Projektarbeit

Geschichte der Programmierung -

und wie sich diese durch die Computerchip-Entwicklung verändert hat

von Chris Anders

Klasse: 1BK1T

Betreuer: Andreas Stradinger

Inhaltsverzeichnis

Anfänge der Programmierung

Obwohl Ada Lovelace als erste Programmiererin bezeichnet wird, gab es zu ihrer Lebzeit keinen funktionieren Computer, aber den ersten programmierbaren Webstuhl. Die erste standardisierte Hochsprache wurde nach ihr benannt.

Grace Hopper erfand den ersten Compiler und die daraus resultierende Programmiersprache COBOL.

In den 1950er Jahren wurde die erste weitverbreitete höhere Programmiersprache Fortran entwickelt, die heute immernoch Verwendung findet.

Heute genutzte Programmiersprachen wurden stark von den ersten Sprachen beeinflusst, so findet in Common Lisp erstmals implementierte Verzweigungen heute noch rege Verwendung. Erste Konzepte von Klassen, also einen Bauplan oder eine Vorlage für ein Objekt, wurde durch C++ verbreitet.

Immer mehr Programmiersprachen kristalisierten sich aus den ersten Programmiersprachen heraus. So gibt es im grunde 2 Sprachfamilien für höhere Programmiersprachen. Die Basic- und die C-inspirierte Sprachen. Basic verwendet eher Schlüsselwörter und C eine Kombination aus Schlüsselwörtern und Sonderzeichen. Oft entstehen Probleme beim Wechsel einer Programmiersprache aus der Basic-Sprache zu einer C- Sprache und vice versa, da beide Familien unterschiedliche syntaktische und semantische Ansätze haben. Ein Wechsel in der gleichen Sprachfamilie stellt meist weniger ein Problem dar.

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. 1).

Der Taktgeber sendet zyklisch Signale über Datenbusse. Die Kontrolleinheit dekodiert Programmbefehle und sendet Daten und Befehle an andere Komponenten weiter. Register speichern einzelne numerischen Werte. 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.

Das dazugehörige Programm findet sich unter "https://snc.gbs-sha.de/f/43393971". Dieser Link enthält den Sourcecode, sowie ausführbare Dateien für Linux und Windows (32 und 64 Bit) und MacOS 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

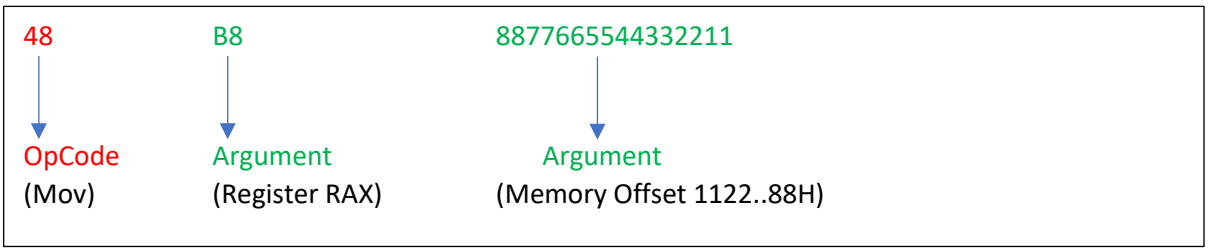
werden. Je nach Architektur ist die Größe und die Verteilung der Argumente unterschiedlich.

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 gibt an, dass diese eine Hexadezimalzahl ist)

Das Beispiel zeigt deutlich die Struktur einer Instruktion:



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

Es gibt verschiedene Ebenen von Programmiersprachen. Umso näher die Sprache an dem Maschinencode des Prozessors ist umso schneller läuft das Programm, aber sie ist auch für den Menschen unverständlicher. Je mehr Abstraktionsebenen eine Sprache besitzt, desto verständlicher ist sie für den Menschen, aber dafür muss man mit Performanceeinbußen rechnen.

Höhere Programmiersprachen werden durch einen Compiler entweder direkt in Maschinencode oder in einen Zwischencode übersetzt. Dies wird im darauffolgendem Unterkapitel erklärt. Es gibt 2 unterschiedliche Ansätze für Programmiersprachen. Die eine orientiert sich am Prozessor, ist also funktional. Die andere bilden Objektmodelle der Welt um Probleme besser beschreiben zu können. Diese Sprachen mit Objektmodellen werden Objektorientierte Programmiersprachen genannt. Wobei heutzutage sind die Grenzen bei den meisten Sprachen fließend. Man kann das eine Modell mit dem anderem simulieren oder es ist direkt in die Sprache eingebaut.

Die erste von Menschen lesbare Sprache war Fortran, die heutzutage immer noch Verwendung findet. Von dort an kristallisierten sich immer mehr Sprachen und Sprachfamilien heraus. Eine der bekanntesten ist die C Familie. Heutige häufig genutzte Sprachen sind C++, C#, Java, Python und JavaScript. Je mehr Aufgaben und Leistung ein Prozessor bekam, umso mächtiger und leistungsfähiger wurden die Sprachen.

Am Anfang mussten Programmierer*innen sehr viele Befehle auswendig lernen und diese auf ein Speichermedium wie zum Beispiel die Lochkarten übertragen. Heutige Programmierer*innen haben einen viel weniger großen Aufwand, um Programmieren zu können.

Compiler

Ein Compiler, von Englisch "to compile – zusammenführen", übersetzt Programmcode in für den Computer lesbare Sprache. Dabei besitzen alle Compiler 3-4 Grundphasen (Abb. 2).

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 (Abb. 3) wird zu Befehlen konvertiert, die der Prozessor oder die Laufzeitumgebung ausführen kann.

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

Maschinencode -> Zahlenblöcke werden zu Bytes konvertiert

Assembly -> Befehle werden zu einem Syntaxbaum konvertiert und in äquivalentem 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 sehr 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 ausgibt, könnte so wie in Abbildung 4 aussehen.

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. 5), 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. 6).

Die Fehlersuche in einem Programm wird durch das Debugging (Abb. 7) 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 und an größeren Projekten zu arbeiten. Erst nach und nach kamen immer mehr Features hinzu, die die Arbeit erleichterten.

Fazit

In den Anfängen der Programmierung war alles viel schwieriger. Es gab viel weniger Performance und Speicher für die Programme. Dadurch mussten sich Entwickler sich einiges einfallen lassen um Platz zu sparen. Damals gab es keine Unterstützung beim Entwickeln. Heute jeder sich eine oder mehrere Sprachen aussuchen und ihre Entwicklerwerkzeuge selbst wählen. Die meisten sind sehr stark im Umfang an Funktionen, aber manche sind einfach, aber effektiv. Es lohnt sich für jeden, egal wie alt er oder sie ist, sich mit dem Programmieren auseinander zu setzen. Denn heutzutage besitzt fast alles einen Prozessor bzw. ist auf dem Weg dorthin. Unsere Welt wird immer digitaler und nur wer sich in diesem Bereich ein bisschen auskennt, kann sicher und bewusst mit den heutigen Endgeräten und dem Internet umgehen.

Anhang

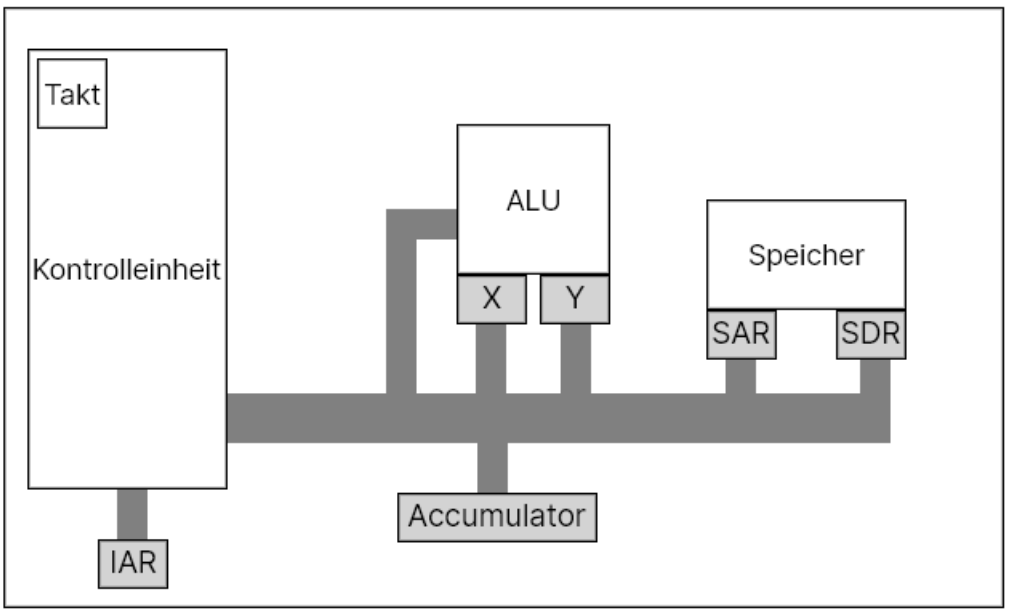


Abbildung 1: Minimalmaschine

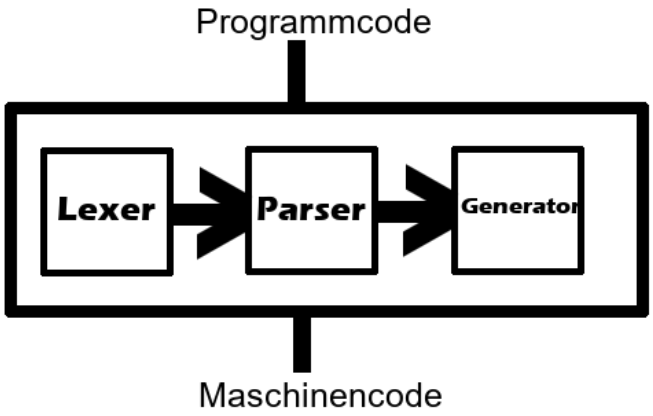


Abbildung 2: Compiler

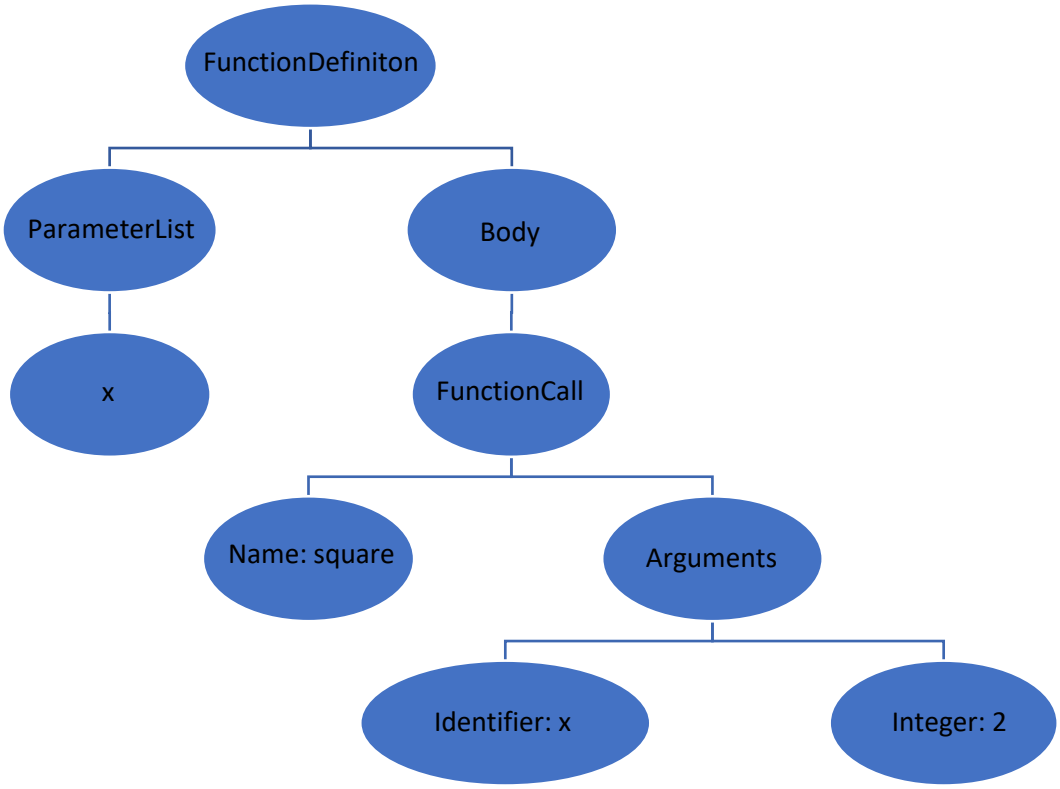


Abbildung 3: Syntaxbaum

```
.class private auto ansi beforefieldinit TestApplication.Program
    extends [System.Runtime]System.Object
{
    // Methods
    .method private hidebysig static
        void Main (
            string[] args
        ) cil managed
    {
        // Method begins at RVA 0x2050
        // Code size 13 (0xd)
        .maxstack 8
        .entrypoint

        // {
        IL_0000: nop
        // Console.WriteLine("Hello World!");
        IL_0001: ldstr "Hello World!"
        IL_0006: call void [System.Console]System.Console::WriteLine(string)
        // }
        IL_000b: nop
        IL_000c: ret
    } // end of method Program::Main

    .method public hidebysig specialname rtspecialname
        instance void .ctor () cil managed ...

} // end of class TestApplication.Program
```

Abbildung 4: IL-Code

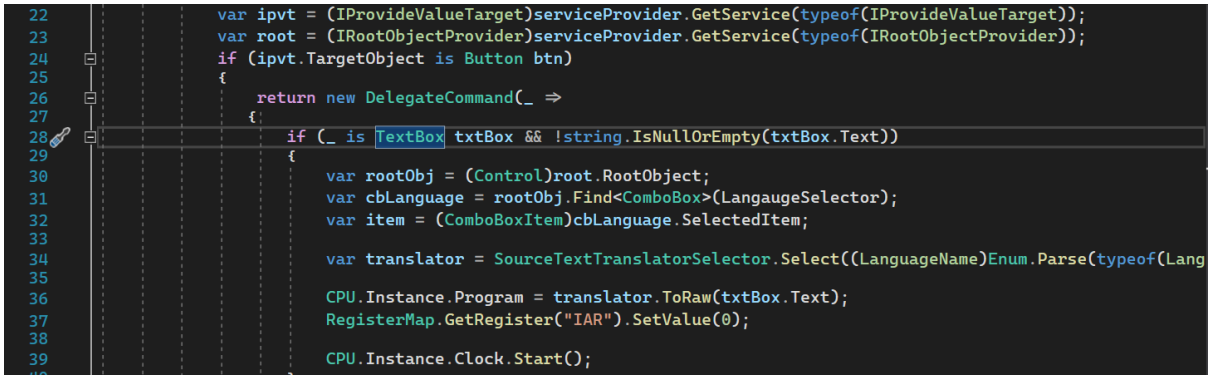


Abbildung 5: Syntax Highlighting

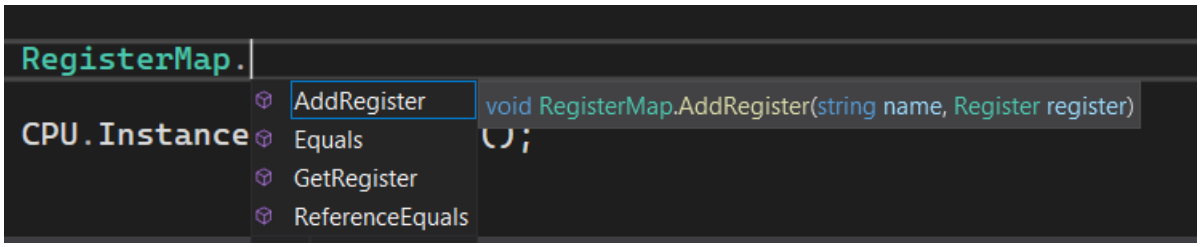


Abbildung 6: Popup mit Methodennamen

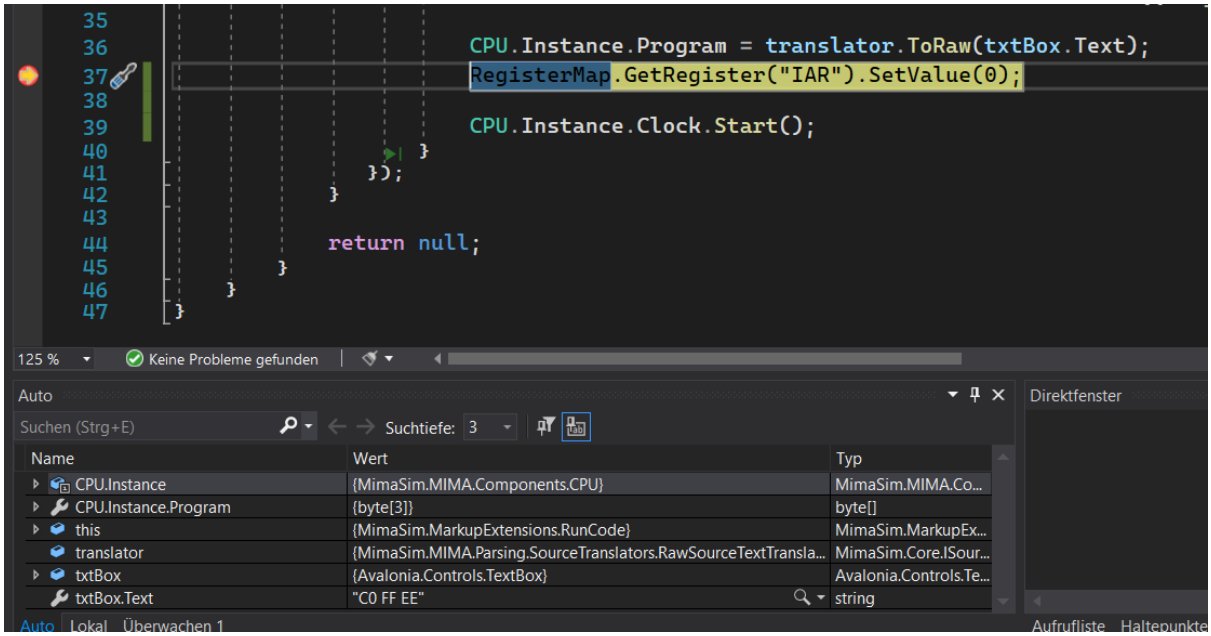


Abbildung 7: Debugging

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