64-041 Übung Rechnerstrukturen

Aufgabenblatt 12 Ausgabe: 18.01., Abgabe: 25.01. 24:00

Gruppe	
Name(n)	Matrikelnummer(n)

Aufgabe 12.1 (Punkte 20)

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

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 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`, `%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 12.2 (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)

```
804002a:   76 da            jbe  .......
804002c:   eb 24            jmp  8040052
```

(b) Ergänzen Sie die Adressen

```
.......:   eb 54            jmp  8050d20
.......:   c7 45 f8 10 00   mov  $0x10,0xfffffff8(%ebp)
```

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

```
804000a:   e9 cb 00 00 00   jmp  .......
804000f:   90               nop
```

Aufgabe 12.3 (Punkte 5+5+20+4·5)

x86-Assembler entschlüsseln: In manchen Kryptologie-Verfahren ist es notwendig, Ausdrücke der Form $a^N \ mod \ m$ mit natürlichen Zahlen a, N und m möglichst effizient zu berechnen. Dabei können und sollten für eine ausreichend hohe Sicherheit alle diese Zahlen 512 Bit und mehr haben. Wir begnügen uns hier mit viel kleineren Zahlen, die sich noch mit der normalen Rechnerarithmetik bearbeiten lassen. Und zwar wählen wir willkürlich:
$a = 10403 \quad N = 500\,000\,002 \quad m = 11119$

(a) Betrachten wir zunächst eine völlig unbrauchbare Methode zur Berechnung des obigen Ausdrucks indem wir erst a^N berechnen und dann das Ergebnis modulo m nehmen.

Wieviele Dezimalstellen hätte a^N für die oben angegeben Zahlen? Wieviele Stellen hätte diese Zahl im Dualsystem?[2]

(b) Die Größe des obigen Zwischenergebnisses überzeugt schnell davon, dass man anders vorgehen muss. Naheliegend wäre es nach jeder Multiplikation „Modulo m“ zu rechnen, was die Zwischenergebnisse klein hält.

```
x = 1;
for (i = 1; i <= N; i++)
    x = (x*a) % m;
```

Dass das zum richtigen Ergebnis führt, sollte aus der Mathematik heraus klar sein. Laden Sie sich die Dateien `aufg12_pot.c` und `aufg12_pot2.s` herunter. Übersetzen Sie jetzt das Programm, das obigen Algorithmus und noch einen anderen verwendet, den wir uns weiter unten genauer ansehen werden. Nutzen Sie dazu beispielsweise den C-Compiler aus Aufgabenblatt 10, mit der Kommandozeile[3] `gcc -m32 -o pot.exe aufg12_pot.c aufg12_pot2.s` und starten Sie das Programm.

Welche Zeit benötigt der oben angegebene Algorithmus auf Ihrem Rechner zur Berechnung von $a^N \ mod \ m$ mit den oben angegeben Zahlen?

[2] Hinweis: Die Logarithmus-Funktion dürfte zur Beantwortung dieser Fragen hilfreich sein.
[3] Wichtig: der vorgegebene Assemblercode setzt eine 32-bit Umgebung voraus.

(c) Der zweite in dem Programm verwendete Algorithmus ist offenbar, was die Ausführungszeit angeht, um Größenordnungen besser. Der entsprechende C-Code dazu ist in `aufg12_pot.c` auskommentiert und in `aufg12_pot2.s` noch einmal etwas anders implementiert. Kommentieren Sie die einzelnen Befehle dieses Assembler-Codes. Zur Erinnerung: C legt die Parameter von rechts nach links auf den Stack und gibt das Ergebnis einer Funktion in `%eax` zurück.

```
#-----------------------------------------------------
# int pot2(int a, int expo, int m)
#-----------------------------------------------------
pot2:
        pushl   %ebp
        movl    %esp, %ebp
        pushl   %ebx
        pushl   %ecx
        movl    $1, %ebx
        movl    8(%ebp), %ecx
loop1:
        cmpl    $0, 12(%ebp)
        je      ende
        movl    12(%ebp), %eax
        andl    $1, %eax
        je      weiter
        movl    %ebx, %eax
        subl    %edx, %edx
        imull   %ecx, %eax
        idivl   16(%ebp)
        movl    %edx, %ebx
weiter:
        movl    %ecx, %eax
        subl    %edx, %edx
        imull   %ecx, %eax
        idivl   16(%ebp)
        movl    %edx, %ecx
        shrl    12(%ebp)
        jmp     loop1
ende:
        movl    %ebx, %eax
        popl    %ecx
        popl    %ebx
        movl    %ebp, %esp
        popl    %ebp
        ret
```

`imull` erwartet bei der Multiplikation von zwei positiven 32-bit Werten, dass eine Zahl in `%eax` steht und in `%edx` eine Null. Das Ergebnis ist ein 64-bit Wert in den Registern `(%edx,%eax)`.

`idivl` dividiert eine 64-bit Zahl aus `(%edx,%eax)` durch den angegeben Wert und liefert das Ergebnis in Register `%eax` und den Divisionsrest in `%edx`.

(d) Multiplikationen und Division sind auch auf modernen Prozessoren immer noch recht zeitaufwendig. Wir vermuten daher, dass die Anzahl der Multiplikationen/Divisionen unsere beiden Algorithmen entscheidend beeinflusst.

Wieviele Multiplikationen/Division benötigt das erste implementierte Verfahren, um den Ausdruck für eine gegebene Zahl N zu berechnen?

Beim zweiten Algorithmus lässt sich die genaue Zahl der Multiplikationen/Divisionen für ein gegebenes N nicht so einfach angeben, wohl aber eine obere Schranke. Bestimmen Sie eine solche Schranke (je kleiner, desto besser) für die Implementation im Assembler-Code. Die auskommentierte Implementation in C ist aber ähnlich schnell, auch wenn sie mehr Multiplikationen/Divisionen benötigt.

Nehmen wir an, für ein gegebenes N finden Sie für den ersten Algorithmus eine Rechenzeit von 1 Sekunde. Welche Rechenzeit erwarten Sie dann für N^2?

Wie lautet die Antwort für den zweiten Algorithmus, wenn Sie mit ihm für ein gegebenes N ebenfalls eine Rechenzeit von 1 Sekunde gebraucht haben? Hier lässt sich natürlich kein genauer Wert angeben, sondern nur die Größenordnung der Rechenzeit für N^2.

Aufgabe 12.4 (Punkte 10+5)

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

```
1  myst:
2          pushl   %ebx
3          subl    $24, %esp
4          movl    32(%esp), %ebx
5          movl    36(%esp), %edx
6          movl    $1, %eax
7          testl   %edx, %edx
8          je      .L2
9          subl    $1, %edx
10         movl    %edx, 4(%esp)
11         movl    %ebx, (%esp)
12         call    myst
13         imull   %ebx, %eax
14 .L2:
15         addl    $24, %esp
16         popl    %ebx
17         ret
```

(a) Kommentieren Sie die einzelnen Assemblerbefehle.

(b) Was berechnet das Unterprogramm?