Thema, Ziele: Performancebetrachtungen in C++

Allgemeine Bemerkungen
Die Performancebetrachtungen sind stark von der Qualität des Compilers abhängig. Die aktuelle Version des GNU-Compilers (Version 4) erzeugt sehr effizienten Code. Die Untersuchungen in diesem Praktikum sollen mit der Version 3 des GNU-Compilers durchgeführt werden. Die Version 3 sollte normalerweise mittels `g++-3` aufgerufen werden können. Sie finden diese Version auch im Ordner `./bin`.

Die folgenden Optionen der GNU-Compiler könnten nützlich sein:

`-E`	Precompile only, der Output wird auf stdout geschrieben
`-S`	Assembleroutput, ohne Objectfile erzeugen
`-c`	nur compilieren
`-O0`	keine Optimierung
`-O1`	Optimierungsstufe 1 (siehe g++ - Help für Details)
`-O2`	Optimierungsstufe 2
`-O3`	Optimierungsstufe 3
`-Os`	Optimierung auf Codegrösse

Hinweise zum x86-Instruktionssatz finden Sie unter anderem auf folgenden Websites:

http://en.wikipedia.org/wiki/X86_instruction_listings

http://www.cs.uaf.edu/2005/fall/cs301/support/x86/index.html

http://ref.x86asm.net/

Aufgabe 1: Pointer auf lokale Variable

In dieser Aufgabe soll untersucht werden, wie sich der Assemblercode ändert, wenn ein Pointer auf eine lokale Variable definiert wird. Achten Sie darauf, dass Ihre Variablen nicht wegoptimiert werden, weil sie nicht verwendet werden. Die einfachste Variante, dies zu verhindern ist, den Wert in die Console zu schreiben, am besten mittels `printf()` aus der Bibliothek `<cstdio>`.

a) Nehmen Sie das folgende C++-Programm:

```
int main(void)
{
  int number = 34;
  return 0;
}
```

Compilieren Sie dieses Programm mit g++ mit unterschiedlichen Optimierungsstufen. Betrachten Sie jeweils das entstandene Assemblerfile. Achten Sie darauf, dass die Variable nicht wegoptimiert wird (Hinweise siehe oben). Wird die Variable `number` in ein Register oder auf den Stack gelegt?

b) Definieren Sie nun einen Pointer auf die Variable number. Wo kommen nun der Pointer und die int-Variable zu liegen?

Aufgabe 2: Pointer vs. Referenzen

Referenzen erlauben in C++ die Verwendung eines Alias auf ein Objekt, wobei die mühsame Pointernotation vermieden werden kann. Wie sieht der Assemblercode aus? Werden Pointer und Referenzen unterschiedlich übersetzt?

a) Nehmen Sie den folgenden C++-Code:

```
#include <cstdio>
struct A
{
  char c;
  int i;
};

int main(void)
{
  A myA = {'A', 34};
  printf("%c\n%d\n", myA.c, myA.i);

  return 0;
}
```

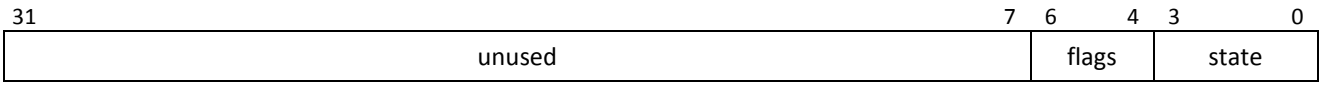
Kommt die Variable `myA` auf den Stack oder in Register zu liegen? Beachten Sie, dass einzelne Werte nur deshalb auf den Stack müssen, weil sie mit `printf()` ausgegeben werden.

b) Definieren Sie nun einen Pointer auf `myA` und untersuchen Sie, wie dies umgesetzt wird. Ist nun `myA` auf dem Stack oder in Registern?

c) Definieren Sie eine Referenz statt ein Pointer auf `myA` und untersuchen Sie die Unterschiede zur Umsetzung eines Pointers.

Aufgabe 3: Bitfelder

Gegeben ist das nachfolgende Register



a) Definieren Sie dieses Register mit Hilfe eines Bitfelds. Die Variable (z.B. r) müssen Sie mit `volatile` kennzeichnen. Wieso?

b) Setzen Sie nun `state` auf den Wert `15` und `flags` auf den Wert `3`. Untersuchen Sie, wie der Assemblercode aussieht.

c) Inkrementieren Sie nun `flags` mittels `++r.flags`. Achten Sie nun auf den Code.

d) Führen Sie mit `flags` eine übliche Bitoperation durch, z.B. `r.flags &= 6`. Achten Sie nun auf den Code.

e) Diskutieren Sie die erhaltenen Ergebnisse. Entspricht das Resultat Ihren Erwartungen? Worüber sind Sie überrascht? Sollen Bitfelder so verwendet werden? Sollen sie überhaupt verwendet werden?

Aufgabe 4: Bitmasken

a) Lösen Sie die Aufgabe 3 mit allen Operationen ausser der Addition direkt mittels Operationen mit Bitmasken. Vergleichen Sie nun den Assemblercode mit dem Code aus Aufgabe 3.

b) Welche Erkenntnisse haben Sie aus den Resultaten der Aufgaben 3 und 4 gewonnen?