Dokumentation

Simulator für den Microcontroller PIC16F84

in der Vorlesung

Rechnerarchitekturen II

für die Prüfung zum

Bachelor of Science

des Studiengangs Informatik
Studienrichtung Informationstechnik

an der

Dualen Hochschule Baden-Württemberg Karlsruhe

Von

Anna Janzen
Elia Herzog

Inhaltsverzeichnis

1. Einleitung

1.1 Das Projekt

Ziel des Projektes ist die Programmierung eines Simulators des Microcontrollers PIC16F84. Das Projekt wird im Rahmen der Vorlesung Systemnahe Programmierung 2 durchgeführt. Durch das Projekt soll das Verständnis des Microcontrollers vertieft und Kenntnisse in der Programmierung erweitert werden.

1.2 Simulationen

Eine Simulation ist die „Nachbildung eines Systems mit seinen dynamischen Prozessen in einem experimentierfähigen Modell, um zu Erkenntnissen zu gelangen, die auf die Wirklichkeit übertragbar sind“ [1]

Simulationen abstrahieren reale Systeme und sollen dadurch schnelle, aber trotzdem aussagekräftige Antworten liefern. Sie kommen vor allem dann zum Einsatz, wenn die realen Ereignisse zu schnell, zu langsam, zu gefährlich oder sogar unmöglich wären, um die Abläufe in überschaubarer Zeit nachvollziehen oder voraussagen zu können. [2]

Die Simulation soll alle Abläufe bzw. Ergebnisse genauso wiedergeben, wie es das reale System machen würde.

In der nachfolgenden Tabelle 1 werden die Vor- und Nachteile einer Simulation dargestellt.

Vorteile	**Nachteile**
Zeitlicher Ablauf kann manipuliert werden, um Ereignisse sichtbar zu machen	Isomorphie zwischen dem System und dem zu bildenden Modell schwer zu gewährleisten [2]
Kann beliebig oft wiederholt werden	Verfälschte Ergebnisse durch falsche Datenbasis [2]
Oft Kostenersparnisse [2]	
Planung neuer und Verbesserung vorhandener Systeme [2]	

2. Realisation

2.1 Programmierumgebung

Die in dieser Dokumentation beschriebene Simulator wurde in der Programmiersprache C# mit WPF in Visual Studio geschrieben. Die Programmiersprache wurde gewählt, da hier die meisten Kenntnisse zu Beginn des Projekts vorhanden waren.

Die Nutzung von WPF bietet dank der Datenbindungen die beste Grundlage, die Daten, die in der Oberfläche angezeigt werden, im Hintergrund trotz mehrerer Bearbeitungsquellen (Oberfläche und Code) aktuell zu halten und umgekehrt.

Als Architekturkonzept wurde Model-View-Viewmodel (MVVM) benutzt. In dieser Architektur kommuniziert eine Anzeige (View) mit seinem passenden Ansichtsmodel (Viewmodel) und dieses wiederum mit verschiedenen Modellen (Model). Pro View wird ein Viewmodel angelegt, dass alle Aktionen des Views bearbeitet und auch als Datenbindungsquelle dient. Die Models bestehen aus allen Datenmodellen eines Projekts. Diese drei Komponenten werden durch die Services ergänzt, die Logiken und Kommunikation zu externen Datenquellen enthalten. Als Hilfestellung zur Implementierung von MVVM wurde das Nuget-Paket MVVM Light von Laurent Bugnion verwendet.

Zur Versionskontrolle und für die Zusammenarbeit an mehreren Geräten wurde Github verwendet.

2.2 Grundkonzept

Die Simulation besteht maßgeblich aus zwei Komponenten, der Speicherabbildung und der „CPU“.

Die CPU wird als CommandService implementiert. Dieser übernimmt die Abarbeitung der Befehle und die Programmsteuerung. Als Grundlage seiner Berechnungen dient ihm die Speicherabbildung. Der CommandService läuft in einem anderen Thread als die Benutzeroberfläche, um deren Bearbeitung und Aktualisierung nicht zu blockieren.

Der Speicher wird in einer Klasse namens Memory zusammengefasst und besteht aus mehreren Untermodellen, z.B. für den RAM. Alle Variablen, auf die der CommandService und die Benutzeroberfläche zugreifen, befinden sich in einem Memory-Objekt im Viewmodel.

In Abbildung 1 ist grob der Programmablaufplan des CommandServices dargestellt. Ist die Ausführung des Programms gestartet, wird in einer Endlosschleife zunächst geprüft, ob das Programm pausiert wurde oder eine Debug-Markierung erreicht wurde. Ist beides nicht der Fall, wird in einer Switch-Bedingung der Befehl identifiziert und die Argumente ausgelesen (decode). Anschließend wird der Befehl in einer separaten Methode ausgeführt. Um die Handhabung von Sprungbefehlen zu verbessern wird der Programmzähler nach Beendigung des Befehls und nicht, wie im PIC eigentlich der Fall im Fetch-Zyklus, aktualisiert. Nach Abarbeitung des Befehls wird die Schleife erneut durchlaufen.

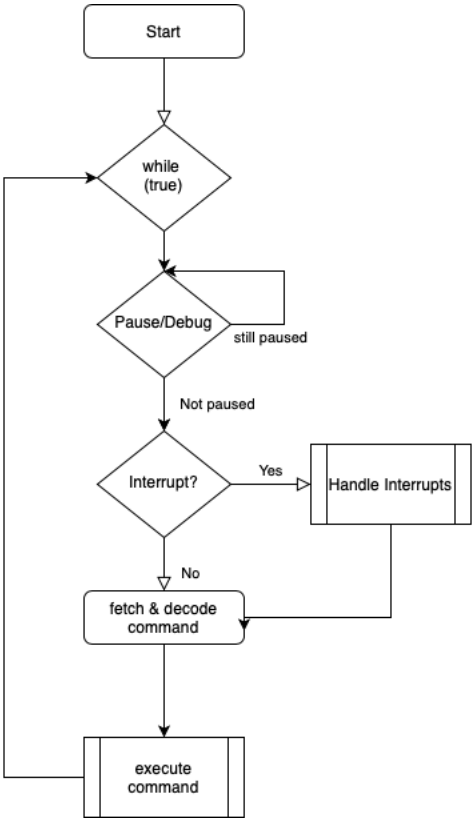


Abbildung 1: Programmablaufplan

2.3 Programmoberfläche

In Abbildung 2 ist die Programmoberfläche der Simulation des PIC16F84 dargestellt.

In einer Menüleiste kann der Anwender eine .LST-Datei in den Simulator laden, die Abarbeitung des Programms starten und pausieren, einen Einzelnen Schritt abarbeiten lassen und den PIC auf den Einschaltzustand zurücksetzen.

Im oberen linken Bereich wird die aktuelle Quarzfrequenz angezeigt und kann aus voreingestellten Werten verändert werden. Zudem wird, abhängig von der gewählten Quarzfrequenz, die Laufzeit in µs angezeigt.

Daneben werden die Bestandteile des Programmzählers angezeigt. Es werden PCL und PCLATH und der gesamte PC, der in verschiedenen Situationen unterschiedlich berechnet wird, dargestellt. Auch der PC-Stack wird angezeigt.

Im unteren linken Bereich wird das aktuell geladene Programm angezeigt. Jede Codezeile kann mit einer Debugmarkierung versehen werden, um vor der Ausführung an dieser Stelle anzuhalten. Der in Ausführung befindliche Befehl wird rot markiert, hier Zeile 0x1E.

Im rechten Bereich werden zunächst die speziellen Register Port A und Port B angezeigt, die über die TRIS-Register bitweise als Ein- oder Ausgang genutzt werden können. Hier kann jedes Bit der beiden Register per Mausklick stimuliert werden. Ist das Bit im TRIS-Register als Ausgang definiert, kann das Bit zwar stimuliert werden, es wird aber erst angezeigt, wenn das Bit wieder als Eingang definiert wird.

Darunter werden der Inhalt des W-Registers und zur schnelleren Lesbarkeit spezielle Flags des Option-Registers, das Carry-, das Digitcarry- und das Zero-Flag angezeigt.

Unten rechts werden in tabellarischer Form die Inhalte aller sich im RAM befindlichen Register angezeigt, sowohl SFR als auch GPR. In dieser tabellarischen Ansicht können durch den Anwender Änderungen an einzelnen Zellen vorgenommen werden.

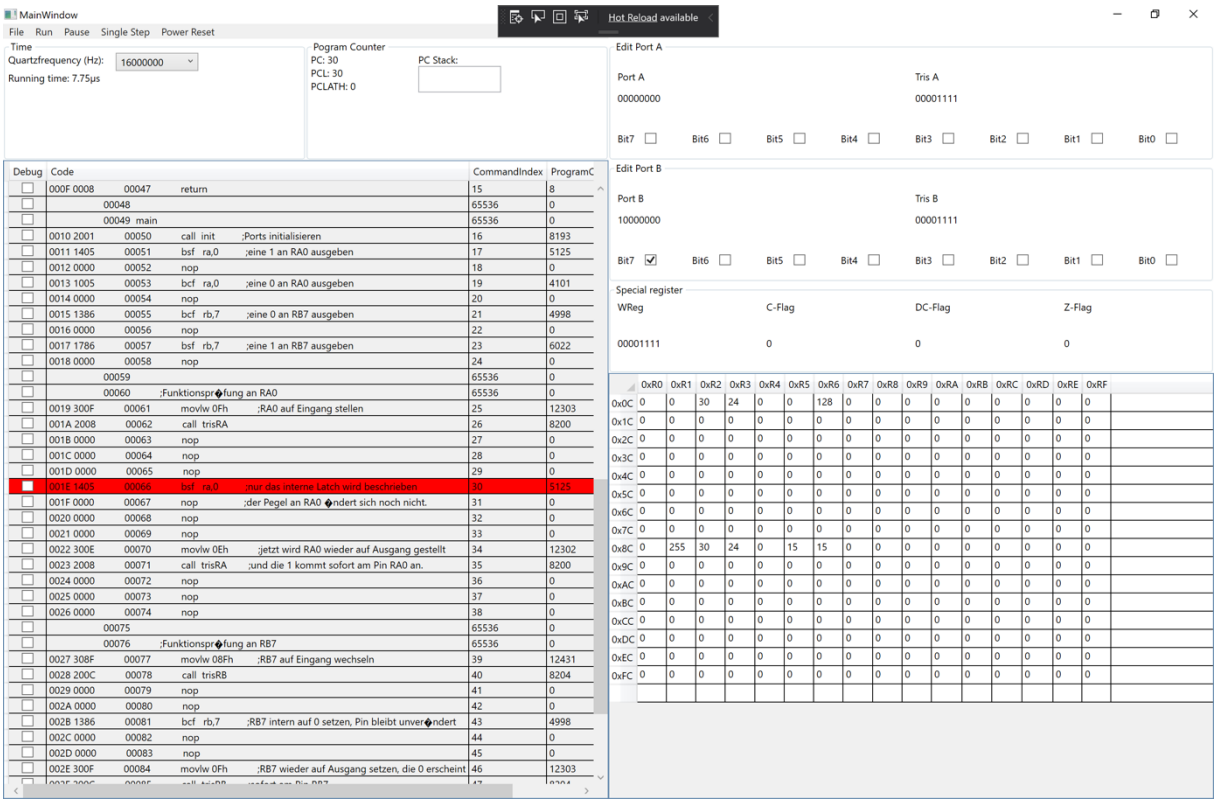
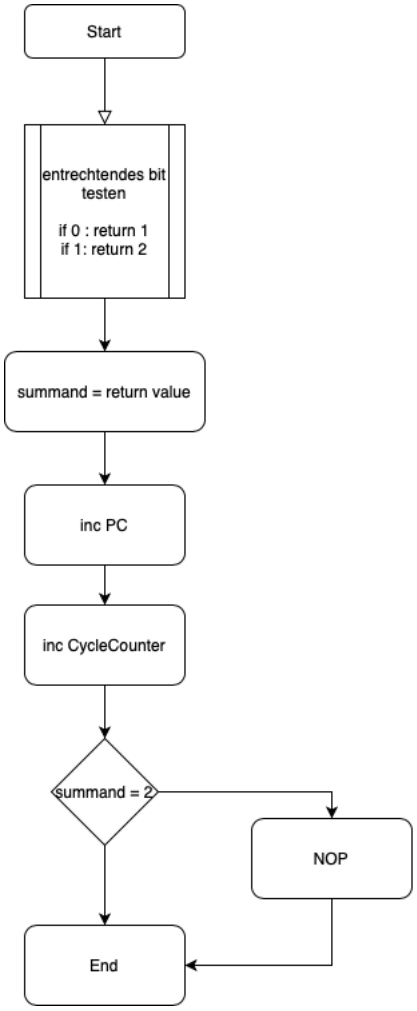


Abbildung 2: Programmoberfläche des Simulators

2.4 Beispielprogramme

Im nachfolgenden werden einzelne Programme detaillierter beschrieben, um die Funktionsweise des Simulators zu verdeutlichen.

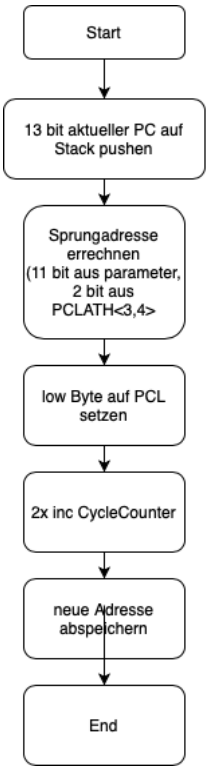
2.4.1 BTFSS




```
public void BTFSS (int file, int bit) //bit test f, skip if set
{
    int summand;
    switch (bit)
    {
        case 0:
            summand = bitTest(memory.RAM[file] & 0b_0000_0001, 1);
            break;
        case 1:
            summand = bitTest(memory.RAM[file] & 0b_0000_0010, 1);
            break;
        case 2:
            summand = bitTest(memory.RAM[file] & 0b_0000_0100, 1);
            break;
        case 3:
            summand = bitTest(memory.RAM[file] & 0b_0000_1000, 1);
            break;
        case 4:
            summand = bitTest(memory.RAM[file] & 0b_0001_0000, 1);
            break;
        case 5:
            summand = bitTest(memory.RAM[file] & 0b_0010_0000, 1);
            break;
        case 6:
            summand = bitTest(memory.RAM[file] & 0b_0100_0000, 1);
            break;
        default: // bit = 7
            summand = bitTest(memory.RAM[file] & 0b_1000_0000, 1);
            break;
    }
    //PC hochzählen
    ChangePC_Fetch((byte)summand);

    //cycles: 1
    memory.CycleCounter++;

    if (summand == 2) //bei überspringen ein nop ausführen (damit breakpu
    {
        BeginLoop();
        //weiterer cycle (theoretisch im nop)
        memory.CycleCounter++;
    }
}
```

Ein Bit aus einem Register wird getestet. Ist es 0, wird der nächste Befehl ausgeführt, ist es 1, wird statt diesem ein NOP ausgeführt.

2.4.2 CALL



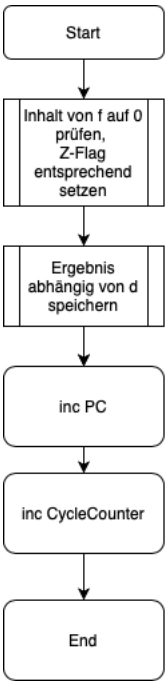

```
public void CALL(int address) // call subroutine -> fertig
{
    //ganzen 13 bit PC auf Stack pushen
    int returnAdress = memory.RAM.RAMList[0].Byte2.Value + (memory.RAM.RAMList[0].Byte10.Value << 8) +1;
    memory.PCStack.Push((short)returnAdress);

    //cycles: 2, aber beide in GOTO

    GOTO(address);
}
```

```
public void GOTO (int address) //go to address -> fertig
{
    //bit 0-10 aus address, bit 11-12 aus PCLATH <3,4>
    int new_pc = address + ((memory.RAM[Constants.PCLATH_B1] & 0b_0001_1000) << 11);

    //execution löschen
    SrcModel[memory.RAM.PC_Without_Clear].IsExecuted = false;

    //PCL setzen
    int newPCL = address & 0b_0111_1111;
    memory.RAM[Constants.PCL_B1] = Convert.ToByte(newPCL);

    //cycles: 2
    memory.CycleCounter++;
    memory.CycleCounter++;

    memory.RAM.PC_was_Jump = true;
    memory.RAM.PC_JumpAdress = new_pc;
}
```

Der Programmzähler wird auf die mitgegebene 11 Bit Adresse eingestellt. Dabei werden die Bits 11 und 12 des neuen PC aus den Bits 3 und 4 des PCLATH-Registers geschrieben. In diesem Simulator wird dafür eine Variable (PC_was_Jump) auf true gesetzt, um in der nachfolgenden Berechnung des Programmzählers die „PC_JumpAdress“ zu benutzen.

Die aktuelle 13 Bit Adresse wird auf den PC-Stack gepusht.

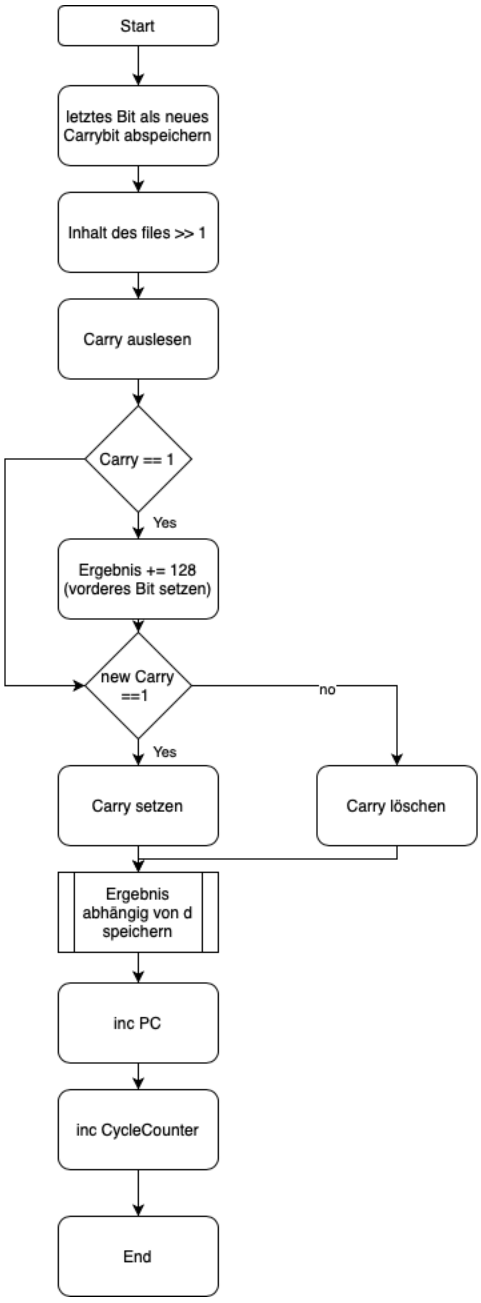
2.4.3 MOVF




```
public void MOVF (int file, int d) // move f
{
    int result = memory.RAM[file];

    checkZ(result);

    StoreSwitchedOnD(file, result, d);

    //PC hochzählen
    ChangePC_Fetch(1);

    //cycles: 1
    memory.CycleCounter++;
}
```

Der Wert im Register wird auf 0 überprüft und das Z-Flag entsprechend gesetzt. Je nach gesetztem d-Bit wird das Ergebnis im selben Register oder im W-Register gespeichert.

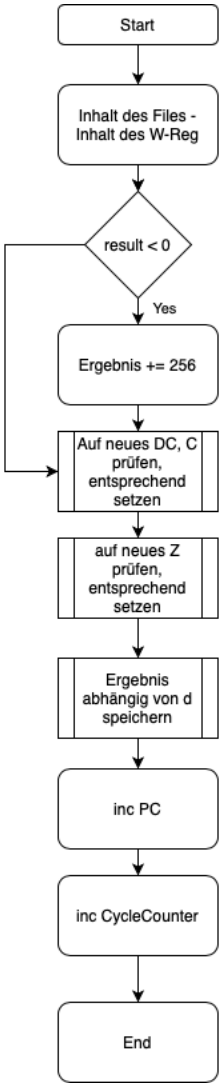
2.4.4 RRF




```
public void RRF (int file, int d) // rotate right through carry
{
    int newCarry = memory.RAM[file] & 0b_0000_0001;
    int result = memory.RAM[file] >> 1;
    int carry = memory.RAM[Constants.STATUS_B1] & 0b_0000_0001;

    if (carry == 1)
    {
        result = result + 128;
    }
    if (newCarry == 1)
    {
        //carry setzen
        memory.RAM[Constants.STATUS_B1] |= 0b_0000_0001;
    }
    else
    {
        // carry löschen
        memory.RAM[Constants.STATUS_B1] &= 0b_1111_1110;
    }

    StoreSwitchedOnD(file, result, d);

    //PC hochzählen
    ChangePC_Fetch(1);

    //cycles: 1
    memory.CycleCounter++;
}
```

Der Inhalt eines Registers wird um eine Stelle nach rechts geshiftet. Dabei wird das aktuelle Carry-Bit das höchstwertigste Bit im Register und das niedrigstwertigste Bit im Register (vor dem Shift) wird das neue Carry-Bit.

2.4.5 SUBWF

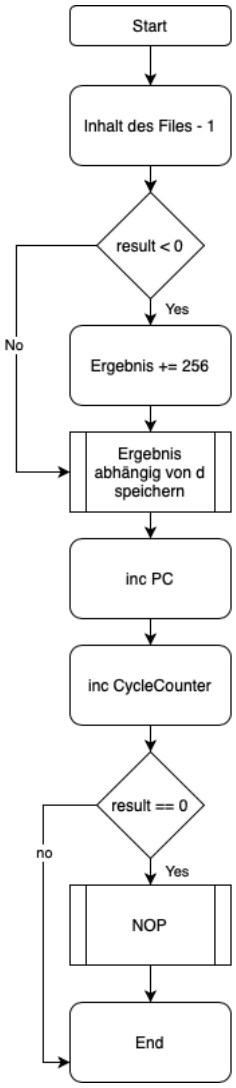



```
public void SUBWF (int file, int d) // substract w from f
{
    int result = memory.RAM[file] - memory.W_Reg; // literal in f ändern, auf d prüfen

    if (result < 0)
    {
        result = result + 256;
    }

    check_DC_C(memory.RAM[file], memory.W_Reg, "-");
    checkZ(result);

    StoreSwitchedOnD(file, result, d);

    //PC hochzählen
    ChangePC_Fetch(1);

    //cycles: 1
    memory.CycleCounter++;
}
```

Der Inhalt des W-Registers wird vom Inhalt eines Registers subtrahiert. Wenn das Ergebnis kleiner als 0 ist, muss auf das Ergebnis 256 addiert werden, um den Überlauf des PIC richtig darzustellen. Die Flags werden neu beschrieben und das Ergebnis abhängig vom Wert des d-Bits abgespeichert.

Carry – und Digitcarry haben hier eine umgekehrte Logik.

2.4.6 DECFSZ

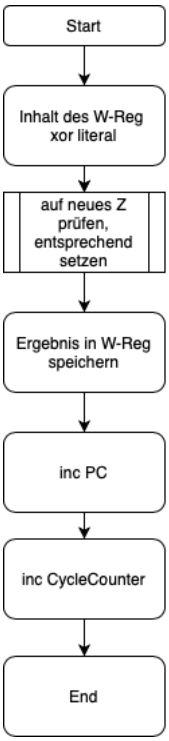



```
public void DECFSZ(int file, int d) //Decrement f, Skip if 0
{
    int result = memory.RAM[file] - 1;
    if (result < 0)
    {
        result = result + 256;
    }
    StoreSwitchedOnD(file, result, d);

    //PC hochzählen
    ChangePC_Fetch(1);

    //cycles: 1
    memory.CycleCounter++;

    if (result == 0) //bei überspringen ein nop ausführen (damit
    {
        BeginLoop();
        NOP();
    }
}
```

Der Inhalt eines Registers wird um 1 verringert. Wenn das Ergebnis kleiner als 0 ist, muss auf das Ergebnis wiederum 256 addiert werden, um den Überlauf des PIC richtig darzustellen. Das Ergebnis wird abhängig vom d-Bit gespeichert.

Wenn jetzt im Register eine 0 steht, wird statt des nächsten Befehls ein NOP ausgeführt.

2.4.7 XORLW




```
public void XORLW (int literal) //exclusive OR literal with w -> fertig
{
    int result;

    result = memory.W_Reg ^ literal;

    checkZ(result);
    memory.W_Reg = (short)result;

    //PC hochzählen
    ChangePC_Fetch(1);

    //cycles: 1
    memory.CycleCounter++;
}
```

Der Inhalt des W-Registers wird mit dem übergebenen Literal per XOR verknüpft. Ist das Ergebnis 0, wird das Z-Flag gesetzt. Das Ergebnis wird im W-Register gespeichert.

2.5 Realisierung der Flags

Flags sind, wie im PIC auch, einfach Bits in Special Function Registern. Sie werden zu entsprechenden Zeitpunkten gesetzt oder gelöscht und wenn eine Funktion von ihnen abhängig ist, wird ihr Wert überprüft und von ihm abhängig die Ausführung fortgeführt.

2.6 Interrupts

Bei Interrupts wird dieselbe Logik angewandt, wie bei den übrigen Flags.

Auf gesetzte Interuptflags wird zu Beginn der Hauptschleife geprüft, da der Interrupt, wie im PIC auch, den aktuellen Befehl nicht abbrechen kann.

Wurde ein Interrupt ausgelöst, wird der Programmzähler auf den Interruptvektor 0x04 gesetzt.

In Abbildung 3 sieht man zunächst die Abfrage auf das Global Interrupt Enable Bit. Ist dieses Gesetzt, werden die übrigen Interrupt Enable Bits geprüft. Ist eines gesetzt, wird in einer ausgelagerten Methode auf das jeweilige Interrupt Flag geprüft, wie in Abbildung 4 am Beispiel des RB0 Interrupts gezeigt. Ist dieses wiederum gesetzt, wird die Interrupt Routine aufgerufen, die in Abbildung 5 zu sehen ist.

```
private void CheckForInterrupts() //INTCON DA MAIN MAN
{
    //GIE gesetzt?
    if((memory.RAM[Constants.INTCON_B1] & 0b1000_0000)>0)
    {
        //GIE gesetzt
        //T0IE gesetzt?
        if ((memory.RAM[Constants.INTCON_B1] & 0b0010_0000) > 0)
        {
            //T0IE gesetzt
            CheckForTimer0Interrupt();
        }
        //EEIE gesetzt?
        if ((memory.RAM[Constants.INTCON_B1] & 0b0100_0000) > 0)
        {
            //EEIE gesetzt
            CheckForEEPROMWrittenInterrupt();
        }
        //INTE gesetzt?
        if ((memory.RAM[Constants.INTCON_B1] & 0b0001_0000) > 0)
        {
            //INTE gesetzt
            CheckForRB0Interrupt();
        }
        //RBIE gesetzt
        if ((memory.RAM[Constants.INTCON_B1] & 0b0000_1000) > 0)
        {
            //RBIE gesetzt
            CheckForPortBInterrupt();
        }
    }
}
```

Abbildung 3: Abfrage auf Interrupt Enable Bits

```
private void CheckForRB0Interrupt()
{
    //INTF Interrupt Flag gesetzt?
    if((memory.RAM[Constants.INTCON_B1] & 0b0000_0010) >0)
    {
        Interrupt();
    }
}
```

Abbildung 4: Abfrage auf Interrupt RB0 Flag

```
private void Interrupt()
{
    //Push PC to Stack
    memory.PCStack.Push((short)memory.RAM.PC_Without_Clear);
    //Set PC to Interrupt Vector
    memory.RAM[Constants.PCL_B1] = (byte)Constants.PERIPHERAL_INTERRUPT_VECTOR_ADDRESS;
    memory.IsISR = true;
}
```

Abbildung 5: Interrupt Routine

2.7 TRIS-Register

Das TRIS-Register ist ebenfalls eine Speicherstelle im SFR. Von ihm ist abhängig, on die Werte in Port A bzw. in Port B aus dem Code heraus oder aus der Benutzeroberfläche heraus geändert werden dürfen. Dies wird im Setter der entsprechenden Register abgefragt.

Steht das zu beschreibende Bit im TRIS auf Ausgang, wird die Bearbeitung des Bits in der Benutzeroberfläche, sowohl in der Extra-Anzeige, als auch in der Darstellung aller Register, gesperrt. Das Bit kann nur durch den Code verändert werden.

Steht das zu beschreibende Bit im TRIS auf Eingang, wird der Wert des Registers nur verändert, wenn die Änderung von der Benutzeroberfläche aus geschieht. Wird das Bit durch den Code verändert, wird der Wert zunächst in eine Latch-Variable geschrieben und der Wert des Registers erst aktualisiert, wenn das Bit im TRIS auf Ausgang gestellt wird.

3. Zusammenfassung

Die vorliegende Simulation des PIC16F84 bildet nahezu alle Funktionen des Microcontrollers ab. Das Verhalten des PIC wurde durch intensives Studium des Datenblatts nachempfunden und implementiert. Lediglich die Programmierung des EEPROMs, der Watchdog und die sleep-Routine wurden nicht implementiert.

Die Abbildung der Funktionen des PIC16F84 stellten während der Implementierung keine großen Schwierigkeiten dar. Lediglich das Verständnis des Verhaltens des Programmzählers war durch unvollständige Dokumentation komplizierter. Dasselbe trifft auf das TRIS-Latch zu.
Oft mussten umständliche Wege gegangen werden, um Daten anzuzeigen. So musst z.B. der Stack des Programmzählers selbst implementiert werden, da ein normaler Stack die Anzeige auf der Benutzeroberfläche nicht aktualisiert, da er kein ObservableObject mit CollectionChanged-Schnittstelle ist. Ähnliche Probleme traten an verschiedenen Stellen auf.

Bei der Durchführung des Projekts wurden Kenntnisse umfangreich verbessert und erweitert. Dies betrifft die Programmiersprache C#, WPF, das Architekturkonzept MVVM sowie Datenbindungen.

Literaturverzeichnis

[1] VDI, VDI 3633, 1993.

[2] W. Prof. Dr.-Ing. habil. Dangelmaier und C. Prof. Dr. Laroque, „Simulation," 13 Mai 2020. [Online]. Available: https://www.enzyklopaedie-der-wirtschaftsinformatik.de/lexikon/technologien-methoden/Operations-Research/Simulation.