Themenblatt

Projekt: PIC-Simulator

Allgemeines:

In dieser Vorlesung oder Praktikum soll ein Simulator für den Mikrocontroller PIC 16F84 geschrieben werden.

Als Grundlage dient dazu in erster Linie Das Datenblatt des Herstellers.

In diesem Themenblatt werden einige Inhalte dieses Datenblattes ergänzend erklärt.

Wichtig!
Das Datenblatt PIC16F84 Version DS30430C von 1998 beinhaltet bei der Befehlsbeschreibung der Subtraktionsbefehle Fehler bei der Beschreibung des Carry- und DC-Flags. Vom Fehlerteufel sind auch die Befehle DECFSZ, INCFSZ, IROLW und IORF betroffen. In der in Moodle eingestellten Version sind diese Fehler kenntlich gemacht und richtig gestellt.

noch unvollständig

Inhaltsverzeichnis

1. Das Projekt: PIC16F84 Simulator

Dieses Projekt wurde entwickelt um das Verständnis für die Funktionsweise eines Mikroprozessors oder Mikrocontrollers zu vertiefen. Gleichzeitig bot sich die Möglichkeit die im Studium gelernte Hochsprache (i.d.R. Java oder C) anzuwenden und zu verfestigen. Durch den großen Umfang dieses Projekts kommen sehr viele Kenntnisse der Hochsprache auf den Prüfstand. Die Studenten erkennen selbst, wo noch Defizite sind bzw. sind gezwungen diese aufzuarbeiten um das Projekt erfolgreich abschließen zu können.

Dieses Projekt wird seit über 15 Jahren an verschiedenen Fakultäten durchgeführt. Dabei sind die Ergebnisse zum Teil hervorragend, denn dieses Projekt hat für viele Studenten einen Wettkampfcharakter. Sei es mit anderen Gruppen, wer schafft was oder der Wettkampf gegen sich selber nach dem Motto „Das schaffe ich auch!“.

Wichtig zum erfolgreichen Gelingen dieses Projekts ist eine genügend fundierte Kenntnis der verwendeten Programmiersprache[1]. Das sollte den Studenten bereits zum Zeitpunkt der Vorlesung „Programmieren in JAVA[2]“ bekannt sein. Am Besten geschieht dies in Form eines Übersichtsblatt. So können sie die Vorlesungsinhalte besser einordnen und zielorientierter lernen.

1 Manche Programmiercracks erlernen aus eigenem Antrieb dafür noch eine neue Programmiersprache.
2 Oder welche Sprache auch immer gelehrt wird.

2. Was ist ein PIC-Simulator?

Mit einem Simulator, egal ob ein Mikrocontroller oder ein Automobil-Crashtest simuliert werden soll, versucht man die Wirklichkeit so gut wie möglich mittels Algorithmen nachzubilden. Je nach dem wie exakt diese Rechenregeln sind und wie viel der möglichen Randbedingungen berücksichtigt werden, wird das Ergebnis der Simulation mehr oder weniger gut.

Bei der extrem schwierigen Wettervorhersage kommt diese Problematik am Besten zum Vorschein. Je weiter der Blick in die Zukunft erfolgt, desto unschärfer werden die Ergebnisse. Im Prinzip werden hunderte verschiedener Rechenmodelle (Algorithmen und Randbedingungen) durchgerechnet und miteinander verglichen. Hinzu kommen die Vergleiche mit früheren Wettersituationen und deren Weiterentwicklung. All das wird dann zusammengeführt und bildet die Vorhersage, wie das Wetter in den nächsten Tagen wahrscheinlich wird.

Vergleichsweise bessere Ergebnisse liefern die Simulationen von Crashtests. Hier sind die Einflussfaktoren deutlich geringer als bei der Wettervorhersage. Auch die Algorithmen sind wesentlich besser ausgearbeitet.

In diesem Projekt soll der Mikrocontroller PIC 16F84 simuliert werden. Dabei soll die Funktionalität und nicht die internen Strukturen nachgebildet werden. Wenn dieser simulierte Mikrocontroller ein Programm abarbeitet, soll an den virtuellen Ein- und Ausgängen das gleiche Verhalten vorliegen wie bei einem echten Controller dieses Typs. Die Simulation soll auch nur die logische und nicht die elektrische Ebene nachbilden. Somit sind Einschwingverhalten, Schaltschwellen, ein realer Zeitbezug, Temperaturverhalten, Stromverbrauch, Verlustleistung, Spannungsschwankungen usw. in diesem Projekt vernachlässigbar.

2.1. Die Bedienung des Simulators

Die Testprogramme sind teilweise so aufgebaut, dass sie nur dann richtig ausgeführt werden, wenn an dem einen oder anderen I/O-Pin ein Pegelwechsel auftritt. Dieser Pegelwechsel sollte per Maus oder Tastatur durchführbar sein.

Da jeder Computer und damit auch ein Mikrocontroller nach dem EVA-Prinzip arbeitet, muss auch die Ausgabe auf irgendeine Weise visualisiert werden. Im einfachsten Fall, wird der aktuelle Ausgangspegel eines I/O-Pins durch eine 0 bzw. 1 oder unterschiedliche Farben gekennzeichnet.

Neben der Stimulierung der I/O-Pins sollte auch ein schrittweise Abarbeitung des Programms möglich sein. Somit wären Schaltflächen mit den Funktionalitäten RUN, STEP, RESET und LOAD das Minimum was notwendig ist.

Damit man ein Programm austesten kann, d.h. man prüft es auf logische Fehler, sind die Darstellung des Quelltextes mit Markierung des aktuellen Befehls sicher sinnvoll. Ebenso die Darstellung der Register und deren Inhalte ist unbedingt erforderlich.

Von Vorteil ist auch eine Anzeige der Zeit (Laufzeit), die dieses Programm in Echtzeit benötigen würde. Dies ist von der Quarzfrequenz und der Art der Befehle abhängig. So wird bei einem 4 MHz Quarz ein Befehl in 1 µs abgearbeitet. Bei 1 MHz Quarzen benötigt der gleiche Befehl dagegen 4 µs. Man unterscheidet zwischen Quarztakt und Befehlstakt. Beim PIC benötigen alle, bis auf wenige Ausnahmen, vier Quarztakte für einen Befehlstakt.

Soll der Watchdog implementiert werden, dann muss dieser über eine eigene Schaltfläche ein- und ausschaltbar sein. Im Original ist dieser Schalter nur bei der Programmierung des Bausteins aktivierbar und nicht während der Laufzeit. Diese Schaltfläche ist somit das entsprechende Bit im Konfigurationswort.

Die verschiedenen Testprogramme prüfen die einzelnen Befehle und Funktionen des Controllers. Je nach Schwierigkeitsgrad werden die Testprogramme mit unterschiedlichen Punkten bewertet. Falls ein Testprogramm nur in Teilen funktioniert, wird auch nur ein Teil der Punkte gutgeschrieben.

Nicht geprüft werden die Funktionen des Power-Up-Timers, des Oszillator-Startup-Timer, der „2 cycles delay function“ beim Timer und der einstufigen Pipeline.

3. Testprogramme

Die Funktion des PIC-Simulators wird mit Hilfe mehrerer Testprogramme geprüft. Sie decken den Großteil der PIC16F84 Funktionalität ab. Die Testprogramme enthalten alle Informationen die der Simulator zur Befehlsausführung benötigt. Es handelt sich dabei um eine Textdatei mit der Dateiendung LST.

3.1. Aufbau einer Testdateien

In Listing 1 ist ein einfaches Beispiel einer dieser Testdateien zu sehen.

```
                    00001           ;TPicSim1
                    00002           ;Programm zum Test des 16F84-Simulators.
                    00003           ;Es werden alle Literal-Befehle geprüft
                    00004           ;(c) St. Lehmann
                    00005           ;Ersterstellung: 23.03.2016
                    00006           ;mod. 18.10.2018 Version HSO
                    00007           ;
                    00008           list c=132          ;Zeilenlänge im LST auf 132
                                                             Zeichen setzen
                    00009
                    00010
                    00011           ;Definition des Prozessors
                    00012           device 16F84
                    00013
                    00014           ;Festlegen des Codebeginns
                    00015           org 0
                    00016   start
0000 3011           00017           movlw 11h           ;in W steht nun 11h, Statusreg.
                                                             unverändert
0001 3930           00018           andlw 30h           ;W = 10h, C=x, DC=x, Z=0
0002 380D           00019           iorlw 0Dh           ;W = 1Dh, C=x, DC=x, Z=0
0003 3C3D           00020           sublw 3Dh           ;W = 20h, C=1, DC=1, Z=0
0004 3A20           00021           xorlw 20h           ;W = 00h, C=1, DC=1, Z=1
0005 3E25           00022           addlw 25h           ;W = 25h, C=0, DC=0, Z=0
                    00023
                    00024
                    00025   ende
0006 2806           00026           goto ende           ;Endlosschleife, verhindert
                                                             Nirwana
                    00027
                    00028
```

Listing 1: Testprogramm 1 (TPicSim1.LST)

Die Einteilung der zu extrahierenden Teile ist Spalten orientiert. Diese sind teilweise in hexadezimaler Notation geschrieben und haben folgende Bedeutung:

Spalte 1	4-stellig	Adresse im Programmspeicher an der der Befehl steht
Spalte 2	4-stellig	Befehl mit Befehlscode und ggf. weiteren Argumenten
Saplte 3	5-stellig	laufende Zeilennummerierung (dezimal)
Spalte 4		Label und Symbole aus dem Assemblerprogramm
Spalte 6		Mnemonische Befehle und Argumente
Spalten unabhängig		Kommentare werden mit einem Semikolon eingeleitet

Für den Simulator sind nur die Spalten 1 und 2 wichtig. Hier sind alle notwendigen Informationen des Befehls beschrieben. Im Assemblerprogramm können dagegen Symbole und Labels benutzt werden, die anfangs keinen absoluten Bezug zu den Adressen haben. In Listing 1 steht in Zeile 26 das Label *ende*. Der Controller kann mit dieser Zeichenkette aber nichts anfangen. Er benötigt die absolute Adresse des Ziels. Diese Adresse berechnet der Assembler und fügt sie dem Befehlscode hinzu.

Befehlscode:	`10 1kkk kkkk kkkk`	
Absolutadresse des Labels	`000 0000 0110`	
Gesamter Befehl	`10 1000 0000 0110`	= 2806

3.2. Hinweis für das Simulatorprojekt

Sie müssen somit die Inhalte der einzelnen Zeilen dieser Testdatei so zerlegen, dass Sie die gewünschte Information (Adresse und Befehl) erhalten. Diese werden dann in einem Integer-Array, welches den Programmspeicher darstellt, abgelegt.

4. Der Programm- und Datenspeicher

Wie im Originalcontroller benötigt der Simulator auch einen Programm- und Datenspeicher. Diese Speicher sollen so aufgebaut sein, dass man über einen Zeiger auf die einzelnen Elemente zugreifen kann. Dies lässt sich am einfachsten durch ein Array realisieren.

4.1. Abbild des Programmspeichers

Der Programmspeicher des 16F84 ist 1024 x 14 Bit groß. Damit wird ein Array der Größe 1024 benötigt. Sinnvollerweise wählt man das Variablentyp einen Integer, da auch im Controller nur Binärzahlen im Programmspeicher stehen. Der Variablentyp Integer wird Rechner intern immer als Binärzahl abgespeichert.

Ein Integer bietet auch den Vorteil, dass aus dem Gemisch aus Befehlscode und Argumenten mittels einfacher logischer Verknüpfung mit den passenden Masken die einzelnen Teile sicher getrennt werden können (siehe dazu ..).

Je nach verwendeter Programmiersprache muss die Größe des Integertyps beachtet werden. So sind bei einem 16-Bit Integer die oberen 2 Bits in diesem Array immer 0. Bei einem 32-Bit Integer sind es die oberen 18 Bits. Es kann durchaus vorkommen, dass man die oberen Bits per UND-Verknüpfung auf 0 setzten muss, damit keine Fehler entstehen (z.B. bei der Bildung des Komplements).

4.2. Abbild des Datenspeichers

Der PIC hat beim RAM-Speicher eine Datenbreite von 8 Bits. Dennoch sollte man dieses Array auch mittels Integer realisieren. Der Grund wird unter .. besprochen.

Der RAM-Speicher des PIC-Controllers ist in Bänke aufgeteilt. Jede Bank umfasst maximal 128 Adressen, wobei sich nicht unbedingt hinter jeder Adresse ein physikalischer Speicher befindet. Manche Inhalte sind sowohl auf Bank 0 als auch auf Bank 1 zu finden. Das nennt man Adressspiegelung.

Der Zugriff auf die einzelnen Bänke wird mit dem RP0-Bit im Statusregister gesteuert. Bei den Befehlen ist die Adresse für den direkten Speicherzugriff lediglich 7 Bit lang, was genau der Größe einer Bank entspricht. Soll ein Zugriff innerhalb der Bank 0 erfolgen, muss das RP0-Bit auf 0 stehen, beim Zugriff auf

Bank ein steht in RP0 eine 1.

4.3. Warum kein Byte-Typ?

Ein Byte umfasst 8 Bits. Die Daten im Datenspeicher können mittels arithmetischer und logischer Befehle verknüpft werden. Dabei treten u.U. Überläufe des Zahlenbereichs auf. Verwendet man statt Byte Integertypen, so kann das Ergebnis größer als 255 werden. Diese Zahl ist auch die Grenze die beim Überschreiten das Carryflag setzt. Es reicht somit eine Überprüfung auf *größer 255* um den Wert des Carryflags zu kennen. Am Ende wird einfach eine Maske mit 255 über diese Variable gelegt und der 8-Bit Wert ist wieder garantiert.

Beispiel:

	0000 0000 0001 0001
+	0000 0000 1111 1011
Übertrag	*0000 0001 1110 0110*
16-Bit Ergebnis	0000 0001 0000 1100

Dieser Übertrag in das obere Byte (MSB) der 16-Bit Zahl entspricht dem Carryflag.

5. Der Stack

Der PIC-Controller besitzt einen 8-stufigen Stack. Dieser kann nicht per Befehl beeinflusst werden. Er dient lediglich für die Speicherung der Rückkehradresse bei Unterprogrammaufrufen. Der Stack wird über einen unsichtbaren Stackpointer verwaltet. Auch auf dieses Register ist kein direkter Zugriff möglich. Der Stackpointer selbst umfasst lediglich 3 Bits, was für die oben genannten 8 Einträge ausreicht.

5.1. Die Funktionsweise des Stacks

Nachdem bei einem CALL-Befehl die aktuelle Rückkehradresse auf den Stack gelegt wurde, wird der Stackpointer um eins erhöht. Bei einem RETURN, RETLW oder RETFIE wird zuerst der Stackpointer um eins erniedrigt und dann der Stackeintrag gelesen.

Werden mehr als acht CALLs geschachtelt, wird die Rückkehradresse des neunten CALLs die des ersten CALLs überschreiben. Das bedeutet: erreicht der Stackpointer den Wert 7 und wird nochmals inkrementiert, dann kommt er zum Wert 0. Man spricht hier von einem Ringpuffer.

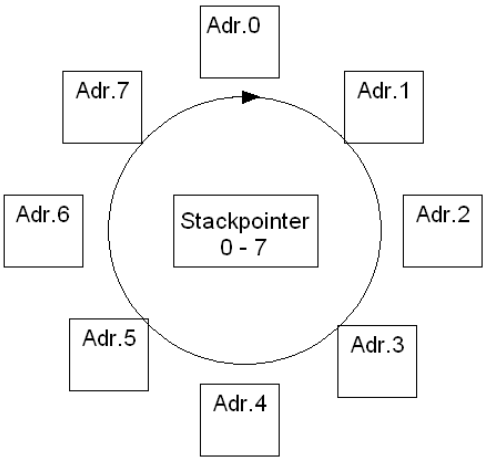

Abbildung 1: Der Stack als Ringpuffer

6. Die Flags im Statusregister

Im Statusregister gibt es drei wichtige Flags (Zustandsbits) die vom Mikrocontroller in Abhängigkeit des Ergebnisses einer Operation gesetzt oder zurück gesetzt werden.

Nachdem ein Befehl vom Mikrocontroller abgearbeitet wurde, weiß der Controller nichts mehr von diesem Befehl. Einzig die Statusbits bilden sozusagen eine Verbindung in die Vergangenheit. An Hand dieser Flags können in Abhängigkeit des Programmverlaufs Entscheidungen getroffen werden. Hat beispielsweise ein Zähler den Stand 0 erreicht, wird dies durch das Zeroflag signalisiert. D.h. der Controller setzt in diesem Fall das Z-Flag im Statusregister.

Neben dem Zeroflag gibt es auch das Carry- und das Digitcarryflag. Alle drei werden nachfolgend genauer beschrieben.

Nicht jeder Befehl beeinflusst die Flags. Welche dies sind, ist in der Befehlsübersicht dokumentiert.

Die Flags bleiben unverändert, bis ein entsprechender Befehl sie anpasst.

6.1. Das Z-Flag (Zeroflag)

Das Z-Flag gibt an, ob das Ergebnis einer Operation Null ist. In diesem Fall wird das Z-Flag auf 1 gesetzt, in allen anderen Fällen auf Null.

Beispiel:
(Inhalt der Variablen COUNTER sei 3)

```
DECF    COUNTER         ;counter = 2, Z-Flag = 0
DECF    COUNTER         ;counter = 1, Z-Flag = 0
DECF    COUNTER         ;counter = 0, Z-Flag = 1
DECF    COUNTER         ;counter = 255, Z-Flag = 0
```

Hinweis:
Im manchen Datenblättern sind in der Befehlsbeschreibung Fehler vorhanden. Ggf. muss man in anderen Datenblättern (vorherige oder folgende Revision) zu Rate ziehen.

6.2. Das C-Flag (Carryflag)

Das C-Flag gibt einen Zahlenüberlauf bzw. -unterlauf an. Bei Rechnern, dazu gehört auch ein Mikrocontroller, ist der Zahlenbereich mit dem die Maschine rechnet begrenzt. Während die Mathematiker mit Zahlen zwischen -Unendlich und +Unendlich arbeiten, ist der Zahlenbereich bei Rechner einschränkt.

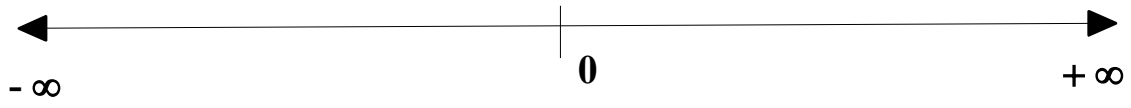

Abbildung 2: Die Mathematiker arbeiten mit einem Zahlenstrahl.

Übliche Zahlenbereiche sind 8-Bit, 16-Bit, 32-Bit oder 64-Bit. Mit 8-Bit lassen sich Zahlen zwischen 0 und 255 binär codieren.

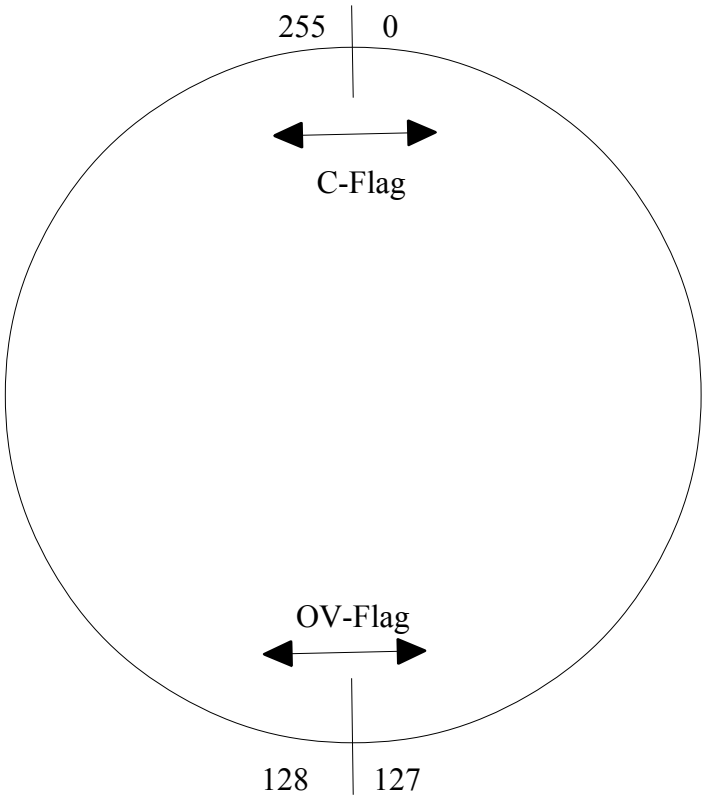

Abbildung 3: Rechner arbeiten dagegen mit einem Zahlenkreis.
(Das OV-Flag gibt es beim PIC 16Xxx nicht)

Ergibt sich bei der Addition eine Zahl, die die obere Grenze überschreiten muss, setzt der Controller das C-Flag.

Beispiel:

```
MOVLW   240
MOVWF   COUNTER          ;in Speicherplatz COUNTER 240 laden
MOVLW   15
ADDWF   COUNTER,W        ;Ergebnis in W = 255 => C-Flag = 0
                         ;und zusätzlich wird Z-Flag = 0
```

aber:

```
MOVLW   240
MOVWF   COUNTER          ;in Speicherplatz COUNTER 240 laden
MOVLW   16
ADDWF   COUNTER,W        ;Ergebnis in W = 0 => C-Flag = 1
                         ;und zusätzlich wird Z-Flag = 1
```

aber:

```
MOVLW   240
MOVWF   COUNTER          ;in Speicherplatz COUNTER 240 laden
MOVLW   17
ADDWF   COUNTER,W        ;Ergebnis in W = 1 => C-Flag = 1
                         ;und zusätzlich wird Z-Flag = 0
```

VORSICHT – PIC-Mikrocontroller !!!

6.3. Das seltsame Verhalten des Carryflags

Das Carryflag soll bei arithmetischen Befehlen den Bereichsüberlauf anzeigen. Neben der Addition gilt dies auch für die Subtraktion. Der PIC[3] setzt das Carryflag beim Subtraktionsbefehl falsch.

So sollte es sein:
Subtrahiert man von der Zahl 1 die Zahl 5, so wird das Ergebnis negativ. Da der

3 Die meisten anderen Prozessoren machen es dagegen richtig.

PIC nur mit vorzeichenlosen Ganzzahlen arbeitet, ist das Ergebnis 252 (0FCH). Das Ergebnis dieser Subtraktion ist somit größer als der Minuend. Man überschreitet die obere Grenze in Abbildung 3 was das Setzen des Carryflags erzwingt.

Beim PIC ist es so:
Die Subtraktion wird mittels Addition des Zweierkomplements[4] durchgeführt. Damit sieht obige Beispielrechnung wie folgt aus:

```
      0000 0001
-     0000 0101 → 1111 1010 (1er-Kompl.)  →1111 1011 (2er-Kompl.)
```

wird zu

	0000 0001	
+	1111 1011	
Übertrag	0 0000 011	linke Stelle im Übertrag ist das Carry
Ergebnis	1111 1100	

Nun sollte der PIC das Carryflag invertieren, da die Operation eine Subtraktion war. Dieses Invertieren haben die Entwickler einfach übersehen[5].

Beim nächsten Zahlenbeispiel wird von der Zahl 5 die Zahl 3 abgezogen. Da hier das Ergebnis im gültigen Zahlenbereich bleibt (es wird keine Grenze überschritten) müsste das Carry Null sein. Beim PIC ist es jedoch gesetzt.

3 = 0000 0011 1111 1100 (1er-Kompl.), 1111 1101 (2er-Kompl.)

	0000 0101	
+	1111 1101	
Übertrag	1 1111 101	linke Stelle im Übertrag ist das Carry
Ergebnis	0000 0010	

Hier ist, obwohl das Ergebnis richtig ist, das Carryflag gesetzt. Es fehlt auch bei diesem Beispiel das Invertieren des Carryflags im Falle einer Subtraktion.

4 Was übrigens alle Prozessoren bei Integer-Arithmetik so machen.

5 In alten Datenblättern ist dieser Umstand noch als Fehler (Error) beschrieben worden, in neueren Datenblättern haben die Marketingleute aus dem Fehler ein Feature gemacht und das Carry im Falle einer Subtraktion zum Borrow-Bit (Borgenbit) umdefiniert.

Daher gilt: Wenn der PIC eine Subtraktion durchführt:
Ergebnis 0 oder größer → Carry gesetzt
Ergebnis kleiner 0 → Carry gelöscht

6.4. Das DC-Flag (Digitcarryflag)

Diese Flag wird auch als Halfcarry bezeichnet. Es zeigt einen Überlauf von den unteren vier Bits an. Dies ist besonders dann interessant, wenn man bei einem 8-Bit-Rechner den Zahlenbereich auf 4-Bit reduzieren will. Betrachten wir dazu, etwas abgewandelt, das obigen Zahlenbeispiel unter diesem Aspekt.

```
MOVLW   241
MOVWF   COUNTER         ;in Speicherplatz COUNTER 241 laden
MOVLW   14
ADDWF   COUNTER,W       ;Ergebnis in W = 255 => C-Flag = 0
                        ;DC = 0, Z = 0
```

Die 240 sieht in der Binärschreibweise wie folgt aus: `1111 0001`
Die 15 in binärer Schreibweise: `0000 1110`
Das Ergebnis lautet: `1111  1111`
Dass kein Übertrag von Bit 3 auf Bit 4 erfolgt lässt sich noch deutlicher zeigen, wenn man nur die unteren Bits betrachtet und die oberen per Maske auf 0 setzt.

```
                        0000 0001
                        0000 1110
-------------------------------------
Ergebnis                0000 1111
```

Rechnet man aber:

```
MOVLW   241
MOVWF   COUNTER         ;in Speicherplatz COUNTER 241 laden
MOVLW   15
ADDWF   COUNTER,W       ;Ergebnis in W = 0 => C-Flag = 1
                        ;DC = 1, Z = 1
```

Die 241 sieht in der Binärschreibweise wie folgt aus:	1111 0001
Die 15 in binärer Schreibweise:	0000 1111
Übertrag	1 1111 111
Das Ergebnis lautet:	0000 0000

Dieser Übertrag entspricht dem C-Flag. Dieser entspricht dem DC-Flag.

Addiert man zu einer Zahl den Wert 0, wird das Carry- und DC-Flag zurückgesetzt. Subtrahiert man dagegen von einer Zahl der Wert 0, wird das Carry- und DC-Flag gesetzt. Das kommt daher, dass die Bildung des 2-er Komplements durch das Invertieren erfolgt. Das Hinzuaddieren der Eins wird durch das Setzen es Übertrageingangs am Volladdierer der niederwertigsten Stelle realisiert. In Abbildung 3 ist ein 4-Bit Addierer bzw. Subtrahierer dargestellt. Er besteht aus vier Volladdierer. Die vier Bits von Wert 1 gehen an die jeweiligen E1-Ein-

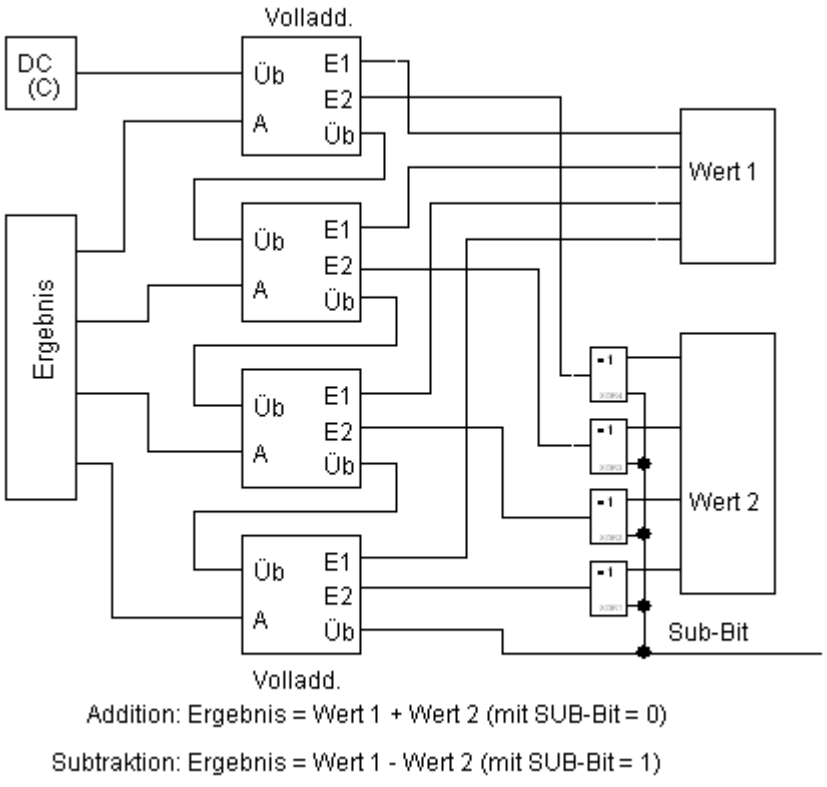

Abbildung 4: 4-Bit-Addier-Subtrahierwerk

gänge. Die Bits von Wert 2 werden über XOR-Gatter geführt. Diese Gatter dienen dazu das Bit ggf. zu invertieren. Liegt an den ÜB-Eingängen eine 0 an, er-

folgt keine Invertierung. Das erkennt man an den beiden ersten Zeilen in Tabelle 1.

In den beiden folgenden Zeilen ist das SUB-Bit 1, was eine Invertierung bedeutet. Ob das SUB-Bit 1 oder 0 ist, hängt vom Befehl ab, der ausgeführt wird. Bei einer Addition wird es auf 0 (→ keine Invertierung) bei einer Subtraktion auf 1 (→ Invertierung) gesetzt.

Eingang x		**SUB-Bit**	**Ausgang x**	**Ergebnis**
0	XOR	0	0	Ausgang = Eingang
1	XOR	0	1	Ausgang = Eingang
0	XOR	1	1	Ausgang = $\overline{\text{Eingang}}$
1	XOR	1	0	Ausgang = $\overline{\text{Eingang}}$

Tabelle 1: Bildung der 1-er Komplements

VORSICHT – PIC-Mikrocontroller !!!

6.5. Das seltsame Verhalten des Digitcarryflags

Auch das Digitcarry wird bei einer Subtraktion vom PIC-Controller falsch gesetzt. Analog zum Carryflag haben die Entwickler das Invertieren des Digitcarrys im Fall der Subtraktion vergessen.

7. Der Programmzähler

Der PIC hat einen etwas eigenwilligen Programmzähler (PC-Register). Es handelt sich um ein Register mit einem Umfang bis zu 13 Bits. Damit sind maximal 8 k Speicher adressierbar. Bei Controller mit einem kleineren Programmspeicher ist der Umfang des Programmzählers entsprechend reduziert.

Beispiel:

Programmspeicher	Anzahl der Bits im Programmzählers
8k	13
4k	12
2k	11
1k	10
0,5k	9

7.1. Sichtbarkeit des Programmzählers

Der Programmzähler des PICs ist nicht komplett in dem Registersatz sichtbar. Lediglich die unteren acht Bit sind in das Register PCL eingeblendet. Die oberen Bits sind nicht auslesbar oder sonst durch Befehl direkt änderbar.

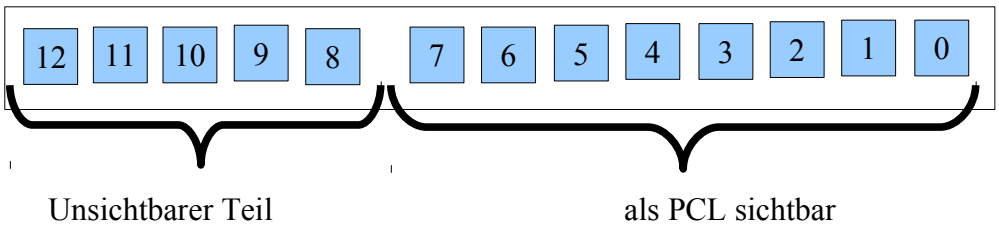

Abbildung 5: Aufbau des Programmzählers

Der obere, unsichtbare Teil des Programmzählers verhält sich je nach Nutzung unterschiedlich, wobei u.U. das PCLATH-Register mitverwendet wird.

7.2. Das Verhalten des PC-Register beim Fetchzyklus

Grundsätzlich gilt, dass nach einem Fetchzyklus der Programmzähler immer um eins erhöht wird. Dabei spielt der aktuelle Inhalt des PC-Registers keine Rolle. Kommt der Zähler allerdings an die individuelle Grenze des Programmspeichers entsteht ein sogenanntes Wrap-Around.

Bei einem Controller mit 1k Programmspeicher ist diese Endadresse 3FFH (= 1023_d)[6]. Holt der Controller an der Adresse 3FEH einen Befehl, wird der Programmzähler auf 3FFH erhöht. Beim nächsten Fetchzyklus springt der Programmzähler auf 000H was einem Programmneustart entspricht. In der Regel ist das Assemblerprogramm jedoch so strukturiert, dass ein solcher Fall nicht eintreten kann.

Fazit:
Bei Fetchzyklen wird der Programmzähler einfach immer um eins erhöht. Erreicht er seine Grenze, beginnt er wieder bei Null.

7.3. Das Verhalten bei Sprungbefehlen

Bei Sprüngen mittels GOTO-Befehl oder Unterprogrammaufrufen mittels CALL-Befehl wird dem eigentlichen Befehlscode eine 11-stellige Zieladresse beigefügt. Diese Zieladresse liest der Controller bei einem Fetchzyklus zusammen mit dem Befehlscode in einem Rutsch ein. Mit einer 11-stelligen Adresse lassen sich alle Adressen in einem 2k großen Speicherblock ansprechen. Bei PIC-Controller mit mehr als 2k Programmspeicher ist dieser Gesamtspeicher in solche 2k großen Blöcke (Page) zerteilt. Ohne eine spezielle Befehlssequenz kann man aus diesem Block nicht heraussspringen. Weder mit GOTO noch mit CALL. Aber ein einfaches Erhöhen nach einem Fetchzyklus überwindet sehr wohl diese 2k Begrenzung.

Um diese Grenze mittels GOTO oder CALL zu überschreiten, muss das PCLATH-Register entsprechend eingestellt werden. Denn dieses Register vervollständigt die als Argument übergebene Zieladresse. Das Ergebnis dieser Erweiterung wird in den Programmzähler geschrieben.

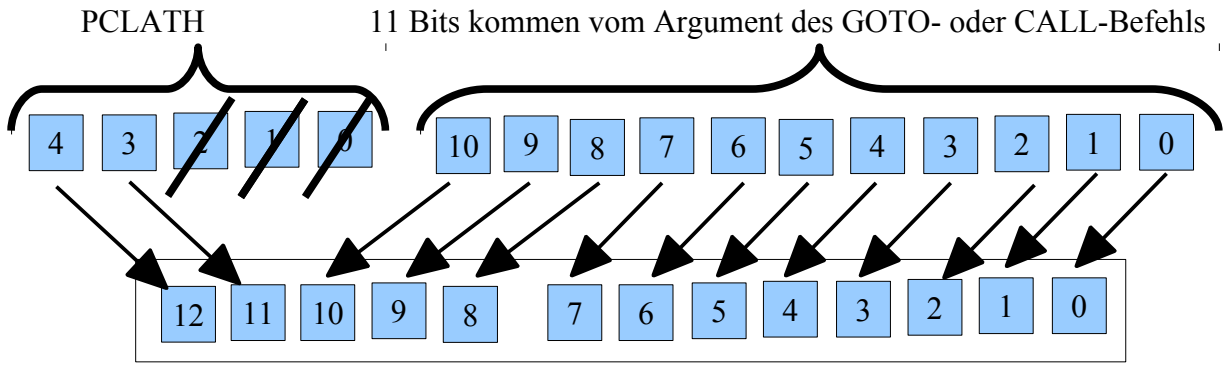

Abbildung 6: Zusammensetzen der Zieladresse bei Sprungbefehlen

6 Die Anzahl der Speicheradressen ist 1024. Da die Zählung aber bei 0 beginnt, ist die höchste Adresse eben 1023d.

In Abbildung 6 sind im PCLATH-Register die Bits 0 bis 2 durchgestrichen. Sie sind in diesem Fall ohne Bedeutung und werden deshalb ignoriert.

7.4. Das Verhalten bei Manipulation des PCL-Registers

Das PCL-Register ist Teil der adressierbaren RAM-Speicher. Es besitzt die Adresse 02H bzw. 82H[7]. Damit kann dieses Register wie die übrigen durch arithmetische und logische Befehle verändert werden. Das Verändern dieses Registers bedeutet aber einen Sprung an eine andere Adresse, da nach der Befehlsabarbeitung ein neuer Fetchzyklus den Befehl holt, dessen Speicherort durch den Programmzähler adressiert wird. Dies erlaubt das einfache Berechnen von Zieladressen im Programmspeicher.

Das PCL-Register verhält sich dabei wie ein 8-Bit Register. Wird also eine Zahl zum PCL-Register addiert, wird dessen Inhalt mit der Zahl in der ALU (8 Bit breit) verknüpft. Ein ggf. auftretender Überlauf erscheint als Carry. Das Ergebnis wird anschließend (je nach Destinationbit) im W-Register oder dem PCL abgelegt.

Beim Zurückschreiben in das PCL-Register wird der neue Programmzählerwert mit Hilfe von PCLATH berechnet.

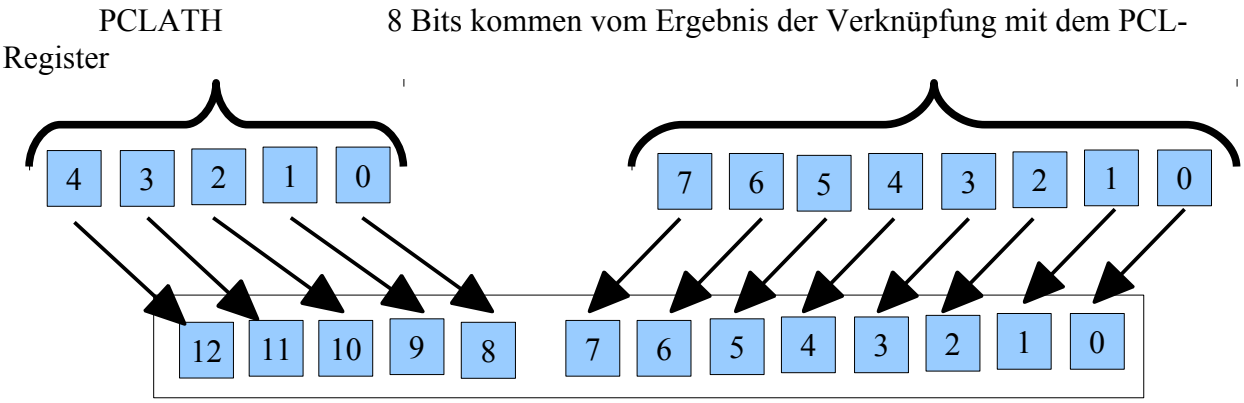

Abbildung 7: Zusammensetzen der Zieladresse bei Manipulation des PCL-Registers

Hier werden alle 5 Bits des PCLATH-Registers in den oberen Teil des Programmzählers übernommen.

7 Es liegt sowohl in der Bank 0 als auch in der Bank 1

7.5. Hinweis für das Simulatorprojekt

Man sollte den Programmzähler als eigenständige Variable im Hintergrund führen und darauf die entsprechenden Operationen durchführen. Am Ende wird dann der untere Teil als PCL visualisiert.

8. Die Ports

Beim PIC-Controller können die einzelnen I/O-Pins unabhängig von einander als Ein- oder Ausgang initialisiert werden. Dazu sind zwei Register vorhanden. Das eigentliche PORT-Register auf Bank 0 und das TRIS-Register auf Bank 1. Das TRIS-Register ist für die Datenrichtung, also Ein- oder Ausgang, zuständig, das PORT-Register für die Signale die ausgegeben bzw. eingelesen werden.

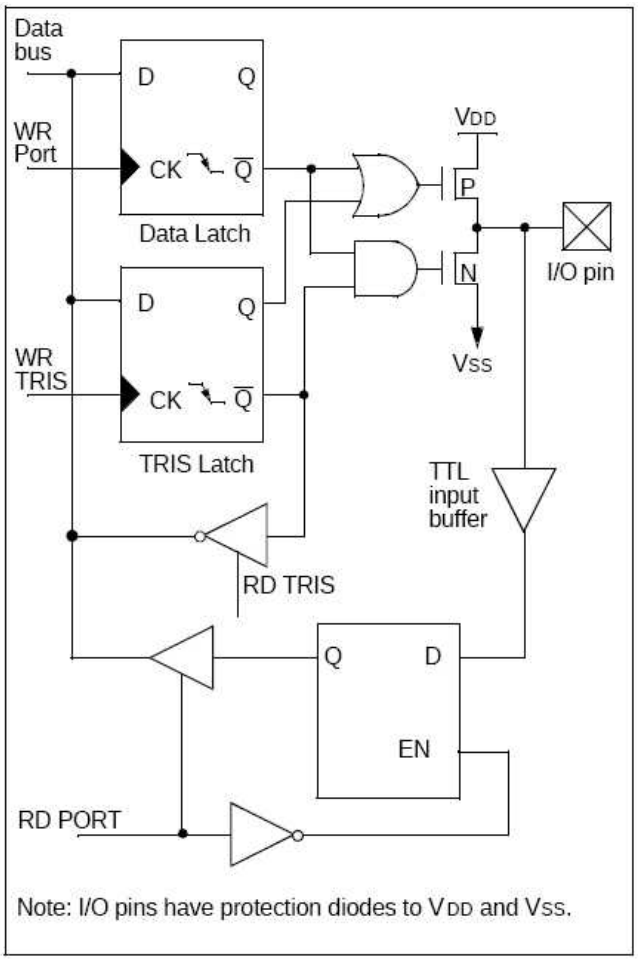

Abbildung 8: Beschaltung eines I/O-Pins

Das Eingangsignal gelangt vom I/O-Pin zum Eingangspuffer und von dort zum D-Eingang eines transparenten Datenlatchs. Das RD PORT-Signal wird durch den entsprechend Lesebefehl (z.B. MOVF port,w) erzeugt. Danach steht der Wert am Ausgang des Flip-Flops und gelangt über einen Puffer zum internen Datenbus.

Beim Schreiben auf den Port gelangt der Wert zum DATA LATCH und wird durch das WR PORT Signal in das Latch übernommen. Dieses Signal wird auch durch einen entsprechenden Schreibbefehl (z.B. MOVWF port) erzeugt.

Das Signal liegt nun am Q-Ausgang statisch an. Ob dieses Signal am I/O-Pin erscheint, entscheidet der Inhalt des TRIS LATCH.

Die beiden Ausgänge des TRIS LATCH werden über ein ODER-Gatter (oben) und ein UND-Gatter (unten) mit dem Q-Ausgang es DATA LATCH verknüpft. Steht im TRIS LATCH eine 1, wird er Ausgang des ODER-Gatters immer 1 sein, egal welcher Wert im DATA-LATCH steht. Diese 1 am Gatterausgang sperrt den oberen p-Transistor[8]. Auch der untere Transistor sperrt, da an dessen Gate eine 0 anliegt, denn das UND-Gatter bekommt aus dem TRIS LATCH eine 0 an den einen Eingang. Das bewirkt immer eine 0 am Ausgang, gleichgültig welche Pegel am anderen Eingang anliegt.

Ist im TRIS LATCH aber eine 0 gespeichert, bestimmt der Q-Ausgang es DATA LATCHs welches Gatter seinen Transistor sperrt und welches den Transistor durchsteuert.

8.1. Das TRIS-Register

Eine 1 im TRIS-Register bedeutet, dass der korrespondierende I/O-Pin als Eingang arbeitet. Bei einer 0 ist dieser Pin ein Ausgang.

Das TRIS-Register kann jederzeit geändert werden. Somit lässt sich per Programm der I/O-Pin zu einem Eingang oder Ausgang machen. Diese Möglichkeit ist wichtig, um bidirektionale Datenübertragung zu realisieren zu können.

8.2. Das PORT-Datenregister

Wird an einer Stelle im PORT-Register eine 1 geschrieben, schaltet der Ausgangstreiber 5V auf den Pin. Bei einer 0 wird der Ausgangspin auf 0V gelegt.

Falls in das PORT-Register geschrieben wird, obwohl die Pins als Eingang arbeiten, gehen diese Ausgabewerte nicht verloren. Sie werden in jedem Fall im Port zwischengespeichert, es kommt aber davon nichts am Pin an. Wird aber dieser I/O-Pin aus Ausgang geschaltet, erscheint die im DATA LATCH gespeicherte Information sofort am Pin.

8.3. Der Portpin RA 4

Dieser Pin besitzt keinen p-Transistor. Er kann somit nur eine 0 aktiv erzeugen.

8 Um einen p-Transistor durchzuschalten, muss sein Gate 0 sein. Der n-Transistor schaltet durch, wenn sein Gate positiv ist.

Eine 1 muss mittels Pullup-Widerstand erzeugt werden. Diese Art der Ausgangsbeschaltung nennt man Open-Drain-Ausgang[9].

Dieser Pin dient auch als Zähleingang für das Timer0-Register (TMR0).

8.4. Der Portpin RB 0

Der RB0-Pin dient auch als Interrupteingang. Dabei ist die wirksame Flanke über das Option-Register wählbar.

8.5. Die Portpins RB 4 bis RB 7

Auch diese vier Pins können einen Interrupt auslösen. Allerdings nur dann, wenn er entsprechende Pin auch als Eingang arbeitet. Die Flanke kann nicht gewählt werden. Sowohl die positive als auch die negative Flanke sind wirksam.

9 Open-Drain ist vergleichbar mit einem Open-Collector bei bipolaren Transistoren.

9. Timer, Watchdog und Vorteiler

Timer, Watchdog und Vorteiler sollten immer gemeinsam betrachtet werden, da der Vorteiler sowohl mit dem Timer als auch dem Watchdog zusammenarbeitet. Weil der Vorteiler nur ein Mal vorhanden ist, kann er nur dem einen oder dem anderen zugewiesen werden.

9.1. Der Timer 0 (TMR0-Register)

Der PIC besitzt ein spezielles Zählregister. Damit lassen sich entweder externe Impulse zählen oder es zählt mit jedem ausgeführten Befehl. Die Auswahl wird über das OPTION-Register festgelegt. In Abbildung 9 sind die möglichen Signalwege dargestellt.

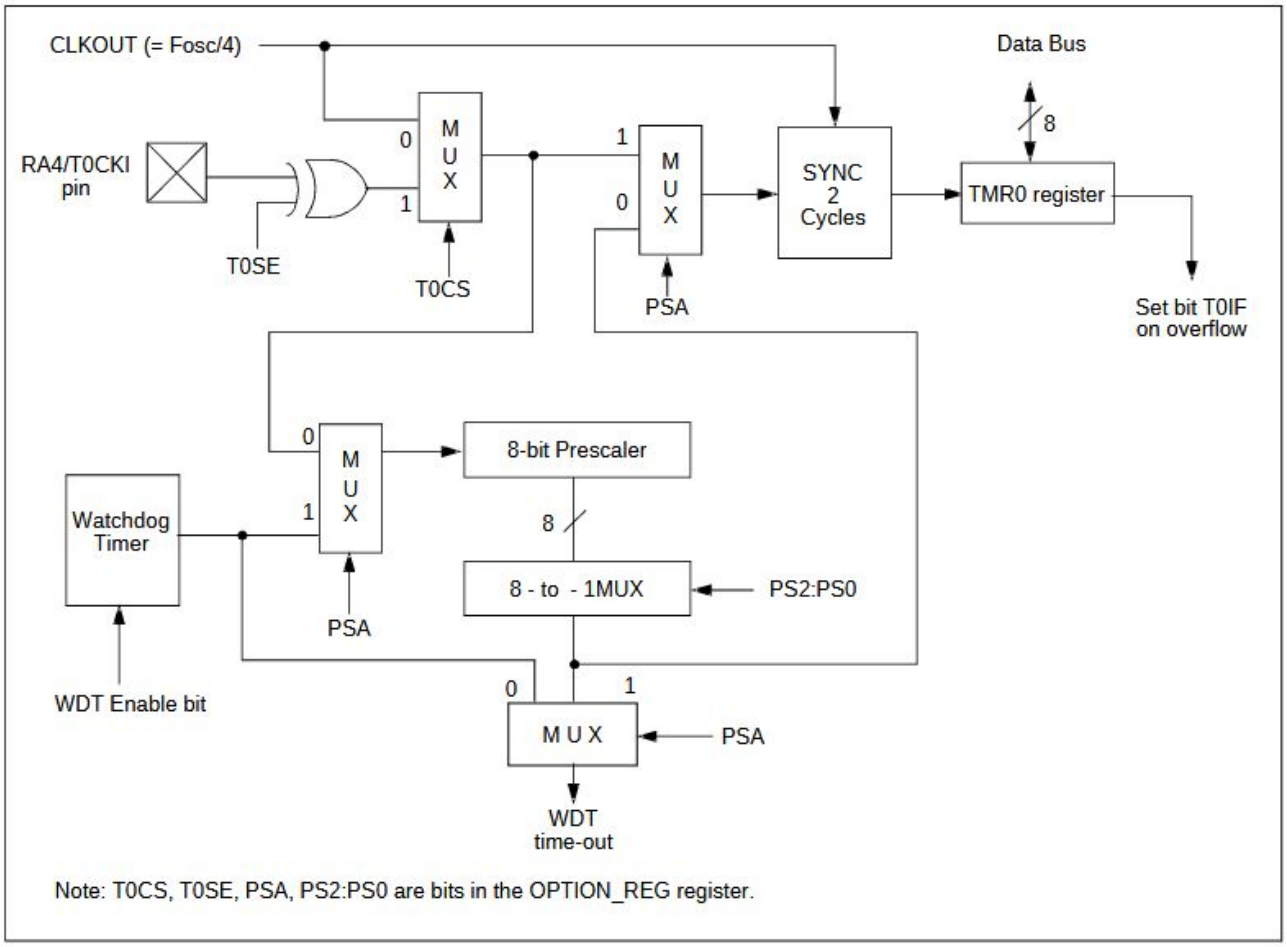

Abbildung 9: Das Timer / Watchdog Modul

Der Timer ist ein 8-Bit Register, das bei einem Zählerüberlauf das T0IF-Bit setzt. Dieses lässt sich per Polling abfragen oder es kann einen Interrupt auslösen.

Soll der Befehlstakt (Fosc/4) zum Zählen genutzt werden, muss das T0CS (Timer 0 Clock Source) auf 0 stehen. Steht hier eine 1, wird der an RA4 anliegende Impuls durch den Multiplexer (MUX1) geführt. Hinter dem MUX1 geht das Signal zum nächsten Multiplexer und gleichzeitig auf den Vorteiler (Prescaler). Dieser Multiplexer entscheidet, ob das Signal direkt auf das Zählregister TMR0 gelangt oder das Signal zuerst den Vorteiler durchlaufen muss. Auch dieses PSA-Bit (Prescaler Assignment Bit) findet man im OPTION-Register. Der Block, der eine Synchronisation durchführt lässt sich nur schwer programmieren. Aus diesem Grund muss dies nicht im Simulator realisiert werden.[10]

Das Timerregister (TMR0) kann wie ein normales Register gelesen und beschrieben werden. Dadurch kann man einen beliebigen Startwert für den Zähler definieren und ihn nach jedem Überlauf neu setzen. Das Beschreiben dieses Registers zieht eine Verzögerung um zwei Befehlstakte nach sich. Das ist bei der Zeitberechnung entsprechend zu berücksichtigen.

Es gibt nur einen Vorteiler. Man kann ihn entweder dem Timer (TMR0-Register) oder dem Watchdog mit Hilfe des PSA-Bits zuordnen. Die Teilerfaktoren unterscheiden sich, je nach dem ob der Vorteiler dem Watchdog oder dem Timer zugewiesen ist.

9.2. Der Watchdog

Der Watchdog ist ein nachtriggerbares Monoflop. Er muss während des Programmierens des Bausteins im Konfigurationsregister aktiviert werden. Die kleinste Zeit für den Watchdog beträgt 18 ms.

Er dient dazu, den zeitlichen Verlauf des laufenden Programms zu überwachen. Dazu setzt man an einer strategisch wichtigen Stelle den Befehl CLRWDT ein. Damit wird, wenn das Programm ordnungsgemäß läuft, der aktivierte Watchdog immer wieder auf den Startwert gesetzt. Kommt das Programm aber außer Tritt oder wenn es in einer Schleife „hängen“ bleibt, fehlt dieses Rücksetzen. Sobald die Zeit des Watchdogs abgelaufen ist, wird ein Reset erzeugt, der den Controller wieder an den Programmanfang bringt. Die 18 ms Watchdogzeit lassen sich durch den Vorteiler – in diesem Fall eigentlich ein Nachteiler – bis zu ca. 2,3 Sekunden verlängern.

In Abbildung 9 ist das Blockschaltbild des Timer/Watchdog Hardwaremoduls zu sehen. Je nach dem wie die Bits im Optionregister gesetzt oder zurückgesetzt sind, verlaufen die Signalwege. Die Blöcke die mit MUX bezeichnet sind, sind

10 Wer es dennoch implementieren will, muss die genaue Funktionalität im Datenblatt nachlesen.

sogenannte Multiplexer. Von zwei Eingängen schalten sie den einen oder anderen auf den Ausgang durch[11].

9.3. Vorteiler beim Watchdog

Bei einem Reset des Controllers sind alle Bits im Optionregister auf 1. Das bedeutet, dass der Vorteiler dem Watchdog zugewiesen ist. Der Teilerfaktor des Watchdogs ist auf den Maximalwert von 128 eingestellt. Damit ergibt sich eine Maximalzeit von 128 * 18ms = 2,3s, in der der Watchdog mit einem CLRWDT zurückgesetzt werden muss, falls er aktiviert wurde. Andernfalls wird ein Reset ausgelöst.

Will man stromsparende Geräte aufbauen, kommt der SLEEP-Befehl zum Einsatz. Dieser Sleepmodus kann durch ein Timeout des Watchdogs beendet werden.

Die Teilerfaktoren (Prescaler Rate Select) des Vorteilers sind folgende:

PS2:0	Teilerfaktor
000	1 : 1
001	1 : 2
010	1 : 4
011	1 : 8
100	1 : 16
101	1 : 32
110	1 : 64
111	1 : 128

Tabelle 2: Teilerfaktoren des Vorteiles zugewiesen zum Watchdog

Der Timer0 wird in diesem Fall durch Impulse am RA 4 I/O-Pin inkrementiert. Die zählende Flanke ist der High-Low-Übergang (T0SE). Die Auswahl trifft ein einfaches XOR-Gatter das als ein- und ausschaltbarer Inverter arbeitet.

Da der Vorteiler dem Timer nicht zur Verfügung steht, zählt dieser bei jeder passenden Flanke um eins hoch. Das entspricht einem Teilerfaktor von 1 : 1.

In Abbildung 10 zeigt der rote Pfad den Signalweg für den Timer. Für den Wat-

11 Es gibt auch Multiplexer, die einen Eingang auf einen von zwei Ausgänge durchschalten können

chdog ist der Verlauf grün dargestellt.

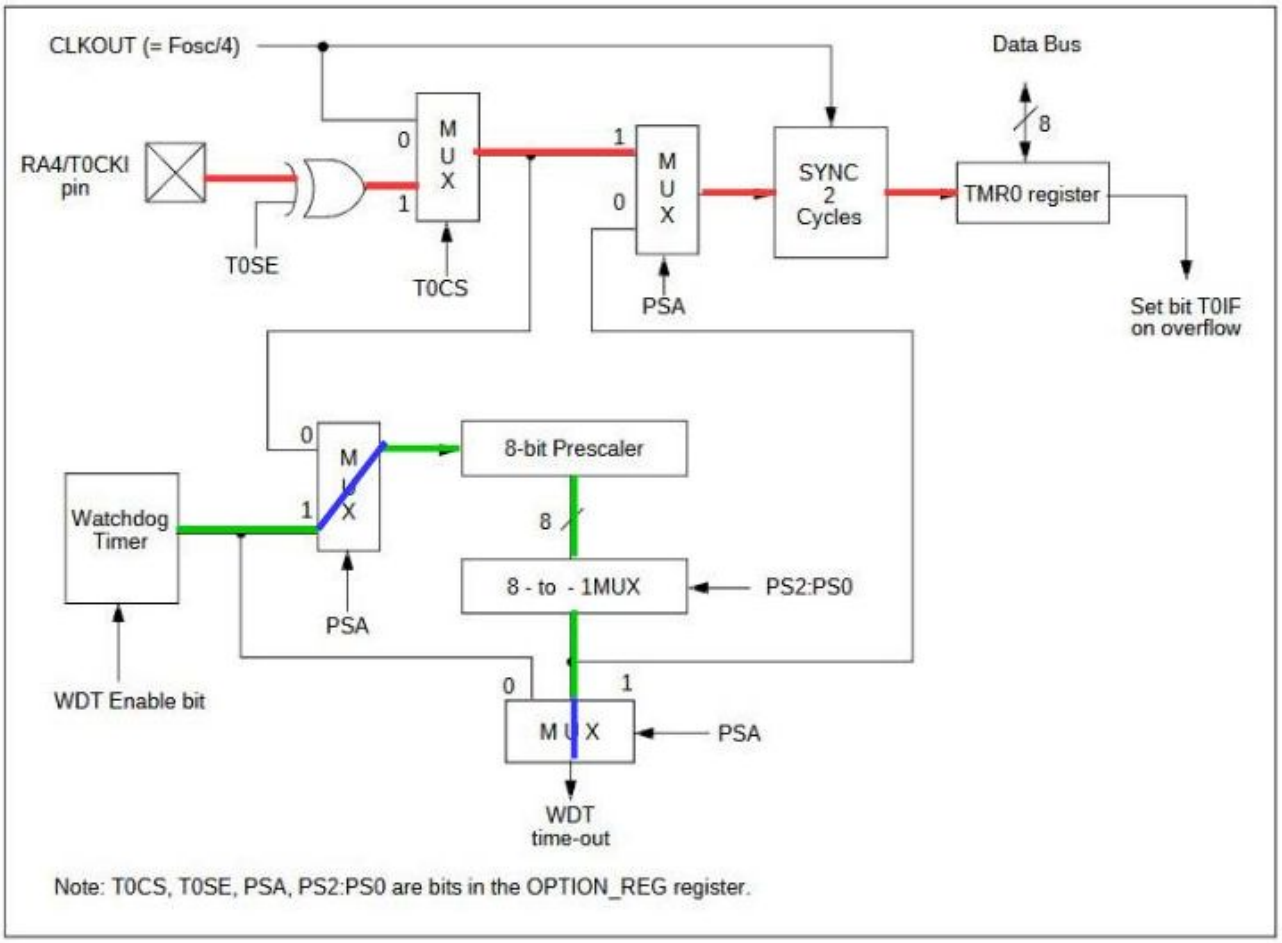

Abbildung 10: Die Signalwege nach einem Reset

Der Vorteiler kann weder direkt gelesen noch direkt beschrieben werden. Allerdings löscht – genauer setzt zurück – ein CLRWDT und ein SLEEP auch den Vorteiler.

9.4. Vorteiler beim Timer

Falls der Vorteiler dem Timer zugeordnet und der Watchdog aktiv ist, beträgt dessen Timeoutzeit gerade einmal 18ms. Das hört sich sehr kurz an, aber bei einer Befehlszykluszeit von 1 µs sind das immerhin 18.000 Befehle.

Die Signalwege für diesen Fall sind in Abbildung 11 zu sehen.

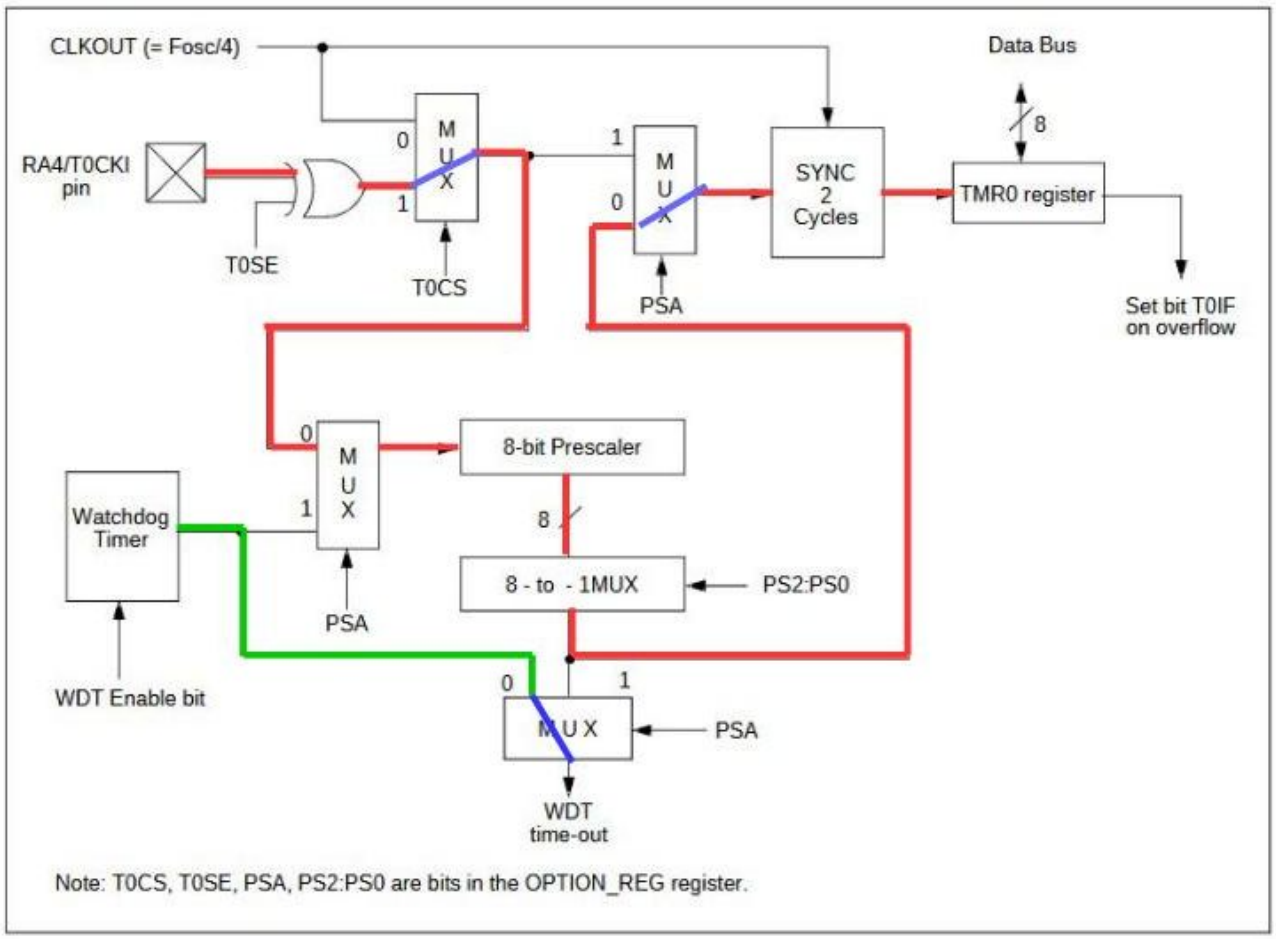

Abbildung 11: Vorteiler beim Timer

Die Teilerfaktoren des Vorteilers sind nun wie folgt definiert:

PS2:0	Teilerfaktor
000	1 : 2
001	1 : 4
010	1 : 8
011	1 : 16
100	1 : 32
101	1 : 64
110	1 : 128
111	1 : 256

Tabelle 3: Teilerfaktoren des Vorteiles zugewiesen zum Timer

Der Vorteiler kann weder direkt gelesen noch direkt beschrieben werden. Allerdings löscht – genauer setzt zurück – ein Schreibzugriff auf den Timer0 (Filere-gister 01) den Vorteiler.

9.5. Falle bei der Vorteilernutzung

Das Beschreiben des Option-Registers und damit auch das Ändern der Zuweisung des Vorteilers bzw. des Einstellen der Teilerfaktoren, hat keinen Einfluss auf den Vorteiler selbst. Das kann aber zu einem unerwarteten Reset führen. Deshalb sind einige Vorsichtsmaßnahmen bei der Änderung der Vorteiler Zurodnung zu treffen.

Soll beispielsweise der Vorteiler vom Timer zum Watchdog[12] wechseln, sollte dies mit folgender Programmsequenz erledigt werden, auch dann wenn der Watchdog nicht aktiv ist:

```
BCF STATUS, RP0     ;auf Bank 0 umschalten (falls nötig)
CLRF TMR0            ;Timer0 und Vorteiler löschen
BSF STATUS, RP0           ;auf Bank 1 umschalten
CLRWDT                    ;Watchdog löschen (läuft damit min.
                     ;18ms)
MOVLW xxxx1xxxB[13]       ;Zuordnung und Vorteilerwert ändern
MOVWF OPTION_REG          ;
BCF STATUS, RP0           ;wieder zurück auf Bank 0
```

Auch bei einem Wechsel vom Watchdog zum Timer ist Vorsicht angebracht.

```
CLRWDT               ;Löscht Watchdog und Vorteiler
BSF STATUS, RP0           ;auf Bank 1 umschalten
MOVLW xxxx0xxxB           ;Zuordnung und Vorteilerwert ändern
MOVWF OPTION_REG          ;
BCF STATUS, RP0           ;wieder zurück auf Bank 0
```

9.6. Genaue Zeitbasis notwendig?

Benötigt man bei den PIC-Mikrocontrollern eine genaue Zeitbasis, muss einiges beachtet werden. Die übliche Vorgehensweise ist die folgende: Man legt eine Zeit fest, mit der Timer0-Interrupts ausgelöst werden sollen, z.B. ein Interrupt pro Millisekunde. Für diese Größe berechnet man die Werte für den Vorteiler und den Timer0-Wert. Da das Ganze von der Quarzfrequenz abhängt sieht die Rechnung für diesen Fall so aus:

Quarzfrequenz 4,0 MHz ergibt 1 µs Befehlstakt für den Timer. Somit muss der Zähler, bestehend aus Vorteiler und Timer0 auf 1000 zählen. Der Timer0 kann

12 Dieser Wechsel kommt durchaus öfters vor. Z.B. wenn man in einen längeren Stromsparmodus gehen möchte. Aber auch dann, wenn man den Vorteiler für den Timer0 auf 1:1 setzen möchte.

13 Das B bedeutet, dass es sich um eine Binärzahl handelt, x steht für Werte die der Anwender festlegen muss

max. auf 256 (incl. 0) zählen, den Vorteiler kann man nur auf Teilerverhältnisse von 1:1, 1:2, 1:4, 1:8 usw. einstellen. Somit wäre eine Möglichkeit diese 1ms Zeitbasis zu erzeugen indem der Vorteiler auf 1:4 eingestellt und der Timer0 auf 250 zählt. Der Wert der dann in den Timer0 geladen werden muss berechnet sich zu 256 – 250 = 6. Man lädt 6 und lässt ihn auf 256 hoch zählen. Beim Überlauf von 255 auf 0 (= 256) wird dann der Interrupt ausgelöst. In der Interruptserviceroutine (ISR) muss aber der Timer 0 erneut mit 6 geladen werden. Dieser Schreibzugriff löscht aber den Vorteiler was zu einem späteren Auftreten des nächsten Interrupts führt. Damit wird die gewünschte Zeitbasis alles andere als genau[14]. Eine weitere Verzögerung stammt von der Synchronisierungslogik die beim Schreibzugriff auf den Timer0 eine Verzögerung um 2 Zyklen erzwingt.

Abhilfe erreicht man, indem man das Nachladen des Timer0-Registers in der ISR vermeidet. Das lässt sich dadurch realisieren, dass der Nachladewert 0 ist. Damit teilt das Timer0-Register durch 256. Wählt man einen Vorteilerwert von 1:4, ergibt sich eine Quarzfrequenz von 4,096 MHz. Diese „krummen“ Quarzfrequenzen sind in der Digitaltechnik sehr weit verbreitet, da sie mittels Binärteiler ohne Rest teilbar sind.

9.7. Hinweis für das Simulatorprojekt:

Die Verzögerungslogik vor dem Timer muss nicht implementiert werden. Wer will darf es natürlich. Das Löschen des Vorteilers beim Schreibzugriff auf Timer0 ist dagegen leicht zu realisieren.
Da der Watchdog nicht per Assemblerprogramm ein- bzw. ausgeschaltet werden kann, ist es sinnvoll einen eigenen Button zum Aktivieren vorzusehen. Es entspricht dem WDTE-Bit im Konfigurationsregister.

9.8. Reset (20.12.2018)

Es gibt bei PIC unterschiedliche Möglichkeiten den Prozessor zurück zusetzen. Diese unterscheidliche Situationen werden in den Statusbits $\overline{PD}$ und $\overline{TO}$ signalisiert…..

14 Es sind Abweichungen von mehreren Sekunden pro Stunde möglich.

10. Interrupts

Interrupts sind Programmunterbrechnungen die zeitlich i.d.R. nicht vorhersehbar sind. Beim PIC16F84 ist keine Priorität der einzelnen Interrupts festlegbar. Alle sind gleichberechtigt und sie dürfen nicht verschachtelt werden. Damit diese Interruptverschachtelung nicht passieren kann, wird das GIE-Bit beim Eintritt in den Interrupt gesperrt und durch den RETFIE-Befehl wieder gesetzt.

Jede Interruptquelle hat ein eigenes Interrupt-Enable-Bit. Damit lassen sich einzelne Interruptquelle aktivieren bzw. deaktivieren. Zusätzlich besitzt jede Quelle ein eigenes Interruptflag. Dieses Flag wird immer dann gesetzt, wenn die entsprechende Bedingung erfüllt ist. Das passiert unabhängig vom GIE-Bit. Damit können die Ereignisse auch per Polling erfasst werden.

Eine Verzweigung aus dem laufenden Programm zur Interrupteinsprungadresse 0x0004 kann nur dann erfolgen, wenn das GIE, das individuelle Interrupt-Enable-Bit und das Interrupt-Flag-Bit gesetzt sind.

z.B. Timerinterrupt
GIE = 1 + T0IE = 1 + T0IF = 1 => Interrupt wird ausgelöst

Der Sprung zur Adresse 0x0004 erfolgt in Form eines Unterprogrammaufrufs. Daher wird die Rückkehradresse auf dem Stack hinterlegt. Das muss bei der Verschachtlung von „normalen“ Unterprogrammen mit berücksichtigt werden (siehe dazu auch Kapitel 5).

In der Regel muss der Assemblerprogrammierer bei der Abarbeitung der Interrupts die gesetzten Interruptflags zurücksetzen. Tut er das nicht, verzweigt der Controller immer und immer wieder zur Adresse 0x0004, da die oben erwähnte Bedingung erfüllt bleibt.

z.B. Timerinterrupt

```
0004        btfsc       INTCON, INTF        ;durch RB0 ausgelöst?
0005        goto        INT_ISR             ;ja
0006        btfsc       INTCON, T0IF        ;ein Timerüberlauf?
0007        goto        TIMER_ISR[15]       ;ja
            ggf. weitere Interruptquellen prüfen
```

15 ISR = Interrupt Service Routine. Das Unterprogramm, das die notwendige Reaktion auf diesen Interrupt ausführt.

11. Konfigurationsregister

Dieses Register kann nur während des Programmiervorgangs beschrieben bzw. gelesen werden. Hier werden u.a. der Oszillatortyp, der Programmschutz, der Watchdog und Power-Up-Timer eingestellt.

Alle diese Einstellungen, mit Ausnahme des Watchdogs, sind für das Simulatorprojekt ohne Interesse.

Um im Simulator den Watchdog ein- und ausschalten zu können, sieht man am Besten einen Button dafür vor, auch wenn dies nicht der Realität entspricht. Denn im Original lässt sich der Watchdog nur während der Programmierphase aktivieren oder sperren..

12. Die Adressierungsarten

Der PIC-Mikrocontroller kennt nur drei Adressierungsarten, die direkte, die indirekte und die unmittelbare Adressierung.

12.1. Unmittelbare Adressierung[16]

Bei der unmittelbaren Adressierung folgt das Argument unmittelbar auf den Befehlscode. Bei PIC-Mikrocontroller sind dies die sogenannten Literal-Befehle.

Beispiel:

```
          MOVLW   55H
```

Da der PIC neben dem Befehlscode auch das Argument unter einem Mal in den Befehlsdekoder einliest, benötigt er keinen zusätzlichen Zugriff auf den Programmspeicher, wie es bei Von-Neumann-Rechnern üblich ist.

12.2. Direkte Adressierung

Bei der direkten Adressierung wird die Adresse direkt als Zahlenwert dem Befehl beigestellt. Der Binärwert des Befehls zeigt diesen Sachverhalt am Besten.

Beispiel:

```
          MOVWF   12H          00 0000 1001 0010
(allgemeine Darstellung        00 0000 1fff ffff)
```

Die letzten 7 Stellen (fff ffff[17]) sind die Adresse 12H. Der Wert in W wird mit diesem Befehl an die RAM-Speicheradresse 12H geschrieben.

Bei den Bitbefehlen ist neben der Adresse des Fileregisters auch die Adresse (Position) des gewünschten Bits im Befehl.

Beispiel:

```
          BCF  3,1             01 0000 1000 0011
(allgemeine Darstellung        01 00bb bfff ffff)
```

Hier steht das bbb für die Bitadresse (0 bis 7) und fff ffff für die RAM-Adresse.

16 Mehr dazu im Anhang auf Seite 42

17 f steht für Fileregister.

12.3. Indirekte Adressierungsart

Manchmal ist es einfacher, man könnte die Adresse als Zeiger benutzen. Z.B. wenn man eine Summe über einen Speicherbereich bilden will. Dazu stellt der PIC-Mikrocontroller ein sogenanntes File-Select-Register (FSR) im Zusammenspiel mit dem Indirect-Register (IND) zur Verfügung.

Für unser Beispiel der Bildung der Summe aus dem Speicherbereich 10H bis 17H würde das Programm mit der direkten Adressierung wie folgt lauten:

```
        CLRW                ;löscht das Ergebnisreg.
        ADDWF    10H,W      ;add. den Inhalt von 10H
        ADDWF    11H,W      ;zum W-Register
        ADDWF    12H,W
        ADDWF    13H,W
        ADDWF    14H,W
        ADDWF    15H,W
        ADDWF    16H,W
        ADDWF    17H,W
```

Eleganter lässt sich das mit der indirekten Adressierung lösen. Dabei wird in das FSR die Adresse (Zeiger) geladen. Über den Zugriff auf das IND-Register greift man indirekt auf die Speicherzelle zu, deren Adresse im FSR steht.

Somit würde das obige Beispiel mit der indirekten Adressierung wie folgt aussehen:

```
        MOVLW    10H        ;Zeiger ins FSR laden
        MOVWF    FSR        ;FSR = Adresse 04H
        MOVLW    8          ;Größe des Bereiches
        MOVWF    COUNT      ;Zählregister
        CLRW                ;Ergebnisreg. löschen
loop    ADDWF    IND,W      ;IND = Adresse 00H
        INCF     FSR        ;Zeiger weiterstellen
        DECFZS   COUNT      ;8 x durchlaufen
        GOTO     loop
```

Am Ende des Programms steht in COUNT der Wert 0, in FSR der Wert 17H.

13. Befehle (20.12.2018)

Die PIC-Mikrocontroller besitzen einen sogenannten RISC[18]-Befehlssatz. Der Vorteil dieses Befehlsatzes ist die Geschwindigkeit mit der, diese in der Regel einfachen Befehle, abgearbeitet wird. In diesem Fall benötigen die meisten Befehle nur vier Quarztakte was bei einem 4 MHz Quarz eine Befehlszeit von 1µs bedeutet.

Des weiteren liest der Controller nicht nur den Befehlscode sondern auch dazugehörende Argumente bereits beim ersten Fetchzyklus komplett ein. Da die Argumente unterschiedlich lang sind, hat auch der Befehlscode eine variable Länge.

Beispiel:

```
CLRWDT      00 0000 0110 0100 ;Befehlscode 14 Bit, da keine Argumenten
MOVLW n     11 00xx nnnn nnnn ;Befehl 6 Bit[19], Konstante 8 Bit
MOVWF f     00 0000 1fff ffff ;Befehl 6 Bit, Filereg.-Adr. 7 Bit
MOVF f,d    00 1000 dfff ffff ;Befehl 6 Bit, Fileadr. 7 Bit, Destination=1
BSF b,f     01 01bb bfff ffff ;Befehl 4 Bit, Bitposition 3 Bit Fileadr. 7
GOTO a      10 1aaa aaaa aaaa ;Befehl 3 Bit, Adresse 11 Bit
```

Bei der Befehlsanalyse muss dieser Umstand berücksichtigt werden!

SLEEP der Sleepmodus kann dadurch beendet werden, dass der Watchdog abläuft. In diesem Fall wird der nächste Befehl (in der Pipeline) ausgeführt. Ebenso beendet ein Interrupt Ereignis den Sleep. Sind die entsprechenden Bits gesetzt, wird nach der Ausführung des Befehls in der Pipeline zur Adresse 0x0004 verzweigt.

13.1. Der Befehlsdecoder

Nachdem ein Befehl aus dem Programmspeicher eingelesen wurde, muss dieser in einen Befehlscode und die diversen Argumente aufgeteilt werden. Das lässt sich am einfachsten mit einer Maske und einer logischen Verknüpfung[20] erreichen.

Will man beispielsweise die Befehle GOTO und CALL herausfiltern, dann ver-

18 RISC = Reduced Instruction Set Computer (Rechner mit reduziertem Befehlssatz)
19 X bedeutet don‘t care. Damit gibt es für diesen Befehl vier verschiedene Befehlcodes
20 Voraussetzung ist ein Programmspeicher als Integer Array

undet (logisches UND) man den eingelesenen Befehl mit 0x3800 (= 11 1000 0000 0000). Beim CALL-Befehl ist das Ergebnis 0x2000, beim GOTO 0x2800. Da es drei Befehlsgruppen mit Argumenten gibt, sind somit mindestens drei Masken notwendig. Für die Befehle ohne Argumente, können die gesamten vierzehn Bits direkt abgefragt werden. Diese Vergleiche können in einer CASE-Anweisung einfach der passenden Methode zugeordnet werden.

Das Auftrennen der Argumente kann man entweder in der entsprechenden Methode durchführen oder bereits vor dem entsprechenden Methodenaufrufen. Im letzteren Fall kennt man zu diesem Zeitpunkt noch nicht, welche Argumente wirklich benötigt werden. Deshalb erzeugt man auf Verdacht alle Argumente und übergibt die, für den Befehl notwendigen Argumente[21], der Methode.

Beispiel:

global definierte Variablen	**Maske**	**Ergebnis**
Sprungadresse	0x7FF	
Fileregisteradresse	0x7F	
Konstante	0xFF	
Bitadresse	0x380	7 x nach rechts schieben
Destinationbit	0x80	7 x nach rechts schieben

Werden die Argumente erst innerhalb der Methode bestimmt, werden natürlich die selben Masken wie oben beschrieben verwendet.

13.2. Ergänzungen zur Befehlsbeschreibung

COMF bildet das 1er-Komplement Das Ergebnis kommt je nach Destinationbit ins W- oder f-Register.

CLRWDT Löscht den Watchdogzähler und den Vorteiler, falls dieser dem Watchdog zugeordnet ist (PSA-Bit)

DECFSZ zuerst wird der Wert dekrementiert, in W oder f abgespeichert und dann der nächste Befehl übersprungen, falls der Wert 0 wurde.

INCFSZ zuerst wird der Wert inkrementiert, in W oder f abgespeichert und dann der nächste Befehl übersprungen, falls der Wert 0 wurde

21 Man kann auch die entsprechneden Variablen als global efinieren und hat dann in jeder Methode darauf Zugriff. Das vereinfacht den Methoddenaufruf etwas.

MOVF ist der Wert im Fileregister f 0, dann wird das Z-Flag gesetzt. Das Ergebnis kommt je nach Destinationbit ins W- oder f-Register.

ADDWF Der Wert im W-Register wird zum Wert im Fileregister f addiert. Das Ergebnis kommt je nach Destinationbit ins W- oder f-Register.

SUBWF Subtrahiert den Inhalt aus dem W-Register vom Inhalt des Filere gisters. Das Ergebnis kommt je nach Destinationbit ins W- oder f-Register.
Vorsicht: Das Carry- und Digit-Carry-Flag werden anderst als er wartet behandelt. Eine Subtraktion wird durch die Addition des 2-er Komplements durchgeführt. Normalerweise muss zum Schluss das C- und DC-Flag invertiert werden. Das passiert beim PIC-Control ler **nicht.**[22]

SLEEP Löscht Watchdogzähler und ggf. den Vorteiler, setzt das TO-Bit, löscht das PD-Bit. Die I/O-Pins halten ihre aktiven Zustände. Wird der SLEEP durch einen Watchdogreset, einem MCLRReset oder einem Interrupt an RB 0 bzw. RB 4 bis RB 7 beendet. Die beiden Bits TO und PD im Statusregister geben Auskunft, wie der SLEEP beendet wurde.

POWER ON RESET	$\overline{PD} = 1$	$\overline{TO} = 1$
MCLR	$\overline{PD} = 0$	$\overline{TO} = 1$
Timeout des Watchdogs	$\overline{PD} = 0$	$\overline{TO} = 0$

Beim Auftreten eines Interrupts wird das Programm beim nächsten, auf den SLEEP folgenden, Befehl weitergeführt. Dabei spielt das GIE-Bit zuerst keine Rolle. Wenn es gesetzt ist und ein anderes Enablebit ebenso, wird der in der Pipeline stehende Befehl abgearbeitet und dann zur Adresse 0x0004 verzweigt.

22 Die Marketingstrategen erheben diesen Fehler zu einem Feature indem sie bei der Subtraktion as Carry als Borrowbit uminterpretieren. Siehe dazu auch Kapitel 6.3 und 6.5.

14. Anhang

14.1. Begriffserklärungen

Literal	konstanter Wert
immediate	unmittelbar
Von-Neumann-Rechner	Rechner bei dem Programm- und Datenspeicher in einem gemeinsamen Adressraum liegen.
Harvard-Architektur	Rechner der zwei getrennte Adressräume (Programm- und Datenbereich) besitzt.
Suffix	angehängte Kennung (12H ≙ 12 hexadezimal)
Prefix	vorangestellte Kennung (0x12 ≙ 12 hexadezimal)

weitere folgen!

14.2. Darstellungen von Zahlen

Speziell bei Assemblerprogrammieren nutzt man mehrere Zahlensysteme, je nachdem welche für die aktuelle Darstellung sinnvoller sind. Die drei wichtigsten Zahlensysteme sind:

- Dezimal
- Binär
- Hexadezimal

Um diese einzelnen Systeme zu unterscheiden, werden sie durch Suffixe oder Prefixe gekennzeichnet. Oft lässt sich der Default per RADIX-Anweisung für den Assembler festlegen.

Dezimal:	65d	
Hexadezimal	41H	(0x41)
Binär	01000001B	(0b01000001)

14.3. Adressierungsarten (allgemein)

Die hier aufgeführten Adressierungsarten sind nicht in jedem Mikroprozessor- oder Mikrocontroller-Typ implementiert.

a) Implizierte Adressierung

Diese Adressierungsart wird i.d.R. für Registerzugriffe genutzt.

Beispiel:

```
LD A,B
```

Hier wird der Inhalt des Registers B in das Register A kopiert.

Rechner ohne einen typischen Registersatz kennen diese Adressierungsart praktisch nicht.

b) Unmittelbare Adressierung

Bei Von-Neumann-Rechnern sind hier zwei Zugriffe auf den Programmspeicher notwendig. Der erste Zugriff ist der Fetch-Zyklus, der zweite lädt das Argument welches in ein Register geladen werden soll.

Beispiel:

```
LD   A,55H
```

Der Befehl LD A,55H wird im Speicher als Bytefolge 3E 55 abgelegt. Dabei ist 3EH der Befehlscode für LD A,nn und 55H das Argument nn. Liegt 3EH beispielsweise an der Adresse 1234H so findet man das Argument an Adresse 1235H.

Ein Doppelregister kann mit einem 16-Bit-Wert direkt geladen werden. Hier sind dann zwei zusätzliche Zugriffe auf den Programmspeicher notwendig um das Argument vollständig zu laden.

Beispiel:

```
LD   HL,2345H
```

Im Speicher steht dann folgende Bytesequenz: 21 45 23[23]

c) Direkte Adressierung

Bei dieser Adressierungsart steht die Zieladresse als Zahlenwert direkt beim

23 Der Rechner arbeitet in diesem Fall im Little Endian Format

Befehl. Vor allem Sprungbefehle nutzen die direkte Adressierung. Man unterscheidet hier noch zwischen absoluter und relativer Adressierung.

Beispiel:

```
JP    100AH          ;Springe zur Adresse 100AH.
JR    5              ;Springe um 5 Adressen weiteren
JR    0FEH           ;Springe um 1 Adresse zurück
```

weitere folgen!

14.4. Testprogramm 1

```
                    00001           ;TPicSim1
                    00002           ;Programm zum Test des 16F84-Simulators.
                    00003           ;Es werden alle Literal-Befehle geprüft
                    00004           ;(c) St. Lehmann
                    00005           ;Ersterstellung: 23.03.2016
                    00006           ;mod. 18.10.2018 Version HSO
                    00007           ;
                    00008           list c=132          ;Zeilenlänge im LST auf
                                                        ;132 Zeichen setzen
                    00009
                    00010
                    00011           ;Definition des Prozessors
                    00012           device 16F84
                    00013
                    00014           ;Festlegen des Codebeginns
                    00015           org 0
                    00016  start
0000 3011           00017           movlw 11h           ;in W steht nun 11h,
                                                        ;Statusreg. unverändert
0001 3930           00018           andlw 30h           ;W = 10h, C=x, DC=x, Z=0
0002 380D           00019           iorlw 0Dh           ;W = 1Dh, C=x, DC=x, Z=0
0003 3C3D           00020           sublw 3Dh           ;W = 20h, C=1, DC=1, Z=0
0004 3A20           00021           xorlw 20h           ;W = 00h, C=1, DC=1, Z=1
0005 3E25           00022           addlw 25h           ;W = 25h, C=0, DC=0, Z=0
                    00023
                    00024
                    00025  ende
0006 2806           00026           goto ende           ;Endlosschleife, verhin
                                                        ;dert Nirwana
                    00027
                    00028
```

15. Quellen

Microchip: Datenblatt PIC 16F84