64-041 Übung Rechnerstrukturen

Aufgabenblatt 12 Ausgabe: 14.01., Abgabe: 21.01. 24:00

Gruppe	
Name(n)	Matrikelnummer(n)

Aufgabe 12.1 (Punkte 9*4)

Adressierungsarten: Kreuzen Sie bitte an, welche der unten angebenen Adressierungsarten Sie den folgenden Befehle zuordnen würden. Es ist auch mehr als ein Kreuz pro Befehl möglich. Konzentrieren Sie sich dabei mehr auf die beteiligten Datentransfers als auf die Codierung der Befehle.

Bemerkung: Die Befehle **call** und **jmp** sind im x86-Befehlsformat so codiert, dass nicht die absolute Adresse, sondern der Offset zum aktuellen Stand des Programmzählers im Befehlswort steht.

Befehl	unmittelbar	Register	Direkt	Indirekt	Indiziert	PC+Offset
`movl %esp, %ebp`						
`movl 8(%ebp), %edx`						
`addl $1, %edx`						
`pushl %edx`						
`jmp ende`						
`call fu1`						
`popl %ebp`						
`ret`						
`leal 4(%ebx,%eax,2), %edx`						

Aufgabe 12.2 (Punkte 7*5)

Flags: Fast alle Prozessoren haben ein Carry-Flag und ein Overflow-Flag. Zur Erinnerung:
Das Carry-Flag wird bei einer arithmetischen Operation gesetzt, wenn sich in der höchstwertigsten Stelle ein Übertrag ergibt.
Das Overflow-Flag wird gesetzt, wenn das Ergebnis der Operation das "falsche" Vorzeichen hat. Z.B., wenn die Addition zweier positiver Zahlen ein negatives Ergebnis liefert.

Kreuzen Sie an, welches Flag bei Ausführung der folgenden Operationen gesetzt werden würde.

Operation	Inhalt %eax	Inhalt %ebx	CF	OF
`addl %eax, %ebx`	`0x00000001`	`0x00000002`		
`addl %eax, %ebx`	`0x00000001`	`0xFFFFFFFF`		
`addl %eax, %ebx`	`0x00000002`	`0x7FFFFFFF`		
`addl %eax, %ebx`	`0x80000000`	`0xFFFFFFFE`		
`subl %eax, %ebx`	`0x00000002`	`0x00000001`		
`subl %eax, %ebx`	`0x00000001`	`0x00000002`		
`subl %eax, %ebx`	`0x00000001`	`0x80000000`		

Aufgabe 12.3 (Punkte 20 + 9)

Multiplikation: Der Grundgedanke des schnellen Potenzier-Algorithmus aus Aufgabe 11.3 eignet sich auch hervorragend, um eine Multiplikation (nicht negativer) Zahlen zu implementieren. Und wozu braucht man das, wenn doch die x86-Prozessoren eigene Befehle dafür haben, die mit Sicherheit schneller sind als jede Assembler-Routine? Nur gibt es eben nicht nur diese großen Prozessoren, sondern auch kleinere Mikroprozessoren oder Microcontroler, wie sie in größerer Zahl z.B. in jedem Auto für Steuerungsaufgaben oder Sensorik im Einsatz sind und nicht unbedingt Multiplikationsbefehle verfügen, so dass man sich eventuell selbst eine Routine in der entsprechenden Assembler-Sprache schreiben muss. Sehen wir uns einen entsprechenden Algorithmus in C (oder JAVA) an:

```
// Berechnet  ergebnis= a*b
ergebnis= 0;
vielfache= a;
while (b != 0)
   { if ((b % 2) == 1) ergebnis= ergebnis + vielfache;
     vielfache= 2*vielfache;
     b= b / 2;
   }
```

Die Ähnlichkeit mit dem Algorithmus aus Aufgabe 11.3 dürfte klar sein.

(a) Schreiben Sie jetzt ein Unterprogramm

```
int  muller(int a, int b)
```

in x86-Assembler, das diesen Algorithmus implementiert. Ein primitives, verbesserungfähiges Hauptprogramm in C, mit dem Sie Ihr Unterprogramm austesten können, können Sie sich herunterladen. Denken Sie bitte daran, dass im Assembler-Unterprogramm weder Multiplikationen noch Divisionen erlaubt sind, denn die würde es auf dem Mikrocontroler/Mikroprozessor auch nicht geben. Weiterhin sollte Ihr Unterprogramm keine auf dem Stack liegenden lokalen Variablen verwenden. Geben Sie die Datei mit Ihrem (kommentierten!) Unterprogramm bitte als Lösung der Aufgabe mit ab.

Hinweis: Wie man das Hauptprogramm und das Assembler-Unterprogramm übersetzt, sollte aus Aufgabe 11.3 klar sein. Im C-Hauptprogramm müssen Sie auf jeden Fall die Routine *muller* auskommentieren, und dieses Symbol in der Assembler-Datei als *.globl* erklären, damit der Linker richtig arbeiten kann (siehe die Datei poti2.s aus Aufgabe 11.3).

(b) Oben wurde gesagt, dass die so implementierte Multiplikation nur für nicht negative Zahlen funktioniert. Das stimmt nicht ganz. In der oben angegebenen Routine darf a sehr wohl negativ sein. Welches Problem tritt aber auf, wenn b negativ ist? Wie müsste man die Routine in der C-Version umschreiben, damit sie in jedem Fall funktioniert? In der Assembler-Version tritt dieses Problem (hoffentlich) sowieso nicht auf, aber probieren Sie das bitte aus und bringen Sie gegebenfalls die notwendigen Korrekturen an.