64-041 Übung Rechnerstrukturen

Aufgabenblatt 12 Ausgabe: 17.01., Abgabe: 24.01. 24:00

Gruppe	
Name(n)	Matrikelnummer(n)

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

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

Register	Wert
`%eax`	`0x00000100`
`%ecx`	`0x00000001`
`%edx`	`0x0000000c`

Adresse	Wert
`0x100`	`0x0000cafe`
`0x104`	`0x000000ac`
`0x108`	`0x00000013`
`0x10c`	`0x00098700`

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

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

(b) `subl %edx, 4(%eax)`

(c) `imull $16, (%eax,%edx)`

(d) `incl 8(%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: $\langle imm \rangle (\langle Rb \rangle, \langle Ri \rangle, \langle s \rangle) \rightarrow$ `MEM[`$\langle Rb \rangle$`+`$\langle s \rangle$`*`$\langle Ri \rangle$`+`$\langle imm \rangle$`]`

$\Rightarrow$ Beispiel: Befehl `addl %ecx, 12(%eax)`
Operation `MEM[0x0000010c] := MEM[0x0000010c]+1 = 0x00098701`

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 10+10+10 [+ 5 Bonus])

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)

```
804001c:   76 da           jbe  .......
804001e:   eb 24           jmp  8040044
```

(b) Ergänzen Sie die Adressen

```
.......:   eb 54           jmp  8050d42
.......:   c7 45 f8 10 00  mov  $0x10,0xfffffff8(%ebp)
```

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

```
8040000:   e9 cb 00 00 00  jmp  .......
8040005:   90              nop
```

[d] Fällt Ihnen bei Betrachtung der vorigen beiden Aufgabenteile (b) und (c) etwas auf?

[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+5+5 [+10 Bonus])

x86-Assembler entschlüsseln: Gegeben sei folgende Assembler-Routine `myst (int* Feld1, int Z, int* Feld2)`, die drei Argumente hat: einen 32-bit IntegerZ und zwei Zeiger auf Arrays `Feld1` und `Feld2`.

```
#  myst(int* Feld1,  int Z, int*  Feld2)
#  Feld1, Feld2  sind dabei die Adressen von zwei Integer-Arrays

myst:
    pushl    %ebp
    movl     %esp,%ebp
    movl     16(%ebp), %edx
    movl     8(%ebp), %ebx
    movl     $0, %ecx
lo1:
    cmpl     %ecx, 12(%ebp)
    je       done
    movl     (%ebx), %eax
    movl     %eax, (%edx)
    addl     $4, %edx
    addl     $4, %ebx
    incl     %ecx
    jmp      lo1
done:
    movl     %ebp,%esp
    popl     %ebp
    ret
```

(a) Kommentieren Sie die einzelnen Assemblerbefehle. Dabei soll beschrieben werden *was im Algorithmus passiert* (z.B.: „Argument Z wird nach …kopiert“) und *nicht wie der Assemblercode das macht* (z.B.: „move esp nach epb“)! Zur Erinnerung: C legt die Parameter beim Aufruf von rechts nach links auf den Stack.

(b) Was tut das Unterprogramm?

(c) Was würde bei einem Aufruf von `myst(&fe1, -1, &fe2)` geschehen?

[d] Geben Sie, sofern möglich, die Speicherinhalte des Stacks an, nachdem das Programm den Befehl `movl $0, %ecx` ausgeführt hat. Das Register `%esp` enthält dabei den Wert `0xffffffa0`. Angaben zur Funktion, wie beispielsweise „Parameter Z“, genügen hier.

Adresse	Inhalt
0xffffffb4	
0xffffffb0	
0xffffffac	
0xffffffa8	
0xffffffa4	
0xffffffa0	
0xffffff9c	
0xffffff98	
0xffffff94	