64-041 Übung Rechnerstrukturen WS 15/16

Aufgabenblatt 12 Ausgabe: 13.01., Abgabe: 20.01. 24:00

Gruppe	
Name(n)	Matrikelnummer(n)

Aufgabe 12.1 (Punkte 4+4+4+4+4)

`lea`-Befehl: Der Befehl `lea` ⟨*expr*⟩`,` ⟨*dest*⟩ („Load Effective Address“) ist eine Besonderheit der x86-Architektur. Der Ausdruck ⟨*expr*⟩ ist entsprechend dem indizierten Adressierungsmodus aufgebaut aus ⟨*imm*⟩`(`⟨*Rb*⟩`,`⟨*Ri*⟩`,`⟨*s*⟩`)`, siehe Vorlesungsfolien

⟨*imm*⟩	Immediate Offset
⟨*Rb*⟩	Basisregister
⟨*Ri*⟩	Indexregister
⟨*s*⟩	Skalierungsfaktor
⟨*expr*⟩	= ⟨*Rb*⟩+⟨*Ri*⟩*⟨*s*⟩+⟨*imm*⟩

Der Befehl dient eigentlich der Berechnung von Speicheradressen (und verwendet dasselbe Rechenwerk wie die Speicheradressierung), wird aber von Compilern gerne auch zur Berechnung von arithmetischen Ausdrücken verwendet. Die folgende Tabelle enthält verschiedene `leal`-Befehle, die das Resultat jeweils im Register `%edx` ablegen. Geben Sie die Formeln an, welche arithmetischen Ausdrücke den folgenden `lea`-Befehlen entsprechen.

(a) `leal 5(%eax), %edx`

(b) `leal (%ebx,%ecx), %edx`

(c) `leal 7(%eax,%eax,2), %edx`

(d) `leal 0x2c(,%eax,4), %edx`

(e) `leal 8(%eax,%ecx,2), %edx`

Aufgabe 12.2 (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 $C = (F - 32) * 5/9$.

Da im bisher eingeführten x86-Befehlssatz noch kein Befehl für die Division enthalten ist, nähern wie den Umrechnungsfaktor $5/9$ durch den Wert $5/9 \approx 142/256$ 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 f2c(int f)`, die ihr Argument (Grad Fahrenheit), 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`, `%edx` und `%ecx` als „Caller-Save“ klassifiziert sind. Daraus ergibt sich, dass Inhalte der für die Berechnung benötigten Register von der Funktion teilweise ebenfalls auf den Stack gerettet und am Ende wiederhergestellt werden müssen.

Aufgabe 12.3 (Punkte 5+5+5)

PC-relative Adressierung: Die x86-Architektur erlaubt bei Sprungbefehlen (`call`, `jmp`, `je` und Varianten) sowohl die Angabe absoluter Zieladressen, als auch die Berechnung relativ zum Wert des Programmzählers `eip`. Dabei werden die verschiedenen Möglichkeiten als separate Befehle mit unterschiedlichen Opcodes codiert.

Bei PC-relativen Sprüngen wird der Offset vorzeichenbehaftet mit 1, 2 oder 4 Bytes codiert und bezieht sich relativ zur Startadresse des nachfolgenden Befehls[1]

Überlegen Sie sich in den folgenden Beispielen die relevanten Adressen und ersetzen Sie jeweils die Platzhalter `.......` durch die passenden Werte.

(a) Was ist die Zieladresse des Befehls `jbe` („Jump if Below or Equal“) im folgenden Beispiel (Opcode `0x76` und Offset `0xda` im Zweierkomplement)

```
804002b:    76 da           jbe   .......
804002d:    eb 24           jmp   8040044
```

(b) Ergänzen Sie die Adressen

```
........:   eb 54           jmp   8050d10
........:   c7 45 f8 10 00  mov   $0x10,0xfffffff8(%ebp)
```

(c) Ergänzen Sie die Sprungadresse (4-Byte Offset, Byte-Order beachten)

```
8040002:    e9 cb 00 00 00  jmp   .......
8040007:    90              nop
```

[1] Dieses Verhalten ist darauf zurückzuführen, dass ältere x86-Prozessoren im ersten Schritt der Befehlsausführung den Wert des Registers `eip` inkrementierten.

Aufgabe 12.4 (Punkte 10+15+15+5)

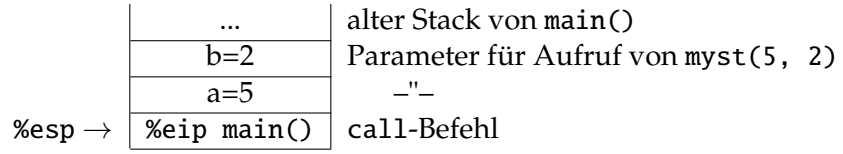
x86-Assembler entschlüsseln: Wir betrachten die rekursive Funktion `int myst (int a, int b)`, die zwei 32-bit Integer-Parameter entgegennimmt und einen 32-bit Integerwert im Register `eax` zurückliefert:

```
myst:
        pushl   %ebx
        subl    $24, %esp
        movl    32(%esp), %ebx
        movl    36(%esp), %edx
        movl    $1, %eax
        testl   %edx, %edx
        je      .L2
        subl    $1, %edx
        movl    %edx, 4(%esp)
        movl    %ebx, (%esp)
        call    myst
        imull   %ebx, %eax
.L2:
        addl    $24, %esp
        popl    %ebx
        ret
```

(a) Kommentieren Sie die einzelnen Assemblerbefehle. Zur Erinnerung: C legt die Parameter beim Aufruf *von rechts nach links* auf den Stack.

(b) Was berechnet das Unterprogramm?

(c) Wie sieht der Stack bei maximaler Verschachtelungstiefe für den Aufruf von `myst (5, 2)` aus? Ergänzen Sie die Skizze.



(d) Was würde bei einem Aufruf von `myst (3, -3)` geschehen?