Dokumentation PIC16F84 Simulator

Studiengang:

Angewandte Informatik

Semester:

2. Semester

Praktikum:

Rechnerarchitekturen

Dozent:

Dipl.-Ing. Stefan Lehmann

Autoren:

Noah Disch

Eduard Wayz

1 Einleitung

In dem vorliegenden Dokument wird die Entwicklung und Implementierung eines Simulators in Java für den Mikrocontroller PIC16F84 dargestellt. Der Fokus liegt hierbei auf den wesentlichen Funktionen und Eigenschaften des Simulators sowie den gewonnenen Erkenntnissen aus der Programmierung. Zudem werden die Herausforderungen aufgezeigt, denen wir bei der Entwicklung des Simulators begegnet sind.

2 Allgemeines

2.1 Funktionsweise eines Simulators

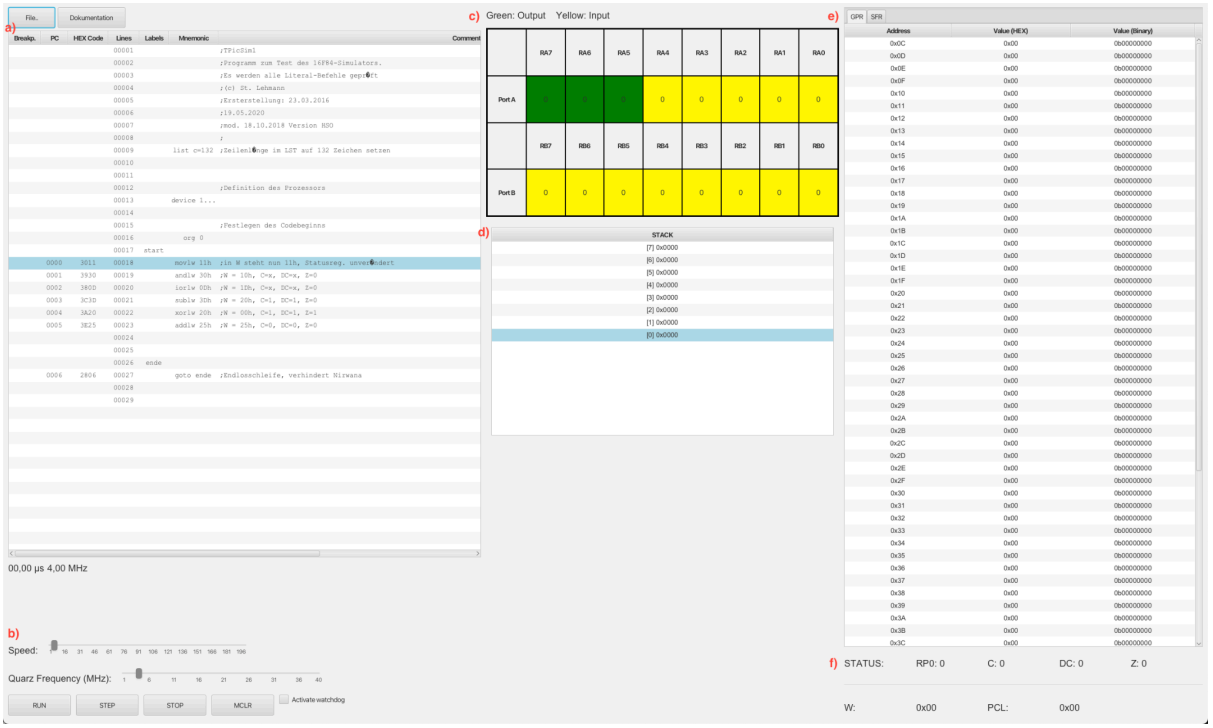
Ein Simulator simuliert, wie der Name schon sagt, etwas. In unserem Fall ist es ein Mikrocontroller, genauer der PIC16F84. Der Simulator bietet die Möglichkeit, eine LST-Datei einzuspielen, welche dann ausgeführt werden kann. Er kennt die Befehle, welcher auch der physische PIC kennt und enthält dieselben Speicher, zumindest in nachgebildeter Form. Die Werte in den Registern sowie Eigenschaften des PICs können direkt eingelesen werden. Der PIC lässt sich einfach zurücksetzen und die Simulation erfüllt den Zweck, geschriebene LST-Dateien zu testen und zu verstehen.

2.2 Vor- und Nachteile der Simulation

Da es sich hierbei um einen Simulator handelt, lohnt es sich, sich damit auseinanderzusetzen, was für einen solchen Simulator im Vergleich zu einem physischen PIC16F84 spricht und welche Nachteile er aber auch mit sich bringt. In diesem Abschnitt werden wir das hier mit einigen wenigen zugrunde liegenden Punkten erläutern.
Angefangen mit dem Punkt, dass er deutlich einfacher zu benutzen und nachzuvollziehen ist als ein physischer PIC. Gemeint ist damit, dass der Simulator direkt gestartet werden kann und auch die veränderten Werte direkt in den entsprechenden Abschnitten nachgeschaut werden können. Zudem liegt der Vorteil in einem noch ausführlicheren Simulator, dass die Werte sehr leicht zu manipulieren sind und der Nutzer dadurch eine große Anzahl an Einstellungsmöglichkeiten hat. Dies bietet ihm eine hohe Flexibilität und ist zum Testen optimal.
Ein Nachteil ist jedoch, dass physische Phänomene oder Störungen nur sehr schwer bis gar nicht zu simulieren sind. Einerseits bringt das den Vorteil, dass der Nutzer erkennt, ob sein Programm fehlerfrei ist, andererseits ist das Ergebnis in der Nutzung eines physischen PICs dadurch teilweise ein anderes. Hierbei muss abgewägt werden, wie schwerwiegend und häufig solche physischen Phänomene oder Störungen in der Praxis vorkommen, um ein Urteil darüber zu fällen, ob ein Simulator genutzt werden kann oder nicht. Das ist ein Kriterium, ob ein solcher Simulator zum Beispiel in Unternehmen eingesetzt werden kann. Denn die Ähnlichkeit zu dem physischen PIC ist der entscheidende Punkt an dieser Stelle.

2.3 Beschreibung der Programmoberfläche

Folgendes Bild zeigt die Simulation in ihrer Ausgangsansicht. Sie lässt sich in die sechs verschiedenen Bereiche “a” bis “f” gliedern, welche in diesem Kapitel noch erläutert werden.



a) Der Bereich “a” zeigt die LST Datei in einer tabellarischen Ansicht (TableView) an. Oberhalb der Tabelle befinden sich zwei Buttons: “File..” und “Dokumentation”. Mit “File..” lässt sich das Dateiauswahl-Menü des eigenen Betriebssystems (siehe Abbildung 1 im Anhang (zeigt es unter macOS)) anzeigen und der zweite Button öffnet die Dokumentation zu diesem Projekt (diese Datei).
Die Tabelle enthält sieben Spalten. Sechs dieser Spalten ("PC" bis “Comments”) sind unmittelbar aus der LST-Datei entnommen, während die Spalte “Breakpoint” ergänzt wurde, um dem Nutzer die Möglichkeit zu geben, Breakpoints zu setzen. (siehe Abbildung 2 im Anhang). Bei jedem Schritt wird der nächste auszuführende Befehl in hellblau markiert.
Unterhalb der Tabelle wird die Laufzeit in Mikrosekunden sowie die simulierte Quarz Frequenz in Megahertz angegeben.

b) Im Bereich “b” sind die primären Steuerelemente der Simulation untergebracht. Oberhalb der vier Knöpfe befinden sich Schieberegler, mit welchen sich die Schrittgeschwindigkeit im “Run”-Modus und die Quarzfrequenz einstellen lassen. Der beim Geschwindigkeitsregler eingestellte Wert gibt die Anzahl der in der Sekunde durchgeführten Schritte an.
Der oben bereits erwähnte „RUN“-Button startet die kontinuierliche Ausführung der Steps des PIC, die in einstellbaren Intervallen erfolgen. Der gesamte View wird bei jedem Schritt neu geladen, um die Werte aktuell zu halten, wobei das Vorhandensein von Breakpoints hier überprüft wird. Wird ein Breakpoint erreicht, stoppt die Simulation, bis erneut auf den

“RUN”-Button gedrückt wird.
Der „STEP“-Button führt einen einzigen Schritt aus und aktualisiert daraufhin den gesamten View. Breakpoints werden hier nicht berücksichtigt und daher einfach übersprungen.
Der Button “STOP” bewirkt das sofortige Beenden der “Run-Funktion”.
Der Knopf “MCLR” steht für “Master-Clear” und bewirkt einen Master-Clear des PIC Bausteins. Dieser kann physisch über einen Low-aktiven Pin ausgelöst werden. Hierbei wird der Programmzähler auf 0 gesetzt. Die meisten anderen Register bleiben dabei unverändert.
Ganz rechts befindet sich die Checkbox für den Watchdog, welcher hiermit aktiviert oder deaktiviert werden kann. Im physischen Baustein kann dies nur bei der Programmierung des Bausteins erfolgen, da man zur Laufzeit keinen Zugriff auf diese Einstellung hat.

c) Im nächsten Bereich, dem Bereich “c”, werden die Ports “Port A” und “Port B” des PICs mithilfe eines GridViews visualisiert. Die Farben zeigen an, ob der Port im Moment ein “Eingang” (“Input”) oder ein “Ausgang” (“Output”) ist, was durch das TRIS-Register definiert wird. Ein Ausgang lässt sich nicht manuell verändern, während die Eingänge durch einen Mausklick umschaltbar (0 -> 1 oder 1 -> 0) sind. Wobei im nächsten Schritt dann der Wert in den jeweiligen Port übernommen wird. Die TRIS-Register lassen sich nicht manuell einstellen.
d) Direkt unterhalb der Port-Darstellung befindet sich die Visualisierung des Stacks, Bereich “d”, welcher (gemäß der typischen Funktionsweise eines Stacks) von unten nach oben gefüllt wird. Der Stackpointer, welcher immer den aktuell “höchsten” Index angibt, wird, analog zur LST-Tabelle, ebenfalls durch hellblaue Markierung visualisiert. Sodass der Nutzer sieht, an welcher Stelle der Stack gerade verändert wird.
e) In einem weiteren Teilbereich werden die General Purpose Registers (GPR) und Special Function Registers (SFR) in einer Tab-Ansicht dargestellt. Dieser ist aber auch wieder ein TableView. Beide Tabs enthalten die Spalten „Address", „Value (Hex)“ und „Value (Binary)“. Zusätzlich enthält der SFR-Tab die Spalte „Name“, welche als Alias für die Adressen dient.
f) Zum Schluss werden im Bereich “f” noch einmal die wichtigsten Werte während des Programms angezeigt. Es ist zwar eine Wiederholung, von dem, was auch in den anderen Bereichen sichtbar ist, aber dennoch sehr hilfreich, da das die Werte sind, die am meisten während des Programms verändert werden.

Alle dargestellten Bereiche sind miteinander verknüpft und ermöglichen so eine kontinuierliche Aktualisierung der Benutzeroberfläche während der Simulation.

3 Realisierung

3.1 Auswahl des Moduls für das Frontend

Bei der Frage um das Konstruieren der GUI hatten wir die Wahl zwischen verschiedenen Modulen, die wir in Java hätten verwenden können. Am Ende haben wir uns dazu entschlossen, die GUI mit dem sehr bekannten Tool JavaFX zu bauen. Diese Entscheidung vereinfachte vieles und brachte aber auch so seine Herausforderungen mit sich. Darauf wird aber nochmal in Kapitel 5 näher eingegangen.

3.2 Grundkonzept des Simulators

Der Simulator funktioniert grundsätzlich, so dass man mithilfe des “File..”-Button eine der vorgegebenen Testprogramme auswählt. Diese Datei wird dann im Kapitel 3.3 beschriebenen Bereich “a” angezeigt und ein komplett neuer PIC wird geladen. Zu Beginn lässt sich die Simulation hauptsächlich dadurch “bewegen”, dass die Knöpfe “RUN” oder “STEP” ausgeführt werden, auch wenn alle anderen Funktionen hier schon funktionstüchtig sind. Jeder Schritt (Step) führt den nächsten Befehl aus, indem dieser aus dem Programm “gefetched” wird; also in den Befehlsdecoder geladen wird. Anschließend wird dieser decodiert und ausgeführt . Nachdem der Befehl ausgeführt wurde, wird die Ansicht neu geladen, was bedeutet, dass die UI-Elemente neu befüllt werden. So wird dafür gesorgt, dass der User bei jedem Schritt genauestens nachvollziehen kann, was der jeweilige Schritt ausgeführt hat. Die Buttons “STOP” und “MCLR” wurden bereits im Abschnitt 2.3 erläutert.

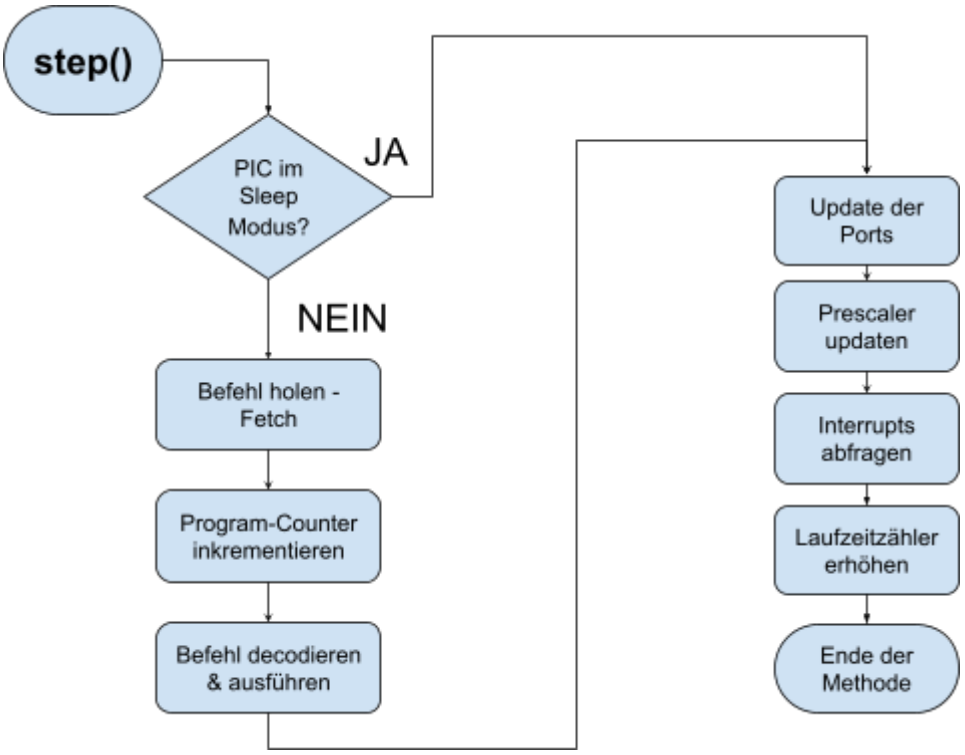
3.3 Struktur und Gliederung

Das Projekt ist in mehrere Ordner/Packages gegliedert, um Module übersichtlich zu kapseln. Der “Oberordner”, die “root”, ist der Ordner "Rechnerarchitektur", welcher als Sammelordner der folgenden Ordner fungiert. Hier sind zum einen die Ordner für die JavaFX-Implementation drin (1x für Windows, 1x für Mac) und zum anderen befindet sich hier auch der nächste große Ordner “src”, auf den gleich noch einmal tiefer eingegangen wird. Zuvor möchte ich noch den Ordner “test_files” erwähnen, welcher auch ein direkter Unterordner von “Rechnerarchitektur” ist. Hier sind alle von der lehrenden Person zur Verfügung gestellten Test-Dateien, sowie eine dazugehörige ReadMe PDF, enthalten.
Des Weiteren möchte ich tiefer auf den “src”-Ordner eingehen. Dieser enthält alle Klassen und alle weiteren Dateien, die für das Funktionieren der Simulation essentiell sind. Dieser Ordner ist gegliedert in weitere drei Unterordner. Angefangen mit dem kleinsten dieser Ordner, welche auch Packages sind: “junit_tests”. Dieser Ordner war ursprünglich für Unit-Tests gedacht, die die ganzen Test-Dateien auf Knopfdruck testen können. Die Änderungen in der GUI lassen sich mit ihnen nicht testen, aber zumindest die Werte in den Registern. Wir haben zwar ein paar Unit Test Programme erstellt und doch war der Aufwand so groß, dass wir es nach ein paar Tests aufgehört haben und uns weiter auf den restlichen Code konzentriert haben.
Der zweitgrößte Ordner ist der “resources”-Ordner. Dieser Ordner enthält alle .fxml-Dateien, welche für das ganze GUI notwendig sind und auch die zugehörigen .css-Dateien.

Zusätzlich befindet sich auch dieses Dokument als PDF in diesem Ordner. (siehe Abbildung 3)
Der größte Unterordner ist “main”. Dieser enthält die ganze Logik des PICs und der verschiedenen .fxml-Dateien. Die einzelnen Klassen und Assoziationen werden im nächsten Abschnitt genauer erläutert.

3.4 Programmstruktur und Ablaufdiagramme

Der Simulator besteht hauptsächlich aus der PIC Klasse, in der sich die meiste Logik befindet. Dort werden die Befehle decodiert und ausgeführt. Diese Klasse hat zusätzlich eine Instanz der RAM-Klasse, welche für die Speicherung von Daten und Variablen des PICs zuständig ist. Diese enthält zwei Instanzen der Bank-Klasse, die eine Wrapper-Klasse mit Lese- und Schreibmethoden für ein Integer Array der Länge 128 ist. Alle Klassen für Speicherung sind in einem Package namens “cardgame”. Dies ist eine Anspielung auf das Kartenspiel “Memory”, bei dem man sich Bilder merken muss. Hinzu kommen zwei Instanzen für die beiden Port-Register beziehungsweise deren Pins. Des Weiteren wurden der Prescaler, sowie Timer und Watchdog in eigene Klasse ausgelagert, um die Hauptklasse - die PIC Klasse - so gut es geht, klein zu halten und zusätzlich in ein eigenes Package “timers” für mehr Übersicht.
Wir haben uns dazu entschieden, die vielen Konstanten und Masken, die benötigt werden, in extra Dateien zu machen, um die Übersichtlichkeit und Lesbarkeit zu erhöhen. Dazu mehr in dem nächsten Abschnitt. Diese sind ebenfalls in einem eigenen Package “libraries”.
Ebenfalls wurden zwei Klassen genutzt, um die LST-Datei einzulesen und die benötigten Informationen herauszufiltern, um schlussendlich den PIC Baustein simulieren zu können, der dieses Programm ausführt.
Im Folgenden wird die wahrscheinlich wichtigste Methode des Projekts als Ablaufdiagramm, aber auch als Code-Snippet vorgestellt: die step()-Methode. Diese Methode simuliert, einen Befehlszyklus des PICs, bei dem ein Befehl aus dem Code-Speicher beziehungsweise der LST-Datei herausgeholt wird, decodiert wird, einem Befehl zugeordnet wird und schlussendlich der passende Befehl ausgeführt wird. Daraufhin werden sowohl die Ports als auch der Prescaler geupdatet und Interrupts abgefragt.



3.5 Beschreibung der verwendeten Konstanten

Als erstes ist zu erwähnen, dass sich dagegen entschieden wurde, den Standard für die Benennung von Variablen und Konstanten in Java einzuhalten, um sich an die Namensgebung im Datenblatt des PICs anzupassen.
Um den Code lesbarer zu machen, haben wir uns dazu entschieden, die meistgebrauchten Konstanten und Masken in Libraries zu speichern, welche dann als Import genutzt werden können. Diese Bibliotheken wurden in dem Package “libraries” gespeichert. Dieses wurde noch in ein weiteres unterteilt, nämlich "register_libraries". In diesem Package wurden für die wichtigsten Register, wie zum Beispiel Status oder Option, die Bits namentlich gespeichert. So musste man nicht die Position des Bits kennen, sondern nur dessen Namen. Hier ein Beispiel für den Simulator relevanten Bits aus dem Status Register, welche häufig gebraucht wurden. Wie bereits erwähnt, musste man nur den Namen, zum Beispiel Zero-Flag, wissen und nicht dass dieses Bit an der Position 2 im Status Register steht, was die Entwicklung enorm vereinfacht und beschleunigt hat und vor allem den Code wartbarer macht.

```
public static final int carry = 0;
public static final int digitcarry = 1;
public static final int zeroflag = 2;
public static final int powerdown = 3;
public static final int timeout = 4;
public static final int rp0 = 5;
```

Zudem war es oft notwendig, bestimmte Special Funktion Register zu beschreiben oder auszulesen beziehungsweise anzusprechen. Um auch hier nicht ständig im Datenblatt des PIC Bausteins nachlesen zu müssen, wurde sich auch hier für eine Library entschieden.

```
public static final int TMR0 = 1;
public static final int OPTION = 1;
```

```
public static final int PCL = 2;
public static final int STATUS = 3;
public static final int FSR = 4;
```

Außerdem war es häufig gefragt, während der Entwicklung und Implementierung des Simulators verschiedene Werte zu maskieren, um nur bestimmte ausgewählte Bits zu betrachten. Auch diese Masken wurden als statische Konstanten in einer Klasse gespeichert. Hier ein Auszug aus dieser Klasse mit verschiedenen Beispielen:

```
public static final int ADDRESS_MASK  = 0b0111_1111;
public static final int BIT_POS_MASK = 0b0011_1000_0000;

// masks for pc and pclath manipulation
// lowest 11 bits
public static final int GOTO_CALL_MASK = 0b0111_1111_1111;
public static final int PCLATH_4_3_MASK = 0b1_1000;
```

Teilweise doppelten sich die Werte der Masken, jedoch wurde diese Doppelung bewusst in Kauf genommen, um durch den unterschiedlichen Namen den Code an anderen Stellen lesbarer zu machen, obwohl beide Masken identisch sind. Ein Beispiel hierfür ist die Maske, um die acht niederwertigsten Bits zu maskieren.

```
public static final int LOWER8BIT_MASK = 0b1111_1111;
```

Diese Maske ist identisch mit der Maske, um ein Literal zu maskieren für die Literalbefehle, die der PIC besitzt.

```
// redundant to LOWER8BIT_MASK but added for readability
public static final int LITERAL_MASK  = 0b1111_1111;
```

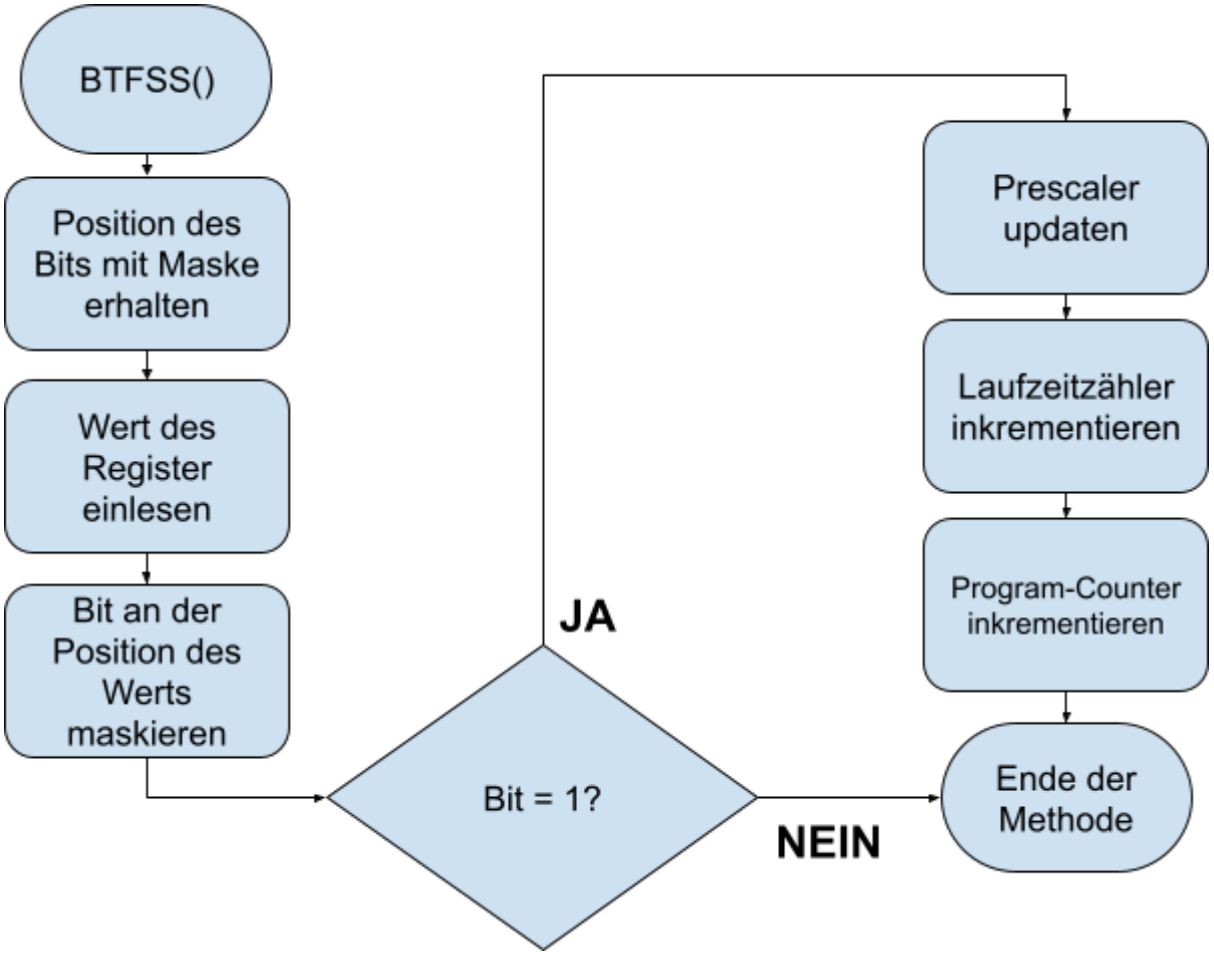
Zu guter Letzt war es essentiell, den eingelesen Befehl aus der LST-File zu decodieren, um zu erkennen, welcher Befehl als nächstes auszuführen ist. Jeder Befehl des PICs hat einen eigenen individuellen Hex-Code. Jedoch sind in diesem Fall die unteren Bits nicht zu betrachten, da diese in den meisten Fällen angeben, welche Adresse gelesen oder beschrieben werden soll. Somit waren hier auch über 30 Masken notwendig, die ebenfalls in eine Klasse ausgelagert wurden. Zu sehen sind die Masken der ersten vier Befehle des PICs.

```
public static final int ADDWF =  0x0700;
public static final int ANDWF =  0x0500;
public static final int CLRF =  0x0180;
public static final int CLRW =  0x0100;
```

3.6 Detailbeschreibung ausgewählter Funktionen BTFSx, CALL, MOVF, RRF, SUBWF, DECFSZ, XORLW

3.6.1 BTFSS

Bei diesem Befehl wird bei einem bestimmten Register im Speicher des PICs exakt ein Bit betrachtet. So muss bei Verwendung dieses Befehl sowohl eine Adresse als auch die Position des Bits mitgegeben werden. Diese Informationen sind im Hex-Code des Befehls in der LST-Datei codiert. Mit Masken kann leicht an diese gelangt werden.
Somit wird hier geprüft, ob das angegebene Bit gesetzt ist beziehungsweise gleich 1 ist. In diesem Fall wird der nächste Befehl übersprungen. (Bit = 0, Befehl wird nicht übersprungen). Wenn es zu einem Überspringen kommt, handelt es sich dabei um eine sogenannte “Two-Cycle-Instruction”. D.h. der PIC braucht einen weiteren Befehlszyklus, um diesen Befehl vollständig abzuarbeiten. Dabei muss erneut der Laufzeitzähler erhöht werden, der Prescaler geupdatet und der Programmzähler inkrementiert werden.




```
private void instr_BTFSS() {
     computeAddress(instruction);
```

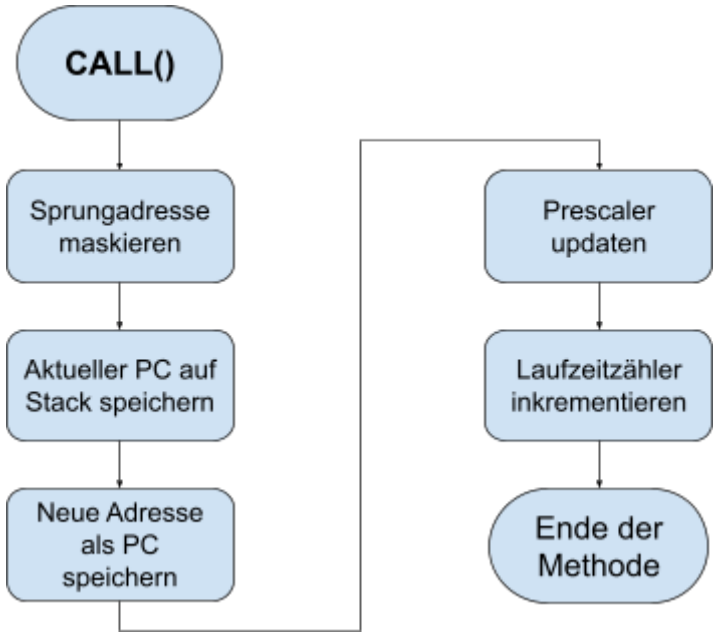
Zunächst wird die Adresse aus dem Befehl heraus maskiert. Außerdem wird ermittelt, ob es sich hierbei um eine indirekte Adressierung handelt. Diese beiden Informationen werden jeweils in einer globalen Variable in der PIC Klasse gespeichert. Danach wird der Wert des Register ausgelesen aus dem Speicher mit der Möglichkeit, dies indirekt zu tun.

```
int value = memory.read_indirect(address, indirect);
```

```
int pos = getPos();
```
Daraufhin wird die Position ermittelt. Dabei wird der Befehl mit maskiert und anschließend um sieben nach rechts geshiftet, da an der Position sieben die Information für die Bitposition steht.
```
private int getPos() {
   return (instruction & Mask_Lib.BIT_POS_MASK) >> 7;
}
```
Mit diesen beiden Informationen kann der Wert des gesuchten Bits ermittelt werden. Hierbei wird eine Hilfsklasse für Bitoperationen genutzt.
```
int bit = BitOperator.getBit(value, pos);
```
Schlussendlich wird der Wert des Bits abgefragt und je nachdem wird der nächste Befehl übersprungen. Dies geschieht durch Inkrementieren des Program-Counters. So wird beim nächsten Fetch-Zyklus ein Zugriff mit einem um eins erhöhten Index gemacht.
```
if (bit == 1) {
   prescaler.update();
   increment_RuntimeCounter();
   memory.increment_PC();
}
```

3.6.2 CALL

Der Call Befehl startet ein kleines Unterprogramm; ein Abschnitt an Befehlen, welche häufiger in genau dieser Reihenfolge gebraucht werden. Dabei springt der PIC in einen anderen Teil der LST-Datei und führt dort Befehle aus. Dabei wird die Adresse gespeichert, wovon der Sprung gestartet ist, um später wieder zurückkehren zu können. Dies kann durch einen der verschiedenen “Return”-Befehlen erfolgen, welche der PIC besitzt. Das Programm geht wieder an den Punkt zurück, der vorher gespeichert wurde.



Als aller Erstes wird aus dem Befehl mit Hilfe einer Maske die 11 bit lange Sprungadresse ermittelt. Diese Maske wird nur bei den beiden Befehlen Call und Goto verwendet. Anschließend wird der aktuelle Wert des Program-Counters auf den Stack gepusht, um diesen später wieder poppen zu können.

```
int k11 = instruction & Mask_Lib.GOTO_CALL_MASK;
stack.push(memory.getPC());
```
Danach wird der 11 bit Wert als neuer Wert für den Program-Counter gesetzt. Hierbei gibt es die Besonderheit, dass die obersten zwei Bit aus dem PCLATH Register genommen werden und als 12. und 13. Bit verwendet werden beziehungsweise Position 10 und 11.
```
memory.pclath_3n4_ontoPC(k11);
```
Ausschnitt aus der Methode “pclath_3n4_ontoPC”:
```
pclath = pclath << 8;
//combine pclath and PC
setPC(pclath | k11_newPC);
```
“pclath” ist dabei der Wert, welcher aus dem PCLATH Register ausgelesen wurde und die Bits an Position 3 und 4 maskiert wurden. Dieser Wert wird dann mit dem neuen Wert für den PC mit Oder verknüpft und anschließend als neuer Program-Counter gespeichert.

3.6.3. MOVF

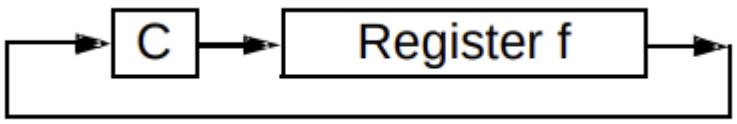
Bei diesem Befehl wird der Inhalt eines Register entweder in das W-Register oder wieder in sich selbst bewegt. Dazu gleich mehr. Das W-Register ist das wichtigste im PIC, da alle Verschiebungen an Werte von Register zu Register über das W-Register erfolgen. Um zu entscheiden, wohin der Inhalt des Registers “f” bewegt wird, muss das Destination-Bit gesetzt werden. Falls dieses Bit gleich 0 ist, so wird der Inhalt in das W-Register geschrieben. Ansonsten wieder zurück in das ursprüngliche Register. Dies hat den Nutzen, dass wenn der Inhalt dieses Registers gleich null ist, so wird in die sogenannte Zero-Flag im STATUS-Register gesetzt. Dazu mehr in Abschnitt 3.7 Flags und deren Implementierung.
```
private void instr_MOVF() {
   computeAddress(instruction);

   int value = memory.read_indirect(address, indirect);

   memory.check_n_manipulate_Z(value);
   writeInMemoryDestinationBit_indirect(address, value, indirect);
}
```
Die Methode “check_n_manipulate_Z” prüft den Wert, ob dieser null gleicht. Falls ja, wird eben die Zero-Flag gesetzt oder eben auf 0 zurückgesetzt. Auch hier ist erneut eine indirekte Adressierung möglich.

3.6.4. RRF

Um diesen Befehl am leichtesten zu verstehen, sollte diese Abbildung betrachtet werden:



(Auszug aus dem Datenblatt des PIC16f84, Seite 67)
Bei dem Kasten links mit dem C handelt es sich um die Carry-Flag. Die Abkürzung RRF steht hierbei für “Rotate Right f through Carry". Der Inhalt des Registers wird um eine Bit-Position nach rechts verschoben. Die freigewordene Stelle im Register wird mit dem Wert

aus dem Carry aufgefüllt. Ebenfalls wird das Bit, das auf der rechten Seite "herausfällt", wird das neue Carry.

```
private void instr_RRF() {
   computeAddress(instruction);
   //get carry flag and value of f
   int value = memory.read_indirect(address, indirect);
   int carry = memory.get_C();
   int bit0 = BitOperator.getBit(value, 0);
```

Als erstes wird die Adresse aus dem Befehlscode maskiert. Anschließend wird der Wert des Registers (value) ausgelesen, die Carry-Flag wird gespeichert und das LSb wird ebenfalls mit der Hilfsklasse “BitOperator” ermittelt. Diese Werte müssen zwischengespeichert werden, da sie sonst bei dem “Shiften” des Werts teilweise verloren gehen würden.
Als erstes wird der eingelesene Wert um eine Stelle nach rechts geshiftet und erneut in dieser Variable gespeichert.

```
//right shift of one bit
value = value >> 1;
```

Daraufhin wird der Wert der Carry-Flag betrachtet. Ist dieser gleich 1 so, muss ebenfalls das Bit an der Position 7 von “value” gesetzt werden:

```
if (carry == 1) {
   value = BitOperator.setBit(value, 7);
}
```

Das gleiche Spiel muss nun erneut für den neuen Wert der Carry-Flag gemacht werden. Hierbei wird nun das LSb von “value” betrachtet. Je nachdem muss das Carry auf 1 oder 0 gesetzt werden. Dies ist leicht mit einer einfachen “if-else”-Abfrage zu implementieren.

```
if (bit0 == 0) {
   memory.unset_C();
} else {
   memory.set_C();
}
```

Am Ende wird nur noch der Wert erneut gespeichert und diese Methode ist beendet.

3.6.5. SUBWF

Bei diesem Befehl wird der Inhalt des Registers “f” minus den Inhalt des W-Registers gerechnet. Das Ergebnis dieser Rechnung kann ebenfalls wieder mit Hilfe des Destination-Bits entweder wieder im W-Register oder im Register “f” gespeichert werden.
Um diese Rechnung durchzuführen, wird intern das Zweierkomplement des Werts des W-Registers gebildet und anschließend wird eine Addition ausgeführt. In der Simulation wird eine einfache Subtraktion vollzogen.

```
int value = memory.read_indirect(address, indirect);
int result = value - W;
```

Nach dem Auslesen des Werts wird die Rechnung durchgeführt und in einer Variablen gespeichert.

```
memory.check_n_manipulate_Z(result);
```

Auch hier wird geprüft, ob das Ergebnis der Rechnung gleich null ist, da in diesem Fall die Zero-Flag gesetzt werden muss. Jedoch wird die Carry-

Flag anders als in anderen Operationen gesetzt, weswegen hier die Prüfung manuell gemacht werden muss und nicht über die Methode, welche auch in Punkt 3.6.4 zu sehen war. Bei einem Ergebnis von größer gleich null wird die Carry-Flag gesetzt; bei einem negativen Ergebnis unsettet. Dies ist leicht mit einer einfachen “if-else”-Abfrage zu bewerkstelligen. Um die Carry-Flag zu setzen, wird eine Methode aus der RAM Klasse verwendet, nämlich “set_C()”. In der PIC Klasse heißt die Instanz der RAM Klasse “memory”.

```
if (result >= 0) { // 0 or positive
   memory.set_C();
} else { // negative
   memory.unset_C();
}
```

Außerdem muss ebenfalls die Digitcarry-Flag besonders gesetzt werden. Beide dieser Fehler resultieren aus einem Hardware Fehler, welcher später jedoch als Feature verkauft wurde.

```
int w_nibble = ~W;
w_nibble = w_nibble & Mask_Lib.NIBBLE_MASK;
int val_nibble = value & Mask_Lib.NIBBLE_MASK;
```

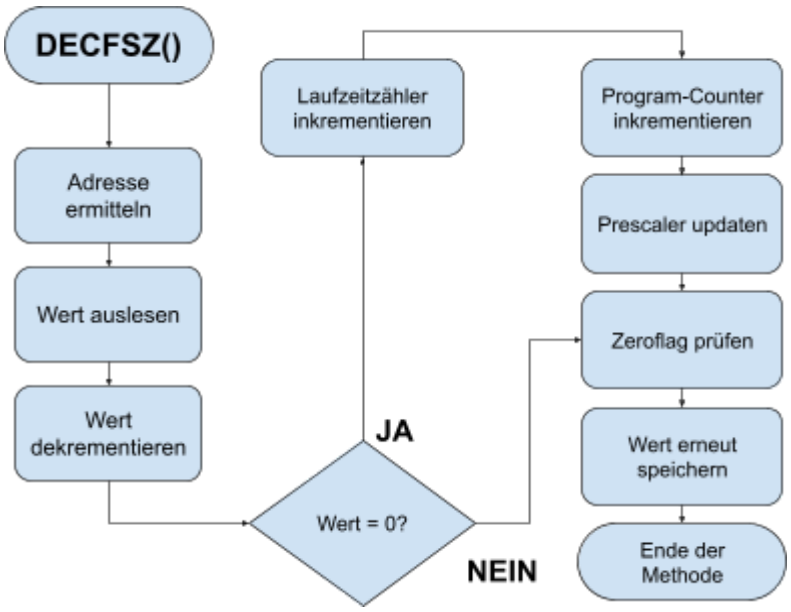
Als erstes müssen die beiden unteren Nibble (entspricht 4 bit) maskiert werden und der Nibble des W-Register zusätzlich invertiert werden mit dem “~”-Operator in Java. Die Reihenfolge dieser Operationen ist in diesem Fall irrelevant.

```
if ((val_nibble + w_nibble) + 1 > Mask_Lib.NIBBLE_MASK) {
   memory.set_DC();
} else {
   memory.unset_DC();
}
```

Daraufhin wird eben die Digitcarry-Flag über eine Methode der RAM Instanz gesetzt beziehungsweise unsettet. Anschließend wird der Wert erneut wieder gespeichert. Je nach Destination-Bit im W-Register oder wieder im Register “f”.

3.6.6. DECFSZ

Der Inhalt des angegebenen Registers wird dekrementiert, d.h. um eins erniedrigt. Falls durch diese Rechnung das Ergebnis gleich null wird, so wird der nächste Befehl übersprungen.




```
private void instr_DECFSZ() {
    computeAddress(instruction);

    int value = memory.read_indirect(address, indirect);
    value--;
    if (value == 0) {
        //2 cycle if skip
        increment_RuntimeCounter();
        memory.increment_PC();
        prescaler.update();
    }
    memory.check_n_manipulate_Z(value);
    writeInMemoryDestinationBit_indirect(address, value, indirect);
}
```

Falls übersprungen wird, handelt es sich erneut um eine “Two-Cycle-Instruction”, d.h. erneut muss zusätzlich der Laufzeitzähler erhöht werden, der Programmzähler inkrementiert werden und zuletzt der Prescaler geupdatet werden. Abschließend wird der Wert wieder entweder in das W-Register oder in das ursprüngliche Register gespeichert. Logischerweise muss hier ebenfalls überprüft werden, ob das Ergebnis dieser Operation gleich null ist, da dann die Zero-Flag gesetzt werden muss.

3.6.7. XORLW

XOR ist eine logische Verknüpfung, die man mit “Entweder - Oder” übersetzen könnte. Dabei ist der Output gleich 1, falls nur eine der beiden Eingangsvariablen wahr und die andere falsch ist. Anbei eine Wahrheitstabelle, um die Verknüpfung zu veranschaulichen:

A	B	Output
0	0	0
0	1	1
1	0	1
1	1	0

In diesem Fall wird der Inhalt des W-Registers mit einem festen im Programm hinterlegten Literal - also einer Konstante - verknüpft und das Ergebnis wird im W-Register ebenfalls gespeichert.

```
private void instr_XORLW() {
    int k = instruction & Mask_Lib.LITERAL_MASK;
    int result = k ^ W;
    memory.check_n_manipulate_Z(result);
    writeInW(result);
}
```

Diese Verknüpfung ist leicht in Java mit dem Operator “^” zu realisieren. Anschließend wird wieder überprüft, ob die Zero-Flag wegen dieser Operation gesetzt werden muss. Dies kann zum Beispiel erreicht werden, wenn ein Register-Wert mit sich selbst verknüpft wird.

3.7 Flags und deren Implementierung

In dem PIC Baustein gibt es viele Flags, mit unterschiedlichsten Funktionen, jedoch gibt es drei Flags, welche am meisten genutzt werden. Diese drei Flags werden auch zusätzlich explizit in der GUI im Bereich “f” angezeigt; nämlich die Carry-Flag, die Zero-Flag und die Digitcarry-Flag. Da diese Flags so oft manipuliert werden, wurden extra Getter-, Setter- und Unsetter-Methoden für diese implementiert, welche die jeweilige Flag auf 1 beziehungsweise auf 0 setzen.

```
public void set_C() {
    setBit(Label_Lib.STATUS, STATUS_lib.carry);
}
public void unset_C() {
    unsetBit(Label_Lib.STATUS, STATUS_lib.carry);
}
```

Zu jeder dieser Flags gibt es eine Methode, welche überprüft, ob diese gesetzt oder nicht gesetzt werden muss. So wird geprüft, ob ein Wert bei einer Rechnung einen Überlauf erzeugt, dementsprechend muss die Carry-Flag gesetzt werden.

```
public void check_n_manipulate_C(int result) {
    if (result > 255) {
        set_C();
    } else {
        unset_C();
    }
}
```

Dies lässt sich leicht überprüfen, indem man testet, ob der errechnete Wert über 255 liegt, da ein Byte, die Größe eines Registers des PICs, maximal den Wert 255 annehmen kann,

also in Binär: 0b1111_1111. Wird dieser überschritten, kommt es eben zu einem Überlauf und das Register hat wieder den Wert 0. Dies kann sehr leicht durch eine Maskierung der unteren acht Bits realisiert werden.

Die Digitcarry-Flag lässt sich dabei nicht nur mit einem Übergabeparameter überprüfen, da hierbei ein Überlauf von dem unteren Nibble in den oberen Nibble getestet wird. Ein Nibble entspricht 4-Bit. Somit besteht eine 8-Bit Zahl aus zwei Nibbles.

```
public void check_n_manipulate_DC(int valA, int valB) {
    // mask both values to only the 4 lowest bits
    int masked_val1 = valA & Mask_Lib.NIBBLE_MASK;
    int masked_val2 = valB & Mask_Lib.NIBBLE_MASK;
    if ((masked_val1 + masked_val2) > Mask_Lib.NIBBLE_MASK) {
        set_DC();
    } else {
        unset_DC();
    }
}
```

Es werden beide Übergabeparameter mit der entsprechenden Maske mit “&” verknüpft und danach die Summe der beiden Werte verglichen. Ist diese größer als 15 - also dem Wert, der binär der Zahl 0b1111 entspricht beziehungsweise der Nibble Maske - dann kommt es zu einem Überlauf von dem einen zu dem anderen Nibble.

3.8 Interrupts

Nach jedem Befehlszyklus (Fetch - Decode - Execute) wird die interne Interrupt Service Routine aufgerufen. Dabei wird jeweils geprüft, ob zum Einen die Flag des Interrupts gesetzt ist, also ob der Interrupt ausgelöst wurde und zum Anderen, ob dieser überhaupt aktiviert wurde über das jeweilige Interrupt Enable Bit, welches im INTCON Register vom Programm selbst gesetzt werden muss. Falls diese beiden Bedingungen erfüllt sind und sich der PIC im Sleep-Modus befindet, wird ein Wake-Up ausgelöst. Andernfalls falls zusätzlich das GIE-Bit (Global Interrupt Enable Bit) gesetzt ist, wird ein Interrupt ausgelöst und der PIC springt an die Adresse 0x0004. Diese ist fest gesetzt und muss bei der Interrupt Service Routine des Assembler-Programms je nachdem beachtet werden. Die alte Adresse wird auf den Stack gepusht und kann mit einem “Return from Interrupt”-Befehl (RETFIE) wieder zurückerlangt werden.

```
private void checkForInterrupts() {
    boolean tmr0_int = check_TMR0_Interrupt();
    boolean rb0_int = check_INT_Interrupt();
    boolean rbChange_int = check_RB_Interrupt();
    boolean eeprom_int = check_EEPROM_Interrupt();

    int GIE = memory.readBit(Label_Lib.INTCON, INTCON_lib.GIE);
    if (tmr0_int || rb0_int || rbChange_int || eeprom_int) {
        if(getSleep()) {
            wakeUp_Interrupt();
        }

        if (GIE == 1) {
```

```
            //interrupt CPU
            memory.unsetBit(Label_Lib.INTCON, INTCON_lib.GIE);
            stack.push(memory.getPC());
            memory.setPC(0x0004);
        }
    }
}
```

Um einen besseren Einblick zu bekommen, wie überprüft wird, ob ein spezieller Interrupt ausgelöst wurde, hier der Code für den TMR0 Interrupt.

```
private boolean check_TMR0_Interrupt() {

   int t0if = memory.readBit(Label_Lib.INTCON,
INTCON_lib.T0IF);

   int t0ie = memory.readBit(Label_Lib.INTCON,
INTCON_lib.T0IE);

   return (t0if == 1) && (t0ie == 1);

}
```

Hierbei werden nur die beiden Bits (TMR0-Interrupt-Flag: T0IF und TMR0-Interrupt-Enable: T0IE) im INTCON Register abgefragt und falls beide gesetzt sind, wird ein “true” zurückgegeben.
Etwas schwieriger sieht es dabei für den Interrupt für den Port Eingang RB0 aus, da hier im Option Register, weitere Einstellungen gemacht werden können, für welche Flanken dieser Interrupt ausgelöst werden kann. Zunächst wird eine Variable benötigt, die den alten Wert im vorherigen Takt speichert, um diesen mit dem neuen Wert zu vergleichen. Dadurch kann eben eine Flanke erkannt werden.

```
if(RB0_old != RB0_new) {
   //rising edge and rising edge selected
   if(RB0_new == 1 &&
       memory.readBit_bank(Label_Lib.OPTION, OPTION_lib.INTEDG, 1) == 1) {
       memory.setBit(Label_Lib.INTCON, INTCON_lib.INTF);
          }  else if(RB0_new == 0 &&
       memory.readBit_bank(Label_Lib.OPTION, OPTION_lib.INTEDG, 1) == 0) {
       //falling edge and falling edge selected
       memory.setBit(Label_Lib.INTCON, INTCON_lib.INTF);
   }
[...]
```

Falls der alte Wert ungleich dem neuen Wert an dem Pin RB0 ist, so liegt eine steigende beziehungsweise fallende Flanke vor. Nun muss abgefragt werden, ob eine fallende oder steigende Flanke den Interrupt auslöst. Dies kann durch das INTEDG (Interrupt Edge Bit) festgelegt werden, welches sich eben im Option Register befindet. Falls es sich um die Richtige handelt, wird die Flag im INTCON Register gesetzt, welche anschließend wie oben bereits beim Timer gesehen überprüft wird. Schlussendlich muss immer der neue Wert, welcher an RB0 anliegt, erneut gespeichert werden.

```
RB0_old = memory.readBit_bank(Label_Lib.PORTB, 0, 0);
```

3.9 TRIS-Register und Latchfunktion

Die TRIS-Register steuern, ob ein Pin eines Ports des PIC Bausteins sich als Eingang oder Ausgang verhält. Hierbei resultiert eine “1” im jeweiligen TRISA beziehungsweise TRISB Register in einem Ausgang im Port Register. TRIS steht hierbei für Tri-State, welcher wie im Namen steht, drei Zustände annehmen kann. High, Low oder Hochohmig. Im hochohmigen Zustand fungiert der Pin als Eingang und muss deswegen einen hohen Widerstand haben, um keine Signalstörung zu erzeugen. Im High- oder Low-Zustand wird das jeweilige Signal nach außen geleitet. Die Besonderheit dabei ist, dass falls ein Pin als Eingang durch das TRIS-Register definiert ist und dennoch ein Wert von dem PIC Baustein hinein geschrieben wird, wird dieser Wert nicht an dem Pin nach außen hin ankommen, aber dennoch geht er nicht verloren. Dieser Wert wird in einem internen D-Latch gespeichert. Sobald dieser Pin wieder durch Manipulation des TRIS-Registers auf Eingang gestellt wird, erscheint dieser Wert aus dem Latch direkt an dem Pin. Dieses besondere Verhalten lässt sich mit Test-File 15 überprüfen.
Im Simulator wurde dieses Verhalten wie folgt realisiert: zunächst ist zu erwähnen, dass es in diesem Simulator zwei Orte für die Speicherung der Werte der Ports gibt, welche unter Umständen voneinander abweichen können. Zum einen gibt es die Pins, welche in der GUI sichtbar und mit der Maus umschaltbar sind und zum anderen gibt es ein Array, welches den Datenspeicher des PICs nachbildet. In diesem gibt es zwei Stellen für die Ports. Kommt es nun zu einem Schreibzugriff auf ein Port Register, so wird dieser Wert an die passende Stelle im Array gespeichert. Nach dem Ausführen eines Befehls werden die Pins in der GUI geupdatet, d.h. jeder Pin, welcher als Ausgang definiert ist, erhält den Wert des Bits im Port Register. Kommt es jedoch zu einem Lesezugriff auf die Ports, so werden die Pins in der GUI ausgewertet und an den PIC weitergegeben. Falls es nun zu der oben beschriebenen Besonderheit kommt, erscheint sofort der Wert am Pin, da sobald wieder auf Ausgang gewechselt wird, der Wert aus dem Speicherplatz im Array auf die GUI übertragen wird. Der interne Wert ist auch nicht sichtbar in der GUI, da der Wert der Pins im Register angezeigt wird, da es sich bei dem Anzeigen der Register ebenfalls um einen Lesevorgang handelt.

3.10 State-Machine (EEPROM)

Das EEPROM war zwar geplant und die Datei Struktur wurde bereits angelegt, jedoch wurde es wieder verworfen, da wir am Ende uns dazu entschieden haben, früher abzugeben und somit diese Funktion nicht mehr implementiert wurde.

4 Organisation

4.1 Beschreibung des zeitlichen Projektverlaufs

Hier lässt sich sagen, dass wir uns hierbei an dem von Herrn Lehmann bereitgestellten Zeitplan orientiert haben und versuchten, die Testprogramme und Visualisierung zum entsprechenden Termin zu haben. So schafften wir es, das Programm zu einem großen Teil schon an dem ersten Abgabetermin zu haben, was uns zusätzliche extra Punkte einbrachte. Dies war für uns ein von Anfang an wichtiges Ziel, ganz unabhängig davon, ob wir die

Punkte am Ende wirklich brauchen würden oder nicht. Durch den straffen Zeitplan konnten wir genau dieses Vorhaben erfüllen.

4.2 Beschreibung der verwendeten Versionsverwaltung

Die Versionsverwaltung lief wie die meisten Projekte weltweit über Git, in unserem Fall konkret GitHub mit einem neuen Repository. Es war die beste Option, um effektiv in einem zweier Team an einem Coding-Projekt wie diesem zu arbeiten. Da die einzelnen Veränderungen und Ergänzungen am Code gut nachvollziehbar waren und es immer die Möglichkeit gab, Änderungen erst einmal zu testen, bevor man es “veröffentlichte”. Zudem brachte es den großen Vorteil, dass man zu jeder Zeit zu einem vorherigen Punkt der Entwicklung springen konnte, falls dies mal benötigt wurde. Auch das gleichzeitige Arbeiten an dem Projekt Code verlief durch die “Merge”-Funktion von Git problemlos. Es ist eine kostenlose Variante, die unterschiedlichen Versionen zu verwalten. Diese hat sich in unserem Projekt definitiv bewährt.

4.3 Aufgabenverwaltung

Für die Aufgabenverwaltung haben wir uns für ein Programm mit dem Namen “Notion” entschieden. Es ist zwar nicht explizit für die Softwareentwicklung bekannt und dennoch machte es das “Baukasten”-System von Notion für uns möglich, es für unser Vorhaben zu verwenden. Durch eine geeignete Datenbank und der richtigen Ansicht konnten wir unsere Aufgaben gut dokumentieren, kategorisieren und die Verantwortlichkeit festlegen. Die richtige Verwendung dieses Tools war aber teilweise ein Problem, wo ich aber nochmal im Fazit drauf eingehen werde.

5 Zusammenfassung und persönliches Fazit

5.1 Zusammenfassung

Im Allgemeinen konnten fast alle Grundfunktionen des PICs nachgebaut werden. So wurde der Standardzyklus von Fetch, Decode, Execute erfolgreich nachgebildet. Dies ist auch schön am Code zu erkennen, da diese Schritte spezifische Methoden erhalten haben. Außerdem wurde ein funktionierender Watchdog sowie ein Timer mit Prescaler implementiert, welcher aber nur einem der beiden zugeordnet sein kann. Das Mapping der Register wurde ebenfalls übernommen, sodass jeder Wert jeweils auf beide Bänke geschrieben wird; ausgenommen von manchen Registern im SFR, darunter: Option Register, Port beziehungsweise. Tris Register usw.

Am Ende liefen alle Testprogramme einwandfrei bis auf Testprogramm 101. In diesem Programm wird die Funktion des Assembler Programmiersprache ausgenutzt, dass man den zugewiesenen Hex-Wert, welcher in jeder Zeile mit einem Befehl steht, der als Wert für den Programmzähler dient, verschiebt. Dabei wird dieser Wert verschoben mit dem Schlüsselwort “org”. So kann bereits in der Zeile in einer früheren Programmzeile einen höheren Wert des Program-Counters erreicht werden. Um dies in dem Simulator nachzubilden, müsste man den File-Parser umschreiben, da dies sich als schwierig erwies,

den File-Parser erneut umzubauen, wurde sich dagegen entschieden, diese Funktion zu implementieren.

5.2 Probleme und deren Lösungen

Wir haben uns für die Nutzung von JavaFX entschieden, was einerseits es uns einerseits möglich gemacht hat, die unterschiedlichen Bereiche entsprechend unserer Erwartung zu erstellen, da es viele verschiedene Komponenten wie den TableView für die LST-Tabelle sowie die Stack-Darstellung benutzt wurde, aber auch den GridPane für die Port-Visualisierung. Andererseits stellte diese Entscheidung auch eine signifikante Herausforderung dar, da wir beide keinerlei Vorerfahrung mit dem Programmieren mit JavaFX mitbrachten.
So hieß es zunächst, das Ganze erstmalig zum Laufen zu bekommen: die richtige SDK herunterladen, die es sowohl einmal für macOS als auch für Windows geben muss und VM -Optionen Betriebssystem spezifisch anpassen. Danach galt es, die grundlegende Struktur einer JavaFX-Anwendung zu verstehen. So mussten wir während der Entwicklung, wie die .fxml-Dateien mit den zugehörigen Java-Controllern zusammenarbeiten, welche Bedeutung die Initialisierung hat und wie komponentenspezifische Methoden wie etwa `refresh()` beim TableView eingesetzt werden mussten.

Diese Komplexität zeigt sich auch im Code. So zeigt zum Beispiel die erste Controller-Klasse TableLSTController deutlich unstrukturierter und redundanter Code als die die fast zuletzt erstellte Controller Klasse StackController. Der Code wurde demnach immer klarer und es wurde für uns deutlicher, auf welche Art und Weise man den Code strukturieren kann, damit er funktionstüchtig bleibt und dennoch schlicht und einfach gehalten bleibt.

Die anfängliche Entscheidung, die Aufgaben innerhalb des Teams klar aufzuteilen - ein Mitglied konzentrierte sich auf die GUI, das andere auf das Backend - erwies sich hierbei als sehr effektiv und wurde deshalb bis zum Ende des Projekts beibehalten.

Das ganze war aber daher sehr geprägt von viel Recherche und dem Verwerfen von Ansätzen. Mehr hierzu aber nochmal im Fazit.

Eine wesentliche Herausforderung bei der Erstellung der grafischen Benutzeroberfläche (GUI) war die notwendige kontinuierliche Aktualisierung der Oberfläche nach jedem einzelnen Schritt. Um den gewünschten Ablauf zu gewährleisten, war es erforderlich, eine öffentlich zugängliche, statische Instanz innerhalb der Klasse zu initialisieren, welche auf sich selbst referenziert (z. B. im `PortController`). Diese Instanz musste einmalig initialisiert werden, um

anschließend im `ButtonsController` die Methode `buildUI()` des `PortControllers` aufzurufen. Ohne diese spezifische Struktur wäre eine fortlaufende Aktualisierung des Views nach jedem Schritt nicht möglich gewesen. Die Identifikation und Implementierung dieser Logik war daher für alle betroffenen Controller zwingend erforderlich, was jedoch ein tiefgehendes Wissen in der Strukturierung und im Zusammenspiel von JavaFX-Komponenten erforderte.

Eine weitere signifikante Herausforderung stellte das korrekte Größe Management der GUI-Elemente dar. Wie im nachfolgenden Kapitel detailliert beschrieben wird, entschieden wir uns bewusst gegen die Verwendung eines UI-Builders und gestalteten das Projekt eigenständig. Dies erforderte eine gut abgestimmte Kombination von Eigenschaften innerhalb der .fxml-Dateien, der verknüpften CSS-Dateien sowie der Controller-Klassen. Dabei war es besonders herausfordernd, die geeigneten Stellen für bestimmte Attribute zu identifizieren. Oftmals funktionierten Attribute wie `vgrow="ALWAYS"` oder `fillHeight="true"` zunächst nicht wie erwartet, da sich später herausstellte, dass sie an insgesamt drei verschiedenen Stellen definiert werden mussten. Die systematische Ermittlung dieser Zusammenhänge erwies sich als zeitaufwendig und erforderte mehrere Stunden intensiver Fehlersuche und Experimente. Zusätzlich bestand die Anforderung von uns, ein responsives Layout umzusetzen, um sicherzustellen, dass die Benutzeroberfläche auf unterschiedlichen Bildschirmauflösungen und variablen Fenstergrößen richtig dargestellt wird. Dies erforderte den gezielten Einsatz von JavaFX-Komponenten wie z. B. GridPane, wobei wir Layout-Attribute wie `GridPane.rowSpan="4"` und `percentWidth="10"` nutzen mussten, um eine flexible und anpassungsfähige Darstellung der GUI zu erreichen.

5.3 Persönliche Erfahrung und Reflexion

Eduard Wayz:
Wie bereits erwähnt, entschieden wir uns zu Beginn des Projekts dafür, die Aufgabenteilung klar zu gestalten. Während sich der Eine hauptsächlich mit der GUI beschäftigte, konzentrierte sich der Andere größtenteils auf das Backend und darauf, dass der PIC intern reibungslos funktioniert und die Testprogramme liefen. Diese Aufteilung war sehr sehr gut, da sich einer von uns dadurch in die Grundlagen von JavaFX einarbeiten konnte und die Zeit hatte, die ganzen neuen Dinge zu verstehen und zu etablieren. Während der Partner die Zeit hatte, viele Dinge zu etablieren, die wir zu einem großen Teil zwar als Konzepte aus den Vorlesungen kannten, aber die Herausforderung mit sich brachten, diese in einem richtigen Projekt zu etablieren.

Diese Vorbereitung durch die Vorlesungen im Thema Programmieren war maßgeblich, da der PIC von sich aus Eigenschaften und Funktionen mit sich brachte, die allein schon von hoher Komplexität zeugten und die dann auch implementiert werden mussten.
Es ist unschrittbar, dass sich diese Aufteilung als äußerst effektiv und erfolgversprechend erwiesen hat, was ich auch im Endprodukt zeigte. Auch wenn wir am Ende aus Zeitgründen keine Möglichkeit mehr hatten, Funktionen wie zum Beispiel das EEPROM zu implementieren, so haben wir das Projekt mit einer Abgabe beendet, die weit über den zu bestehenden Punkten liegt - und das bei einer der frühzeitigeren Abgabeterminen.
Hier möchte ich aber auch etwas ausführlicher auf die in Kapitel 4.3 erwähnte Aufgabenverwaltung eingehen. “Notion” war hierbei wie bereits erwähnt das Tool, welches wir gewählt hatten. Es war eigentlich sehr gut dafür geeignet und dennoch scheiterte es ein wenig an der Umsetzung und an dem “Know-How”. Es waren zu große Aufgaben, die wir zu undetailliert in Notion reingeschrieben hatten und auch das Halten und Abarbeiten der Aufgaben war oft sehr schwammig, da wir uns öfter dann doch mal “schnell” um die Sachen gekümmert haben, die offensichtlich waren oder man gerade auch einfach mehr drauf hatte. Das sind Sachen wie auch das festlegen, wann man welche Aufgabe erledigt oder angeht, die ich das nächste Mal anders machen würde. Mein Fazit hierzu ist ziemlich klar: “Notion” als Aufgabenverwaltungstool, auch im Team, kann ich definitiv weiterempfehlen und doch ist es wichtig, sich an ein paar Grundsätze zu halten und den Umgang damit zu lernen. Ein Guide hierzu hätte uns gut getan. So kann ich das nächste Mal einerseits meine Erfahrung aus diesem Projekt mitnehmen und von Anfang an umsetzen, andererseits kann ich davor auch recherchieren, wie andere Menschen solche Tools nutzen und was sie empfehlen können.
Es war insgesamt definitiv ein Projekt, welches rückblickend große Freude bereitet hat und an welchem ich gerne gearbeitet habe. So schaue ich gerade beim Schreiben auch ein bisschen traurig auf die Sache zurück, da sie schon vorbei ist und doch hat mir das ganze nochmal neu aufgezeigt, dass ich gerne an Softwareprojekten und vor allem im Team arbeite.

Noah Disch:
Zunächst einmal war es ein sehr spannendes Projekt. Es war das erste Mal für mich, an einem so langfristigen Projekt in einem Team mitzuarbeiten. Zuvor hatte ich nur Erfahrungen alleine gemacht. Dadurch hatte man viel mehr Freiheiten in der Gestaltung des Codes. Was zugleich Fluch und Segen war. Zum einen konnte man selbst den Code aufbauen, wie es einem gefiel, aber zum anderen entstand dadurch eine gewisse Unsicherheit, ob diese langfristigen Entscheidungen, die man während der Entwicklung treffen musste, irgendwann im späteren Verlauf des Projekts große Probleme mit sich bringen werden. Ich würde behaupten, im Aufbau des Backends des PICs habe ich keine gravierenden Fehlentscheidungen getroffen.
Anfangs erwies sich die strikte Trennung zwischen Backend und Frontend als sehr effektiv, da dadurch das Testen mit der GUI früh möglich wurde. Jedoch haben wir

unterschätzt, wie lange es dauern würde, bis die GUI vollständig fertig wird, da wir zu Beginn damit gerechnet haben, dass die Person, welche für das Frontend zuständig ist, später zu dem Backend dazustoßen würde. Dies war am Ende nicht der Fall, wodurch die Unterstützung einer Person im Backend fehlte, vor allem für Bugfixing und Testing. Praktischerweise konnte ich mich dann an andere Gruppen wenden, welche mir teilweise helfen konnten und ich ihnen ebenfalls. Es war im Endeffekt ein Geben und Nehmen, welches für alle von Vorteil war, da mehr Augen mehr Fehler sehen.
Ich habe auch viel über Gruppendynamik und Organisierung gelernt. Wie wichtig es ist, viel und regelmäßig zu kommunizieren, sodass man sich gegenseitig nicht im Weg steht. Unsere interne Kommunikation fand auf mehreren Kanälen statt, zum Beispiel auch durch die Commit-Nachrichten konnte man Infos, zu dem was man gemacht hat, an die andere Person weitergeben.
Zudem war es im Verhältnis zu anderen Praktika eine ganz andere Arbeitsweise. Dadurch, dass man völlig frei war in der Zeitgestaltung, außer dass man zu einem bestimmten Zeitpunkt fertig sein musste, war man komplett flexibel. Es war sehr angenehm, die Aufgaben zu erledigen, wenn es gerade am besten gepasst hat. Jedoch hatte man ständig dieses Projekt im Hinterkopf, wodurch man nie richtig deswegen abschalten konnte, bis es endgültig abgegeben wird.
Des Weiteren habe ich viel über meinen Coding-Style gelernt. Man kann gut erkennen, wer welchen Code geschrieben hat, aufgrund des Aufbaus und der Struktur, was ich sehr interessant finde. Außerdem hat mir die Kapselung und Struktur der Klasse unseres Projekts sehr gefallen, da alles sehr gut sortiert ist. Aber auch, habe ich gemerkt, wie enorm wichtig gute und aussagende Kommentare essentiell für solche Projekte sind. Denn wenn man mal ein bis zwei Wochen kaum bis gar nicht an dem Simulator weiterentwickelt, ist es schwer zu verstehen, was man in der Vergangenheit gemacht hat und vor allem auch warum. Hierbei waren nicht nur die Java-Doc Kommentare wichtig, sondern vor allem die eigenen Kommentare zwischen den Zeilen. Ohne diese wäre es unmöglich gewesen, nach meiner Pause wegen eines Kurzzeit Erasmus Aufenthalts so gut wieder einsteigen zu können.
Ich bin auch sehr zufrieden mit der verwendeten Versionsverwaltung, welche wir genutzt haben: GitHub. Es lief fast alles einwandfrei, gemeinsam an diesem Projekt gleichzeitig zu arbeiten. Nur ein einziges Mal wurde eine minimale Änderung nicht übernommen, welche am Ende innerhalb von 5 min wiederhergestellt werden konnte.
Am Anfang des Projekts haben wir uns überlegt, für die ganzen Testprogramme Unit-Tests zu schreiben, was das Testen deutlich einfacher gemacht hätte, vor allem um Programme mehrfach testen zu können. Dieser Ansatz entspricht nicht ganz dem Ansatz von Unit Tests, bei denen man einzelne Abschnitte testet. Diese Tests zu schreiben hat im Nachhinein deutlich länger gedauert als erwartet. Dennoch finde ich, dass ich in zukünftigen Projekten diese erneut verwenden werde, da das Schreiben der ersten paar Tests viel Einblick in den Code und dessen Funktion gegeben hat.

Allgemein war es eine sehr spannende und lehrreiche Erfahrung. Ich habe viel über mich als Programmierer und wie ich im Team arbeite gelernt. Es war sehr interessant, ein völlig eigenes Projekt von Grund auf zu entwickeln und zu schreiben.

6 Anhang

6.1 Weitere Anlagen

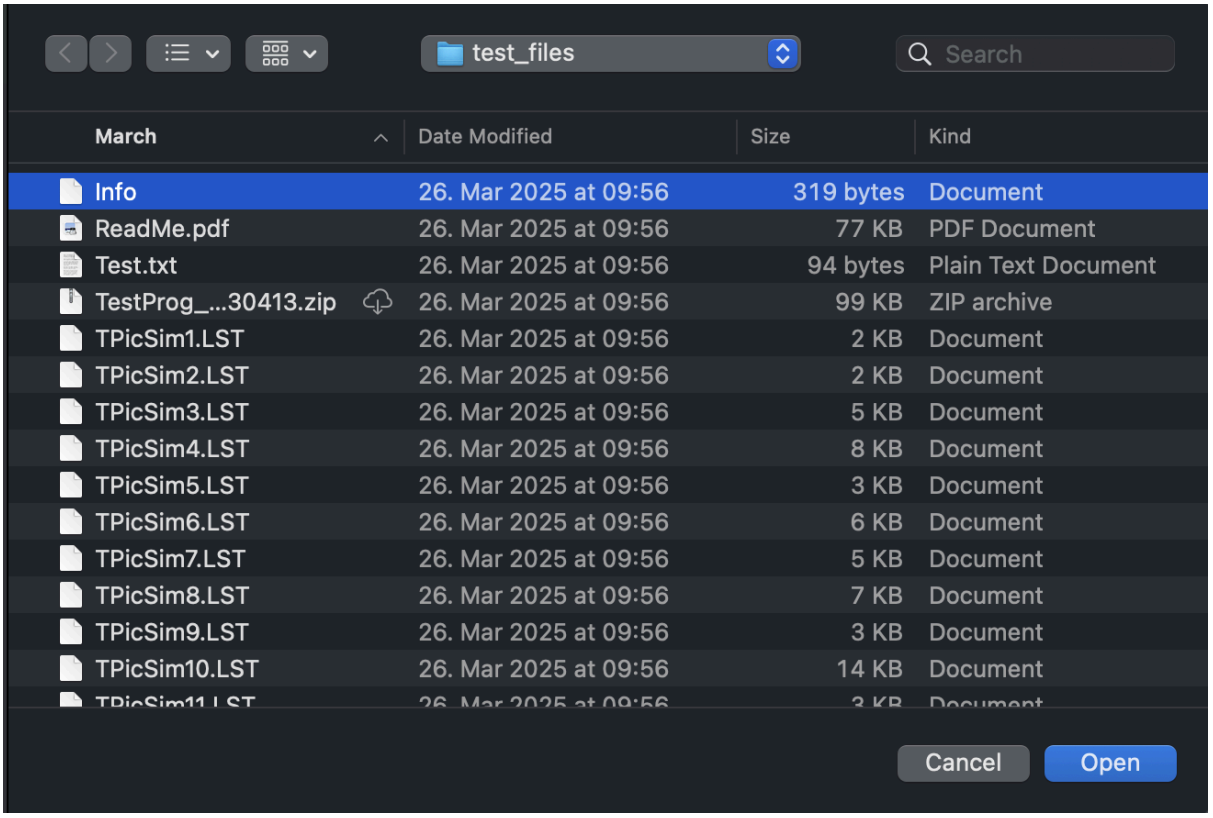


Abbildung 1 Dateiauswahl-Menü

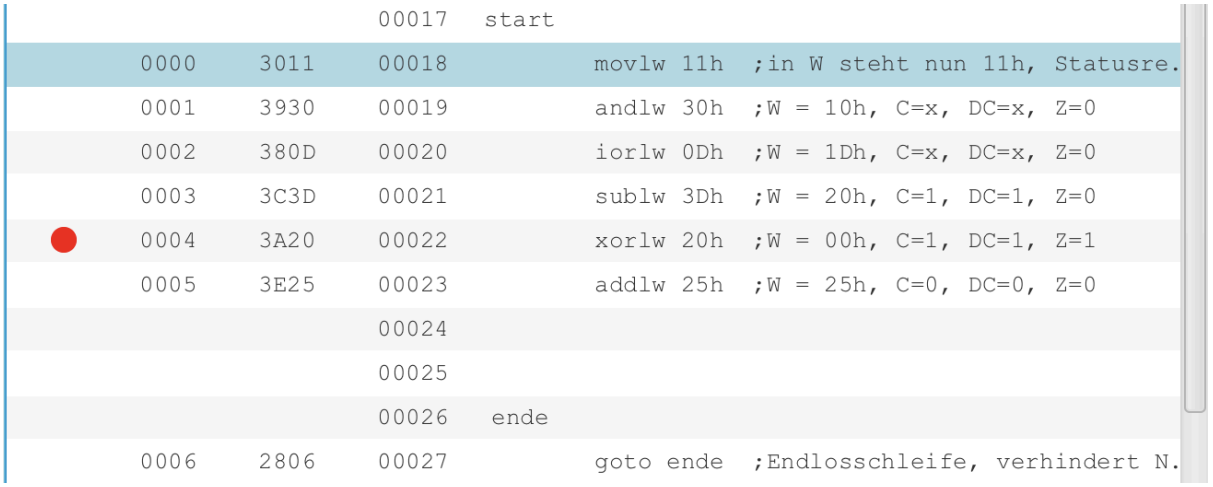


Abbildung 2 Breakpoints

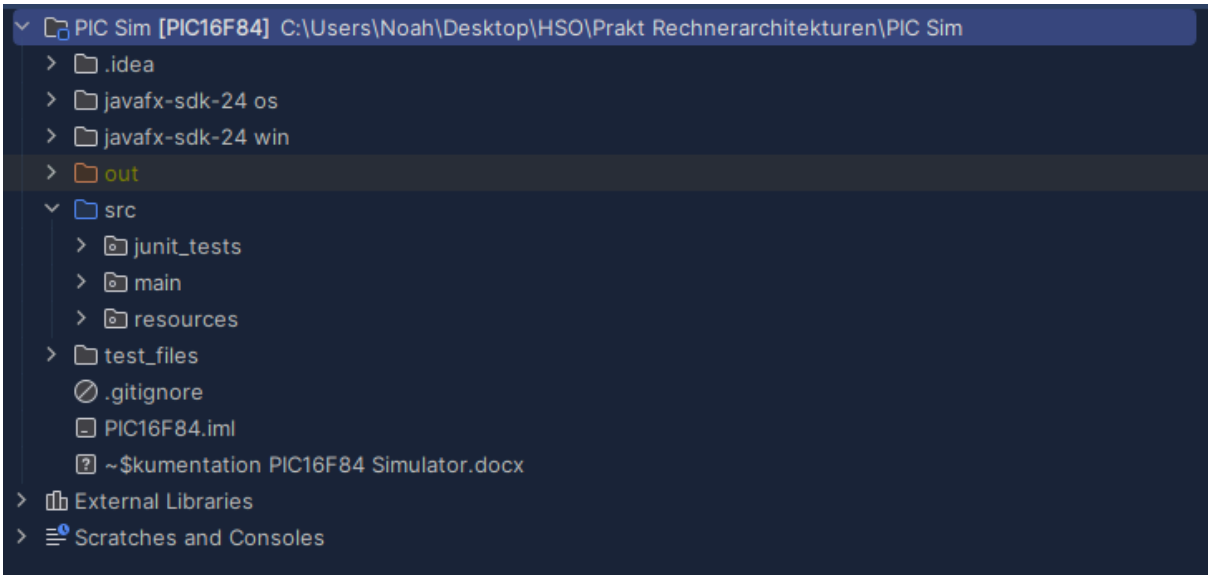


Abbildung 3 Screenshot der Dateistruktur aus der IDE IntelliJ