DHBW
Duale Hochschule
Baden-Württemberg
Karlsruhe

Dokumentation zum Pic Simulator im Rahmen der Vorlesung Systemnahe Programmierung 2

30. April 2021

von

Dominik Klein und Robin Weisenburger

Inhaltsverzeichnis

Abbildungsverzeichnis

Listingverzeichnis

1 Einleitung

In der Einleitung wird die Aufgabenstellung erläutert. Außerdem werden auf die Vor-und Nachteile eines Simulators eingegangen und die GUI Oberfläche erklärt.

1.1 Aufgabenstellung

Im Rahmen der Vorlesung Systemnahe Programmierung 2 soll ein Simulatorprogramm für den Microcontroller *16F84* programmiert werden. Die Programmierung dient zum tieferen Verständnis der Funktionsweise eines Microcontrollers. Ziel ist es, die Arbeitsweise eines Microcontrollers in einer beliebigen Programmiersprache nachzubilden. Zur Realisierung des Simulators ist das Wissen aus vergangenen Semestern erforderlich, insbesondere aus der Vorlesung Systemnahe Programmierung 1.

1.2 Arbeitsweise eines Simulators

Mithilfe eines Simulators kann ein reales System betrachtet werden. Im realen System ist es nicht immer möglich, eine genau Beobachtung des Verhaltens eines Systems zu erlangen. Mit einem Simulator kann Abhilfe geschaffen und die Nachvollziehbarkeit eines System erreicht werden.

1.3 Vor- und Nachteile eines Simulators

Zu den Vorteilen eines Simulators gehört die Flexibilität in der Programmierung und Gestaltung. Außerdem wird mithilfe eines Simulators eventueller Schaden am realen System vermieden. Dem gegenüber steht die umfangreiche und zeitintensive Entwicklung des

Simulators. Trotz eines hohen Maßes an Aufwand lässt sich die Realität nicht vollständig auf eine Simulation abbilden.

1.4 GUI Oberfläche

Abbildung 1.1 zeigt die Programmoberfläche des PicSims nach dem Laden eines Programms. Zum Laden eines Programms kann entweder der *Load File* verwendet werden oder die vorgegebenen Test Dateien direkt über das Dropdown-Menü und den *Load Instant* Button in den PicSim geladen werden. Vor dem Start des geladenen Programms können Änderungen im Fileregister, in den Spezialregistern und in den Ports A und B vorgenommen werden. Dazu sind die entsprechenden Zellen anzuklicken und der Wert einzutragen. Außerdem können Breakpoints gesetzt werden. Dazu ist der Kasten, welcher sich links neben des Programm Zeilen befindet anzuklicken. Ein roter Punkt signalisiert den Breakpoint. Die Auswahl der Quarzfrequenz ist ebenfalls möglich. Die Verarbeitung der geladenen Datei beginnt mit dem Klick auf Start. Der Start Button startet den gesamten Durchlauf des Programms, während durch den Next Button lediglich ein Befehl abgearbeitet wird. Am Ende des Programms kann die Simulation mit dem Stop beendet werden. Ein Reset der Laufzeit sowie der Register und ein gesamter Reset sind ebenfalls möglich. Dabei wird durch den Klick auf den Reset Button ein Software Reset und beim Klick auf den Hard Reset Button ein Power-On Reset ausgeführt. Alle Anzeigen mit Ausnahme des Stacks sind im Hexadezimalsystem dargestellt. Der Stack wird in Binär dargestellt. Bei Veränderungen in den Registern ist darauf zu achten, dass die Eingabe in Hexadezimal getätigt wird. Die Doku sowie Informationen über den PicSim können über das Dropdown-Menü in der Menü Leiste aufgerufen werden.

Abbildung 1.1: PicSim Übersicht

2 Realisierung

In diesem Kapitel werden neben dem Konzept ebenfalls der Aufbau sowie die Gründe für die Wahl der Programmiersprache erläutert. Außerdem wird zu jedem Befehl der entsprechende Programmablaufplan vorgestellt. Zu ausgewählten Befehlen wird die Erklärung mithilfe des Codes vertieft. Am Ende dieses Kapitels werden die Funktionsweise der Breakpoints sowie die Implementierung der Interrupts dargestellt.

2.1 Konzept

Das Grundkonzept des programmierten Simulators basiert auf dem Frontend-Backend Prinzip. Das Frontend nimmt die Eingaben des Users sowie des Pfads zum Programm entgegen und leitet diese Informationen an das Backend weiter. Im Backend finden das Einlesen der Daten sowie die Decodierung und Verarbeitung der Befehle statt. Nach jeder Abarbeitung eines Befehls im Backend werden die Ergebnisse an das Frontend übermittelt und in der GUI dargestellt.

2.2 Aufbau

Bei der Entwicklung des Simulators wurde auf Multi-Threading verzichtet. Daher wird während der Laufzeit zwischen dem Frontend und dem Backend nach jeder Abarbeitung eines Befehls hin und her gewechselt. Durch die Wahl dieses grundlegenden Aufbaus, konnte beispielsweise der Next Button, welcher beim Klick jeweils einen Befehl abarbeitet, deutlich einfacher und komfortabler implementiert werden.

2.3 Programmiersprache

Als Programmiersprache wurde C++ gewählt. Für die Erstellung der grafischen Oberfläche ist die Wahl auf Qt gefallen. Primär lässt sich die Entscheidung mit dem bereits vorhandenen Wissen des Teams begründen. Da mit C++ eine tiefgründige Programmierung, teilweise sogar auf Bitebene ermöglicht, bietet diese Programmiersprache optimale Voraussetzungen für die Programmierung eines Simulators. Da in Zusammenhang mit C++ das Framework Qt gängig bei der Entwicklung von grafischen Oberflächen ist, wurde dies ausgewählt. Die nutzbare Objektorientierung in C++ ermöglicht es, die einzelnen Komponenten des PicSims optimal zu implementieren.

2.4 Beschreibung der Funktionen

In diesem Abschnitt werden die implementierten Befehle mithilfe eines Programmablaufplans dargestellt und kurz erläutert. Die Befehle: BTFSS, BTFSC, CALL, MOVF, RRF, SUBWF, DECFSZ, XORLW werden darüber hinaus mithilfe des implementierten Codes näher beleuchtet. Abbildung 2.1 zeigt die zu den folgenden Programmablaufplänen zugehörige Legende.

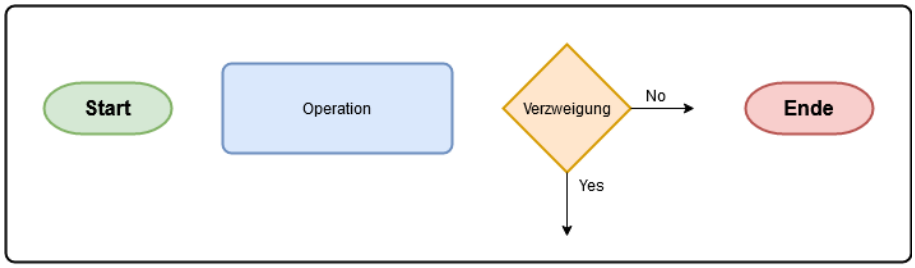

Abbildung 2.1: Legende zu den Programmablaufplänen der Befehle

2.4.1 ADDLW

Der Befehl ADDLW addiert den Inhalt des W-Registers auf ein 8-Bit Literal und speichert das Ergebnis zurück in das W-Register. Dabei werden sowohl Zeroflag als auch Carry und DigitCarry beeinflusst. Dieser Befehl entspricht einem Zyklus.

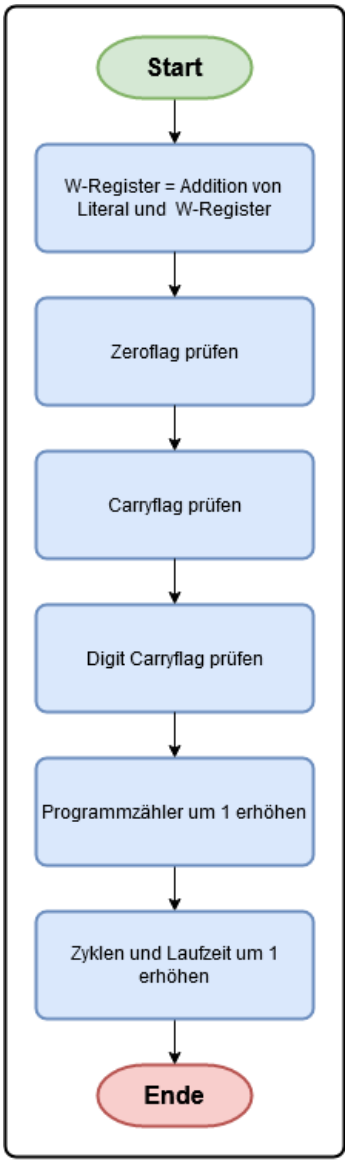

Abbildung 2.2: Programmablaufplan zum Befehl ADDLW

2.4.2 ADDWF

Der Befehl ADDWF addiert den Inhalt des W-Registers auf ein Fileregister. Das Ergebnis wird in Abhängigkeit vom Destination-Bit entweder im Fileregister oder im W-Register gespeichert. Bei dieser Operation werden Zeroflag, Carry und DigitCarry beeinflusst. ADDWF entspricht einem Zyklus.

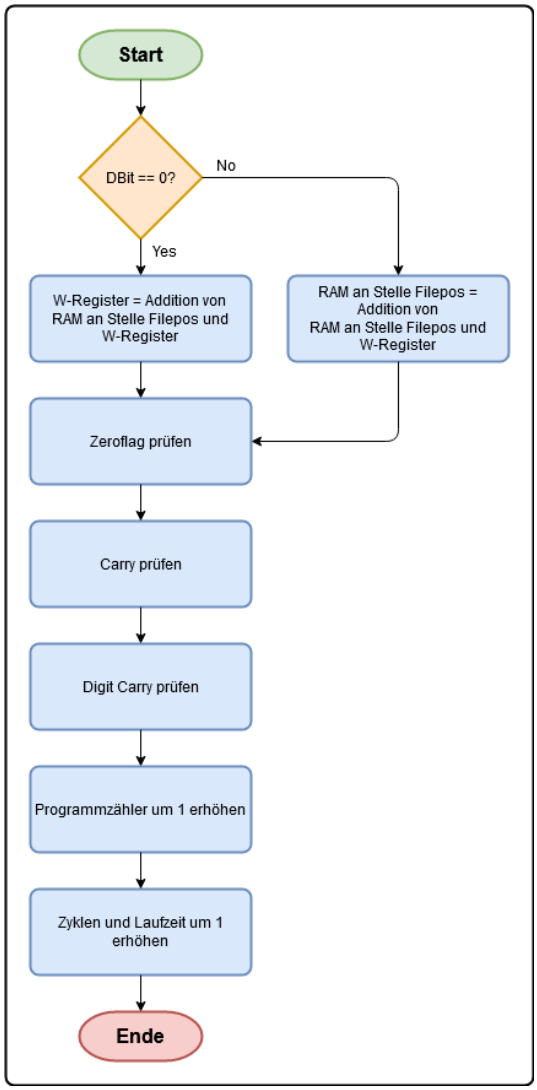

Abbildung 2.3: Programmablaufplan zum Befehl ADDWF

2.4.3 ANDLW

Dieser Befehl führt eine UND Verknüpfung des W-Registers mit einem 8-Bit Literal durch. Das Ergebnis wird in das W-Register gespeichert. Dabei wird das Zeroflag beeinflusst. ANDLW entspricht einem Zyklus.

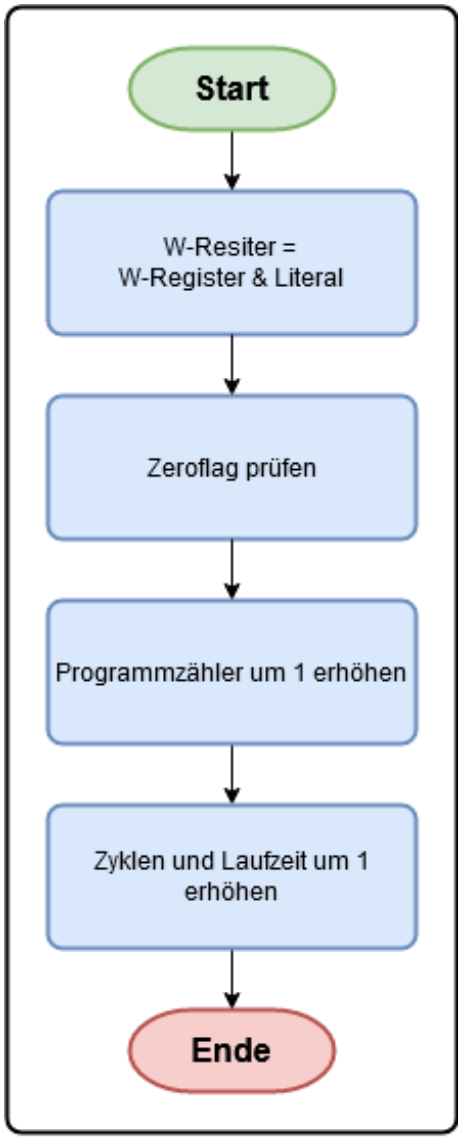

Abbildung 2.4: Programmablaufplan zum Befehl ANDLW

2.4.4 ANDWF

Dieser Befehl führt eine UND Verknüpfung des W-Registers mit einem Fileregister durch. Das Ergebnis wird in Abhängigkeit vom Destination-Bit entweder im Fileregister oder im W-Register gespeichert. Dabei wird das Zeroflag beeinflusst. ANDWF entspricht einem Zyklus.

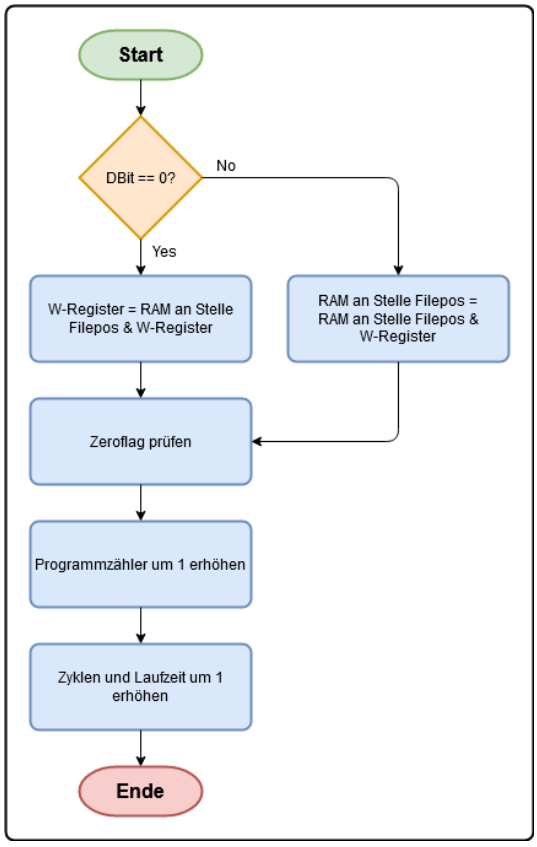

Abbildung 2.5: Programmablaufplan zum Befehl ANDWF

2.4.5 BCF

Dieser Befehl löscht das entsprechende Bit im Fileregister. BCF entspricht einem Zyklus.

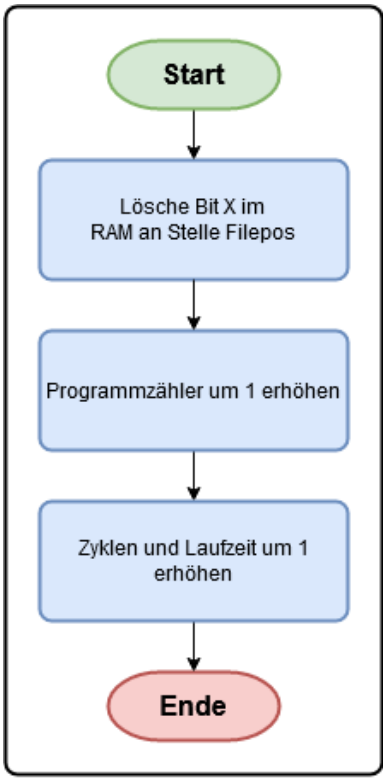

Abbildung 2.6: Programmablaufplan zum Befehl BCF

2.4.6 BSF

Dieser Befehl setzt das entsprechende Bit im Fileregister. BSF entspricht einem Zyklus.

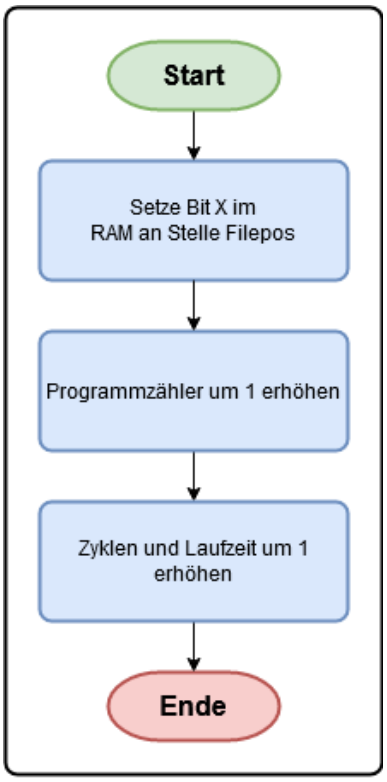

Abbildung 2.7: Programmablaufplan zum Befehl BSF

2.4.7 BTFSC

Dieser Befehl prüft ein Bit im Fileregister. Ist dieses Bit 0, wird der nächste Befehl mithilfe eines Nops übersprungen. Falls das Bit 1 ist, wird der nachfolgende Befehl ausgeführt. Das Überspringen entspricht zwei Zyklen. Falls nicht gesprungen wird, entspricht dies nur einem Zyklus. In Listing A.1 wird der entsprechende Code dargestellt.

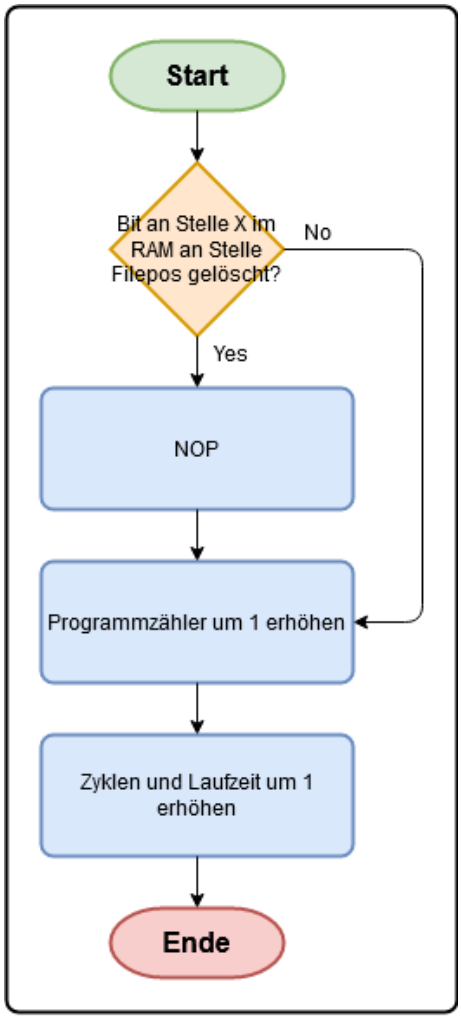

Abbildung 2.8: Programmablaufplan zum Befehl BTFSC

2.4.8 BTFSS

Dieser Befehl prüft ein Bit im Fileregister. Ist dieses Bit 1, wird der nächste Befehl mithilfe eines Nops übersprungen. Falls das Bit 0 ist, wird der nachfolgende Befehl ausgeführt. Das Überspringen entspricht zwei Zyklen. Falls nicht gesprungen wird, entspricht dies nur einem Zyklus. In Listing A.2 wird der entsprechende Code dargestellt.

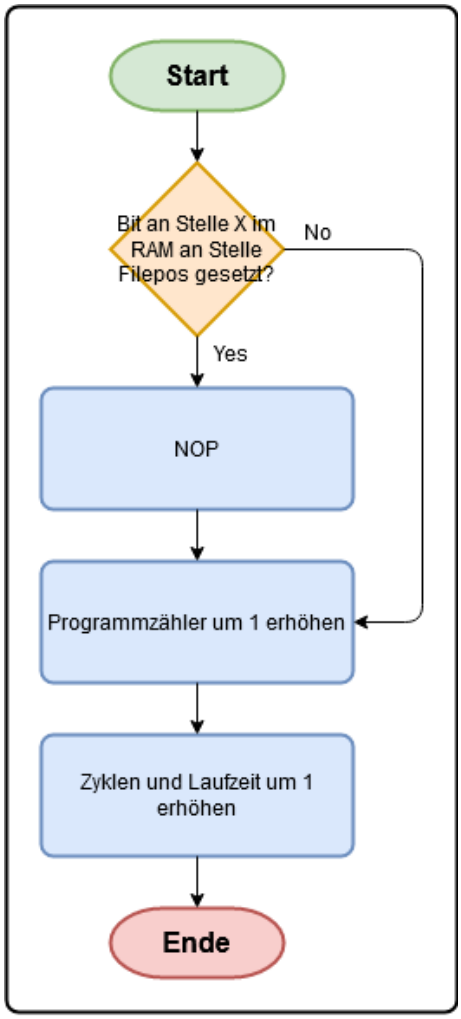

Abbildung 2.9: Programmablaufplan zum Befehl BTFSS

2.4.9 CALL

Dieser Befehl legt den aktuellen Programmzähler auf den Stack. Dies wird zum späteren Rücksprung benötigt. Aus dem 11 Bit Literal des CALL Befehls sowie dem Bit 4 und 3 von PCLATH wird die Sprungadresse gebildet. Diese wird dem Programmcounter zugewiesen. Der CALL Befehl entspricht 2 Zyklen. In Listing A.3 wird der entsprechende Code dargestellt.

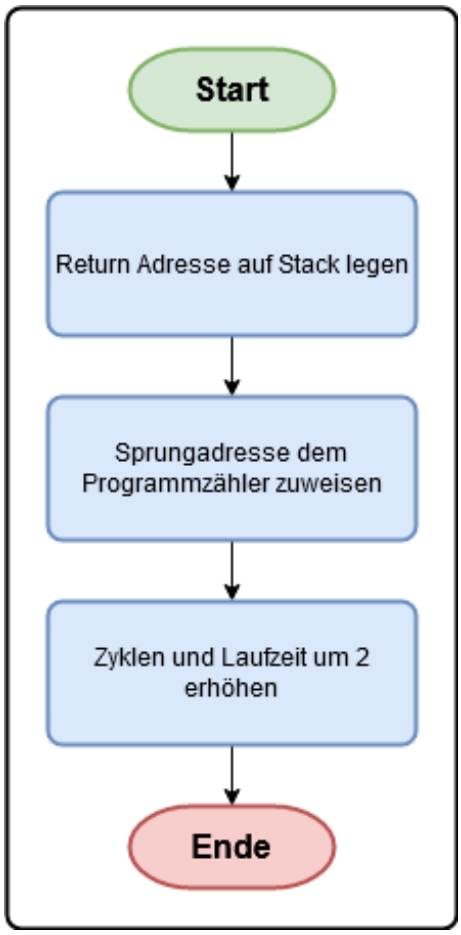

Abbildung 2.10: Programmablaufplan zum Befehl CALL

2.4.10 CLRF

Dieser Befehl löscht das entsprechende Fileregister. Dabei wird das Zeroflag beeinflusst. CLRF entspricht einem Zyklus.

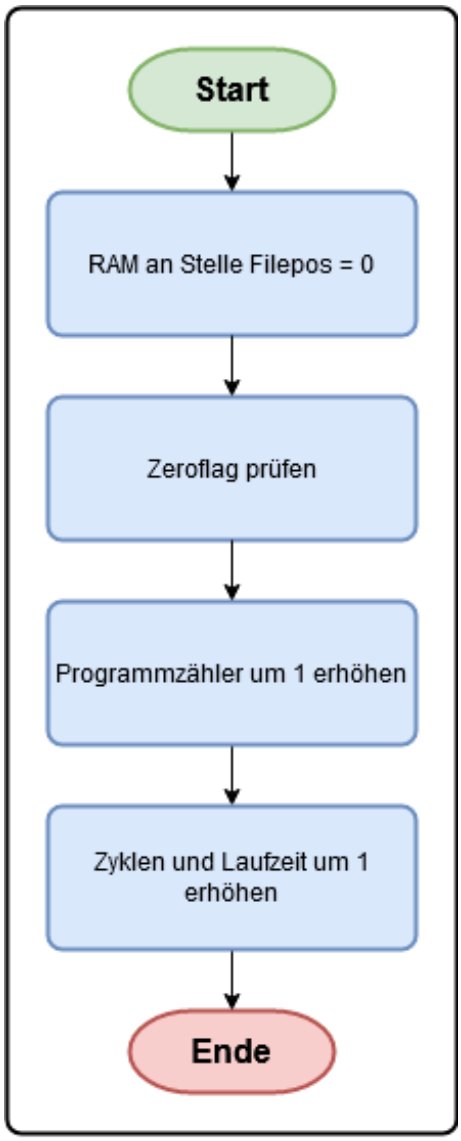

Abbildung 2.11: Programmablaufplan zum Befehl CLRF

2.4.11 CLRW

Dieser Befehl löscht das W-Register. Dabei wird das Zeroflag beeinflusst. CLRW entspricht einem Zyklus.

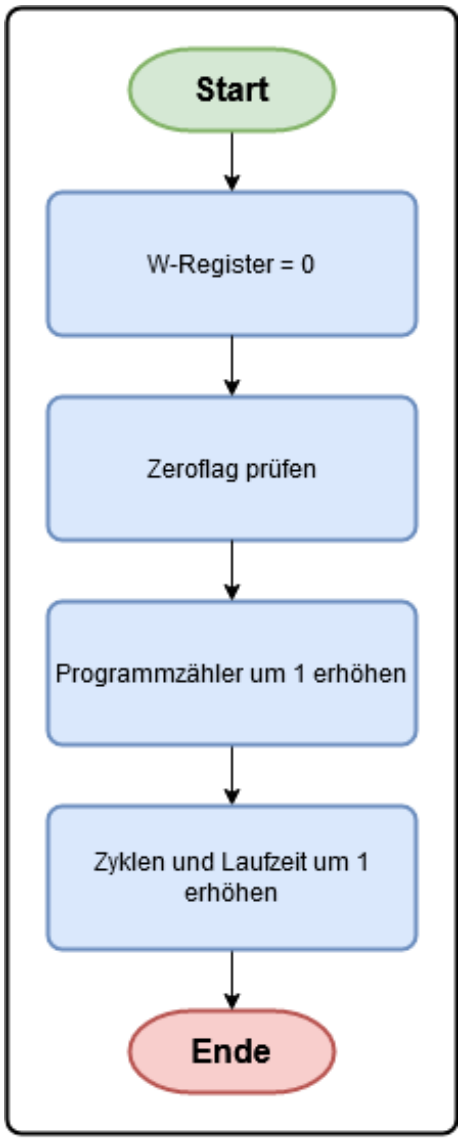

Abbildung 2.12: Programmablaufplan zum Befehl CLRW

2.4.12 COMF

Dieser Befehl bildet das Komplement eines Fileregister. In Abhängigkeit des Destination Bits wird das Ergebnis entweder im W-Register gespeichert oder zurück in das Fileregister geschrieben. Dabei wird das Zeroflag beeinflusst. COMF entspricht einem Zyklus.

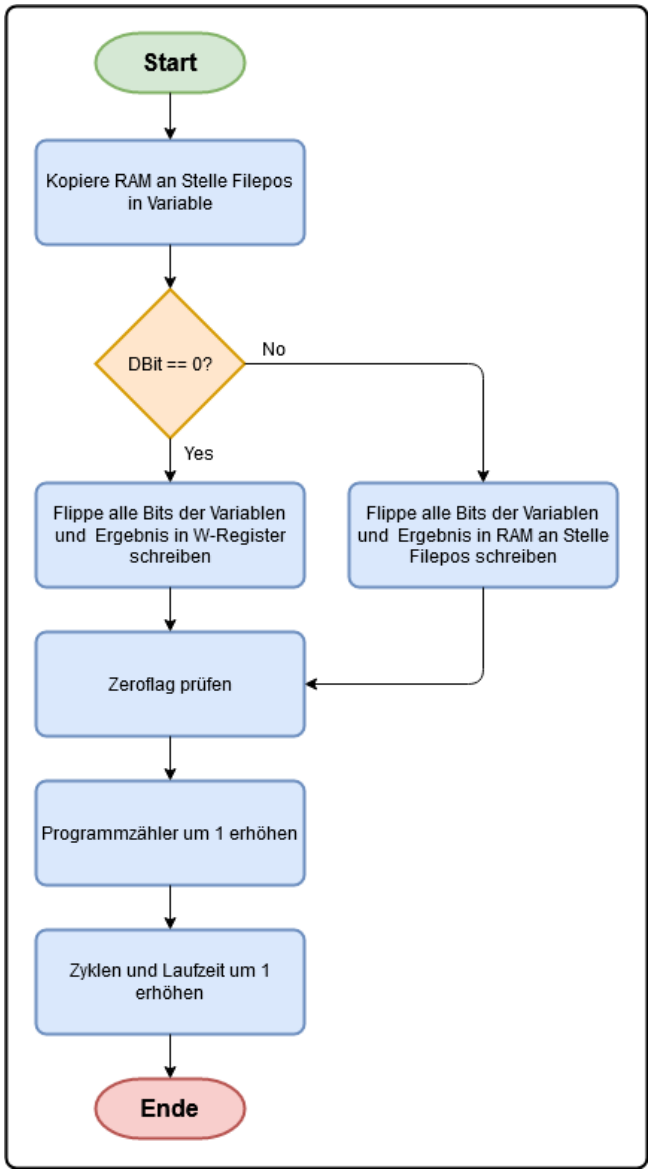

Abbildung 2.13: Programmablaufplan zum Befehl COMF

2.4.13 DECF

Dieser Befehl dekrementiert den Inhalt eines Fileregisters. Dabei wird das Zeroflag beeinflusst. Das Ergebnis der Dekrementierung wird in Abhängigkeit vom Destination-Bit entweder im W-Register oder im Fileregister gespeichert. DECF entspricht einem Zyklus.

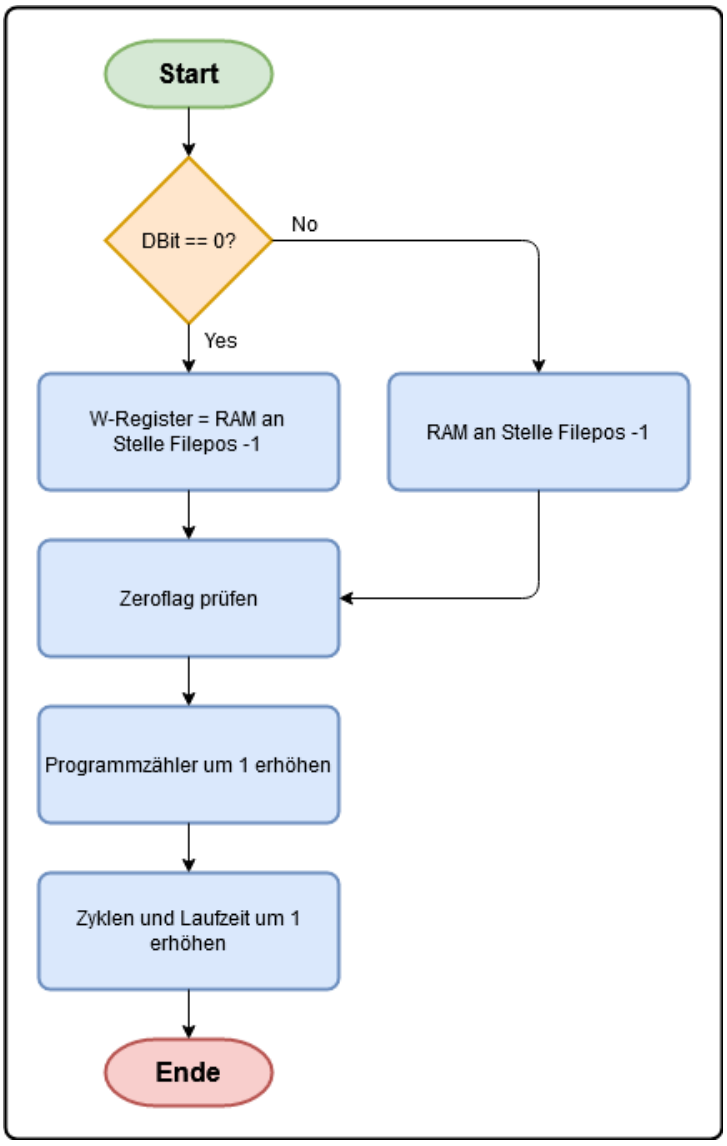

Abbildung 2.14: Programmablaufplan zum Befehl DECF

2.4.14 DECFSZ

Dieser Befehl dekrementiert den Inhalt eines Fileregisters. Das Ergebnis der Dekrementierung wird in Abhängigkeit vom Destination-Bit entweder im W-Register oder im Fileregister gespeichert. Falls das Ergebnis der Dekrementierung 0 ist, wird ein NOP ausgeführt und der nächste Befehl übersprungen. Ist das Ergebnis nicht 0, wird der nachfolgende Befehl ausgeführt. Wird ein Befehl übersrungen, entspricht dies zwei Zyklen. Falls das Ergebnis nicht 0 ist, entspricht dies einem Zyklus. In Listing A.4 wird der entsprechende Code dargestellt.

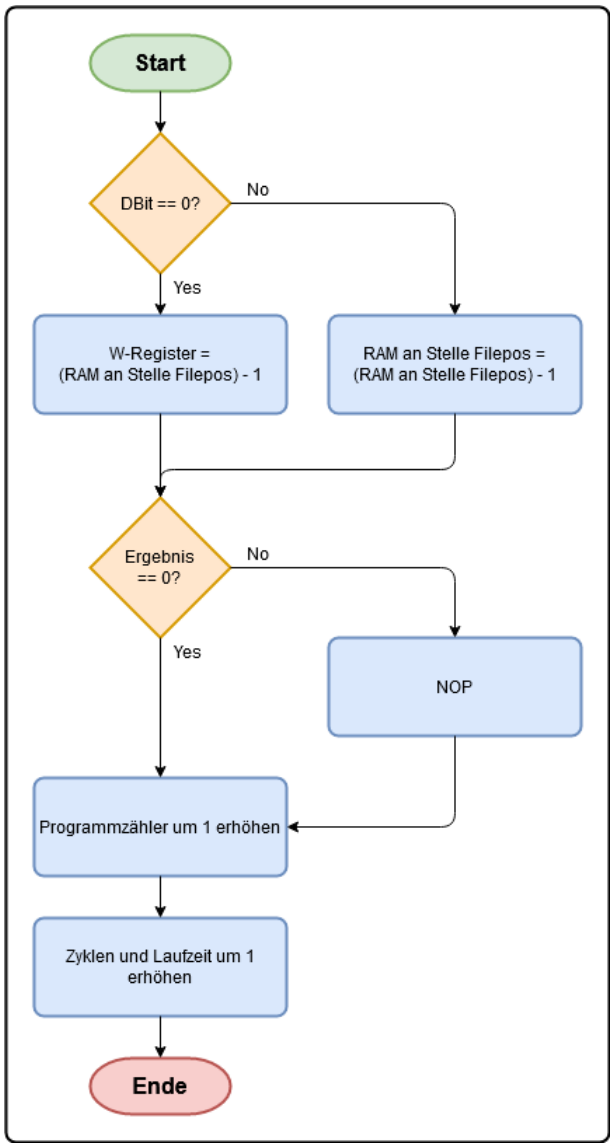

Abbildung 2.15: Programmablaufplan zum Befehl DECFSZ

2.4.15 GOTO

Dieser Befehl bildet aus dem 11 Bit Literal sowie dem Bit 4 und 3 von PCLATH die Sprungadresse auf die gesprungen werden soll. Diese wird dem Programmcounter zugewiesen. Der GOTO Befehl entspricht 2 Zyklen.

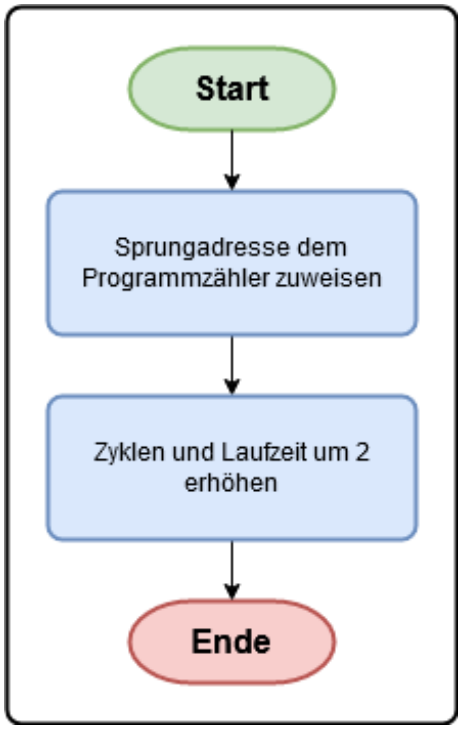

Abbildung 2.16: Programmablaufplan zum Befehl GOTO

2.4.16 INCF

Dieser Befehl inkrementiert den Inhalt eines Fileregisters. Dabei wird das Zeroflag beeinflusst. Das Ergebnis der Inkrementierung wird in Abhängigkeit vom Destination-Bit entweder im W-Register oder im Fileregister gespeichert. INCF entspricht einem Zyklus.

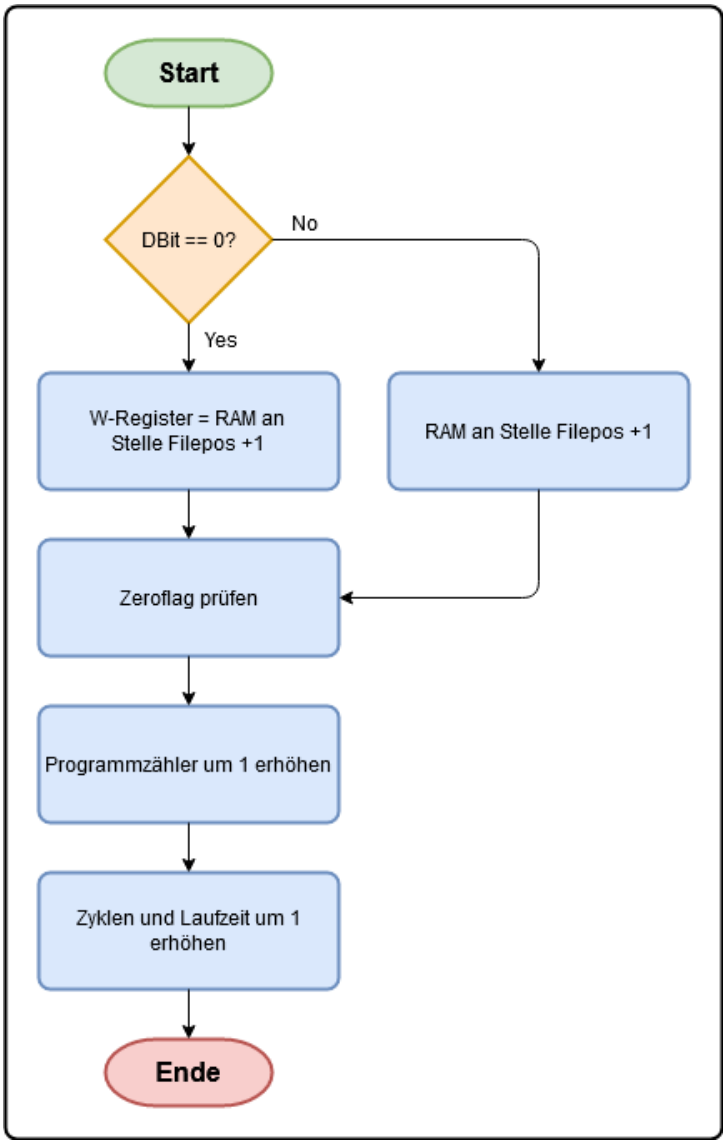

Abbildung 2.17: Programmablaufplan zum Befehl INCF

2.4.17 INCFSZ

Dieser Befehl inkrementiert den Inhalt eines Fileregisters. Das Ergebnis der Inkrementierung wird in Abhängigkeit vom Destination-Bit entweder im W-Register oder im Fileregister gespeichert. Falls das Ergebnis 0 ist, wird ein NOP ausgeführt und der nächste Befehl übersprungen. Ist das Ergebnis nicht 0, wird der nachfolgende Befehl ausgeführt. Wird ein Befehl übersrungen, entspricht dies zwei Zyklen. Falls das Ergebnis nicht 0 ist, entspricht dies einem Zyklus.

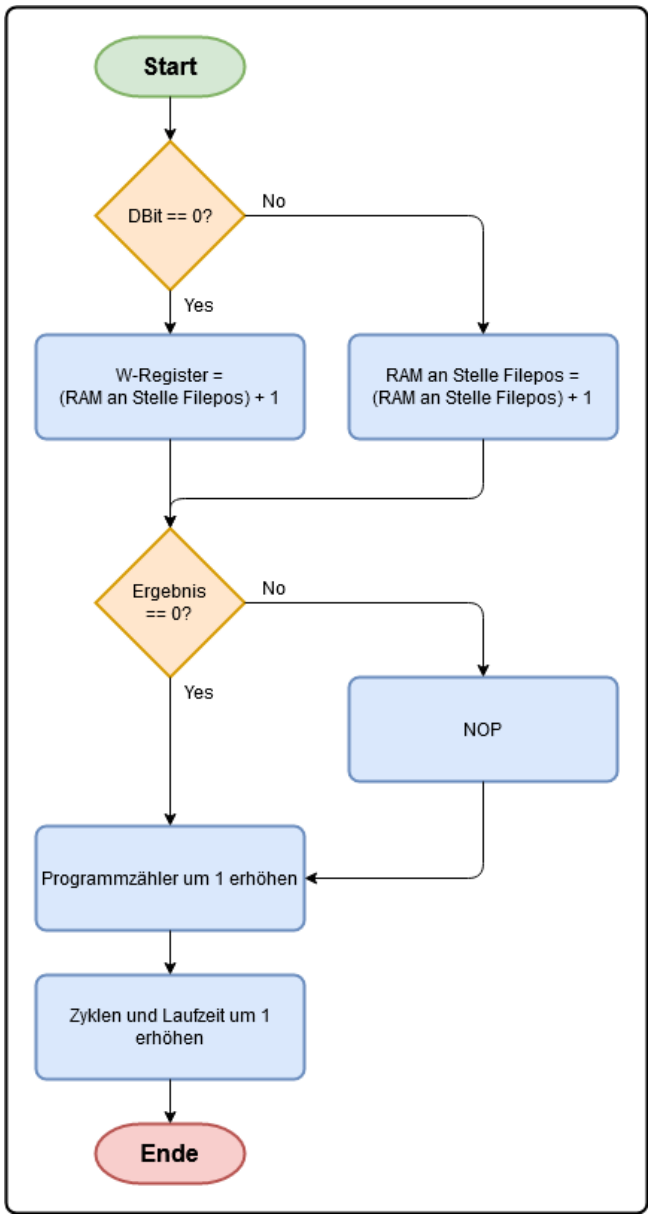

Abbildung 2.18: Programmablaufplan zum Befehl INCFSZ

2.4.18 IORLW

Dieser Befehl bildet das logische inklusiv Oder mit dem W-Register und einem 8-Bit Literal. Das Ergebnis wird in das W-Register geschrieben. IORLW beeinflusst das Zeroflag und entspricht einem Zyklus.

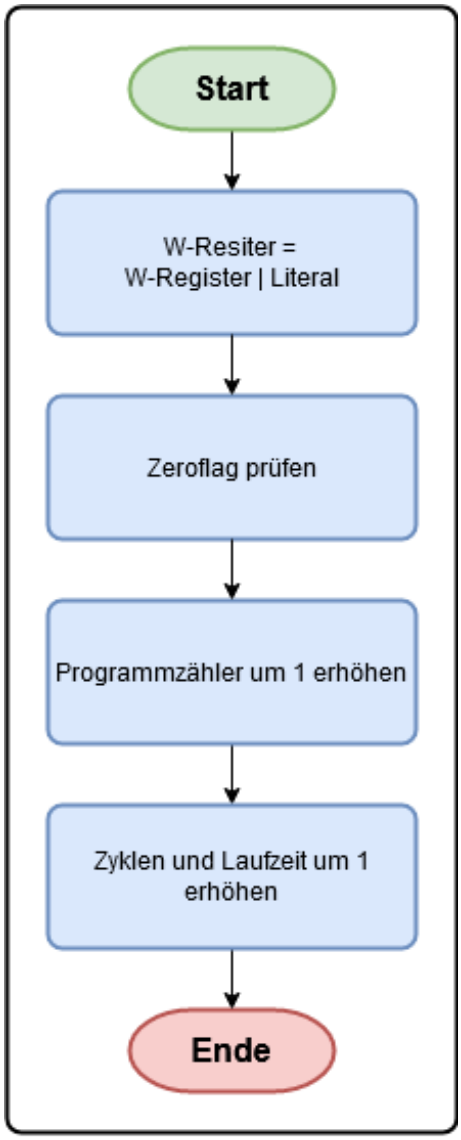

Abbildung 2.19: Programmablaufplan zum Befehl IORLW

2.4.19 IORWF

Dieser Befehl bildet das logische inklusiv Oder mit dem W-Register und Fileregister. Das Ergebnis wird in Abhängigkeit vom Destination-Bit entweder im W-Register oder im Fileregister gespeichert. IORWF beeinflusst das Zeroflag und entspricht einem Zyklus.

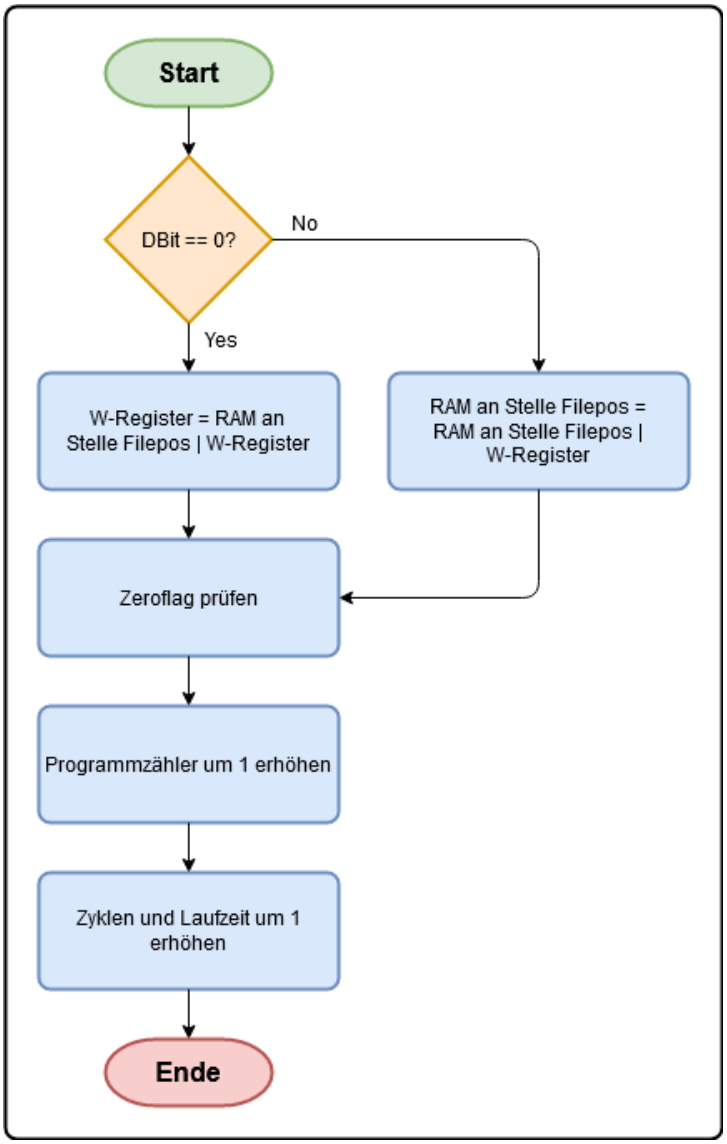

Abbildung 2.20: Programmablaufplan zum Befehl IORWF

2.4.20 MOVF

Dieser Befehl kopiert den Inhalt eines Fileregisters in Abhängigkeit des Destination-Bits entweder in das W-Register oder in das gleiche Fileregister. Da dieser Befehl das Zeroflag beeinflusst, eignet sich MOVF zum überprüfen eines Fileregisters auf 0. MOVF entspricht einem Zyklus. In Listing A.5 wird der entsprechende Code dargestellt.

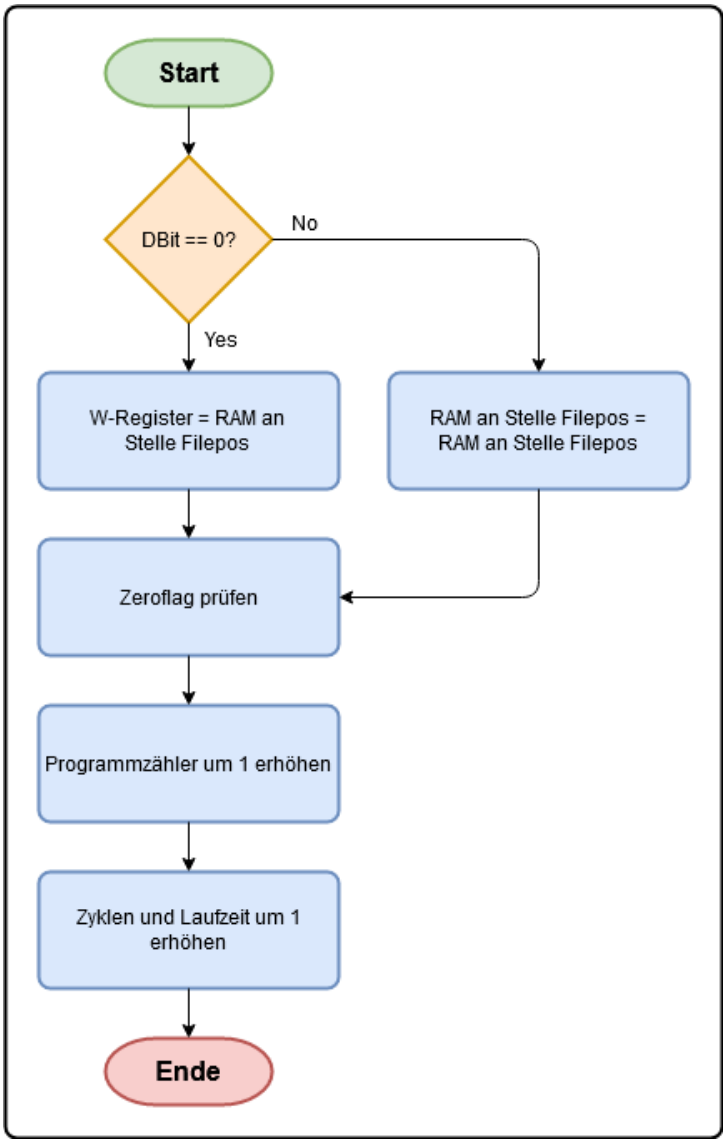

Abbildung 2.21: Programmablaufplan zum Befehl MOVF

2.4.21 MOVLW

Dieser Befehl schreibt das 8-Bit Literal in das W-Register. MOVLW entspricht einem Zyklus.

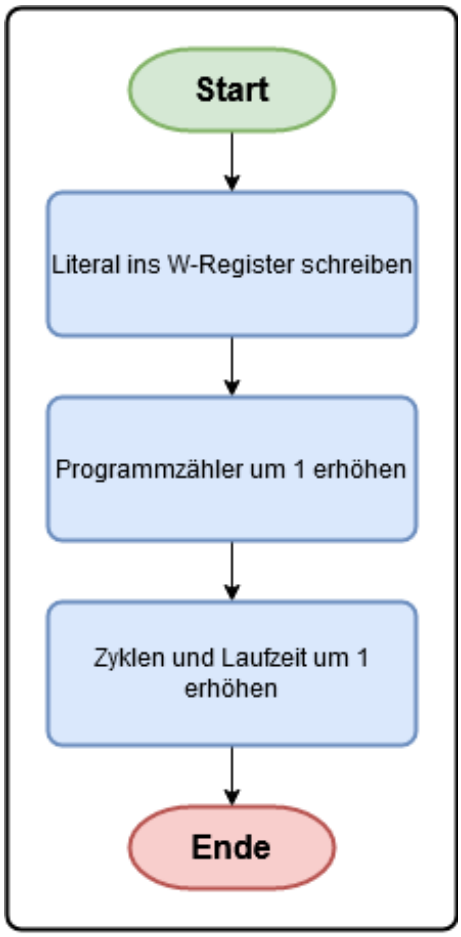

Abbildung 2.22: Programmablaufplan zum Befehl MOVLW

2.4.22 MOVWF

Dieser Befehl kopiert den Inhalt des W-Registers in ein Fileregister. MOVWF enspricht einem Zyklus.

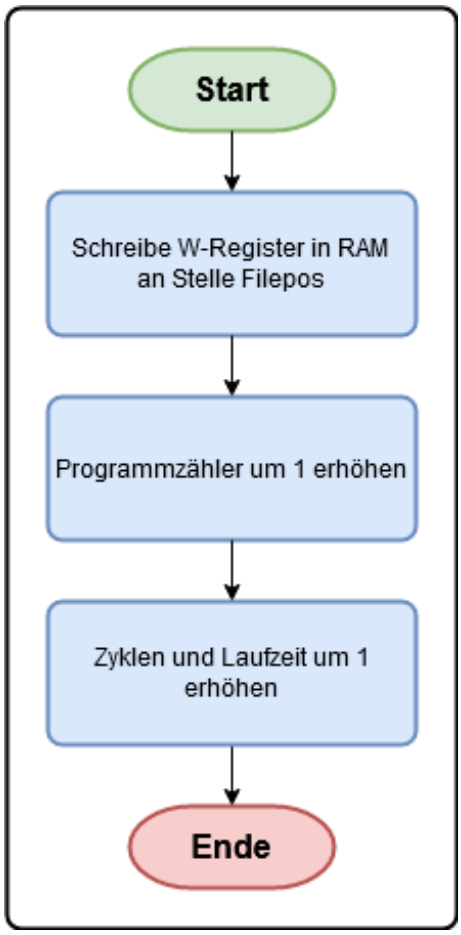

Abbildung 2.23: Programmablaufplan zum Befehl MOVWF

2.4.23 NOP

Dieser Befehl führt keine Operation durch. Jedoch entspricht NOP einem Zyklus. Häufig wird ein NOP als Hilfsbefehl verwendet.

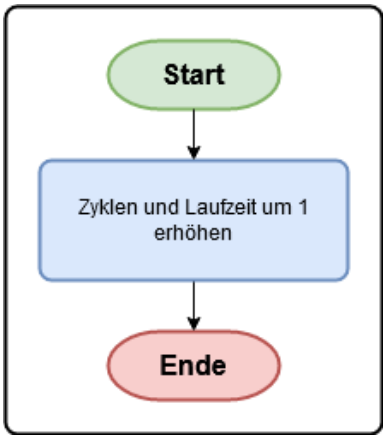

Abbildung 2.24: Programmablaufplan zum Befehl NOP

2.4.24 RETFIE

Dieser Befehl beendet die Interrupt-Service-Routine. Der oberste Stack Eintrag wird vom Stack in den Programmcounter geschrieben. Das Global Interrupt Enable Bit wird zurück auf 1 gesetzt, um weitere Interrupts zu ermöglichen. RETFIE entspricht zwei Zyklen.

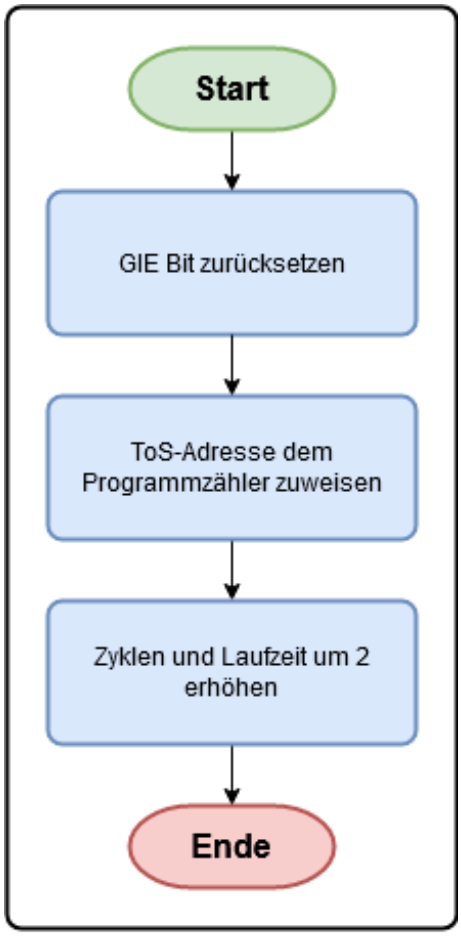

Abbildung 2.25: Programmablaufplan zum Befehl RETFIE

2.4.25 RETLW

Dieser Befehl beendet ein Unterprogramm. Dazu wird der oberste Stack Eintrag in den Programmcounter geschrieben. Außerdem wird das 8-Bit Literal in das W-Register geladen. RETLW entspricht zwei Zyklen.

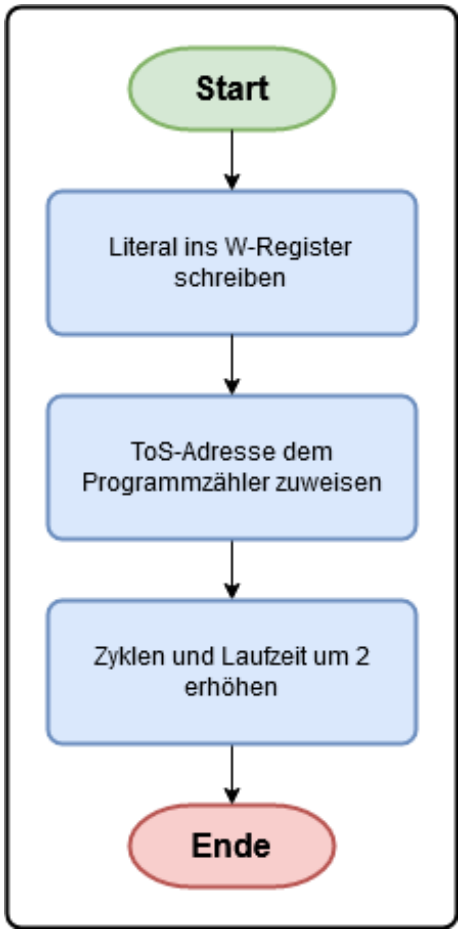

Abbildung 2.26: Programmablaufplan zum Befehl RETLW

2.4.26 RETURN

Dieser Befehl beendet ein Unterprogramm. Dazu wird der oberste Stack Eintrag in den Programmcounter geschrieben. RETURN entspricht zwei Zyklen.

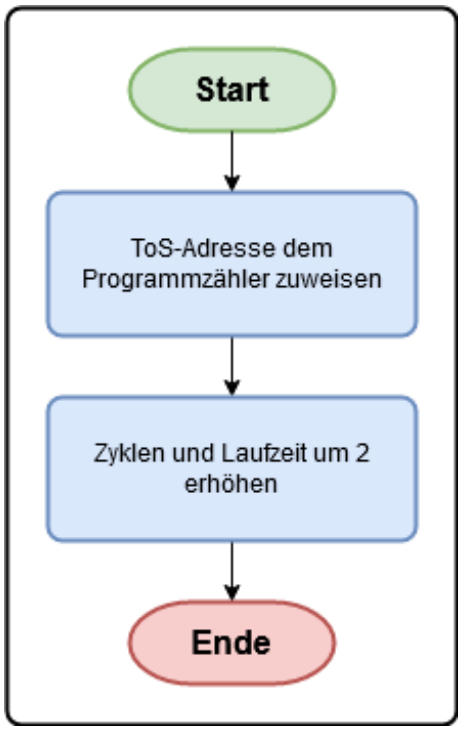

Abbildung 2.27: Programmablaufplan zum Befehl RETURN

2.4.27 RLF

Dieser Befehl führt eine bitweise links Verschiebung des Inhalts eines Fileregisters durch. Dabei wird durch das Carry rotiert. RLF entspricht einem Zyklus.

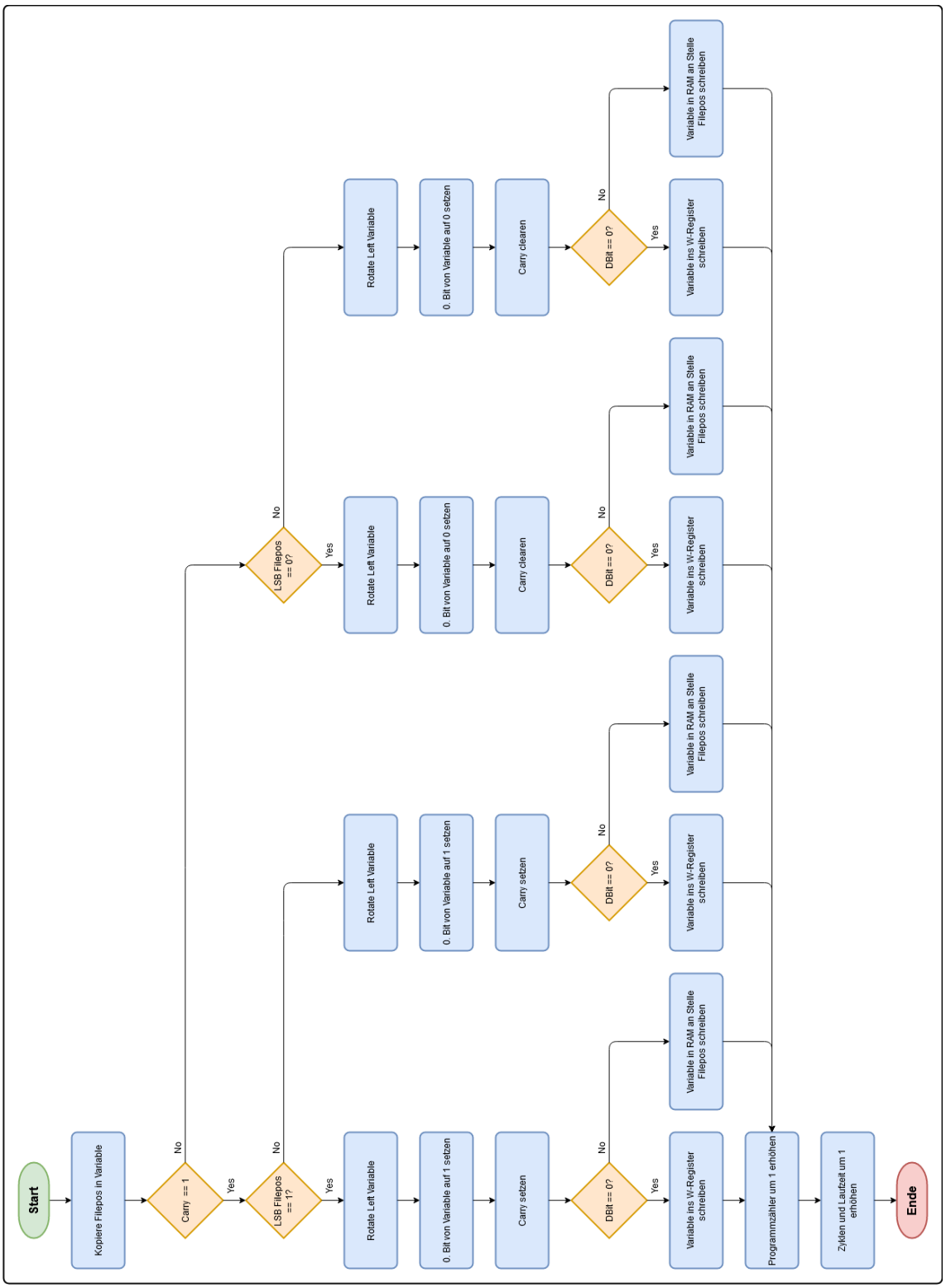

Abbildung 2.28: Programmablaufplan zum Befehl RLF

2.4.28 RRF

Dieser Befehl führt eine bitweise rechts Verschiebung des Inhalts eines Fileregisters durch. Dabei wird durch das Carry rotiert. RRF entspricht einem Zyklus. In Listing A.6 wird der entsprechende Code dargestellt.

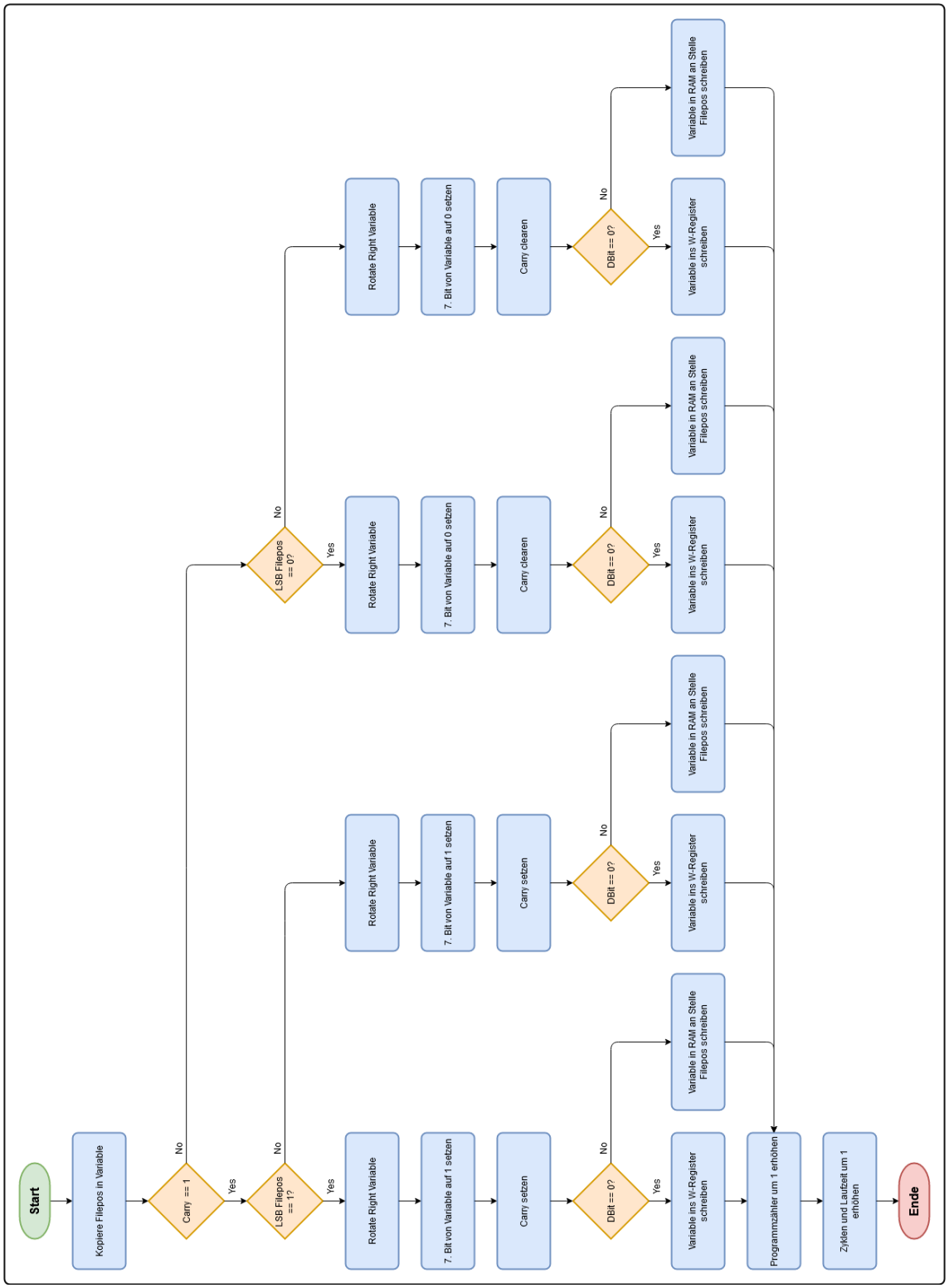

Abbildung 2.29: Programmablaufplan zum Befehl RRF

2.4.29 SUBLW

Dieser Befehl addiert das 2er Komplement des W-Registers auf das Literal. Das Ergebnis wird zurück ins W-Register geschrieben. Dabei werden Carry, Zeroflag und DigitCarry beeinflusst. SUBLW entspricht einem Zyklus.

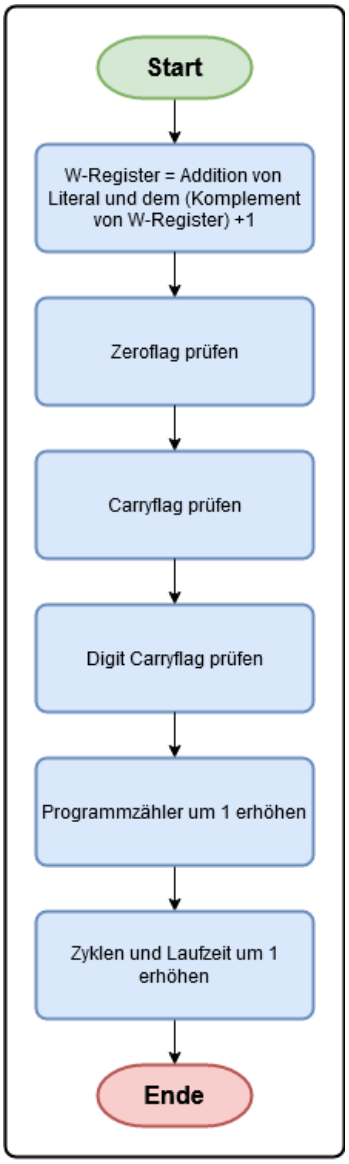

Abbildung 2.30: Programmablaufplan zum Befehl SUBLW

2.4.30 SUBWF

Dieser Befehl addiert das 2er Komplement des W-Registers auf den Inhalt eines Fileregisters. Das Ergebnis wird in Abhängigkeit vom Destiantion-Bit entweder zurück ins W-Register oder in das Fileregister geschrieben. Dabei werden Carry, Zeroflag und Digit-Carry beeinflusst. SUBWF entspricht einem Zyklus. In Listing A.7 wird der entsprechende Code dargestellt.

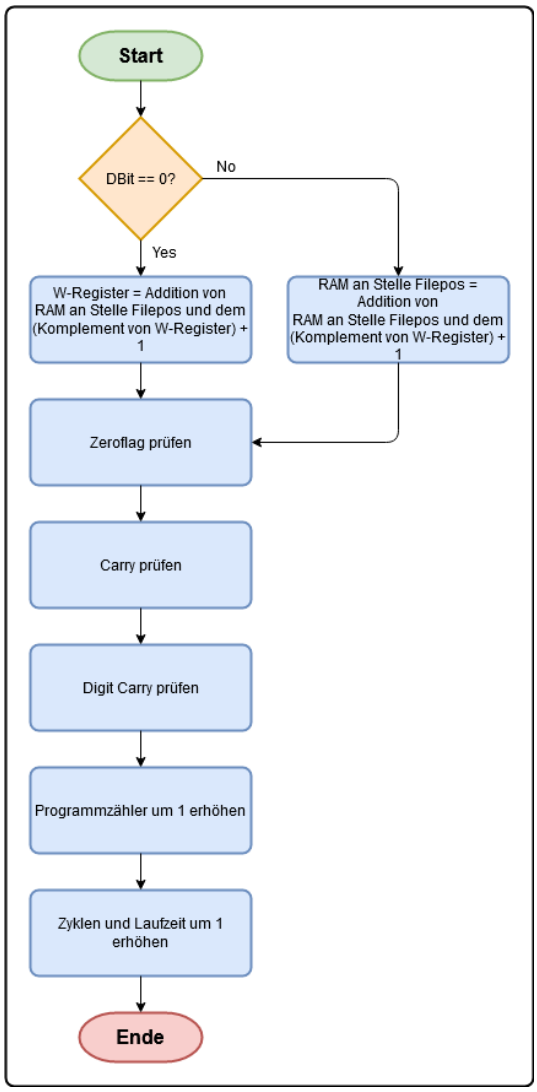

Abbildung 2.31: Programmablaufplan zum Befehl SUBWF

2.4.31 SWAPF

Dieser Befehl vertauscht das Low-Byte mit dem High-Byte eines Fileregisters. Das Ergebnis wird in Abhängigkeit vom Destiantion-Bit entweder zurück ins W-Register oder in das Fileregister geschrieben. SWAPF entspricht einem Zyklus.

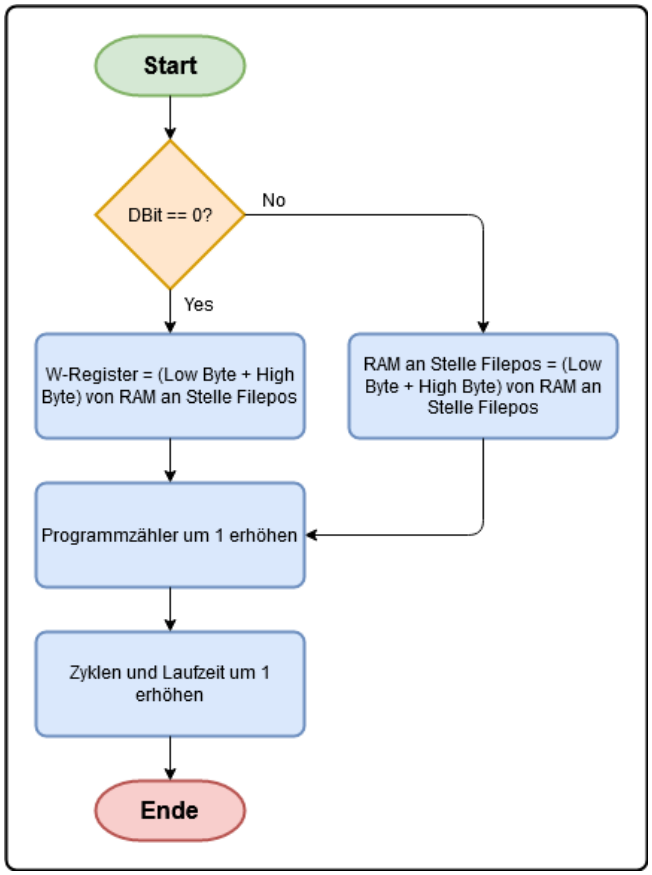

Abbildung 2.32: Programmablaufplan zum Befehl SWAPF

2.4.32 XORLW

Dieser Befehl bildet das logische exklusiv Oder mit dem W-Register und einem 8-Bit Literal. Das Ergebnis wird in das W-Register geschrieben. XORLW beeinflusst das Zeroflag und entspricht einem Zyklus. In Listing A.8 wird der entsprechende Code dargestellt.

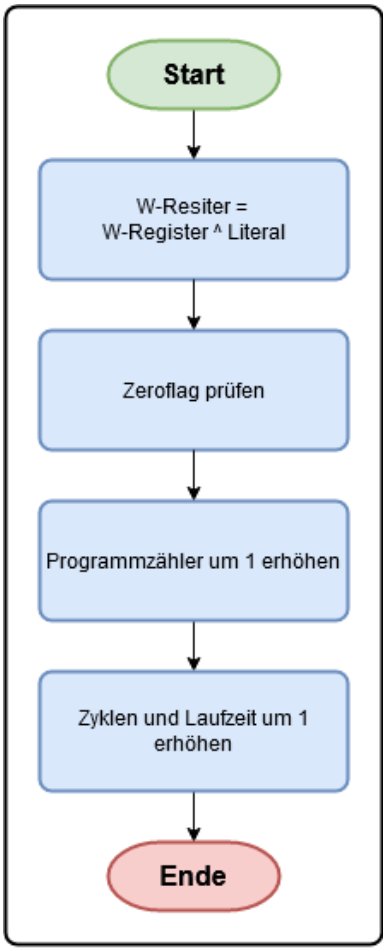

Abbildung 2.33: Programmablaufplan zum Befehl XORLW

2.4.33 XORWF

Dieser Befehl bildet das logische exklusiv Oder mit dem W-Register und Fileregister. Das Ergebnis wird in Abhängigkeit vom Destination-Bit entweder im W-Register oder im Fileregister gespeichert. XORWF beeinflusst das Zeroflag und entspricht einem Zyklus.

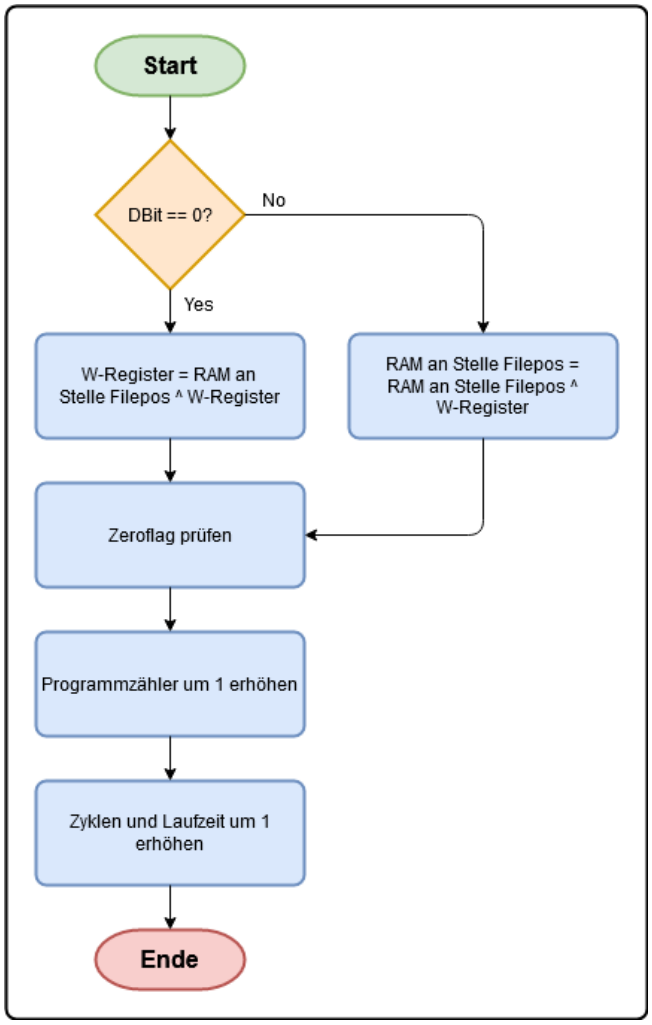

Abbildung 2.34: Programmablaufplan zum Befehl XORWF

2.5 Flags und deren Wirkungsmechanismen

Die Flags werden mithilfe von Funktionen gesetzt oder gelöscht. Die zeroFlag Funktion erwartet einen Wert, falls dieser Wert 0 entspricht, wird das 2. Bit im RAM-Array mit dem Index 3 gesetzt. Umgekehrt wird das Bit auf 0 gesetzt. Das Carry- und das Digit Carryflag werden mithilfe der Addition Funktion gesetzt. Diese Funktion bildet einen Volladdierer ab. Dabei wird das Digit Carry bei einem Übertrag von der 3. auf die 4. Bitstelle und das Carry bei einem Übertrag an der 7. Bitstelle gesetzt. Das Setzen und Löschen der Flags beeinflusst direkt das RAM-Array am Index 3 und soll das Statusregister abbilden. Dieses Register wird dann wiederum aktualisiert an die GUI weitergegeben.

2.6 Implementierung der Interrupts

Bei der Entwicklung wurde auf den Interrupt durch den EEPROM verzichtet, alle weiteren Interrupts sind implementiert. Die Interrupts bilden dabei nicht direkt die Realität ab. Eine Zeile in der LST-Datei entspricht einem Programmdurchlauf im Simulator. Der Simulator prüft bei jedem Durchlauf, ob ein Interrupt ausgelöst wurde. Für jeden Interrupt existiert eine Funktion. Diese werden nacheinander aufgerufen. Wird ein Interrupt erkannt, so werden weitere Interrupts gesperrt, bis das GIE-Bit wieder zurückgesetzt wurde. Anschließend wird an die Interrupt-Routinen-Adresse 4 gesprungen. Der dort stehende Code wird ausgeführt. Je nach Interrupt-Behandlung wird mithilfe von RETFIE an die auf dem Stack liegende Adresse zurückgesprungen. In diesem Schritt wird das GIE wieder auf 1 gesetzt und das Programm läuft weiter.

2.7 Realisierung der Breakpoints

Die LST-Dateien werden in eine Tabelle geladen und angezeigt. Auf der linken Seite wird eine Spalte vorangestellt. Die Spalte lässt sich auf jeder Zeile anklicken und beim Klicken wird ein Breakpoint erstellt. Ein Breakpoint wird mit einem roten Kreis dargestellt und kann durch ein weiteres Klicken darauf wieder entfernt werden. Beim Einlesen der LST-Datei wird ein Array erstellt, das markiert, in welcher Zeile ausführbarer Code steht. Nur in diesen Zeilen lässt sich ein Breakpoint setzen. Sobald das Programm in eine Zeile springt, die mit einem Breakpoint markiert ist, wird die Variable „run“ auf „false“

gesetzt und das Programm stoppt. Mit einem erneuten Start oder Next-Befehl kann dieser Breakpoint übersprungen werden. Dabei wird die Variable „run“ wieder aus „true“ gesetzt. Erreicht das Programm nochmals die Stelle des Breakpoints, so wird es erneut gestoppt.

2.8 Realisierung der TRIS Register

Die TRIS-Register stehen in Zusammenhang mit den Ports A und B. Je nachdem, wie die Richtung auf TRISA und TRISB definiert ist (1 bedeutet Input, 0 bedeutet Output), können die entsprechenden Bits auf PortA beziehungsweise PortB gelesen oder geschrieben werden. In der GUI wird dieser Zustand auch mit einem „i“ für Input und „o“ für Output angezeigt. Dieser Zustand beeinflusst auch Interrupts, die über PortB erfolgen. Die Latchfunktion selbst wurde nicht implementiert.

3 Fazit

Zusammenfassend lässt sich sagen, dass das Projekt Pic-Simulator sehr umfangreich und zeitaufwendig war. Die GUI hat uns anfangs vor eine Herausforderung gestellt. Die Erfahrungen mit dem Umgang einer GUI waren bisher nur in geringem Maße vorhanden. Die Wahl der Programmiersprache C++ hat die Umsetzung, durch Vorkenntnisse, erleichtert. Mit dem Framework von Qt gab es leichte Berührungspunkte im Vorfeld, sodass innerhalb des Teams schnell klar war, dieses Framework zu verwenden.
Zu Beginn der Entwicklung wurde auf eine GUI komplett verzichtet, sodass jeglicher Output über die Konsole ausgegeben wurde. Nachdem die ersten Programme schon liefen, haben wir schnell festgestellt, dass die Wahl unserer Datenstruktur nicht optimal war. Leider haben wir durch die Umstrukturierung des kompletten Programms einiges an Zeit verloren. Die neue Programmstruktur erlaubte nun Operationen auf Bitebene durchzuführen. Auch der RAM war nun auf Bits ausgelegt.
Positiv war, dass wir das Handbuch und diverse Testprogramme hatten. Dadurch fiel es leichter, die Logik zu verstehen und umzusetzen. Durch die Testprogramme konnte direkt überprüft werden, ob Fehler eingebaut wurden. Schade ist allerdings, dass das Handbuch einige Fehler beinhaltet, die einen zeitlich doch sehr aufgehalten hatten. Zudem lieferten die Testprogramme, im bereitgestellten PicSimulator andere Werte als im Listing beschrieben, was ebenfalls in Verwirrung endete. Nur durch manuelles Nachrechnen der einzelnen Programme konnte so Klarheit geschaffen werden.
Der PicSimulator ist das erste Projekt während der Theoriephase gewesen, das wir selbstständig bearbeiten mussten. Für dieses erste Projekt ist der Umfang doch sehr groß und benötigt viel mehr Zeit als die Vorlesungsstunden hergeben. So musste auch vieles nebenher oder an Wochenenden entwickelt werden. Unserer Meinung nach entspricht der Arbeitsaufwand für den PicSimulator viel mehr einer Klausur als einem Testat und hätte eine Note verdient.
Mit dem Ziel, das Projekt so schnell wie möglich, aber dennoch gewissenhaft abzuschließen haben wir nur 11 von 15 Testprogrammen umgesetzt. Da 50 Punkte zum Bestehen des Testats ausreichen, haben wir uns einen entsprechenden Puffer ausgerechnet und danach

die Entwicklung eingestellt. Es ist schade, da die Entwicklung des Simulators Spaß gemacht hat, allerdings andere Kurse nun mehr Aufmerksamkeit benötigen, da dort bald eine Klausur ansteht.
Abschließend lässt sich sagen, dass das Verständnis zum Pic nun ein völlig anderes ist, als das, was im Kurs davor gelernt wurde. Die Prozesse wurden verstanden und die Arbeitsweisen sowie Register und deren Funktionen durchdrungen. Mit diesem Wissen wäre die Klausur aus letztem Semester noch besser ausgefallen! =)

A Anhang

A.1 Anhang

```
1         if (decoded.cmd == "BTFSC") {
2             BYTE ramcontent = ram1.getRam(decoded.filepos);
3             if (ramcontent.test(decoded.literal) == 0) {
4                 nop();
5             }
6             programCounter = programCounter.to_ulong() + 1;
7             ram1.ramArray[2] = createPCL().to_ulong();
8             cycle++;
9             runtime = runtime + multiplier;
10        }
```

Listing A.1: Code zum Befehl BTFSC

```
1         if (decoded.cmd == "BTFSS") {
2             BYTE ramcontent = ram1.getRam(decoded.filepos);
3             if (ramcontent.test(decoded.literal) == 1) {
4                 nop();
5             }
6             programCounter = programCounter.to_ulong() + 1;
7             ram1.ramArray[2] = createPCL().to_ulong();
8             cycle++;
9             runtime = runtime + multiplier;
10        }
```

Listing A.2: Code zum Befehl BTFSS

```
        if (decoded.cmd == "CALL") {
            utility convert(decoded.literal);
            utility fillup(convert.toBinary(), 11);
            std::string inttobin = fillup.fillup();
            stack1.push(programCounter.to_ulong() + 1);
            programCounter = stoi((pclath43() + inttobin), 0,
                2);
            ram1.ramArray[2] = createPCL().to_ulong();
            cycle = cycle + 2;
            runtime = runtime + (2 * multiplier);
        }
```

Listing A.3: Code zum Befehl CALL

```
1         if (decoded.cmd == "DECFSZ") {
2             if (decoded.dBit == 0) {
3                 wreg = ram1.getRam(decoded.filepos).to_ulong
                      () - 1;
4                 if (wreg == 0) {
5                     nop();
6                 }
7             } else {
8                 cycle = ram1.setRam(decoded.filepos, ram1.
                      getRam(decoded.filepos).to_ulong() - 1,
                      cycle);
9                 createPC(decoded.filepos);
10                if (ram1.getRam(decoded.filepos).to_ulong()
                      == 0) {
11                    nop();
12                }
13            }
14            programCounter = programCounter.to_ulong() + 1;
15            ram1.ramArray[2] = createPCL().to_ulong();
16            cycle++;
17            runtime = runtime + multiplier;
18        }
```

Listing A.4: Code zum Befehl DECFSZ

```
1         if (decoded.cmd == "MOVF") {
2             if (decoded.dBit == 0) {
3                 wreg = ram1.getRam(decoded.filepos);
4                 ram1.zeroFlag(wreg.to_ulong());
5             } else {
6                 ram1.zeroFlag(ram1.getRam(decoded.filepos).
                      to_ulong());
7             }
8             programCounter = programCounter.to_ulong() + 1;
9             ram1.ramArray[2] = createPCL().to_ulong();
10            cycle++;
11            runtime = runtime + multiplier;
12        }
```

Listing A.5: Code zum Befehl MOVF

```
1     if (decoded.cmd == "RRF") {
2         BYTE toshift = ram1.getRam(decoded.filepos);
3         if (ram1.getRam(3).test(0) == 1) // Test Carry Ram
4         {
5             if (toshift.test(0) == 1) // test lsb = 1? -->
                  carry
6             {
7                 toshift >>= 1;
8                 toshift.set(7);
9                 ram1.setCarry(true);
10                if (decoded.dBit == 0) {
11                    wreg = toshift.to_ulong();
12                } else {
13                    cycle = ram1.setRam(decoded.filepos,
                          toshift.to_ulong(), cycle);
14                    createPC(decoded.filepos);
15                }
16            } else { // test lsb = 0 --> set no carry!
17                toshift >>= 1;
18                toshift.set(7);
19                ram1.setCarry(false);
20                if (decoded.dBit == 0) {
21                    wreg = toshift.to_ulong();
22                } else {
23                    cycle = ram1.setRam(decoded.filepos,
                          toshift.to_ulong(), cycle);
24                    createPC(decoded.filepos);
25                }
26            }
27        } else {// no carry before instruction!
28            if (toshift.test(0) == 1) // test lsb = 1? -->
                  carry
29            {
30                toshift >>= 1;
31                ram1.setCarry(true);
```

```
32                  if (decoded.dBit == 0) {
33                      wreg = toshift.to_ulong();
34                  } else {
35                      cycle = ram1.setRam(decoded.filepos,
                            toshift.to_ulong(), cycle);
36                      createPC(decoded.filepos);
37                  }
38              } else {// test lsb = 0 --> set no carry!
39                  toshift >>= 1;
40                  ram1.setCarry(false);
41                  if (decoded.dBit == 0) {
42                      wreg = toshift.to_ulong();
43                  } else {
44
45                      cycle = ram1.setRam(decoded.filepos,
                            toshift.to_ulong(), cycle);
46                      createPC(decoded.filepos);
47                  }
48              }
49          }
50          programCounter = programCounter.to_ulong() + 1;
51          ram1.ramArray[2] = createPCL().to_ulong();
52          cycle++;
53          runtime = runtime + multiplier;
54      }
```

Listing A.6: Code zum Befehl RRF

```
if (decoded.cmd == "SUBWF") {
    if (decoded.dBit == 0) {
        additionReturn additionret = ram1.doaddition(ram1.
           getRam(decoded.filepos), (~wreg).to_ulong() + 1);
        wreg = additionret.result;
        ram1.zeroFlag(wreg.to_ulong());
        if (additionret.carry == 1) {
            ram1.setCarry(true);
        } else {
            ram1.setCarry(false);
        }
        if (additionret.dcarry == 1) {
            ram1.setDCarry(true);
        } else {
            ram1.setDCarry(false);
        }
    } else {
        additionReturn additionret = ram1.doaddition(ram1.
           getRam(decoded.filepos), (~wreg).to_ulong() + 1);
        cycle = ram1.setRam(decoded.filepos, additionret.
           result.to_ulong(), cycle);
        ram1.zeroFlag(ram1.getRam(decoded.filepos).to_ulong()
           );
        if (additionret.carry == 1) {
            ram1.setCarry(true);
        } else {
            ram1.setCarry(false);
        }
        if (additionret.dcarry == 1) {
            ram1.setDCarry(true);
        } else {
            ram1.setDCarry(false);
        }
        createPC(decoded.filepos);
    }
```

```
32      programCounter = programCounter.to_ulong() + 1;
33      ram1.ramArray[2] = createPCL().to_ulong();
34      cycle++;
35      runtime = runtime + multiplier;
36  }
```

Listing A.7: Code zum Befehl SUBWF

```
1      if (decoded.cmd == "XORLW") {
2          wreg = wreg.to_ulong() ^ decoded.literal;
3          ram1.zeroFlag(wreg.to_ulong());
4          programCounter = programCounter.to_ulong() + 1;
5          ram1.ramArray[2] = createPCL().to_ulong();
6          cycle++;
7          runtime = runtime + multiplier;
8      }
```

Listing A.8: Code zum Befehl XORLW