gut auf größere Wortbreiten (z.B. 1024 Bit) erweiterbar. Eine verbesserte Version ist die folgende:

```
long_add2:
    pushl %ebp
    movl  %esp,%ebp
    pushl %edi
    movl  16(%ebp),  %edi
    movl  12(%ebp),  %edx
    movl  8(%ebp),   %ebx

    clc
    movl  $4, %ecx
schleife:
    movl  (%ebx), %eax
    adcl  (%edx), %eax
    movl  %eax, (%edi)
    pushf
    addl $4, %ebx
    addl $4, %edx
    addl $4, %edi
    popf
    decl %ecx
    jnz  schleife

    popl %edi
    movl %ebp,%esp
    popl %ebp
    ret
```

(c) Kommentieren Sie auch hier die einzelnen Assemblerbefehle.
`clc` setzt ist dabei das Carry-Flag auf 0. `pushf` schreibt das Flag-Register auf den Stack, `popf` holt es wieder herunter. Gehen Sie bitte bei ihren Kommentaren darauf ein, warum diese Befehle für ein einwandfreies Funktionieren des Unterprogramms notwendig sind.

(d) Warum sollte das Unterprogramm Ihrem bisherigen Kenntnisstand nach eigentlich etwas Falsches liefern? Oder anders gefragt: Welcher der Befehle verhält sich etwas anders als vielleicht erwartet?

Bemerkung: Wer etwas spielen möchte, findet die Assembler-Routinen und ein kurzes Testprogramm auf der Web-Seite. Für die Übersetzung insbesondere auf 64-Bit-Systemen gilt das im letzten Übungsbogen Gesagte.