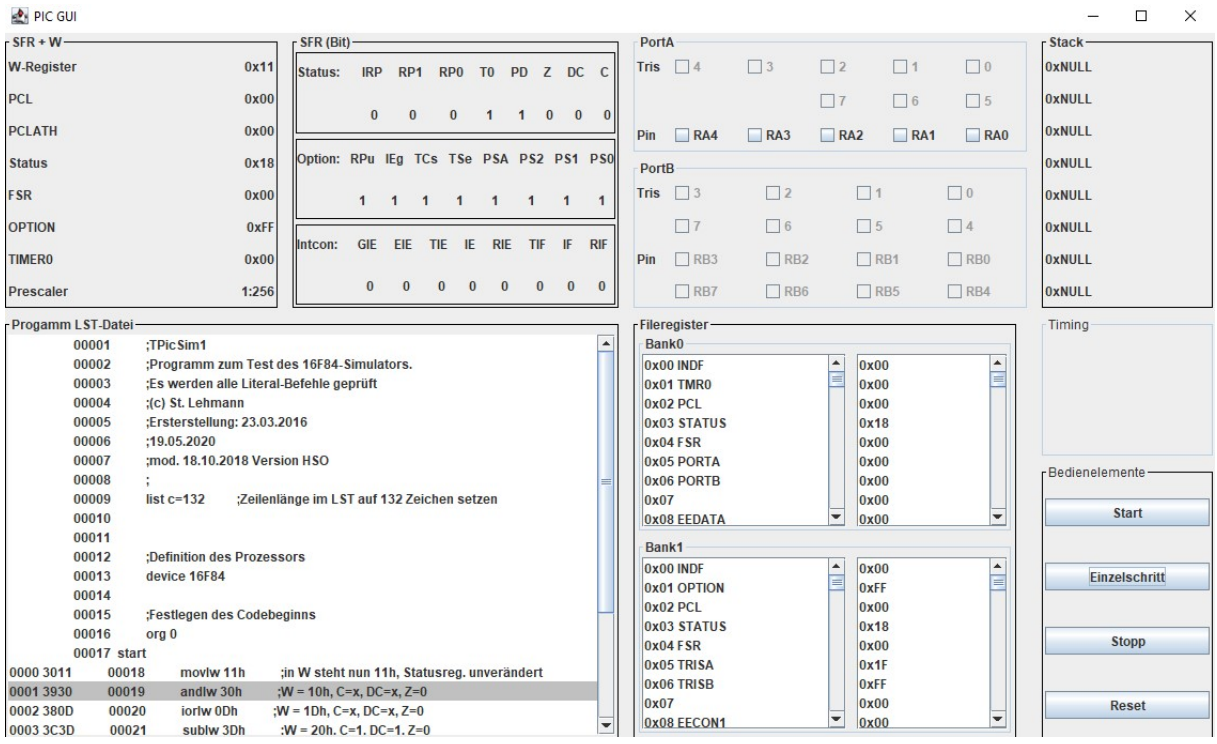
Simulator PIC16F84

Dokumentation



Rechnerarchitektur

Hochschule Offenburg

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

Sommersemester 2022

Nicklas Schwende, Necip Özcay

Allgemeine Einleitung Simulator

Ein Simulator ist grundsätzlich dafür da, um Erkenntnisse über ein reales System zu gewinnen. Hierfür wird dieses reale System nicht benötigt. Auch kann der Simulator kontrolliert abgeändert werden um Spezifische Simulationen abzubilden. Die Vorteile hierbei sind, dass hierbei genauestens beobachtet werden kann, was genau im Hintergrund des Simulators passiert und was nicht. Der Nachteil ist, dass ein Simulator immer ein Simulator bleibt und somit die Realität nicht perfekt wiedergeben kann.

Im Rahmen des Praktikums im Fach Rechnerarchitektur an der Hochschule Offenburg wurde der Mikrocontroller PIC16F84 in Java realisiert. Die Programmiersprache war frei wählbar, wir entschieden uns jedoch für Java, da wir mit dieser Sprache die meiste Erfahrung haben.

Es wurden nicht alle Funktionen des PIC16F84 simuliert, jedoch sind die Grundfunktionalitäten realisiert worden.

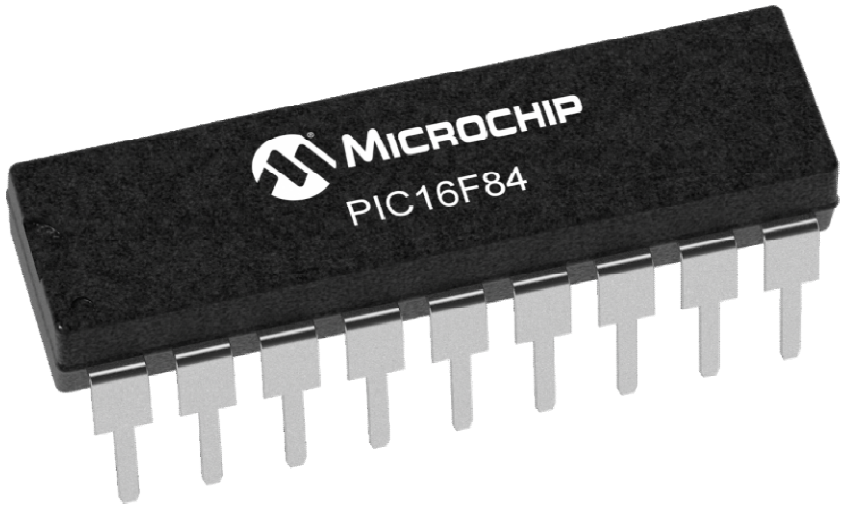


Abbildung 1 PIC16F84

Funktionen der Grafischen Benutzeroberfläche

Über die Benutzeroberfläche kann der genaue Ablauf des aktuell simulierten Files eingesehen werden.

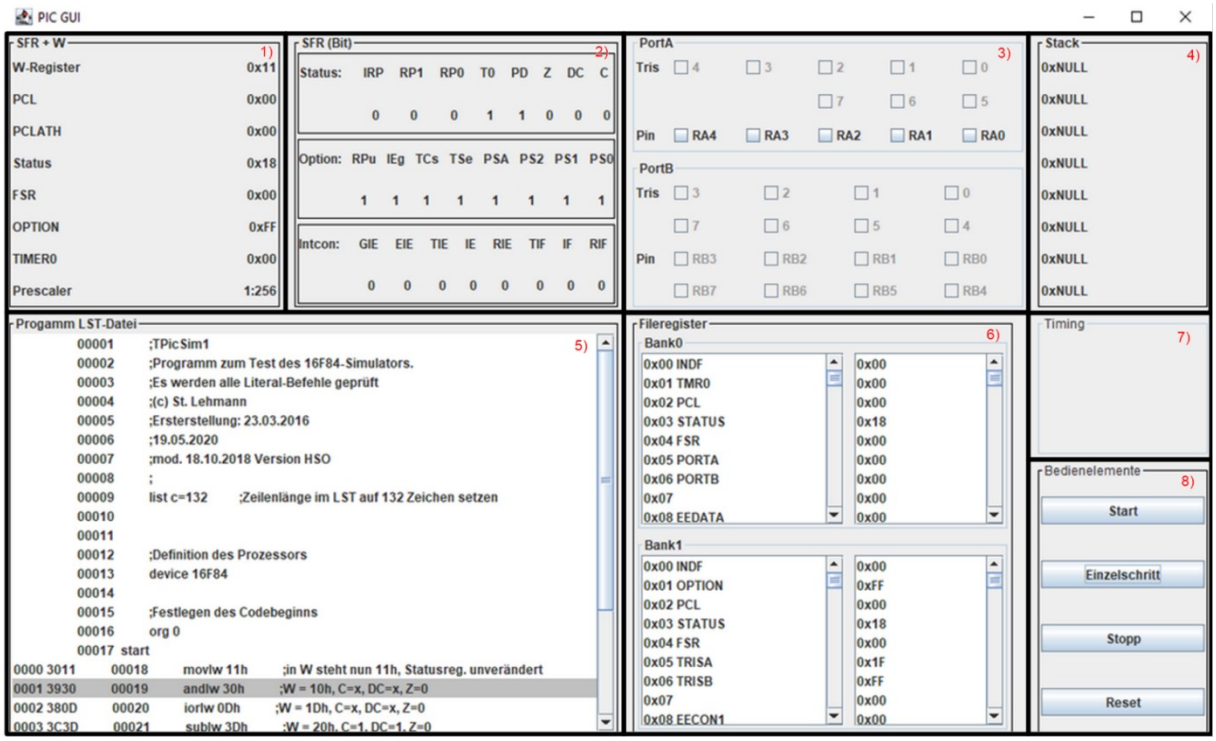


Abbildung 2 Komplettansicht Simulator

1) Wichtige SFR und W-Register Werte
2) SFR Bits
3) Port A und Port B
4) Stack
5) LST Programmvorschau
6) Fileregister Bank0 und Bank1
7) Timing (nicht implementiert)
8) Bedienelemente

Das Programm wird über die Main gestartet. Die Programmauswahl findet über die Eingabe des gewünschten Files statt. Nach dem ersten Klick auf Einzelschritt oder Start in 8) wird die GUI aktualisiert.

Wichtige SRF und W-Register Werte

Hier werden Relevante und oft benutzte Werte visualisiert.

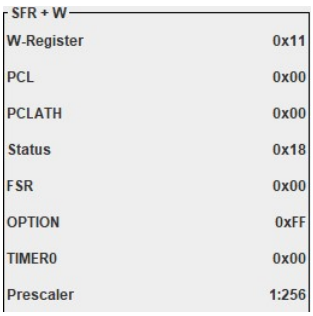


Abbildung 3 SFR + W

SFR Bits

Hier werden Bestimmte SFR als Bits angezeigt. Eine Manipulation ist nicht möglich.

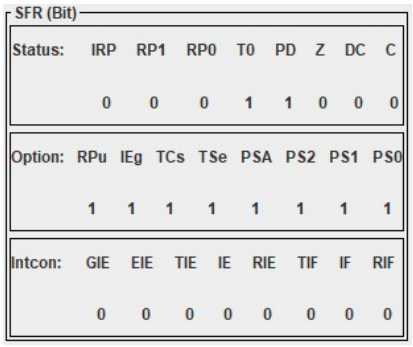


Abbildung 4 SFR Bits

Port A und Port B

Hier wird Port A und Port B als Checkboxen, sowie die dazugehörenden Tris Register abgebildet.

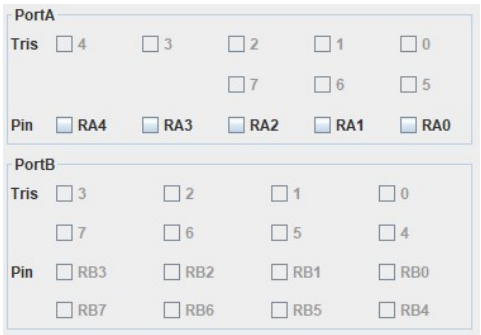


Abbildung 5 Port A und Port B

Stack

Der Stack ist ein Array aus acht Elementen, welches als Ringpuffer mit dem LIFO Prinzip arbeitet. Bei ablegen in den Stack ändern sich die Werte entsprechend.

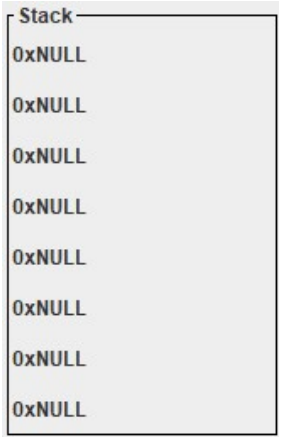


Abbildung 6 Stack

Programmvorschau

In der Programmvorschau wird die Geladene LST-Datei angezeigt. Die grau markierte Zeile ist die nächste auszuführende Befehlszeile. Die rote Zeile markiert eine als Breakpoint markierte Zeile. Diese kann durch einen einfachen Klick auf diese markiert werden. Hier wird der Simulator angehalten.

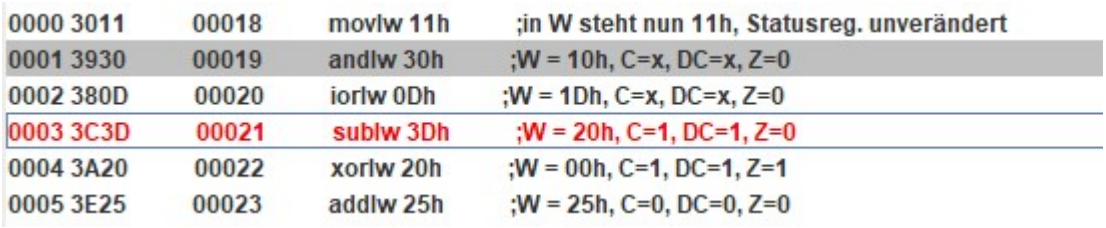


Abbildung 7 Programmvorschau

Fileregister Bank0 und Bank1

Hier werden die Fileregister des PIC16F84 dargestellt. Auf der Linken Hälfte sind jeweils die Adressen des Registers gegeben und auf der rechten Seite die Werte in den Registern. Oben ist die Bank0 und unten die Bank1 abgebildet.

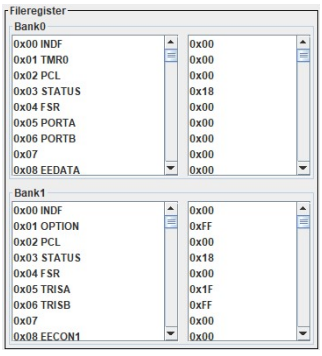


Abbildung 8 Fileregister

Bedienelemente

Hier kann der Simulator gesteuert werden. Start lässt den Simulator ohne Begrenzung laufen, Stopp lässt ihn an der aktuellen Stelle anhalten. Mit Einzelschritt kann man im Programm den nächsten Schritt ausführen.

Abbildung 9 Bedienelemente

Realisation

Gliederung

Das Projekt ist in drei Teile unterteilt. Benutzeroberfläche und Simulator und Main. Die Main startet den Simulator über den Aufruf von *Decoder.obj.decoder.decodeString(String src).* Die GUI beinhaltet alle GUI relevanten Objekte, wie Java Swing Komponenten und den Aufruf der Klasse zur Erstellung eines eigenen Threads, welche nun die GUI startet. Der Simulator läuft auf dem Haupthread des Programms.

Das hauptsächliche Programm ist in Decoder und AllObjects unterteil. AllObjects fungiert hier als ROM Speicher und beinhaltet alle zu erstellenden Objekte (ALU, Stack, Ram, MainFrame, Decoder, ProgrammMemory, Prescaler und Timer), welche durch das Singleton Pattern nur ein Mal erstellt werden können. Der Decoder benutzt regelmäßig die erstellten Objekte in AllObjects.

Programmablauf

Die Funktion decodeString(src) liest die Datei, welche über src als Pfad übergeben wird Zeile für Zeile ein und speichert die relevanten Zeilen in dem gewünschten Datentyp in einer ArrayList<>. Nachdem alle Zeilen erfolgreich eingelesen und Gespeichert wurden, wird das Programm Über nextStep() gesteuert. Hiermit wird die Funktion functionCalls(Integer pc) mit dem Übergabeparameter pc = ProgrammCounter aufgerufen, welcher die Befehle in der zu Beginn gespeicherten ArrayListe<> an der Stelle des aktuellen ProgrammCounters ausliest und decodiert. Dieser Wert wird über Bitmasken in Operationscode und Operationsvalue unterteilt. Über eine Switch Case wird der Operationcode verglichen und der passende Befehl wird ausgeführt.

Im Folgenden werden noch die Befehle BTFSC/BTFSS, CALL, MOVF, RRF, SUBWF, DECFSZ und XORLW in kurzen Programmsequenzen erläutert.

BTFSx

BTFSC (Bit Test, Skip if Clear) und BTFSS (Bit Test, Skip if Set) wurden beide ähnlich implementiert.

```
/** If bit 'b' in register 'f' is '1' then the next ...*/
 Knaeggel
public void btfsc(Integer f) {
    int addressInRam = obj.alu.and(f, 0b0111_1111);
    int valueOnAdress = obj.ram.getRamAt(addressInRam);
    int bitToCheck = obj.alu.getIntValFromBitToBit( startBit: 8,  endBit: 10, f);

    //indirect addr.
    if (addressInRam == 0) {...}

    if ((obj.ram.getNthBitOfValue(bitToCheck, valueOnAdress) == 0)) {
        obj.programMemory.skipNextInstruction();

    }
    //System.out.println("btfsc");
}

/** If bit 'b' in register 'f' is '0' then the next ...*/
 Knaeggel
public void btfss(Integer f) {
    int addressInRam = obj.alu.and(f, 0b0111_1111);
    int valueOnAdress = obj.ram.getRamAt(addressInRam);
    int bitToClear = obj.alu.getIntValFromBitToBit( startBit: 8,  endBit: 10, f);

    //indirecr addr.
    if (addressInRam == 0) {...}

    if (obj.ram.getNthBitOfValue(bitToClear, valueOnAdress) == 1) {
        obj.programMemory.skipNextInstruction();

    }
    //System.out.println("btfss");
}
```

Abbildung 10 Codeausschnitt BTFSC und BTFSS

Der Übergabeparameter f wird zunächst mit einer Bitmaske maskiert, um die entsprechende Adresse im Ram herauszufinden. Danach wird der wert, welcher an der Adresse im Ram steht ausgelesen. Als letztes wird die Zahl, welches Bit nun überprüft werden soll aus dem Übergabeparameter f ausgelesen mit getIntValFromBitToBit(8, 10, f). In der letzten if abfrage würd überprüft ob entsprechend bei BTFSS das Bit gesetzt ist, wenn ja dann überspringe die nächste Instruktion. Bei BTFSC wird auf ein nichtgesetztes Bit überprüft und falls dies zutrifft wird die nächste Instruktion entsprechend wieder Übersprungen.

CALL

Mit dem Call befehl kann an jede Stelle im Programmablauf gesprungen werden. Bei jedem Call wird der aktuelle Programmzähler für das Return im Stack abgelegt. Hierbei wird der ProgrammCounter auf den Wert i gesetzt und Bit 11 und 12 werden aus dem PCLATH drei und vier geholt.

```
public void call(Integer i) {
    int pcOfThisInstruction = Ram.programmCounter - 1;
    Timer.timerIncrementCount--;
    if (!obj.programMemory.checkCycle(pcOfThisInstruction)) {
        obj.stack.pushOnStack(Ram.programmCounter);
        Ram.programmCounter = i;
        Ram.programmCounter = obj.ram.setBit( bit: 11, Ram.programmCounter, obj.ram.getSpecificPCLATHBit( n: 3));
        Ram.programmCounter = obj.ram.setBit( bit: 12, Ram.programmCounter, obj.ram.getSpecificPCLATHBit( n: 4));

        // ...
    } else {
        //System.out.println("call " + i + " cycle 2");
    }

}
```

Abbildung 11 Codeausschnitt Call

MOVF

Der Inhalt von Register f wird an ein Ziel in Abhängigkeit der Destination geschrieben.

```
public void movf(Integer f) {
    int dest = obj.ram.getNthBitOfValue( n: 7, f);
    int addressInRam = obj.alu.and(f, 0b0111_1111);
    int valueOnAdress = obj.ram.getRamAt(addressInRam);

    if (dest == 0) {
        Ram.wRegister = valueOnAdress;
    } else if (dest == 1) {
        obj.ram.setRamAt(addressInRam, valueOnAdress);
    }
    if (valueOnAdress == 0) {
        obj.ram.setZeroBit(true);
    } else {
        obj.ram.setZeroBit(false);
    }
    //System.out.println("movf");
}
```

Abbildung 12 Codeausschnitt MOVF

Zunächst wird das Destination Bit ausgelesen, welches das siebte Bit des Übergabeparameters f ist. Danach wird die Adresse wieder maskiert und der Wert welcher an der Adresse steht wird ausgelesen. Wenn Destination = 0 dann wird der Wert an der Adresse in das W-Register geschrieben. Wenn Destination = 1 dann wird der Wert wieder an das Register aus dem es Kam geschrieben.

Hiermit kann auch das Zero Bit getestet werden, da dieses Beeinflusst wird, falls der Wert im Register 0 sein sollte.

RRF

```
public void rrf(Integer f) {
    int dest = obj.ram.getNthBitOfValue( n: 7, f);
    int addressInRam = obj.alu.and(f, 0b0111_1111);
    int valueOnAdress = 0;
    if (addressInRam != 0) {
        valueOnAdress = obj.ram.getRamAt(addressInRam);

        //indirect addr.
    } else if (addressInRam == 0) {
        valueOnAdress = obj.ram.getRamAt(obj.ram.getFSR());
    }

    int carry = obj.ram.getSpecificStatusBit( n: 0);

    if (obj.ram.getNthBitOfValue( n: 0, valueOnAdress) == 1) {
        if (carry == 1) {
            obj.ram.setCarryBit(false);
        }
        obj.ram.setCarryBit(true);
    } else {
        obj.ram.setCarryBit(false);
    }
    valueOnAdress >>= 1;
    valueOnAdress = obj.alu.and(valueOnAdress, 0xFF);
    valueOnAdress = obj.ram.setBit( bit: 7, valueOnAdress, carry);

    if (dest == 0) {
        Ram.wRegister = valueOnAdress;
    } else if (dest == 1) {
        if (addressInRam != 0) {
            obj.ram.setRamAt(addressInRam, valueOnAdress);
            //indirect addr.
        } else if (addressInRam == 0) {
            obj.ram.setRamAt(obj.ram.getFSR(), valueOnAdress);
        }

    }

    //System.out.println(Integer.toBinaryString(obj.ram.getStatus()));
    //System.out.println("rrf");
}
```

Abbildung 13 Codeausschnitt RRF

Hier wird der Inhalt von Register f ausgelesen und durch das Carry nach rechts rotiert. Bit Nummer sieben gibt wieder das Ziel an, wenn Destination = 0 dann wird das Rotierte Ergebnis in das W-Register geschrieben, wenn Destination = 1 dann wird das Ergebnis wieder in das Ursprungsregister geschrieben.

SUBWF

```
public void subWF(Integer f) {
    int dest = obj.ram.getNthBitOfValue( n: 7, f);
    int addressInRam = obj.alu.and(f, 0b0111_1111);
    int valueOnAdress = obj.ram.getRamAt(addressInRam);

    boolean b = obj.alu.isDigitCarry(Ram.wRegister, valueOnAdress);

    int result = valueOnAdress - Ram.wRegister;

    if (result ⩾ 0) {
        Ram.wRegister = Decoder.obj.alu.and(Ram.wRegister, 0xFF);
        Decoder.obj.ram.setCarryBit(true);
    } else {
        Decoder.obj.ram.setCarryBit(false);
    }

    if (result == 0) {
        Decoder.obj.ram.setZeroBit(true);
    } else {
        Decoder.obj.ram.setZeroBit(false);
    }
    Decoder.obj.ram.setDigitCarryBit(b);

    if (result < 0) {
        result = 256 + result;
    }

    if (dest == 0) {
        Ram.wRegister = result;
    } else if (dest == 1) {
        obj.ram.setRamAt(addressInRam, result);
    }
    //System.out.println("subwf");
}
```

Abbildung 14 Codeausschnitt SUBWF

Subtrahiert den Inhalt des W-Registers von dem wert, welcher in Register addressInRam steht. Das die Destination wird wieder über das siebte Bit übergeben. Wenn die Destination = 0 dann wird das Ergebnis wieder in das W-Register geschrieben, wenn Destination = 1 dann wird das Ergebnis wieder in das Register addressInRam geschrieben.

DECFSZ

```
public void decfsz(Integer f) {
    int dest = obj.ram.getNthBitOfValue( n: 7, f);
    int addressInRam = obj.alu.and(f, 0b0111_1111);
    int valueOnAdress = obj.ram.getRamAt(addressInRam);
    valueOnAdress--;
    if (valueOnAdress ≠ 0) {

        for (int i = Ram.programmCounter; i > 1; i--) {
            if (obj.programMemory.checkCycle(i)) {
                obj.programMemory.cycleList.remove(Integer.valueOf(i));
            }
        }
    } else {
        obj.programMemory.skipNextInstruction();
    }

    if (dest == 0) {
        Ram.wRegister = valueOnAdress;
    } else if (dest == 1) {
        obj.ram.setRamAt(addressInRam, valueOnAdress);
    }

    //System.out.println("decfsz");
}
```

Abbildung 15 Codeausschnitt DECFSZ

Der Inhalt des Registers addressInRam wird dekrementiert. Wenn die Destination = 0 dann wird das Ergebnis wieder in das W-Register geschrieben, wenn Destination = 1 dann wird das Ergebnis wieder in das Register addressInRam geschrieben.

XORLW

```
/**
 * The contents of the W register are
 * XOR'ed with the eight bit literal 'i'.
 * The result is placed in the W register.
 *
 * @param i 8 bit literal
 */
1 usage   nschwen1 +2
public void xorLW(Integer i) {
    Ram.wRegister = obj.alu.xor(Ram.wRegister, i);

    if (Ram.wRegister == 0) {
        obj.ram.setZeroBit(true);

    }

    // ...
}
```

Abbildung 16 Codeausschnitt XORLW

Der Inhalt von Register i wird mit dem Inhalt des W-Register mit einem Exklusiven Oder verrechnet und wieder in das W-Register geschrieben. Falls der Inhalt 0 sein sollte, wird das Zero-Flag gesetzt.

Interrupts

Es gibt beim PIC16F84 vier wichtige Interrupt quellen. Der Timer0, der RB0 , der RB4-RB7 und den EE-Write-Complete-Interrupt. Für jeden Interrupt muss das GIE Bit gesetzt sein. In dieser Simulation wurden der Timer0, der RB0 und der RB4-RB7 Interrupt realisiert.

Der Timer0 Interrupt wird bei einem überlauf des Timer0 Registers von 0xFF auf 0x00 dabei muss das T0IE und das T0IF gesetzt sein. Der RB0 Interrupt wird ausgelöst, wenn ein Flankenwechsel an PortB RB0 festgestellt wird, dabei muss das INTF und das INTE Bit gesetzt sein. Der RB4-RB7 Interrupt wird ausgelöst, wenn ein Flankenwechsel an PortB RB4-RB7 festgestellt wird. Hier muss das RBIE und das RBIF gesetzt sein. Die PortB Interrupts werden durch überprüfen der Checkboxen vor jeder Instruktion ausgelöst falls ein Wechsel passiert ist.

```
public void checkForInterrupt(int iOpCode) {

    if (obj.ram.getSpecificIntconBit( n: 7) == 1) {
        if (obj.timer.timerInterrupt && obj.ram.getSpecificIntconBit( n: 5) == 1) {
            executeTimerInterrupt(iOpCode);
        }
        if (obj.ram.getSpecificIntconBit( n: 4) == 1) {
            if (obj.mainFrame.RB0Checked()) {
                executeRB0Interrupt(iOpCode);
            }
        }

        if (obj.ram.getSpecificIntconBit( n: 3) == 1) {
            if (obj.mainFrame.RB4toRB7Checked()) {
                executeRB4toRB7Interrupt(iOpCode);
            }
        }
    }

}
```

Abbildung 17 Codeausschnitt der Interrupt Überprüfung

Fazit

Das Projekt war eine gute und interessante Möglichkeit unsere Java-Kenntnisse zu festigen und zu erweitern. Wir hatten im ersten Versuch vor einem Jahr leider Startschwierigkeiten, da wir nicht genau wussten wie und wo wir anfangen sollen. Durch diese hatten wir einen recht großen Zeitdruck, welchem wir gegen Ende nicht stand halten konnten und somit im ersten Versuch scheiterten. Dieses Jahr hingegen hatten wir schon eine grobe Vorstellung, was auf uns zukommt und konnten uns unsere Aufgaben besser einteilen. Dies hat uns ermöglicht frühzeitig fertig zu werden und den Zeitdruck zu minimieren. Da wir mehr Zeit hatten den PIC16F84 besser zu verstehen konnten die meisten Funktionen recht einfach implementiert werden. Jedoch kamen mit der Zeit immer mehr Erkenntnisse dazu und so kam es doch recht oft vor, dass man alten Code wieder abändern musste, da man anfangs beispielsweise die Indirekte Adressierung noch nicht „auf dem Schirm“ hatte. Wir mussten auch den Stack erneut abändern, da wir die Funktionsweise dessen anfangs noch nicht gekannt hatten. Da wir das Projekt von neuem aufgebaut haben und nicht den Alten Code von vor einem Jahr wiederverwendet haben, ließen sich anfängliche Fehler, welche schwer zu beheben waren sofort vermeiden. Auch konnte man manche Codesegmente mit in Java gegebenen Klassen austauschen und sparte somit viele Zeilen Code und eine Menge Zeit. Desweiteren haben wir noch nicht wirklich in Java eine GUI gebaut und hatte somit mithilfe von Swing über den Intellij-WindowBuilder einfach eine funktionale GUI schaffen.

Im Nachhinein, falls wir das Projekt erneut in Angriff nehmen sollten, würden wir den Ram anstatt in einem zweidimensionalen Array mit der länge 128 lieber in einem einzigen Array der Länge 256 realisieren. Desweiteren haben wir uns vorgenommen direkt die Indirekte Adressierung im Hinterkopf zu behalten um nicht im Nachhinein alten Code auf ineffiziente weise überarbeiten zu müssen.