DHBW
Duale Hochschule
Baden-Württemberg
Karlsruhe

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

Abbildungsverzeichnis

Tabellenverzeichnis

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

1. Vorwort

Im Fach Rechnertechnik II soll der Aufbau und die Funktionsweise eines Microcontrollers 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.

2. Einleitung

Im Rahmen der Vorlesung Systemnahe Programmierung soll ein Simulator für den Microcontroller PIC16F84 der Firma Microchip Technology Inc. entwickelt werden.

Dabei soll ein grobes Konzept entworfen werden. Dieses Konzept soll später mit Hilfe der Techniken aus der Vorlesung Software-Engineering in Zweiergruppen ausgearbeitet werden.

Es sollen alle Befehle des PIC16F84 in Software abgebildet werden, sodass die zur Verfügung gestellten Tests funktionsfähig sind.

Zuletzt soll eine Dokumentation der geleisteten Arbeit eingereicht werden.

3. Allgemeines

3.1 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.

3.2 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.

3.3 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]

[1] Quelle: https://de.wikipedia.org/wiki/Simulation aufgerufen am 29.05.2017

4. Simulator

Im folgenden Kapitel wird der Aufbau des Simulators erläutert.

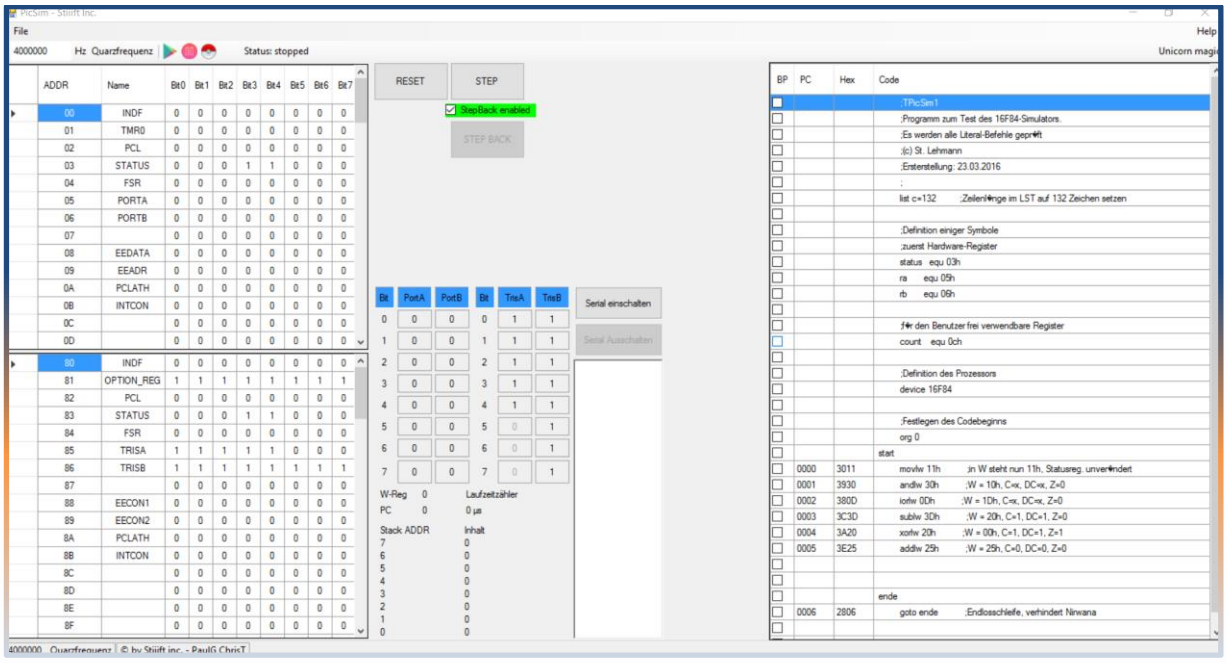


Abbildung 1: GUI

4.1 GUI

Die GUI besteht aus mehreren Elementen, welche nachfolgend erläutert werden.

4.1.1 Buttons

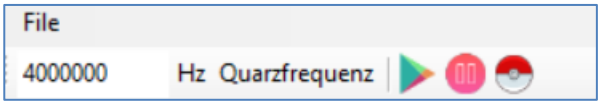


Abbildung 2:Buttons Toolstrip links

Buttons Toolstrip links:

- File

  Öffnet compilierte Assemblerdateien

- Quarzfrequenz

  Durch klicken auf Quarzfrequenz oder das drücken der Enter-Taste wird der links stehende Wert als Quarzfrequenz übernommen

- Play

  Startet die Abarbeitung der Befehle

- Pause

  Pausiert die Abarbeitung der Befehle

- Stop

  Resettet den Simulator

Buttons Toolstrip rechts:

Abbildung 3: Buttons Toolstrip rechts

- Help

  Öffnet die Dokumentation/Hilfe

- Unicorn magic

  Startet den Unicorn-Mode

Buttons zentral:

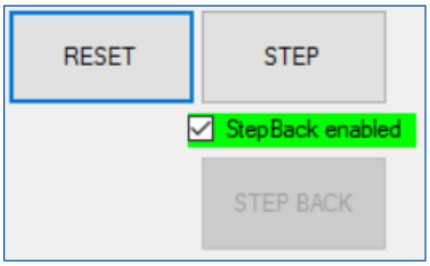


Abbildung 4: Buttons zentral

- Reset

  Resettet den Simulator

- Step

  Führt einen einzelnen Befehl aus

- StepBack

  Macht den vorigen Befehl rückgängig (bis zu 100 Mal)

- StepBack enabled

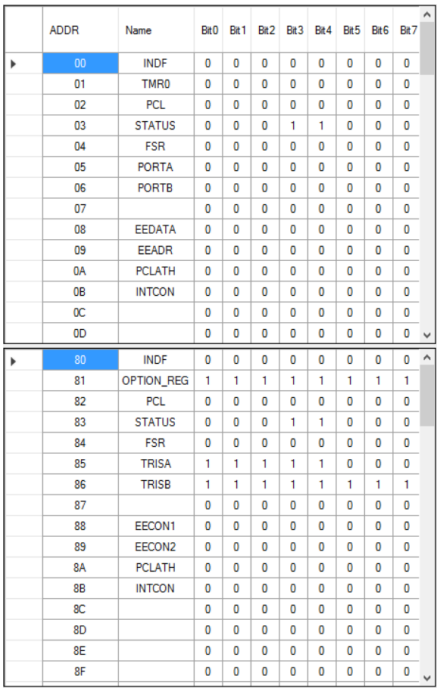
  Wenn ohne Häkchen → StepBack wird nicht gespeichert (kein StepBack mehr möglich).

  Deaktivierung führt zu Performanceerhöhung des Simulators.

4.1.2 Speicher

Der Speicher des Microcontrollers wird in einer DataGridView am linken Rand des Programmes dargestellt. Durch das Klicken auf die einzelnen Bit-Stellen können diese auf 0 bzw. 1 gesetzt werden.

Es ist darauf zu achten, dass das höchstwertige Bit ganz rechts steht (Bit 7).



ADDR	Name	Bit0	Bit1	Bit2	Bit3	Bit4	Bit5	Bit6	Bit7
00	INDF	0	0	0	0	0	0	0	0
01	TMR0	0	0	0	0	0	0	0	0
02	PCL	0	0	0	0	0	0	0	0
03	STATUS	0	0	0	1	1	0	0	0
04	FSR	0	0	0	0	0	0	0	0
05	PORTA	0	0	0	0	0	0	0	0
06	PORTB	0	0	0	0	0	0	0	0
07		0	0	0	0	0	0	0	0
08	EEDATA	0	0	0	0	0	0	0	0
09	EEADR	0	0	0	0	0	0	0	0
0A	PCLATH	0	0	0	0	0	0	0	0
0B	INTCON	0	0	0	0	0	0	0	0
0C		0	0	0	0	0	0	0	0
0D		0	0	0	0	0	0	0	0
80	INDF	0	0	0	0	0	0	0	0
81	OPTION_REG	1	1	1	1	1	1	1	1
82	PCL	0	0	0	0	0	0	0	0
83	STATUS	0	0	0	1	1	0	0	0
84	FSR	0	0	0	0	0	0	0	0
85	TRISA	1	1	1	1	1	0	0	0
86	TRISB	1	1	1	1	1	1	1	1
87		0	0	0	0	0	0	0	0
88	EECON1	0	0	0	0	0	0	0	0
89	EECON2	0	0	0	0	0	0	0	0
8A	PCLATH	0	0	0	0	0	0	0	0
8B	INTCON	0	0	0	0	0	0	0	0
8C		0	0	0	0	0	0	0	0
8D		0	0	0	0	0	0	0	0
8E		0	0	0	0	0	0	0	0
8F		0	0	0	0	0	0	0	0

Abbildung 5: Speicher

4.1.3 Programm

Das Programm wird nach dem Ladeprozess (klicken auf File und auswählen der Assemblerdatei) in die DataGridView am rechten Rand des Programmes geladen.

Durch das klicken auf die Kästchen in der ersten Spalte dieser DataGridView können Breakpoints gesetzt werden. Die zweite Spalte beinhaltet den PC, die dritte Spalte den HexCode und die vierte Spalte den Code ausgeschrieben.

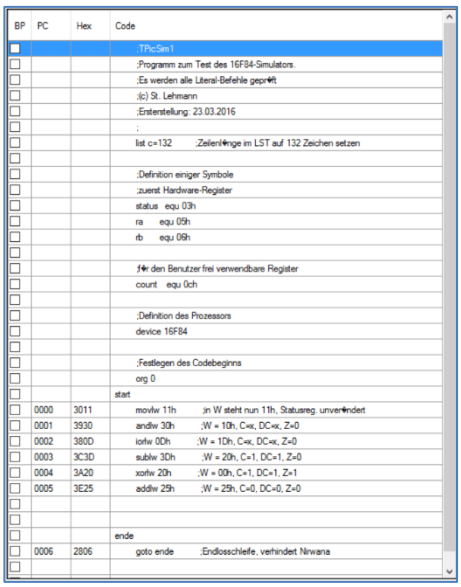


Abbildung 6: Programm

4.1.4 Stack

Der Stack wird in diesem Fenster dargestellt.

Er wird von unten nach oben gefüllt.

Stack ADDR	Inhalt
7	0
6	0
5	0
4	0
3	0
2	0
1	0
0	0

Abbildung 7: Stack

4.1.5 Tris-Register

Die Tris-Register können über das nebenstehende Fenster verändert werden. Die Beschriftung zeigt an, welcher Wert momentan im Speicher gesetzt ist. Ein Klick auf die Beschriftung ändert den Wert zu 0 bzw. 1.

Bit	PortA	PortB	Bit	TrisA	TrisB
0	0	0	0	1	1
1	0	0	1	1	1
2	0	0	2	1	1
3	0	0	3	1	1
4	0	0	4	1	1
5	0	0	5	0	1
6	0	0	6	0	1
7	0	0	7	0	1

Abbildung 8: Tris-Register

5. Programmstruktur

Im Folgenden werden die Klassen kurz erläutert.

Loader:	Der Loader liest die Datei ein und extrahiert die Befehle aus der Assembler-Datei und schreibt diese in ein Array.
Decoder:	Der Decoder interpretiert den aktuellen Befehl und iniziiert darauf hin die Befehlsabarbeitung durch die Klasse **Befehle**
Befehle:	In der Klasse Befehle erfolgt die Abarbeitung der Befehle. Dabei wird auf den Speicher, die Klasse **Memory** zugegriffen.
Memory:	Die Klasse Memory stellt unter anderem den Speicher des PIC16F84 dar. Sie beinhaltet auch alle Register, sowie alles, was mit der Speicherverwaltung zu tun hat.
Interrupter:	Der Interrupter prüft, ob Interrupts vorliegen. Die Interruptflags werden vor jeder Befehlsdecodierung überprüft. Liegt ein Interrupt vor, so leitet er die Interrupt-Service-Routine ein (Springt an die Adresse 4). (Siehe 7.9)
Resetter:	Der Resetter enthält Methoden, um im Falle eines Resets den Speicher, Register etc. zu initialisieren.
Const:	Die Klasse Const enthält Abkürzungen (Konstanten) zur Abkürzung der wichtigen Statusregister.
SerialPorts:	Die Klasse SerialPorts enthält alle Methoden zur Abwicklung der Seriellen Verbindung.
Form1:	Die Klasse Form1 nimmt die Eingaben der GUI entgegen. Sie ist das Herzstück des Simulators. Von dort werden alle Threads gestartet.
Programm:	Die Klasse Programm ist der Einstiegspunkt der Software, von hier wird **Form1** geladen.

Tabelle 1: Klassen

6. 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.

6.1 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;
    }
}
```

6.2 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;
    }
}
```

6.3 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;
    }
}
```

7. Implementierung der Befehle

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

- BTFSC (Bit Test f, Skip if Clear)
- BTFSS (Bit Test f, 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 2 auf Seite 15 stichwortartig erläutert.

Im Anschluss wird die Funktion der Interrupts kurz erläutert.

PreInstructions	PostInstructions

`public void PreInstructions(int binCode)`
`{`
`IsStepBackEnabled();`
 	
`GetTimerValOld();`
 	
`fileAdress = binCode & 0x007F;`
`IndirekteAdressierung(fileAdress);`
 	
`fileVal = getFileVal(fileAdress);`
`literal = binCode & 0x00FF;`
`destination = binCode & 0x0080;`
`// d bit`
`}`
	`public void PostInstruction()`
	`{`
	`CheckPrescalerMode();`
	`CheckTimerMode();`
	`mem.pc++;`
	`InkrementWDT();`
	`mem.IncLaufzeitzaehler();`
	`CheckIOInterrupts();`
	`}`

1. Speichern des Zustandes für StepBack() 2. Speichern des TimerValues für Vergleich 3. Extraktion der FileAdresse aus Binärcode 4. Wenn FileAdresse = 0 →Indirekte Adressierung 5. Wert an Adresse speichern 6. Literal Binärcode extrahieren 7. Destinationsbit aus BinärCode extrahieren	2. Überprüfen des Prescaler und jeweilige Einstellung 3. Timermodus überprüfen und Timer nach Prescaler erhöhen 4. Erhöhung PC 5. Erhöhung Watchdogtimer nach Prescaler 6. Erhöhung Laufzeitzähler 7. IO Interrupts überprüfen

Tabelle 2: Erläuterung Pre- & PostInstructions

Das weitere Vorgehen wird im Folgenden an beispielhaften Befehlen erläutert.

Field	Description
f	Register file address (0x00 to 0x7F)
W	Working register (accumulator)
b	Bit address within an 8-bit file register
k	Literal field, constant data or label
x	Don't care location (= 0 or 1) The assembler will generate code with x = 0. It is the recommended form of use for compat-ibility with all Microchip software tools.
d	Destination select; d = 0: store result in W, d = 1: store result in file register f. Default is d = 1
PC	Program Counter
TO	Time-out bit
PD	Power-down bit

Abbildung 9: Abkürzungen – Auszug aus PIC Doku

7.1 BTFSC (Bit Test f, Skip if Clear)

Syntax:	[*label*] BTFSC f,b
Operands:	0 ≤ f ≤ 127 0 ≤ b ≤ 7
Operation:	skip if (f<b>) = 0
Status Affected:	None
Description:	If bit 'b' in register 'f' is '1', the next instruction is executed. If bit 'b' in register 'f' is '0', the next instruction is discarded, and a NOP is executed instead, making this a 2TCY instruction.

Abbildung 10: BTFSC – Auszug PIC

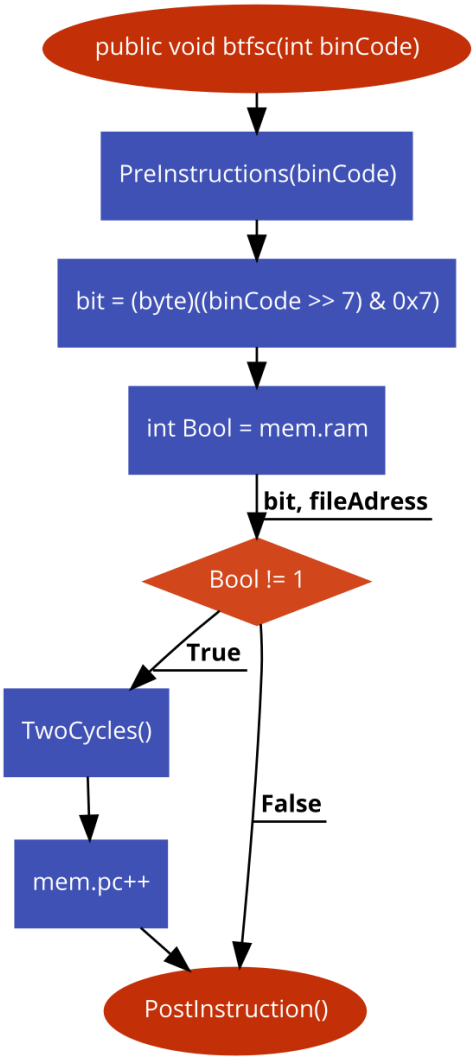


Abbildung 11: PAP BTFSC

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();
}
```

7.2 BTFSS (Bit Test f, Skip if Set)

Syntax:	[*label*] BTFSS f,b
Operands:	0 ≤ f ≤ 127 0 ≤ b < 7
Operation:	skip if (f<b>) = 1
Status Affected:	None
Description:	If bit 'b' in register 'f' is '0', the next instruction is executed. If bit 'b' is '1', then the next instruction is discarded and a NOP is executed instead, making this a 2TCY instruction.

Abbildung 12: BTFSS – Auszug PIC

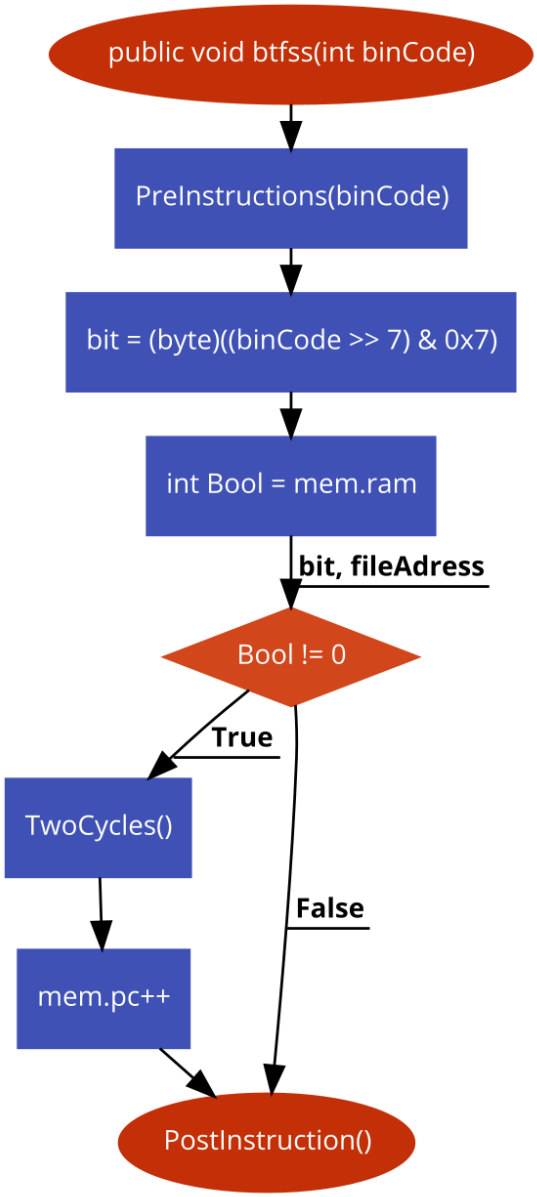


Abbildung 13: PAP BTFSS

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();
 }
```

7.3 CALL (Call Subroutine)

Syntax:	[*label*] CALL k
Operands:	0 ≤ k ≤ 2047
Operation:	(PC)+ 1→ TOS, k → PC<10:0>, (PCLATH<4:3>) → PC<12:11>
Status Affected:	None
Description:	Call Subroutine. First, return address (PC+1) is pushed onto the stack. The eleven-bit immediate address is loaded into PC bits <10:0>. The upper bits of the PC are loaded from PCLATH. CALL is a two-cycle instruction.

Abbildung 14: CALL – Auszug PIC

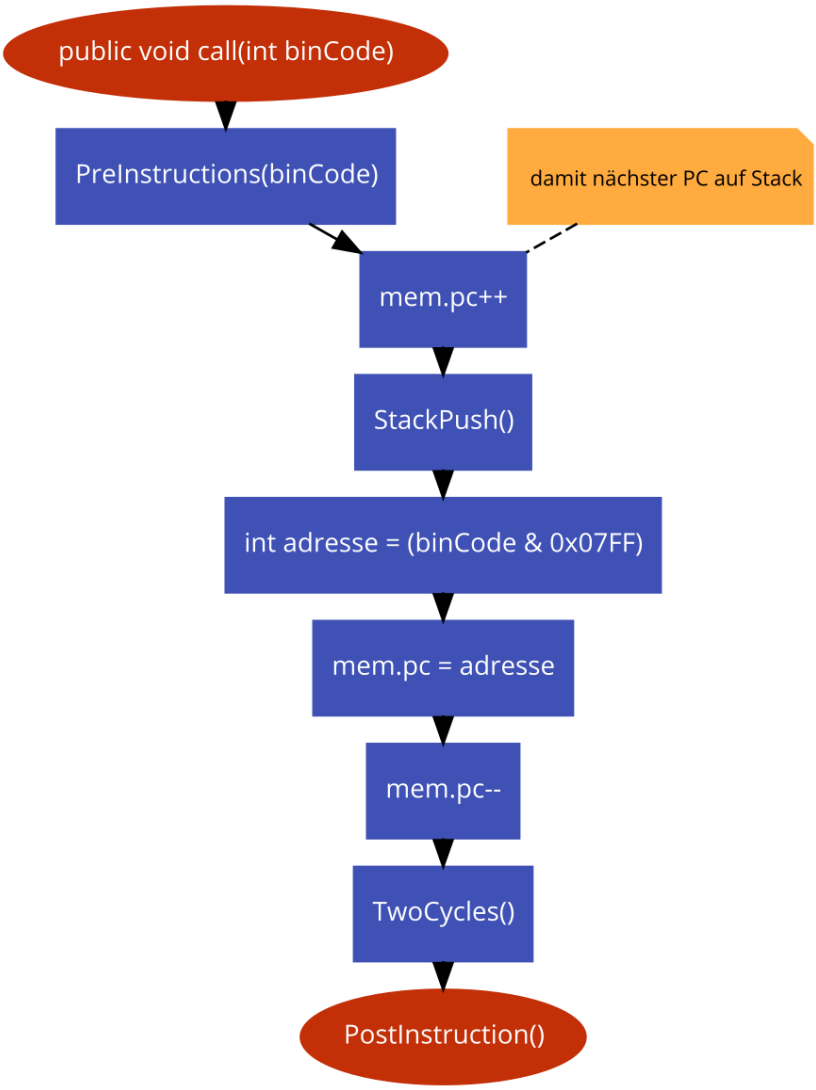


Abbildung 15: PAP CALL

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();
}
```

7.4 MOVF (Move f)

Syntax:	[*label*] MOVF f,d
Operands:	$0 \leq f \leq 127$ $d \in [0,1]$
Operation:	(f) → (destination)
Status Affected:	Z
Description:	The contents of register f are moved to a destination dependant upon the status of d. If d = 0, destination is W register. If d = 1, the destination is file register f itself. d = 1 is useful to test a file register, since status flag Z is affected.

Abbildung 16: MOVF – Auszug PIC

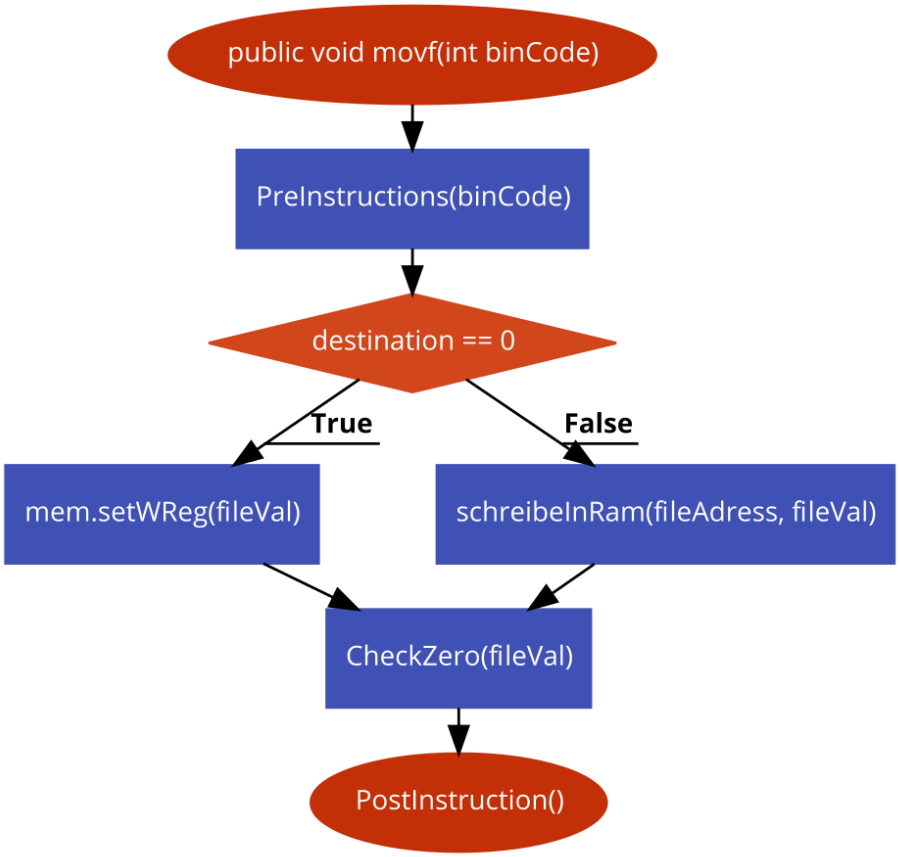


Abbildung 17: PAP MOVF

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();
}
```

7.5 RRF (Rotate Right f through Carry)

Syntax:	[*label*] RRF f,d
Operands:	0 ≤ f ≤ 127 d ∈ [0,1]
Operation:	See description below
Status Affected:	C
Description:	The contents of register 'f' are rotated one bit to the right through the Carry Flag. If 'd' is 0, the result is placed in the W register. If 'd' is 1, the result is placed back in register 'f'.

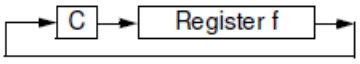


Abbildung 18: RRF - Auszug PIC

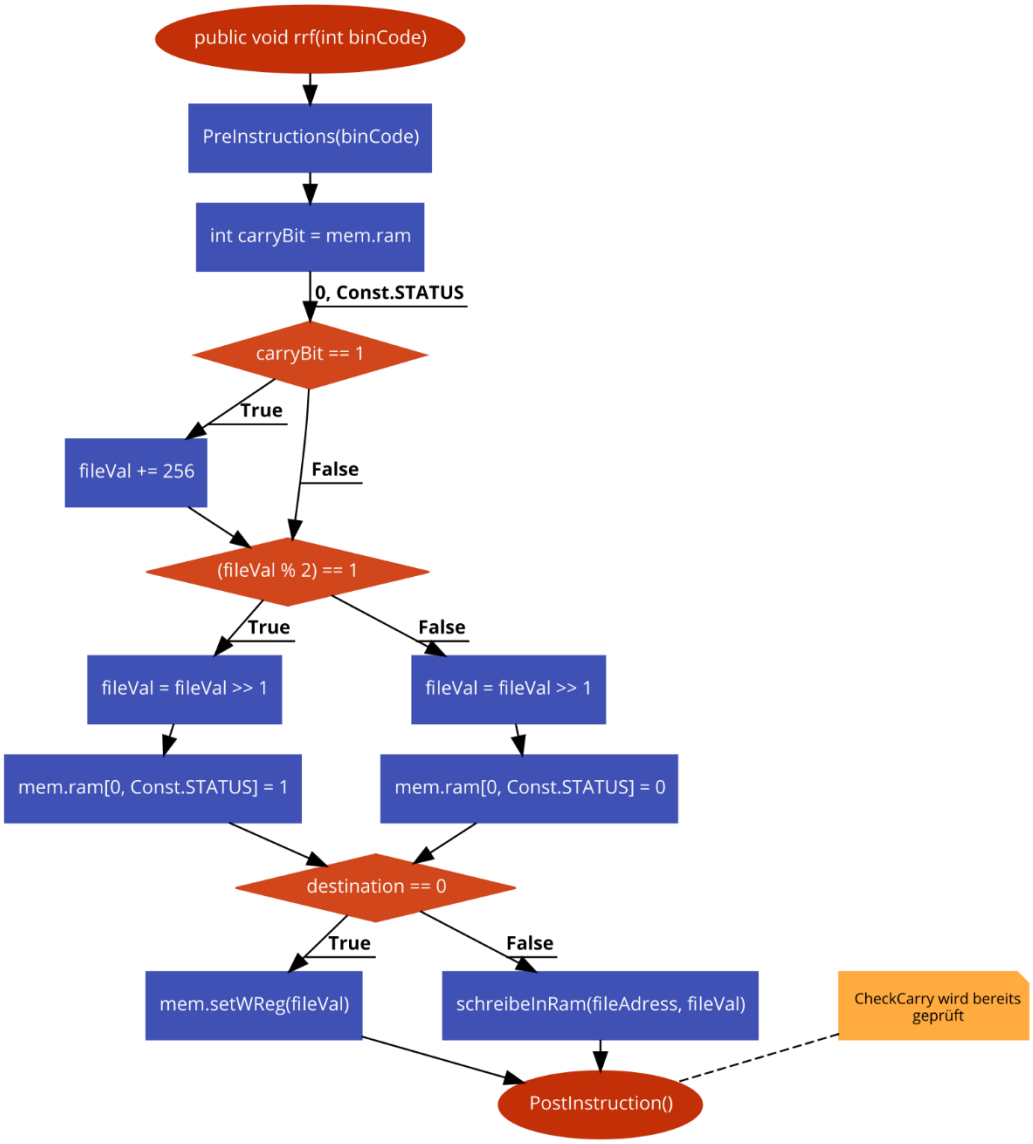


Abbildung 19: PAP RRF

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();
 }
```

7.6 SUBWF (Subtract W from f)

Syntax:	[*label*] SUBWF f,d
Operands:	$0 \leq f \leq 127$ $d \in [0,1]$
Operation:	(f) - (W) → (destination)
Status Affected:	C, DC, Z
Description:	Subtract (2's complement method) W register from register 'f'. If 'd' is 0, the result is stored in the W register. If 'd' is 1, the result is stored back in register 'f'.

Abbildung 20: SUBWF - Auszug PIC

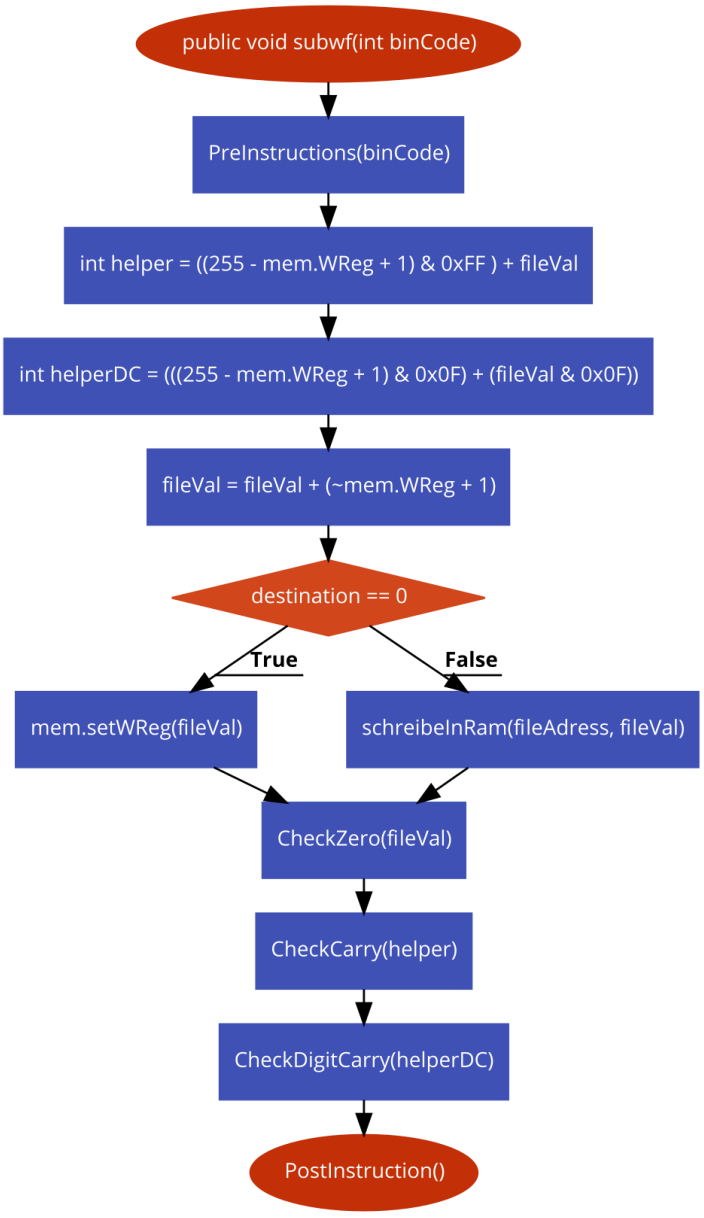


Abbildung 21: PAP SUBWF

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();
 }
```

7.7 DECFSZ (Decrement f, Skip if 0)

Syntax:	[*label*] DECFSZ f,d
Operands:	$0 \leq f \leq 127$ $d \in [0,1]$
Operation:	(f) - 1 → (destination); skip if result = 0
Status Affected:	None
Description:	The contents of register 'f' are decremented. If 'd' is 0, the result is placed in the W register. If 'd' is 1, the result is placed back in register 'f'. If the result is 1, the next instruction is executed. If the result is 0, then a NOP is executed instead, making it a 2TCY instruction.

Abbildung 22: DECFSZ - Auszug PIC

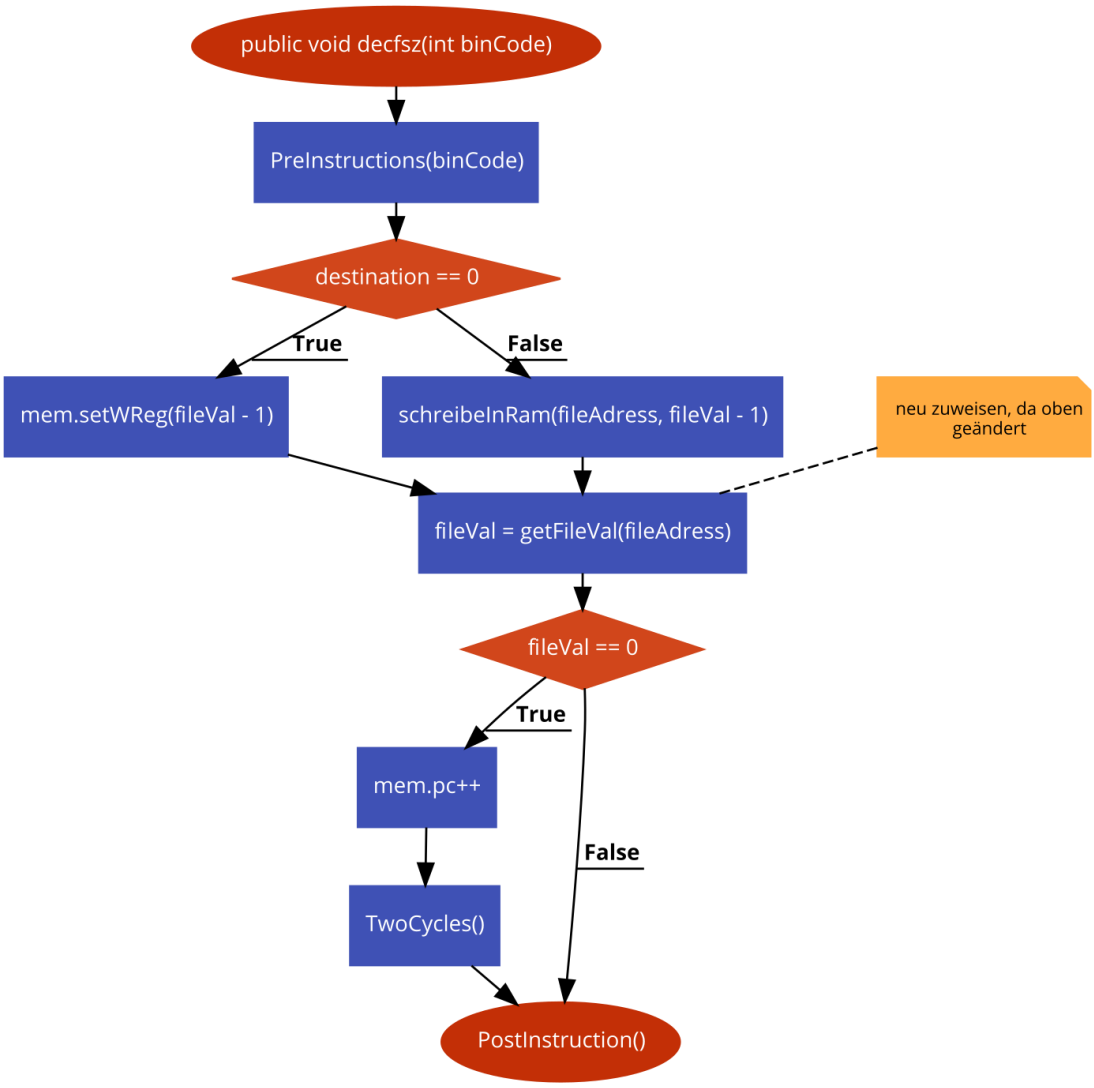


Abbildung 23: PAP DECFSZ

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();
 }
```

7.8 XORLW (Exclusive OR Literal with W)

Syntax:	[*label*] XORLW k
Operands:	$0 \leq k \leq 255$
Operation:	(W) .XOR. k → (W)
Status Affected:	Z
Description:	The contents of the W register are XOR'ed with the eight-bit literal 'k'. The result is placed in the W register.

Abbildung 24: XORLW - Auszug PIC

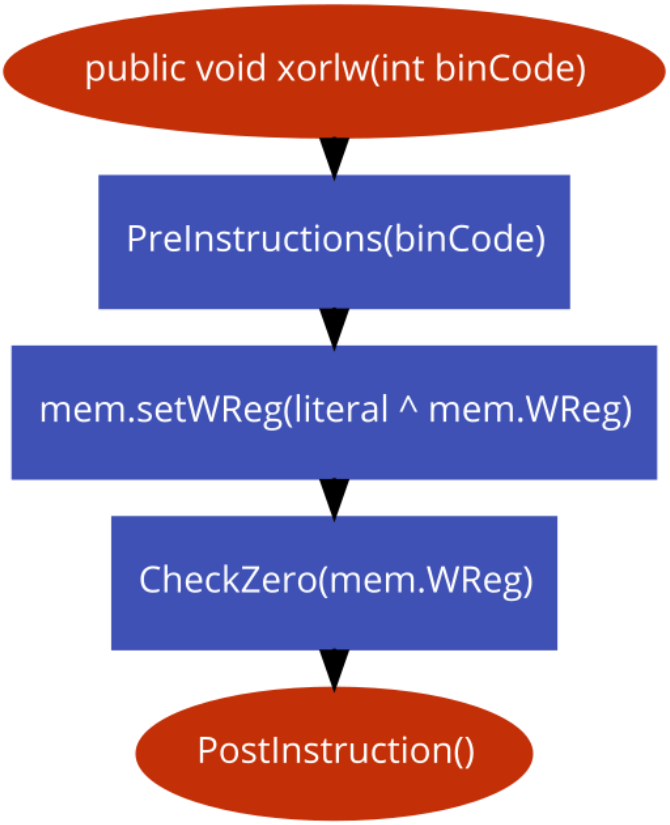


Abbildung 25: PAP XORLW

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();
}
```

7.9 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 abgeprüft.

FIGURE 6-10: INTERRUPT LOGIC

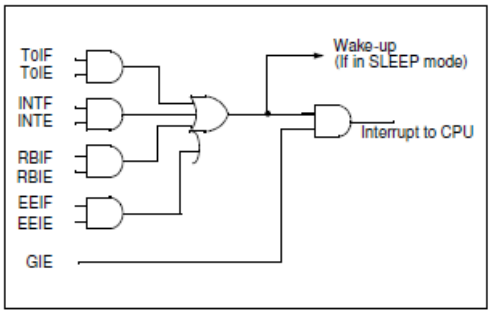


Abbildung 26: Interrupt Logik – Auszug PIC

Im der nebenstehenden Abbildung 26 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.

8. 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.

9. 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. Lediglich eine Verbesserung der Interrupts und die Implementation der EEPROMS stehen noch offen.

Die Wahl der Programmiersprache bereuen wir nicht, da C# mit Visual Studio durch die Windows-Forms 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 umzustrukturieren.

Literaturverzeichnis

Dokumentation des PIC16F84

https://de.wikipedia.org/wiki/Simulation