

Dokumentation
Simulation eines PIC 16F84 Microcontrollers
unter OSX 10.9

Studiengang Informatik – Informationstechnik
an der
Dualen Hochschule Baden-Württemberg Karlsruhe

von
Dennis Stengele und Irtaza Syed

| | |
|--------------|------------------------------|
| Kurs | TINF12B3 |
| Vorlesung | Systemnahe Programmierung II |
| Betreuer | Dipl. Ing Stefan Lehman |
| Abgabetermin | 16.04.2014 |

Inhaltsverzeichnis

| | | |
|-------|--|----|
| 1 | Vorwort..... | 4 |
| 2 | Simulation..... | 5 |
| 3 | Programmiersprache und Benutzeroberfläche..... | 6 |
| 3.1 | Register..... | 7 |
| 3.2 | General Purpose Register | 8 |
| 3.4 | Menüleiste..... | 9 |
| 3.5 | Buttons..... | 9 |
| 3.6 | Code-Fenster mit Breakpoints | 10 |
| 3.7 | PC Call Stack..... | 11 |
| 3.8 | Laufzeit | 11 |
| 3.9 | Clock Speed..... | 12 |
| 4 | Programmstruktur und wichtige Funktionen..... | 13 |
| 4.1 | Klassen | 13 |
| 4.2 | Interrupts..... | 16 |
| 4.3 | TRIS-Register | 16 |
| 4.4 | Befehlsgruppen..... | 16 |
| 4.4.1 | Byte-Orientierte-Befehle | 17 |
| 4.4.2 | Bit-Orientierte-Befehle | 19 |
| 4.4.3 | Literal-Orientierte-Befehle..... | 20 |
| 5 | Fazit..... | 21 |

Abbildungsverzeichnis

| | |
|---|----|
| Abb. 1: PIC Simu GUI | 6 |
| Abb. 2: Bank 1 und Bank 2 | 7 |
| Abb. 3: General Purpose Register | 8 |
| Abb. 4: Menüleiste | 9 |
| Abb. 5: Steuerelemente | 9 |
| Abb. 6: Code-Fenster mit Breakpoints | 10 |
| Abb. 7: PC Call Stack | 11 |
| Abb. 8: Laufzeit | 11 |
| Abb. 9: Clock Speed | 12 |
| Abb. 10: Klasse PSRegister | 13 |
| Abb. 11: Klasse PSRegisters | 14 |
| Abb. 12: Flowchart PIC Simulation | 15 |
| Abb. 13: Opcode mit Instruktion vergleichen | 16 |
| Abb. 14: Funktion des Befehls CLRWDI | 17 |
| Abb. 15: DECFSZ | 18 |
| Abb. 16: BTFSC | 19 |
| Abb. 17: XORLW | 20 |

1 Vorwort

Im Studienfach Systemnahe Programmierung im 4. Semester soll durch die Programmierung eines PIC 16F84¹ Microcontrollers die Funktionsweise und der Aufbau eines Microcontrollers vertieft werden. Das Verhalten eines realen PIC Microcontrollers soll möglichst genau nachgebildet werden. Um dies zu ermöglichen wird das Datenblatt des PIC 16F84 Microcontrollers verwendet.

In dieser Dokumentation wird die Benutzeroberfläche und die Programmstruktur der Simulationssoftware beschrieben.

¹ siehe Datenblatt 30430D

2 Simulation

Eine Simulation ist ein Verfahren zur Nachbildung von realen oder gedachten Systemen. Dafür wird ein Modell/Simulator entwickelt, der die wichtigsten Merkmale und Funktionen des zu simulierenden Systems darstellt. So können bei der Simulation an dem Modell Experimente durchgeführt werden, um Erkenntnisse über das reale System zu gewinnen.

Im Falle der PIC 16F84 Microcontroller Simulation wird eine Software mit einer graphischen Benutzeroberfläche entwickelt. Das Datenblatt des realen PIC 16F84 Microcontrollers dient zur möglichst genauen Implementierung der einzelnen Funktionen des Microcontrollers.

Im Folgenden sind einige Vor- und Nachteile einer Simulation erläutert:

Vorteile:

- Kostengünstig, da keine Hardware bereitgestellt werden muss.
- Das Laden eines neuen Assemblerprogramms gestaltet sich softwareseitig einfach und schnell, während es auf der Hardware einen Mehraufwand bedeutet.
- Das Problem mit beschädigter oder fehlerhafter Hardware kann nicht auftreten, da alles auf Software basiert.
- Die Zwischenergebnisse können auf einer graphischen Benutzeroberfläche übersichtlicher dargestellt werden. Fehler können somit besser erkannt werden.

Nachteile:

- Eine Simulation kann zu einem verfälschten Ergebnis führen wenn die Software nicht richtig implementiert worden ist.
- Das Implementieren einer Simulation wird bei komplexer Hardware Zeit- und Kostenaufwändiger. Dabei steigt auch die Wahrscheinlichkeit des Fehlverhaltens eines Simulators.

3 Programmiersprache und Benutzeroberfläche

Die graphische Benutzeroberfläche wird auf dem Betriebssystem OSX 10.9, mit der Entwicklungsumgebung Xcode 5.1.1 entwickelt.

Als Programmiersprache wird Objective-C verwendet. Objective-C ist eine objektorientierte Sprache und eine Erweiterung der Programmiersprache C.

Des Weiteren ist Objective-C die primäre Sprache von Cocoa (Mac OS X), die für die Erstellung von Anwendungen für Mac OS X und iOS verwendet.

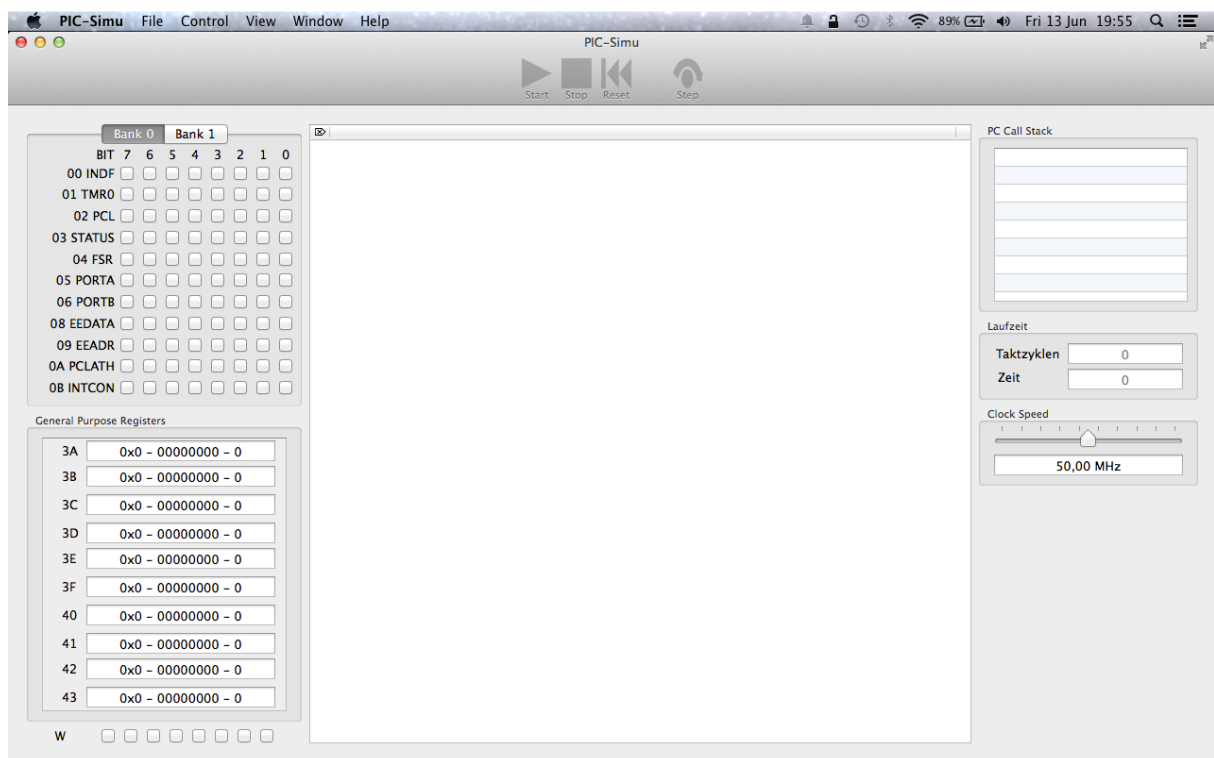
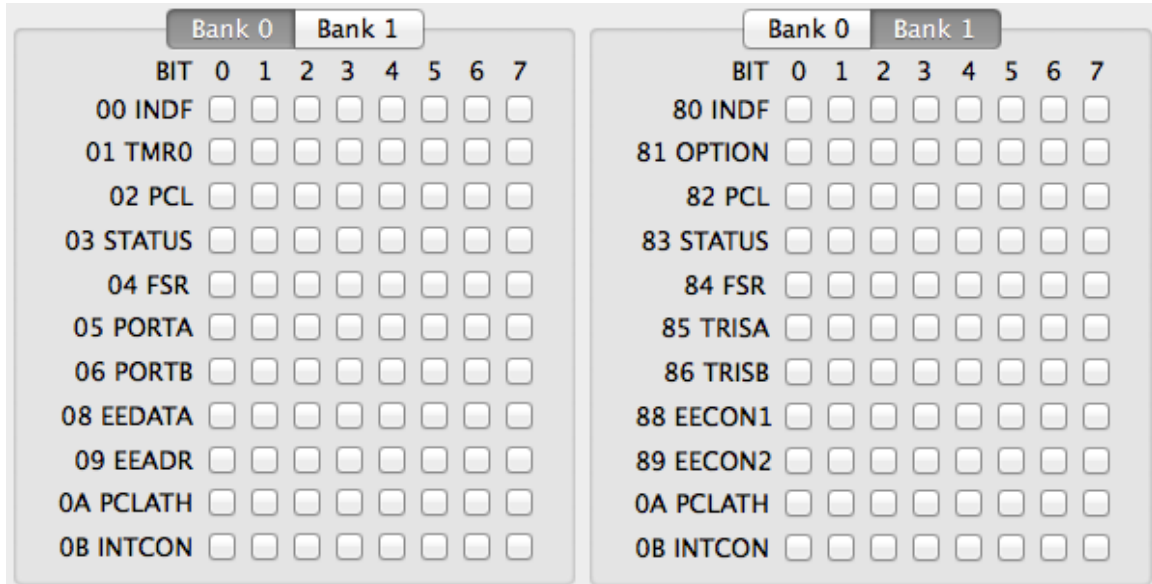


Abb. 1: PIC Simu GUI

Die GUI besteht aus vielen wichtigen Elementen die in Gruppen zusammengefasst eine bestimmte Funktion des Microcontrollers simulieren:

3.1 Register



| Bank | Register | BIT 0 | BIT 1 | BIT 2 | BIT 3 | BIT 4 | BIT 5 | BIT 6 | BIT 7 |
|--------|-----------|--------------------------|--------------------------|--------------------------|--------------------------|--------------------------|--------------------------|--------------------------|--------------------------|
| Bank 0 | 00 INDF | <input type="checkbox"/> | <input type="checkbox"/> | <input type="checkbox"/> | <input type="checkbox"/> | <input type="checkbox"/> | <input type="checkbox"/> | <input type="checkbox"/> | <input type="checkbox"/> |
| Bank 0 | 01 TMRO | <input type="checkbox"/> | <input type="checkbox"/> | <input type="checkbox"/> | <input type="checkbox"/> | <input type="checkbox"/> | <input type="checkbox"/> | <input type="checkbox"/> | <input type="checkbox"/> |
| Bank 0 | 02 PCL | <input type="checkbox"/> | <input type="checkbox"/> | <input type="checkbox"/> | <input type="checkbox"/> | <input type="checkbox"/> | <input type="checkbox"/> | <input type="checkbox"/> | <input type="checkbox"/> |
| Bank 0 | 03 STATUS | <input type="checkbox"/> | <input type="checkbox"/> | <input type="checkbox"/> | <input type="checkbox"/> | <input type="checkbox"/> | <input type="checkbox"/> | <input type="checkbox"/> | <input type="checkbox"/> |
| Bank 0 | 04 FSR | <input type="checkbox"/> | <input type="checkbox"/> | <input type="checkbox"/> | <input type="checkbox"/> | <input type="checkbox"/> | <input type="checkbox"/> | <input type="checkbox"/> | <input type="checkbox"/> |
| Bank 0 | 05 PORTA | <input type="checkbox"/> | <input type="checkbox"/> | <input type="checkbox"/> | <input type="checkbox"/> | <input type="checkbox"/> | <input type="checkbox"/> | <input type="checkbox"/> | <input type="checkbox"/> |
| Bank 0 | 06 PORTB | <input type="checkbox"/> | <input type="checkbox"/> | <input type="checkbox"/> | <input type="checkbox"/> | <input type="checkbox"/> | <input type="checkbox"/> | <input type="checkbox"/> | <input type="checkbox"/> |
| Bank 0 | 08 EEDATA | <input type="checkbox"/> | <input type="checkbox"/> | <input type="checkbox"/> | <input type="checkbox"/> | <input type="checkbox"/> | <input type="checkbox"/> | <input type="checkbox"/> | <input type="checkbox"/> |
| Bank 0 | 09 EEADR | <input type="checkbox"/> | <input type="checkbox"/> | <input type="checkbox"/> | <input type="checkbox"/> | <input type="checkbox"/> | <input type="checkbox"/> | <input type="checkbox"/> | <input type="checkbox"/> |
| Bank 0 | 0A PCLATH | <input type="checkbox"/> | <input type="checkbox"/> | <input type="checkbox"/> | <input type="checkbox"/> | <input type="checkbox"/> | <input type="checkbox"/> | <input type="checkbox"/> | <input type="checkbox"/> |
| Bank 0 | 0B INTCON | <input type="checkbox"/> | <input type="checkbox"/> | <input type="checkbox"/> | <input type="checkbox"/> | <input type="checkbox"/> | <input type="checkbox"/> | <input type="checkbox"/> | <input type="checkbox"/> |
| Bank 1 | 80 INDF | <input type="checkbox"/> | <input type="checkbox"/> | <input type="checkbox"/> | <input type="checkbox"/> | <input type="checkbox"/> | <input type="checkbox"/> | <input type="checkbox"/> | <input type="checkbox"/> |
| Bank 1 | 81 OPTION | <input type="checkbox"/> | <input type="checkbox"/> | <input type="checkbox"/> | <input type="checkbox"/> | <input type="checkbox"/> | <input type="checkbox"/> | <input type="checkbox"/> | <input type="checkbox"/> |
| Bank 1 | 82 PCL | <input type="checkbox"/> | <input type="checkbox"/> | <input type="checkbox"/> | <input type="checkbox"/> | <input type="checkbox"/> | <input type="checkbox"/> | <input type="checkbox"/> | <input type="checkbox"/> |
| Bank 1 | 83 STATUS | <input type="checkbox"/> | <input type="checkbox"/> | <input type="checkbox"/> | <input type="checkbox"/> | <input type="checkbox"/> | <input type="checkbox"/> | <input type="checkbox"/> | <input type="checkbox"/> |
| Bank 1 | 84 FSR | <input type="checkbox"/> | <input type="checkbox"/> | <input type="checkbox"/> | <input type="checkbox"/> | <input type="checkbox"/> | <input type="checkbox"/> | <input type="checkbox"/> | <input type="checkbox"/> |
| Bank 1 | 85 TRISA | <input type="checkbox"/> | <input type="checkbox"/> | <input type="checkbox"/> | <input type="checkbox"/> | <input type="checkbox"/> | <input type="checkbox"/> | <input type="checkbox"/> | <input type="checkbox"/> |
| Bank 1 | 86 TRISB | <input type="checkbox"/> | <input type="checkbox"/> | <input type="checkbox"/> | <input type="checkbox"/> | <input type="checkbox"/> | <input type="checkbox"/> | <input type="checkbox"/> | <input type="checkbox"/> |
| Bank 1 | 88 EECON1 | <input type="checkbox"/> | <input type="checkbox"/> | <input type="checkbox"/> | <input type="checkbox"/> | <input type="checkbox"/> | <input type="checkbox"/> | <input type="checkbox"/> | <input type="checkbox"/> |
| Bank 1 | 89 EECON2 | <input type="checkbox"/> | <input type="checkbox"/> | <input type="checkbox"/> | <input type="checkbox"/> | <input type="checkbox"/> | <input type="checkbox"/> | <input type="checkbox"/> | <input type="checkbox"/> |
| Bank 1 | 0A PCLATH | <input type="checkbox"/> | <input type="checkbox"/> | <input type="checkbox"/> | <input type="checkbox"/> | <input type="checkbox"/> | <input type="checkbox"/> | <input type="checkbox"/> | <input type="checkbox"/> |
| Bank 1 | 0B INTCON | <input type="checkbox"/> | <input type="checkbox"/> | <input type="checkbox"/> | <input type="checkbox"/> | <input type="checkbox"/> | <input type="checkbox"/> | <input type="checkbox"/> | <input type="checkbox"/> |

Abb. 2: Bank 1 und Bank 2

Die Special-Function-Register des Microcontrollers PIC1684 sind in zwei Tabs, Bank 0 und Bank 1 untergebracht. Die einzelnen Bits der Register können durch klicken in den entsprechenden Checkboxes gesetzt oder gelöscht werden.

3.2 General Purpose Register

| General Purpose Registers | |
|---------------------------|--------------------|
| 0C | 0x0 - 00000000 - 0 |
| 0D | 0x0 - 00000000 - 0 |
| 0E | 0x0 - 00000000 - 0 |
| 0F | 0x0 - 00000000 - 0 |
| 10 | 0x0 - 00000000 - 0 |
| 11 | 0x0 - 00000000 - 0 |
| 12 | 0x0 - 00000000 - 0 |
| 13 | 0x0 - 00000000 - 0 |
| 14 | 0x0 - 00000000 - 0 |
| 15 | 0x0 - 00000000 - 0 |

Abb. 3: General Purpose Register

Die General Purpose Register werden in einer Tabelle dargestellt, deren Werte in Hexadezimal -, Binär- und Dezimaldarstellung angezeigt werden.

3.4 Menüleiste

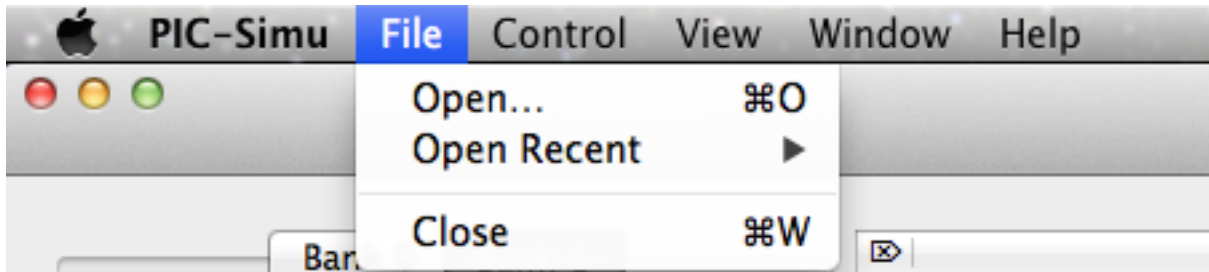


Abb. 4: Menüleiste

Eine Assembler Datei kann aus der Menüleiste mit einem Klick auf „Open“ oder mit dem Tastaturkürzel „Cmd-O“ geöffnet werden. Des Weiteren befinden sich in der Menüleiste Funktionen zur Anpassung von Programmfenster sowie die Steuerungsbuttons, die im Folgenden beschrieben werden.

3.5 Buttons

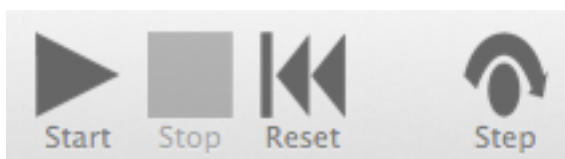


Abb. 5: Steuerelemente

Die Buttons werden für die Steuerung der Simulation benötigt. Die Funktionalitäten beinhalten starten, stoppen, resettet einer Simulation. Mit dem Button „Step“ kann die geöffnete Assembler Datei manuell, Schritt für Schritt abgearbeitet werden.

3.6 Code-Fenster mit Breakpoints

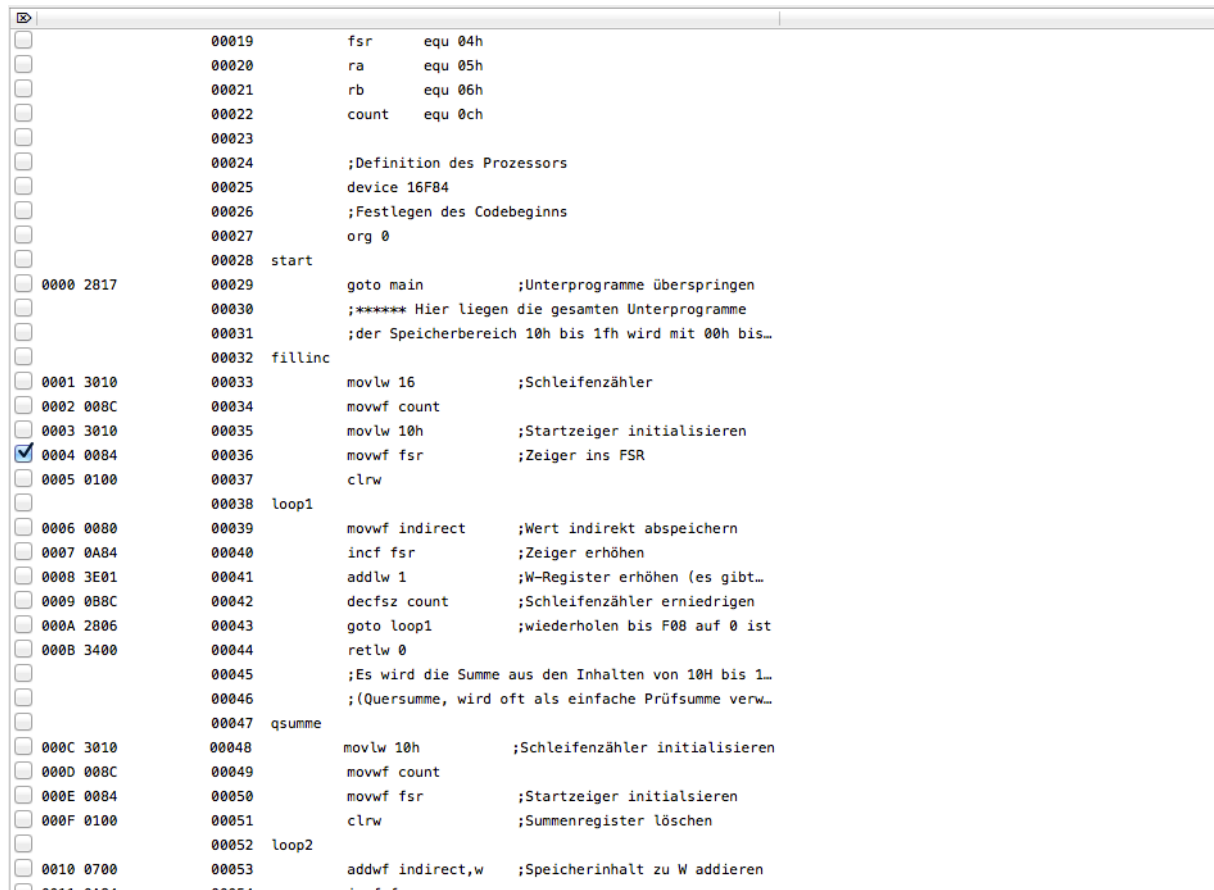


Abb. 6: Code-Fenster mit Breakpoints

Das Code-Fenster ist beim Start des Programms leer. Über der Menü-Leiste können die LST-Files eingelesen werden. Die Breakpoints können durch Klicken auf die Checkboxes erstellt werden.

3.7 PC Call Stack

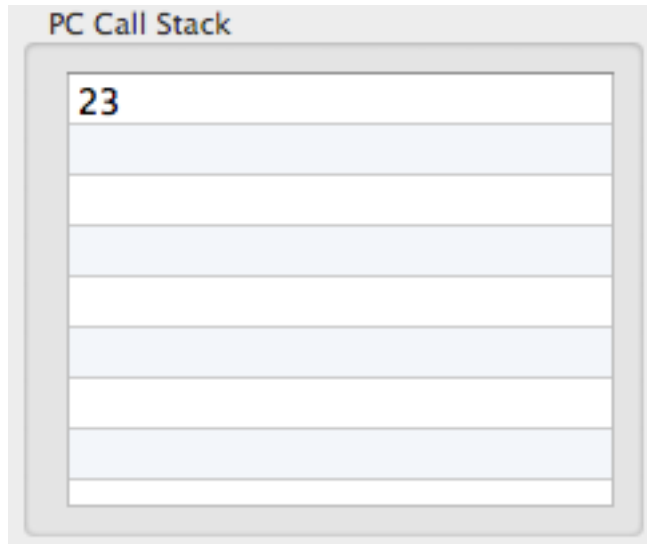


Abb. 7: PC Call Stack

3.8 Laufzeit

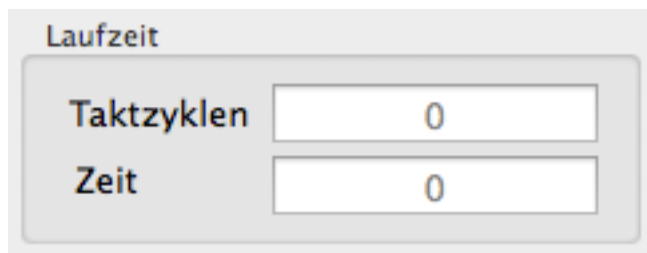


Abb. 8: Laufzeit

In den Feldern Taktzyklen und Zeit wird angezeigt, wie viele Taktzyklen bzw. wie viel Zeit eine Simulation braucht. Beides wird während der Simulation aufgezählt und in den Feldern angezeigt.

3.9 Clock Speed

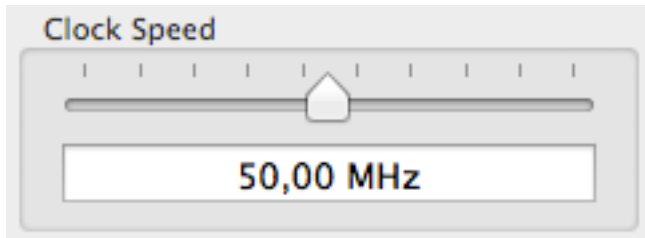


Abb. 9: Clock Speed

Über einen Schieberegler kann man die Quarzfrequenz einstellen. Die kleinste einstellbare Frequenz ist 1 MHz und die größte 100 MHz.

4 Programmstruktur und wichtige Funktionen

4.1 Klassen

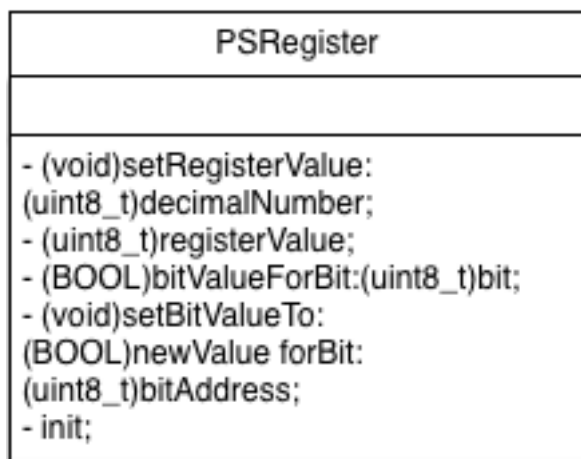


Abb. 10: Klasse PSRegister

In der PSRegister-Klasse wird der Aufbau eines einzelnen Registers beschrieben. Der Wert des Registers wird in 8 BOOL-Werten gespeichert, die die einzelnen Bits repräsentieren. Zum einfachen Zugriff auf die Werte gibt es Methoden, die es erlauben, einzelne Bits zu lesen und zu schreiben (bitValueForBit, setBitValueTo). Mit weiteren Methoden ist es möglich, den Wert des Registers auf einen bestimmten Wert zu setzen (setRegisterValue, registerValue). Die dafür benötigte Belegung der Bits wird in der Methode berechnet.

| PSRegisters |
|---|
| <ul style="list-style-type: none"> - (PSRegister *)registerforAddress: (uint8_t)registerAddress; - (uint16_t)pc; - (void)setPc:(uint16_t)newPc; - (void)incrementPc; - (void)incrementTmr; - (BOOL)checkTmrInt; - (BOOL)checkrb0Int; - (BOOL)checkportbInt; - (void)resetOldRb0; - (void)resetTmrCounter; |

Abb. 11: Klasse PSRegisters

Die PSRegisters-Klasse enthält die gesammelten Register, die im Simulator vorhanden sind.

Dazu sind Methoden vorhanden um auf ein PSRegister-Objekt mit einer bestimmten Adresse zuzugreifen und verschiedene Methoden, die für mehrere Funktionen genutzt werden, die im Folgenden beschrieben werden.

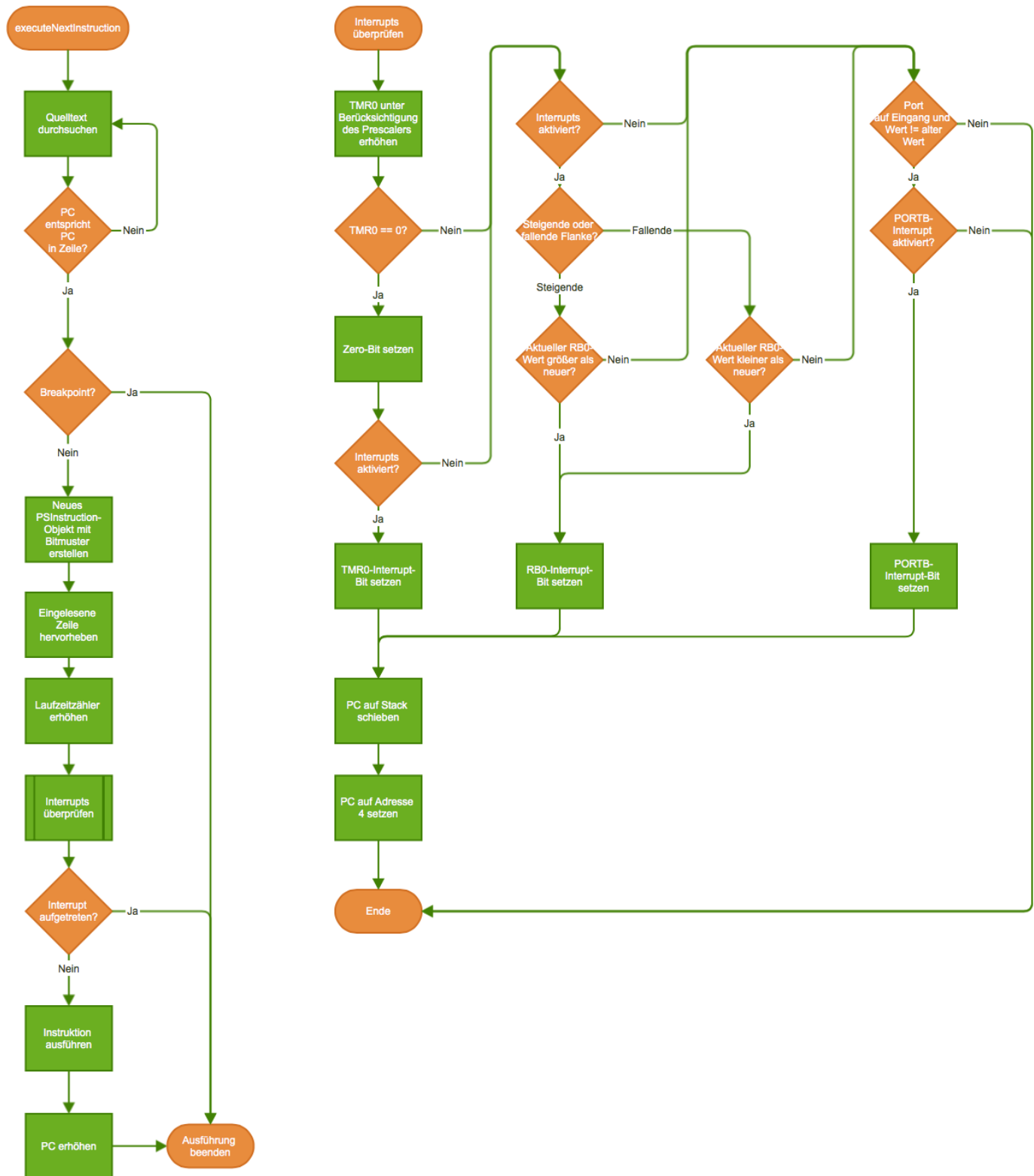


Abb. 12: Flowchart PIC Simulation

4.2 Interrupts

Die einzelnen Interrupts werden hintereinander überprüft. Bei Vorhandensein wird das entsprechende Bit gesetzt und der Programmzähler auf den Wert 4 gesetzt. Der genaue Ablauf lässt sich dem Flowchart (siehe Abb. 12) entnehmen.

4.3 TRIS-Register

Die TRIS-Register werden unter anderem bei der Interruptüberprüfung genutzt, da der PORTB-Interrupt nur auf einem Port auftreten kann, der auch auf Ausgang geschaltet ist.

4.4 Befehlsgruppen

Für den Microcontroller PIC16F84 gibt es drei Arten von Befehlen die sich im 14-Bit langen Opcode voneinander unterscheiden. Alle Befehlsfunktionen werden in der Klasse PSInstruction implementiert. In der Klasse werden zuerst die einzelnen Bitfelder der Instruktion gelesen. Danach wird die gelesene Instruktion mit dem Opcode des jeweiligen Befehls verglichen und bei Übereinstimmung den dazugehörigen Namen des Befehls übergeben:

```
if (instructionBinary == 0b0000000001100100) { //CLRWDT
    self.instruction = @"CLRWDT";
    return self;
}
```

Abb. 13: Opcode mit Instruktion vergleichen

Die zugewiesenen Namen werden dann benutzt, um die eigentlichen Funktionen zu implementieren die das Verhalten der einzelnen Befehle simulieren:


```
if ([self.instruction isEqualToString:@"CLRWDT"]) {  
    pic.wdt = 0b00000000;  
    return;  
}
```

Abb. 14: Funktion des Befehls CLRWDT

In Abb. 14 wird als Beispiel der Wert des Watchdog Timers auf 0 gesetzt wenn der String „CLRWDT“ entspricht.

Im Folgenden werden die einzelnen Befehlsgruppen mithilfe jeweils einer Beispielfunktion beschrieben:

4.4.1 Byte-Orientierte-Befehle

DECFSZ: Decrement f, Skip if 0.

DECFSZ verringert den Wert aus der Speicherzelle f um 1. Falls das Ergebnis 0 ergibt, dann wird der nachfolgende Befehl ignoriert.

Genauso wird auch die Funktion im Code implementiert. Der Wert aus der Speicherzelle f wird mit dem Dekrement Operator um 1 verringert. Danach wird in einer if-Anweisung geprüft, ob das Ergebnis 0 oder ungleich 0 ist. Bei 0 wird der Programmzähler um 1 erhöht und somit die nächste Zeile im Assembler-Code übersprungen (=> NOP) ansonsten soll die nächste Zeile ausgeführt werden.

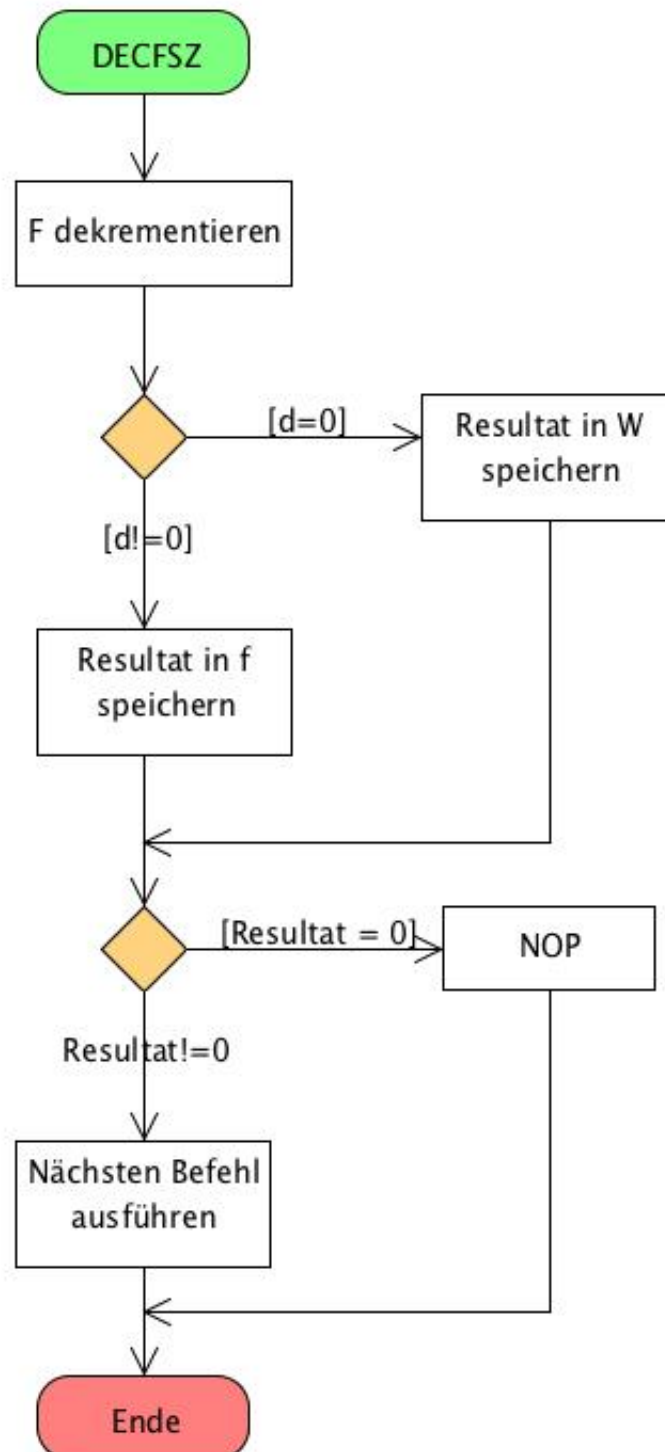


Abb. 15: DECFSZ

4.4.2 Bit-Orientierte-Befehle

BTFSC: Bit Test File, Skip if Clear

Der Befehl BTFSC prüft, ob ein Bit *b*, im File-Register *f* gesetzt ist. Wenn ja wird der direkt folgende Befehl ausgeführt. Ist abgefragtes Bit im File-Register jedoch nicht gesetzt, wird er übersprungen.

Im Source-Code wird der Programmzähler erhöht, wenn das Bit nicht gesetzt ist. Dadurch wird die nächste Zeile im Assembler-Code übersprungen.

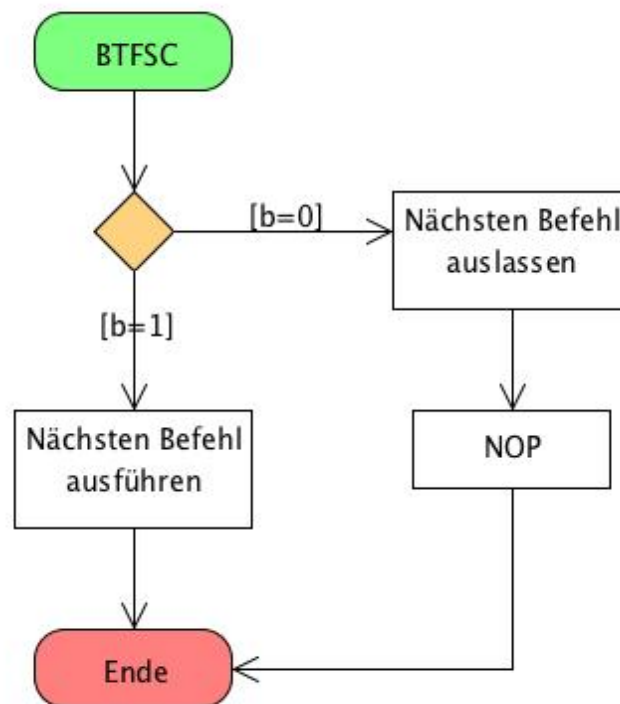


Abb. 16: BTFSC

4.4.3 Literal-Orientierte-Befehle

XORLW: Exclusive OR literal with W

XORLW verknüpft W und eine Zahl mit der Exklusiv-ODER-Funktion. Im Code wird dies mit dem Caret-Zeichen \wedge ermöglicht.

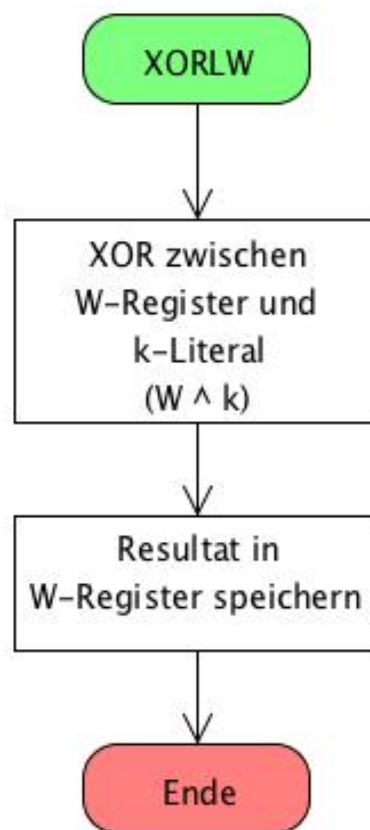


Abb. 17: XORLW

5 Fazit

Die in diesem Projekt zur Umsetzung benötigten Kenntnisse erstreckten sich über ein weitläufiges Feld an Informationen, die wir bereits in früheren Vorlesungen erlernt hatten. Vor allem Vorkenntnisse aus den Fächern Digitaltechnik, Rechnertechnik I und Software-Engineering I und II sind mit in dieses Projekt eingeflossen. Dies bot die Möglichkeit Erlerntes anzuwenden und umzusetzen, um zugleich Wissen zu vertiefen, und die Erfahrung zu machen, fächerübergreifend zu arbeiten.

Der Simulator ermöglicht das Testen eines Assemblerprogramms. Dabei können die Inhalte verschiedener Register angezeigt werden. Des Weiteren wird der Inhalt des Arbeitsregisters und des Stacks sowie der Laufzeitzähler visualisiert. Mit Hilfe von Breakpoints lässt sich der Programmablauf an einer beliebigen Stelle stoppen und somit können die Registerwerte in Ruhe ausgelesen werden.

Es bleiben jedoch noch Verbesserungsmöglichkeiten offen. So kann zum Beispiel noch das EEPROM oder die Hardwareansteuerung implementiert werden.

Zusammenfassend ist zu sagen, dass die Programmierung des PIC Simulators uns die Hardware und die Funktionen des Controllers näher gebracht hat. Die gewählte Programmiersprache Objective-C und die Entwicklungsumgebung Xcode haben wir dadurch wesentlich besser kennengelernt was bei zukünftigen Projekten und Studienarbeiten die Einarbeitungszeit drastisch reduziert.