DHBW
Duale Hochschule
Baden-Württemberg
Karlsruhe

Simulator fr den Mikrocontroller PIC16F84A

von

Michael Stahlberger und Julian Khn

Studiengang Informationstechnik
an der Dualen Hochschule Karlsruhe

Inhaltsverzeichnis

Abbildungsverzeichnis

Listings

Kapitel 1

Einleitung

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

1.1 Was ist ein Simulator?

Eine Simulation ist ein mglichst realittsnahes Nachbilden von Geschehen der Wirklichkeit. Aus Sicherheits- und Kostengrnden ist es fr fast alle Anwendungsgebieten notwendig. Die gewonnenen Erkenntnisse knnen nach einer Simulation auf die Realitt bertragen werden. Eine Simulation findet meistens nicht in Echtzeit statt (wie z.B. bei einer Emulation) sondern wird zu analytischen Zwecken langsamer als in der Realitt nachgebildet.

1.2 Vor- und Nachteile eines Simulators

Vorteile: Durch eine Simulation knnen Versuche die unter gefhrlichen Umstnden stattfinden mssen sicher nachgestellt werden (z.B. Crash-Simulationen mit Autos und Crash-Test-Dummys). Aber auch Versuche die aus Kostengrnden in der Realitt oftmals schwierig nachzustellen sind knnen durch Simulationen begrenzt ersetzt werden. Durch den verlangsamten Ablauf einer Simulation sind auerdem Fehler oder Ergebnisse leichter nachzuvollziehen als in der Wirklichkeit. Im Falle des Mikrocontrollers knnen Programme

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

Nachteile: Eine Simulation ist meist durch begrenzte Ressourcen eingeschrnkt. Sei es die Rechenleistung einer Computersimulation oder Geld und Zeit die fr eine Simulation eingesetzt werden mssen. Oftmals wird deswegen nur ein vereinfachtes Modell der Wirklichkeit eingesetzt. Durch diese Vereinfachung kann es zu ungenauen Messergebnisse oder Situationen kommen die in der Realitt vielleicht gar nicht vorkommen. Fr den PIC16-Simulatior ist es wichtig mglichst fehlerfrei und genau zu arbeiten da Fehler innerhalb der Simulation auf falsche Rckschlsse auf das fr den Mikrocontroller entwickelte Programm fhren knnte. Auch zu bedenken ist es das die Laufzeit in der Simulation nicht der Realzeit entspricht und somit das Programm in der Realitt schneller sein wrde.

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 auen nicht sichtbar. Ebenso verfgen sie, im Gegensatz zum PC, nicht ber eine direkte Bedien- und Prorgrammierschnittstelle 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 groe Befehle whrend die Gre des separaten Datenbusses nur 8 Bit betrgt.

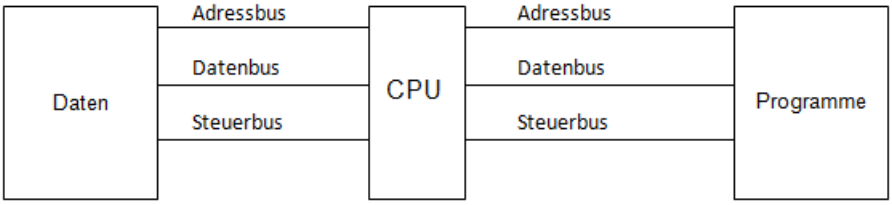


Abbildung 1.1: Harvard-Architektur

Durch die Architektur bentigen fast alle Anweisungen nur einen Instruction Cycle (Abarbeitung eines Maschinenbefehls). Der PIC16 besitzt einen Stack mit Speicherplatz fr 8 Adressen sowie 2 externe und 2 interne Interrupt Quellen. Darber hinaus besitzt der Pic16F ein groes Register, welches in zwei Bnke unterteilt ist. Das Umschalten der Bnke erfolgt im Programmcode. Die Speicherbereiche knnen auch direkt ber ihre Registeradresse angesprochen werden.

Kapitel 2

Tools

2.1 Entwicklungsumgebung Eclipse

Eclipse ist eine freie Entwicklungsumgebung welche ursrpnglich fr die Sprache Java entwickelt wurde. Mittlerweile gibt es Eclipse Plug-ins fr weitere Programmiersprachen wie C, C++ oder Pearl. Wie die meisten Entwicklungsumgebungen bietet Eclipse eine Vielzahl, dem Entwickler ntzlicher, Funktionen. Dazu gehren das Debuggen des Programmcodes, automatische Erstellen von Get- und Set- Methoden sowie automatische Codevervollstndigung. ber automatische Kontexthilfe liefert Eclipse Vorschlge um Fehler zu beheben. Eclipse kann leicht durch den Eclipse Marketplace mit verschienden Plug-Ins erweitert werden. Fr die Verbindung von Eclipse mit dem eingesetzten Versionsverwaltungs System Git wurde das Plug-In EGit verwendet.

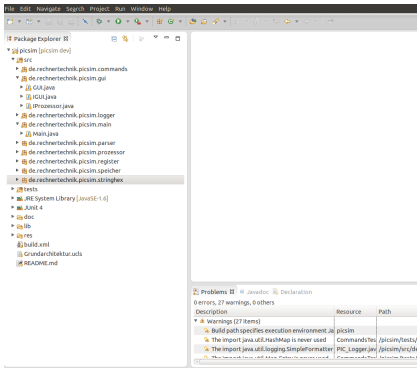


Abbildung 2.1: Eclipse

2.2 Versionsverwaltung Git

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

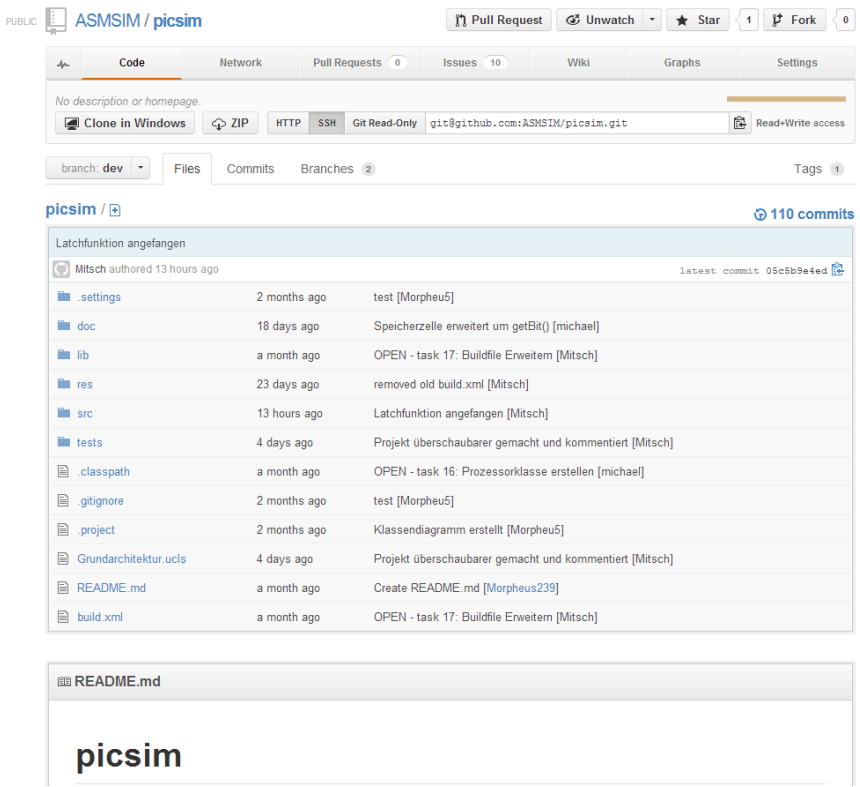


Abbildung 2.2: Github

Kapitel 3

Programmstruktur und Aufbau

3.1 Benutzeroberflche

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

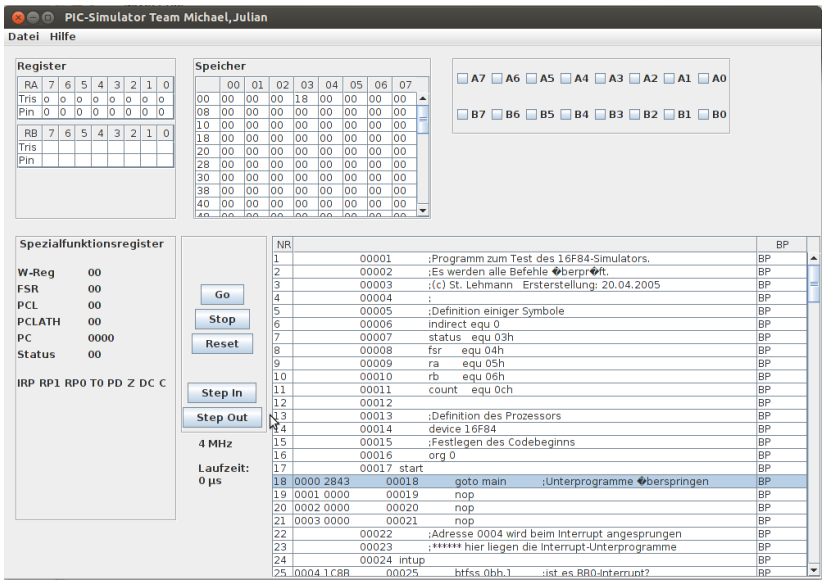


Abbildung 3.1: GUI

ber Datei und Datei ffnen lassen sich Quellcode-Dateien ffnen. In die Dateiauswahl ist ein Dateifilter integriert der ausschlielich .LST-Dateien anzeigt.

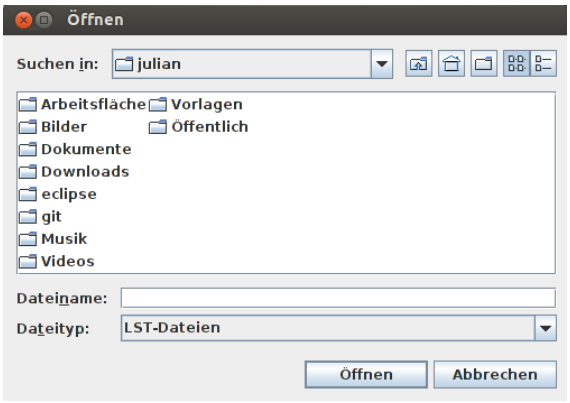


Abbildung 3.2: Datei ffnen

Mit den Buttons SStep-In", SStep-Out", "Go", SStopünd "Reset"lsst sich der Simulationsablauf steuern. Mit Step-In und Step-Out lsst sich jeweils nur ein Programmschritt ausfhren, mit Go wird das Programm komplett abgearbeitet. Mit Stop stoppt der Simulationsvorgang und mit Reset wird er komplett zurckgesetzt.

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.

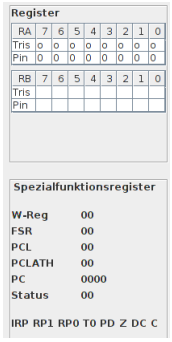


Abbildung 3.4: Register und Spezialfunktionsregister

In der rechten unteren Ecke wir der Quelltext des zu simulierenden Programmes angezeigt. Dieser wir wie beschrieben ber "Datei ffnenïn die Tabelle geschrieben.

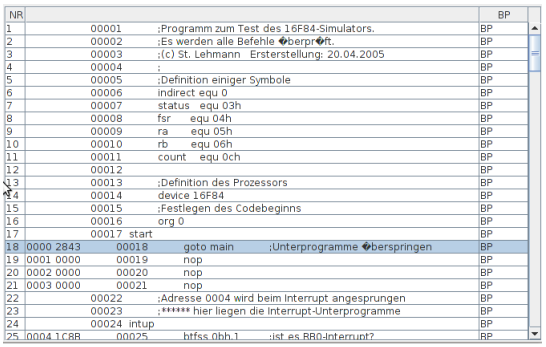


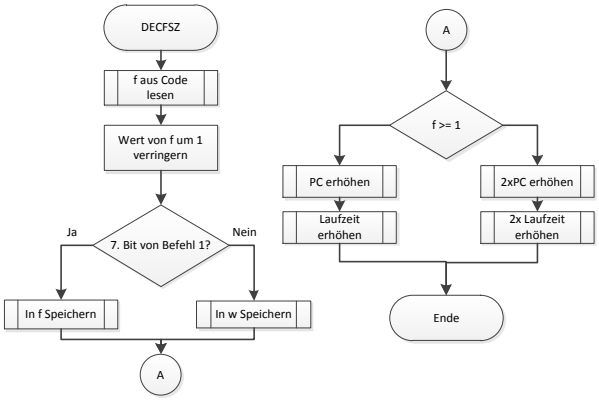
Abbildung 3.5: Textfeld fr den Quelltext

3.2 Programmablauf

3.3 Beschreibung einiger Befehle

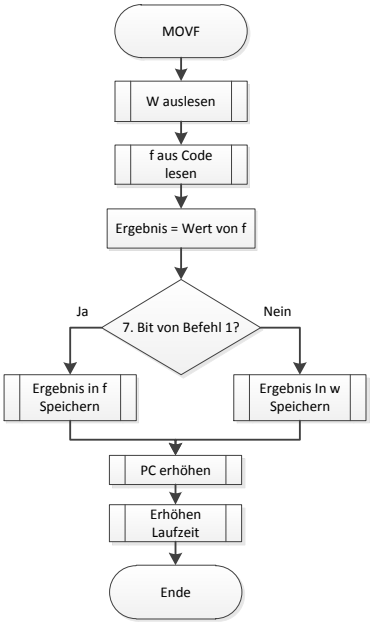
3.3.1 DECFSZ

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



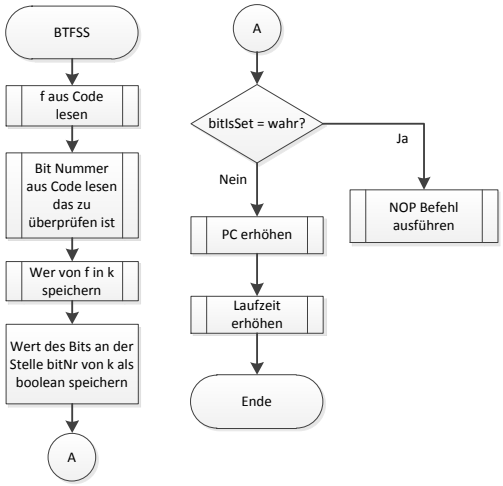

```
1  public static void asm_decfsz(Integer akt_Befehl, Prozessor cpu) {

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   }
```

3.3.2 MOVF



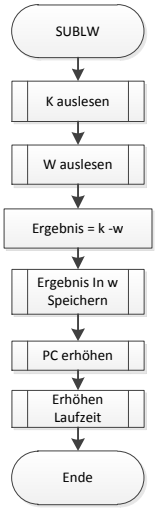

```
1  public static void asm_movf(Integer befehl, Prozessor cpu) {
2      Integer w = cpu.getW();
3      Integer f = getOpcodeFromToBit(befehl, 0, 6);
4      Integer result = cpu.getSpeicherzellenWert(f);

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   }
```

3.3.3 BTFSS



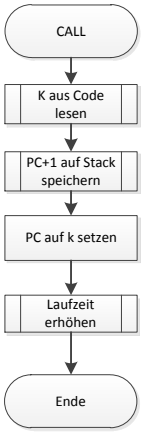

```
1  public static void asm_btfss(Integer akt_Befehl, Prozessor cpu) {

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

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

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

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

22   }
```

3.3.4 SUBLW



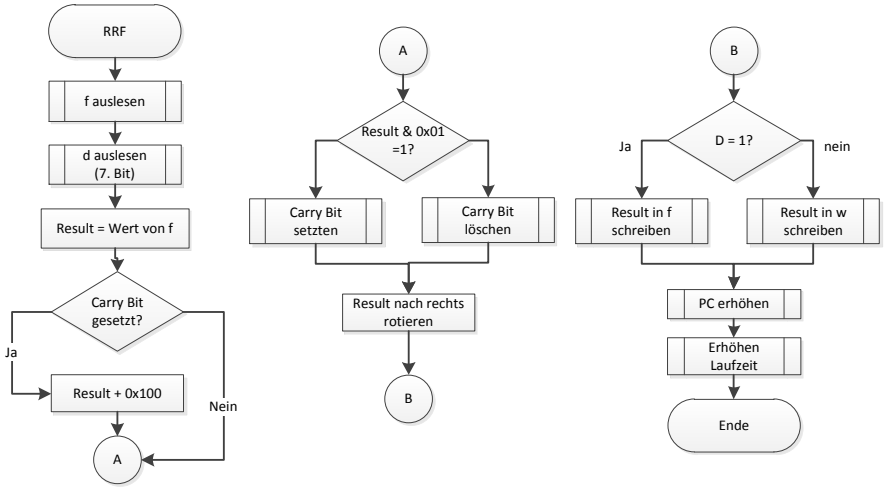

```
1  public static void asm_sublw(Integer akt_Befehl, Prozessor cpu) {
2      // Extrahiere K
3      Integer k = getOpcodeFromToBit(akt_Befehl, 0, 7);

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   }
```

3.3.5 CALL



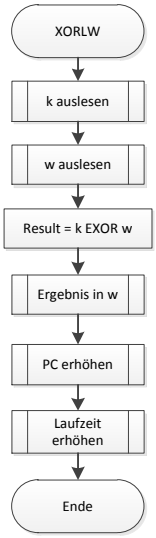

```
1  public static void asm_call(Integer akt_Befehl, Prozessor cpu) {

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   }
```

3.3.6 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 verlsst der PIC16F seine normale Routine und springt in eine Interruptroutine, die er abarbeitet um dann wieder an die Stelle des normalen Ablaufs zurckzukehren.
Im Simulator wird zwischen jedem Funktionsaufruf der einen Befehl beschreibt berprft 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;

11     //Global Interrupt enabled
12     if(GIE){

14       //RB0 Interrupt enabled
15       if(INTE){
16         PIC_Logger.logger.info("RB0 Interrupt check...");

18         Integer PortB = cpu.get_RAM_Value(0x06);
19         boolean RB0 = ( ( PortB & 0x01) == 0x01 )?true:false;

21         //TODO rising oder faling edge
22         Integer OPTION_REG  = cpu.get_RAM_Value(0x81);
23         boolean INTEDG  = ( ( OPTION_REG & 0x40) == 0x40 )?true:false;  //true = rising, fa

25           if(oldValue == false && RB0 == true && INTEDG){
26             externerInterruptRB0(cpu, INTCON, RB0);
27           }

29           else if(oldValue == true && RB0 == false && !INTEDG){
30             externerInterruptRB0(cpu, INTCON, RB0);
31           }

33           //No Interrupt
34           else{
35           }

37           //Save old Value
38           oldValue = RB0;
39       }
40     }
41   }
```

Bevor der Interrupt ausgefhrt wird muss zuerst geprft werden ob das GIE (Global Interrupt enable) und das INTCON Bit gesetzt worden ist.

3.5 TRIS-Register

3.6 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 gehrten:

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

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

- Funktionsgenerator
- 7 Segment Anzeige

Die Abgrenzungen mssen soweit weiterhin eingehalten werden. Der Simulator funktioniert nur mit einem korrekt funktionierenden Programmcode, und er bernimmt 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. Zeitaufwndigere Abschnitte wurden jedoch zusammen realisiert (wie z.B. Befehle ausprogrammieren). Durch Projekte in anderen Studienfchern 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 zunchst aufwendiger erschienen als sie wirklich waren die Zeit knapp. Jedoch konnten Zum Schluss alle zuvor im Pflichtenheft bestimmten Muss-Kriterien erfllt werden.