Dokumentation PIC16F84 Simulator

Lukas Nonnenmacher & Jan Gerber

Inhalt

1.0 Grundlagen

Das Ziel dieses Projektes ist es, ein Programm zu erstellen welches die Funktion des Mikrokontrollers PIC16F84 simuliert. Die Programmiersprache und Entwicklungsumgebung sind frei wählbar. Eine Sammlung von auszuführenden Programmen liegt vor, diese enthalten die sogenannten Opcodes, dies sind Integer-Werte die von einem bestimmten Prozessortyp als Befehle interpretiert werden können. In unserem Fall entspricht es dem Befehlscode des PIC16F84. Es handelt sich dabei um einen Befehlssatz für ein RISC Prozessor. RISC steht für Reduced Instruction Set Computer. Der reduzierte Befehlssatz hat zum einen den Vorteil, dass die Befehle geringe Komplexität aufweisen und daher sehr schnell ausführbar sind, was eine höhere Taktfrequenz ermöglicht und zum anderen sind die

Assembler-Befehle für den Programmierer überschaubar. Unser Mikrokontroller unterscheidet lediglich fünfunddreißig Befehle.

1.1 Programmiersprache

Für die Umsetzung dieses Projektes haben wir C-Sharp als Programmiersprache gewählt. C-Sharp ist eine objektorientierte Programmiersprache. Bei einem Softwareprojekt dieser große helfen Objekte, strukturiert und übersichtlich zu programmieren. Außerdem ist C-Sharp Teil von Microsofts .NET, .NET kann als eine umfangreiche Ansammlung von Frameworks gesehen werden, unteranderem beinhaltet .NET die Windows Presentation Foundation, welche wir nutzen um Objekte auf der Oberfläche unserer Windows-Anwendung darzustellen.

1.2 Entwicklungsumgebung

Durch die Hochschule erhalten wir einen Microsoft Dreamspark Account, durch diesen haben wir Zugriff auf einen Großteil der Microsoft Tools, darunter auch die Entwicklungsumgebung Visual Studio 2015. Wir lernten dieses Produkt bereits in Programmieren 1 im ersten Semester kennen. Visual Studio gehört zu den wichtigsten Entwicklungsumgebungen überhaupt. Es bietet sowohl einen Editor für die Windows Presentation Foundation Benutzeroberfläche als auch ein Add-On für GIT, welches wir als Versions und Quellcodeverwaltung einsetzen.

1.3 Frameworks

Um dem Benutzer eine passende Benutzeroberfläche anzubieten nutzen wir Windows Presendation Foundation (kurz WPF). WPF ist Teil von .NET und eignet sich für den Einsatz mit C-Sharp. Vom Gesamtumfang den .NET bietet nutzen wir allerdings nur einen kleinen Teil. Mit WPF ist es ebenso möglich Web-Anwendungen zu entwickeln, wir beschränken uns jedoch auf eine reine Windows-Anwendung. Der direkte Aufbau der Oberfläche wird in XAML geschrieben. XAML steht für Extensible Application Markup Language, in dieser Sprache werden die Größe und Position sowie die Verweise auf die Inhalte der Oberflächenelemente angegeben.

Um die Verwendung von WPF zu vereinfachen und den XAML Code gering zu halten setzen wir ein zusätzlichen Framework ein: Caliburn Micro. Caliburn Micro ist ein kleines Framework, das als Bindeglied zwischen XAML als Oberflächensprache und unserem objektorientiertem Code in C-Sharp fungiert. Es gibt bestimmte Namenskonventionen vor, die es ermöglichen die Ein und Ausgabe über get und set-Methoden der Objektvariablen zu steuern. Insgesamt soll die Verwendung der Frameworks eine Model-VieModel-View Struktur ermöglichen.

2.0 Programmstruktur

2.1 Interpretation des Codes

Um die Befehle aus der Datei zu finden wird zunächst der Dateipfad benötigt. Dieser gibt der Benutzer mit Hilfe des OpenFileDialog Elements an. Das Element liefert einen String mit Dateinamen zurück. Danach wird ein selbst definiertes Objekt „ProgrammModel“ angelegt. An dessen Konstruktor wird der Dateiname übergeben. In dieser Klasse wird der Operationscode herausgefiltert und in ein Dictionary, welches zwei Integer Werte enthält, gespeichert. Im ersten Integer Wert wird die Zeilennummer gespeichert, im zweiten der Befehl inklusive der Argumente. Nun wird der Befehlsumwandler eingesetzt. Diese Klasse bekommt das Dictionary der Klasse „ProgrammModel“ übergeben. Hier wird ein neues Dictionary angelegt, dies soll die Zeilennummer im ersten Feld und den Befehl im zweiten Feld speichern. Ein Befehl besteht aus einem Objekt das entsprechend dem Befehl benannt ist, dem Programmcounter, und zwei Argumenten. Für Befehle mit nur einem Argument beleibt das zweite Argumentfeld leer. Um Festzustellen um welchen Befehl es sich handelt wird die Funktion „wandleBefehl“ aufgerufen. Diese vergleicht nur den Befehlsteil, nicht die Zeilennummer, mit einer Bitmaske, trifft die Bitmaske für einen Befehl zu, wird die entsprechende Funktion für jeweiligen Befehl aufgerufen.

```
if ((befehlOpcode & ADDLW) == ADDLW) {
                return newADDLW(key, befehlOpcode);
            }
```

Trifft keine der Bitmasken zu, wird eine Fehlermeldung ausgegeben.
Um mehr Übersicht zu behalten wurde der Bitwert als Integervariable definiert:

```
private int ADDWF = Convert.ToInt32("00011100000000", 2);
```

Die Aufteilung des Codes in Befehl und Argumente findet erst im jeweiligen Befehlsobjekt statt.

```
private BefehlViewModel newANDWF(int programmCounter, int befehlOpcode) {
            BitArray bitArray = new BitArray(new int[] { befehlOpcode });
             return new BefehlANDWF(programmCounter,(befehlOpcode &
             Convert.ToInt32("1111111", 2)), bitArray[7]);
}
```

Nach dem Drücken des „Datei öffnen“ Buttons und der Auswahl einer Datei besteht jetzt ein Dictionary: `Dictionary<int, BefehlViewModel> opcodesObjImpl = new Dictionary<int, BefehlViewModel>();`
Dieses enthält die Zeilennummern und die Objekte die dem jeweiligen Befehl entsprechen. Innerhalb eines Befehlsobjektes sind der Programmcounter und die Argumente für den Befehl gespeichert.

2.2 Ablauf des Simulationsvorganges

Der Simulationsvorgang kann gestartet werden, sobald alle Befehle eingelesen und interpretiert wurden. Zum starten des Simulationsvorgangs stehen auf der Benutzeroberfläche zwei Buttons zur Verfügung. Der Button „Start“ führt alle Befehle automatisch nacheinander aus, der Button „Schritt vorwärts“ zählt den Programmcounter jeweils um einen Schritt hoch, führt als Befehl für Befehl durch. Dieser Modus kann von Nutzen sein, falls der Benutzer nachvollziehen möchte was bei den einzelnen Befehlen geschieht. Wird einer dieser Buttons gedrückt, so wird die entsprechende Funktion in der Klasse „MainViewModel“ aufgerufen. Darin wird überprüft ob die Befehle bereits gespeichert sind, trifft dies zu wird der Speicher auf null gesetzt. Danach wird ein neuer Thread gestartet, das Verwenden von Threads ist erforderlich da das Programm sonst nur eine Aufgabe zeitgleich ausführen kann, dies hätte zur Folge das keine Benutzereingaben mehr entgegengenommen werden sobald das Programm gestartet wurde. Bei Programmen mit endlosschleifen, was bei Mikrocontrollerprogrammen häufig der Fall ist, bestände somit auch nicht mehr die Möglichkeit das Programm zu beenden. Innerhalb des „worker_StartProgrammThread“ wird überprüft ob ein Breakpoint gesetzt ist, Stopp gedrückt wurde oder ein Interrupt stattgefunden hat. Trifft keiner dieser Fälle zu, wird der Befehl ausgeführt auf den der Programmcounter verweist. Jeder Befehl ist gleich aufgebaut, mit dem Unterschied das es Befehle mit einem oder zwei Parametern gibt. Er enthält einen Konstruktor, get Methoden für die Argumente und eine Funktion „ausführen“. In dieser Funktion wird nur der Befehl tatsächlich ausgeführt. Neben der eigentlichen Ausführung des Befehls werden die entsprechenden Flags gesetzt, zum Beispiel das Carry-Bit, die Programmlaufzeit um den entsprechenden Wert erhöht, also ein oder zwei Zyklen, und der Programmcounter wird erhöht.

2.3 Interrupts

Der Mikrokontroller PIC16F84A unterscheidet zwischen vier Interrupts, je nach Interrupt werden im INTCON Registern verschiedene Bits gesetzt. In unserem Programm wird, genau wie beim PIC, zuerst nur geprüft ob ein Interrupt stattgefunden hat. Welcher Interrupt ausgelöst wurde muss vom Programmierer Softwaretechnisch überprüft werden. Für jeden Interrupt besteht in unserem Simulator eine eigene Funktion welche aufgerufen wird und die entsprechenden Bits setzt. Im Folgenden werden die Eigenschaften jedes Interrupt beschrieben und wie wir dies in unserem Programm umgesetzt haben.

- PortB Interrupt:
  Wird ausgelöst durch PortB<7-4>, setzt INTCON<0>, wird enabled durch INTCON<3>

```
private void interruptPortB() {
            speicher.setRegister(0x0B, 0, true); //set INTCON<0>).
            if (speicher.getRegister(0x0B,7) &&
speicher.getRegister(0x0B, 3)) { //GIE  && INTCON<3>
                speicher.Interrupt = true;
                speicher.setRegister(0x0B, 7, false); //clear GIE
            }
}
```

- External Interrupt RB0/INT pin:
  Wird ausgelöst durch RB0, wird enabled durch INTCON<4>, setzt INTCON<1>, in Option_REG<6> wird festgelegt ob die steigende oder fallende Flanke erkannt wird.

```
private void interruptINT() { //TODO INT Interrupt
            if (speicher.getRegisterOhneBank(0x81, 6)) { // INTEDG bit
(OPTION_REG<6>)
                if (speicher.getRegisterOhneBank(6,0)) { // if rising
edge
                    speicher.setRegister(0x0B, 1, true); //INTFbit(INTCON
< 1 >)
                    if (speicher.getRegister(0x0B, 7) &&
speicher.getRegister(0x0B, 4)) { //GIE  && INTCON<4>
                        speicher.Interrupt = true;
                        speicher.setRegister(0x0B, 7, false); //clear GIE
                    }
                }
            } else {
                if (!speicher.getRegisterOhneBank(6, 0)) { // if falling
edge
                    speicher.setRegister(0x0B, 1, true);//INTF bit(INTCON
< 1 >)
                    if (speicher.getRegister(0x0B, 7) &&
speicher.getRegister(0x0B, 4)) { //GIE  && INTCON<4>
                        speicher.Interrupt = true;
                        speicher.setRegister(0x0B, 7, false); //clear GIE
                    }
                }
            }
 }
```

- TMR0 overflow interrupt:
  Wird ausgelöst durch overflow von TMR0, setzt INTCON<2>, wird enabled durch INTCON<5>

```
private void erhoeheTimer0counter() {
            if (Register[1] == 255) {
                Register[1]++;
                //setzte Interrupt Flag
                setRegister(0x0B, 2, true); //Overflow sets bit
T0IF(INTCON < 2 >).
                if (getRegister(0x0B, 5) && getRegister(0x0B, 5)) {
//Ueberpruefe ob GIE AND INTCON<5>
                    Interrupt = true;
                    setRegister(0x0B, 7, false); //clear GIE
                }
            } else {
                Register[1]++;
            }
}
```

2.5 Programmlaufzeit

Die Simulation der Programmlaufzeit ist relativ einfach umgesetzt. Jeder Befehl der von einem Mikrocontroller ausgeführt wird, benötigt definiert viele Zyklen, dieser werden im Simulator immer bei der Ausführung des entsprechenden Befehl um die im Datenblatt angegebene Zykluszeit erhöht. Dieser Wert wird mit der eingestellten Frequenz multipliziert und auf der Benutzeroberfläche ausgegeben.

2.6 Oberfläche

Die Oberfläche ist komplett in XAML umgesetzt. Der Code ist in einer einzigen Datei „MainView.XAML“ zusammengefasst.

Zur Steuerung des Programms werden dem Benutzer fünf Buttons zur Verfügung gestellt, im Folgenden wird deren Funktion einzeln erklärt:

- Datei öffnen: Dies ist der erste Button den der Nutzer betätigt, er öffnet den Windows Explorer. Durch eine Voreinstellung werden bereits nur Dateien mit der Endung .LST angezeigt. Hat der Benutzer eine Datei ausgewählt, schließt sich das Explorerfenster automatisch.
- Start: Dieser Button verursacht den automatischen Durchlauf durch den Programmcode. Alle Befehle werden solange abgearbeitet bis das Programm zu Ende ist oder der Benutzer das Programm unterbricht. Beim Drücken des Stopp-Buttons wird der aktuelle Befehl fertig bearbeitet.
- Stop: Dieser Button unterbricht den automatischen Durchlauf durch den Programmcode. Nach dem Drücken des Buttons bleibt der Programmcounter gespeichert, das Programm kann also von der aktuellen Stelle fortgesetzt werden.
- Nächster Schritt: Mit Hilfe dieses Buttons kann der Benutzer Befehl für Befehl durch den Assemblercode gehen. Es wird immer ein Befehl ausgeführt und dann auf eine neue Benutzereingabe gewartet.

Für die Ausgabe von Speicherwerten stehen verschiedene Felder zur Verfügung. Im Folgenden wird die Ausgabe jedes Feldes beschrieben:

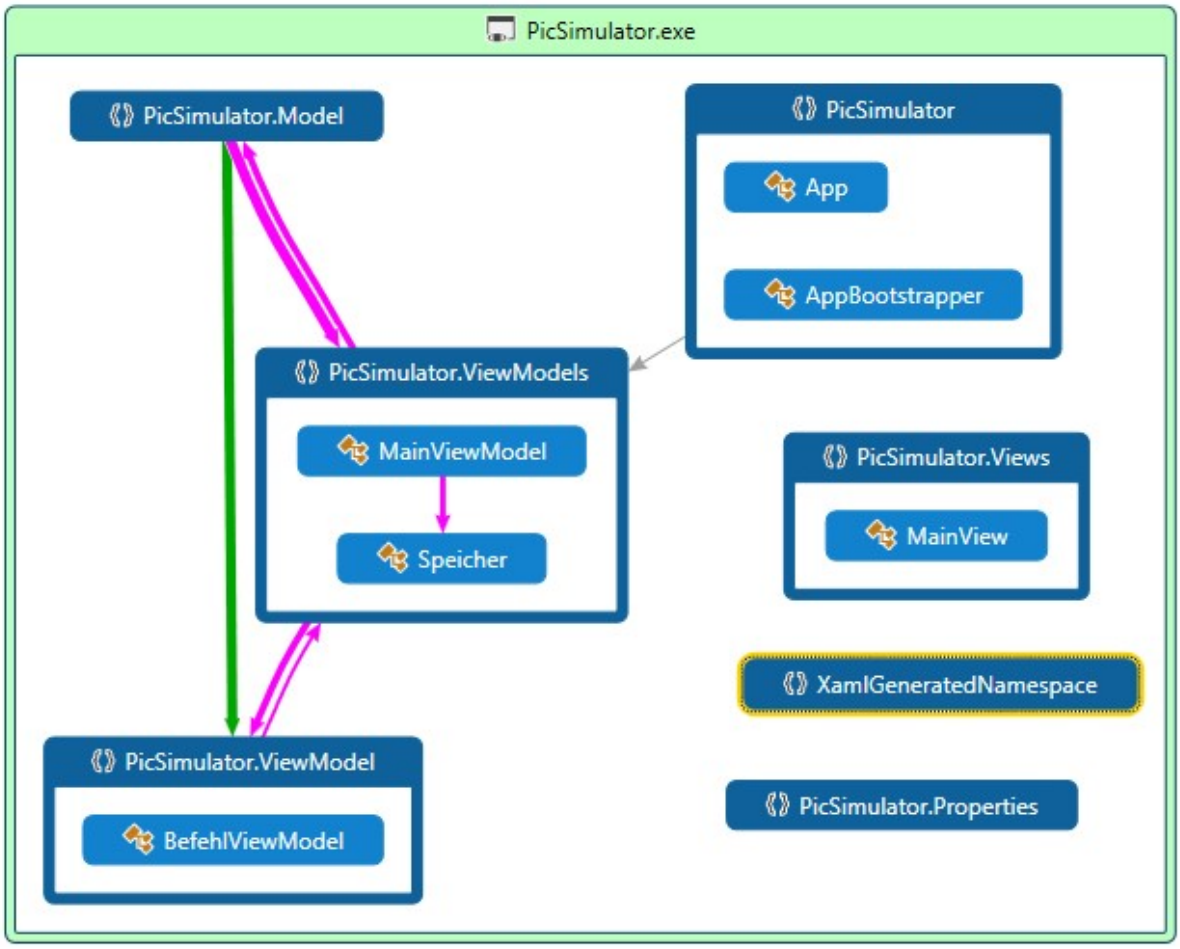
- ListView Befehlsliste: Dieser Element gibt Zeilenweise die Befehle aus, welche aus der Quelldatei gelesen wurden. Ausgegeben werden der Befehlsname, der Wert der beiden Argumente und die Zeilennummer. Am Anfang jeder Zeile befindet sich eine Checkbox, diese kann als Breakpoint genutzt werden. So wird es dem Benutzer ermöglicht das Programm an einer bestimmten Stelle anzuhalten. Zusätzlich zeigt dieser Listview an, welcher Befehl im Moment ausgeführt wird, indem er die Zeile grau einfärbt.
- Grid Spezialregister: In diesem Anzeigeelement wird der Wert der Spezialregister angezeigt, dazu gehören: PCL, Status Reg, Port A, PortB, INTCON und das W-Reg. Die Werte dieser Register werden Hexadezimal ausgegeben. Die Umwandlung von Integer in Hexadezimal findet erst in XAML, mit Hilfe der Funktion „StringFormat“ statt.
- Grid RAM: Der RAM wird in einer Tabelle dargestellt, jedes Byte wird in einer eigenen Zelle beschrieben. Der Wert des Bytes wird als Hexadezimalzahl dargestellt. Eine Tabellenzeile enthält 8 Byte, im ersten Feld der Tabelle ist durch die Adresse beschrieben, welche Bytes ausgegeben werden.
- ListView IO-Ports: Port A und Port B können von der Benutzeroberfläche aus editiert werden, jedes einzelne Bit wird in einer eigenen Checkbox dargestellt. Die

Checkboxen befinden sich in einem Gridview, welcher sich wiederum in einem Listview befindet. Ob der Benutzer den Wert der Checkbox ändern kann oder nicht, hängt von den Werten in TRIS A und TRIS B ab. Diese können nur innerhalb des Programmcodes geändert werden.

- ListView Stack: Der Stack wird nicht wie der RAM als Tabelle, sondern als Liste ausgegeben. Wird ein neuer Eintrag auf dem Stack angelegt, wird die Liste um einen Eintrag erweitert.
- Grid Cycles: Diese Tabelle enthält nur zwei Felder, eines Beschreibt das Feld, nämlich die Ausgabe der durchlaufenen Programmzyklen, im zweiten Feld werde diese ausgegeben.

2.7 Klassendiagramme

Insgesamt haben wir uns an das Model – ViewModel – View Framework gehalten, das folgende Klassendiagramm zeigt die Übersicht sehr deutlich.

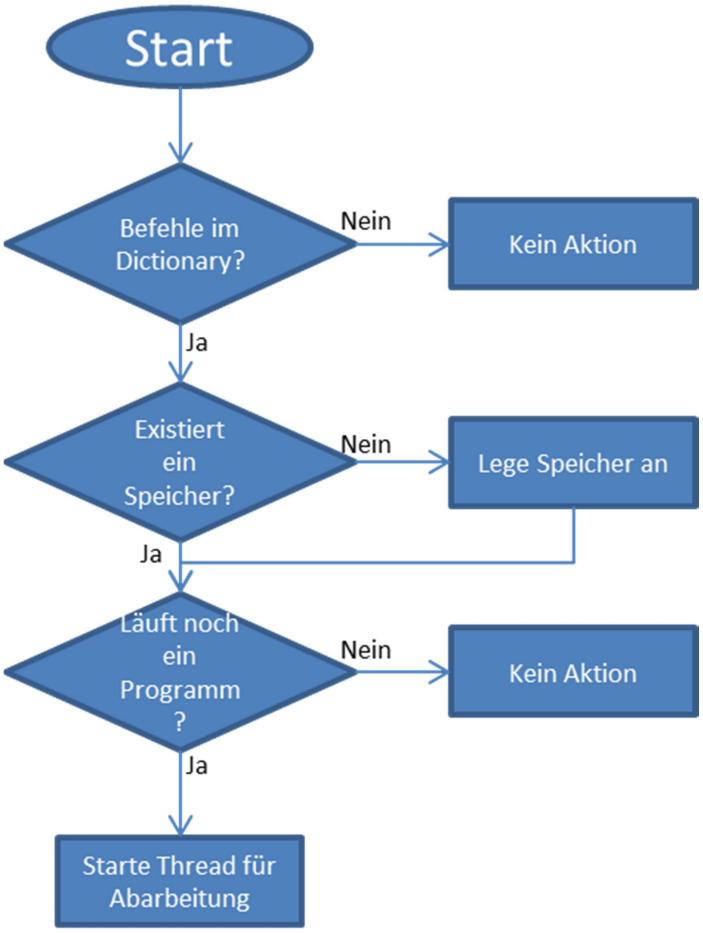


Im Model selbst wurde für jeden Befehl eine eigene Klasse angelegt.

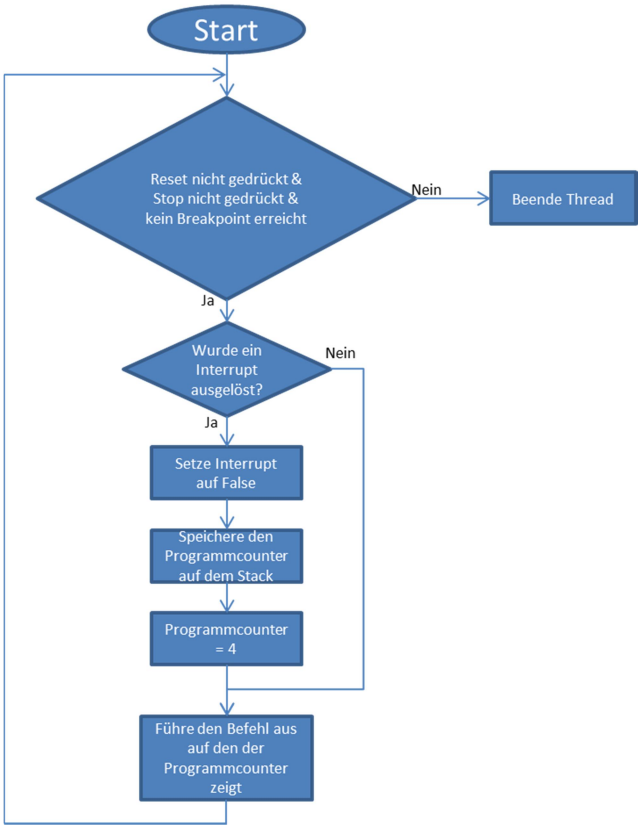
3.0 Ablaufdiagramme

3.1 Drücken des Startbuttons

Voraussetzung: die Datei wurde eingelesen, Dictionary mit Positionen und Befehlen ist befüllt, der Start Button wurde gedrückt.



3.2 Automatischer Programmdurchlauf



4.0 Organisatorisches

4.1 Versions und Quellcodeverwaltung

Zur Versionskontrolle und Quellcodeverwaltung verwenden wir GIT. Visual Studio bietet dafür eine Integration, direkt in der IDE an. Diese ermöglicht es, Änderungen zu Comitten und mit einer Nachricht zu versehen. Es ist so möglich das alle Teammitglieder gleichzeitig am gleichen Projekt arbeiten. Verändern beide den gleichen Code so bekommt man beim Synchronisieren einen Fehler, man muss nun die beiden Versionen der Datei einander gegenüberstellen und sich für eine Lösung entscheiden, dies kann auch zeilenweiße geschehen. Wir haben uns für GIT entschieden da wir bereits in unserem Software Engeneering Projekt damit arbeiten.

4.3 Fazit

Zu Beginn des Projektes mussten wir eine Programmiersprache auswählen, mit C++ und Java hatten wir bereits Erfahrungen gemacht. Trotzdem entschieden wir uns dafür C-Sharp zu verwenden. C-Sharp bietet den Vorteil, dass es Teil von .NET ist und somit viele Anbindungsmöglichkeiten bietet. Am Anfang beschäftigten wir uns hauptsächlich damit, die Besonderheiten von C-Sharp zu verstehen und ein Grundlegendes Konzept auszudenken. Die Besonderheit an unserem Klassendiagramm besteht darin, das jeder Befehl in einer eigenen Klasse umgesetzt ist. Als das Grundgerüst für unser Programm stand, konnten wir sehr schnell voranschreiten. Das Umsetzen der Assembler-Befehlslogik bereitete uns keine Schwierigkeiten. Dafür traten beim Ansprechen der Benutzeroberfläche die Probleme auf,

wir verbrachten viele Stunden damit die Objekte im Programm mit denen auf der Oberfläche zu verbinden. Neu war für uns auch das Arbeiten mit verschiedenen Threads. Als wir diese noch nicht eingesetzt hatten, blockierte das laufende Programm den Eingriff in die Benutzeroberfläche. Nach kurzer Einarbeitungszeit waren wir in der Lage bei drücken des Start Buttons einen neuen Thread zu eröffnen, dies beseitigte die Probleme die wir zuvor hatten. Durch die Nutzung von Microsoft Visual Studio und GIT konnten wir die Aufgaben sehr bald klar verteilen und getrennt Zuhause an dem Projekt arbeiten. Durch Internetrecherche und den gelegentlichen Austausch mit anderen Gruppen kamen wir insgesamt gut voran. Durch das Projekt konnten wir unser Wissen im Thema Assembler und Objektorientiertem Programmieren noch einmal vertiefen, neu dazu kam der Umgang mit C-Sharp und WPF, auch wenn die Anwendung von letzterem sehr mühselig war. Mit unserem Ergebnis sind wir sehr zufrieden.

Jan Gerber & Lukas Nonnenmacher