Beispielangaben zu
OOP (Objekt-Orientierte Programmierung)

SS 2009

Beispiel 3

Relevante Themen:

- Untertyp-Beziehungen
- Zusicherungen
- Wiederverwendung von Code

Design und Implementierung:

Schreiben Sie in C++ einen CPU-Simulator myCPU. Das Programm liest als ersten Kommandozeilenparameter den Dateinamen des auszuführenden Programmes ein, welches die Programminstruktionen als Assemblercode enthält. Der zweite Kommandozeilenparameter, welcher optional ist, erlaubt die Angabe einer Defaultbelegung der ersten Elemente des Programmspeichers. Wird kein Speicherbelegungsfile angegeben, so haben die Elemente des Programmspeichers einen undefinierten Wert.

Die Synopsis des Programms ist folgende: myCPU -c ⟨programfile⟩ [-m ⟨memfile⟩]

Der zu implementierende Prozessor hat 256 generell verwendbare Register, welche als R0 bis R255 ansprechbar sind. Das Register R0 stellt zugleich den Programmzähler dar, kann aber auch direkt modifiziert werden. Sollte der Programmzähler (R0) auf einen ungültigen Wert gesetzt werden, so soll mit einer Fehlermeldung abgebrochen werden. Der aktuelle Wert des Programmzählers entspricht der Zeilennummer (beginnend mit Eins) im Programmfile.
Weiters besitzt der Prozessor ein ZERO-Flag und ein SIGN-Flag, welche durch einen speziellen Test-Befehl gesetzt werden. Die Adressierung der Instruktionen ist nur über Register und Adresslabel möglich (siehe Beschreibung von ADDR). Konstanten können daher nur über die Initialisierung des Programmspeichers benutzt werden.

Format der **Programmdatei**:
Die Programmdatei enthält pro Zeile maximal eine Assemblerinstruktion (kann auch Leerzeilen enthalten, welche zu ignorieren sind). Whitespaces (Leerzeichen, Tabulatoren) sind zu ignorieren. Zeilen, welche mit „#“ beginnen, sind zu ignorieren (Kommentarzeilen).

Für die Beschreibung der Assemblerinstruktionen gelten folgende Konventionen:

- REGx steht für ein beliebiges Register. Am Beginn der Programmausführung haben alle Register den Wert 0 (Null).

- ADDR ist ein Labelname, welcher einer Sprungadresse im Programmcode entspricht. Sollte ein Labelname in einem Sprungbefehl benutzt werden, welcher nicht definiert wurde, so ist mit einer Fehlermeldung abzubrechen.
- DEV entspricht einer vordefinierten symbolischen Adresse für ein Gerät (Device). So werden beispielsweise WDEZ für Dezimalausgabe und WHEX für Hexadezimalausgabe verwendet.

Folgende **Assemblerinstruktionen** sind zu unterstützen:

inc REG1

Erhöht den Wert des Registers um Eins: REG1 = REG1 + 1
Beispiel: inc R2

dec REG1

Erniedrigt den Wert des Registers um Eins: REG1 = REG1 - 1

add REG1, REG2, REG3

Führt folgende Addition durch: REG1 = REG2 + REG3
Beispiel: add R4, R4, R7

sub REG1, REG2, REG3

Führt folgende Subtraktion durch: REG1 = REG2 - REG3

mul REG1, REG2, REG3

Führt folgende Multiplikation durch: REG1 = REG2 * REG3

div REG1, REG2, REG3

Führt folgende Addition durch: REG1 = REG2 / REG3. Sollte REG3 den Wert 0 (Null) haben ist mit einer Fehlermeldung abzubrechen.

load REG1, REG2

Ladet den Inhalt des Hauptspeichers mit Index REG2 in das Register REG1.

store REG1, REG2

Speicher den Inhalt des Registers REG1 in den Hauptspeichers auf Index REG2.

test REG1

Dieser Befehl testet den Inhalt von REG1 und setzt das ZERO-Flag genau dann, wenn REG1 = 0 hat. Das SIGN-Flag wird genau dann gesetzt, wenn der Wert des Registers negativ ist. Andere Instruktionen haben keinen Einfluss auf den Wert des ZERO- und das SIGN-Flags.

label Name :

Definiert ein Adresslabel „Name“ auf das direkt mit Sprungbefehlen verzweigt werden kann.
Beispiel: label LabelX

jumpa ADDR

Der Programmzähler wird ADDR gesetzt.
Beispiel: jumpa LabelX

jumpz ADDR

Der Programmzähler wird auf ADDR gesetzt genau dann wenn das ZERO-Flag gesetzt ist.

jumps ADDR

Der Programmzähler wird auf ADDR gesetzt genau dann wenn das SIGN-Flag gesetzt ist.

write DEV, REG1

Schreibt den Inhalt von Register REG1 auf das Ausgabegerät DEV, wobei DEV für die Gerätenamen WDEZ oder WHEX steht. Dabei wird die Zahl im Dezimal- bzw. Hexformat auf die Standardausgabe geschrieben, gefolgt von einem Zeilenumbruch.
Beispiel: write WDEZ, R7

Der **Programmspeicher** ist als ein dynamisch wachsendes Array zu implementieren, wobei auf die Speicherelemente mit den Instruktionen load und store zugegriffen wird. Das erste Element im Programmspeicher hat den Index 0 (Null).

Das **Speicherbelegungsfile** dient zur Initialisierung der ersten Elemente im Programmspeicher. Das Speicherbelegungsfile enthält pro Zeile maximal eine Zahl, wobei der Inhalt der N-ten Zeile (startend mit Eins) in den Programmspeicher mit Index (N-1) geschrieben wird. Leerzeilen bedeuten, dass das entsprechende Speicherelement nicht initialisiert wird.
Da die Assemblerinstruktionen keine Verwendung von Konstanten als Parameter unterstützen, müssen entsprechende Konstanten im Speicherbelegungsfile definiert werden. Steht beispielsweise im Speicherbelegungsfile in der zweiten Zeile der Wert 20, so kann dieser mit folgendem Programmstück in das Register R3 zugewiesen werden (unter der Annahme, dass Register R1 noch den Default-Wert 0 hat):

```
#   copy the value zero into R2:
add R2,R1,R1
inc R2
#   read memory[R2]:
load R3, R2
```

Erstellen Sie dazu eine Klasse CCPU für die CPU, eine abstrakte Klasse CInstruction für die Assemblerbefehle, eine abstrakte Displayklasse CDisplay für die Ausgabe, eine Klasse CMem für den Programmspeicher, eine Klasse CProgram für den Programmspeicher, sowie eine Klasse CDat für den Datentyp der Assemblerinstruktionen.

Von CInstruction sind die einzelnen Assemblerbefehle abzuleiten. Von CDisplay sind die jeweiligen Displays abzuleiten.
Die Klasse CDat implementiert alle Rechenoperationen, welche für die jeweiligen Assemblerinstruktionen benötigt werden und kann eine aus dem Speicherbelegungsfile geladene Zeile parsen um den Wert zuzuweisen. Für dieses Beispiel ist als Datentyp von CDat der C++-Datentyp int zu verwenden (keine speziellen Zahlenformate bzw. Rechenregeln). Jedoch sollte das Programm so entworfen werden, dass durch alleiniges Umschreiben der Klasse CDat mit einem beliebigen anderen Datentyp gearbeitet werden kann (natürlich vorausgesetzt, dass dann auch das Speicherbelegungsfile im kompatiblen Format ist).

Beispiele:

Speicherbelegungsfile	Programmcode

10	`#  set R2 = 10`
	`load R1, R2`
	
	`#  start of loop`
	`label Loop:`
	`inc R3`
	`sub R4, R3, R2`
	`test R4`
	`jumpz EndLoop`
	`write WDEZ, R3`
	`jumpa Loop`
	
	`label EndLoop:`

Obiges Programmstück gibt die Zahlen 1 – 10 aufsteigend im Dezimalformat aus.

Die in der Spezifikation nicht genauer ausgeführten Teile können selbst sinnvoll ausgestaltet werden, wobei jedoch die folgenden Anforderungen einzuhalten sind. Alternative Ableitungshierarchien werden akzeptiert, wenn eine passende Begründung angegeben wird.

Anforderungen:
Kommentieren Sie den Code, um das Verstehen des Codes zu vereinfachen. Zusätzlich soll zu Beginn jeder Klasse eine Klassenbeschreibung stehen. Am Beginn jeder Datei soll außerdem Name, MNr, KZ der Gruppenmitglieder sowie die Beispielnummer stehen. Ein nicht kommentierter Code wird negativ bewertet!

Ausnahmen (Exceptions) müssen immer abgefangen werden, sofern möglich. Überlegen Sie sich dabei aber auch, welche Ausnahmen zu einer Programmbeendigung führen müssen und in welchen Fällen es ausreicht, eine Warnung bzw. Fehlermeldung auszugeben. Bei jeder Ausnahmebehandlung ist auf jeden Fall eine Fehlermeldung auf die Fehlerausgabe (cerr) auszugeben. Als Orientierung seien einige mögliche Quellen für Ausnahmen aufgezählt: Speichermangel, inkorrekter Input, Schreiben auf Datei fehlgeschlagen, etc.

Schreiben Sie klare Zusicherungen an den passenden Stellen im Programm. Dort, wo man Zusicherungen kompakt als Code formulieren kann, wird empfohlen, die Zusicherungen zusätzlich als „assert()“ hinzuschreiben, damit der Compiler Code generieren kann zur automatischen Prüfung der Zusicherungen zur Laufzeit. Mit der Compileroption „-DNDEBUG“ kann man den fertig ausgetesteten Code generieren sodass „assert()“ keinen Overhead mehr verursacht.

Benutzen Sie zur Programmentwicklung auch ein Makefile, welches zumindest die Targets „clean“, „all“ und „run“ enthält. Mittels „make run“ soll das Programm direkt aufgerufen werden können (mit sinnvollen Argumenten versehen).

Es sind die Programmierrichtlinien der LVA einzuhalten.

Schreiben Sie zu dem Programm ein Protokoll, das folgende Informationen beinhaltet:

- Identifikation (Name, MNr, Kz)
- Aufgabenstellung (kann direkt von dieser Angabe übernommen werden, Beispiele mit Grafiken sind nicht zu übernehmen)
- Klassendiagramme (inkl. Beschreibung)
- Schematische Beschreibung der Allokation und Freigabe von Ressourcen
- Schematische Beschreibung der Ausnahmebehandlung (mögliche Ursachen von Exceptions, sowie die Strategie bei deren Behandlung (u.a., ob lokal oder eher global behandelt))
- Dokumentation des Arbeitsaufwandes
- Dokumentation von ev. aufgetretenen Problemen
- Kommentierter Programmcode

Der Umfang des Protokolls soll (ohne Mitzählen der Codeseiten) zwischen 5 bis 10 Seiten sein.

Abgabe:

Legen Sie ein Verzeichnis „Beispiel3“ an, in dem Sie alle Ihre Programmdateien hineingeben. Geben Sie auch das Protokoll in dieses Verzeichnis hinein.
Packen Sie das Verzeichnis folgendermaßen ein:

```
tar –zcvf Beispiel3.tgz Beispiel3
```

und geben Sie dieses Archivfile elektronisch ab. Ohne diese elektronische Abgabe sowie das Mitbringen eines schriftlichen Ausdruckes des Protokolls kann die Abgabe nicht positiv gewertet werden.

Der relevante Stoff für das Abgabegespräch ist das Kapitel 1 sowie das Kapitel 2 des Skriptums. Es ist dabei auch notwendig, dass Sie den Text, wo zutreffend, für die Lösung des Beispiels anwenden.