

Simulator für den Mikrocontroller PIC16F84A



von

Michael Stahlberger und Julian Kühn

Studiengang Informationstechnik
an der Dualen Hochschule Karlsruhe

Inhaltsverzeichnis

1	Einleitung	4
1.1	Was ist ein Simulator?	4
1.2	Vor- und Nachteile eines Simulators	4
1.3	Mikrocontroller PIC16F84A	5
2	Tools	7
2.1	Entwicklungsumgebung Eclipse	7
2.2	Versionsverwaltung Git	8
3	Programmstruktur und Aufbau	9
3.1	Benutzeroberfläche	9
3.2	Programmablauf	12
3.3	Beschreibung einiger Befehle	15
3.3.1	DECFSZ	15
3.3.2	MOVF	16
3.3.3	BTFSS	17
3.3.4	SUBLW	18
3.3.5	CALL	19
3.3.6	RRF	20
3.3.7	XORLW	21
3.4	Interrupt	22
3.5	TRIS-Register	24
3.6	Klassendiagramm	25
4	Zusammenfassung	26
4.1	Umsetzung	26
4.2	Fazit	27

Abbildungsverzeichnis

1.1	Harvard-Architektur	6
2.1	Eclipse	7
2.2	Github	8
3.1	GUI	9
3.2	Datei öffnen	10
3.3	Kontroll-Buttons	10
3.4	Register und Spezialfunktionsregister	11
3.5	Textfeld für den Quelltext	11
3.6	Sequenzdiagramm für das Öffnen und Parsen einer Datei . . .	12
3.7	Sequenzdiagramm für das Erkennen eines Befehls	13
3.8	Klassendiagramm	25

Listings

Listings/exA.java	13
Listings/DECFSZ.java	15
Listings/MOVF.java	16
Listings/BTFSS.java	17
Listings/SUBLW.java	18
Listings/CALL.java	19
Listings/RRF.java	20
Listings/XORLW.java	21
Listings/Interrupt.java	22
Listings/InterruptRB0.java	23
Listings/InterruptOcc.java	23
Listings/InterruptTime.java	24
Listings/TRIS.java	24

Kapitel 1

Einleitung

Die Zielsetzung des Projektes ist es einen Simulator für einen PIC16F84A Mikrocontroller zu schreiben. Dabei sollen verschieden Debug-Funktionen ebenfalls implementiert werden. Es soll Möglich sein ein korrekt geschriebenes Programm zu testen und zu debuggen.

1.1 Was ist ein Simulator?

Eine Simulation ist ein möglichst realitätsnahes Nachbilden von Geschehen der Wirklichkeit. Aus Sicherheits- und Kostengründen ist es für fast alle Anwendungsgebieten notwendig. Die gewonnenen Erkenntnisse können nach einer Simulation auf die Realität Übertragen werden. Eine Simulation findet meistens nicht in Echtzeit statt (wie z.B. bei einer Emulation) sondern wird zu analytischen Zwecken langsamer als in der Realität nachgebildet.

1.2 Vor- und Nachteile eines Simulators

Vorteile: Durch eine Simulation können Versuche die unter gefährlichen Umständen stattfinden müssen sicher nachgestellt werden (z.B. Crash-Simulationen mit Autos und Crash-Test-Dummys). Aber auch Versuche die aus Kostengründen in der Realität oftmals schwierig nachzustellen sind können durch Simulationen begrenzt ersetzt werden. Durch den verlangsamten Ablauf einer Simulation sind außerdem Fehler oder Ergebnisse leichter nachzuvollziehen als in der Wirklichkeit. Im Falle des Mikrocontrollers können Programme

vor ihrem praktischen Einsatz getestet und debuggt werden um so mögliche Fehler im Praxiseinsatz frühzeitig zu erkennen und auszubessern.

Nachteile: Eine Simulation ist meist durch begrenzte Ressourcen eingeschränkt. Sei es die Rechenleistung einer Computersimulation oder Geld und Zeit die für eine Simulation eingesetzt werden müssen. Oftmals wird deswegen nur ein vereinfachtes Modell der Wirklichkeit eingesetzt. Durch diese Vereinfachung kann es zu ungenauen Messergebnissen oder Situationen kommen die in der Realität vielleicht gar nicht vorkommen. Für den PIC16-Simulator ist es wichtig möglichst fehlerfrei und genau zu arbeiten da Fehler innerhalb der Simulation auf falsche Rückschlüsse auf das für den Mikrocontroller entwickelte Programm führen könnte. Auch zu bedenken ist es das die Laufzeit in der Simulation nicht der Realzeit entspricht und somit das Programm in der Realität schneller sein würde.

1.3 Mikrocontroller PIC16F84A

Ein Mikrocontroller ist eine Art Mikrorechnersystem, bei welchem neben ROM und RAM auch Peripherieeinheiten wie Schnittstellen, Timer und Bussysteme auf einem einzigen Chip integriert sind. Die Hauptanwendungsgebiete sind die Steuerungs-, Mess- und Regelungstechnik, sowie die Kommunikationstechnik und die Bildverarbeitung. Mikrocontroller sind in der Regel in Embedded Systems, in die Anwendung eingebettete Systeme, und somit in der Regel von außen nicht sichtbar. Ebenso verfügen sie, im Gegensatz zum PC, nicht über eine direkte Bedien- und Programmierschnittstelle zum Benutzer. Sie werden in der Regel einmal programmiert und installiert.

Der PIC16F84 Mikrocontroller ist ein 8 Bit Mikrocontroller mit RISC-Architektur (Reduced-Instruction-Set-Computing). Es wird also auf komplexe Befehle verzichtet und mit jedem Befehl kann auf jedes Register zugegriffen werden. Der Mikrocontroller besitzt durch die eingesetzte Harvard-Architektur bis zu 14 Bit große Befehle während die Größe des separaten Datenbusses nur 8 Bit beträgt.

1.3. MIKROCONTROLLER PIC16F84A

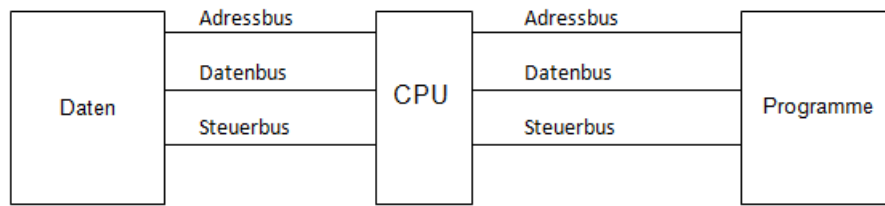


Abbildung 1.1: Harvard-Architektur

Durch die Architektur benötigen fast alle Anweisungen nur einen Instruction Cycle (Abarbeitung eines Maschinenbefehls). Der PIC16 besitzt einen Stack mit Speicherplatz für 8 Adressen sowie 2 externe und 2 interne Interrupt Quellen. Darüber hinaus besitzt der Pic16F ein großes Register, welches in zwei Bänke unterteilt ist. Das Umschalten der Bänke erfolgt im Programmcode. Die Speicherbereiche können auch direkt über ihre Registeradresse angesprochen werden.

Kapitel 2

Tools

2.1 Entwicklungsumgebung Eclipse

Eclipse ist eine freie Entwicklungsumgebung welche ursprünglich für die Sprache Java entwickelt wurde. Mittlerweile gibt es Eclipse Plug-Ins für weitere Programmiersprachen wie C, C++ oder Pearl. Wie die meisten Entwicklungsumgebungen bietet Eclipse eine Vielzahl, dem Entwickler nützlicher, Funktionen. Dazu gehören das Debuggen des Programmcodes, automatische Erstellen von Get- und Set- Methoden sowie automatische Codevervollständigung. Über automatische Kontexthilfe liefert Eclipse Vorschläge um Fehler zu beheben. Eclipse kann leicht durch den Eclipse Marketplace mit verschiedenen Plug-Ins erweitert werden. Für die Verbindung von Eclipse mit dem eingesetzten Versionsverwaltungs System Git wurde das Plug-In EGit verwendet.

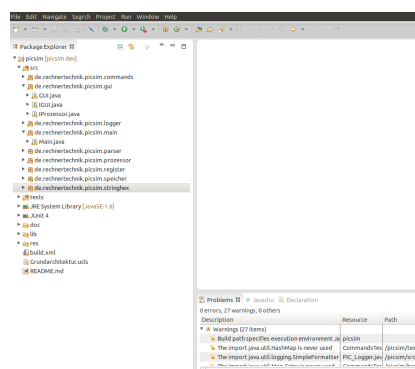


Abbildung 2.1: Eclipse

2.2 Versionsverwaltung Git

Git ist eine freie Software zur verteilten Versionsverwaltung von Dateien, die ursprünglich für die Quellcode-Verwaltung des Linux-Kernels entwickelt wurde. Git speichert die Daten nicht auf einen zentralen Server sondern bei jedem User zunächst lokal in einem s.g. Repository. So besitzt jeder User den gesamten Code so wie die Versionsgeschichte zunächst auf seinem eigenen PC. Ein Remote Repository ist ein Repository das nicht lokal auf dem eigenen Rechner verfügbar ist sondern zentral auf einem Server ausgelagert wird. über einen Push Befehl kann das Remote Repository mit dem lokalem Repository überschrieben werden. Wird ein Fetch Befehl ausgeführt wird dass Remote Repository mit dem lokalem Repository verglichen und zusammengeführt (merge Befehl) werden. Im Projekt wurde GitHub, ein webbasierter Hosting-Dienst für Software-Entwicklungsprojekte verwendet.

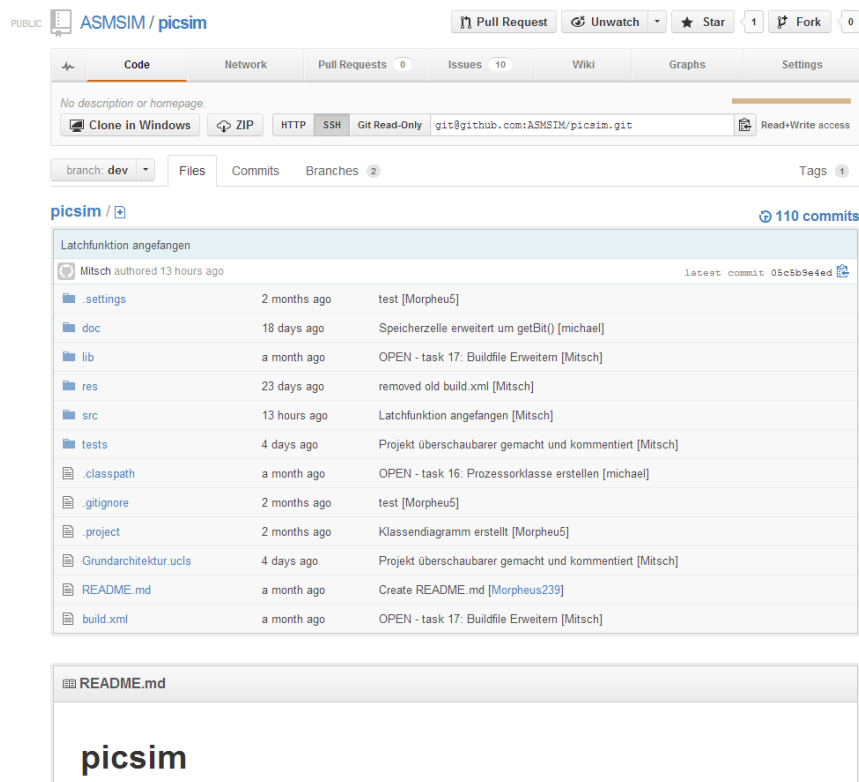


Abbildung 2.2: Github

Kapitel 3

Programmstruktur und Aufbau

3.1 Benutzeroberfläche

Beim Starten der Applikation öffnet sich das User Interface des Simulators auf dem die Funktionen des PICs nachvollzogen werden können.

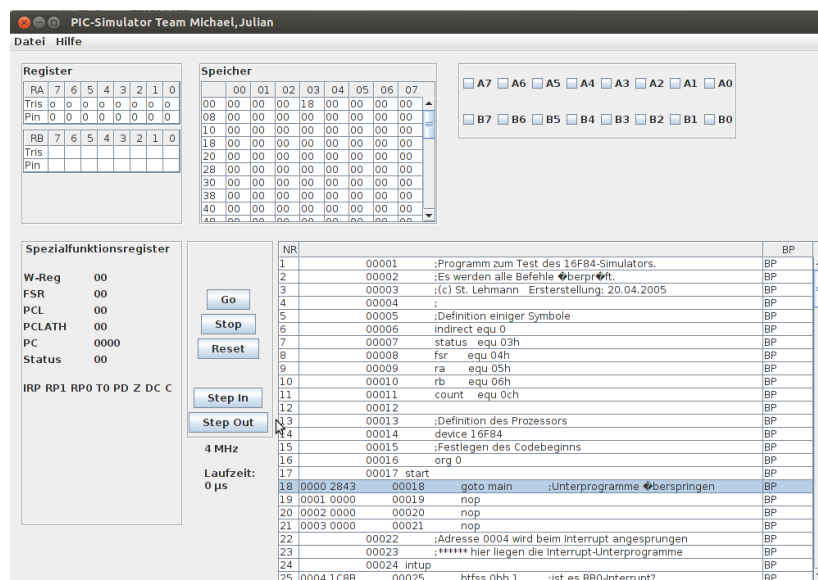


Abbildung 3.1: GUI

3.1. BENUTZEROBERFLÄCHE

Über Datei und Datei öffnen lassen sich Quellcode-Dateien öffnen. In die Dateiauswahl ist ein Dateifilter integriert der ausschließlich .LST-Dateien anzeigt.

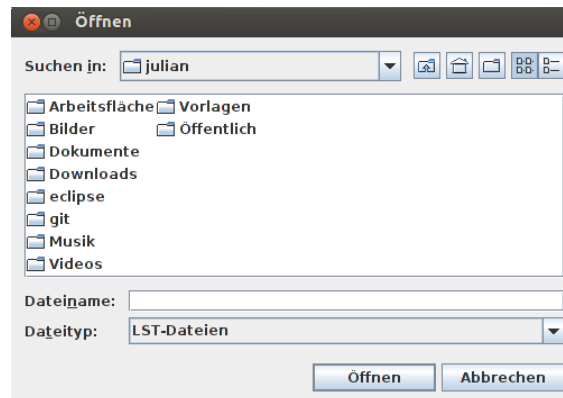


Abbildung 3.2: Datei öffnen

Mit den Buttons „Step-In“, „Step-Out“, „Go“, „Stop“ und „Reset“ lässt sich der Simulationsablauf steuern. Mit Step-In und Step-Out lässt sich jeweils nur ein Programmschritt ausführen, mit Go wird das Programm komplett abgearbeitet. Mit Stop stoppt der Simulationsvorgang und mit Reset wird er komplett zurückgesetzt.

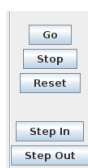


Abbildung 3.3: Kontroll-Buttons

Die Register werden in der linken oberen Ecke des User Interfaces angezeigt. Die Werte des Spezialfunktionsregister wird darunter dargestellt. Die Darstellung der Werte erfolgt im Hexadezimalsystem.

Die Speicherinhalte werden in Form einer Tabelle am oberen Rand des User Interfaces dargestellt. Die Werte werden im Hexadezimalsystem wiedergegeben.

3.1. BENUTZEROBERFLÄCHE

Register								
RA	7	6	5	4	3	2	1	0
Tris	0	0	0	0	0	0	0	0
Pin	0	0	0	0	0	0	0	0
RB	7	6	5	4	3	2	1	0
Tris								
Pin								

Spezialfunktionsregister								
W-Reg	00							
FSR	00							
PCL	00							
PCLATH	00							
PC	0000							
Status	00							
IRP RP1 RP0 T0 PD Z DC C								

Abbildung 3.4: Register und Spezialfunktionsregister

In der rechten unteren Ecke wird der Quelltext des zu simulierenden Programmes angezeigt. Dieser wird wie beschrieben über „Datei öffnen“ in die Tabelle geschrieben.

NR				BP
1	00001	;	Programm zum Test des 16F84-Simulators.	BP
2	00002	;	Es werden alle Befehle überprüft.	BP
3	00003	;	(c) St. Lehmann Erstellung: 20.04.2005	BP
4	00004	;		BP
5	00005	;	Definition einiger Symbole	BP
6	00006	indirect	equ 0	BP
7	00007	status	equ 03h	BP
8	00008	fsr	equ 04h	BP
9	00009	ra	equ 05h	BP
10	00010	rb	equ 06h	BP
11	00011	count	equ 0ch	BP
12	00012			BP
13	00013	;	Definition des Prozessors	BP
14	00014	device	16F84	BP
15	00015	;	Festlegen des Codebeginns	BP
16	00016	org	0	BP
17	00017	start		BP
18	0000 2843	00018	goto main ;	BP
19	0001 0000	00019	nop	BP
20	0002 0000	00020	nop	BP
21	0003 0000	00021	nop	BP
22		00022	;	BP
23		00023	;	BP
24		00024	intup	BP
25	0004 1C8B	00025	btfss 0bh,1 ;	BP

Abbildung 3.5: Textfeld für den Quelltext

3.2 Programmablauf

Im nachfolgendem Sequenzdiagramm wird dargestellt welche Methoden in welcher Klasse beim öffnen einer Datei über den Button in der Menüleiste ausgeführt werden:

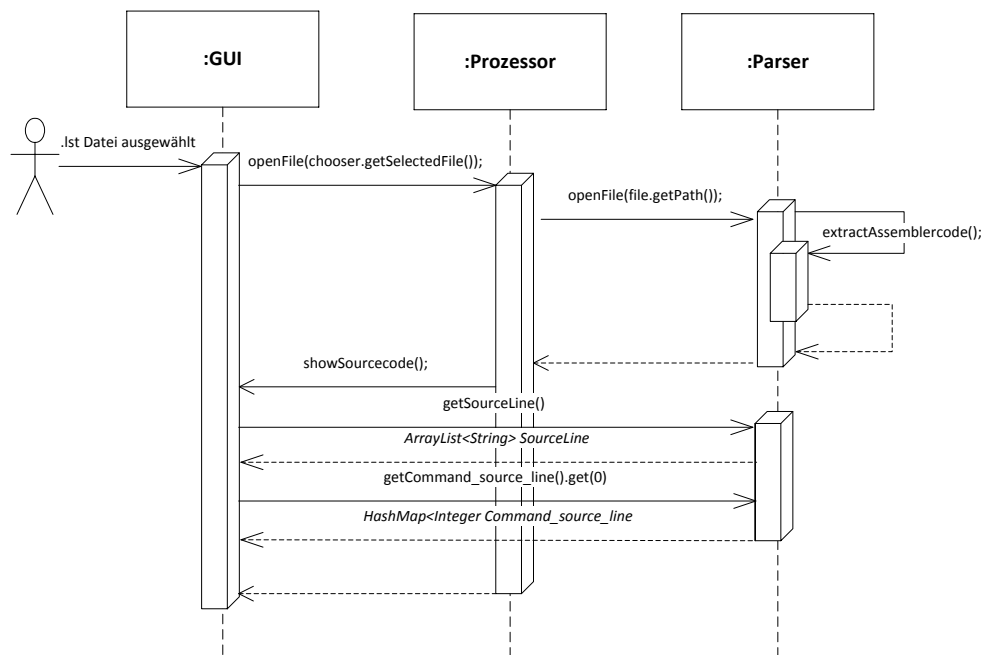


Abbildung 3.6: Sequenzdiagramm für das Öffnen und Parsen einer Datei

Die Methode `showSourcecode(parser.getSourceLine(), parser.getCommand_source_line().get(0))` ruft als Übergabeparameter die Methoden `getSourceLine()` und `getCommand_source_line().get(0)` des Parsers auf. Diese Übergabeparameter sind aus Platzgründen nicht im Diagramm enthalten, werden aber trotzdem aufgerufen.

Die Methode `extractAssemblerCode()` extrahiert aus der `.lst` Datei jeweils die Zeilen als String (`SourceLine()`) und den Befehlscode der ersten Zeichen (`Command_source_line()`). In einer Schleife werden die String-Zeilen im User Interface ausgegeben.

3.2. PROGRAMMABLAUF

```

1  private void extractAssemblercode() {

3      for(int i = 0; i < sourceLine.size(); i++) {

5          if(!(sourceLine.get(i).charAt(0) == ' ')) {

7              String dst = sourceLine.get(i).substring(5, 9);
8              // System.out.println(dst);
9              Integer befehl = (int) Integer.parseInt(dst, 16);
10             // System.out.println(befehl);
11             asmProg.add(befehl);

13             command_source_line.put(asmProg.size() - 1, i);
14         }

16     }
17 }

```

Durch das klicken des Go-Buttons beginnt die Simulation. Wie einzelne Befehle erkannt werden wird im folgendem Sequenzdiagramm dargestellt.

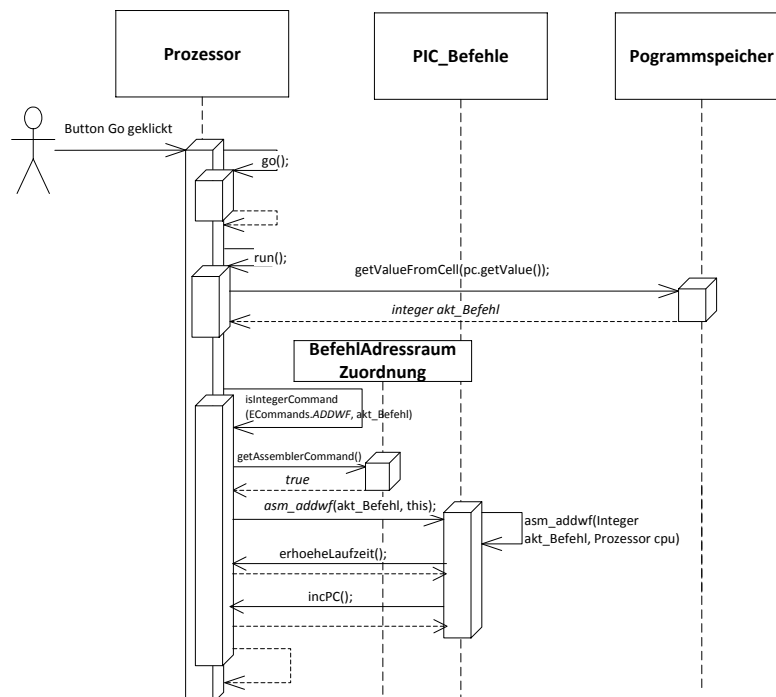


Abbildung 3.7: Sequenzdiagramm für das Erkennen eines Befehls

3.2. PROGRAMMABLAUF

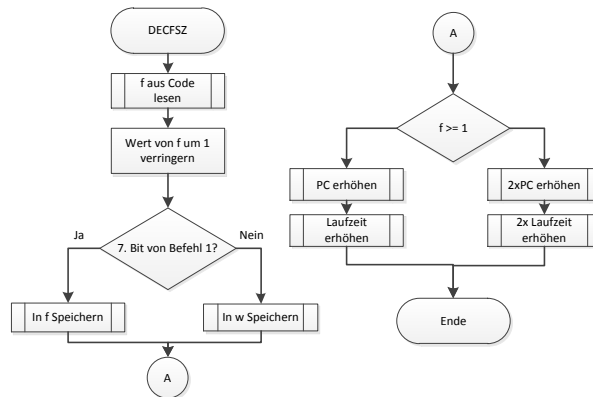
Durch den Go Befehl wird die Schrittweite des run Vorgangs verändert und fängt nun an jeden Befehl zu erkennen und abzuarbeiten. Zunächst wird der aktuelle Befehl aus dem Programmspeicher geladen. Danach wird überprüft ob sich der aktuelle Befehl im Adressbereichsraum des Befehls ADDWF befindet. In diesem Beispiel liefert die Methode true zurück und die eigentliche Ausführungsroutine in der Klasse PIC_Befehle wird aufgerufen. Nach dem Ausführen der Methode wird noch die Laufzeit und der Programm Counter erhöht.

Würde der aktuelle Befehl nicht im Adressraum von ADDWF liegen wäre der Rückgabewert false und die Methode würde den textuell nächsten Adressraum eines Befehls überprüfen. Bei der Untersuchung jedes Befehls wird zusätzlich noch über die Methode checkInterrupt(this) überprüft ob ein Interrupt stattgefunden hat. Wie die Abarbeitung der Befehle im genauen stattfindet wird im nächsten Kapitel näher erläutert.

3.3 Beschreibung einiger Befehle

3.3.1 DECFSZ

In diese Beispiel wird das Ergebnis entweder in f oder in w gespeichert.



```

1  public static void asm_decfsz(Integer akt_Befehl, Prozessor cpu) {

3      Integer f = getOpcodeFromToBit(akt_Befehl, 0, 6); // zum speichern
4      Integer result = cpu.getSpeicherzellenWert(f) - 1;
5      // Speicherort abfragen
6      if(getOpcodeFromToBit(akt_Befehl, 7, 7) == 1) {

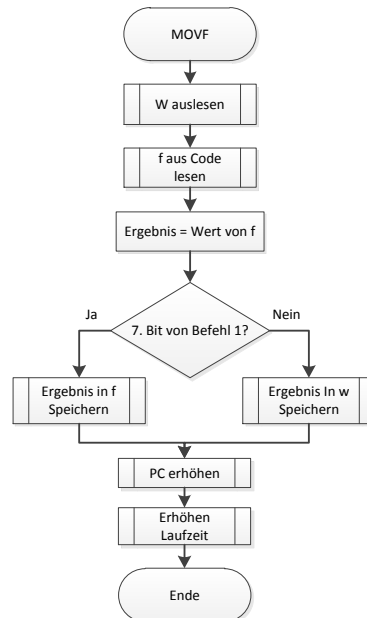
8          // in f Register speichern
9          cpu.setSpeicherzellenWert(f, result, false);
10     }
11     else {
12         // in w Register speichern
13         cpu.setW(result, false);
14     }

16     // Result neu einlesen (evtl overflow)
17     result = cpu.getSpeicherzellenWert(f);

19     if(result >= 1) {
20         cpu.incPC();
21         erhoeheLaufzeit(cpu,1);
22     }
23     else if(result == 0) {
24         cpu.incPC();
25         cpu.incPC();
26         erhoeheLaufzeit(cpu,2);
27     }
28 }
  
```


3.3. BESCHREIBUNG EINIGER BEFEHLE

3.3.2 MOVF

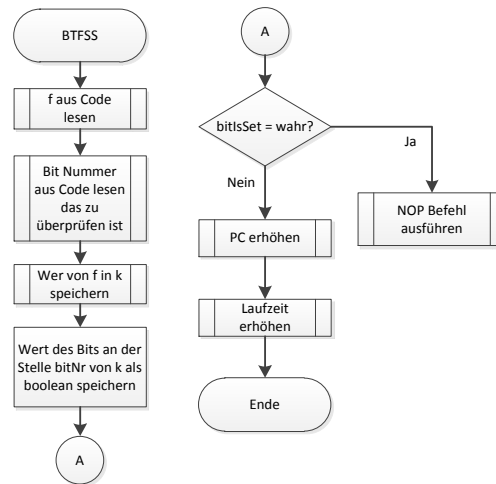


```
1 public static void asm_movf(Integer befehl, Prozessor cpu) {
2     Integer w = cpu.getW();
3     Integer f = getOpcodeFromToBit(befehl, 0, 6);
4     Integer result = cpu.getSpeicherzellenWert(f);

6     if(getOpcodeFromToBit(befehl, 7, 7) == 1) {
7         cpu.setSpeicherzellenWert(f, result, true);
8     }
9     else {
10        cpu.setW(result, true);
11    }
12    cpu.incPC();
13    erhoeheLaufzeit(cpu,1);
14 }
```

3.3. BESCHREIBUNG EINIGER BEFEHLE

3.3.3 BTFSS



```
1  public static void asm_btfsf(Integer akt_Befehl, Prozessor cpu) {

2

3      Integer f = getOpcodeFromToBit(akt_Befehl, 0, 6);
4      Integer bitNr = getOpcodeFromToBit(akt_Befehl, 7, 9);

5

6      Integer k = cpu.getSpeicherzellenWert(f);
7      boolean bitIsSet = getBit(k, bitNr);

8

9      // Bit is set
10     if(bitIsSet) {
11         asm_nop(akt_Befehl, cpu);
12     }
13     // Bit not set
14     else {
15         // Nothing
16     }

17

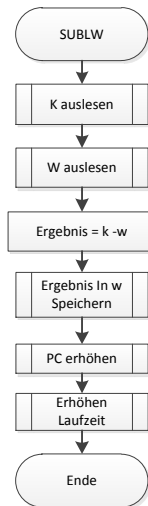
18     // PC++
19     cpu.incPC();
20     erhoeheLaufzeit(cpu,1);

21

22 }
```

3.3. BESCHREIBUNG EINIGER BEFEHLE

3.3.4 SUBLW



```
1  public static void asm_sublw(Integer akt_Befehl, Prozessor cpu) {
2      // Extrahiere K
3      Integer k = getOpcodeFromToBit(akt_Befehl, 0, 7);

5      // Get W
6      Integer w = cpu.getW();

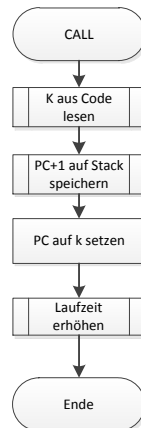
8      // K - W
9      Integer result = k - w;

11     // Ergebnis in W
12     cpu.setW(result, true);

14     // PC++
15     cpu.incPC();
16     erhoeheLaufzeit(cpu, 1);
17 }
```

3.3. BESCHREIBUNG EINIGER BEFEHLE

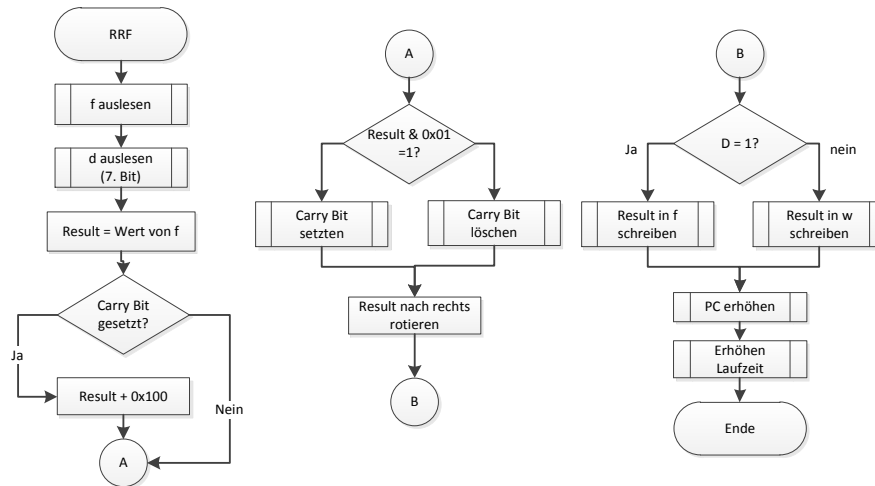
3.3.5 CALL



```
1  public static void asm_call(Integer akt_Befehl, Prozessor cpu) {  
  
3      // Extrahiere K  
4      Integer k = getOpcodeFromToBit(akt_Befehl, 0, 10);  
  
6      // Push PC + 1 auf Stack  
7      Integer pc_inc = cpu.getPCValue() + 1;  
8      cpu.getStack().push(pc_inc);  
  
10     // PC auf K Wert setzen  
11     cpu.setPCL(k);  
12     erhoeheLaufzeit(cpu, 2);  
  
14 }
```

3.3. BESCHREIBUNG EINIGER BEFEHLE

3.3.6 RRF



```
1 public static void asm_rrf(Integer befehl, Prozessor cpu) {
2     Integer f = getOpcodeFromToBit(befehl, 0, 6);
3     Integer d = getOpcodeFromToBit(befehl, 7, 7);
4     Integer result = cpu.getSpeicherzellenWert(f);

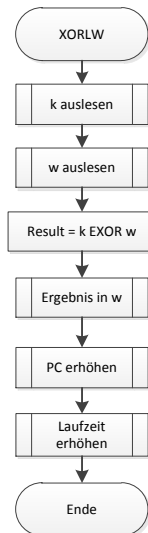
6     if(cpu.getStatus(bits.C)){
7         result += 0x100;
8     }

10    if((result & 0x01) == 1){
11        cpu.setStatus(bits.C);
12    }
13    else{
14        cpu.clearStatus(bits.C);
15    }

17    result = result >> 1;

19    if(d == 1) {
20        cpu.setSpeicherzellenWert(f, result, true);
21    }
22    else {
23        cpu.setW(result, true);
24    }
25    cpu.incPC();
26    erhoeheLaufzeit(cpu,1);
27 }
```

3.3.7 XORLW



```
1  public static void asm_xorlw(Integer akt_Befehl, Prozessor cpu) {  
  
3      // Extrahiere K  
4      Integer k = getOpcodeFromToBit(akt_Befehl, 0, 7);  
  
6      // Get W  
7      Integer w = cpu.getW();  
  
9      // XOR K ^ W  
10     Integer result = k ^ w;  
  
12     // Ergebnis in W  
13     cpu.setW(result, true);  
  
15     // PC++  
16     cpu.incPC();  
17     erhoeheLaufzeit(cpu, 1);  
18 }
```

3.4 Interrupt

Bei einem Interrupt verlässt der PIC16F seine normale Routine und springt in eine Interruptroutine, die er abarbeitet um dann wieder an die Stelle des normalen Ablaufs zurückzukehren.

Im Simulator wird zwischen jedem Funktionsaufruf der einen Befehl beschreibt überprüft ob ein Interrupt stattfindet.

```
1 //Check Interrupt in Prozessor Klasse
2 interruptHandler.checkInterrupt(this);

4 //in der Interrupt Klasse
5 public void checkInterrupt(Prozessor cpu) {

7     Integer INTCON = cpu.get_RAM_Value(0x0b);
8     boolean GIE = ((INTCON & 0x80) == 0x80) ? true : false;
9     boolean INTE = ((INTCON & 0x10) == 0x10) ? true : false;
10    boolean TOIE = ((INTCON & 0x20) == 0x20) ? true : false;

12    System.out.println(Integer.toHexString(INTCON));

14    // Global Interrupt enabled
15    if(GIE) {

17        // RBO Interrupt enabled
18        if(INTE) {
19            PIC_Logger.logger.info("RBO Interrupt check...");
20            checkExternalInterrupt(cpu, INTCON);
21        }

23        // Timer0 Interrupt enabled
24        if(TOIE) {
25            PIC_Logger.logger.info("TMR0 Interrupt check...");
26            checkTimer0Interrupt(cpu, INTCON);
27        }

29    }

31 }
```

Bevor der Interrupt ausgeführt wird muss zuerst geprüft werden ob das GIE (Global Interrupt enable), das INTE Bit im INTCON Register oder das T01E Bit für den Timer0 Interrupt gesetzt worden ist.

Ist das INTE Bit gesetzt wird die Methode checkExternalInterrupt ausgeführt. In dieser Methode wird zunächst überprüft ob es sich um eine stei-

3.4. INTERRUPT

gende oder fallende Flanke handelt (oldValue speichert immer den vorherigen Zustand von RB0). Nach der Überprüfung wird die Methode externerInterruptRB0 ausgeführt.

```
1 private void checkExternalInterrupt(Prozessor cpu, Integer INTCON) {  
  
3     Integer PortB = cpu.get_RAM_Value(0x06);  
4     boolean RB0 = ((PortB & 0x01) == 0x01) ? true : false;  
  
6     Integer OPTION_REG = cpu.get_RAM_Value(0x81);  
7     boolean INTEDG = ((OPTION_REG & 0x40) == 0x40) ? true : false; // true =  
8     // Found rising edge  
9     if(oldValue == false && RB0 == true && INTEDG) {  
10        PIC_Logger.logger.info("Interrupt Rising");  
11        externerInterruptRB0(cpu, INTCON);  
12    }  
13    // Found falling edge  
14    else if(oldValue == true && RB0 == false && !INTEDG) {  
15        PIC_Logger.logger.info("Interrupt Falling");  
16        externerInterruptRB0(cpu, INTCON);  
17    }  
18    // No Interrupt  
19    else {  
20    }  
21    // Save old Value  
22    oldValue = RB0;  
23 }
```

In dieser Methode wird das Interrupt Flag gesetzt und die Methode InterruptHasOccured aufgerufen in der der eigentliche Interruptvorgang ausgeführt wird. Dort wird zunächst das GIE Bit auf 0 gesetzt, der Programm Counter auf dem Stack gespeichert und zum eigentlichen Interruptvektor gesprungen.

```
1 private void interruptHasOccured(Prozessor cpu){  
  
3     Integer INTCON = cpu.get_RAM_Value(0x0b);  
  
5     cpu.setSpeicherzellenWert(0x0b, (INTCON & 0x7F), false); // Disable GIE  
  
7     Integer pcl = cpu.getPCValue(); // Programmcounter holen  
8     cpu.getStack().push(pcl); // PCL auf Stack  
  
10    cpu.setPCL(0x04); // Springe zum Interruptvektor  
11 }
```


3.5. TRIS-REGISTER

Für den Timer 0 Interrupt wird in der `checkInterrupt` Methode nach erfolgreicher Überprüfung `checkTimer0Interrupt` aufgerufen in der zunächst überprüft wird ob ein Timer 0 Interrupt stattgefunden hat. Daraufhin wird die Methode `InterruptHasOccured` aufgerufen wird (wie bei RB0).

```
1 private void checkTimer0Interrupt(Prozessor cpu, Integer INTCON) {
2     boolean TOIF = getBitValue(INTCON, 2);
3     //Is Interrupt?
4     if(TOIF){
5         //Timerinterrupt
6         // Interrupt hat stattgefunden
7         PIC_Logger.logger.info("TMR0 Interrupt hat stattgefunden");
8         interruptHasOccured(cpu);
9     }
10 }
```

3.5 TRIS-Register

Die TRIS-Register (TRI-State Enable) ist ein programmierbares 8 Bit Register welches einen Pin als Input oder Output konfiguriert. Jeder Port besitzt ein TRIS Register welches deren Pinzustände beschreibt.

Im Simulator wird bevor ein Port-Bit gesetzt wird zunächst sein TRIS-Status überprüft. Der gewünscht Pin wird nur verändert wenn der richtige Zustand im TRIS Register steht.

```
1 public void setBitPort(Integer bitNr, Integer adresse){
2     Integer tris = get_RAM_Value(adresse+0x80);
3     Integer port = get_RAM_Value(adresse);
4
5     boolean trisBit = PIC_Befehle.getBit(tris, bitNr);
6     Integer setValue = port | (int)Math.pow(2, bitNr);
7
8     //Nur verndern wenn Tris richtig gesetzt ist
9     if(trisBit){
10         setSpeicherzellenWert(adresse, setValue, false);
11     }
12 }
```

3.6 Klassendiagramm

Im Folgendem sind die verschiedenen Klassen des Simulators und deren Zusammenhang in einem Klassendiagramm dargestellt.

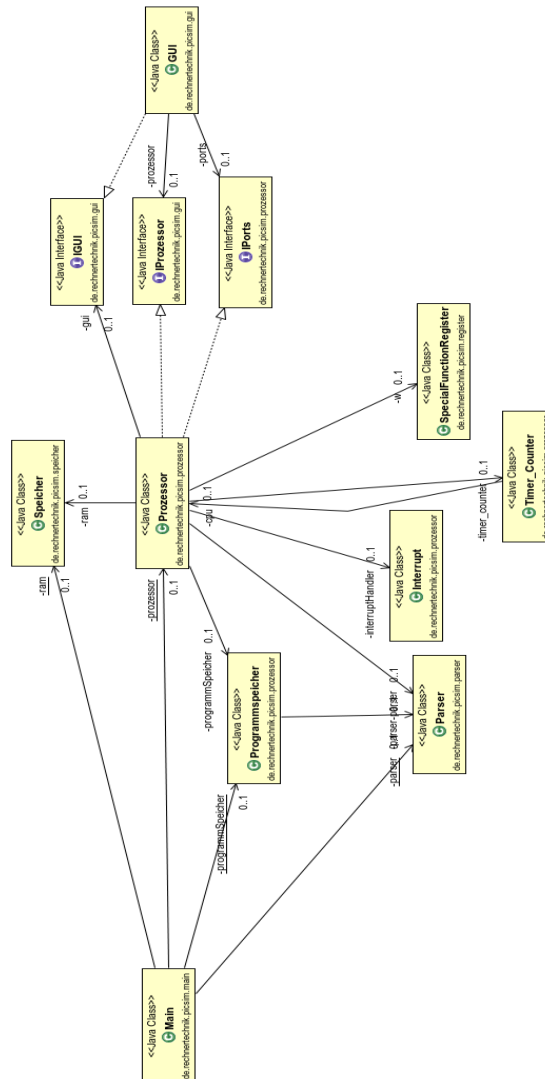


Abbildung 3.8: Klassendiagramm

Kapitel 4

Zusammenfassung

4.1 Umsetzung

Zu Beginn des Projektes wurde ein Pflichtenheft erstellt in dem die Muss, Kann und Abgrenzungskriterien festgelegt wurden. Diese wurden soweit alle eingehalten. Zu den Muss Kriterien gehörten:

- Quellcode sichtbar anzeigen, einlesen und ausführen
- Einzelschritte, Start, Stopp
- Register
- Ports
- Flags anzeigen
- Interrupt
- Hilfe anzeigen
- externer Takt

Die Kann Kriterien wurde aus zeitlichen Gründen nicht umgesetzt. Diese waren:

- Funktionsgenerator
- 7 Segment Anzeige

Die Abgrenzungen müssen soweit weiterhin eingehalten werden. Der Simulator funktioniert nur mit einem korrekt funktionierenden Programmcode, und er übernimmt keinerlei Aufgaben eines Compilers.

4.2 Fazit

Durch verschiedene Erfahrungslevel im Bereich Software Entwicklung teilten wir die Aufgaben von Beginn an untereinander auf wie z.B. GUI Programmierung, Prozessor Programmierung und Dokumentation. Zeitaufwendigere Abschnitte wurden jedoch zusammen realisiert (wie z.B. Befehle ausprogrammieren). Durch Projekte in anderen Studiengängen konnten wir einige Erfahrungen zwischen den Projekten im Bereich Entwicklungsumgebung oder Versionierung (Git) austauschen. Zeitlich lag das Projekt parallel zu einem anderen im Fach Software Engineering wodurch es oftmals durch unerwarteten Problemen die zunächst aufwendiger erschienen als sie wirklich waren die Zeit knapp. Jedoch konnten Zum Schluss alle zuvor im Pflichtenheft bestimmten Muss-Kriterien erfüllt werden.