Thema, Ziele: Performancebetrachtungen in C++

Aufgabe 1: Pointer auf lokale Variable

a) siehe `./Loesung/A1-1`

Das Programm wurde mit Optimierungsstufe 2 übersetzt. Die Variable wird im Register `eax` gespeichert. Der Move auf den Stack (Stackpointer `esp`) wird benötigt für den Aufruf der `printf()` Funktion.

```
movl   $34, %eax
movl   %eax, 4(%esp)
call   _printf
```

b) siehe `./Loesung/A1-2`

Jetzt wird die Variable auf den Stack gelegt.

```
movl   $34, -4(%ebp)
leal   -4(%ebp), %eax
movl   %eax, 4(%esp)
```

Aufgabe 2: Pointer vs. Referenzen

a) siehe `./Loesung/A2-2`

```
#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;
}
```

Die Variable `myA` wird in zwei Register gelegt, in `eax` und `edx`.

b) siehe `./Loesung/A2-2`

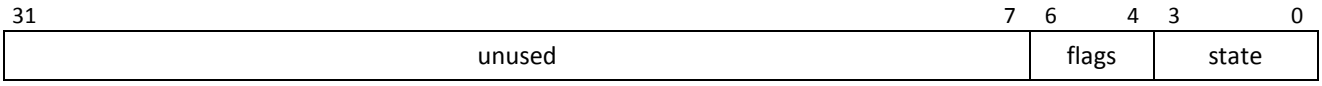
Bei der Verwendung eines Pointers werden die Werte auf den Stack gelegt. Für die schnelleren Zugriffe werden die Werte gleichzeitig auch in Register gespeichert.

c) siehe `./Loesung/A2-3`

Bei der Verwendung einer Referenz wird derselbe Code erzeugt wie bei einem Pointer.

Aufgabe 3: Bitfelder

Gegeben ist das nachfolgende Register



a) siehe `./Loesung/A3`

`volatile` muss verwendet werden, damit das Register keinesfalls wegoptimiert wird. Zudem können die Werte von Registern ebenfalls durch die Hardware geändert werden. Deshalb muss der Code so generiert werden, dass das Register jedesmal zuerst gelesen wird bevor damit gearbeitet wird. Wenn dies nicht gemacht würde, könnte passieren, dass mit einer Kopie des Registerinhalts gearbeitet wird, der in der Zwischenzeit durch die Hardware geändert wurde.

b) siehe `./Loesung/A3`

Die Registervariable wird auf `-4(%ebp)` gelegt, d.h. auf den Stack. Aus dem Zugriff auf `state` wird direkt eine OR-Operation (`orl`). Da die Variable `volatile` ist, werden die Werte laufend zwischen dem Stack und dem Register `eax` hin- und hergeschoben. Der Wert auf dem Stack muss immer der richtige und aktuelle sein.

```
movl    $0, -4(%ebp)
movl    -4(%ebp), %eax
...
orl     $15, %eax
```

Das direkte Setzen von `flags` wird ebenfalls in eine OR-Operation umgewandelt.

```
orl     $48, %eax
```

c) siehe `./Loesung/A3`

Beim Inkrementieren von `flags` wird mit Schiebeoperationen (zuerst nach rechts, dann nach links) gearbeitet. Dieser Zugriff wird sehr ineffizient.

```
shrb    $4, %al
andl    $-113, %edx
andl    $7, %eax
incl    %eax
andl    $7, %eax
sall    $4, %eax

orl     %eax, %edx
```

d) siehe `./Loesung/A3`

Auch die Operation `r.flags &= 6` wird sehr ineffizient durchgeführt, da das Muster zuerst an Bitposition `0` geschoben wird, dann wird die Operation durchgeführt und abschliessend wieder zurückgeschoben.

e) Bitfelder sind bekanntlich nicht standardisiert. Wenn die einzelnen Felder direkt gesetzt werden, dann erzeugt der GNU-Compiler effizienten Code, er nimmt direkt eine Bitoperation. Wenn hingegen Operationen durchgeführt werden wie `r.flags &= 6`, dann wird sehr ineffizienter Code generiert. Die meisten Compiler können das auch nicht besser. Bitfelder sollten deshalb aus meiner Sicht nicht verwendet werden, da nebst der nicht vorhandenen Portabilität zudem sehr ineffizienter Code entsteht.

Aufgabe 4: Bitmasken

a) siehe `./Loesung/A4`

Bei dieser Variante entsteht sehr effizienter Code. Eine Anweisung wie `r &= 6 << 4;` wird direkt in ein `AND` umgewandelt, die Operation **`6 << 4`** berechnet der Compiler, nicht das Laufzeitsystem.

b) Bitmasken verwenden, Bitfelder nicht.