

Dokumentation

Simulator für den PIC16F84A

Autoren
Datum

Kerstin Buchholz, Maciej Niestoruk
21. Juni 2015

Inhaltsverzeichnis

1	Einleitung	1
2	Grundlagen	2
2.1	Simulation	2
2.2	Microcontroller	3
3	Ausführung	5
3.1	Entwicklungsumgebung und Programmiersprache	5
3.2	Oberfläche	6
3.3	Innere Programmstruktur	8
3.4	Befehlsgruppen	9

1 Einleitung

Im Verlauf der Rechnertechnikvorlesung des 4. Semesters soll ein Simulator für den Microkontroller PIC16F84 entwickelt werden. Dieser soll die Funktionsweise des PIC so detailliert wie möglich darstellen

2 Grundlagen

2.1 Simulation

2.1.1 Was ist Simulation?

Eine Simulation oder auch ein Simulator stellen eine möglichst realitätsnahe Nachbildung eines Systems in der Wirklichkeit dar. Simulationen dienen zur Veranschaulichung und Analyse von dynamisch verhaltenden Systemen.

2.1.2 Vor- und Nachteile eines Simulators

Vorteile einer Simulation sind meist offensichtlich:

- Viele reale Systeme können mit analytischen Modellen nicht genau genug beschrieben werden, während in einem Simulationsmodell die realen Abläufe sehr gut nachgebildet werden können.
- Simulation bietet die Möglichkeit, Systeme in ganz unterschiedlichen Umgebungen und unter unterschiedlichen Bedingungen zu analysieren. Diese Bedingungen können aus der realen Umwelt stammen, können aber auch vollkommen irreal sein.
- Modifikationen am System sind oft durch einfache Änderungen am Simulator modellierbar.
- Experimente können prinzipiell relativ einfach und kostengünstig durchgeführt werden. Da in der Simulation praktisch alle Faktoren beeinflussbar sind, lassen sich Experimente unter quasi beliebigen Bedingungen durchführen
- Simulation erlaubt es Systeme auch in sehr kurzen oder sehr langen Zeitintervallen zu beobachten, wenn der Simulator die Zeit entsprechend streckt oder staucht.
- Mit heutigen Rechnerkapazitäten lassen sich viele und aufwändige Simulationsexperiment mit den vorhandenen Ressourcen durchführen.

Doch eher seltener wirdend die Nachteile einer Simulation beachtet:

- Die meisten Simulationsmodelle sind stochastisch, so dass Resultate nur geschätzt werden können. Die dazu verwendeten Methoden basieren fast immer auf zwar plausiblen aber nicht beweisbaren Annahmen, so dass für einzelne Modelle falsche Resultate ermittelt werden.
- Simulationsmodelle haben einen hohen Datenbedarf, der bei vielen realen Experimenten nicht befriedigt wird. So kann ein Modell nicht besser als der schwächste Punkt sein. In der Simulation bedeutet dies oft, dass zwar detaillierte Modelle erstellt werden, die benötigten Parameter aber nur auf Grund von sehr wenigen Beobachtungen geschätzt werden und in manchen Fällen nur auf Grund von nicht validierten Annahmen gesetzt werden.
- Simulationsmodelle sind aufwändig und teuer in der Entwicklung,
 - neben den üblichen Problemen bei der Erstellung großer Programme
 - treten simulationsspezifische Probleme, wie die Datenmodellierung, auf
- Simulationsmodelle liefern in manchen Fällen extreme Datenmengen, die oft nicht detailliert genug analysiert werden, so dass Resultate oft überinterpretiert oder falsch interpretiert werden.

2.2 Microcontroller

Ein Mikrocontroller ist eine Art Mikrorechnersystem, bei welchem neben ROM und RAM auch Peripherieeinheiten wie Schnittstellen, Timer und Bussysteme auf einem einzigen Chip integriert sind.

Die Hauptanwendungsgebiete sind die Steuerungs-, Mess- und Regelungstechnik, sowie die Kommunikationstechnik und die Bildverarbeitung. Mikrocontroller sind in der Regel in Embedded Systems, in die Anwendung eingebettete Systeme, und somit in der Regel von außen nicht sichtbar. Ebenso verfügen sie, im Gegensatz zum PC, nicht über eine direkte Bedien- und Programmierschnittstelle zum Benutzer. Sie werden in der Regel einmal programmiert und installiert.

2.2.1 Der PIC16F84

Der Pic16F84 ist ein 8bit Mikrocontroller mit Reduced-Instruction-Set-Computing-Architektur (RISC). Dadurch wird auf komplexe Befehle verzichtet und es kann mit allen Befehlen auf alle Register zugegriffen werden. Durch die Harvard Architektur des Pic16 sind Daten und Programmbefehle in zwei verschiedenen Registern, was den Programmbefehlern ermöglicht

14-Bit groß zu sein, während der separate Datenbus nur 8-Bit groß ist. Ein großer Vorteil dieser Architektur ist, dass fast alle Anweisungen des Pic16 nur einen Instruction Cycle benötigen.

Der Pic16F besitzt einen Stack mit Speicherplatz für acht Adressen und zwei externe sowie zwei interne Interruptquellen. Darüber hinaus besitzt der Pic16F ein großes Register, welches in zwei Bänke unterteilt ist. Das Umschalten der Bänke erfolgt im Programmcode. Die Speicherbereiche können auch direkt über ihre Registeradresse angesprochen werden.

3 Ausführung

3.1 Entwicklungsumgebung und Programmiersprache

Um so wenig Aufwand wie möglich mit der Visualisierung zu haben, entschieden wir uns für die Entwicklungsumgebung VisualStudio2013 und der Programmiersprache c#
VisualStudio hat den Vorteil, dass sich ein Großteil der Oberfläche per Drag-and-Drop gestalten lässt und für C# eine sehr große Anzahl an WindowsForms zur Verfügung stellt.
Somit lassen sich Programme für Windows-Plattformen sehr leicht realisieren.

3.2 Oberfläche

Bei der Gestaltung der Benutzeroberfläche des Picsimulators lag der Augenmerk auf Übersichtlichkeit und intuitive Benutzung. Über diese Oberfläche ist es dem Benutzer möglich die Simulation zu steuern.

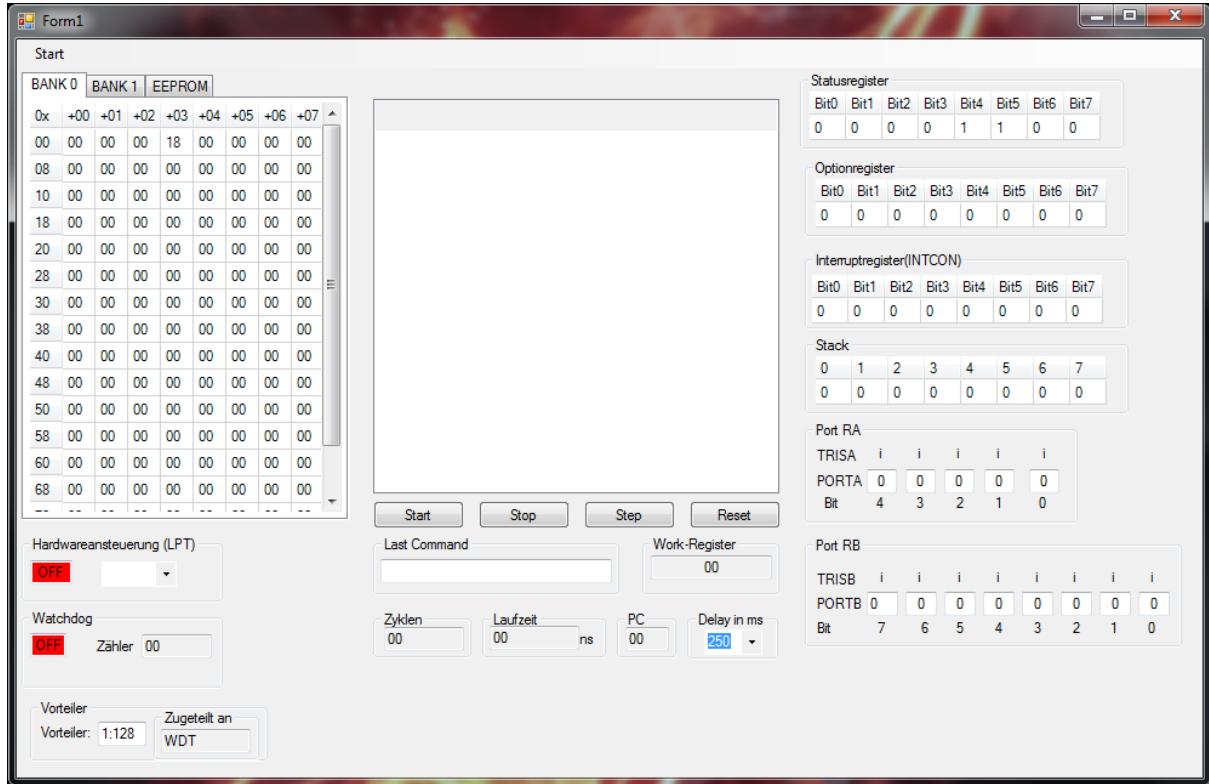


Abb. 3.1: Oberfläche des PicSimulators

3.2.1 Datei laden

Links oben, über den Menü-Eintrag SStart> "Datei laden" kann eine zuvor fertig compilierte *.lst Datei in den Simulator geladen werden.

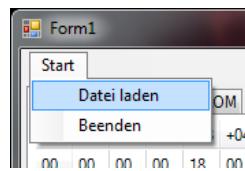


Abb. 3.2: Datei laden

3.2.2 Visualisieren des Programmcodes

Damit immer ersichtlich ist, an welchem Punkt sich das Programm befindet, wird der Programmcode der *.lst als Liste angezeigt.

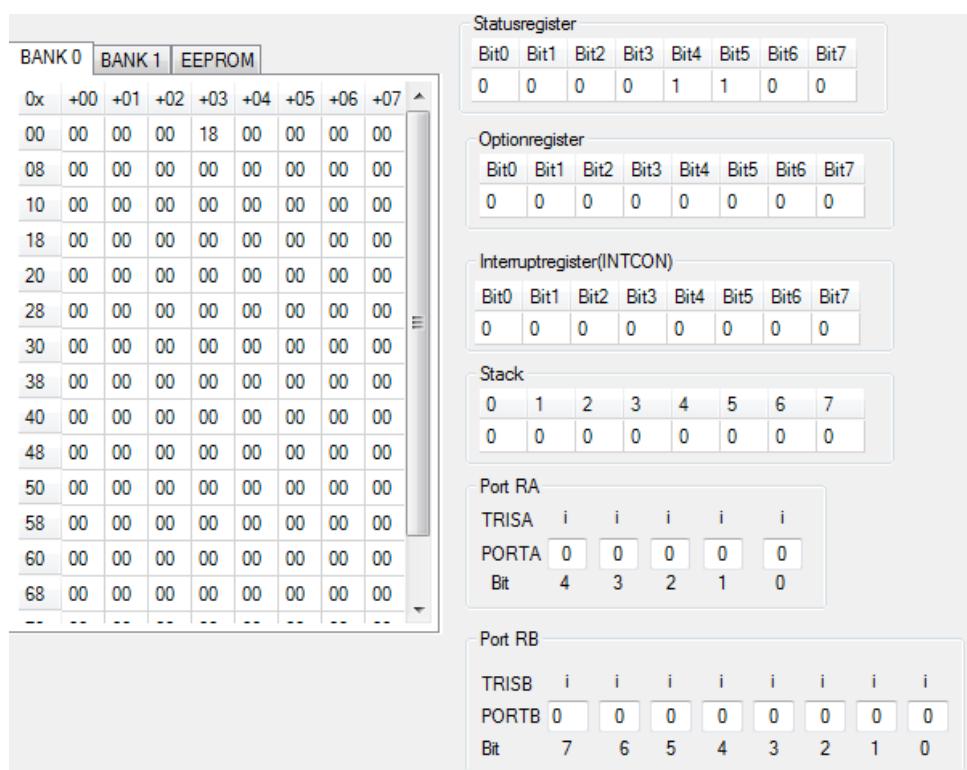


0000 2817	00029	goto main ;Unterprogramme
0001 3010	00033	movlw 16 ;Schleifenzähler
0002 008C	00034	movwf count
0003 3010	00035	movlw 10h ;Startzeiger initialisieren
0004 0084	00036	movwf fsr ;Zeiger ins FSR
0005 0100	00037	clrw
0006 0080	00039	movwf indirect ;Wert indirekt abrufen
0007 0A84	00040	incf fsr ;Zeiger erhöhen
0008 3E01	00041	addlw 1 ;W-Register erhöhen
0009 0B8C	00042	decfsz count ;Schleifenzähler
000A 2806	00043	goto loop1 ;wiederholen bis
000B 3400	00044	retlw 0 ;Schleifenzähler
000C 3010	00048	movlw 10h ;Schleifenzähler
000D 008C	00049	movwf count
000E 0084	00050	movwf fsr ;Startzeiger initialisieren
000F 0100	00051	clrw ;Summenregister löschen
0010 0700	00052	addwf indirect ;Summe erhöhen

Abb. 3.3: Datei laden

3.2.3 Die Register

Die großen Register wie Bank0 und 1 sowie EEPROM sind über die Tabs anwählbar. Die kleineren Register sind direkt auf der Oberfläche sichtbar.



BANK 0								BANK 1								EEPROM																					
0x	+00	+01	+02	+03	+04	+05	+06	+07	00	01	02	03	04	05	06	07	00	01	02	03	04	05	06	07	00	01	02	03	04	05	06	07					
00	00	00	00	18	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00				
08	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00				
10	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00				
18	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00				
20	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00				
28	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00				
30	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00				
38	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00				
40	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00				
48	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00				
50	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00				
58	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00				
60	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00				
68	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00				
..

Statusregister

Bit0	Bit1	Bit2	Bit3	Bit4	Bit5	Bit6	Bit7
0	0	0	0	1	1	0	0

Optionregister

Bit0	Bit1	Bit2	Bit3	Bit4	Bit5	Bit6	Bit7
0	0	0	0	0	0	0	0

Interruptregister(INTCON)

Bit0	Bit1	Bit2	Bit3	Bit4	Bit5	Bit6	Bit7
0	0	0	0	0	0	0	0

Stack

0	1	2	3	4	5	6	7
0	0	0	0	0	0	0	0

Port RA

TRISA	i	i	i	i	i	i	i
PORTA	0	0	0	0	0	0	0
Bit	4	3	2	1	0		

Port RB

TRISB	i	i	i	i	i	i	i
PORTB	0	0	0	0	0	0	0
Bit	7	6	5	4	3	2	1

Abb. 3.4: Die Register

3.2.4 Programm Starten

Über den Button *Start* unter der Programmcode liste kann das geladene Programm gestartet werden. Mit *Stop* wird das Programm Pausiert und dann kann über *Step* das Programm Schrittweise durchlaufen werden. Möchte man das Programm neu starten, kann man über *Reset* das Programm zurücksetzen.

Im Feld *Last Command* Wird der zuletzt ausgeführte Befehl und im *Work-Register* der Inhalt des Arbeitsregisters angezeigt. Darunter sind die *Zyklen*, die *Laufzeit* und im *PC* der ProgrammCounter zu sehen. Unter *Delay in ms* ist die Geschwindigkeit des Programms einstellbar.

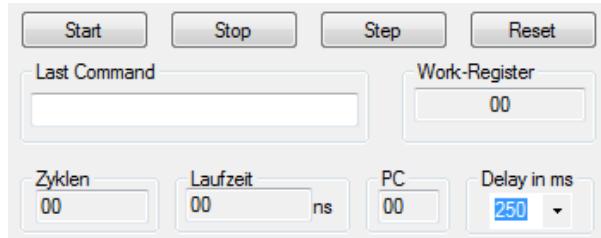


Abb. 3.5:

3.3 Innere Programmstruktur

3.3.1 Simulationsvorgang Ablauf

Wird eine Datei eingelesen wird der der Simulator zurückgesetzt. Der Benutzer kann auf Start klicken, um die Simulation zu starten. Eine Verzögerung kann beim Feld Delay eingestellt werden, um auch jeden Befehl verfolgen zu können. Man kann den Simulator mit dem Stop-Button anhalten. Breakpoints können gesetzt werden. Es ist auch möglich mit dem Step-Button jeden Befehl einzeln auszuführen. Nach jedem Befehl werden die Register aktualisiert und in der GUI angezeigt. Sind die Interrupt-enabled-Flags gesetzt wird nach jedem Befehl überprüft ob ein Interrupt ausgelöst wurde und die entsprechende ISR ausgeführt. Wird der Reset-Button gedrückt oder läuft der Watchdogtimer über, wird der Simulator samt Programmcounter zurückgesetzt.

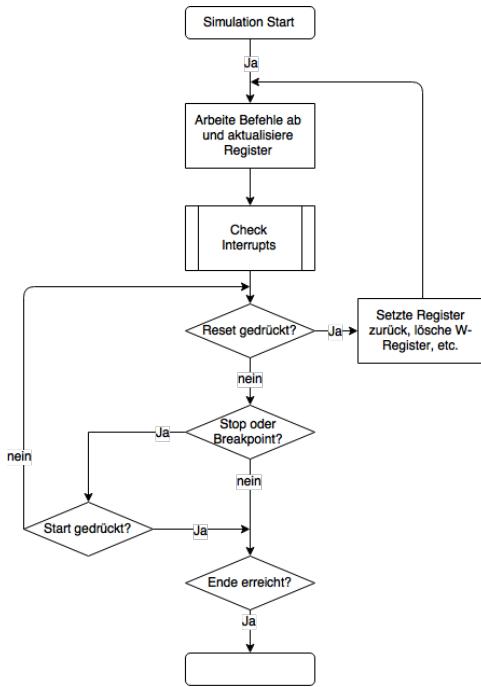


Abb. 3.6: Simulationsvorgang

3.4 Befehlsgruppen

3.4.1 Byteorientierte File Register Befehle

Bei den byteorientierten File Register Befehlen handelt es sich meist um Operationen zwischen einem Fileregister und dem Akkumulator. Das Ergebnis wird entweder in das W-Register ($d = 0$) oder ins Fileregister ($d = 1$) geschrieben. Bei Addition und Subtraktion werden das Carryflag, DigitCarry und Zerobit gesetzt/bzw. geclearnt. Die meisten Operationen überprüfen jedoch lediglich ob das Ergebnis 0 ist.

MOVF

Der Befehl MOVF schreibt den Wert des bestimmten Registers entweder in sich selbst, wenn $d = 1$ oder in das W-Register, wenn $d = 0$. Der Befehl ist z.B. dann sinnvoll wenn nach Setzen des Zerobits das Register getestet werden soll. Für diesen Befehl wird eine Hilfsvariable benötigt, da das Ergebnis entweder in W oder im Register selbst geschrieben wird. 3.7

```

if (f == 0) f = (byte)iReg[0x04]; // indirect
//
int temp = iReg[f]; // temporary result

// Z status
  
```

```

if (temp == 0) // is the result of the logic operation zero
{
    iReg[0x03] |= 0x04; // Z = 1
}
else
{
    iReg[0x03] &= 0xFB; // Z = 0
}

if (d == 0) // Save in W or f
    iWReg = temp;
else
    iReg[f] = temp;

lCycles++;
iPC++;

```

RRF

Die Rotate Right Funktion shiftet ein den inhalt eines Registers um eine Stelle nach rechts. Dies entspricht einer Ganzzahldivision durch 2. Das Ergebnis wird abhängig von d entweder in W oder ins Fileregister geschrieben. [3.8](#)

```

if (f == 0x00) f = (byte)iReg[0x04]; // Indirect
// 

int temp = iReg[f] >> 1;      // temporary variable
if (temp < 0 )
{
    iReg[0x03] |= 0x01; //C=1;
    temp &= 0xff;      // keep 8 bits only
}
else
{
    iReg[0x03] &= 0xFE; //C=0;
}
if (d == 0) // save in either iWReg or iReg[f]
{
    iWReg = temp;
}
else
{
    iReg[f] = temp;
}

```

```
lCycles++;
iPC++;
```

SUBWF

Der Befehl SUBWF subtrahiert von einem Fileregister den Inhalt des W-Registers. Das Ergebnis wird entweder ins W-Register oder ins Fileregister geschrieben. Das Besondere an SUBWF: Das Carryflag wird hier gesetzt wenn das Ergebnis 0 ist oder wenn es keinen Überlauf gibt! Auch hier wird eine Hilfsvariable benötigt. [3.9](#)

```
if (f == 0x00) f = (byte)iReg[0x04]; // Indirect
//  
  

int temp = iReg[f] - iWReg; // temporary variable
if (temp < 0)
{
    iReg[0x03] &= 0xFE; //C=0;
    iReg[0x03] &= 0xFB; //Z = 0; //DC wird nach Korrektur
    bestimmt
    temp += 256; //Ueberlauf
}
else
{
    iReg[0x03] |= 0x01; //C=1;
}  
  

if (temp == 0)
{
    iReg[0x03] |= 0x04; //Z=1;
    iReg[0x03] |= 0x01; //C=1;
}
else
{
    iReg[0x03] &= 0xFB; //Z=0;
    iReg[0x03] &= 0xFE; //C=0;
}  
  

if (temp >= 0x10)
{
    iReg[0x03] |= 0x02; //DC=1;
}
else
{
    iReg[0x03] &= 0xFD; //DC=0;
}
```

```

if (d == 0) // save in either iWReg or iReg[f]
{
    iWReg = temp;
}
else
{
    iReg[f] = temp;
}
lCycles++;
iPC++;

```

DECFSZ

Dieser Befehl dekrementiert den Inhalt des bestimmten Fileregisters und schreibt das Ergebnis entweder ins Fileregister oder in W. Wird die 0 dekrementiert so ist das Ergebnis 255. Ist das Ergebnis 0 wird ein NOP ausgeführt und der nächste Befehl übersprungen. Ansonsten wird der nächste Befehl ausgeführt. DECFSZ dauert 1-2 Befehlszyklen.[3.10](#)

```

if (f == 0) f = (byte)iReg[0x04]; // indirect

int temp = iReg[f] - 0x01; // temporary result of
                           decrementation

if (temp < 0)
    temp += 256;

if (d == 0) // Save in W or f
{
    iWReg = temp;
}
else
{
    iReg[f] = temp;
}
if (temp == 0)
{
    //NOP!
    lCycles += 2;
    iPC += 2;
}
else
{
    lCycles++;
    iPC++;
}

```

3.4.2 Bitorientierte File Register Befehle

BSF

Der Befehl BSF (Bit set f) setzt in einem File Register ein Bit. Dem Befehl werden die Adresse bzw. ein Label und die Nummer des zu setzenden Bits als Parameter übergeben. Ein Fileregister hat 8 Bit, nummeriert von 0-7. Der Befehl wird im PIC-Simulator umgesetzt, indem eine Bytemaske durch Shiften von 0x01 nach links erzeugt wird und mit dem entsprechenden Fileregister (Feld im Array iReg) verodert wird.[3.11](#)

```
if (f == 0x00) f = (byte)iReg[0x04]; // Indirect
if ((iReg[0x03] & 0x20) > 0) f += 0x80; // Bank 1 check

iReg[f] = iReg[f] | b; // set bit
lCycles++;
iPC++;
```

BTFSC

Mit BTFSC (Bit Test f Skip if Clear) wird eine Abfrage eines Fileregisters gemacht, ob ein bestimmtes Bit 0 ist. Dazu wird das Fileregister mit der Bytemaske b verundet. Ist das Bit gesetzt wird der folgende Befehl ausgeführt. Ist dies nicht der Fall wird ein NOP ausgeführt und der nächste Befehl übersprungen. Dieser Befehl wird häufig bei Schleifen oder Portabfragen verwendet. [3.12](#)

```
if (f == 0x00) f = (byte)iReg[0x04]; // Indirect
if ((iReg[0x03] & 0x20) > 0) f += 0x80; // Bank 1 check

if ((iReg[f] & b) > 0) // Bit set?
{
    lCycles++;
    iPC++;
}
else
{
    //NOP
    lCycles += 2;
    iPC += 2;
}
```

3.4.3 Literal- und Kontroll Befehle

[3.13](#)

CALL

```
stack.Push(iPC + 1);
1Cycles += 2;
iPC = k;
```

GOTO

```
1Cycles += 2;
iPC = k; // jump to address
```

MOVLW

Bei MOVLW wird ein Literal mit einem Wert von 0 bis 255 in das W-Register geschrieben. Dazu wird der Parameter k direkt iWReg zugewiesen.

```
iWReg = k;
1Cycles++;
iPC++;
```

XORLW

XORLW führt eine Exklusivoderverknüpfung des W-Register mit einem Literal aus. Ist das Ergebnis 0 wird das Zeroflag gesetzt. Der Befehl dauert 1 Zyklus und das Ergebnis wird in W gespeichert. XORLW führt eine Exklusivoderverknüpfung des W-Register mit einem Literal aus. Ist das Ergebnis 0 wird das Zeroflag gesetzt. Der Befehl dauert 1 Zyklus und das Ergebnis wird in W gespeichert.[3.14](#)

```
iWReg = iWReg ^ k; // XOR
if (iWReg == 0)
    iReg[0x03] |= 0x04; // Z=1;
else
    iReg[0x03] &= 0xFB; // Z=0;
1Cycles++;
iPC++;
```

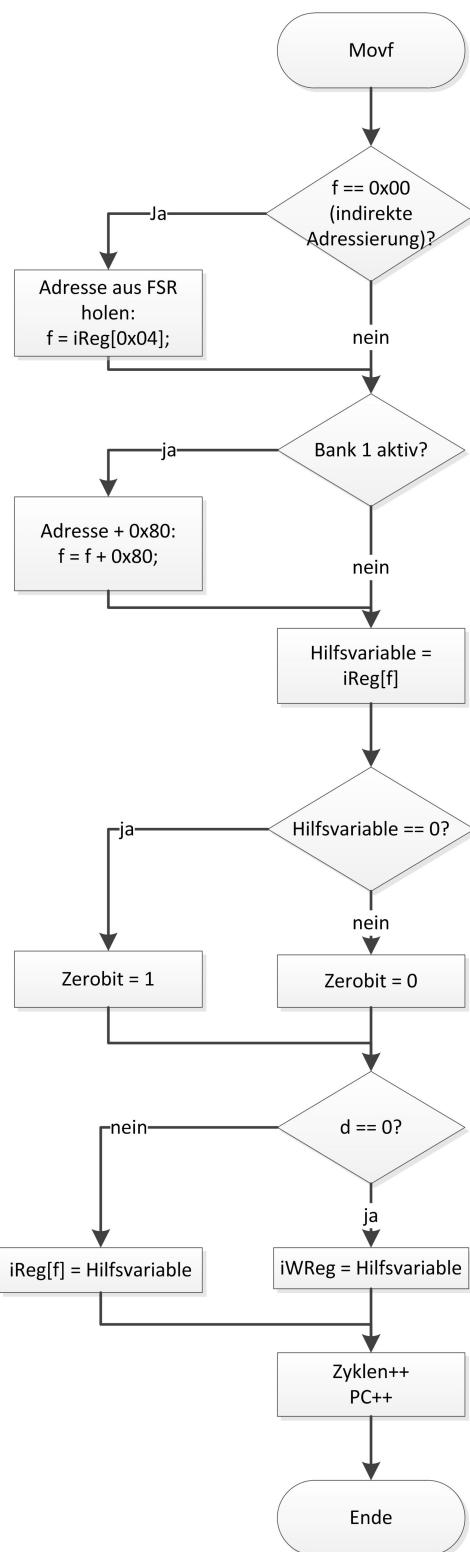


Abb. 3.7: MOVF

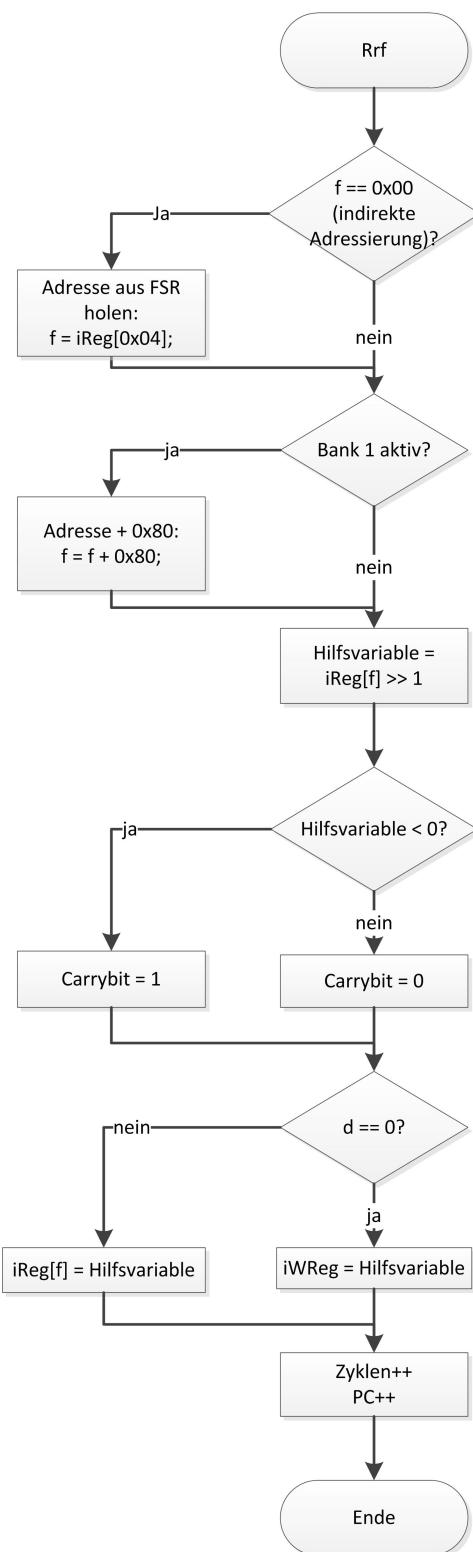


Abb. 3.8: RRF

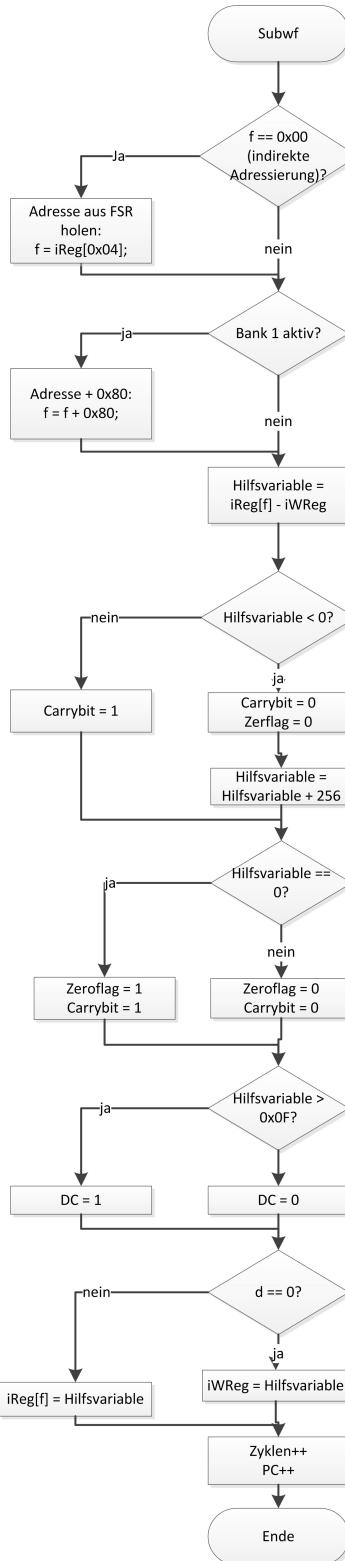


Abb. 3.9: SUBWF

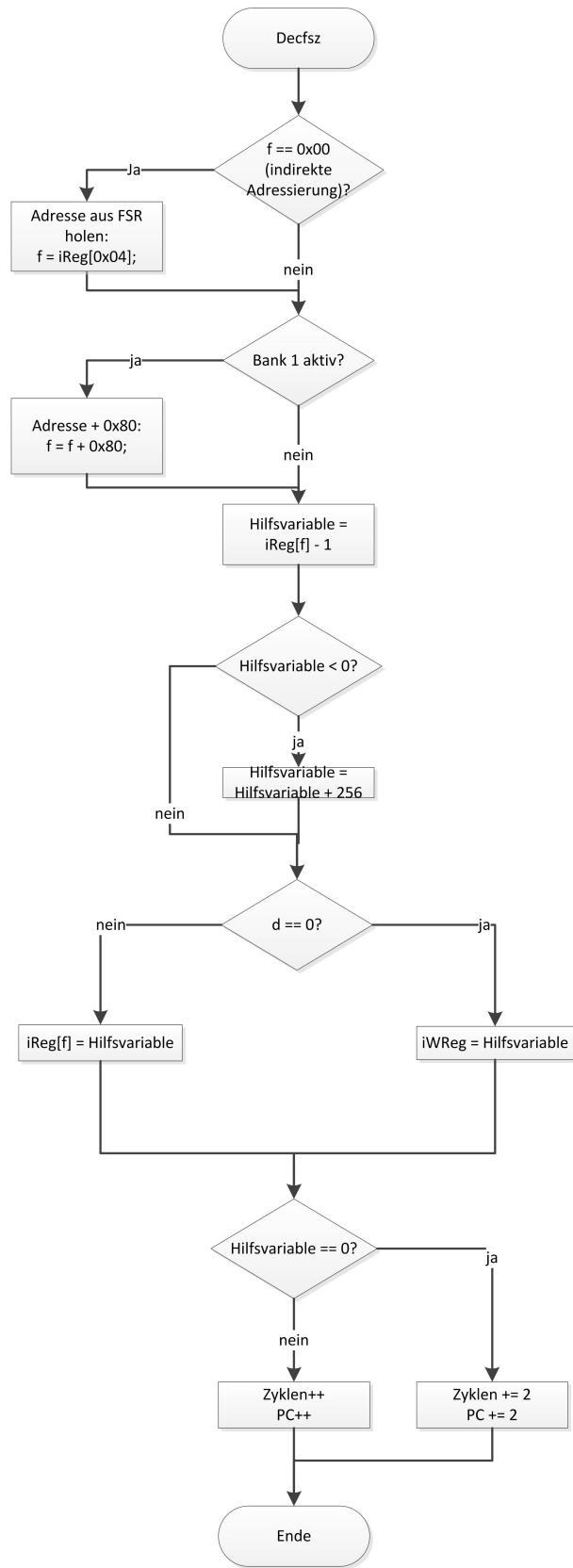


Abb. 3.10: DECFSZ

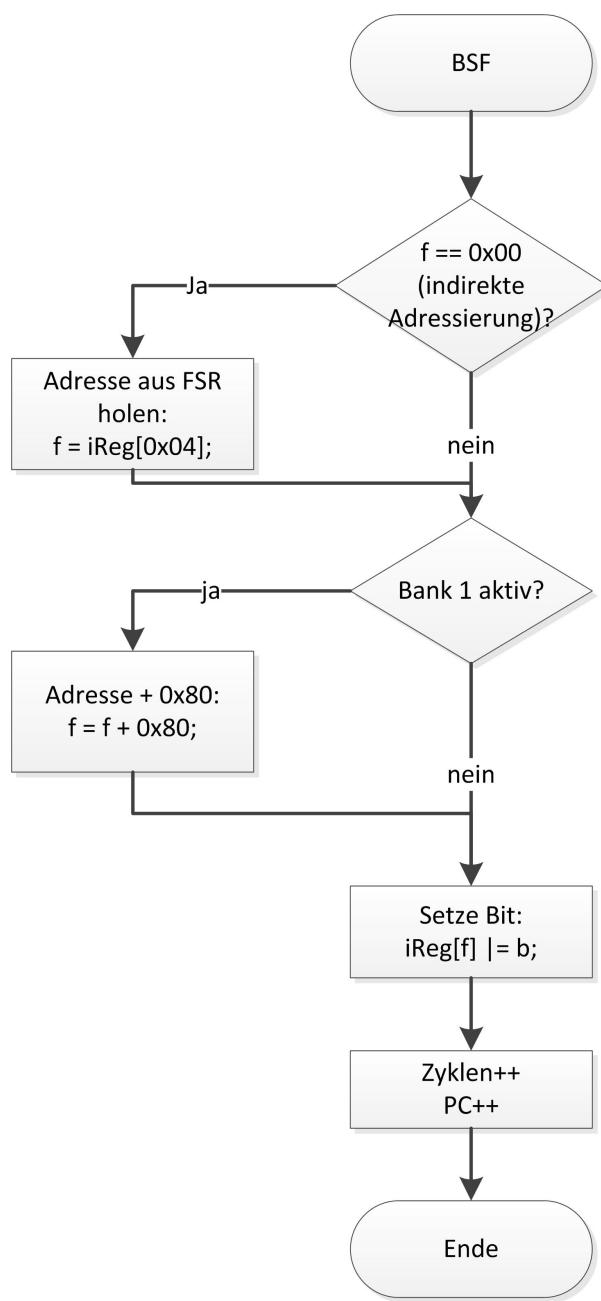


Abb. 3.11: BSF

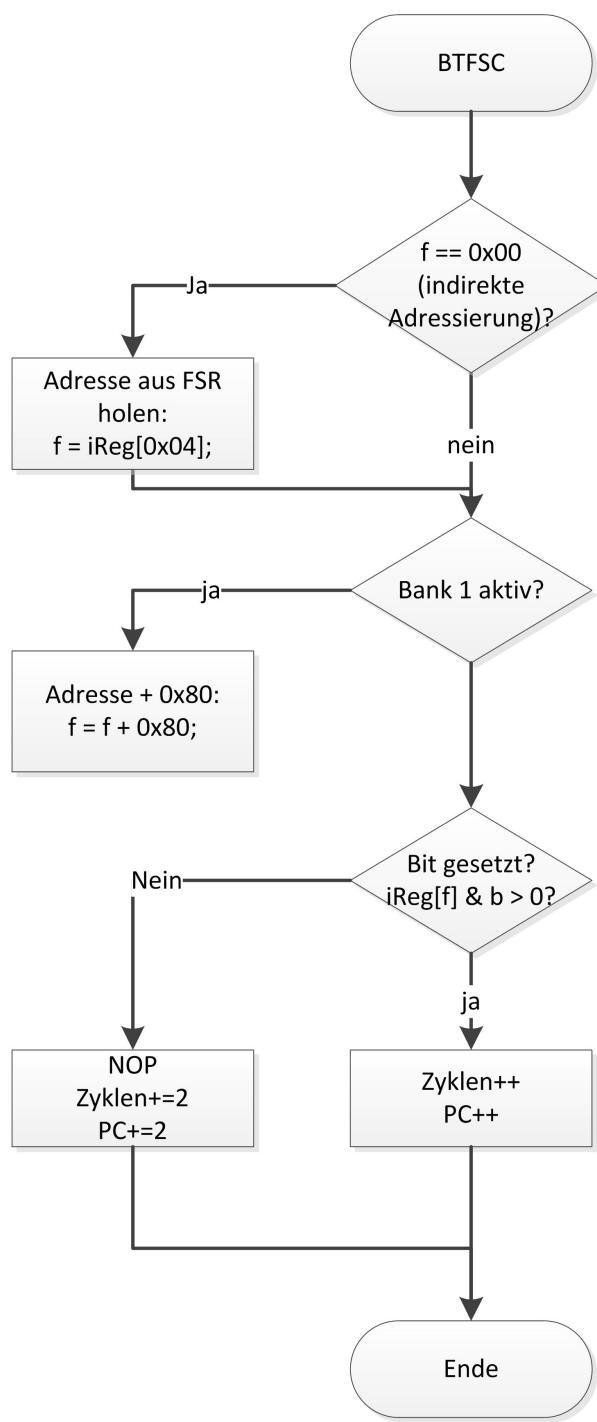


Abb. 3.12: BTFSC

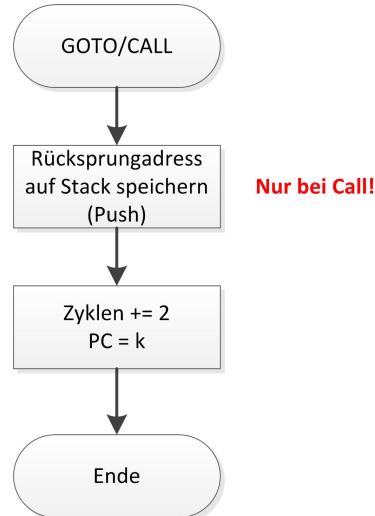


Abb. 3.13: GOTO/CALL

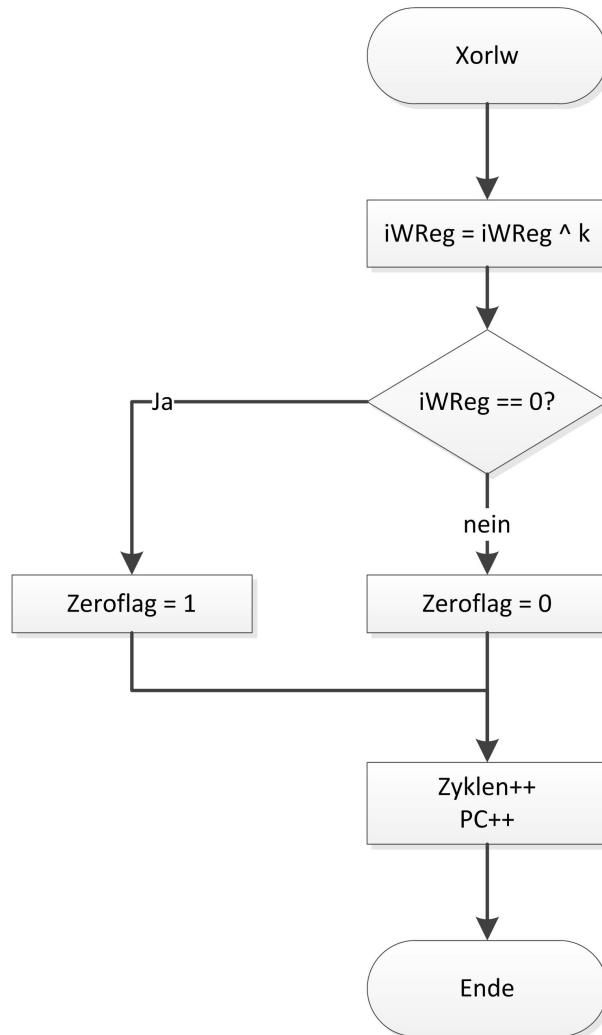


Abb. 3.14: XORLW