Technische Universität München
Institut für Informatik
Dr. rer. nat. Tobias Weinzierl
Kaveh Rahnema

März 2013

Kompaktkurs Einführung in die Programmierung

Übungsblatt 1: Einführung

`http://www5.in.tum.de/` → „Teaching“ → „Summer 13“

Hinweise zu den Übungsblättern

Alle Aufgaben sind mit einem Symbol folgender Bedeutung gekennzeichnet:

- (T) : Theoretische Aufgaben die mit Stift und Papier gelöst werden können.
- (P) : Praktische Aufgaben die mit Hilfe eines Rechners implementiert werden sollen.
- (★) : Zusatzaufgaben für Fortgeschrittene. Sie stellen **KEINEN** Klausurrelevant Stoff dar! Auf jedem ungeraden Übungsblatt (3,5,7,9,11) wird es eine Programmieraufgabe zur Erstellung eines parallelen Programmes geben. Zur Parallelisierung wird die *Open Multi-Processing* (OpenMP) Schnittstelle benutzt. Auf jedem geraden Übungsblatt (4,6,8,10) wird es serielle Zusatzaufgaben für Fortgeschrittene geben.

Lernziele dieses Übungsblattes

- Übersetzen und Ausführen von Programmen.
- Ausgabe von Ergebnissen/Zeichen auf Konsole.
- Funktionsweise astrakter Maschinen (Low-Level Assembler).

1) Übersetzen und Ausführen von Programmen (P)

Als Beispielprogramm dient das Programm `prog0.cpp` welches Sie auf der Website zum Kurs finden.

Anleitung für Sunhalle: Vorausgesetzt wird, dass man einen Webbrowser offen hat sowie ein Terminalfenster, um folgende Kommandos eingeben zu können (am Anfang befinden Sie sich in Ihrem Homeverzeichnis).

- Ein neues Verzeichnis anlegen: `mkdir EiPro`
- in dieses Verzeichnis wechseln: `cd EiPro`
- Mittels des Browsers das Programm `prog0.cpp` in das neue Verzeichnis speichern
- Im Terminalfenster: Inhalt des Verzeichnisses anzeigen lassen: `ls` (sollte `prog0.cpp` ausgeben)
- Zum Übersetzen und Linken gibt es nun zwei Möglichkeiten:
  - Explizit: übersetzen mit `g++ -c prog0.cpp`, anschließend im Terminalfenster `ls` (sollte nun auch den Objektcode `prog0.o` ausgeben), abschließendes Linken des Objektcodes über `g++ prog0.o`
  - Implizit: direktes Übersetzen und Linken mit dem Befehl `g++ prog0.cpp`

- Inhalt des Verzeichnisses anzeigen lassen: `ls` (sollte zumindest `a.out` und `prog0.cpp` ausgeben)
- Programm ausführen: `./a.out` (sollte a: 37, b: 41, c: -4, d : 6, e: 156 ausgeben)
- Programm bearbeiten: `nedit prog0.cpp` (z.B. aus 37 eine andere Zahl machen, abspeichern, übersetzen, ausführen,...)

Auf heimischem Rechner: Kurze Hinweise für die Benutzer von Windows und Dev-C++.

- Das Programm kann man mit Dev-C++ öffnen (File → Open Project or File), bearbeiten und übersetzen (Execute → Compile).
- Das ausführbare Programm heißt hier in der Voreinstellung nicht `a.out`, sondern so wie der Quelltext, also `prog0.exe`.
- Zum Ausführen ist ein Terminalfenster (in Windows: Eingabeaufforderung) hilfreich. Das Navigieren in das richtige Verzeichnis mittels `cd` kann hier etwas komplizierter sein, je nachdem, in welchem Verzeichnis man am Anfang landet. Den Inhalt eines Verzeichnisses sieht man mit `dir`. Wenn man im richtigen Verzeichnis ist, wird das Programm einfach mittels `prog0` ausgeführt.
- Wenn man das Programm innerhalb von Dev-C++ ausführt, verschwindet die Ausgabe, bevor man sie lesen kann. Um dies zu umgehen kann man im Programm vor der schließenden geschweiften Klammer noch eine Zeile

  ```
  std::cin.get();
  ```

  einfügen. Dann wartet das Programm an dieser Stelle, bis man die Enter-Taste drückt und man kann das Programm auch innerhalb von Dev-C++ ausführen (Execute → Run) und beobachten. Dieses syntaktisch komplexe Konstrukt wird in späteren Blättern und Vorlesungen genauer erklärt.

2) Hello world! (P)

Schreiben Sie ein eigenes erstes Programm `hello.cpp` welches

```
Hello world!
```

auf der Konsole (Bildschirm) ausgibt. Sie können sich dazu am Programm `prog0.cpp` orientieren.

3) Durchschnittsrechner (P)

Diese Aufgabe dreht sich um die Implementierung eines einfachen Durchschnittsrechners, welcher zwei Werte einliest und den Durchschnitt der beiden Variablen ausgibt. Da Sie bereits mit der Ausgabe von Daten an die Konsole vertraut sind, setzen wir nun mit dem Einlesen von Daten mittels std::cin fort. Wenn Sie eine einzelne Zahl von der Konsole lesen möchten, können Sie std::cin verwenden, welches durch das Einbinden von iostream verfügbar wird.

std::cin konvertiert die textuelle Benutzereingabe automatisch zu dem Typ, der in der Zielvariable (im obigen Beispiel “double“ von foo) festgelegt ist. Ihre Aufgabe ist es nun, ein Programm “averageCalculator.cpp“ zu schreiben, welches zwei Variablen mittels std::cin einliest und den Durchschnitt der beiden Werte ausgibt. Beide Werte sollen vom Typ “double“ sein.

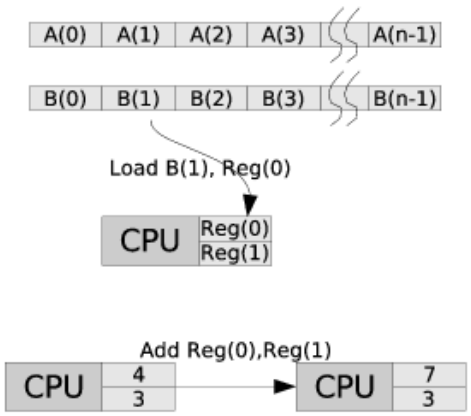
```
double foo;
std::cin >> foo;
```

Lösung:

```
#include <iostream>
int main () {
double a, b;
std::cout << "Please enter two double values: " << std::endl;
std::cout << "a = ";
std::cin >> a;
std::cout << "b = ";
std::cin >> b;
std::cout << "Average is: " << (a+b)/2 << std::endl;
return 0;
}
```

4) Low-level Assembler (T)

Gegeben sei ein hypothetischer Rechner mit einer CPU von 2 Registern, die pro Taktzyklus eine der Operationen `load,store,add,mult` abarbeiten kann. Im Hauptspeicher können 2 Arrays (Felder / Vektoren) A und B der Länge n gehalten werden. Siehe folgende Abbildung:



i) **Angenommen, im Speicher liegen die Arrays $A, B \in \mathbb{R}^4$, $n = 4$. Schreiben Sie ein Programm in low-level Assembler, welches die Summe $A + B$ berechnet und das Resultat in A speichert.**

```
for i = 0...3
    load       A(i),Reg(0)
    load       B(i),Reg(1)
    add
    store      Reg(0),A(i)
```

ii) **Schreiben Sie nun ein Programm welches die 2-norm des Arrays A ($\|A\|_2$) berechnet und den Wert an erster Position von A, also in `A(0)`, speichert. Sie können davon ausgehen, dass Ihnen die Assembler Funktion `sqrt Reg(i)` zur Verfügung steht, die**

die Wurzel der Zahl im i-ten Register bildet.

```
for i = 0...3
    load       A(i),Reg(0)
    load       A(i),Reg(1)
    mult
    store      Reg(0),A(i)
load    A(0),Reg(0)
for i = 1...3
    load       A(i),Reg(1)
    add
sqrt     Reg(0)
store    Reg(0),A(0)
```

5) Speedup durch Pipelining (★)

Mit Hilfe des Rechners aus Aufgabe 2) (siehe Abbildung) berechnet folgender Programmcode $A(2) = A(3) + 2B(5)$:

```
load B(5),Reg(0)
load    2,Reg(1)
mult
load A(3),Reg(1)
add
store Reg(0),A(2)
```

i) **Angenommen, im Speicher liegen die Arrays $A, B \in \mathbb{R}^6$, $n = 6$. Schreiben Sie ein Programm, das $2A + B$ berechnet und das Resultat in A speichert. Wieviele Taktzyklen benötigt Ihr Programm?**

```
for i = 0...5
    load       A(i),Reg(0)
    load       2,Reg(1)
    mult
    load       B(i),Reg(1)
    add
    store      Reg(0),A(i)
```

Das Programm benötigt** 36 **Taktzyklen benötigt.

ii) **Der CPU wird ein zweites Registerpaar ($Reg(2)$ und $Reg(3)$) hinzugefügt. Desweiteren existiere eine Operation `add2` die "Addiere $Reg(0)$ und $Reg(1)$" und "Addiere $Reg(2)$ und $Reg(3)$" in einem Taktzyklus berechnet. Die Resultatwerte werden in $Reg(0)$ und $Reg(2)$ gespeichert. Schreiben Sie den Assembler code mit Hilfe der Operation `add2` um und vergleichen Sie die Anzahl an Taktzyklen die jetzt benötigt werden.**
Programm Sequenz für die ersten beiden Iterationen:

```
load   A(0),Reg(0)
load   A(1),Reg(2)

load   2,Reg(1)
load   2,Reg(3)

mult   Reg(0),Reg(1)
mult   Reg(2),Reg(3)

load   B(0),Reg(1)
load   B(1),Reg(3)

add    Reg(0),Reg(1)
add    Reg(2),Reg(3)

store  Reg(0),A(0)
store  Reg(2),A(1)
```

> *ersetzt durch* `add2`

Auf diese Weise werden mit Hilfe der Operation `add2` ***nur*** 33 ***Zyklen anstatt*** 36 ***benötigt. Somit werden pro Iteration*** $\frac{1}{2}$ ***Zyklen gewonnen. Eine weitere Verbesserung könnte die Verwendung einer zusätzlichen Operation*** `mult2` ***sein. Nach Flynn'scher Klassifikation ist der Typ der zugrundeliegenden Architektur SIMD (Single Instruction Multiple Data).***

iii) Dem Rechner wird eine zweite CPU mit 2 Registern hinzugefügt. Schreiben Sie das low-level Assembler Programm unter dieser Prämisse neu.
Programm aus Teilaufgabe i) kann in 2 unabhängige Schleifen gespalten werden.

```
for i = 0...2                 |    for i = 3...5
    siehe Teilaufgabe i)      |        siehe Teilaufgabe i)
```

iv) Nehmen Sie an, dass die CPU in der Lage ist zwei Operationen pro Taktzyklus zu bearbeiten, sofern diese nicht die gleichen Ressourcen (z.B. Hauptspeicher) verwenden. Somit wird zum Beispiel die Kombination

`load B(5),Reg(0) and load A(2),Reg(1)`

nicht unterstützt da beide Operationen auf den Hauptspeicher zugreifen. Andererseits ist

`mult and load B(4),Reg(1)`

erlaubt. Schreiben Sie hiermit Ihr low-level Assembler Programm neu und zählen Sie die Anzahl der Taktzyklen die nun benötigt werden.