PIC-SIMULATOR DOKUMENTATION

JANNIK SCHELLER & CHRISTIAN RENNER

Inhalt

Allgemeines

Simulator Arbeitsweise

Ein Simulator erstellt ein fiktives Modell eines in der Realität existierenden Objekts. Dabei soll dem Bediener ermöglicht werden unter Testbedingungen Phänomene zu reproduzieren oder darzustellen, die bei der tatsächlichen Umgebung wahrscheinlich auftreten. Je genauer ein Simulator das in der Realität existierende Objekt simulieren kann, desto besser ist er, da aus einer genaueren Darstellung der Wirklichkeit Ereignisse auch genauer reproduziert werden können. Im Fall des PIC-Simulators handelt es sich um eine Computersimulation. Computersimulationen machen Sinn, da man mit starker und skalierbarer Rechenpower viele unterschiedlich wahrscheinliche Ereignisse auf die Simulation projektieren kann. In unserem Fall handelt es sich um eine Dynamische Simulation des PIC-16F84, da wir unter anderem bei einer Mikrocontroller-Simulation der Zeit eine wesentliche Rolle zuordnen. Mit dem Implementierten Simulator können Assembler Programme getestet werden bevor sie auf den PIC-Microcontroller geladen werden. Ein großer Vorteil davon ist, dass der Microcontroller keinen Verschleiß hat und so nicht so schnell kaputt geht, wenn er simuliert wird. Außerdem können Probleme und Fehler am Assembler-Code mit dieser Simulation früh erkannt und dann verbessert werden.

Vor- und Nachteile einer Simulation

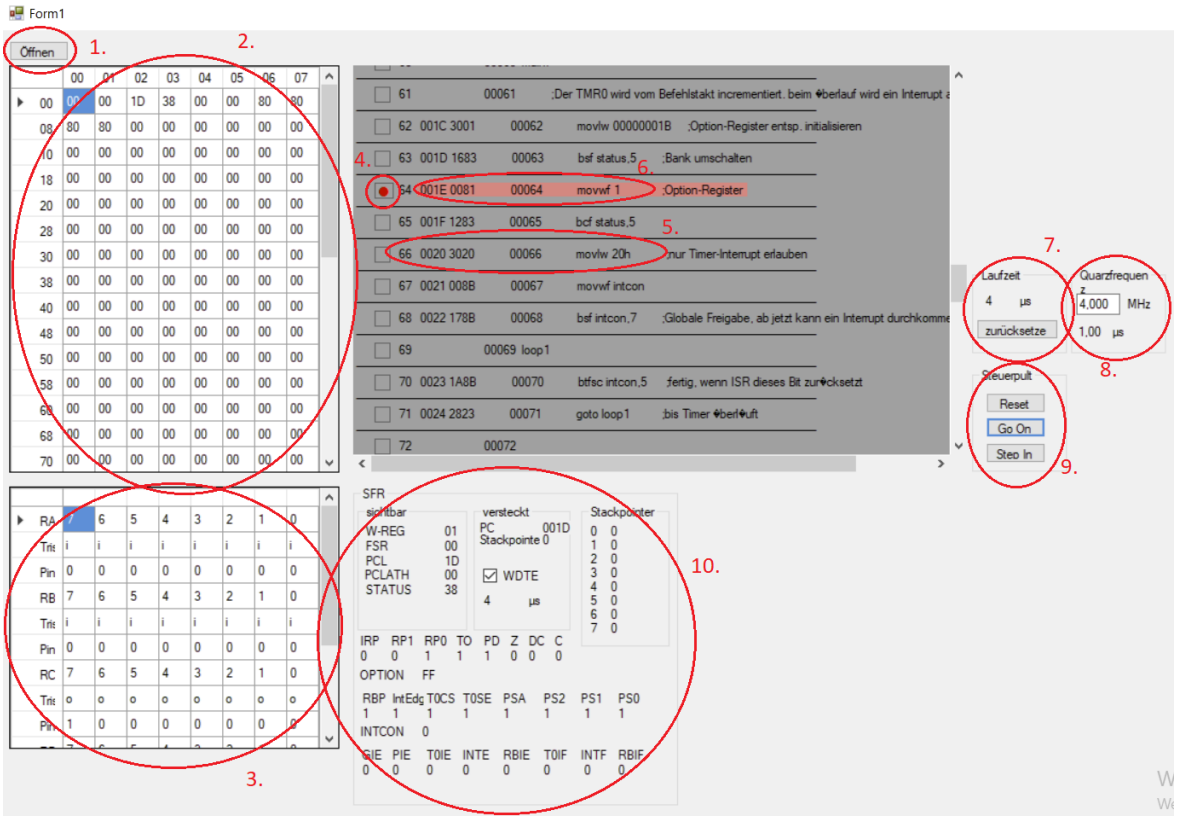
Vorteile

An Simulationen kann man nichts kaputt machen. Damit kann das Schadensrisiko an Geräten oder auch Menschen signifikant reduziert werden, bspw. bei der Simulation eines Atomkraftwerks. Eine Simulation kann dadurch gut zu Schulungszwecken (beispielsweise bei einem Flugsimulator) eingesetzt werden.

Nachteile

Bei Simulationen gibt es immer einen Informationsverlust im Verhältnis zum Original (beispielsweise, wenn man den Tunneleffekt, der innerhalb des Prozessors des Microcontrollers auftreten kann, vernachlässigt). Damit erzielt man nie eine hundertprozentige Übereinstimmung mit der Realität. Außerdem kann die Simulation selbst fehlerhaft sein, und dadurch könnten falsche Annahmen entstehen, die in der Realität schwerwiegende Auswirkungen hätten.

Programmoberfläche und deren Handhabung.



1. Button zum Öffnen des Code-Listings
2. Visualisierung des Speichers des Microcontrollers mit Funktion zum Ändern des Speicherinhalts
3. Visualisierung der PORT und TRIS Register mit Funktion zum Ändern des Registerinhalts
4. Möglichkeit Breakpoints zu setzen, um den Assembler-Code besser zu debuggen
5. Assembler Code, mit Zeilenangabe, Befehl als obcode und obcode Zeile
6. Anzeige, in welcher Codezeile sich der Benutzer des Simulators aktuell befindet
7. Laufzeitzähler mit Rücksetzfunktion
8. Quarzfrequenz mit Edit Möglichkeit und Anzeige der daraus resultierenden Befehlsdauer
9. Steuerpult mit…
   a. Reset zum Resetten des Programms
   b. Go zum Ausführen des Programms
   c. Step In zum schrittweisen Ausführen des Programms
10. Special Function Register mit…
    a. Visualisierung der sichtbaren Register (W-Register, FSR, PCL, PCLATH und STATUS)

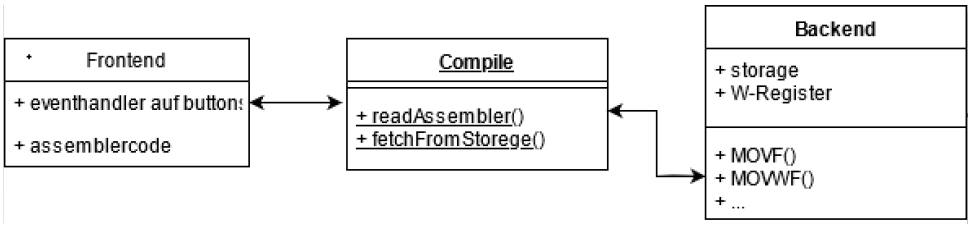
b. Visualisierung der versteckten Register (PC und Stackpointer)
c. Visualisierung der einzelnen Status-Bits
d. Visualisierung der OPTION-Register und der Einzelnen darin enthaltenen Bits
e. Visualisierung des INTCON Registers und der darin enthaltenen Bits

Realisierung

Grundkonzept

Zunächst war unser Grund Konzept, das Pic in 3 wesentliche Teile aufzuteilen:

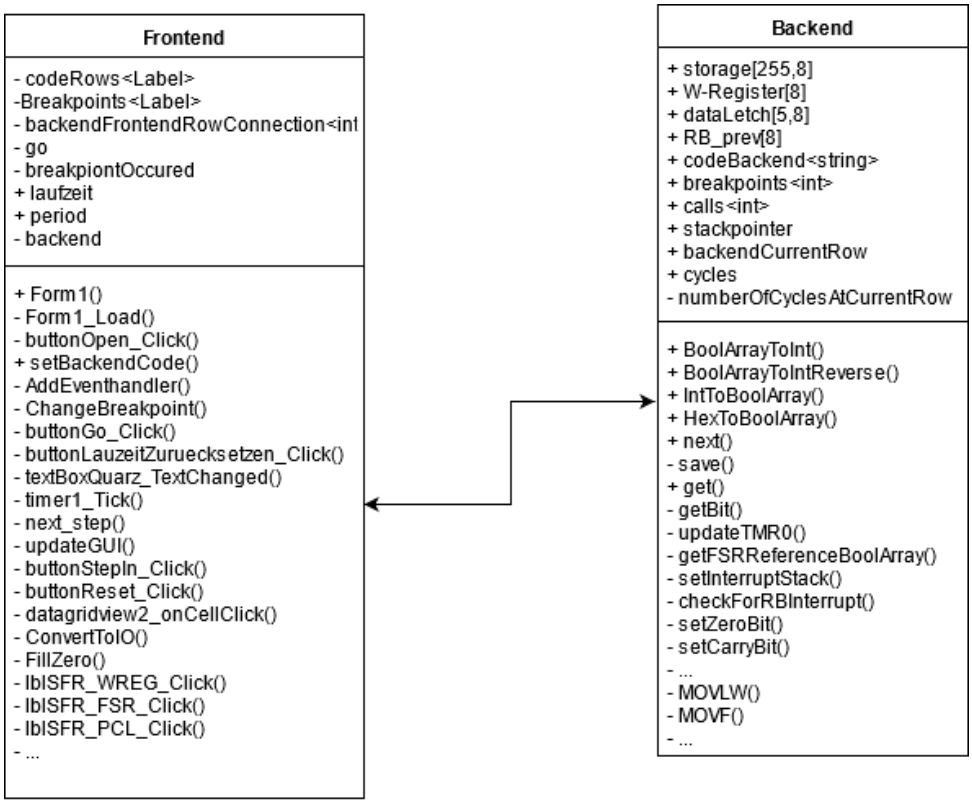
1. Frontend-Visualisierungs-Teil
2. Mittlerer Compilierender Teil
3. Backend Teil, in dem die Logik des Pics implementiert werden soll



Ziemlich schnell haben wir dann bemerkt, dass wir nicht den Assembler Code interpretieren müssen, sondern wir den fertigen opcode verwenden können. Damit ist bei uns die Compile Klasse weggefallen und wir konnten eine unmittelbare Verbindung zwischen Backend und Frontend erstellen.

Hier wird jetzt keine Compile Klasse mehr benötigt und die Opcodeabfrage findet einfach im Backend statt

Programmstruktur



Im Folgenden gibt es eine Erklärung der Variablen und Funktionen dieser Klassen:

Frontend

1. Variablen
   a. CodeRows ist eine Liste von Labeln, in denen der Asssembler Code dargestellt wird.
   b. In Breakpoints werden die Labels für das Setzen von Breakpoints erstellt.
   c. backendFrontendRowConnection ist eine int liste, mit der die aktuelle Frontend Codezeile mit der Codezeile im backend verknüpft werden kann.
   d. Go ist ein bool, der speichert ob der benutzer auf den go button gedrückt hat
   e. breakpointOccured wird gesetzt, wenn ein breakpoint erreicht wird. Damit wird der Timer gestoppt.
   f. in laufzeit wird die aktuelle Programm Laufzeit gespeichert.
   g. In period wird die aus der Quarz Frequenz resultierende Laufzeit für einen Befehl gespeichert.
   h. Backend ist das verwendete Objekt der Backend Klasse.
2. Funktionen

a. Form1() ist Konstruktor für Form1
b. Form1_Load enthält vorhergehende Initialisierungen des Speichers, sowie das Erstellen einiger GUI-Elemente
c. buttonOpen_Click() enthält den Zugriff auf den Windows-Explorer zum Öffnen der .lst-Files
d. setBackendCode() schreibt den obcode nacheinander in die code-Liste im backend.
e. AddEventhandler() fügt Eventhandler für onclickevents auf breakpoints hinzu
f. ChangeBrakpoint() wird aufgerufen, wenn ein in AddEventhandler() definiertes Event aufgetreten ist. In changeBreakpoint() wird dann der Breakpoint gesetzt.
g. buttonGo_Click() ist die Funktion, die aufgerufen wird, wenn auf den go-butten geklickt wird. In dieser Funktion wird der Timer gestartet, der in einem festgelegten Intervall Befehle aufruft und diese auch ausführt. Außerdem lässt sich die automatische Ausführung pausieren und im Fall eines Breakpoints fortsetzen.
h. buttonLaufzeitZuruecksetzen_Click() setzt den Laufzeitzähler zurück.
i. textBoxQuarz_TextChanged() wird aufgerufen, wenn sich der Text in der eingabetextbox für die quarzfrequenz verändert. Darin wird dann die period variable an die neue quarz-Geschwindigkeit angeglichen.
j. Timer1_Tick() wird alle 100ms ausgeführt und startet den nächsten Befehl, wenn der go-Button gedrückt wurde und an der Aktuellen Zeile kein Breakpoint existiert.
k. Next_step() führt im backend den nächsten Befehl aus, ändert die Färbung der aktuellen Codezeile und updatet den Laufzeitzähler
l. updateGUI() wird individuell aufgerufen und verändert die grafischen Visualisierungen der Register und Speichereinheiten.
m. buttonStepIn()_Click() wird bei Klick auf den button step in ausgeführt. Darin wird nur next_step() aufgerufen, um die nächste Code Zeile auszuführen.
n. buttonReset_Click() setzt alle gespeicherten und veränderten Bits zurück und beginnt das Programm von neuem.
o. datagridview2_onCellClick() wird bei Klick auf ein Feld in TRIS oder PORT Register aufgerufen und verändert den Wert grafisch und im Speicher.
p. ConvertToIO() konvertiert true in i und false in o
q. FillZero() fügt führende Nullen an die hex zahlen im Speicher hinzu.
r. Alle weiteren Funktionen wie bspw. lblSFR_WREG_Click() oder lblSFR_FSR_Click() sind Funktionen, die aufgerufen werden, wenn auf die entsprechende Visualisierung des Registers oder des Bits geklickt wird und dann den Wert im Speicher updaten

Backend

1. Variablen

a. Storage ist ein zweidimensionales Bool-Array, mit welchem der Speicher realisiert wird.
b. In W-Register werden die 8 bit des W-Registers gespeichert.
c. Der Data Letch wurde ebenfalls mit Hilfe eines zweidimensionalen Bool-Arrays implementiert.
d. RB_prev speichert den vorherigen Stand des PORTB Registers, um Änderungen in diesem zu detektieren.
e. codeBackend ist eine Liste, in der die Opcodes des gesamten Codes stehen.
f. Breakpoints bildet die im Frontend gesetzten Breakpoints auf die im backend verwendeten codezeilen ab.
g. In calls werden die Rücksprungaddressen für call Befehle oder Interrupts gespeichert.
h. backendCurrentRow ist die aktuell auszuführende Codezeile im backend.
i. numberOfCyclesAtCurrentRow speichert die Anzahl der in der aktuellen Codezeile benötigten Zyklen.

2. Funktionen

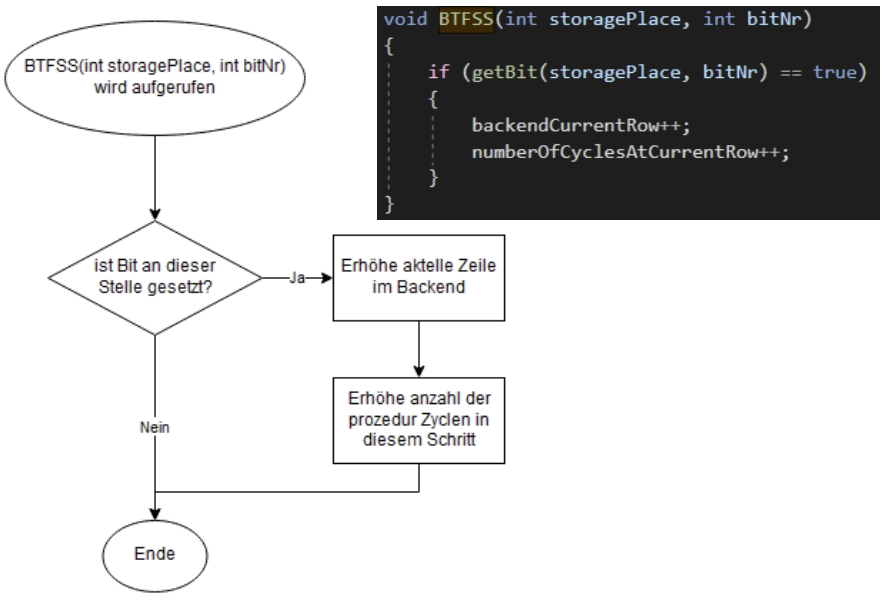
a. IntPow() ist eine Funktion die Potenzen für integer berechnet
b. BoolArrayToIntReverse() wandelt ein gedrehtes bool-Array in einen Integer.
c. BoolArrayToInt() wandelt ein normales Bool Array in einen Integer.
d. IntToBoolArray() wandelt einen Integer in ein Bool Array.
e. HexToBoolArray() wandelt eine Hexadezimalzahl in ein Bool Array
f. Next() wird auf Impuls der timer_Click oder der buttonStepIn_Click Funktion ausgeführt. In dieser Funktion wird der Opcode so analysiert, damit man eindeutig auf jedes Mnemonic zugreifen kann. Außerdem wird hier nach jedem Befehl abgefragt, ob ein Interrupt vorgekommen ist.
g. Get() gibt die Bits an der Speicherstelle des Übergabeparameters zurück.
h. getBit() gibt das Bit an einer übergebenen Stelle im Speicher zurück
i. updateTMR0() aktualisiert den TMR0
j. getFSRReferenceBoolArray() wird zum Verwenden der indirekten Adressierung benutzt.
k. setInterruptStack() updatet den Stack (calls) wenn Interrupts auftreten.
l. checkForRBInterrupt() überprüft, ob an RB ein Interrupt aufgetreten ist.
m. Außerdem gibt es Funktionen zum Setzen der einzelnen Status Bits.
n. Darüber hinaus gibt es Funktionen für jeden Mnemonic (z.b. MOVF oder MOVWF), welche den jeweiligen Befehl ausführen.

Programmiersprache

C# ist eine objektorientierte vielseitige Programmiersprache. Da C# ursprünglich exklusiv für Windows entwickelt wurde, lassen sich mit C# einfach Automatisierungen in Windows Umgebungen realisieren. Zunächst ist bei C# die Integration in Windows einfach, es werden bspw. keine speziellen Konfigurationen benötigt. Außerdem ist C# eine .NET-Sprache, dadurch kann ein C# Programm auf jedem beliebigen Windows Rechner ausgeführt werden, da das .NET-Framework bei heutigen Windows-PCs vorinstalliert ist. Außerdem ist C# eine verhältnismäßig sichere Sprache, da der Code kompiliert wird und so für vermeintliche Angreifer nur als binär-datei vorliegt. Des Weiteren ist C# weniger ausführlich als andere .NET-basierte Sprachen, was die Programmierung erheblich erleichtert. Außerdem lassen Sich mit Windows Forms und Visual Studio verhältnismäßig einfach GUI Elemente Implementieren und Eventhandler Anlegen.

Funktionsbeschreibung

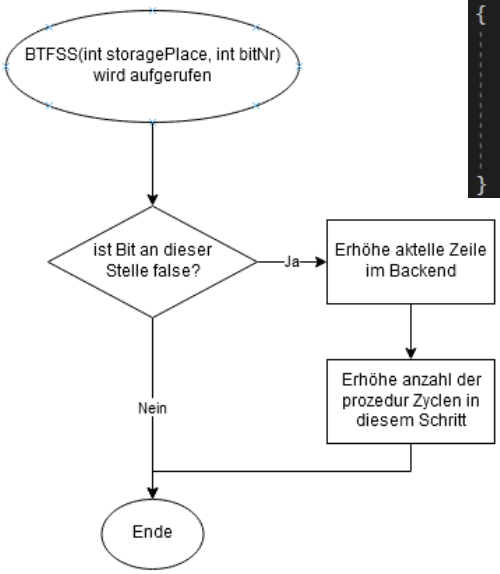
BTFSS(int storagePlace, int bitNr)




```
void BTFSS(int storagePlace, int bitNr)
{
    if (getBit(storagePlace, bitNr) == true)
    {
        backendCurrentRow++;
        numberOfCyclesAtCurrentRow++;
    }
}
```

Wenn BTFSS aufgerufen wird werden eine Speicher-Adresse und eine Bit-Nummer übergeben. Wenn das Bit an dieser Position im Speicher gesetzt ist wird die nächste Codezeile übersprungen (backendCurrentRow++) und der Laufzeitzähler muss einen Zyklus mehr hochzählen. Wenn es nicht gesetzt ist passiert nichts. Der nächste Befehl wird dann automatisch ausgeführt.

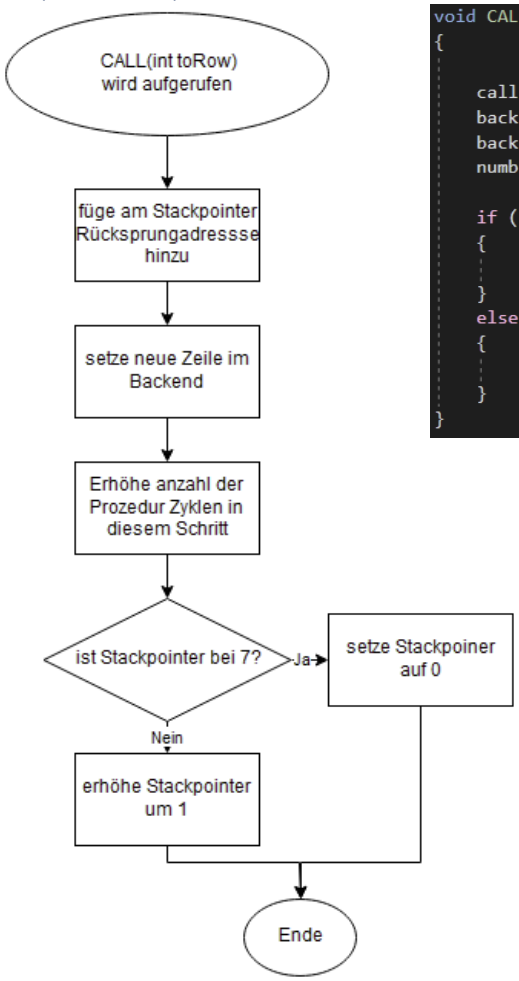
BTFSC(int storagePlace, int bitNr)




```
void BTFSC(int storagePlace, int bitNr)
{
    if (getBit(storagePlace,bitNr) == false)
    {
        backendCurrentRow++;
        numberOfCyclesAtCurrentRow++;
    }
}
```

Wenn BTFSC aufgerufen wird, werden eine Speicher-Adresse und eine Bit-Nummer übergeben. Wenn das Bit an dieser Position im Speicher nicht gesetzt ist, wird die nächste Codezeile übersprungen (backendCurrentRow++) und der Laufzeitzähler muss einen Zyklus mehr hochzählen. Wenn es gesetzt ist, passiert nichts. Der nächste Befehl wird dann automatisch ausgeführt.

CALL(int toRow)




```
void CALL(int toRow)
{

    calls.Insert(stackpointer, backendCurrentRow + 1);
    backendCurrentRow = toRow;
    backendCurrentRow--;
    numberOfCyclesAtCurrentRow++;

    if (stackpointer == 7)
    {
        stackpointer = 0;
    }
    else
    {
        stackpointer++;
    }
}
```

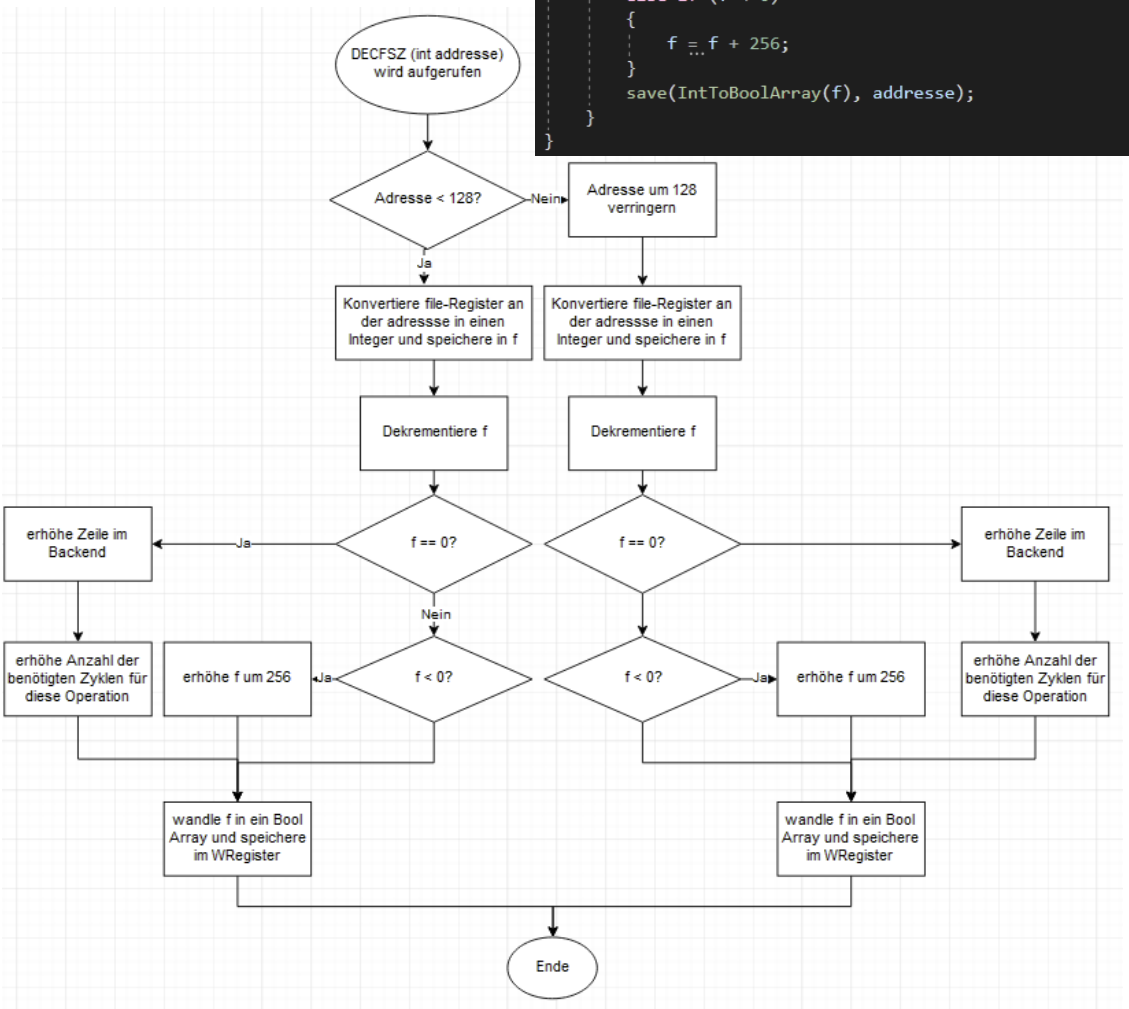
Wenn Call aufgerufen wird, wird zunächst die Rücksprungadresse auf dem Stack „gesichert“. Dann wird die neue Zeile Im backend gesetzt, diese muss dann noch dekrementiert werden, da sonst automatisch hochgezählt wird nach dem Befehl. Anschließend wird noch die Anzahl der Zyklen für diesen Befehl erhöht, damit der Laufzeitzähler stimmt. Um den Ringpuffer zu implementieren wird der Stackpointer erhöht oder auf 0 zurückgesetzt, je nachdem ob er bei 7 angekommen ist oder nicht.

DECFSZ(int addresse)

Wenn DECFSZ aufgerufen wird, wird zunächst das Destination Bit mit der Abfrage addresse < 128 abgefragt. Anschließend wird nach der Wandlung des Bool Arrays aus dem Speicher in einen Integer dieser integer dekrementiert. Dann wird je nach Ergebnis die nächste Zeile geskippt, oder ein überlauf erzeugt, oder einfach nur dekrementiert. Am ende wird je nach destination Bit das Ergebnis ins WRegister oder ins File-Register geladen.

```
void DECFSZ(int addresse)
{
    if (addresse < 128)
    {
        int f = BoolArrayToIntReverse(get(addresse));
        f--;
        if (f == 0)
        {
            backendCurrentRow++;
            numberOfCyclesAtCurrentRow++;
        }
        else if (f < 0)
        {
            f = f + 256;
        }
        WRegister = IntToBoolArray(f);
    }
    else
    {
        addresse = addresse - 128;

        int f = BoolArrayToIntReverse(get(addresse));
        f--;
        if (f == 0)
        {
            backendCurrentRow++;
            numberOfCyclesAtCurrentRow++;
        }
        else if (f < 0)
        {
            f = f + 256;
        }
        save(IntToBoolArray(f), addresse);
    }
}
```



Realisierung der Flags

Die Flags werden immer direkt bei den Befehlen, wo sie beeinflusst werden, gesetzt oder rückgesetzt. Für das Digit Carry wurden beispielsweise die letzten 4 Bit addiert bzw. subtrahiert um damit abzufragen, ob das Bit gesetzt werden muss. Das Flag zum Umschalten der Bank wird normal mit dem BSF Befehl (oder manuell Über die GUI) gesetzt. In der Save Funktion wird dieses Bit abgefragt und dementsprechend auf die richtige Bank gespeichert.

Zusammenfasung

Implementierungsstand

Es wurden alle Literalbefehle mit direkter und indirekter Adressierung implementiert. Außerdem wurden der TMR0 (mit Vorteiler und berücksichtigung des OPTION-Registers), der RB0 und der RB4-RB7 Interrupt programmiert. Auch der Watchdog wurde mit Vorteiler implementiert. In der GUI wurden Breakpoints, Laufzeitzähler, Toggle-Funktion, eine frei wählbare Quarzfrequenz (im Zusammenhang mit dem Laufzeitzähler), das Markieren des nächsten Befehls und Fenster sowie Edit Möglichkeit für Register. Außerdem wurden die Sleepfunktion, der Watchdog und die TRIS- und PORT Register implementiert.

Fazit

Durch die Programmierung des PIC-Simulators konnte das Wissen aus den Vorlesungen des letzten Semesters ausführlich auf den Prüfstand gestellt werden. Dabei sind immer wieder einzelne Verständnisprobleme zum Vorschein gekommen, welche wiederum in den meisten Fällen vom jeweils anderen Teampartner geklärt werden konnten. In den Fällen, wo es bei beiden zu Wissenslücken kam, konnte durch die gemeinsame oder einzelne Recherche der Themenblätter und der PIC-Dokumentation auch dieses fehlende Wissen letztendlich erworben werden.
Neben dem Lösen von Verständnisproblemen konnte zudem ein sehr intensiver Einblick in das Verhalten und in die Eigenarten des Microcontrollers erhalten werden. Das Wissen um den PIC und die hardwarenahe Programmierung konnte somit weiter gefestigt und vertieft werden. So waren einige Besonderheiten des PICs am Anfang noch sehr schwer nachzuvollziehen, wurden aber im Laufe des Projekts zu Selbstverständlichkeiten.

Im Projektverlauf konnten wir selbst auch einige neue Erkenntnis erwerben. So nehmen wir zum Beispiel mit, dass wir uns von Anfang an noch tiefer mit der Programmstruktur beschäftigen müssen. Wichtig war es für uns zu lernen, dass es im Laufe des Projekts immer wieder zu Abweichung von der zuvor festgelegten Programmstruktur kommt, weshalb eine frühzeitige

Dokumentation der gemeinsam überlegten Struktur notwendig ist.
Durch immer wieder kleinere oder größere Anpassungen an der Struktur kam es im Quellcode zu einigen nicht gewollten Abhängigkeiten, welche durch gemeinsames Bugfixing gelöst werden konnten. Durch die frühe Dokumentation der Struktur wollen wir diese Abhängigkeiten in zukünftigen Projekten verhindern.

Des Weiteren haben wir gelernt, dass in zukünftigen Projekten eine rechtzeitige Zeitplanung der einzelnen Arbeitsschritte notwendig ist, um einen Zeitengpass am Ende zu vermeiden. Durch den Zeitengpass mussten am Ende einige Arbeitsschritte schneller als ursprünglich geplant implementiert werden, welches an einigen Stellen im Quellcode auch sichtbar wird.

Trotz des engen Zeitplans am Ende, hat uns das Projekt viel Spaß gemacht. Wir konnten gemeinsam als Team Probleme lösen und immer wieder uns über erfolgreiche durchgelaufene Tests freuen. Über weitere Erfolge konnten wir uns insbesondere dann freuen, wenn wir Selbständig an unterschiedlichen Code Teilen gearbeitet haben und diese dann Zusammenführung miteinander funktionierten.
Aufgrund einer Projektarbeit in einem anderen Kurs haben wir auch an einigen Stellen PairProgramming ausprobiert. Auch hierbei hat uns das gemeinsame Programmieren viel Spaß gemacht und die Ergebnisse des PairProgramming haben uns von dieser Methode sehr überzeugt.