Technische Universität Darmstadt

Fachbereich Informatik

Prof. Dr. Andreas Koch

Allgemeine Informatik 2 im SS 2008

Programmierprojekt

Bearbeitungszeit: 14.05. bis 06.06.2008

Organisatorisches (WICHTIG!):

- Das Programmierprojekt ist in der Gruppe zu bearbeiten, in der Sie sich zum Projekt angemeldet haben (also mit ein oder zwei Gruppenmitgliedern).
- Dieses Projekt ist mit 20 Punkten angesetzt, das sind 20% Ihrer Endnote im Fach Allgemeine Informatik 2. Die Abschlussklausur macht die restlichen 80% aus.
- Verwenden Sie als Vorlage das BlueJ-Projekt **propro**, das Sie als **propro.zip** auf unserer Webseite finden, und erweitern Sie dieses. Fangen Sie kein neues Projekt an!
- Tragen Sie in die Readme-Datei des BlueJ-Projekts (das Symbol links oben auf der BlueJ-Arbeitsfläche) Namen und Matrikelnummern aller Gruppenmitglieder ein!
- Abgabe des Projekts:
  - Spätestens am 6. Juni 2008!
  - Verpacken Sie das Projektverzeichnis mit allen Dateien und Unterverzeichnissen in eine ZIP-Datei, die als Namen die Matrikelnummern der Gruppenmitglieder trägt (getrennt durch einen Unterstrich), also z.B. **1234567_1414141.zip** oder **1357531.zip**.
  - Im Terminal / in der Konsole geht das so: `zip -r 1234567.zip propro`
  - Schicken Sie die ZIP-Datei per E-Mail an **ai08@esa.informatik.tu-darmstadt.de**.
  - Der Betreff der Mail muss dabei gleich dem Namen der ZIP-Datei sein.
  - Im Text der Mail sollen noch mal die Namen und Matrikelnummern aller Gruppenmitglieder stehen.
  - Nur ein Gruppenmitglied schickt das Projekt ein!
  - Bei mehreren Abgaben einer Gruppe wird nur die letzte gewertet.
- Formatieren und kommentieren Sie ihren Quellcode hinreichend (JavaDoc!).
- **Achtung:** „Der Fachbereich Informatik misst der Einhaltung der Grundregeln der wissenschaftlichen Ethik großen Wert bei. Zu diesen gehört auch die strikte Verfolgung von Plagiarismus. Mit der Abgabe ihrer Lösung bestätigen Sie, dass Sie der alleinige Autor / die alleinigen Autoren des gesamten Materials sind. Bei Unklarheiten zu diesem Thema finden Sie weiterführende Informationen auf http://www.informatik.tu-darmstadt.de/Plagiarism oder sprechen Sie Ihren Betreuer an.“
  - **Das bedeutet nicht nur, dass „Abschreiben“ verboten ist, sondern auch „Abschreiben lassen“ und „Lösungen vergleichen“ - egal, ob das durch Weitergabe von Programmcode oder mündlich erfolgt!**

Grundlegendes

Außer Java gibt es noch viele andere Programmiersprachen, die sich teilweise sehr stark voneinander unterscheiden. In diesem Projekt beschäftigen wir uns mit einer vereinfachten **Assemblersprache**. Im Gegensatz zu einer höheren Programmiersprache wie Java, die für Menschen relativ leicht verständlich ist, ist eine Assemblersprache eine recht direkte Abbildung des Maschinencodes eines bestimmten Prozessors. Nähere Informationen und einen Vergleich zwischen Assembler- und höherer Programmiersprache finden Sie z.B. in der Wikipedia unter http://de.wikipedia.org/wiki/H%C3%B6here_Programmiersprache.

Unser fiktiver Prozessor verfügt über drei Speichertypen:

- 32 Register R0 bis R31 (entsprechen Variablen)
- Programmspeicher P (enthält das auszuführende Programm)
- Datenspeicher M (enthält Daten, die vom Programm benötigt werden)

Alle Speicherarten werden als **`int`**-Arrays repräsentiert.

Außerdem gibt es einen **Programm-** oder **Befehlszähler**, der die aktuelle Adresse im Programmspeicher beinhaltet.

Assemblerprogramme für unseren Prozessor bestehen aus verschiedenen Befehlen und deren Parametern, die z.B. auf Registern arithmetische Operationen durchführen, Daten aus dem Datenspeicher laden oder dorthin speichern, im Programm hin- und herspringen, Daten von der Tastatur einlesen oder Daten auf dem Bildschirm ausgeben.

Jeder Befehl besteht aus einem Namen, die meisten Befehle erfordern aber auch bis zu drei Parameter (**a**, **b** und **c**). Parameter können nur ganze Zahlen sein. Bei Parametern, die sich auf Register beziehen, darf der Zahl ein „**R**“ vorangestellt sein. Ein gültiger Befehl wäre also z.B. „**ADDI R1 R2 15**“, der aber auch als „**ADDI 1 2 15**“ geschrieben sein kann.

Leerzeilen und alles, was mit „/“ beginnt, wird als Kommentar betrachtet und nicht beachtet.

Arithmetische Operationen:

ADD a b c	**Ra** = **Rb** + **Rc**
SUB a b c	**Ra** = **Rb** - **Rc**
MUL a b c	**Ra** = **Rb** * **Rc**
DIV a b c	**Ra** = **Rb** / **Rc**
MOD a b c	**Ra** = **Rb** % **Rc**
CMP a b c	**Ra** = SIGNUM(**Rb** - **Rc**) →Vorzeichenfunktion: ergibt +1, 0 oder -1

Die Operationen **ADDI**, **SUBI**, **MULI**, **DIVI**, **MODI** und **CMPI** sind fast identisch. Der einzige Unterschied ist, dass nicht mit dem Register **c** gearbeitet wird, sondern direkt mit dem Wert **c** (**I** steht für „immediate“).

Logische/bitweise Operationen:

NOT a b	**Ra** = NOT **Rb** → **~Rb** in Java
AND a b c	**Ra** = **Rb** AND **Rc** → **Rb & Rc** in Java
OR a b c	**Ra** = **Rb** OR **Rc** → **Rb | Rc** in Java
XOR a b c	**Ra** = **Rb** XOR **Rc** → **Rb ^ Rc** in Java
SH a b c	arithmetischer Shift: das Bitmuster der Zahl in **Rb** wird um **Rc** Stellen verschoben, das Ergebnis wird in **Ra** gespeichert. Wenn **Rc** negativ ist,

handelt es sich um einen Shift nach rechts (in Java: **`Ra = Rb >> -Rc`**), sonst um einen Shift nach links (**`Ra = Rb << Rc`**).

Nähere Informationen zu bitweisen Operationen finden Sie z.B. in der Wikipedia unter http://de.wikipedia.org/wiki/Bitweiser_Operator.

Die Operationen **`ANDI`**, **`ORI`**, **`XORI`** und **`SHI`** sind fast identisch. Der einzige Unterschied ist, dass nicht mit dem Register **`c`** gearbeitet wird, sondern direkt mit dem Wert **`c`** (**`I`** steht für „immediate“).

Operationen zur Interaktion mit dem Datenspeicher:

`LDO a b c`	Lädt den Wert an der Stelle **`Rb + c`** im Datenspeicher in das Register **`Ra`**
`LDX a b c`	Lädt den Wert an der Stelle **`Rb + Rc`** im Datenspeicher in das Register **`Ra`**
`STO a b c`	Speichert den Wert von **`Ra`** in den Datenspeicher an die Stelle **`Rb + c`**
`STX a b c`	Speichert den Wert von **`Ra`** in den Datenspeicher an die Stelle **`Rb + Rc`**

Operationen zur Kontrolle des Programmflusses:

`BEQ a b`	Springe um **`b`** Befehle, falls **`Ra`** gleich 0 ist
`BNE a b`	Springe um **`b`** Befehle, falls **`Ra`** ungleich 0 ist
`BLT a b`	Springe um **`b`** Befehle, falls **`Ra`** kleiner als 0 ist
`BGE a b`	Springe um **`b`** Befehle, falls **`Ra`** größer als oder gleich 0 ist
`BLE a b`	Springe um **`b`** Befehle, falls **`Ra`** kleiner als oder gleich 0 ist
`BGT a b`	Springe um **`b`** Befehle, falls **`Ra`** größer als 0 ist

Dabei wird nach vorne gesprungen, wenn **`b`** größer als 0 ist und zurück, wenn **`b`** kleiner als 0 ist. **`BEQ R0 2`** springt also zum übernächsten Befehl, **`BEQ R0 -1`** zum vorherigen.

`JSR a`	Speichere die Adresse des nächsten Befehls in **`R31`**, springe dann zu Befehl **`a`**
`RET a`	Wenn **`a`** gleich 0: beende das Programm, sonst: springe zum Befehl in **`Ra`**

Springen bedeutet, dass der Programmzähler einfach auf einen neuen Wert gesetzt wird.

Operationen zur Ein- und Ausgabe:

`RDD a`	Lies eine Dezimalzahl von der Tastatur in **`Ra`** ein
`WRD a`	Gib die Dezimalzahl in **`Ra`** auf dem Bildschirm aus (und ein Leerzeichen)
`WRH a`	Gib die Zahl in **`Ra`** in Hexadezimalschreibweise auf dem Bildschirm aus (…)
`WRL`	entspricht **`System.out.println();`**

Hexadezimalschreibweise: Darstellung einer Zahl zur Basis 16 (statt 10 wie im Dezimalsystem). Es gibt also zusätzlich zu den Ziffern 0 bis 9 auch die Ziffern A bis F, die dezimal den Werten 10 bis 15 entsprechen. Die Zahl 1000 lässt sich hexadezimal also als 3E8 darstellen, da $1000 = 3 * 16^2 + 14 * 16^1 + 8 * 16^0$. Wie Sie eine Zahl in das Hexadezimalsystem umrechnen, bleibt Ihnen überlassen.

Vorgabe

Im vorgegebenen Projekt gibt es bereits die Klassen **`Parser`** (schon fertig) und **`Simulator`** (da haben Sie noch einiges zu tun). Die Klasse **`Parser`** dient dazu, Assemblerprogramme aus einer Datei in den Programmspeicher einzulesen. Dazu wandelt die Methode **`parse`** die Befehlsnamen in sogenannte **`OpCodes`** („operation codes“) um, mit denen sich dann leichter arbeiten lässt (diese OpCodes sind in beiden Klassen als Konstanten definiert). Außerdem entfernt sie bei den Parametern ein evtl. vorangestelltes „**`R`**“. Da ein Befehl bis zu drei Parameter haben kann, belegt jeder Befehl vier Elemente im Programmspeicher-Array!

Außerdem kann der Parser mittels der Methode **`load`** Daten aus einer Datei in den Datenspeicher lesen.

Aufgabe 1: Simulator

In dieser Aufgabe schreiben Sie einen Simulator, der unseren fiktiven Prozessor simuliert und Assemblerprogramme ausführen kann.

a) Vervollständigen Sie den Konstruktor der Klasse Simulator. Initialisieren Sie zuerst die Arrays: es gibt 32 Register, der Datenspeicher soll 10000 Elemente umfassen, der Programmspeicher soll Platz für 10000 Befehle (und deren Parameter!) bieten. Das Register 0 soll immer den Wert 0 enthalten, der Programmzähler beginnt ebenfalls bei 0.

b) Schreiben Sie das Kernstück des Simulators: die Methode **`private void execute()`**.

   Diese soll in einer Schleife solange den jeweils aktuellen Befehl (also den im Programmspeicher, der an der Stelle des Programmzählers steht) einlesen und ausführen, bis das Programm zu Ende ist (also bis der Befehl **`RET 0`** oder der Programmzähler über die Grenzen des Programmspeichers läuft).

   Nach jedem Befehl muss der Programmzähler auf die Adresse des nächsten Befehls gesetzt werden (der bei Sprungbefehlen nicht der nächste im Programmtext sein muss!). Achten Sie dabei darauf, dass jeder Befehl im Programmspeicher vier Einträge einnimmt!

   Für den Befehl **RDD** haben wir die Methode **`private int input()`** bereitgestellt. Diese liest eine Zahl von der Tastatur ein und gibt sie als Ergebnis der Methode zurück. Sollte dabei ein Fehler auftreten, wird das Programm mit einer Fehlermeldung beendet.

c) Kommentieren Sie die Klasse Simulator und deren Methoden im JavaDoc-Format.

d) Im Projektverzeichnis haben wir Ihnen einige kleine Assemblerprogramme bereitgestellt:

   add.txt Addiert zwei Zahlen und gibt das Ergebnis aus.

   fak.txt Liest eine Zahl ein und berechnet deren Fakultät.

   ggt.txt Liest zwei Zahlen ein und berechnet den größten gemeinsamen Teiler.

   test.txt Testet einige Befehle, sollte auf dem Bildschirm **`1528`** und **`5F8`** ausgeben.

   array.txt Erwartet im Datenspeicher ab Stelle 1000 ein „Array“ mit Werten ungleich 0. Dieses wird dann angezeigt, anschließend werden alle Werte verdoppelt und das Array noch mal angezeigt. Die Datei **_array.txt** enthält ein Beispiel-Array, beim Aufruf der **`main`**-Methode übergeben Sie also als Parameter **`{“array.txt”, “_array.txt“}`**.

   Benutzen Sie diese (und eigene Programme!), um Ihren Simulator zu testen.

Aufgabe 2: Eigene Programme

Benutzen Sie einen Texteditor (im Poolraum z.B. **`gedit`**), um eigene Assemblerprogramme zu schreiben (testen Sie diese natürlich auch!).

a) Schreiben Sie ein Programm **fibo.txt**, das nach Eingabe einer Zahl n die ersten n Zahlen der Fibonacci-Folge auf dem Bildschirm ausgibt. Die ersten beiden Zahlen sind 1, alle weiteren errechnen sich aus der Summe der beiden vorhergehenden Zahlen. Sie können davon ausgehen, dass n größer als 2 ist.

b) Schreiben Sie ein Programm **skalar.txt**, dass ab Speicherstelle 1000 hintereinander die Dimension n der Vektoren, die n Werte des ersten Vektors und die n Werte des zweiten Vektors erwartet, insgesamt also 2n + 1 Werte. Anschließend soll das Skalarprodukt der beiden Vektoren errechnet und auf dem Bildschirm ausgegeben werden. Wenn im Datenspeicher ab Stelle 1000 also die Werte **4 8 15 16 23 42 1 2 3** stehen, soll auf dem Bildschirm die Zahl 452 ausgegeben werden (452 = 8*42 + 15*1 + 16*2 + 23*3).

Testen Sie Ihren Simulator und Ihre Assemblerprogramme ausgiebig – wir tun das auch! Denken Sie auch daran, Ihre Programme vernünftig zu formatieren und zu kommentieren.

Sollten Sie eine Aufgabe nicht bearbeiten können: versuchen Sie trotzdem, die folgenden Aufgaben sinnvoll zu bearbeiten – es gibt auch Teilpunkte.

Viel Spaß und Erfolg bei der Bearbeitung des Projekts!