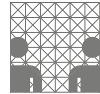
64-041 Übung Rechnerstrukturen



Aufgabenblatt 11 Ausgabe: 20.01., Abgabe: 27.01. 12:00

Gruppe	
Name(n)	Matrikelnummer(n)

Aufgabe 11.1 (Punkte 5+5+5+5+5+5)

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

Register	Wert
`%eax`	`0x00000100`
`%ecx`	`0x00000002`
`%edx`	`0x0000000C`

Adresse	Wert
`0x100`	`0x0000ABBA`
`0x104`	`0x000000DC`
`0x108`	`0x000000EF`
`0x10C`	`0x00054321`

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 auf die entsprechende Adresse, ggf. mit Byte-Offset vor der Klammer

⇒ zum Beispiel bewirkt der Befehl: `addl %ecx, 12(%eax)`
die Operation: `MEM[0x0000010C] = 0x00054323`

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 11.2 (Punkte 5)

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 an, der ohne Immediate-Operand auskommt.

Aufgabe 11.3 (Punkte 10)

Programmzähler auslesen: Es gibt keinen x86-Assemblerbefehl, der es erlaubt, den Programmzähler `%eip` direkt auszulesen. Schreiben Sie ein kurzes Assemblerprogramm, das den Programmzähler in das Register `%eax` kopiert. Hinweis: Sie dürfen den Stack zur Zwischenspeicherung verwenden.

Aufgabe 11.4 (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 $\mathsf{F} = (9/5 * \mathsf{C}) + 32$.

Da im bisher eingeführten x86-Befehlssatz noch kein Befehl für die Division enthalten ist, nähern wie den Umrechnungsfaktor $9/5$ durch den Wert $9/5 \approx 461/256$ 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 fahrenheit (int celsius)`, 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`, `%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 11.5 (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]

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

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
```

Aufgabe 11.6 (Punkte 20 [+10 Bonus])

Rekursion: Analysieren Sie das folgende in C-Syntax notierte Programm und geben Sie einen mathematischen Ausdruck für das Ergebnis der Funktion an $\mathrm{myst}(a, b) = ..., a, b \geqslant 0$

```
int myst (int a, int b)
{ if (a == 0)
        return 1;
  else  return b * myst(a-1, b);
}
```

(a) Schreiben Sie ein rekursives x86-Assemblerprogramm, das die Funktion des C-Codes berechnet. Die beiden Parameter werden durch die rufende Prozedur/Funktion auf dem Stack abgelegt. Der Resultatwert soll gemäß Konvention im Register `%eax` gespeichert werden.

[b] Skizzieren sie den Stack bei maximaler Verschachtelungstiefe für den Aufruf von `myst` mit den Parametern a=3 und b=6 an.