Projektarbeit

Geschichte der Programmierung -

und wie sich diese durch die Computerchip-Entwicklung verändert hat

von Chris Anders

Klasse: 1BK1T

Betreuer: Alessa Waldvogel

Inhaltsverzeichnis

[Inhaltsangabe 2](#_Toc65565728)

[Geschichte der Prozessoren 3](#_Toc65565729)

[Aufbau und Funktionsweise einer Minimalmaschine 4](#_Toc65565730)

[Programmierung mit maschinennaher Sprache 5](#_Toc65565731)

[Höhere Programmiersprachen 6](#_Toc65565732)

[Compiler 6](#_Toc65565733)

[Maschinencode vs. Bytecode 8](#_Toc65565734)

[Entwicklerwerkzeuge 10](#_Toc65565735)

[Fazit 12](#_Toc65565736)

[Anhang 13](#_Toc65565737)

[Selbstständigkeitserklärung 14](#_Toc65565738)

Inhaltsangabe

Geschichte der Prozessoren

Aufbau und Funktionsweise einer Minimalmaschine

Die Minimalmaschine ist ein Lernmodell und basiert auf die Von-Neumann-Architektur. Diese ist in 5 Komponenten aufgebaut (siehe Abb. 7).

Der Taktgeber sendet zyklisch Signale über Datenbusse. Die Kontrolleinheit dekodiert Programmbefehle und sendet Daten und Befehle an andere Komponenten weiter. Register speichern einen einzelnen numerischen Wert. Der Speicher ist eine aufeinanderfolgende Menge von Speicherzellen, die mit einer Adresse ansprechbar sind. Die Arithmetik Logic Unit (kurz ALU) führt Berechnungen und logische Operationen durch und gibt das Ergebnis an das Akkumulator Register weiter.

Anhang 1 ist ein Link zu einem selbstgeschrieben Simulator, der eine ähnliche Architektur verwendet. Dieser enthält den SourceCode, sowie ausführbare Dateien für Linux und Windows (32 und 64 Bit).

Programmierung mit maschinennaher Sprache

Maschinencode ist die Sprache, die der Prozessor versteht. Jeder Prozessortyp hat seine eigene Variante dieses Codes. Die Grundlagen sind jedoch gleich. Die Maschinensprache setzt sich aus aneinander gereihten Instruktionen zusammen. Jede Instruktion besteht aus einem OP-Code, also der Anweisung was genau getan werden soll und den dazugehörigen Argumenten, die in die Instruktion kodiert werden. Je nach Architektur ist die Größe und die Verteilung der Argumente unterschiedlich.

Ein Beispiel einer Instruktion an Hand des Intel IA 32 Prozessors(1):

|  |  |
| --- | --- |
| Instruktion als Bytes | 48 B8 8877665544332211 |
| Instruktion als Text | MOV RAX,1122334455667788H |
| Beschreibung | Schreibe den Wert von Memory Offset 1122…88H in Register RAX |

(Das Präfix "h" am Ende einer Zahl sagt aus, dass diese in Hexadezimaler Schreibweise ist)

Das Beispiel zeigt deutlich die Struktur einer Instruktion:

48 B8 8877665544332211

OpCode Argument Argument

(Mov) (Register RAX) (Memory Offset 1122..88H)

Die Reihenfolge der Argumente ist je nach Prozessor unterschiedlich. Die meisten Instruktionen benötigen 1 bis 2 Argumente, selten 3. Es gibt auch Ausnahmen, die keine Argumente, zum Beispiel die "ret"-Anweisung, benötigen.

(1) <https://www.intel.com/content/dam/www/public/us/en/documents/manuals/64-ia-32-architectures-software-developer-instruction-set-reference-manual-325383.pdf> (S. 45)

Höhere Programmiersprachen

[ToDo: Hier Text einfügen]

Compiler

Ein Compiler, von Englisch "to compile – zusammenführen", übersetzt Programmcode in für den Computer lesbare Sprache. Dabei besitzen alle Compiler 4 Grundphasen (Abbildung 1).

Folgender Code soll als Beispiel übersetzt werden:

f(x) = x exponent 2

1. Phase: Der Lexer zerlegt den Programmcode in sogenannte Lexeme/Tokens, die den Code gruppieren. Ein Lexem könnte zum Beispiel eine Ganzzahl, eine Zeichenkette oder ein Sonderzeichen sein.

Identifier: f  
OpenBrace  
Identifier: x  
CloseBrace  
Whitespace  
Equals  
Whitespace  
Identifier: x  
Whitespace  
Identifier: exponent  
Whitespace  
Integer: 2  
EndOfLine

2. Phase: Die Lexeme werden, basierend auf grammatikalischen Regeln, in einen Abstrakten-Syntax-Baum konvertiert. Dieser repräsentiert den Quellcode, mit der der Compiler arbeiten kann. Wie die Regeln angewendet werden ist von Compiler zu Compiler unterschiedlich.

3. Phase: Der Syntaxbaum wird auf Semantik überprüft.

4. Phase: Der Syntaxbaum (Abbildung 2) wird zu Befehlen konvertiert, die der Prozessor oder die Laufzeitumgebung ausführen können.

Heutige Compiler besitzen noch Zwischenschritte, wie zum Beispiel Codeoptimierungen. Der beigefügte Simulator verwendet 3 verschieden komplexe Compiler.

Maschinencode -> Zahlenblöcke werden zu bytes konvertiert

Assembly -> Wird zu einem Syntaxbaum konvertiert und in equivalentem Bytecode umgewandelt.

Hochsprache -> Abstraktion durch Sprachelemente, Konvertierung zu Syntaxbaum und Umwandlung in Bytecode.

Maschinencode vs. Bytecode

Bytecode bezeichnet eine Zwischensprache, die von einer Laufzeitumgebung während der Programmausführung in Maschinencode übersetzt wird, die für den jeweiligen Prozessor spezifisch ist.

Ein klarer Nachteil des Bytecodes ist die schlechtere Performance, da das Programm nicht direkt ausgeführt wird. Aber die Laufzeitumgebungen sind recht gut optimiert, so dass der Unterschied der Laufzeitgeschwindigkeit nur gering ausfällt. Außerdem kann das Programm nicht direkt auf die Hardware zugreifen, sondern der Bytecode muss erst Ahead-of-Time, also vor der Ausführung, in Maschinencode übersetzt werden.

Ein klarer Vorteil gegenüber des Maschinencodes ist die gute Portierbarkeit von Programmen, da der Assembly Code nicht für jeden Prozessor einzeln vor der Ausführung übersetzt werden muss. Dadurch, dass es für jeden Prozessor einen eigenen Assembly Dialekt gibt, muss gegebenenfalls der Source-Code angepasst werden. Was bei größeren Projekten sehr viel Aufwand bedeuten würde. Also ist die Entwicklung von Cross-Plattform Programmen weniger aufwendig.

Eine Garbage-Collection kümmert sich bei den meisten Laufzeitumgebungen, um nicht mehr benötigten Speicher. Somit muss der Entwickler selbst keine Speicherbereinigung mehr durchführen und Speicherfehler können somit vermieden werden. Ein weiterer Vorteil sind Laufzeit-Checks, die auf bestimmte Eigenschaften des Codes prüfen, wie z.B. die Indexierung von Arrays(1). Dadurch werden Buffer-Overflow(2) Angriffe umgangen.

Die .Net Laufzeitumgebungen verwenden den sogenannten IL-Code (Intermediate Language Code). Er wurde entwickelt, damit mehrere Programmiersprachen auf einer Laufzeitumgebung ausgeführt werden können. Mittlerweile kann dieser auch auf mehreren Plattformen ausgeführt werden.

Ein Beispiel für so ein Programm, das Hello World auf der Konsole ausgiebt, könnte so wie in Abbildung 3 aussehen.

(1) Ein Array ist eine aufeinanderfolgende Liste von Elementen, die mit einem Index ansprechbar ist, um ein Element zu lesen oder zu verändern.

(2) Bei einem Buffer-Overflow Angriff wird versucht außerhalb des Arrays zu schreiben, um somit Schadhaften Code auszuführen.

Entwicklerwerkzeuge

Die sogenannten Entwicklerwerkzeuge helfen dabei Programme zu schreiben. Meist sind diese in einer Integrierten Entwicklungsumgebung, kurz IDE zusammen gebündelt. Diese enthält einen Compiler und einen starken Texteditor.

Der Texteditor unterstützt heute einerseits durch Syntax-Highlighting (Abb. 4), also bestimmte Textelemente einer Sprache wie Schlüsselwörter werden farblich hervorgehoben, um die Lesbarkeit zu erhöhen. Zusätzlich bietet der Editor Code-Completion an, der bei der Eingabe als Popup erscheint und zeigt, welche Befehle in dem konkreten Kontext der Sprache möglich ist. Das hilft besonders beim Lernen einer neuen Programmiersprache, da man nicht stumpf Befehle auswendig lernen muss, sondern man kann nach dem "Learning by Doing" Prinzip vorgehen.

Zusätzlich bieten heute einige Entwicklungseditoren Code-Actions an. Dabei wird die aktuelle Datei in einen Syntaxbaum geparst und verschiedene syntaktische Änderungen angeboten. Weitere Vorteile gegenüber normalen Texteditoren sind Zeilennummern zur Orientierung und bei stark typisierten Sprachen wird beim Aufrufen einer Funktion gezeigt, welche möglichen Eingabeparameter verwendet werden können oder müssen (Abb. 5).

Die Fehlersuche in einem Programm wird durch das Debugging (Abb. 6) vereinfacht. Hier kann man Schritt für Schritt das Programm ausführen lassen und schauen welche Werte in den verschiedenen Variablen stecken. Dieses Vorgehen erleichtert immens die Fehlersuche, da man nicht selbst immer wieder die Inhalte von Variablen ausgeben muss. Es können zusätzlich bei den Breakpoints, also den Punkten an denen das Programm angehalten werden soll, Bedingungen verknüpft werden. Sollte es zu syntaktischen Fehlern kommen werden diese in einem Fenster angezeigt und man kann direkt zu dem Code springen, der einen Fehler enthält.

Zusätzliche Unterstützung beim Einbinden von Bibliotheken bieten die Paketmanager, mit denen man Bibliotheken installieren oder aktualisieren kann.

Viele Editoren bieten eine Integration einer Versionsverwaltung an. Eine Versionsverwaltung bietet die Möglichkeit in der Chronik des Programmes hin- und herzuspringen. Zusätzlich kann die Versionsverwaltung genutzt werden um gemeinsam an Projekten zu arbeiten. Ein sehr prominentes Beispiel einer Versionsverwaltung bietet GIT bzw. GitHub.

Am Anfang der Programmierung gab es keine Hilfswerkzeuge. Die Entwickler mussten sehr viel auswendig lernen und hatten es sehr schwer Fehler zu finden. Erst nach und nach kamen immer mehr Features hinzu, die die Arbeit erleichterten.

Fazit

Anhang

1. <http://furesoft.ml/shared/Mima>

Abbildung : 3 Grundphasen eines Compilers

Abbildung 2: Syntaxbaum



Abbildung 3: IL-Code



Abbildung 4: Popup



Abbildung 5: Syntax Highlighting



Abbildung 6: Debugging

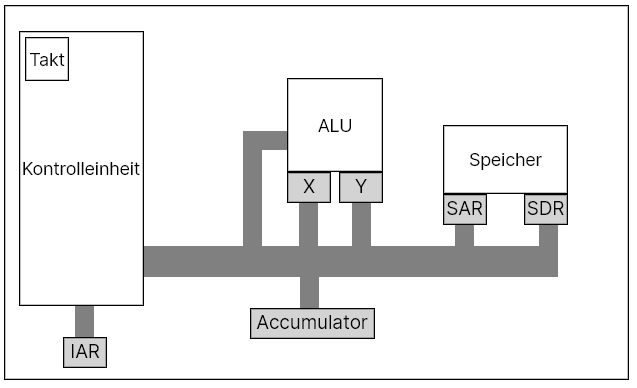


Abbildung 7: Minimalmaschine

Selbstständigkeitserklärung

Hiermit bestätige ich, dass ich diese Arbeit selbstständig und mit gutem Gewissen verfasst habe. Ich versichere, dass ich schriftliche Übernahmen aus anderen Quellen gekennzeichnet habe.

Ort, Datum Chris Marco Anders