Softwaretechnik
2002
Prof. Dr. H. Klaeren und M E Leypold

Eberhard Karls
Universität
Tübingen

Übungsblatt 5

Entwurf eines Disassembler

Ausgabe: 4.6.2002
Abgabe: 13.6.2002, 12:00

Inhaltsverzeichnis

Vorwort

In diesem Übungsblatt sollt Ihr selbst ein kleines Programm entwerfen. Ich werde zu Anfang die Aufgabe (das zu entwerfende Programm) beschreiben, und dann einige Hinweise zur Ausführung geben. Da zwei dieser Hinweise recht umfangreich geworden sind, habe ich diese auf je einem separaten Handout untergebracht:

- *Das EVA Paradigma*.
- *Beschreibung von Modulen in natürlicher Sprache*.

Bitte beachtet, dass Ihr das Programm in Aufgabe 1 (Disassembler) *nur entwerfen* (wie, das wird in der Aufgabe näher erläutert), aber auf keinen Fall programmieren sollt. Also: *Keine Panik!*

1 Systementwurf: Ein einfacher Disassembler

Ein Auftrag: *ThinkCrime Incorporated* hat schon wieder einen neuen Auftrag erhalten. Wie immer in schwierigen Fällen greift Dein Chef Bertold Brecheisen wegen Deiner Softwaretechnikkenntnisse auf Dich zurück und erpresst Dich zur Mitarbeit.

Dieses Mal geht es darum, einen (einfachen, nicht interaktiven) Disassembler für den Binärcode eines Mikroprozessorsystems zu entwickeln. Ein Kunde (der lieber ungenannt bleiben möchte), will Software für einen alten 8-Bit-Mikrocontroller, die nur noch im binären Format vorliegt, analysieren. Man weiß nicht so recht, ob er die Quellen im Verlauf der letzten Jahre verlegt hat, oder fremden, ungewarteten Code verändern möchte. Dies spielt jedoch keine Rolle.

Ich werde zuerst kurz das System und das Format, in dem die Programme für diese Mikrocontroller vorliegen, beschreiben, und dann näher auf die Leistung des verlangten Disassemblers eingehen.

Nicht alle der hier gegebenen Informationen sind wirklich wichtig für den Entwurf, aus diesem Grund empfiehlt es sich, die Beschreibung des Mikrocontrollers zuerst zu überfliegen und unter Umständen später noch einmal genauer durchzulesen.

1.1 Der Mikrocontroller

Programmformat Die Programme für die Sorte von Systemen, um die es hier gehen soll, bestehen aus zwei Abschnitten: Einem Teil mit Maschineninstruktionen (Code-Segment), der an eine bestimmte Adresse in den Speicher geladen wird, und einem Teil mit (statischen) Daten (Daten-Segment), der an eine andere Adresse in den Speicher geladen wird. Die Programmausführung beginnt immer mit der ersten Instruktion des Code-Segments.

Eine Maschineninstruktion besteht immer aus einem Byte[1] (Opcode), auf welches – je nach dem Adressierungsart des im Opcode kodierten Befehls – entweder direkt die nächste Instruktion folgte, oder ein Operand aus ein oder zwei Bytes.[2]

[1] Ein Byte ist 8 Bit lang.

[2] Der – fiktive – Mikrocontroller, um den es hier geht, ist vom 6502 (weiland in Apple II und C64 eingesetzt) inspiriert. Wer über den 6502 mehr erfahren möchte, sei auf die die folgenden URLs hingewiesen: Unter `http://www.obelisk.demon.co.uk/6502` findet sich ein guter Überblick über die Architektur und den Befehlssatz (inklusive Adressmodi) des 6502, unter `http://zeus.eed.usv.ro/misc/mirrors/cc/6502.htm` eine Referenz des Befehlssatzes.

Die handelsüblichen Compiler und Assembler für diese Sorte von Systemen[3], wie sie der Kunde besitzt, erzeugen die Programme in einem bestimmten einheitlichen Format:

- Am Anfang der Datei befindet sich ein Header, der Ladeposition (siehe oben) und Länge von Daten- und Code-Segment (in Byte) angibt. Diese Angaben sind jeweils als 16-Bit-Zahlen abgelegt und zwar zuerst Ladeposition und Länge des Code-Segment, dann folgen unmittelbar dieselben Angaben für das Daten-Segment.
- Unmittelbar nach dem Header folgt das Code-Segment (Byte für Byte als Speicherabbild) und unmittelbar danach das Daten-Segment.

16-Bit-Zahlen werden, sowohl im Header als auch in den Maschineninstruktionen, als 2 Bytes (das "least significant byte" zuerst) gespeichert.

Mnemoniks Wie vielleicht bekannt ist, besitzt jede Maschineninstruktion eine binäre Darstellung (so, wie sie dann in der Programmdatei gespeichert ist), als auch eine (mehr oder weniger lesbare) textuelle Notation, in der die Art der Instruktion durch eine symbolische Abkürzung dargestellt wird, der Operand als Hexadezimalzahl und der Adressmode als Annotation des Operanden (Runde Klammern, eckige Klammern, besondere Zeichen vor dem Operanden usw.). Beispielsweise bedeutet die Bytefolge `A9 1A`[4], dass das Akkumulator-Register mit 26 geladen werden soll. In "mnemonischer" Schreibweise wird dies mit `LDA #1A` dargestellt. Dabei ist `LDA` die Art der Instruktion (der *Mnemonik*) und das Zeichen "#" signalisiert den Adressmode "immediate" (der Operand ist keine Adresse, sondern soll unmittelbar verarbeitet werden).

Das Dekodieren des Speicherabbildes eines Programms (also einer Folge von Bytes) in Maschineninstruktionen und Ihre Ausgabe in "Mnemonischer" Notation wird als *Disassemblierung* bezeichnet. Die Disassemblierung ist hier besonders einfach, da wir genau wissen, wo die Maschineninstruktionen liegen und bekannt ist, dass das Daten-Segment aus einer einzigen ununterbrochenen Folge von Maschineninstruktionen besteht (es befinden sich keine "ungültigen" Opcodes dazwischen).

Zur Beschreibung des Mikroprozessors, um den es vorerst geht, wird der Kunde ein Tabelle liefern, in der Zeile für Zeile die verschiedene (gültigen) Opcodes beschrieben sind. Jeder Zeile besteht aus

1. dem Opcode in hexadezimaler Form,
2. der Angabe des Adressmodus, in (genau!) zwei Zeichen codiert,
3. und dem Mnemonik zum Opcode.

Die Angaben sind jeweils durch Kommas voneinander getrennt. Eine Zeile dieser Beschreibungsdatei sieht beispielsweise folgendermassen aus:

```
A9,IM,LDA
```

Indirekt in der Angabe des Adressmodus auch enthalten, wie lang der Operand der Maschineninstruktion ist. Beispielsweise haben die Adressmodi *zero page* und *immediate* jeweils Operanden von 1 Byte Länge, aber die Adressmodi *implicit* und *accumulator* jeweils keinen Operanden.

[3] Für die *fiktiven* Systeme, um die es hier geht! Ich behaupte nicht, dass die (damals) üblichen Assembler und Compiler für den 6502 irgendein bestimmtes Format für die Binärdateien eingehalten hätten.

[4] Beim 6502. Notation ist hier hexadezimal.

1.2 Aufgabenstellung:

Der Kunde hat sich Werkzeugprogramme und Hardware entwickelt, mittels deren er die Programme für die Mikrocontroller von den verschiedenen Medien (Lochstreifen, Magnetband, 5.25-Zoll-Disketten) auf ein modernes Unix-System übertragen kann. Er wünscht sich nun – um eine Grundlage für seine weiteren Analysen zu haben, ein C++-Programm, welches die Programmdatei liest und das Code-Segment in mnemonischer (disassemblierter) Form ausgibt. Jede Zeile der Ausgabe soll mit einer (hexadezimalen) Adresse beginnen, und danach die mnemonische Darstellung der Instruktion erhalten:

```
Program       : FNNOX.COM
Start  of Code: $1000
Length of Code:  1213 Bytes
Start  of Date: $2000
Length of Data:   210 Bytes

            1000     NOP
            1001     LDA #10
            1003     ADD $203A
            1006     STA $03
```

...

Der Kunde liefert die bereits erwähnte Beschreibungsdatei für die Mikrocontroller. Die Informationen aus dieser Datei werden dazu benötigt, die Programmdatei zu parsen und die Opcode in Mnemoniks zu übersetzen. Die disassemblierte Darstellung des Programms soll (vorerst) auf die Standardausgabe ausgegeben werden, so dass der Anwender sie mit den Funktionen, die Unix zur Ausgabeumleitung zur Verfügung stellt, drucken oder in einer Datei speichern kann.

Deine Aufgabe ist es nun, eine Architekturbeschreibung dieses einfachen Disassemblers zu entwerfen. Zu dieser Architekturbeschreibung gehören:

- Eine Liste aller Module mit einem kurzen beschreibenden Satz, der die wesentliche Leistung des Moduls charakterisiert.
- Ein Diagramm, aus dem die Benutzungsbeziehungen zwischen den Modulen entnommen werden können.
- Eine detaillierte Beschreibung der einzelnen Module: Dazu gehören die Headerdateien und eine natürlichsprachige Beschreibung der Invarianten der exportierten Datenstrukturen und der Berechnungsleistung der einzelnen Operationen.

Übrigens ist es vermutlich einfacher, als erstes die detaillierte Beschreibung der einzelnen Module anzufertigen. Wie diese Beschreibung aussehen sollte, ist im Handout *Beschreibung von Modulen in natürlicher Sprache* und im Abschnitt mit den Hinweisen noch näher erläutert.

1.3 Tipps zur Ausführung

Die folgenden Tipps zur Durchführung des Systementwurfs solltet Ihr als unverbindliche Hinweise betrachten: Mit einigen könnt Ihr möglicherweise nichts anfangen, andere treffen für Euren individuellen Ansatz möglicherweise nicht zu. Lasst Euch dadurch nicht beunruhigen.

Woher kommen Module? Die einfachste, aber bei weitem nicht einzige Art, wie man Module finden kann, besteht darin, typische Datenstrukturen des Programms zu identifizieren und die Operationen, die auf Ihnen durchgeführt werden. Ist es möglich oder sinnvoll, dass auf eine Datenstruktur ausschließlich durch bestimmte Operationen zugegriffen wird, so packt man die Datenstruktur und die Prozeduren, welche die Operationen implementieren in ein Modul. In solchen Fällen spricht man von *abstrakten Datentypen*.

Um Euch hier nicht auf die falsche Spur zu setzen: Es ist unwahrscheinlich und häufig auch unwirtschaftlich, dass sich ein ganzes Programm ausschließlich aus abstrakten Datentypen zusammensetzen lässt. In vielen Fällen benötigt man auch mehr oder weniger transparente Datentypen. Ein typisches Beispiel wäre ein Puffer, der von einer geeigneten Lese-Funktion (die von einem externen Gerät liest) mit Daten gefüllt wird, und in dem mit einem Zeiger frei navigiert werden soll. Man kann zwar versuchen, so etwas mit Iteratoren oder ähnlichen Konstrukten zu abstrahieren, aber es erscheint doch ungeheuer aufwändig und in den meisten Fällen wird es angemessener sein, den Puffer mit `'item foo[max_len]'` zu deklarieren, d. h. die Struktur des Puffers sichtbar zu lassen.

Die typischen Aufgaben für die abstrakte Datentypen dagegen in Frage kommen, sind Prozeduren, die auf einer Datenstruktur operieren, und in denen komplexe Leistungen erbracht werden, und Datenstrukturen, auf denen nichttriviale Invarianten eingehalten werden müssen. Im ersteren Fall ermöglicht das Verbergen der tatsächlichen Implementation (Darstellung) der Struktur, diese auch im Nachhinein so anzupassen, dass die komplexe Leistung effizienter erbracht werden kann. Im letzteren Fall bleibt die Notwendigkeit, sich mit der Erhaltung der Invariantenauseinanderzusetzen, auf eine Handvoll Prozeduren beschränkt, die, wenn sich die Darstellung der Daten ändert, dann auch als einzige geändert werden müssen.

Zusammenfassend: Haltet im vorliegenden Fall Ausschau nach abstrakten Datentypen, aber nicht alle zu definierenden Datenstrukturen müssen mit Gewalt abstrakt gemacht werden.

Nicht auf Optimierungen schielen! Ein Fehler, der immer wieder gerne begangen wird, besteht darin, zu früh Effizienzüberlegungen zum Maßstab des Entwurfs zu machen. Dies ist in der überwiegenden Zahl der Fälle falsch, und führt zu einem unklaren, verwirrenden Entwurf (siehe auch Bentley, 1998, S. 81ff).

Als allgemeine Regel, von der nur mit gutem Grund abgewichen werden sollte, kann für kleine und mittlere Programme aufgestellt werden, dass Effizienzüberlegungen während des Entwurfsprozesses höchstens sehr vorsichtig angestellt werden sollten. Vorrangiges Ziel des Entwurfs sollte es nämlich sein, eine möglichst verständliche Gliederung der stattfindenden Datenverarbeitung zu präsentieren.

In der Praxis hat sich herausgestellt, dass sich solche Programme auch gut im Nachhinein optimieren lassen, indem einzelne “Hot Spots” ermittelt und entschärft werden, da aufgrund der Modularität Änderungen einfacher und mit weniger Konsequenzen für den Rest des Programms durchgeführt werden können.

Selbstverständlich muss Effizienzüberlegungen bereits im Entwurf Raum gegeben werden, wenn Effizienz bzw. schwer zu erfüllende Performance-Anforderungen zu den explizit gewünschten Attributen der Software gehören. Aber auch dann ist es sinnvoll, zuerst einen Entwurf anzufertigen, und dessen Effizienz im voraus zu schätzen.

Zusammenfassend möchte ich für die vorliegende Aufgabe empfehlen, sich keine Gedanken über die Effizienz des Programms zu machen: Bei einer Aufgabe dieser Größenordnung spielt Effizienz noch keine wirkliche Rolle.

EVA-Paradigma: Aussage und Bedeutung dieses Paradigmas sind im Handout *Das EVA Paradigma* näher erläutert. Dort wird auch das Problem der Speichereffizienz bei einer Gestaltung von Programmen nach dem EVA-Paradigma angesprochen: Dies ist in dieser Aufgabe kein Problem (warum?). Gerade der Disassembler lässt sich mit Hilfe des EVA-Paradigmas sehr schön strukturieren.

Adressmodi und die Prozessorarchitektur Eigentlich sollte es möglich sein, den Entwurf ohne eine exakte Kenntnis der Adressierungsarten und der Prozessorarchitektur ausführen. Die obige Beschreibung des Prozessor enthält alle nötigen Informationen, was noch nicht bekannt ist, kann entsprechend abstrahiert werden.

Für diejenigen jedoch, die ohne mehr Information über den Prozessor nicht glauben auskommen zu können, legen wir diesem Übungsblatt einen Ausdruck einiger Webseiten von `http://www.obelisk.demon.co.uk/6502/` bei: Für die Zwecke der Aufgabenstellung können wir so tun, als ob es sich bei dem in Frage stehende Mikroprozessor um einen 6502 handelt.

Speichermanagement Praktisch die meisten Programmier- und Entwurfsfehler bei Projekten, in denen C oder C++ als Sprachen eingesetzt werden, werden durch den Gebrauch von Zeigern und die Notwendigkeit verursacht, den größten Teil des Speichermanagements selbst zu implementieren – d. h. Speicher bei Bedarf zu allozieren, und später wieder freizugeben.

Bei Programmen, die eine genau definierte, endliche Aufgabe erledigen, kann man sich das Leben allerdings etwas einfacher gestalten:

- In vielen Fällen lässt sich eine Obergrenze für die Menge des Speichers angeben, der während des Programmlaufs alloziert werden muss. Wenn diese Obergrenze niedrig genug liegt, kann man sich das Freigeben des Speichers vollständig sparen, und entgeht auf diese Art Problem, die daher rühren, dass Speicher freigegeben wird, der eigentlich noch anderswo referenziert wird. Der gesamte belegte Speicher wird unter vernünftigem Betriebssystemen (Unix) bei Prozessende sowieso wieder an das Betriebssystem zurückgegeben.
- In anderen Fällen kann man den maximalen Speicherplatzbedarf (für alle "vernünftigen" Eingaben) schätzen. Wenn dieser Niedrig genug ist, belegt man mit *malloc()* den maximalen Speicher beim Programmstart und spart sich dadurch den Aufwand, Speicher bei Bedarf nachzuallozieren.

1.4 Beschreibung von Interfaces:

Zur Beschreibung eines Moduls gehören die Headerdateien (welche Typaspekte, Aufrufsyntax und Speicherstrategien beschreiben), Beschreibungen der Invarianten der (nichtopaken) Datenstrukturen, und eine Beschreibung der Berechnungsleistung der einzelnen Prozeduren.

Letzte kann durch eine formale Spezifikation (z. B. in VDM-SL) erfolgen. In vielen Fällen ist aber eine explizite Modellierung zu aufwändig, dann kann man die Leistung der Prozeduren in natürlicher Sprache beschreiben. Wie das aussehen kann, und wie man sich dabei

vom Geist der formalen Spezifikation leiten lassen kann, ist im Handout *Beschreibung von Modulen in natürlicher Sprache* skizziert.

1.5 Kontrollfragen:

Wichtiges Bewertungskriterium für diese Aufgabe ist die Modularität des erarbeiteten Entwurfs. Gute Modularität erkennt man unter anderem daran, dass (voraussehbare) Änderungen nur an lokal begrenzten Stellen des Quelltextes vorgenommen werden müssen: Im Idealfall nur in einem Modul. Die folgenden Fragen können als Test für diesen Aspekt Deines Entwurfs dienen:

Wo muss (gemäß Deinem Entwurf) die Implementation geändert werden, wenn ...

1. ... der Kunde möchte, dass in der Ausgabe symbolische Sprungziele generiert werden. Statt

   ```
   1001          LDA #10
   1003          ADD $203A
   1006          JMP $1003
   ```

   soll also nun

   ```
   1001          LDA #10
   1003 loop12:  ADD $203A
   1006          JMP loop12
   ```

   ausgegeben werden.

2. ... das Programm für einen anderen Mikroprozessor angepasst werden soll, insbesondere, für einen mit einer sehr viel größeren Wortbreite (nehmen wir an 64 Bit)?

3. ... wenn sich nach der Änderung zur Disassemblierung von Programmen für einen 64-Bit-Mikrocontroller herausstellte, dass die Zuordnung von Opcodes zu Mnemoniks aufgrund der viel größeren Zahl an Opcodes einen großen Teil der Rechenzeit schluckt, also effizienter gemacht werden muss.

4. ... wenn sich das Format der Mikroprozessorbeschreibung ändert, weil zum Beispiel ein Feld mit Angaben über Taktzyklen der einzelnen Befehle eingefügt wird.

5. ... der Kunde eine andere Notation der Adressierungsarten wünscht.

6. ... der Kunde eine Änderung des Formats der Ausgabe wünscht (Spaltenbreite), oder gar die Einführung von Parametern, mit denen sich das Ausdruckformat steuern lässt.

7. ... der Kunde wünscht, dass der Disassembler auch noch einen geeigneten Seitenumbruch (die Seite zu 60 Zeilen mit jeweils Seitenzahl und Dateiname in Kopf und Fuß) vornimmt.

Beantworte diese Fragen schriftlich, und gib auch diese Aufzeichnungen mit ab.

Abzugeben sind: Liste aller Module, mit je einem beschreibenden Satz der jeweiligen Aufgabe. Diagramm der Benutzungsbeziehungen zwischen den Modulen. Beschreibung aller Modulinterfaces (exportierte Konstrukte als C++-Prototypen (Headerfiles) und Beschreibung der erbrachten Funktionen). Kurzbeschreibungen der Änderungen von Entwurf oder Implementation (Ort und Art), die auf die beschriebenen Änderungen der Anforderungen hin vorgenommen werden müssten.

Bewertungskriterium ist sind Sinnhaftigkeit, Übersichtlichkeit und Modularität des Entwurfs.

Viel Spaß!

Literatur

[Bentley 1998] BENTLEY, Jon: *Programming pearls*. Addison-Wesley, 1998. – ISBN 0-201-10331-1