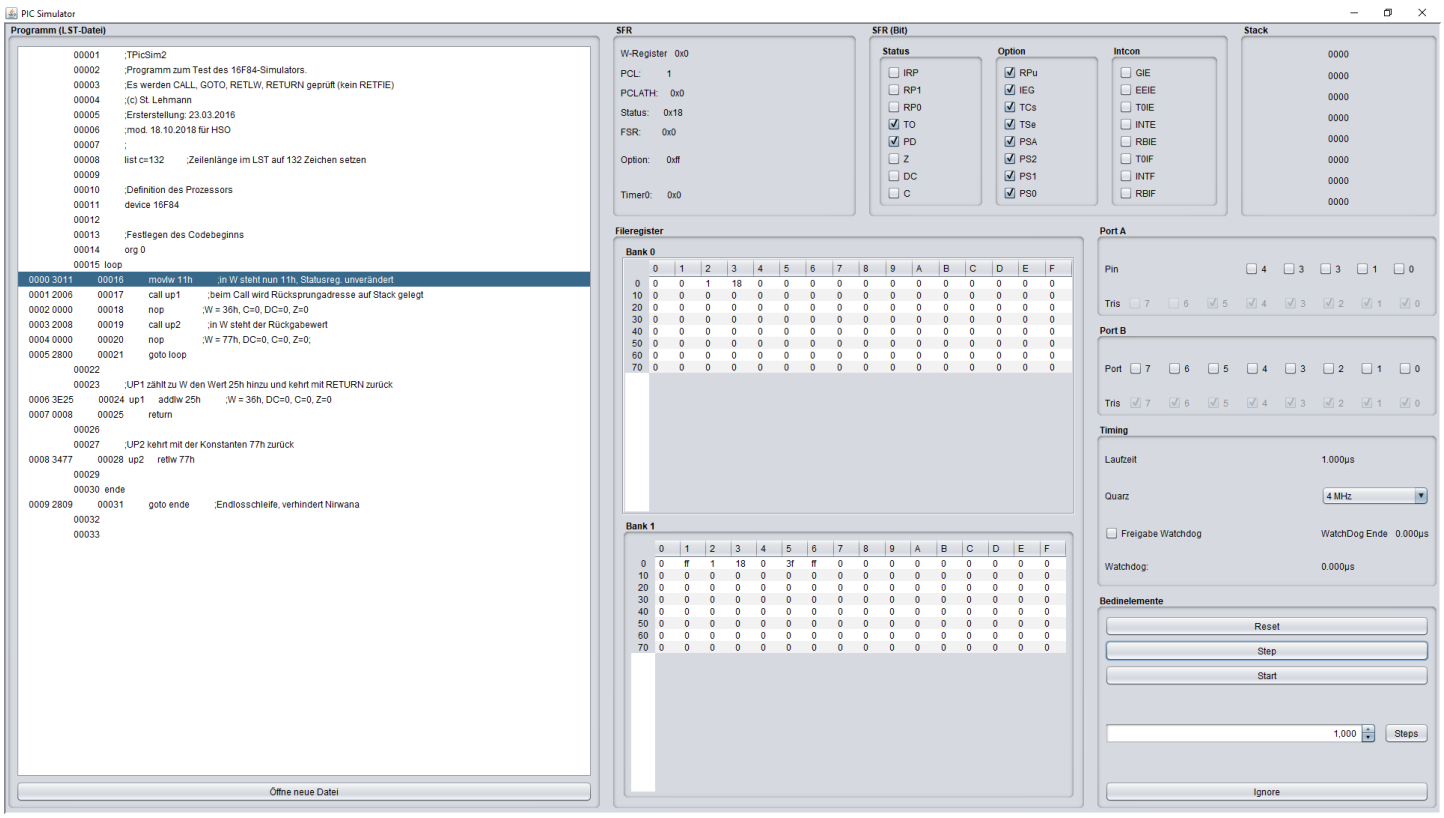
SIMULATOR PIC16F84

Dokumentation



Abstract

Die Dokumentation eines PIC16F84 Simulators in Java. Erklärungen zur Nutzung, Funktion und Umsetzung.

Rechnerarchitektur

Hochschule Offenburg

bei Lehmann, Stefan, Dipl.-Ing. (FH)

Sommersemester 2021

Anton Kesy, Michael Antropov

Inhaltsverzeichnis

Einleitung Simulator

Ein Simulator ist eine spezifische Nachbildung, welche möglichst realitätsnah realisiert ist. Dadurch können grundlegende und spezielle Nutzung des Simulierten nachgebildet werden ohne das Simulierte zu benötigen. Auch kann kontrolliert der Simulator abgeändert werden, um spezifische Situationen abzubilden. Der Vorteil ist, dass die Simulation unter vollständiger Kontrolle und abgestimmten Verhältnissen laufen kann. Somit kann genaustens studiert werden, was wie funktioniert oder ähnliches. Auch für Testen und lernen der Nutzung kann ein Simulator sehr nützlich sein. Doch ist eine Simulation oft nicht mehr als eine Simulation und kann die Realität nicht perfekt abspiegeln. Die Komplexität der Entwicklung von Simulatoren steigt auch fast schon exponentiell, wenn das Simulierte komplexer ist.

In diesem Projekt wurde ein Simulator für den Microcontroller PIC16F84 in Java realisiert. Die Programmiersprache konnten wir nicht wählen, aber da wir beide die meiste Erfahrung mit Java zu dieser Zeit hatten, hätten wir auch Java zur Entwicklung gewählt. Die Funktionalität des PIC16F84 wurde nicht vollständig in diesem Simulator realisiert. Dennoch sind Grundfunktionalität vollständig simulierbar. Alle Instruktionen sind unterstützt und somit können einfache Programme ohne Abweichung abgespielt werden. Interrupts, PCLATH, Timer und Watchdog mit dem Prescaler sind auch simulierbar.

[1]

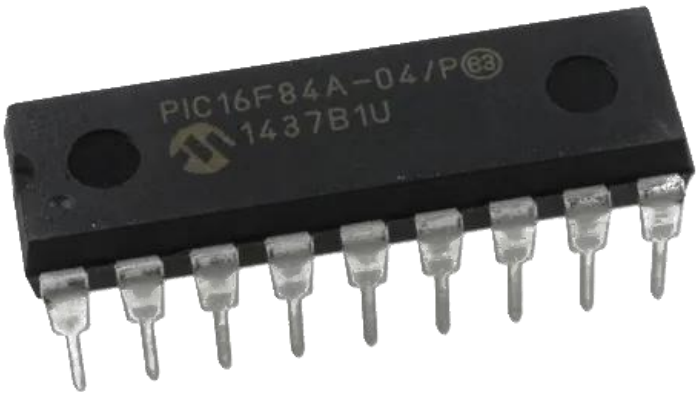


Abbildung 1 PIC1684A

[1] Bild von https://articulo.mercadolibre.com.ar/ 13.Mai.2021

Funktion der Programmoberfläche

Die Benutzeroberfläche bietet eine detaillierte Einsicht in den genauen Ablauf den Microcontrollers. Es können die Befehle manuell abgearbeitet werden. Auch können die Register und Flags jederzeit manipuliert werden.

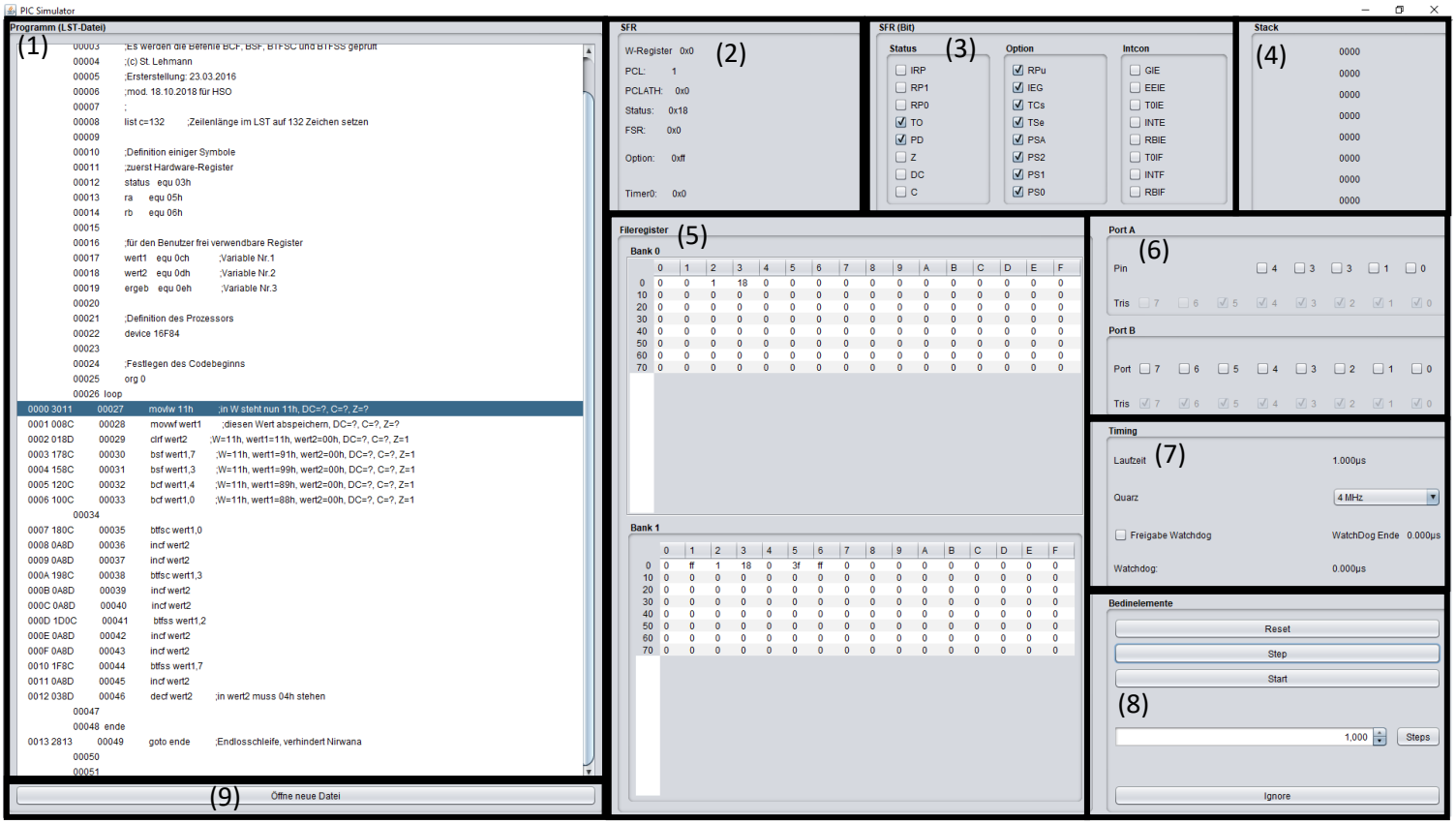


Abbildung 2 Komplettansicht Simulator

1. LST-Programm Vorschau
2. Wichtige Werte
3. SFR Bits
4. Stack
5. Fileregister in 2 Banken
6. Port A & Port B
7. Timing
8. Schrittbedienelemente
9. Auswahl der LST

Nach dem Öffnen des Programmes muss zunächst ein Programm im LST Format ausgewählt werden (9). Der File-Reader interpretiert das ausgewählte Programm und der Simulator erstellt einen neuen PIC mit dem Programm im Speicher geladen. Jetzt können die Einstellungen, wie zum Beispiel die Quarzfrequenz (7), angepasst werden. Der PIC steht noch auf Reset und wartet diesen und das zugehörige NOP durchzuführen. Danach kann mit Einzel-, N-Schritten oder automatisch der Simulator gesteuert werden (8).

LST-Programmvorschau

In diesem Fenster wird die geladene LST-Datei als Text angezeigt. Die markierte Zeile zeigt auf den nächsten Befehl an, welche der PIC ausführen würde bei einem Schritt. Durch Doppelklick auf eine Zeile kann ein Breakpoint erstellt oder entfernt werden. Der Simulator stoppt bei automatischem Durchlauf oder N-Schritten vor diesem Breakpoint.

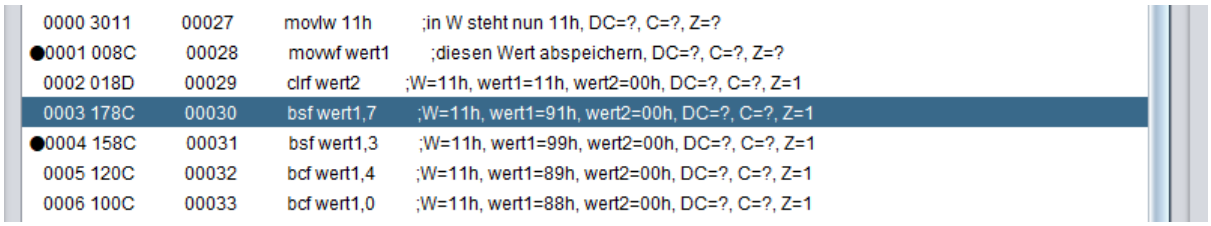


Abbildung 3 Schwarze Punkte stellen Breakpoints dar, blau markiert ist der nächste Schritt

Wichtige Werte

Hier sind alle relevanten und oft genutzten Werte dargestellt, wie der Programmzähler, der PCLATH und das W-Register.

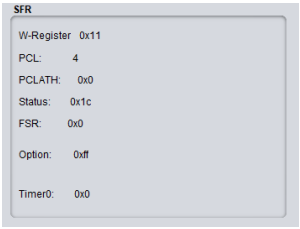


Abbildung 4 Wichtige Werte

SFR Bits

In diesem Fenster werden die SFR Bits angezeigt und durch betätigen der Checkboxen können diese auch direkt manipuliert werden. Die Bedeutung der einzelnen Bits ist im Anhang unter SFR Bits anhand des Datenblattes erklärt.

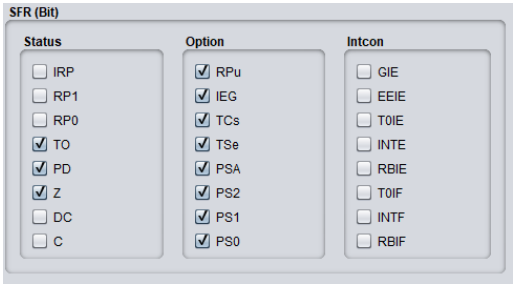


Abbildung 5 SFR Bits

Stack

Der Stack ist durch eine einfache List von acht freien Adressen zum ablegen abgebildet. Beim Legen in den Stack ändern sie die Werte dementsprechend.

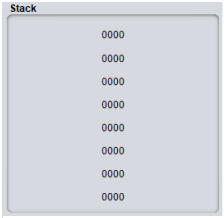


Abbildung 6 Stack

Fileregister in 2 Banken

Der PIC16F84 hat nur zwei RAM Bänke zur Verfügung. Diese sind durch zwei Tabellen, welche jeweils 128 Adressen halten, abgebildet. Durch Doppelklick auf die Werte, können diese einfach und direkt manipuliert werden. Die Aufteilung des Speichers kann aus dem Datenblatt im Anhang entnommen werden.

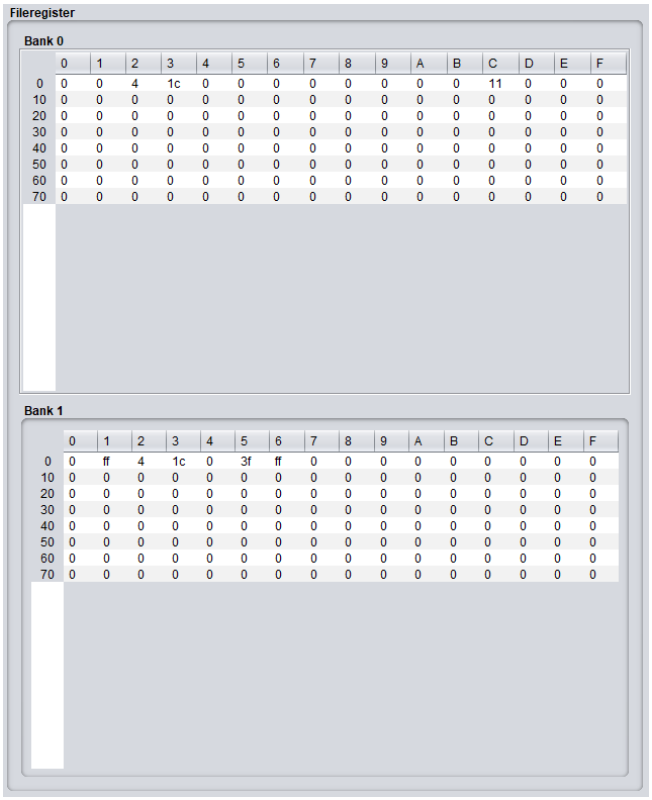


Abbildung 7 FileRegister

Port A & Port B

Hier sind der Port A & Port B durch Checkboxen abgebildet. Darunter ist direkt das zugehörige Tris-Register, welches entscheidet, ob der Port-Pin ein Eingang oder Ausgang ist. Checken der Box gilt als Hochflanke.

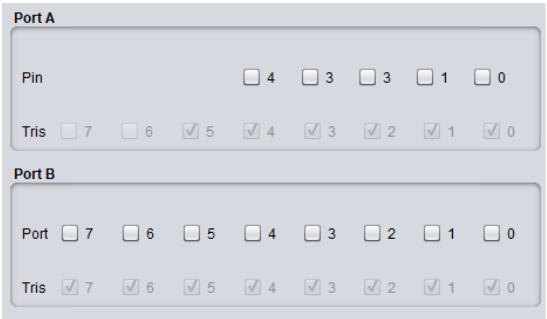


Abbildung 8 Port A & Port B

Timing

In dieser Gruppierung ist alles Wichtige, was mit der Laufzeit in Verbindung gebracht werden kann, abgebildet. Es kann, die Laufzeit des Microcontrollers in Microsekunden abgelesen werden, die Quarzfrequenz manuell einstellen, den WatchDog freigeben und auch dessen Laufzeit und Endzeit ablesen.

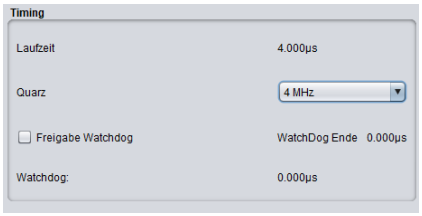


Abbildung 9 Timing

Schrittbedienelemente

Hier wird der gesamte Simulator mit Schritten gesteuert. Die Reset-Taste setzt den PIC auf den Startwert zurück. Mit Step kann ein einzelner Schritt betätigt werden. Start/Stopp kontrolliert den automatischen Ablauf von Schritten. Mit N-Steps können die eingetragenen Schritte direkt betätigt werden. Ignore ignoriert den nächsten Schritt.

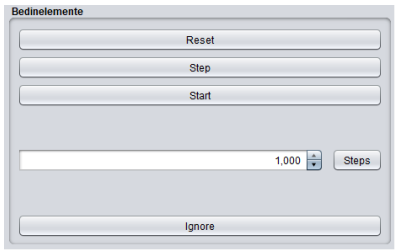


Abbildung 10 Schrittbedienung

Auswahl der LST

Um den PIC zu initialisieren muss er mit einem gültigen Programm geladen werden. Dies erfolgt durch LST-Dateien. Durch betätigen der Auswahl öffnet sich der File-Chooser und es kann die LST-Datei, welche eingespielt werden soll, ausgewählt werden.

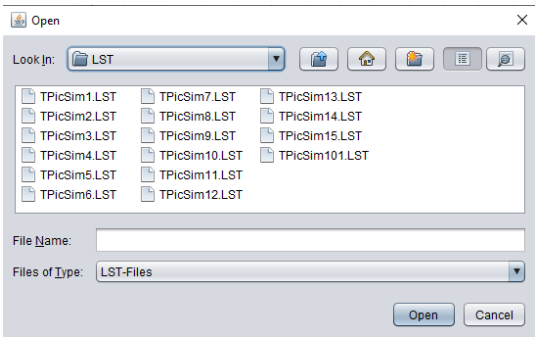


Abbildung 11 FileChooser

Gliederung des Projektes

Die äußerste Trennung ist zwischen der GUI, File-Reader, Tests und dem PIC-Simulator, welche aber dennoch miteinander verbunden sind. In der GUI-Gruppierung sind alle GUI relevanten Objekte, wie Java Swing Komponenten oder der Thread zur automatischen Abarbeitung der CPU, vorhanden. Die PIC Gruppierung ist unter PIC und seine Elemente, wie die ALU, Stack oder RAM eingeteilt und Objekte welche zur Gruppierung Instruktionen gehören, wie zum Beispiel eine Datenklasse für die Instruktionen und die Arbeitsklasse für den Decoder.

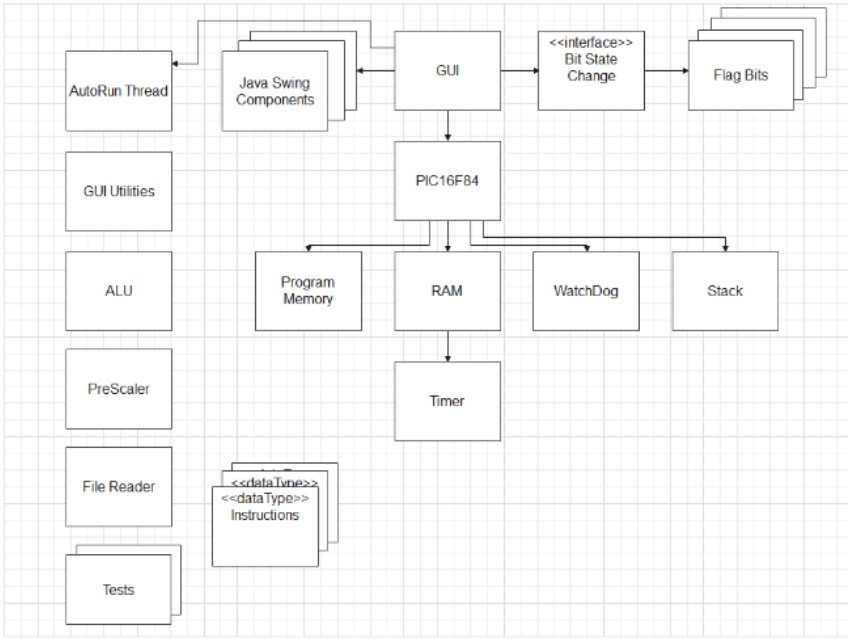


Abbildung 13 Vereinfachtes UML des Projekts

Hier eine vereinfachte Darstellung der Programmstruktur in einem UML Diagramm. Die Benutzeroberfläche (GUI) steuert den gesamten Simulator. Auch besitzt die Benutzeroberfläche eigene Java Swing Komponenten, einen Thread, welchen den Simulator automatisch ablaufen lässt und die Implementierungen eines Interfaces Bit State Change, welche die Status Checkboxen steuern. Durch Benutzereingabe kann der PIC über das GUI gesteuert werden. Es können Zyklen abgearbeitet werden, die Daten ausgelesen und manipuliert werden. Der PIC besitzt mehrere Bausteine. Diese sind: Der Programmspeicher, in welchen die vom FileReader interpretierten LST-Programmdatei als Liste von Befehlen abgespeichert wird. Der Random Access Memory, welcher die Daten als Integer in einem zweidimensionalen Array von der Länge 128 hält und nebenbei noch eine Hilfsklasse, welchen den Timer repräsentiert. Den WatchDog und den Stack, von der Länge acht. Außerhalb dieser Verknüpfung gibt es noch einige Arbeitsklassen, wie die Arithmetische Logische Einheit (ALU), PreScaler für den Timer oder WatchDog, den FileReader zur Interpretation der LST-Dateien und eine Hilfsklasse für die Benutzeroberfläche.

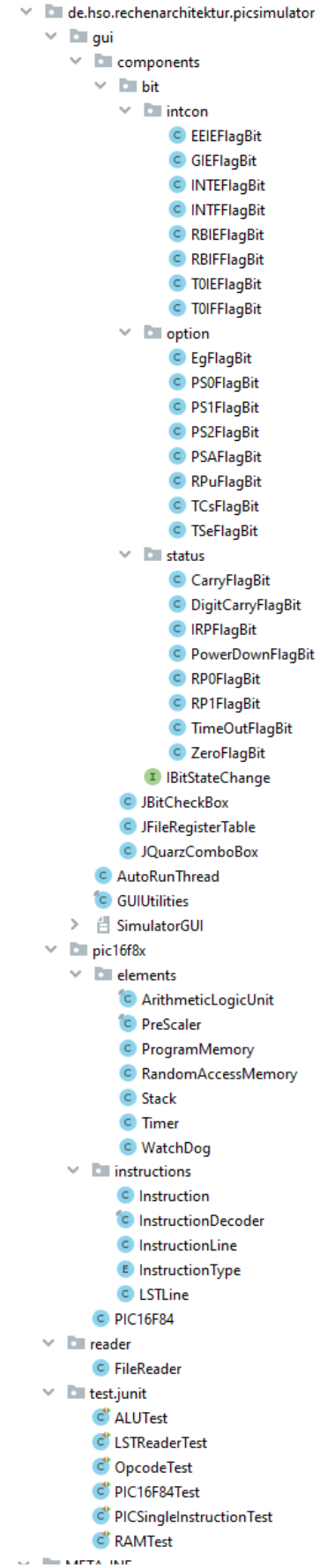


Abbildung 12 Komplette Quellcodehierarchie

Funktionsablauf

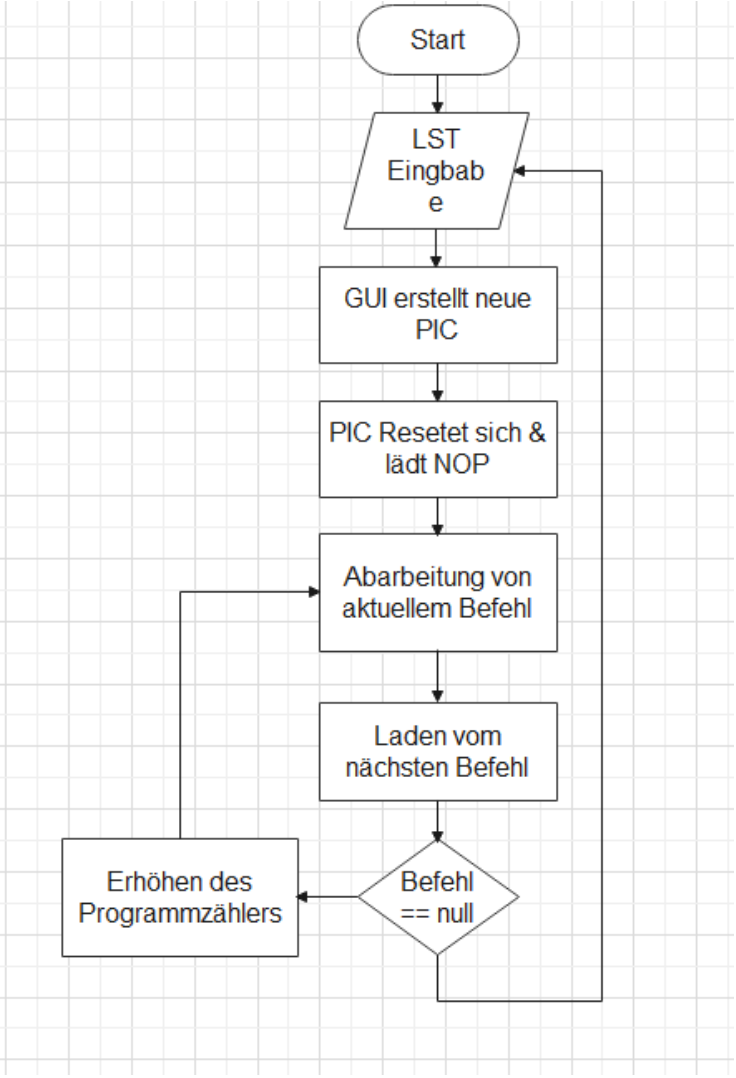


Abbildung 14 Vereinfachtes PIC Flussdiagramm

Der FileReader bekommt eine LST-Datei welche er Interpretiert und in eine Array-Liste von LSTLine und InstructionLine umwandelt. Der PIC16F84 wird mit der InstructionLine-Liste initialisiert und er lädt die Befehle in seinen Programmspeicher. Bei Initialisierung erstell der PIC auch seinen RAM, welcher mit den Standardwerten geladen wird, einen neuen Watchdog und führt danach direkt ein Reset aus. Der Microcontroller ist nun mit der NoOperation-Instruktion geladen. Wenn von der Benutzeroberfläche der nächste Schritt gerufen wird, dann wird die momentan geladene Instruktion abgearbeitet. Dies geschieht im InstructionHandler, welcher mithilfe eines Switchs den aktuellen Befehl auswertet und die auszuführenden Funktionen aufruft. Aktuell ist im PIC ein NOP geladen, welches bei der Ausführung keine Operation betätigt. Am Ende wird die Laufzeit noch um die benötigte Zeit addiert. Diese wird aus den Zyklen, welche der Befehlt benötigt und der Quarzfrequenz des Microcontrollers berechnet. Auch werden der Timer und der Watchdog, wenn eingeschalten, behandelt. Zuletzt wird die nächste Instruktion vorbereitet. Dafür wird zuerst nach Interrupt überprüft, danach der nächste Befehl geladen und der Programmzähler inkrementiert. Danach kann von der Benutzeroberfläche wieder ein Schritt getätigt werden.

Folgend sind noch die Befehle BTFSC/BTFSS, CALL, MOVF, RRF, SUBWF, DECFSZ, XORLW und ihre spezielle Abarbeitung, genauer als Beispiele erklärt.

BTFSx

BTFSC (Bit Test, Skip if Clear) und BTFSS (Bit Test f, Skip if Set) sind beide zusammen implementiert. Sie Testen das Bit an der Stelle f im W-Register ob es bei BTFSC gleich Eins oder bei BTFSS gleich 0 ist. Wenn dies wahr ist, dann wird der nächste Befehlt übersprungen und somit der Programmzähler um insgesamt Zwei erhöht.

```
case BTFSC:
    if (!isBitFActive(currentInstruction.getBD(), ram.getDataFromAddress(currentInstruction.getFK()))) {
        skipNextInstruction();
    }
    break;
case BTFSS:
    if (isBitFActive(currentInstruction.getBD(), ram.getDataFromAddress(currentInstruction.getFK()))) {
        skipNextInstruction();
    }
    break;
```

Abbildung 15 Codeausschnitt Befehlsswitch

```
private boolean isBitFActive(int b, int f) {
    return (f >> (b) & 1) == 1;
}
```

Abbildung 16 Codeausschnitt isBitFActive

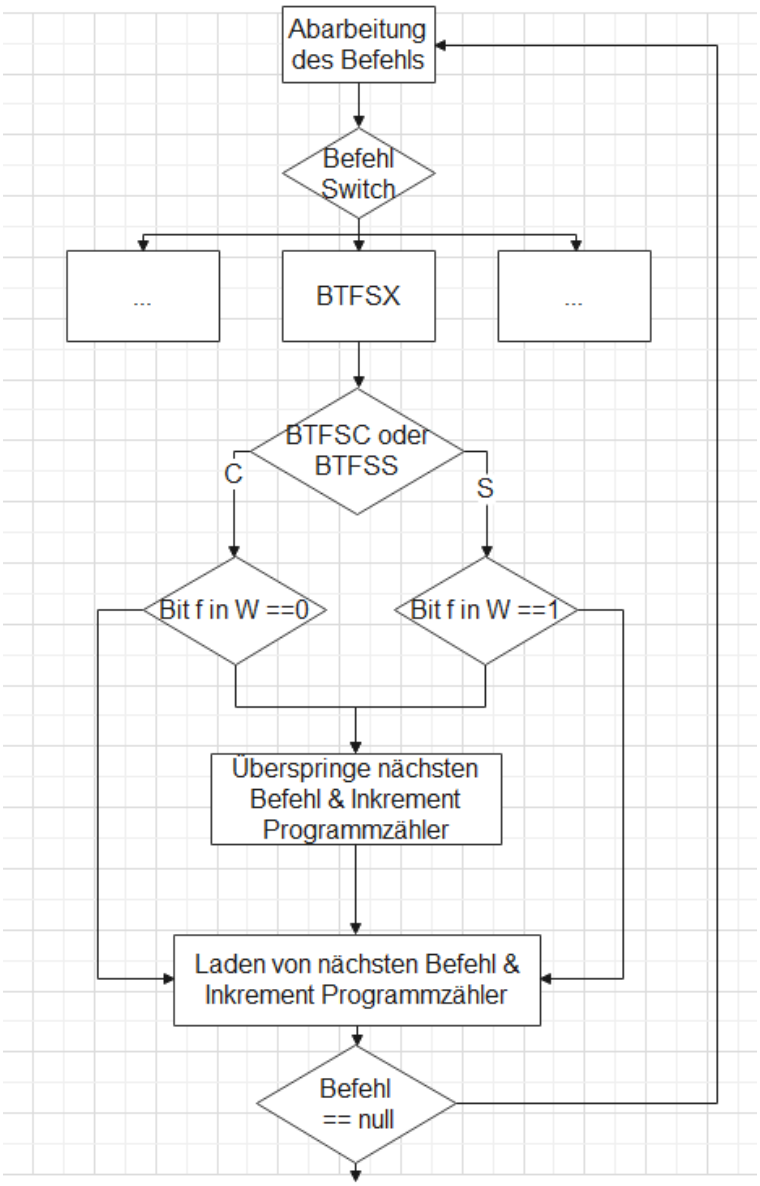


Abbildung 17 Flussdiagramm BTFSX

CALL

Mit dem Call (Call Subroutine) kann auf eine bestimmte Stelle im Programmspeicher gesprungen werden. Mit der Verknüpfung des PCLATH kann damit noch weiter als die eigentlich nur 256 Stellen gesprungen werden. Bei jedem Call wird der aktuelle Programmzähler, für den Return, in den Stack geschrieben. Danach wird der Programmzähler auf die elf Bit große literale Konstante und den PCLATH gestellt.

```
case CALL:
    cycles = 2;
    stack.push(getRam().getPCL());
    getRam().manipulatePCL(ram.getJumpAddress(currentInstruction.getFK()));
    break;
```

Abbildung 19 Codeausschnitt Befehlsswitch

```
public void manipulatePCL(int value) {
    int newPCLValue = 0;
    newPCLValue |= getPCLath();
    newPCLValue <<= 8;
    newPCLValue |= value;
    memory[2][0] = newPCLValue;
    memory[2][1] = newPCLValue;
}
```

Abbildung 18 Codeausschnitt manipulatePCL

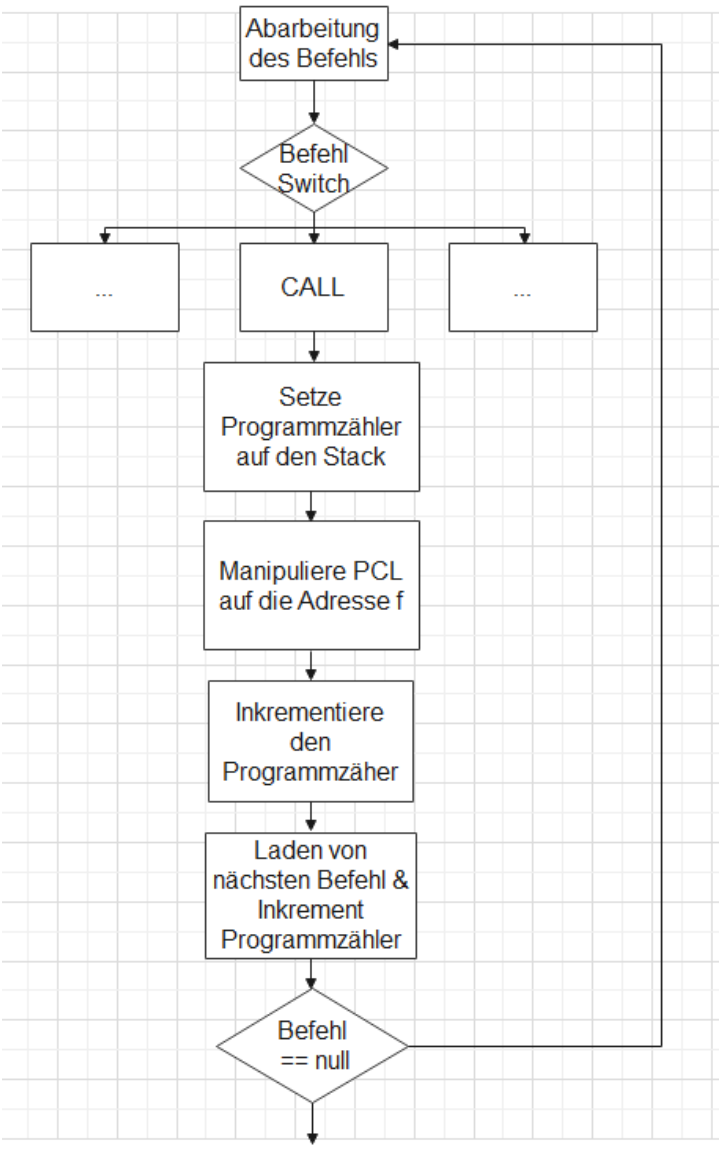


Abbildung 21 Flussdiagramm Call

```
public int getJumpAddress(int f) {
    int value = 0;
    //Add pcl to front bits
    int lath2 = getPCLath();
    lath2 >>= 3;
    lath2 &= 0b11;
    value |= lath2;
    //add f (11bits) to value
    value <<= 11;
    return value | f;
}
```

Abbildung 20 Codeausschnitt getJumpAdress

MOVF

MOVF (Move f) speichert die Daten aus der Adresse f dem Destination-Bit abhängig in das W-Register oder wieder in die Adresse f. Das Zero-Flag wird gesetzt, wenn der Wert Null ist.

```
case MOVF:
    int valueOfAddress = ram.getDataFromAddress(currentInstruction.getFK());
    ram.setZeroFlag(valueOfAddress == 0);
    setResultInDestination(currentInstruction.getBD(), currentInstruction.getFK(), valueOfAddress);
    break;
```

Abbildung 22 Codeausschnitt Befehlsswitch

```
private void setResultInDestination(int d, int f, int value) {
    if (d == 0) {
        wRegister = value;
    } else {
        ram.setDataToAddress(f, value);
    }
}
```

Abbildung 23 Codeausschnitt Destination Bit Auswerter

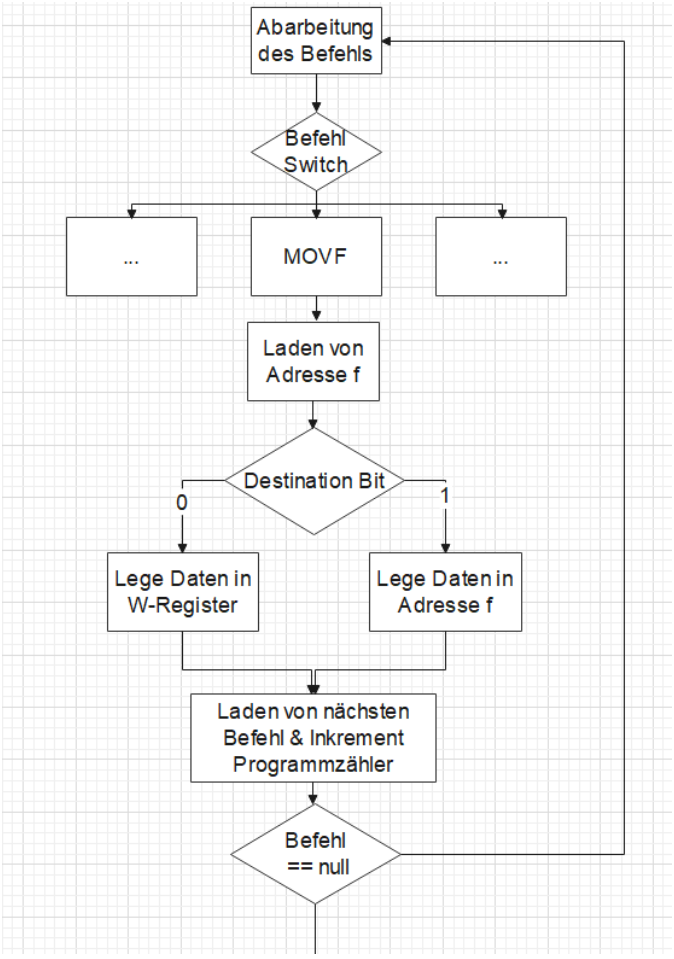


Abbildung 24 Flussdiagramm MOVF

RRF

RRF (Rotate Right f through Carry) rotiert die Daten auf der Adresse f im RAM um einen Bit nach rechts durch den Carry durch. Wenn das Bit d (Destination Bit) auf 0 steht, wird das Ergebnis im W-Register platziert, wenn nicht, dann wieder auf der Adresse f.

```
case RRF:
    result = getRotateRightTroughCarry(ram.getDataFromAddress(currentInstruction.getFK()));
    setResultInDestination(currentInstruction.getBD(), currentInstruction.getFK(), result);
    break;
```

Abbildung 25 Codeausschnitt Befehlsswitch

```
private int getRotateRightTroughCarry(int f) {
    boolean isCarryFlagAfter = isBitFActive( b: 0, f);
    f >>= 1;
    if (ram.isCarryFlag()) {
        f |= 0b1000_0000;
    }
    ram.setCarryFlag(isCarryFlagAfter);
    return f & 0xFF;
}
```

Abbildung 27 Codeausschnitt Rotiere Rechts durch Carry

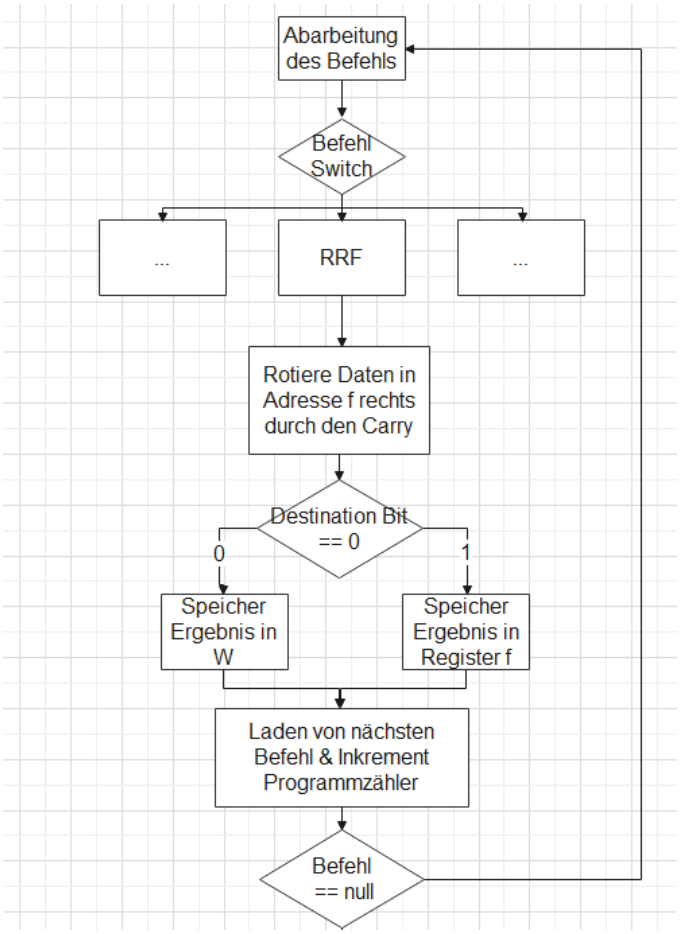


Abbildung 26 Flussdiagramm RRF

SUBWF

SUBWF (Subtract W from f) subtrahiert die Daten im W-Register mit den Werten auf der Adresse f. Die Subtraktion wird mit Hilfe des addieren des Zweierkompliments verwirklicht. Die Arithmetische Logische Einheit überprüft auch direkt bei der Addition, ob die relevanten Flags, hier Carry, Digit-Carry und Zero gesetzt werden müssen und setzt diese direkt, wenn nötig. Wenn das Bit d (Destination Bit) auf 0 steht, wird das Ergebnis im W-Register platziert, wenn nicht, dann wieder auf der Adresse f.

```
case SUBWF:
    result = ArithmeticLogicUnit.sub(ram, ram.getDataFromAddress(currentInstruction.getFK()), wRegister);
    setResultInDestination(currentInstruction.getBD(), currentInstruction.getFK(), result);
    break;
```

Abbildung 28 Codeausschnitt Befehlsswitch

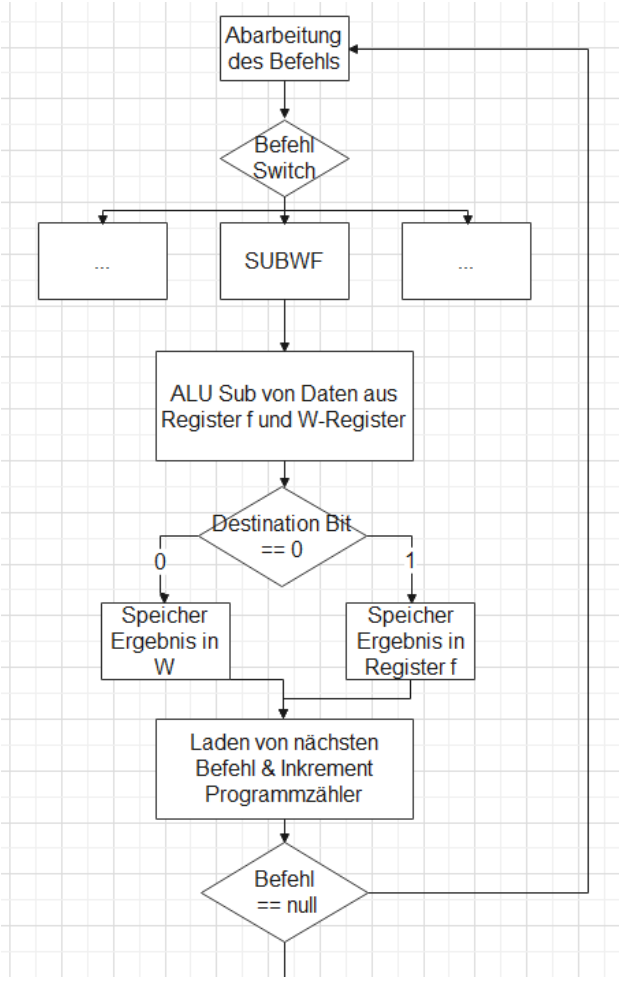


Abbildung 29 Flussdiagramm SUBWF

DECFSZ

DECFSZ (Decrement f, Skip if 0) dekrementiert die Daten aus Adresse f. Wenn das Bit d (Destination Bit) auf 0 steht, wird das Ergebnis im W-Register platziert, wenn nicht, dann wieder auf der Adresse f. Wenn das Ergebnis Null ist, dann wird der nächste Befehlt übersprungen.

```
case DECFSZ:
case INCFSZ:
    result = ram.getDataFromAddress(currentInstruction.getFK());
    result += currentInstruction.getType() == InstructionType.INCFSZ ? 1 : -1;
    result = setValueTo8BitAndSetZeroFlag(result);
    setResultInDestination(currentInstruction.getBD(), currentInstruction.getFK(), result);
    if (result == 0) {
        skipNextInstruction();
    }
    break;
```

Abbildung 30 Codeausschnitt Befehlsswitch

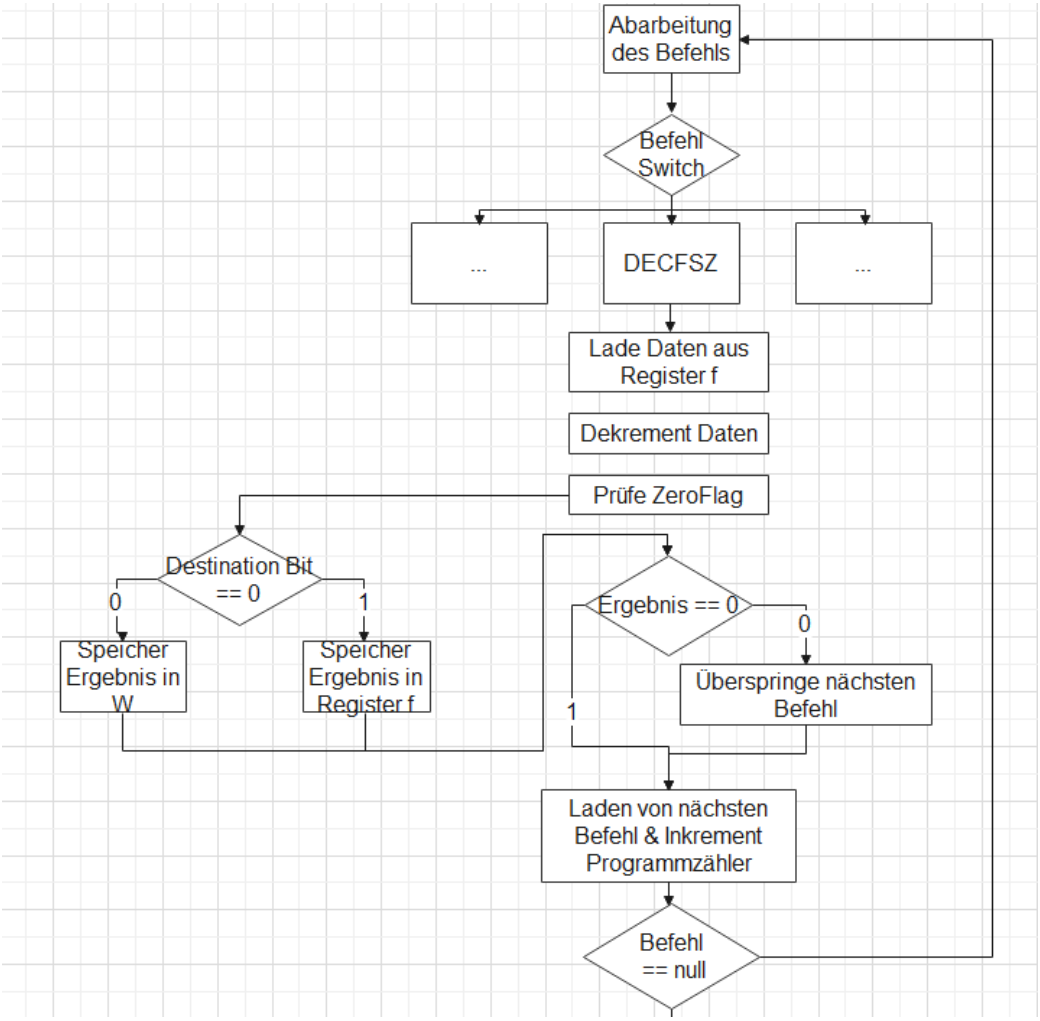


Abbildung 31 Flussdiagramm DECFSZ

XORLW

XORLW (Exclusive OR W with f) macht ein exklusives Oder mit den Werten aus dem W-Register und den Werten aus der Adresse f. Der ALU prüft bei dem XOR nach Flags und setzt, wenn nötig, das Zero-Flag. Wenn das Bit d (Destination Bit) auf 0 steht, wird das Ergebnis im W-Register platziert, wenn nicht, dann wieder auf der Adresse f.

```
case XORLW:
    wRegister = ArithmeticLogicUnit.xor(ram, wRegister, currentInstruction.getFK());
    break;
```

Abbildung 32 Codeausschnitt Befehlsswitch

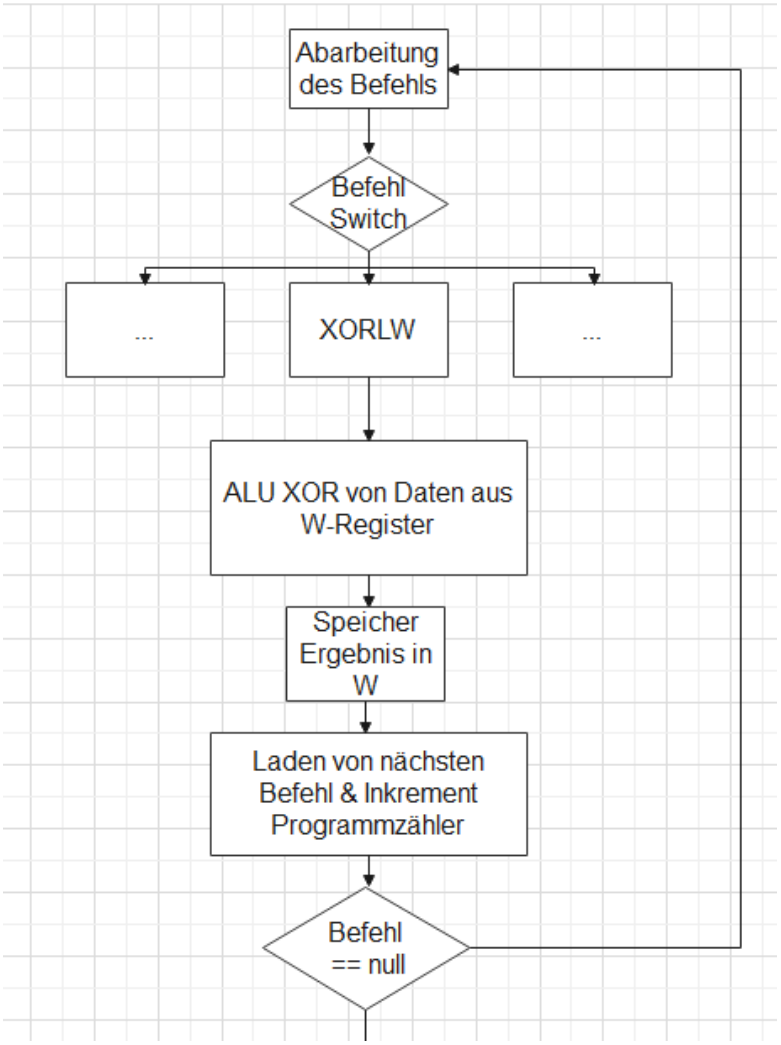


Abbildung 33 Flussdiagramm XORLW

Beschreibung einzelner Komponenten und Funktionen

Interrupts

Es gibt vier wichtige Interrupts bei dem Microcontroller: Den TMR0-Interrupt, welcher aktiviert wird, wenn der Timer überläuft. Den RB-Port-Change-Interrupt, welche sich aktiviert, wenn an den Pins 4-7 von Port B eine Änderung eintritt. Den RB0/INT-Interrupt, welcher durch flanken an Port B Pin 0 ausgelöst wird. Und den EE-Write-Complete-Interrupt, welcher gesetzt wird, wenn das EEPROM fertig mit schreiben ist. Letzteren Interrupt wurde mit dem EEPROM nicht in diesem Simulator realisiert. Alle anderen Interrupts werden bei jedem Zyklus überprüft, ob Interrupts generell aktiviert sind, durch das Global-Interrupt-Enable-Bit angezeit, und dann die einzelnen Flags der Interrupts.

Der TMR0-Interrupt wird bei jedem Timer-Wert-Wechsel überprüft. Die Port-B-Interrupts werden durch einen ActionListener in der Benutzeroberfläche gerufen. Beim aktivieren der Pins wird überprüft ob ein Interrupt aktiviert werden muss oder nicht.

Hier als Beispiel der RB-Port-Change-Interrupts

```
//Set PortB 4-7 change listener
for (int i = 4; i < 8; ++i) {
    final int index = i;
    portBPins[index].addActionListener(e -> pic.switchRB4_7(index));
}
```

Abbildung 35 Codeausschnitt setzen des ActionListener

```
public void switchRB4_7(int index) {
    //checks if RBIE & Port at Index is not output set by tris
    if (ram.isRBIE() && (ram.getTrisB() >> index & 1) == 1) {
        ram.setRBIF(true);
    }
}
```

Abbildung 36 Codeausschnitt Überprüfung auf RB4-7 Interrupt

```
private boolean checkForInterrupts() {
    boolean wasInterrupt = false;
    if (ram.isGIE()) {
        //Timer Interrupt
        if (ram.isT0IE() && ram.isT0IF()) {
            //Timer Interrupt
            stack.push(ram.getPCL());
            ram.setGIE(false);
            ram.setPCL(4);
            wasInterrupt = true;
        }
        //RB0 Interrupt
        if (ram.isINTF() && ram.isINTE()) {
            //Timer Interrupt
            stack.push(ram.getPCL());
            ram.setGIE(false);
            ram.setPCL(4);
            wasInterrupt = true;
        }
        //RB Interrupt
        if (ram.isRBIE() && ram.isRBIF()) {
            //Timer Interrupt
            stack.push(ram.getPCL());
            ram.setGIE(false);
            ram.setPCL(4);
            wasInterrupt = true;
        }
    }
```

Abbildung 34 Codeausschnitt der Interrupt Überprüfung

2

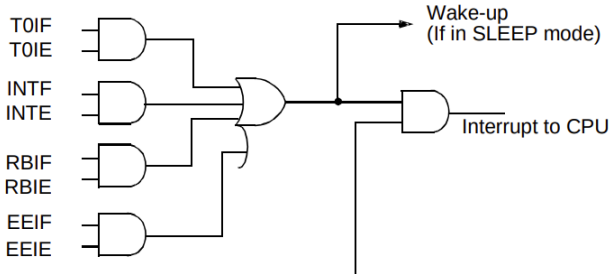


Abbildung 37 Interrupt Logic

[2] Interrupt Logic aus dem Datenblatt PIC16F8X von Microchip

Status-, Option und INTCON-Flags

Die einzelnen Register haben ihren eigenen Platz im Speicher. Das Statusregister ist auf 03h, das Optionregister auf 81h, also der zweiten Bank. Das INTCON steht auf 0Bh. Jedes Bit der einzelnen Adressen hat eine eigene Bedeutung, welche im Anhang auf dem Datenblatt ersichtlich ist. Das Bit 2 des Statusregisters zum Beispiel zeigt an, ob ein Zero-Flag gesetzt ist. Durch setzen der Flag erhöht sich der Wert des Statusregisters um 2^2, also um 4 und beim resetten verringert der Wert sich. Somit steht in den Registern eigentlich die Summe bzw. der Stellvertreterwert der Gruppierung der Flags.

In dem Simulator wurden diese auch genau so realisiert. Es können die Adressen direkt beschrieben und wie Werte genutzt werden, aber durch Hilfsmethoden können auch einfach nur die Flags verändert werden.

In der Benutzeroberfläche implementiert jeder CheckBox-Listener das BitChange-Interface welches den genauen Bit festlegt, welcher diese CheckBox repräsentiert.

```
public int getFlagBitsValue(boolean is0, boolean is1, boolean is2, boolean is3,
                            boolean is4, boolean is5, boolean is6, boolean is7) {
    int flagValue = 0;
    if (is0) {
        flagValue += 0b1;
    }

    if (is1) {
        flagValue += 0b10;
    }

    if (is2) {
        flagValue += 0b100;
    }

    if (is3) {
        flagValue += 0b1000;
    }

    if (is4) {
        flagValue += 0b1_0000;
    }

    if (is5) {
        flagValue += 0b10_0000;
    }
    if (is6) {
        flagValue += 0b100_0000;
    }

    if (is7) {
        flagValue += 0b1000_0000;
    }
    return flagValue;
}
```

Abbildung 38 Codeausschnitt setzte Flag Register Value

```
public boolean isZeroFlag() {
    return (getStatus() & 0b100) == 0b100;
}
```

Abbildung 39 Codeausschnitt isZeroFlag

FileReader

Der FileReader wird mit einer LST-Datei erstellt, welche er Zeile für Zeile interpretiert. Die LST-Dateien werden mit einem Scanner auf ISO_8859_1 geladen und per Zeile ausgelesen. Hier der Code für die Interpretation der übergebenden Zeile.

```
private void interpretLine(String line) {
    //Direkt abbrechen, wenn nicht mit einer Adresse beginnt
    if (line.startsWith(" ")) return;
    //Line splitten und Leerzeichen entfernen
    String[] lineSplit = Arrays.stream(line.split( regex: " ")).filter(t -> t.length() > 0).toArray(String[]::new);
    //Fuegt neue Instruktion mit allen noetigen Informationen in die Liste
    programMemoryMap.add(new InstructionLine(
            lines.size(), //aktuelle Line des Befehls im LST
            Integer.decode("0x" + lineSplit[0]), //Position im ProgramMemory
            InstructionDecoder.decodeInstruction(Integer.decode("0x" + lineSplit[1]))) //Opcode
    );
}
```

Abbildung 40 Codeausschnitt Interpreter der Zeilen

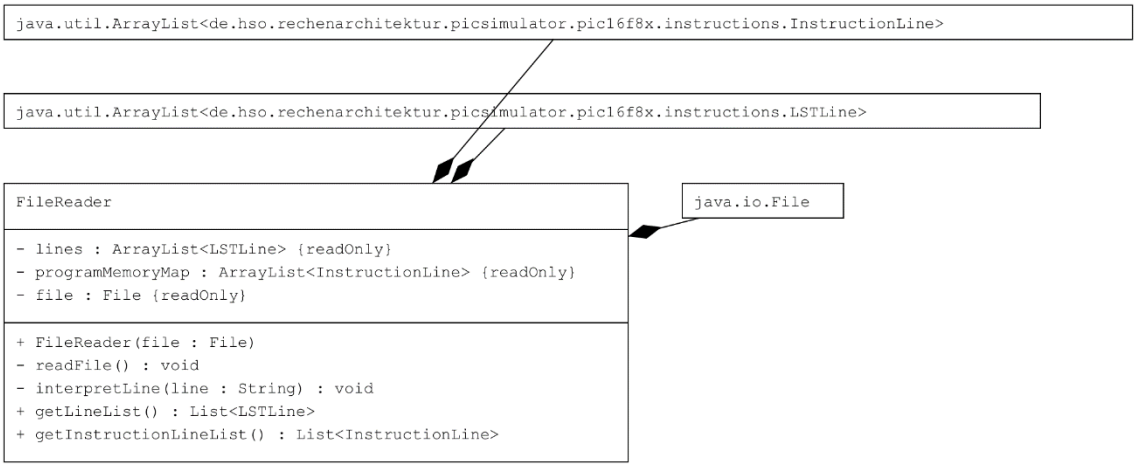


Abbildung 41 UML FileReader

Instruktion Decoder

Der Instruktion Decoder dekodiert die Hex-Werte des Programmcodes, den 14-Bit Opcode (operation code), in Befehle mit Hilfe von Bitmasken. Die zwei höchstwertigen Bits im Opcode entscheiden, welche Gruppierung die Operation besitzt und welche weiteren Bits ausgelesen und eingeteilt werden müssen.

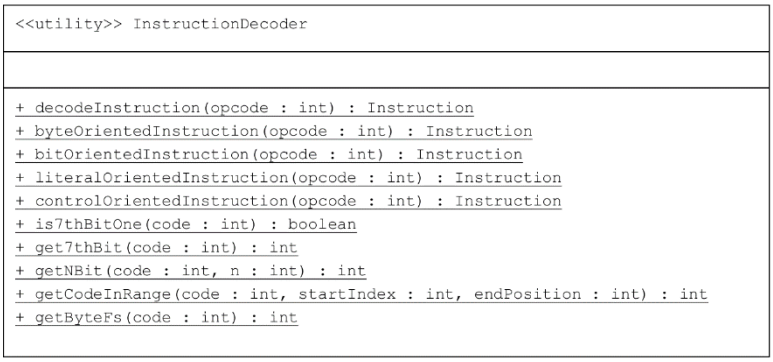


Abbildung 42 UML InstructionDecoder

```java
/**
 * Gibt die gefuellte Instruktion dem opcode entsprechend zurueck
 *
 * @param opcode
 * @return null wenn ungueltiger opcode
 */
public static Instruction decodeInstruction(int opcode) {
    //Fist two binary digits of opcode
    int opcodeMSbs = opcode >>> 12;
    Instruction result;
    //Decides what kind of instruction category opcode is
    switch (opcodeMSbs) {
        case 0b0:
            result = byteOrientedInstruction(opcode);
            break;
        case 0b01:
            result = bitOrientedInstruction(opcode);
            break;
        case 0b11:
            result = literalOrientedInstruction(opcode);
            break;
        case 0b10:
            result = controlOrientedInstruction(opcode);
            break;
        default:
            //Illegal opcode
            result = null;
    }
    return result;
}
```

Abbildung 44 Codeausschnitt der Gruppierung der Instruktionen

```java
/**
 * Gibt BIT-ORIENTED FILE REGISTER Instruktion dem opcode entsprechend zurueck
 *
 * @param opcode
 * @return
 */
public static Instruction bitOrientedInstruction(int opcode) {
    //3rd Byte from right to decide which instruction opcode is
    int instructionCode = opcode & 0b00_1100_0000_0000;
    instructionCode >>>= 10;
    InstructionType instructionType = null;
    switch (instructionCode) {
        case 0b0:
            instructionType = InstructionType.BCF;
            break;
        case 0b1:
            instructionType = InstructionType.BSF;
            break;
        case 0b10:
            instructionType = InstructionType.BTFSC;
            break;
        case 0b11:
            instructionType = InstructionType.BTFSS;
            break;
        default:
            System.out.println("Illegal instruction code " + Integer.toBinaryString(instructionCode));
            break;

    }
    return new Instruction(instructionType, getByteFs(opcode), getCodeInRange(opcode, startIndex: 7, endPosition: 10));
}
```

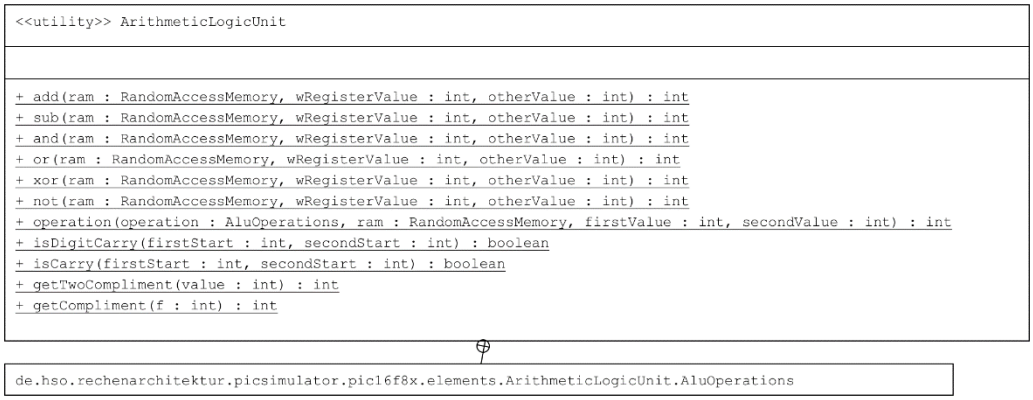
Abbildung 43 Codeausschnitt Einteilung der genauen Befehle

Die List mit den allen Befehlen und den zugehörigen Opcode steht im Anhang. Zuerst wird der Opcode zu der Gruppierung der Operationen ausgewertet und weitergeleitet. Zum Beispiel nehmen wir hier den Opcode 0b0110_1010_1111. Dieser würde, da die höchstwertigen Bits 01 sind, als Bit orientierte File Register Operation eingeteilt werden.

Danach werden in den jeweiligen Gruppen mit einer spezifisichen Bitmaske die genauen Befehle ermittelt. In unserem Beispiel sind die Bits an der Stelle 7-9, von dem minderwertigsten und ab Null gezählt, 11, was bedeutdet, dass unser Befehl ein BTFSS ist. Danach müssen nurnoch die restlichen Bits korrekt gruppiert ausgelesen werden. Bei dem BTFSS stehen die Bit 0-6 für f, also die Adresse welche der Befehlt testen soll. Unser Beispiel ist also ein BTFSS, welches das Bit 0b101, also 5 in der Adresse 0b10_1111 also 47 auf gesetzt überprüft und falls Wahr den nächsten Befehl durch einen NOP austauscht.

Arithmetische Logische Einheit (ALU)

Die Arbeitsklasse ALU wird mit der Operation, dem RAM und den nötigen Werten aufgerufen. Durch eine Switch wird entschieden was wie bearbeitet werden soll. Bei der Subtraktion wird der zweite Wert in durch das Zweierkomplement ersetzt und wie eine Addition normal fortgeführt. Da es einen Fehler in der Konstruktion des Microcontrollers gibt, welche auch hier eingebaut wurde, werden keine extra Vorkehrungen bei der Subtraktion getroffen. Die Flags werden vom ALU direkt in den RAM gesetzt.




```java
public static int operation(AluOperations operation, RandomAccessMemory ram, int firstValue, int secondValue) {
    int result = firstValue;
    switch (operation) {
        case SUB:
            //Zweierkompliment und dann fallthrough zu ADD
            secondValue = getTwoCompliment(secondValue);
        case ADD:
            //Reset affected flags
            ram.setCarryFlag(false);
            ram.setDigitCarryFlag(false);
            //
            result += secondValue;
            //CarryFlag
            ram.setCarryFlag(isCarry(firstValue, secondValue));
            //DigitCarryFlag
            ram.setDigitCarryFlag(isDigitCarry(firstValue, secondValue));
            break;
        case AND:
            result &= secondValue;
            break;
        case OR:
            result |= secondValue;
            break;
        case XOR:
            result ^= secondValue;
            break;
        case NOT:
            //never used
            break;
    }
    //Result auf 8Bit maskieren
    result &= 0xFF;
    //Zero Flag
    ram.setZeroFlag(result == 0);

    return result;
}
```

Fazit

Wir hatten bei der Umsetzung des Projektes viel spaß und haben sehr viel gelernt. Es war eine tolle Möglichkeit unsere Java-Kenntnisse zu vertiefen und mit Spaß und Souveränität etwas Eignes zu erstellen und dabei noch viel zu lernen. Wirklich große Schwierigkeiten hatten wir zu keiner Zeit, dennoch, zu Beginn des Projektes war uns nicht klar, wie wir dieses beginnen oder fertigstellen sollten. Es war alles sehr neu und zu komplex um direkt alles zu verstehen. Nachdem wir das Projekt in kleine Aufgaben eingeteilt hatten und einige von diesen dann auch schnell erledigen konnten, sah es machbarer aus. Das größte Problem war unser Unwissen, wie der PIC funktioniert. Somit konnte nicht wirklich exakt geplant werden. Was bekannt war wurde entwickelt und falls neue Erkenntnisse einkamen wurden oft alte Implementierungen verworfen oder mussten stark abgeändert werden. Es war ein ständiges lernen und anpassen in während der Laufzeit des Projektes. Java war keine Erschwernis für uns, nur hatten wir beide noch nie ein GUI in Java erstellt und mit Hilfe von Swing und dem „Intellij-WindowBuilder“ haben wir es zwar geschafft ein funktionsfähiges GUI zu bauen, aber haben wir dort die „best practice“ und gedachte Anwendung nicht eingehalten. Dies hat zu Beginn aber den Fortschritt sehr zurückgehalten. Zuerst wollten wir den Simulator 1:1 wie das Original realisieren. Dies stellte sich aber als ineffizient dar. Deswegen fingen wir frisch an. Dort wurden dann viele Funktionen des PICs abstrahiert um diese in Java um einiges schneller realisieren zu können. Das Fehlen einer genaueren Planung war ein Segen und Fluch zugleich. Eine Grobplanung welche festlegte, wie im Allgemeinen alles Funktionieren sollte, was für Klassen es gab und wie die GUI aussehen sollte hatten wir uns erstellt. Genauere Spezifikationen über Methoden und Attribute hatten wir nicht abgemacht. Dadurch konnten wir bei neuen Erkenntnissen oder Ideen flexibel alles anpassen. Die generelle Koordination war aber sehr erschwert, was bei einem kleinen Team nicht so schlimm wie bei Größeren ist. Nachdem die Grundfunktionen des Simulators implementiert waren, lief der restliche Entwicklungsprozess relativ schnell und leicht ab. Bei dem Verständnis des Timers und WatchDog hatten wir zu Beginn Schwierigkeiten, welche aber durch exaktes Recherchieren in dem Datenblattes des Herstellers, gelöst wurde. Vieles wurde später dennoch umgeschrieben oder gelöscht, da durch neue Ideen alte Funktionen obsolet wurden. Einiges konnte aber nicht angepasst werden, da es schon zu tief in der Implementation verankert war und der Aufwand alles anzupassen nicht Wert war. Hier zum Beispiel kommt der RAM, welcher als 128 langes 2d-Array verwirklicht wurde, um die Banken einfacher zu erkennen und zu simulieren. An vielen Stellen mussten aber die Zwei Dimensionen des Arrays hintereinander geknüpft werden um korrektes Verhalten zu erlangen.

Wenn wir das Projekt erneut realisieren würden, würden wir alles zu Beginn genauer Planen und festlegen. Da wir dann auch schon die korrekte Funktion des PICs kennen, wäre dies auch ohne Probleme möglich.

Anhang

Befehlsliste

[3]

TABLE 9-2 PIC16FXX INSTRUCTION SET

Mnemonic, Operands		Description	Cycles	14-Bit Opcode MSb			LSb	Status Affected	Notes
BYTE-ORIENTED FILE REGISTER OPERATIONS									
ADDWF	**f, d**	Add W and f	1	00	0111	dfff	ffff	C,DC,Z	1,2
ANDWF	**f, d**	AND W with f	1	00	0101	dfff	ffff	Z	1,2
CLRF	**f**	Clear f	1	00	0001	lfff	ffff	Z	2
CLRW	**-**	Clear W	1	00	0001	0xxx	xxxx	Z	
COMF	**f, d**	Complement f	1	00	1001	dfff	ffff	Z	1,2
DECF	**f, d**	Decrement f	1	00	0011	dfff	ffff	Z	1,2
DECFSZ	**f, d**	Decrement f, Skip if 0	1(2)	00	1011	dfff	ffff		1,2,3
INCF	**f, d**	Increment f	1	00	1010	dfff	ffff	Z	1,2
INCFSZ	**f, d**	Increment f, Skip if 0	1(2)	00	1111	dfff	ffff		1,2,3
IORWF	**f, d**	Inclusive OR W with f	1	00	0100	dfff	ffff	Z	1,2
MOVF	**f, d**	Move f	1	00	1000	dfff	ffff	Z	1,2
MOVWF	**f**	Move W to f	1	00	0000	lfff	ffff		
NOP	**-**	No Operation	1	00	0000	0xx0	0000		
RLF	**f, d**	Rotate Left f through Carry	1	00	1101	dfff	ffff	C	1,2
RRF	**f, d**	Rotate Right f through Carry	1	00	1100	dfff	ffff	C	1,2
SUBWF	**f, d**	Subtract W from f	1	00	0010	dfff	ffff	C,DC,Z	1,2
SWAPF	**f, d**	Swap nibbles in f	1	00	1110	dfff	ffff		1,2
XORWF	**f, d**	Exclusive OR W with f	1	00	0110	dfff	ffff	Z	1,2
BIT-ORIENTED FILE REGISTER OPERATIONS									
BCF	**f, b**	Bit Clear f	1	01	00bb	bfff	ffff		1,2
BSF	**f, b**	Bit Set f	1	01	01bb	bfff	ffff		1,2
BTFSC	**f, b**	Bit Test f, Skip if Clear	1 (2)	01	10bb	bfff	ffff		3
BTFSS	**f, b**	Bit Test f, Skip if Set	1 (2)	01	11bb	bfff	ffff		3
LITERAL AND CONTROL OPERATIONS									
ADDLW	**k**	Add literal and W	1	11	111x	kkkk	kkkk	C,DC,Z	
ANDLW	**k**	AND literal with W	1	11	1001	kkkk	kkkk	Z	
CALL	**k**	Call subroutine	2	10	0kkk	kkkk	kkkk		
CLRWDT	**-**	Clear Watchdog Timer	1	00	0000	0110	0100	$\overline{TO}$,$\overline{PD}$	
GOTO	**k**	Go to address	2	10	1kkk	kkkk	kkkk		
IORLW	**k**	Inclusive OR literal with W	1	11	1000	kkkk	kkkk	Z	
MOVLW	**k**	Move literal to W	1	11	00xx	kkkk	kkkk		
RETFIE	**-**	Return from interrupt	2	00	0000	0000	1001		
RETLW	**k**	Return with literal in W	2	11	01xx	kkkk	kkkk		
RETURN	**-**	Return from Subroutine	2	00	0000	0000	1000		
SLEEP	**-**	Go into standby mode	1	00	0000	0110	0011	$\overline{TO}$,$\overline{PD}$	
SUBLW	**k**	Subtract W from literal	1	11	110x	kkkk	kkkk	C,DC,Z	
XORLW	**k**	Exclusive OR literal with W	1	11	1010	kkkk	kkkk	Z	

Note 1: When an I/O register is modified as a function of itself (e.g., `MOVF PORTB, 1`), the value used will be that value present on the pins themselves. For example, if the data latch is '1' for a pin configured as input and is driven low by an external device, the data will be written back with a '0'.

2: If this instruction is executed on the TMR0 register (and, where applicable, d = 1), the prescaler will be cleared if assigned to the Timer0 Module.

3: If Program Counter (PC) is modified or a conditional test is true, the instruction requires two cycles. The second cycle is executed as a NOP.

[3] Datenblatt PIC16F8X von Microchip

SFR Bits

Status Register

[4]

FIGURE 4-1: STATUS REGISTER (ADDRESS 03h, 83h)

R/W-0	R/W-0	R/W-0	R-1	R-1	R/W-x	R/W-x	R/W-x
IRP	RP1	RP0	$\overline{TO}$	$\overline{PD}$	Z	DC	C
bit7							bit0

R = Readable bit
W = Writable bit
U = Unimplemented bit, read as '0'
- n = Value at POR reset

bit 7: **IRP**: Register Bank Select bit (used for indirect addressing)
0 = Bank 0, 1 (00h - FFh)
1 = Bank 2, 3 (100h - 1FFh)
The IRP bit is not used by the PIC16F8X. IRP should be maintained clear.

bit 6-5: **RP1:RP0**: Register Bank Select bits (used for direct addressing)
00 = Bank 0 (00h - 7Fh)
01 = Bank 1 (80h - FFh)
10 = Bank 2 (100h - 17Fh)
11 = Bank 3 (180h - 1FFh)
Each bank is 128 bytes. Only bit RP0 is used by the PIC16F8X. RP1 should be maintained clear.

bit 4: **$\overline{TO}$**: Time-out bit
1 = After power-up, `CLRWDT` instruction, or `SLEEP` instruction
0 = A WDT time-out occurred

bit 3: **$\overline{PD}$**: Power-down bit
1 = After power-up or by the `CLRWDT` instruction
0 = By execution of the `SLEEP` instruction

bit 2: **Z**: Zero bit
1 = The result of an arithmetic or logic operation is zero
0 = The result of an arithmetic or logic operation is not zero

bit 1: **DC**: Digit carry/$\overline{borrow}$ bit (for `ADDWF` and `ADDLW` instructions) (For $\overline{borrow}$ the polarity is reversed)
1 = A carry-out from the 4th low order bit of the result occurred
0 = No carry-out from the 4th low order bit of the result

bit 0: **C**: Carry/$\overline{borrow}$ bit (for `ADDWF` and `ADDLW` instructions)
1 = A carry-out from the most significant bit of the result occurred
0 = No carry-out from the most significant bit of the result occurred
Note: For $\overline{borrow}$ the polarity is reversed. A subtraction is executed by adding the two's complement of the second operand. For rotate (`RRF`, `RLF`) instructions, this bit is loaded with either the high or low order bit of the source register.

[4] Datenblatt PIC16F8X von Microchip

Option Register

[5]

FIGURE 4-1: OPTION_REG REGISTER (ADDRESS 81h)

R/W-1	R/W-1	R/W-1	R/W-1	R/W-1	R/W-1	R/W-1	R/W-1
$\overline{\text{RBPU}}$	INTEDG	T0CS	T0SE	PSA	PS2	PS1	PS0
bit7							bit0

R = Readable bit
W = Writable bit
U = Unimplemented bit, read as '0'
- n = Value at POR reset

bit 7: **$\overline{\text{RBPU}}$**: PORTB Pull-up Enable bit
1 = PORTB pull-ups are disabled
0 = PORTB pull-ups are enabled (by individual port latch values)

bit 6: **INTEDG**: Interrupt Edge Select bit
1 = Interrupt on rising edge of RB0/INT pin
0 = Interrupt on falling edge of RB0/INT pin

bit 5: **T0CS**: TMR0 Clock Source Select bit
1 = Transition on RA4/T0CKI pin
0 = Internal instruction cycle clock (CLKOUT)

bit 4: **T0SE**: TMR0 Source Edge Select bit
1 = Increment on high-to-low transition on RA4/T0CKI pin
0 = Increment on low-to-high transition on RA4/T0CKI pin

bit 3: **PSA**: Prescaler Assignment bit
1 = Prescaler assigned to the WDT
0 = Prescaler assigned to TMR0

bit 2-0: **PS2:PS0**: Prescaler Rate Select bits

Bit Value	TMR0 Rate	WDT Rate
000	1 : 2	1 : 1
001	1 : 4	1 : 2
010	1 : 8	1 : 4
011	1 : 16	1 : 8
100	1 : 32	1 : 16
101	1 : 64	1 : 32
110	1 : 128	1 : 64
111	1 : 256	1 : 128

[5] Datenblatt PIC16F8X von Microchip

[6]INTCON Register

FIGURE 4-1: INTCON REGISTER (ADDRESS 0Bh, 8Bh)

R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-x
GIE	EEIE	T0IE	INTE	RBIE	T0IF	INTF	RBIF
bit7							bit0

R = Readable bit
W = Writable bit
U = Unimplemented bit, read as '0'
- n = Value at POR reset

bit 7: **GIE:** Global Interrupt Enable bit
1 = Enables all un-masked interrupts
0 = Disables all interrupts

Note: For the operation of the interrupt structure, please refer to Section 8.5.

bit 6: **EEIE**: EE Write Complete Interrupt Enable bit
1 = Enables the EE write complete interrupt
0 = Disables the EE write complete interrupt

bit 5: **T0IE**: TMR0 Overflow Interrupt Enable bit
1 = Enables the TMR0 interrupt
0 = Disables the TMR0 interrupt

bit 4: **INTE**: RB0/INT Interrupt Enable bit
1 = Enables the RB0/INT interrupt
0 = Disables the RB0/INT interrupt

bit 3: **RBIE**: RB Port Change Interrupt Enable bit
1 = Enables the RB port change interrupt
0 = Disables the RB port change interrupt

bit 2: **T0IF**: TMR0 overflow interrupt flag bit
1 = TMR0 has overflowed (must be cleared in software)
0 = TMR0 did not overflow

bit 1: **INTF**: RB0/INT Interrupt Flag bit
1 = The RB0/INT interrupt occurred
0 = The RB0/INT interrupt did not occur

bit 0: **RBIF**: RB Port Change Interrupt Flag bit
1 = When at least one of the RB7:RB4 pins changed state (must be cleared in software)
0 = None of the RB7:RB4 pins have changed state

[6] Datenblatt PIC16F8X von Microchip

Fileregister

[7]

TABLE 4-1 REGISTER FILE SUMMARY

Address	Name	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0	Value on Power-on Reset	Value on all other resets (Note3)
Bank 0											
00h	INDF	Uses contents of FSR to address data memory (not a physical register)								---- ----	---- ----
01h	TMR0	8-bit real-time clock/counter								xxxx xxxx	uuuu uuuu
02h	PCL	Low order 8 bits of the Program Counter (PC)								0000 0000	0000 0000
03h	STATUS(2)	IRP	RP1	RP0	$\overline{TO}$	$\overline{PD}$	Z	DC	C	0001 1xxx	000q quuu
04h	FSR	Indirect data memory address pointer 0								xxxx xxxx	uuuu uuuu
05h	PORTA	—	—	—	RA4/T0CKI	RA3	RA2	RA1	RA0	---x xxxx	---u uuuu
06h	PORTB	RB7	RB6	RB5	RB4	RB3	RB2	RB1	RB0/INT	xxxx xxxx	uuuu uuuu
07h		Unimplemented location, read as '0'								---- ----	---- ----
08h	EEDATA	EEPROM data register								xxxx xxxx	uuuu uuuu
09h	EEADR	EEPROM address register								xxxx xxxx	uuuu uuuu
0Ah	PCLATH	—	—	—	Write buffer for upper 5 bits of the PC (1)					---0 0000	---0 0000
0Bh	INTCON	GIE	EEIE	T0IE	INTE	RBIE	T0IF	INTF	RBIF	0000 000x	0000 000u
Bank 1											
80h	INDF	Uses contents of FSR to address data memory (not a physical register)								---- ----	---- ----
81h	OPTION_REG	$\overline{RBPU}$	INTEDG	T0CS	T0SE	PSA	PS2	PS1	PS0	1111 1111	1111 1111
82h	PCL	Low order 8 bits of Program Counter (PC)								0000 0000	0000 0000
83h	STATUS(2)	IRP	RP1	RP0	$\overline{TO}$	$\overline{PD}$	Z	DC	C	0001 1xxx	000q quuu
84h	FSR	Indirect data memory address pointer 0								xxxx xxxx	uuuu uuuu
85h	TRISA	—	—	—	PORTA data direction register					---1 1111	---1 1111
86h	TRISB	PORTB data direction register								1111 1111	1111 1111
87h		Unimplemented location, read as '0'								---- ----	---- ----
88h	EECON1	—	—	—	EEIF	WRERR	WREN	WR	RD	---0 x000	---0 q000
89h	EECON2	EEPROM control register 2 (not a physical register)								---- ----	---- ----
~~0Ah~~ 8Ah	PCLATH	—	—	—	Write buffer for upper 5 bits of the PC (1)					---0 0000	---0 0000
~~0Bh~~ 8Bh	INTCON	GIE	EEIE	T0IE	INTE	RBIE	T0IF	INTF	RBIF	0000 000x	0000 000u

Legend: x = unknown, u = unchanged, - = unimplemented read as '0', q = value depends on condition.

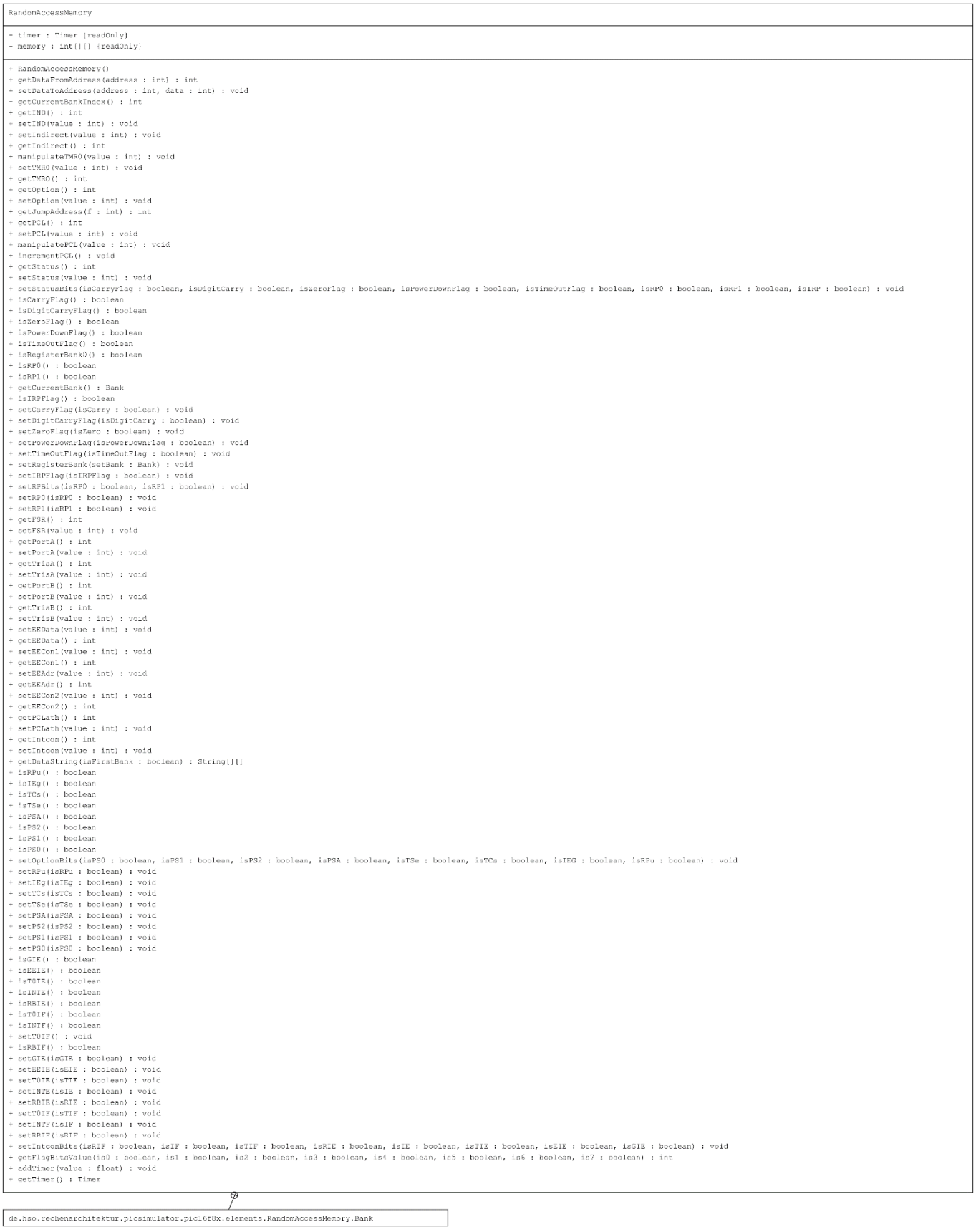
Note 1: The upper byte of the program counter is not directly accessible. PCLATH is a slave register for PC<12:8>. The contents of PCLATH can be transferred to the upper byte of the program counter, but the contents of PC<12:8> is never transferred to PCLATH.

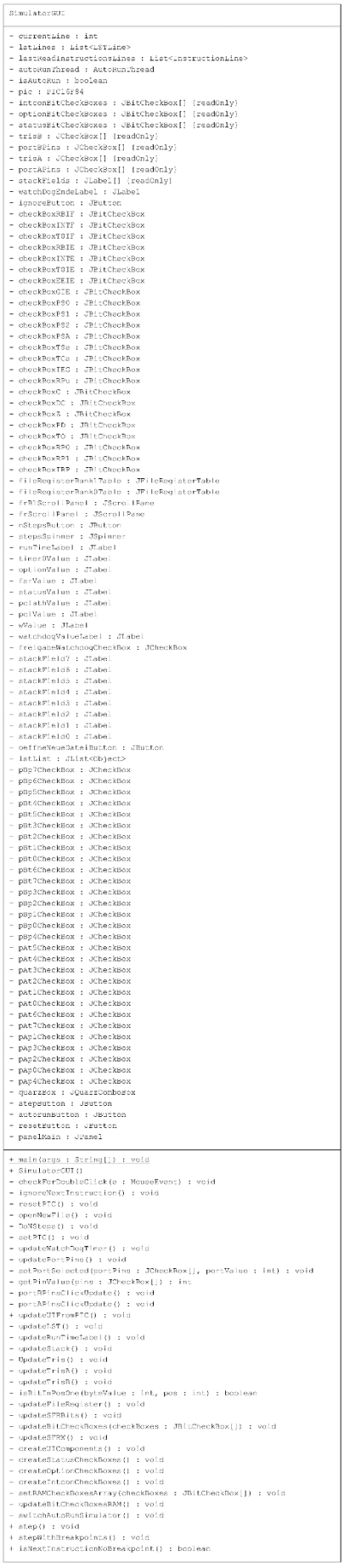
2: The $\overline{TO}$ and $\overline{PD}$ status bits in the STATUS register are not affected by a $\overline{MCLR}$ reset.

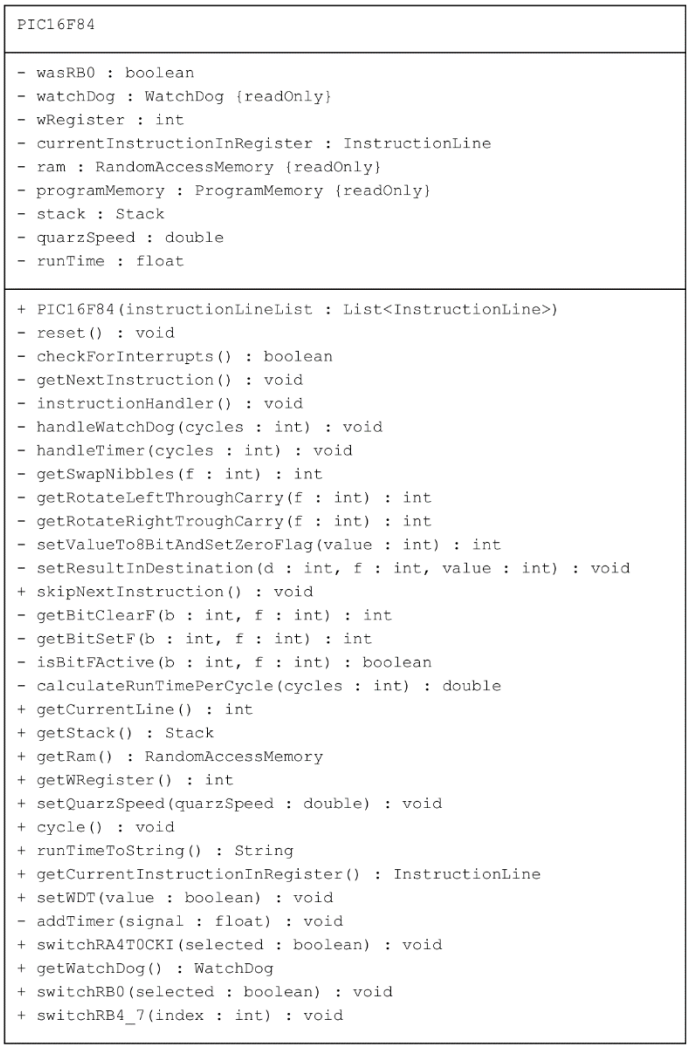
3: Other (non power-up) resets include: external reset through $\overline{MCLR}$ and the Watchdog Timer Reset.

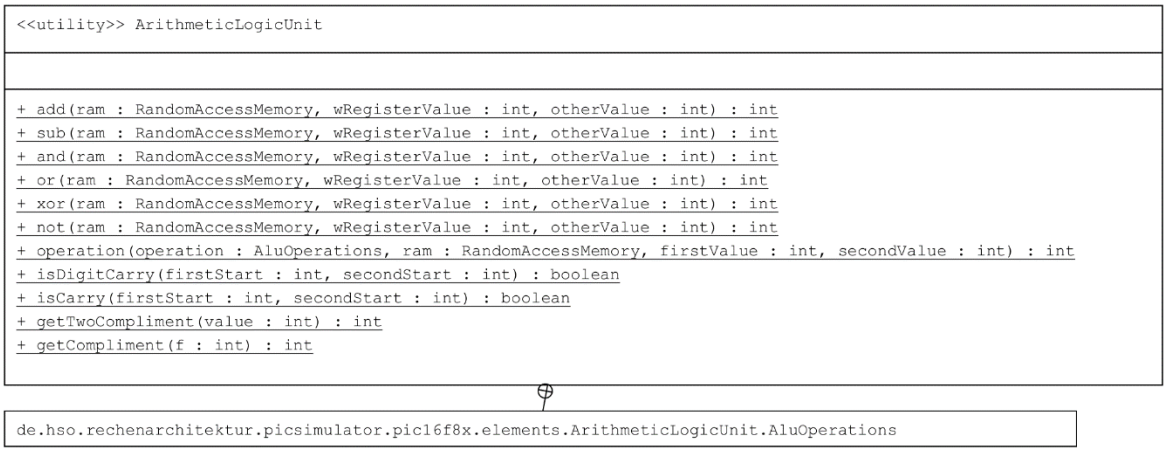
[7] Datenblatt PIC16F8X von Microchip

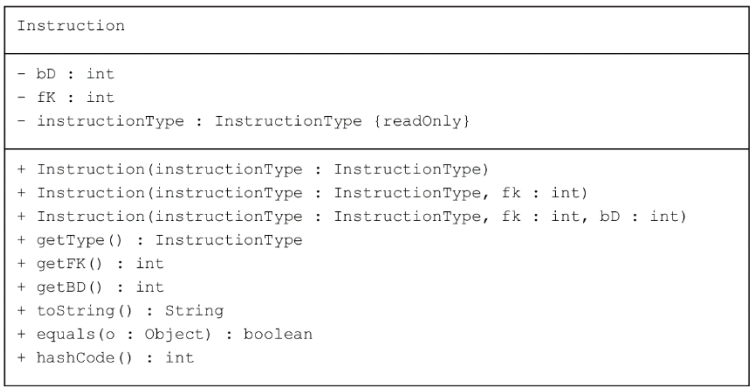
Klassendiagramme

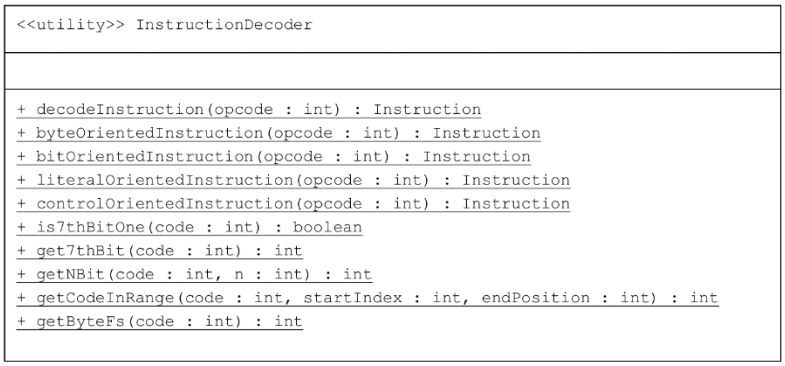


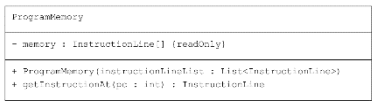

SimulatorGUI
- currentLine : int
- lstLines : List<LstLine>
- lastReadInstructionsLines : List<InstructionLine>
- autoRunThread : AutoRunThread
- isAutoRun : boolean
- pic : PIC16F84
- intconBitCheckBoxes : JBitCheckBox[] {readOnly}
- optionBitCheckBoxes : JBitCheckBox[] {readOnly}
- statusBitCheckBoxes : JBitCheckBox[] {readOnly}
- trisB : JCheckBox[] {readOnly}
- portBPins : JCheckBox[] {readOnly}
- trisA : JCheckBox[] {readOnly}
- portAPins : JCheckBox[] {readOnly}
- stackFields : JLabel[] {readOnly}
- watchDogEndeLabel : JLabel
- ignoreButton : JButton
- checkBoxRBIF : JBitCheckBox
- checkBoxINTF : JBitCheckBox
- checkBoxT0IF : JBitCheckBox
- checkBoxRBIE : JBitCheckBox
- checkBoxINTE : JBitCheckBox
- checkBoxT0IE : JBitCheckBox
- checkBoxEEIE : JBitCheckBox
- checkBoxGIE : JBitCheckBox
- checkBoxPS0 : JBitCheckBox
- checkBoxPS1 : JBitCheckBox
- checkBoxPS2 : JBitCheckBox
- checkBoxPSA : JBitCheckBox
- checkBoxT0e : JBitCheckBox
- checkBoxTCs : JBitCheckBox
- checkBoxIRP : JBitCheckBox
- checkBoxRPu : JBitCheckBox
- checkBoxC : JBitCheckBox
- checkBoxDC : JBitCheckBox
- checkBoxZ : JBitCheckBox
- checkBoxPD : JBitCheckBox
- checkBoxTO : JBitCheckBox
- checkBoxRP0 : JBitCheckBox
- checkBoxRP1 : JBitCheckBox
- checkBoxTRP : JBitCheckBox
- fileRegisterBank1Table : JFileRegisterTable
- fileRegisterBank0Table : JFileRegisterTable
- frBtScrollPanel : JScrollPane
- frScrollPanel : JScrollPane
- nStepsButton : JButton
- stepsSpinner : JSpinner
- runTimeLabel : JLabel
- timer0Value : JLabel
- optionValue : JLabel
- fsrValue : JLabel
- statusValue : JLabel
- pclathValue : JLabel
- pclValue : JLabel
- wValue : JLabel
- watchdogValueLabel : JLabel
- freigabeWatchdogCheckBox : JCheckBox
- stackField7 : JLabel
- stackField6 : JLabel
- stackField5 : JLabel
- stackField4 : JLabel
- stackField3 : JLabel
- stackField2 : JLabel
- stackField1 : JLabel
- stackField0 : JLabel
- oeffneNeueDateiButton : JButton
- lstList : JList<Object>
- pBp7CheckBox : JCheckBox
- pBp6CheckBox : JCheckBox
- pBp5CheckBox : JCheckBox
- pBt6CheckBox : JCheckBox
- pBt5CheckBox : JCheckBox
- pBt3CheckBox : JCheckBox
- pBt2CheckBox : JCheckBox
- pBt1CheckBox : JCheckBox
- pBt0CheckBox : JCheckBox
- pBt6CheckBox : JCheckBox
- pBt7CheckBox : JCheckBox
- pBp3CheckBox : JCheckBox
- pBp2CheckBox : JCheckBox
- pBp1CheckBox : JCheckBox
- pBp0CheckBox : JCheckBox
- pBp4CheckBox : JCheckBox
- pAt5CheckBox : JCheckBox
- pAt4CheckBox : JCheckBox
- pAt3CheckBox : JCheckBox
- pAt2CheckBox : JCheckBox
- pAt1CheckBox : JCheckBox
- pAt0CheckBox : JCheckBox
- pAt6CheckBox : JCheckBox
- pAt7CheckBox : JCheckBox
- pAp1CheckBox : JCheckBox
- pAp3CheckBox : JCheckBox
- pAp2CheckBox : JCheckBox
- pAp0CheckBox : JCheckBox
- pAp4CheckBox : JCheckBox
- quarzBox : qQuarzComboBox
- stepButton : JButton
- autorunButton : JButton
- resetButton : JButton
- panelMain : JPanel
+ main(args : String[]) : void
+ SimulatorGUI()
- checkForDoubleClick(e : MouseEvent) : void
- ignoreNextInstruction() : void
- resetPIC() : void
- openNewFile() : void
- loNSteps() : void
- setPIC() : void
- updateWatchDogTimer() : void
- updatePortPins() : void
- setPortSelected(portPins : JCheckBox[], portValue : int) : void
- getPinValue(pins : JCheckBox[]) : int
- portBPinsClickUpdate() : void
- portAPinsClickUpdate() : void
+ updateGUIFromPIC() : void
- updateSFR() : void
- updateRunTimeLabel() : void
- updateStack() : void
- UpdateTris() : void
- updateTrisA() : void
- updateTrisB() : void
- isBitInPosOne(byteValue : int, pos : int) : boolean
- updateFileRegister() : void
- updateSTRBits() : void
- updateBitCheckBoxes(checkBoxes : JBitCheckBox[]) : void
- updateSTRX() : void
- createUIComponents() : void
- createStatusCheckBoxes() : void
- createOptionCheckBoxes() : void
- createIntconCheckBoxes() : void
- setRAMCheckBoxesArray(checkBoxes : JBitCheckBox[]) : void
- updateBitCheckBoxesRAM() : void
- switchAutoRunSimulator() : void
+ step() : void
+ stepWithBreakpoints() : void
+ isNextInstructionNoBreakpoint() : boolean

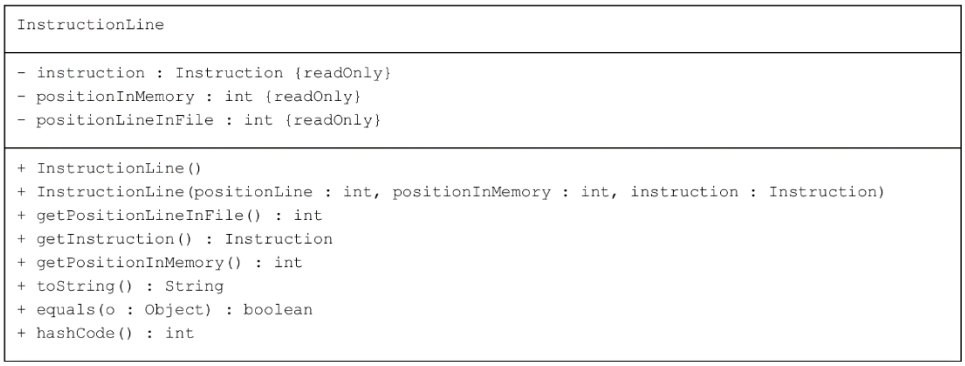

PIC16F84
- wasRB0 : boolean
- watchDog : WatchDog {readOnly}
- wRegister : int
- currentInstructionInRegister : InstructionLine
- ram : RandomAccessMemory {readOnly}
- programMemory : ProgramMemory {readOnly}
- stack : Stack
- quarzSpeed : double
- runTime : float
+ PIC16F84(instructionLineList : List<InstructionLine>)
- reset() : void
- checkForInterrupts() : boolean
- getNextInstruction() : void
- instructionHandler() : void
- handleWatchDog(cycles : int) : void
- handleTimer(cycles : int) : void
- getSwapNibbles(f : int) : int
- getRotateLeftThroughCarry(f : int) : int
- getRotateRightTroughCarry(f : int) : int
- setValueTo8BitAndSetZeroFlag(value : int) : int
- setResultInDestination(d : int, f : int, value : int) : void
+ skipNextInstruction() : void
- getBitClearF(b : int, f : int) : int
- getBitSetF(b : int, f : int) : int
- isBitFActive(b : int, f : int) : boolean
- calculateRunTimePerCycle(cycles : int) : double
+ getCurrentLine() : int
+ getStack() : Stack
+ getRam() : RandomAccessMemory
+ getWRegister() : int
+ setQuarzSpeed(quarzSpeed : double) : void
+ cycle() : void
+ runTimeToString() : String
+ getCurrentInstructionInRegister() : InstructionLine
+ setWDT(value : boolean) : void
- addTimer(signal : float) : void
+ switchRA4T0CKI(selected : boolean) : void
+ getWatchDog() : WatchDog
+ switchRB0(selected : boolean) : void
+ switchRB4_7(index : int) : void

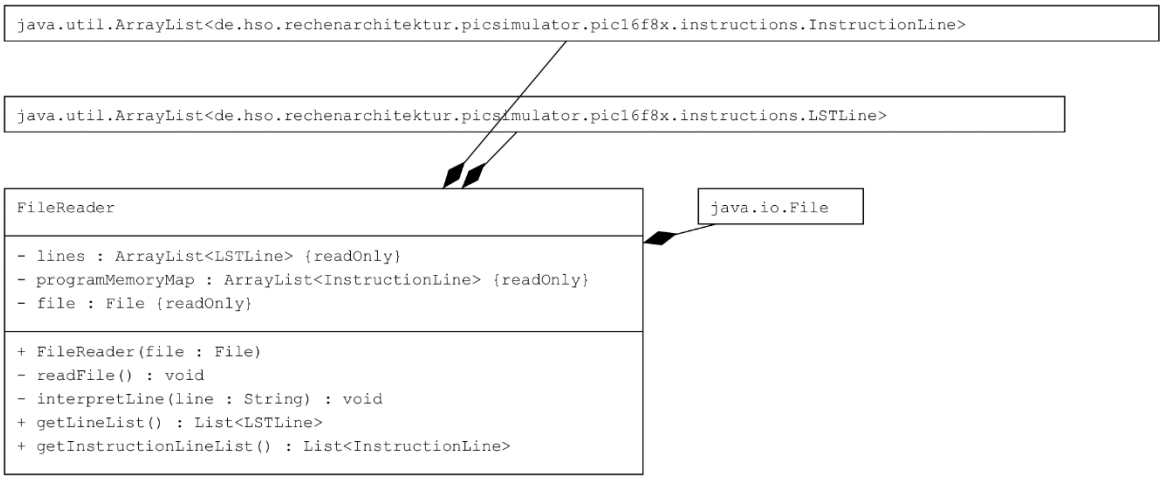

<<utility>> ArithmeticLogicUnit
+ add(ram : RandomAccessMemory, wRegisterValue : int, otherValue : int) : int
+ sub(ram : RandomAccessMemory, wRegisterValue : int, otherValue : int) : int
+ and(ram : RandomAccessMemory, wRegisterValue : int, otherValue : int) : int
+ or(ram : RandomAccessMemory, wRegisterValue : int, otherValue : int) : int
+ xor(ram : RandomAccessMemory, wRegisterValue : int, otherValue : int) : int
+ not(ram : RandomAccessMemory, wRegisterValue : int, otherValue : int) : int
+ operation(operation : AluOperations, ram : RandomAccessMemory, firstValue : int, secondValue : int) : int
+ isDigitCarry(firstStart : int, secondStart : int) : boolean
+ isCarry(firstStart : int, secondStart : int) : boolean
+ getTwoCompliment(value : int) : int
+ getCompliment(f : int) : int
de.hso.rechenarchitektur.picsimulator.pic16f8x.elements.ArithmeticLogicUnit.AluOperations

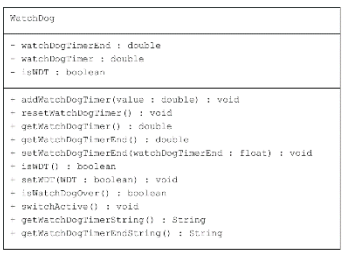

Instruction
- bD : int
- fK : int
- instructionType : InstructionType {readOnly}
+ Instruction(instructionType : InstructionType)
+ Instruction(instructionType : InstructionType, fk : int)
+ Instruction(instructionType : InstructionType, fk : int, bD : int)
+ getType() : InstructionType
+ getFK() : int
+ getBD() : int
+ toString() : String
+ equals(o : Object) : boolean
+ hashCode() : int

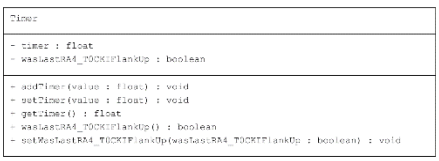

<<utility>> InstructionDecoder
+ decodeInstruction(opcode : int) : Instruction
+ byteOrientedInstruction(opcode : int) : Instruction
+ bitOrientedInstruction(opcode : int) : Instruction
+ literalOrientedInstruction(opcode : int) : Instruction
+ controlOrientedInstruction(opcode : int) : Instruction
+ is7thBitOne(code : int) : boolean
+ get7thBit(code : int) : int
+ getNBit(code : int, n : int) : int
+ getCodeInRange(code : int, startIndex : int, endPosition : int) : int
+ getByteFs(code : int) : int

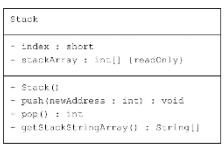

ProgramMemory
- memory : InstructionLine[] {readOnly}
+ ProgramMemory(instructionLineList : List<InstructionLine>)
+ getInstructionAt(pc : int) : InstructionLine


InstructionLine
- instruction : Instruction {readOnly}
- positionInMemory : int {readOnly}
- positionLineInFile : int {readOnly}
+ InstructionLine()
+ InstructionLine(positionLine : int, positionInMemory : int, instruction : Instruction)
+ getPositionLineInFile() : int
+ getInstruction() : Instruction
+ getPositionInMemory() : int
+ toString() : String
+ equals(o : Object) : boolean
+ hashCode() : int


java.util.ArrayList<de.hso.rechenarchitektur.picsimulator.pic16f8x.instructions.InstructionLine>
java.util.ArrayList<de.hso.rechenarchitektur.picsimulator.pic16f8x.instructions.LSTLine>
FileReader
- lines : ArrayList<LSTLine> {readOnly}
- programMemoryMap : ArrayList<InstructionLine> {readOnly}
- file : File {readOnly}
+ FileReader(file : File)
- readFile() : void
- interpretLine(line : String) : void
+ getLineList() : List<LSTLine>
+ getInstructionLineList() : List<InstructionLine>
java.io.File


WatchDog
- watchDogTimerEnd : double
- watchDogTimer : double
- isWDT : boolean
- addWatchDogTimer(value : double) : void
- resetWatchDogTimer() : void
- getWatchDogTimer() : double
- getWatchDogTimerEnd() : double
- setWatchDogTimerEnd(watchDogTimerEnd : float) : void
- isWDT() : boolean
- setWDT(WDT : boolean) : void
- isWatchDogOver() : boolean
- switchActive() : void
- getWatchDogTimerString() : String
- getWatchDogTimerEndString() : String


Timer
- timer : float
- wasLastRA4_TOCKIFlankUp : boolean
- addTimer(value : float) : void
- setTimer(value : float) : void
- getTimer() : float
- wasLastRA4_TOCKIFlankUp() : boolean
- setWasLastRA4_TOCKIFlankUp(wasLastRA4_TOCKIFlankUp : boolean) : void


Stack
- index : short
- stackArray : int[] {readOnly}
- Stack()
- push(newAddress : int) : void
+ pop() : int
- getStackStringArray() : String[]