|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| DHBW-Logo  Entwicklung eines Simulators für den PIC16F4 Microcontroller  Für die Prüfung zum  Bachelor of Engineering  des Studiengangs Informatik Studienrichtung Informationstechnik  an der  Dualen Hochschule Baden-Württemberg Karlsruhe  von  Paul Giesa & Chris Steven Todt  Bearbeitungszeitraum:  03.04.2017 - 19.06.2017   |  |  | | --- | --- | | Kurs | TINF15B3 | | Gutachter/In der Studienakademie | Hr. Lehmann | |  |  | |  |  | |

Inhaltsverzeichnis

Inhaltsverzeichnis 2

Abbildungsverzeichnis 3

Tabellenverzeichnis 4

Abkürzungsverzeichnis 5

1. Vorwort 6

2. Einleitung 7

3. Allgemeines 7

4. Simulator 7

4.1 Aufbau 7

5. Struktur 7

6. Implementierung der Befehle 8

6.1 BTFSC (Bit Test f, Skip if Clear) 10

6.2 BTFSS (Bit Test f, Skip if Set) 12

6.3 CALL (Call Subroutine) 14

6.4 MOVF (Move f) 16

6.5 RRF (Rotate Right f through Carry) 18

6.6 SUBWF (Subtract W from f) 20

6.7 DECFSZ (Decrement f, Skip if 0) 22

6.8 XORLW (Exclusive OR Literal with W) 24

7. Programmiersprache 28

8. Fazit 28

Literaturverzeichnis 29

Abbildungsverzeichnis

[Abbildung 1: Abkürzungen – Auszug aus PIC Doku 9](file:///D:\Stiiift_Inc\PicSim_Doku.docx#_Toc483350574)

[Abbildung 2: BTFSC – Auszug PIC 10](file:///D:\Stiiift_Inc\PicSim_Doku.docx#_Toc483350575)

[Abbildung 3: PAP BTFSC 10](file:///D:\Stiiift_Inc\PicSim_Doku.docx#_Toc483350576)

[Abbildung 4: BTFSS – Auszug PIC 12](file:///D:\Stiiift_Inc\PicSim_Doku.docx#_Toc483350577)

[Abbildung 5: PAP BTFSS 12](file:///D:\Stiiift_Inc\PicSim_Doku.docx#_Toc483350578)

[Abbildung 6: CALL – Auszug PIC 14](file:///D:\Stiiift_Inc\PicSim_Doku.docx#_Toc483350579)

[Abbildung 7: PAP CALL 14](file:///D:\Stiiift_Inc\PicSim_Doku.docx#_Toc483350580)

[Abbildung 8: MOVF – Auszug PIC 16](file:///D:\Stiiift_Inc\PicSim_Doku.docx#_Toc483350581)

[Abbildung 9: PAP MOVF 16](file:///D:\Stiiift_Inc\PicSim_Doku.docx#_Toc483350582)

[Abbildung 10: RRF - Auszug PIC 18](file:///D:\Stiiift_Inc\PicSim_Doku.docx#_Toc483350583)

[Abbildung 11: PAP RRF 18](file:///D:\Stiiift_Inc\PicSim_Doku.docx#_Toc483350584)

[Abbildung 12: SUBWF - Auszug PIC 20](file:///D:\Stiiift_Inc\PicSim_Doku.docx#_Toc483350585)

[Abbildung 13: PAP SUBWF 20](file:///D:\Stiiift_Inc\PicSim_Doku.docx#_Toc483350586)

[Abbildung 14: DECFSZ - Auszug PIC 22](file:///D:\Stiiift_Inc\PicSim_Doku.docx#_Toc483350587)

[Abbildung 15: PAP DECFSZ 22](file:///D:\Stiiift_Inc\PicSim_Doku.docx#_Toc483350588)

[Abbildung 16: XORLW - Auszug PIC 24](file:///D:\Stiiift_Inc\PicSim_Doku.docx#_Toc483350589)

[Abbildung 17: PAP XORLW 24](file:///D:\Stiiift_Inc\PicSim_Doku.docx#_Toc483350590)

Tabellenverzeichnis

[Tabelle 1: Erläuterung Pre- & PostInstructions 9](file:///D:\Stiiift_Inc\PicSim_Doku.docx#_Toc483350693)

Abkürzungsverzeichnis

BTFSC Bit Test File, Skip if Clear

BTFSS Bit Test File, Skip if Set

CALL Call Subroutine

DECFSZ Decrement File, Skip if Zero

PAP Programmablaufplan

PC Programm Counter

RRF Rotate Right f through Carry

SUBWF Subtract W from f

XORLW Exclusive OR Literal with W

# Vorwort

Im Fach Rechnertechnik II soll der Aufbau und die Funktionsweise eines Microcontroller gelernt werden. Über den „Umweg“ ein Simulator-Programm zu schreiben, das die Funktionen eines realen oder imaginären Controllers nachbildet, müssen die Studenten neben dem Studium des Datenblattes auch die bereits erlernten Fertigkeiten aus der Vorlesung Software-Engineering, Digitaltechnik und Rechnertechnik I anwenden. Eine einfache Hardwarebeschaltung an der seriellen oder parallelen Schnittstelle bildet die Brücke zwischen virtueller und realer Welt.

# Einleitung

Im Rahmen der Vorlesung Systemnahe Programmierung sollen die Studenten einen Simulator für den Microcontroller PIC16F84 der Firma Microchip Technology Inc. Entwickeln.

# Allgemeines

## Definition Simulation

Eine Simulation soll dabei helfen ein komplexes System auf eine anschaulichere Ebene herunterzubrechen. Die Durchführung von Analysen soll dabei erleichtert werden. Auch die Durchführung von Test an einem Modell und der damit verbundene Erkenntnisgewinn über das reale System steht bei der Simulation im Vordergrund. Wird das Modell realisiert bzw. implementiert, spricht man von einem Simulator.

## Vorteile

Es sprechen viele Gründe für den Einsatz einer Simulation:

* Analysen am realen System können teuer, aufwendig, gefährlich oder ethisch nicht vertretbar sein.
* Bevor das System in der Realität zum Einsatz kommt, können sämtliche Szenarien mit Hilfe eines Simulators getestet werden.
* In manchen Fällen kann das System in der Realität im Gegensatz zur Simulation nicht beobachtet werden.
* Simulationen können reproduziert werden.

## Nachteile

Auf der anderen Seite gibt es auch Aspekte die nicht für den Einsatz einer Simulation sprechen:

* Eine Simulation verfügt über begrenzte Ressourcen (z.B. Rechenkapazität, Zeit, finanzielle Mittel), sodass die Darstellung der Realität nicht immer 1:1 erfolgen kann.
* Auch Ungenauigkeiten und Abweichungen können zu verfälschten Darstellung des realen Systems führen.[[1]](#footnote-1)

# Simulator

Im folgenden Kapitel wird der Aufbau des Simulators erläutert.

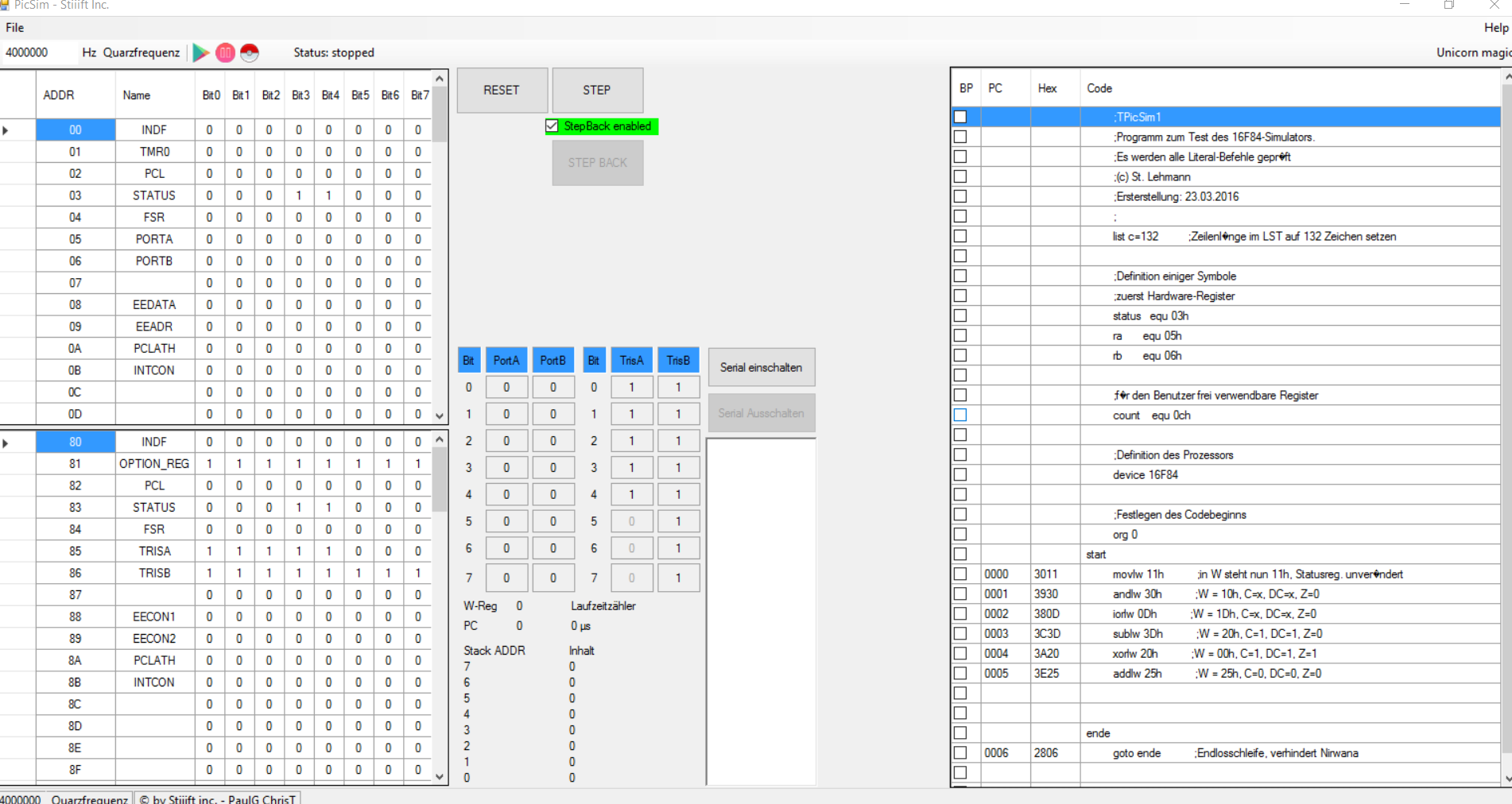


Abbildung 1: GUI

## GUI

Die GUI besteht aus mehreren Elementen, welche nachfolgend

# Struktur

# Implementierung der Status-Bits / Flags

Beim PIC16F84 gibt es drei Statut-Bits, welche im Statusregister (03H & 83H) zu finden sind:

* Zero Bit: 2. Bit im Statusregister
* DigitCarry Bit: 1. Bit im Statusregister
* Carry Bit: 0. Bit im Statusregister

Diese werden in den Funktionen CheckZero(), CheckCarry() und CheckDigitCarry() überprüft. Diese Funktionen werden bei den Befehlen ausgeführt, bei welchen die jeweiligen Status-Bits betroffen sein können. Der PIC16F84 Doku kann entnommen werden, bei welcher Funktion welches Status-Bit geprüft werden muss.

Die Funktionsweise dieser Funktionen wird im Folgenden näher erläutert.

## Zero-Bit

Das Zero-Bit wird gesetzt, wenn das Ergebnis eines Befehles 0 (Zero) ist.

public void CheckZero(int val)

{

if (val == 0)

{

mem.ram[2, Const.STATUS] = 1;

}

else

{

mem.ram[2, Const.STATUS] = 0;

}

}

## DigitCarry-Bit

Das DigitCarry-Bit wird gesetzt, wenn ein Übertrag vom 3. Auf das 4. Bit stattfindet.

public void CheckCarry(int val)

{

if (val > 255)

{

mem.ram[0, Const.STATUS] = 1;

}

else

{

mem.ram[0, Const.STATUS] = 0;

}

}

## Carry-Bit

Das Carry-Bit wird gesetzt, wenn durch eine Operation ein Wert das 7. Bit überschreitet und ein Übertrag vom 7. Auf das 8. Bit stattfindet (Wert > 127).

public void CheckDigitCarry(int val)

{

if (val > 15)

{

mem.ram[1, Const.STATUS] = 1;

}

else

{

mem.ram[1, Const.STATUS] = 0;

}

}

# Implementierung der Befehle

Die Realisierung der Maschienenbefehle des Microcontrollers wird beispielhaft an den folgenden Befehlen tiefgehender erläutert:

* BTFSC (Bit Test, Skip if Clear)
* BTFSS (Bit Test, Skip if Set)
* CALL (Call Subroutine)
* MOVF (Move f)
* RRF (Rotate Right f through Carry)
* SUBWF (Subtract W from f)
* DECFSZ (Decrement f, Skip if 0)
* XORLW (Exclusive OR Literal with W)

Die Umsetzung der Befehle findet in der Klasse "Befehle“ statt. Die Befehle werden aus kompilierten Assemblerdateien ausgelesen. Beim Abarbeiten eines Befehls wird dieser an ein Objekt der Klasse "Decoder" übergeben, welcher den korrekten Befehl ausführt.

Vor jedem Befehl wird die Funktion PreInstructions() und nach jedem Befehl die Funktion PostInstructions() ausgeführt. Diese werden in Tabelle 1: Erläuterung Pre- & PostInstructions auf Seite 9 stichwortartig erläutert.

Im Anschluss wird die Funktion der Interrupts kurz erläutert.

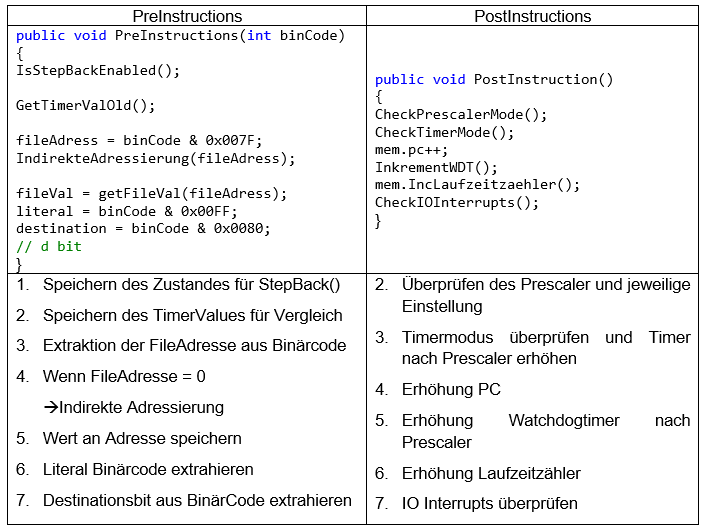


Tabelle 1: Erläuterung Pre- & PostInstructions

Das weitere Vorgehen wird im Folgenden an beispielhaften Befehlen erläutert.

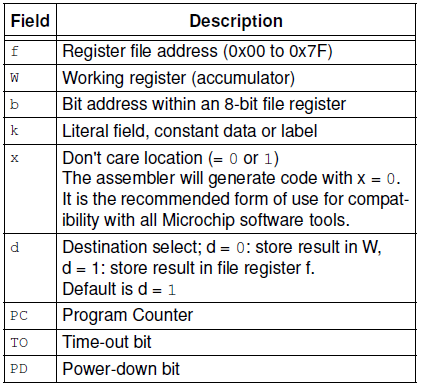


Abbildung 2: Abkürzungen – Auszug aus PIC Doku

## BTFSC (Bit Test f, Skip if Clear)C:\Users\Chris Todt\AppData\Local\Microsoft\Windows\INetCache\Content.Word\BTFSC.PNG

Abbildung 3: BTFSC – Auszug PIC

Abbildung 4: PAP BTFSC

**Implementation**:

Das zu überprüfende Bit wird aus dem Binärcode ausgelesen. Das Auslesen findet über 7-faches rechtsshiften statt, da Bit 7 – 9 die Stelle des Bits darstellen.

Nachfolgend wird überprüft, ob dieses Bit im Wert der Datei nicht gesetzt (== 0) ist. Wenn nicht gesetzt wird der nächste Befehl übersprungen und ein NOP ausgeführt.

public void btfsc(int binCode)

{

PreInstructions(binCode);

bit = (byte)((binCode >> 7) & 0x7);

int Bool = mem.ram[bit, fileAdress];

if (Bool == 0)

{

TwoCycles();

mem.pc++;

}

PostInstruction();

}

## C:\Users\Chris Todt\AppData\Local\Microsoft\Windows\INetCache\Content.Word\BTFSS.PNGBTFSS (Bit Test f, Skip if Set)

Abbildung 5: BTFSS – Auszug PIC

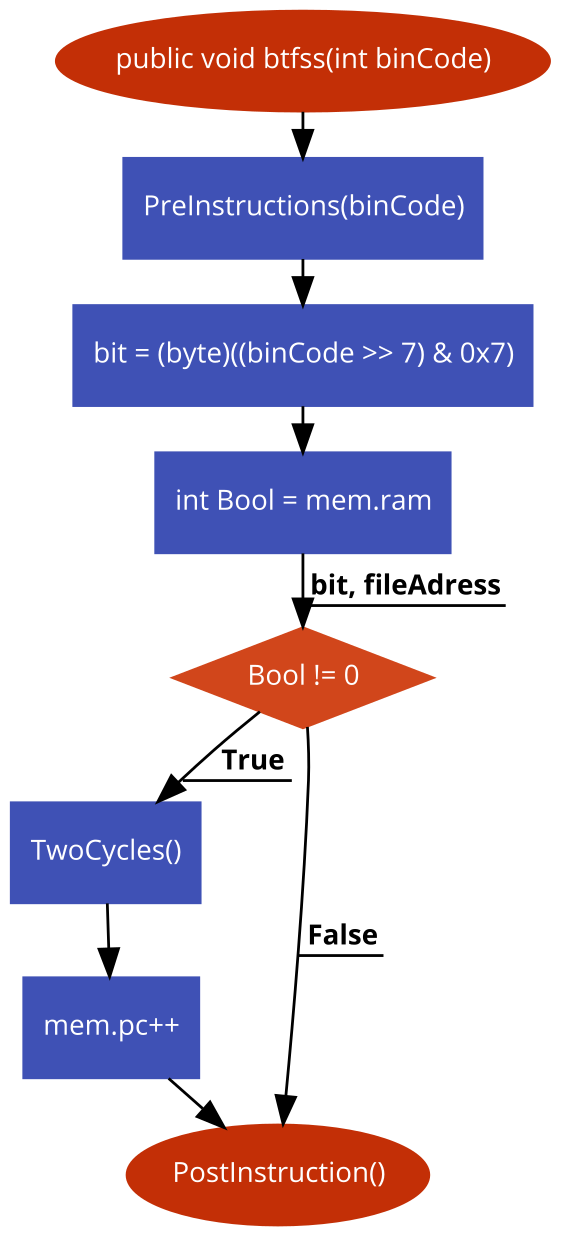


Abbildung 6: PAP BTFSS

**Implementation**:

Das zu überprüfende Bit wird aus dem Binärcode ausgelesen. Das Auslesen findet über 7-faches rechtsshiften statt, da Bit 7 – 9 die Stelle des Bits darstellen.

Nachfolgend wird überprüft, ob dieses Bit im Wert der Datei gesetzt (== 1) ist.   
Wenn gesetzt wird der nächste Befehl übersprungen und der PC erhöht, sowie die Instruktionen für zwei Zyklen ausgeführt.

public void btfss(int binCode)

{

PreInstructions(binCode);

bit = (byte)((binCode >> 7) & 0x7);

int Bool = mem.ram[bit, fileAdress];

if (Bool == 1)

{

TwoCycles();

mem.pc++;

}

PostInstruction();

}

## C:\Users\Chris Todt\AppData\Local\Microsoft\Windows\INetCache\Content.Word\CALL.PNGCALL (Call Subroutine)

Abbildung 7: CALL – Auszug PIC

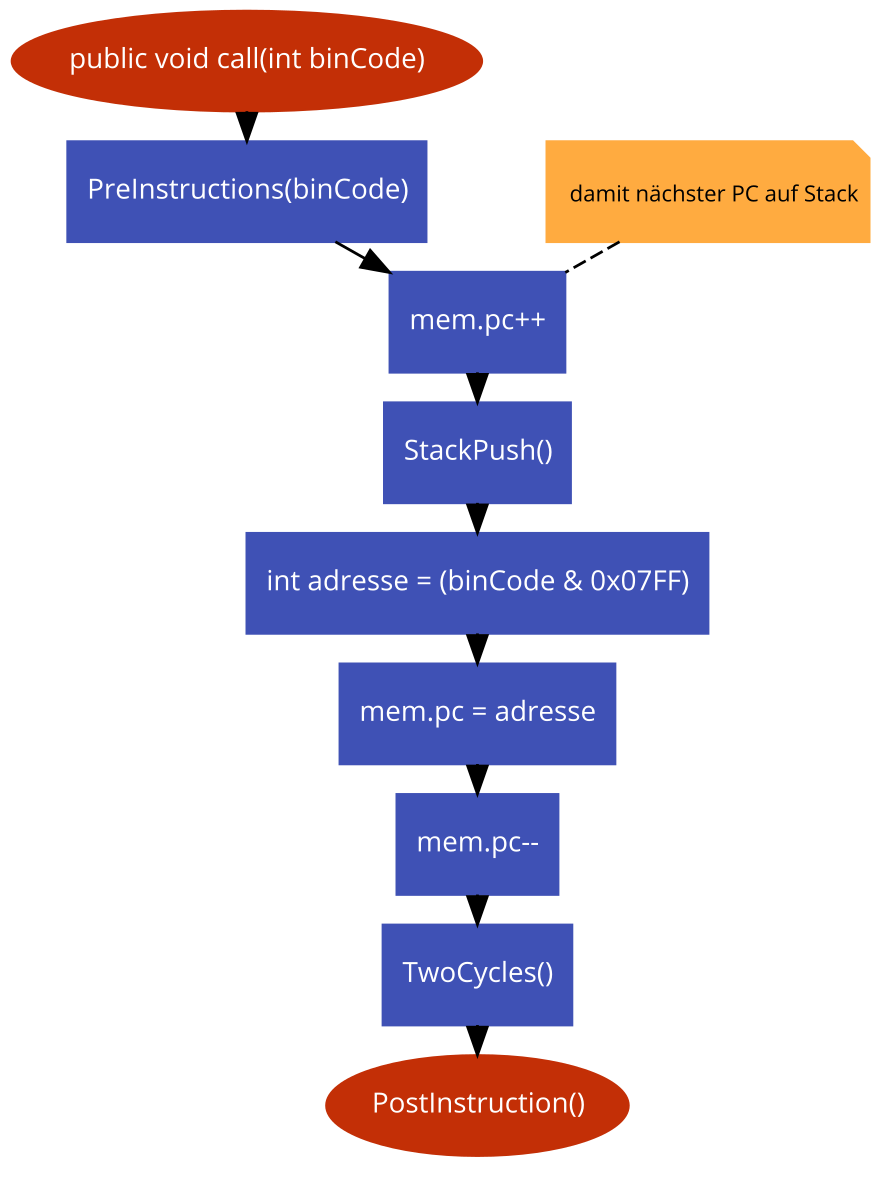


Abbildung 8: PAP CALL

**Implementation**:

Der PC wird erhöht, damit die korrekte Befehlsadresse, die nächste, auf den Stack gespeichert werden kann.

Anschließend wird die Befehlsadresse, zu der gesprungen werden soll, aus dem Binärcode extrahiert, indem dieser mit 0x07FF logisch verundet wird. Der PC wird auf die extrahierte Adresse gesetzt.

public void call(int binCode)

{

PreInstructions(binCode);

mem.pc++; //damit nächster PC auf Stack

StackPush();

int adresse = (binCode & 0x07FF);

mem.pc = adresse;

mem.pc--;

TwoCycles();

PostInstruction();

}

## C:\Users\Chris Todt\AppData\Local\Microsoft\Windows\INetCache\Content.Word\MOVF.PNGMOVF (Move f)

Abbildung 9: MOVF – Auszug PIC

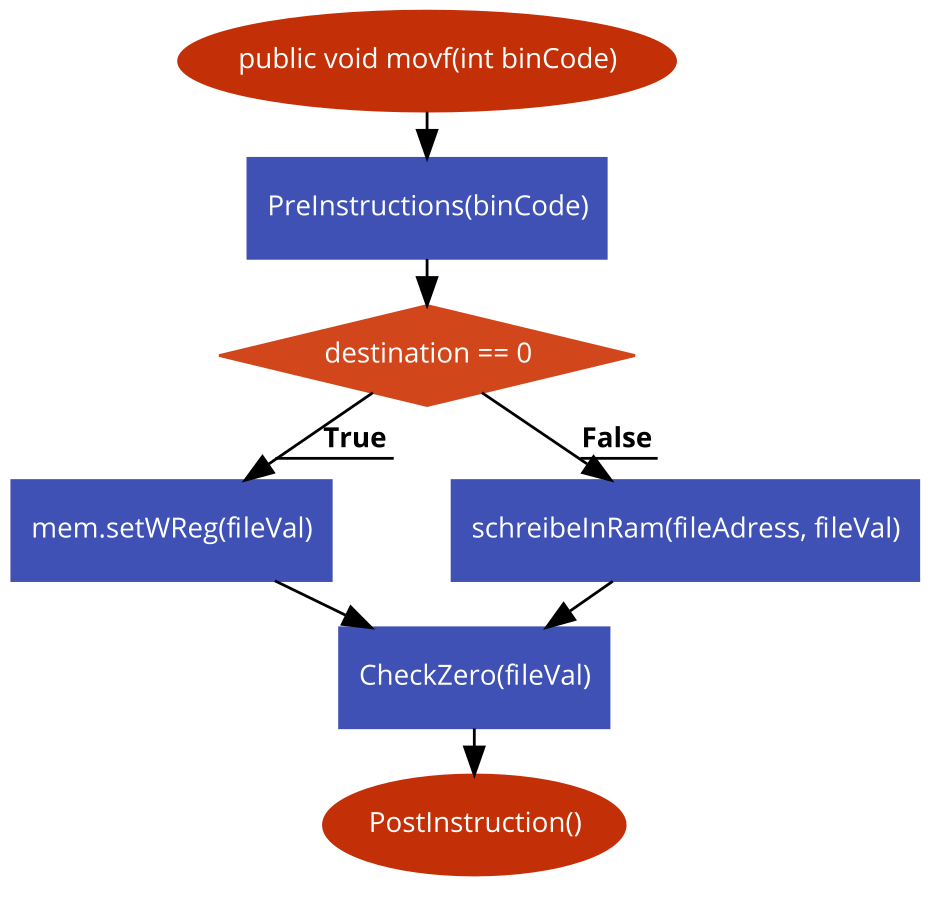


Abbildung 10: PAP MOVF

**Implementation**:

Das Zielbit der Operation wird überprüft. Ist das Zielbit 0, so wird der Wert aus f in das W-Register kopiert. Ist das Zielbit 1, so wird der Wert aus f in f kopiert.

Anschließend wird das Zero-Bit überprüft und evtl. gesetzt.

public void movf(int binCode)

{

PreInstructions(binCode);

if (destination == 0)

{

mem.setWReg(fileVal);

}

else

{

schreibeInRam(fileAdress, fileVal);

}

CheckZero(fileVal);

PostInstruction();

}

## C:\Users\Chris Todt\AppData\Local\Microsoft\Windows\INetCache\Content.Word\RRF.PNGRRF (Rotate Right f through Carry)

Abbildung 11: RRF - Auszug PIC

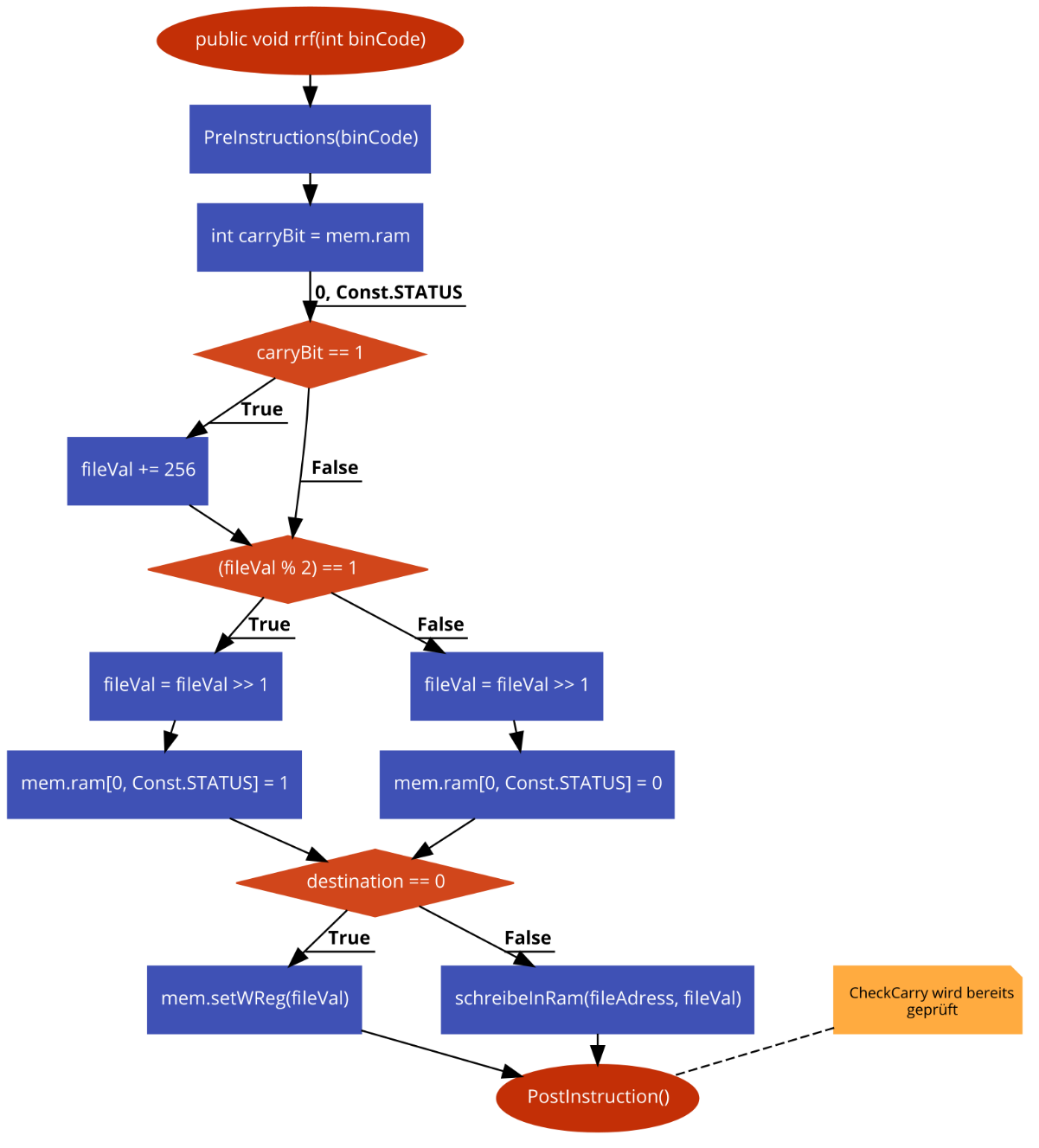


Abbildung 12: PAP RRF

**Implementation**:

Zunächst wird das CarryBit aus dem Speicher gelesen, wenn dieses gesetzt ist wird das 9. Bit des Wertes an der Adresse gesetzt (+256). Somit wird das CarryBit beim bitshiften berücksichtigt. Danach wird geprüft ob das 0. Bit des Wertes an der Adresse gesetzt ist – Wenn ja wird nach dem shiften das CarryBit gesetzt. Ist das 0. Bit nicht gesetzt wird nach dem shifting das CarryBit nicht gesetzt. Ist das Zielbit 0, so wird der Wert aus f in das W-Register kopiert. Ist das Zielbit 1, so wird der Wert aus f in f kopiert.

public void rrf(int binCode)

{

PreInstructions(binCode);

int carryBit = mem.ram[0, Const.STATUS];

if (carryBit == 1)

{

fileVal += 256;

}

if ((fileVal % 2) == 1)

{

fileVal = fileVal >> 1;

mem.ram[0, Const.STATUS] = 1;

}

else

{

fileVal = fileVal >> 1;

mem.ram[0, Const.STATUS] = 0;

}

if (destination == 0)

{

mem.setWReg(fileVal);

}

else

{

schreibeInRam(fileAdress, fileVal);

}

//CheckCarry wird bereits geprüft

PostInstruction();

}

## 

## C:\Users\Chris Todt\AppData\Local\Microsoft\Windows\INetCache\Content.Word\SUBWF.PNGSUBWF (Subtract W from f)

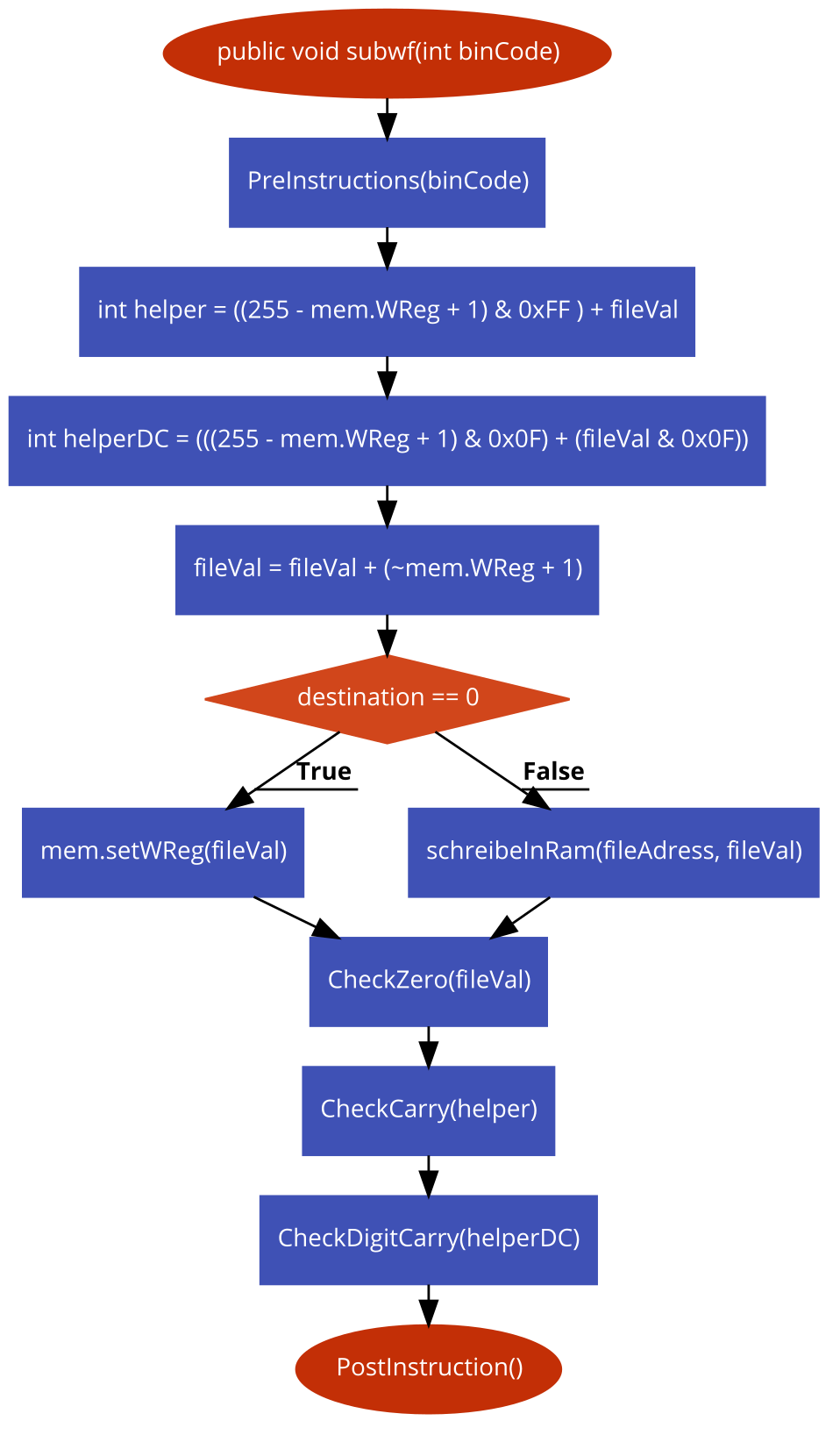


Abbildung 13: SUBWF - Auszug PIC

Abbildung 14: PAP SUBWF

**Implementation**:

Zunächst werden Hilfsvariablen für das Carry und das DigitCarry definiert. Hierbei wird das 2er Komplement des W-Registers gebildet, mit dem später geprüft wird, ob dabei das Carry bzw. DigitCarry gesetzt werden soll.

Das ist notwendig, da bei der Operation **fileVal + (~mem.WReg + 1)** nur für eines der drei Statusregisterbits geprüft werden könnte.

public void subwf(int binCode)

{

PreInstructions(binCode);

int helper = ((255 - mem.WReg + 1) & 0xFF ) + fileVal;

int helperDC = (((255 - mem.WReg + 1) & 0x0F) + (fileVal & 0x0F));

fileVal = fileVal + (~mem.WReg + 1);

if (destination == 0)

{

mem.setWReg(fileVal);

}

else

{

schreibeInRam(fileAdress, fileVal);

}

CheckZero(fileVal);

CheckCarry(helper);

CheckDigitCarry(helperDC);

PostInstruction();

}

## C:\Users\Chris Todt\AppData\Local\Microsoft\Windows\INetCache\Content.Word\DECFSZ.PNGDECFSZ (Decrement f, Skip if 0)

Abbildung 15: DECFSZ - Auszug PIC

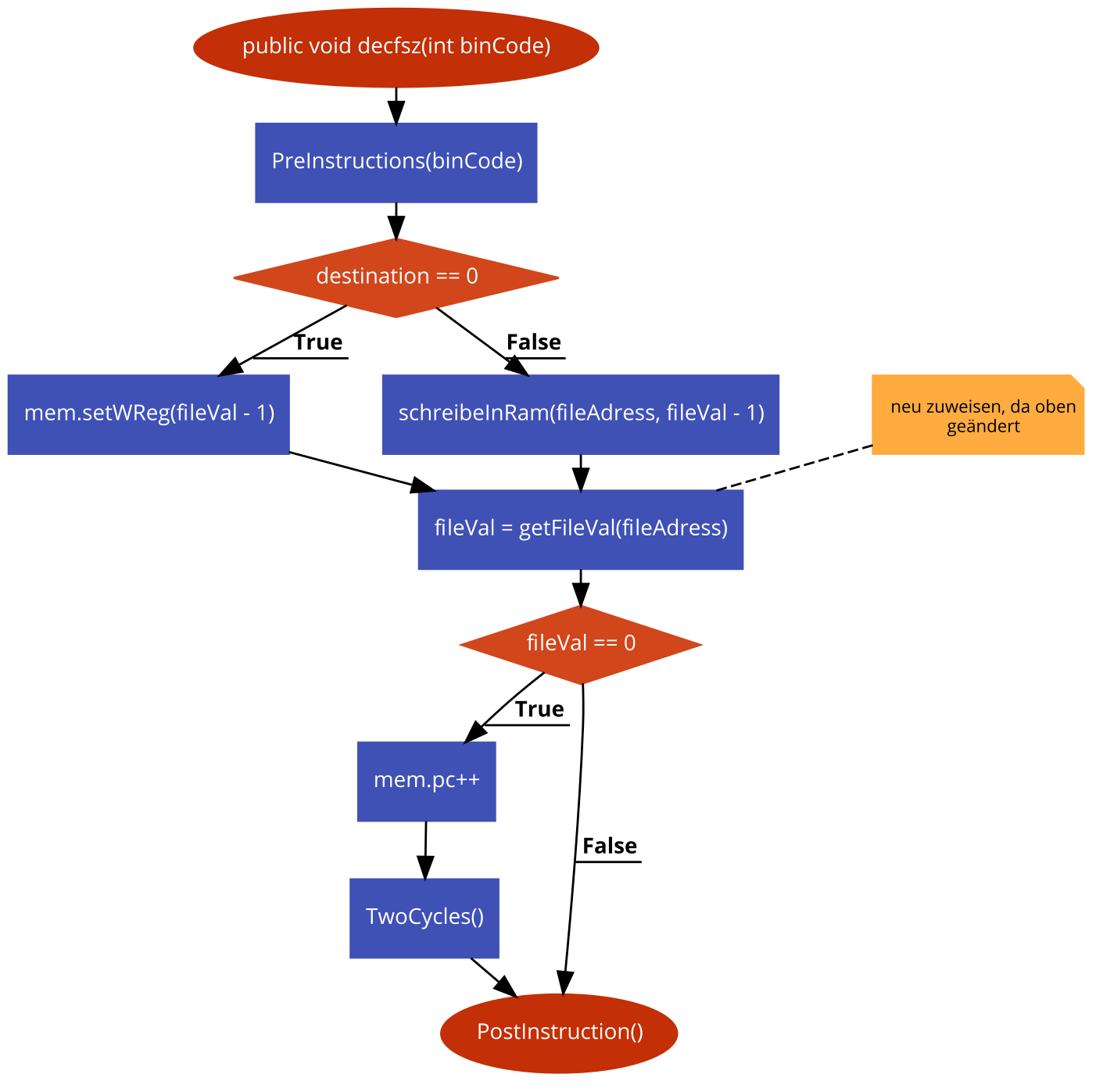


Abbildung 16: PAP DECFSZ

**Implementation**:

Das Zielbit der Operation wird überprüft. Ist das Zielbit 0, so wird der Wert aus f - 1 in das W-Register kopiert. Ist das Zielbit 1, so wird der Wert aus f – 1 an die Zieladresse kopiert.

Der Wert an der neuen Adresse wird gespeichert, ist dieser Wert = 0 wird der PC erhöht und die Instruktionen für zwei Zyklen ausgeführt.

public void decfsz(int binCode)

{

PreInstructions(binCode);

if (destination == 0)

{

mem.setWReg(fileVal - 1);

}

else

{

schreibeInRam(fileAdress, fileVal - 1);

}

//neu zuweisen, da oben geändert

fileVal = getFileVal(fileAdress);

if (fileVal == 0)

{

mem.pc++;

TwoCycles();

}

PostInstruction();

}

## C:\Users\Chris Todt\AppData\Local\Microsoft\Windows\INetCache\Content.Word\XORLW.PNGXORLW (Exclusive OR Literal with W)

Abbildung 17: XORLW - Auszug PIC

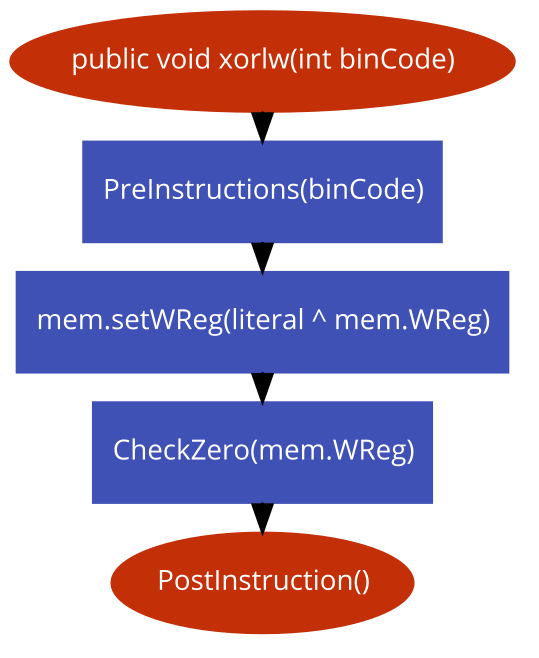


Abbildung 18: PAP XORLW

**Implementation**:

Das W-Register wird mit dem Literal aus dem Binärcode exlusiv verodert und ins W-Register geschrieben.

public void xorlw(int binCode)

{

PreInstructions(binCode);

mem.setWReg(literal ^ mem.WReg);

CheckZero(mem.WReg);

PostInstruction();

}

## Interruptfunktion

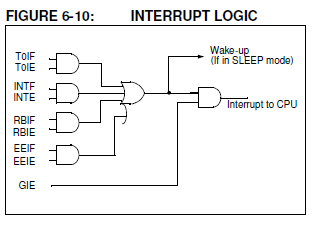
Die I/O Interrupts werden in einem separaten Thread abgefragt. Wird in diesem ein Interrupt festgestellt, so wird ein Bool auf true gesetzt. Dieser wird, zusammen mit den anderen Interrupts, vor jeder Befehlsausführung abgebrüft.

Abbildung 19: Interrupt Logik   
–  
Auszug PIC

Im der nebenstehenden Abbildung 18 kann die Interrupt Logik entnommen werden.

Ist das jeweilige Interrupt-enable Bit gesetzt und ein Interrupt wird wahrgenommen, so wird der Microcontroller aus dem Sleep-Mode aufgeweckt. Nur, wenn auch das GIE (GlobalInterruptEnable) Bit gesetzt wird der Interrupt bis zur CPU durchgereicht:

Syntax: [label] BTFSC f,b

Operanden: 0 ≤ f ≤ 127

0 ≤ b ≤ 7

Befehl : Wenn bit b in f = 0 🡪 überspringe nächsten Befehl

Beeinflusste Flags: ~

Beschreibung

# Programmiersprache

Als Programmiersprache bot sich C# an, da für diese bereits eine Schulung seitens des Südwestrundfunks geboten wurde und durch eine Windows-Forms-Anwendung auf einfache Art und Weise eine GUI entwickelt werden kann.

# Fazit

Das in der Vorlesung Software Engineering I & II gelernte Entwurfsmuster Model-View-Controller konnte, mehr oder weniger, erfolgreich angewendet werden und hat zur ersten Idee zum Aufbau des Programmes mitgewirkt. Durch die Implementierung der Befehle in C# mussten diese ausführlich studiert werden, die Abläufe in einem Microcontroller wurden somit verständlicher.

Trotz anfänglicher Startschwierigkeiten, wie zum Beispiel der Implementierung einiger GUI Komponenten (DataGridView…) oder die Darstellung des internen Speichers, konnte letztendlich erfolgreich ein funktionsfähiger Simulator erstellt werden.

Müsste ein solches Projekt erneut durchgeführt werden, so würde Paul es vorziehen alleine zu arbeiten. Will halt keine coolen Teamkameraden. **¯\\_(ツ)\_/¯**

Die Wahl der Programmiersprache bereuen wir nicht, da C# mit Visual Studio durch die WindowsForms ein einzigartiges Programmiererlebnis für Juniorprogrammierer bietet.

Des Weiteren würden wir uns vorher überlegen, wie wir das Programm am besten strukturieren sollen und welche Klassen gebraucht werden – anstatt einfach drauf los zu programmieren und immer wieder alles zu refactorn.

# Literaturverzeichnis

1. Quelle: <https://de.wikipedia.org/wiki/Simulation> aufgerufen am 29.05.2017 [↑](#footnote-ref-1)