# PIC16F8X Simulator

Inhaltsverzeichnis

[PIC16F8X Simulator 1](#_Toc165205808)

[Einleitung Simulator 2](#_Toc165205809)

[Funktionen der Benutzeroberfläche 2](#_Toc165205810)

[Ablauf einer Simulation 4](#_Toc165205811)

[Interner Aufbau 4](#_Toc165205812)

[Registeraufbau 4](#_Toc165205813)

[Stack 5](#_Toc165205814)

[Programmspeicher 5](#_Toc165205815)

[Vorteiler 6](#_Toc165205816)

[Verarbeitung Befehle 7](#_Toc165205817)

[Auswertung Opcode 7](#_Toc165205818)

[InstructionDecoder 7](#_Toc165205819)

[Befehlsumsetzung 8](#_Toc165205820)

[MOVF 8](#_Toc165205821)

[CALL 10](#_Toc165205822)

[GOTO 10](#_Toc165205823)

[ANDWF/ADDWF 10](#_Toc165205824)

[Einlesen der LST-Datei 10](#_Toc165205825)

[BankAddressResolution 10](#_Toc165205826)

[DirectionalWrite 10](#_Toc165205827)

[Entwurfsmuster 10](#_Toc165205828)

[MVVM 10](#_Toc165205829)

[Fazit 10](#_Toc165205830)

[Abbildung 1: Übersicht Simulator UI 2](#_Toc165032069)

[Abbildung 2: Codeausschnitt SetRegister() 3](#_Toc165032070)

[Abbildung 3: Funktionen für Stack 3](#_Toc165032071)

[Abbildung 4: Opcode Encoding 4](#_Toc165032072)

[Abbildung 5: Isolation von Desitinationbit und f-Register 4](#_Toc165032073)

[Abbildung 6: 14Bit Opcode ANDWF Befehl 4](#_Toc165032074)

[Abbildung 7: Ausschnitt InstructionDecoder 5](#_Toc165032075)

[Abbildung 8: Ablaufdiagramm Befehl MOVF 5](#_Toc165032076)

[Abbildung 9: Implementierung MOVF-Befehl 6](#_Toc165032077)

## Einleitung Simulator

Ein Simulator ist eine präzise Nachbildung, die eine möglichst realistische Umgebung schafft. Dadurch können sowohl grundlegende als auch spezielle Szenarien ohne den Einsatz des Originals simuliert werden. Der Simulator bietet die Möglichkeit, die Bedingungen kontrolliert zu variieren, um spezifische Situationen darzustellen. Dies ermöglicht detaillierte Studien zur Funktionsweise und anderen Aspekten. Simulatoren sind auch hilfreich für Tests und Lernzwecke. Allerdings ist eine Simulation oft eine vereinfachte Darstellung der Realität und kann diese nicht vollständig erfassen. Die Entwicklung von Simulatoren wird zunehmend komplexer, insbesondere bei komplexen Szenarien.

In einem Projekt wurde ein Simulator für den Microcontroller PIC16F84 in C# erstellt. Dabei war die Wahl der Programmiersprache frei. Obwohl nicht alle Funktionen des PIC16F84 im Simulator implementiert wurden, können doch alle grundlegenden Funktionen simuliert werden. Alle Instruktionen sind unterstützt, sodass einfache Programme ohne Abweichungen ausgeführt werden können.

## Funktionen der Benutzeroberfläche

5

1

Ein Bild, das Text, Zahl, Screenshot enthält.

Automatisch generierte Beschreibung

8

10

9

7

6

4

3

2

Abbildung 1: Übersicht Simulator UI

Der Button File (siehe 1) öffnet ein Dateiauswahlfenster und ermöglicht so das Einlesen einer LST-Datei. Sobald die Datei ausgewählt wurde, wird diese im LST-File View (siehe 8) angezeigt. Neben jeder Zeile, die einen Befehl enthält, kann ein Breakpoint gesetzt werden. Der aktuelle Befehl, also der Befehl, der als nächstes ausgeführt wird, ist Orange umrandet. Im Block SFR + W (siehe 1) sind die Wichtigsten Werte unteranderem der PC, FSR, PCL und PCLATH gelistet. Dies dient der einfacheren Bedienung und dem schnellen Zugriff auf die wichtigsten Daten. Auch der Teilfaktor des Vorteilers wird angezeigt. Ist der Vorteiler dem Timer0 zugeschalten, wird dies als Postscaler angezeigt, wenn er dem WatchDogTimer zugewiesen ist, wird es als Prescaler bezeichnet. Der Wert beschreibt, welcher Teilfaktor aktuell über die PSA2-0 Bits dem Vorteiler zugewiesen ist. Im Block SFR (siehe 3) sind für die Register Status, Option und Intcon alle Bits aufgeführt. Block PortA und PortB (siehe 4) Visualisieren pro Port sowohl das Port Register als auch das dazugehörige Tris Register. Sowohl Tris, welches steuert, ob ein Port Eingang oder Ausgang ist, oder die Ports selbst können via Klick auf die Checkbox gesetzt und rückgesetzt werden. Der Block Stack (siehe 5) visualisiert die 8 Bytes des Stacks. Im Settings Block (siehe 6) können alle Einstellungen des Simulators eingestellt werden. Unteranderem kann der ClockSpeed umgestellt werden, was sich auf die Berechnung der Laufzeit auswirkt. Der SimulationSpeed bestimmt das Intervall, in dem die Befehle ausgeführt werden. Runtime und Watchdog zeigen jeweils die aktuelle Zeit an, die das Programm und der Watchdog laufen. Über die Checkbox Watchdog Enabled kann der Watchdog ein- und ausgeschaltet werden. Ist er aus, wird dieser auch bei korrekt gesetztem PSA-Bit nicht gestartet. Im Control Block (siehe 7) kann der Simulator gesteuert werden. Der Button Start startet den automatischen Durchlauf des Programmes. Der Simulator stoppt dann nur bei einem Breakpoint oder einer unendlichen Schleife. Sobald der Simulator läuft, ändert der Button seinen Text auf Stop um den Simulator wieder zu stoppen. Mit Step kann ein einzelner Befehl ausgeführt werden und mit Reset kann der Simulator inklusive der Runtime zurückgesetzt werden. Hierbei werden auch wieder die Standardwerte in die Register geladen. Im Block Fileregister (siehe 10) wird der gesamte Datenspeicher visualisiert. Jedes Register wird als Hexadezimal Zahl angezeigt.

## Ein Bild, das Text, Entwurf, Origami, Diagramm enthält. Automatisch generierte BeschreibungAblauf einer Simulation

Um eine Simulation zu starten, muss zunächst eine LST-Datei ausgewählt werden. Anschließend wird durch einen FileReader die Datei eingelesen und interpretiert. Hierbei werden zwei unterschiedliche Listen erstellt. Eine (SourceFile) welche verwendet wird, um das LST-File in der UI anzuzeigen und eine (program), welche den Programmspeicher des PIC repräsentiert. SourceFile enthält eine exakte Kopie des LST-Files, also inklusive Leerzeilen, Kommentaren usw. In program befinden sich nur die Zeilen aus dem LST-File, welche einen Befehl enthalten. Aus dem LST-File kann entnommen werden, an welcher Adresse im Programmspeicher der jeweils eingelesene Befehl stehen muss. Der 4-stellige hexadezimale Opcode wird eines Befehls wird in Form eines Objektes von Command abgespeichert. Beim Erstellen eines Command wird aus dem 4-stelligen hexadezimalen ein High-Byte und ein Low-Byte extrahiert. Das High-Byte repräsentiert die vorderen zwei Ziffern der Hexadezimalen Zahl und das Low-Byte die hinteren zwei. Wenn der Programmspeicher geladen wurde, sowie das LST-File in der UI angezeigt wird, folgt ein Reset des Datenspeichers. Hierbei werden Register mit Standardwerten initialisiert. Jetzt ist der Simulator bereit, ausgeführt zu werden. Die Ausführung eines Befehles wird Eventgesteuert durchgeführt. Der Simulator verfügt über einen StepTimer, welcher standardmäßig mit dem Wert 50ms initialisiert wird. Der StepTimer zählt immer von diesem Wert herunter auf null und wenn er null erreicht hat, wird ein Event ausgelöst. Der Wert (SimSpeed) steuert also, wie schnell der PIC Instruktionen abarbeitet. Dies hat jedoch keine Auswirkung auf die berechnete Laufzeit. Sobald das Event erzeugt wird, führt der PIC einen Befehl aus. Wenn nur ein einzelner Schritt über den Button Step in der UI ausgeführt wird, kommt der StepTimer nicht zum Einsatz. Zusätzlich wird beim Ausführen eines Befehls anhand der Taktfrequenz (ClockSpeed) die Laufzeit (Runtime) berechnet und erhöht.

# Interner Aufbau

## Registeraufbau

Die Register sind als ein Byte Array mit Größe 256 umgesetzt. Bank 0 und Bank 1 sind direkt nahtlos hintereinander. Je nach Zustand des RP0-Bit wird dann auf Bank 0 oder Bank 1 zugegriffen. Um auf Bank 1 zuzugreifen, wird auf die Adresse des gewünschten Registers der Wert 0x80h addiert. Somit wird auf das Register in Bank 1 zugegriffen.

Die Register: PCL, STATUS, FSR, PCLATH, INTCON, OPTION sind von Bank 0 auf Bank 1 gespiegelt. Wenn also auf Bank 0 oder Bank 1 etwas in diesem Register geändert, muss dies gleichermaßen auch auf dem entsprechenden Register auf der anderen Bank geändert werden.

Hierzu wurde eine Funktion SetRegister() geschrieben, welche das primäre Register setzt und dann unter anderem die oben genannten Sonderfälle prüft und so ebenfalls die gespiegelten Register setzt.

Ein Bild, das Text, Screenshot, Schrift, Software enthält.

Automatisch generierte Beschreibung

Abbildung 2: Codeausschnitt SetRegister()

## Stack

Der Stack ist als ein Integer Array mit 8 Stellen realisiert. Der Stack verfügt über zwei Funktionen, Push und Pop.

Ein Bild, das Text, Screenshot, Schrift enthält.

Automatisch generierte Beschreibung

Abbildung 3: Funktionen für Stack

Die Funktion PushOnStack() schreibt den aktuellen PC an die Stelle im Stack, auf die der Stackpointer aktuell zeigt. Initial steht der Stackpointer auf 0. Sollte der Stackpointer 7 sein, dann wird dieser wieder auf 0 zurückgesetzt.

Die Funktion PopStack() verringert den Stackpointer um 1 und zeigt somit auf den eins tieferen Wert. Sollte der Stackpointer 0 sein, wird auf 7 gesprungen.

## Programmspeicher

Der Programmspeicher wurde durch eine Liste aus Objekten vom Typ Command realisiert. Ein Command besteht aus einem High-Byte und einem Low-Byte. Der Opcode, den wir aus den LST-Dateien auslesen, besteht aus einer 4-stelligen Hexadezimalen Zahl. Die vorderen 2 Hex-stellen bilden das High-Byte, die hinteren 2 das Low-Byte. Beim Initialisieren eines Command wird der ganze Opcode übergeben und dann im Konstruktor die Aufteilung in High- und Low-Byte durchgeführt. Hierbei wird zunächst die Hexadezimale Zahl zu einem Integer konvertiert und anschließend zu einem Byte.

Die Aufteilung in High- und Low-Byte erleichtert sowohl die Dekodierung der Befehle als auch das Separieren von Destinationbit und F-Register. Weitere Informationen hierzu in (Auswertung Opcode).

## Vorteiler

Der Vorteiler kann entweder dem Watchdog (WDT) zugewiesen sein, oder dem Timer0 (TMRO). Die Zuweisung geht über das PSA-Bit im Intcon Register. Steht das PSA-Bit auf 0 ist der Vorteiler dem Timer0 zugewiesen. Steht der PSA-Bit auf 1 ist es dem Watchdog zugewiesen. Jedoch muss der Watchdog dann weiterhin über das WDT Enable bit aktiviert sein.

Die Teilfaktoren unterscheiden sich je nachdem, ob der Vorteiler dem Timer0 oder dem WatchDog zugewiesen ist. Die Bits PS2:0 geben an, mit welcher Zahl Teilfaktor berechnet wird.

Ein Bild, das Text, Screenshot, Zahl, Schrift enthält.

Automatisch generierte Beschreibung

Abbildung 4: Vorteilerfaktoren WatchDog

Ein Bild, das Text, Screenshot, Zahl, Schrift enthält.

Automatisch generierte Beschreibung

Abbildung 5: Vorteilerfaktoren Timer0

Ein Bild, das Text, Screenshot, Display, Software enthält.

Automatisch generierte Beschreibung

Abbildung 6: Codeausschnitt Berechnung Vorteiler

# Verarbeitung Befehle

## Auswertung Opcode

Der Opcode in einem Command ist wie bereits erwähnt unterteilt in High- und Low-Byte.

Zur Auswertung des Destinationbit und f-Register eines Opcodes werden Masken verwendet. Das Encoding eines Opcodes kann aus dem Datenblatt des PIC entnommen werden.

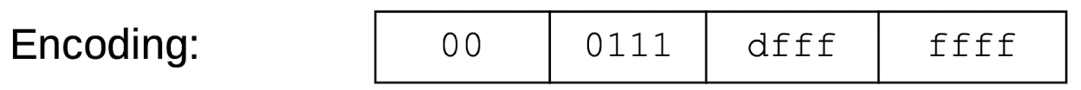


Abbildung 7: Opcode Encoding

In (Abbildung 2: Opcode Encoding) sieht man beispielweise das Encoding des Befehles ADDWF. Um das Destinationbit zu isolieren wird das Low-Byte mit der Maske 128 (1000 0000) logisch UND verknüpft. Dies sorgt dafür, dass die ersten 7 Bit 0 sind und das 8. Bit, welches das Destinationbit repräsentiert unverändert bleibt. Wollen wir also nun wissen, ob das Destinationbit gesetzt war, prüfen wir ob dieses dem Wert 128 entspricht, dann ist es gesetzt, ansonsten nicht.

Ein Bild, das Text, Schrift, Screenshot, Typografie enthält.

Automatisch generierte Beschreibung

Abbildung 8: Isolation von Desitinationbit und f-Register

## InstructionDecoder

Der InstructionDecoder dient dazu, aus dem 14Bit Opcode auszulesen, um welchen der Befehle des PIC es sich handelt. Jedem Befehl ist ein eindeutiger Opcode zugewiesen. Manche Befehle können allein anhand des High-Byte identifiziert werden. Jedoch gibt es auch Befehle, die das gleiche High-Byte aufweisen. Bei diesen ist eine Betrachtung des Low-Byte notwendig.

Der InstructionDecoder ist in als eine Funktion InstructionDecoder() realisiert, die ein Objekt vom Typ Command übergeben bekommt.

Die genaue Funktionsweise wird im Folgenden anhand des Befehles ANDWF erläutert. Der 14Bit Opcode kann aus dem Datenblatt des PICs entnommen werden.



Abbildung 9: 14Bit Opcode ANDWF Befehl

Das High-Byte des Opcode von ANDWF ist eindeutig. Hier muss also lediglich das High-Byte überprüft werden, um auf diesen Befehl schließen zu können. Entspricht also das High-Byte eines Commands 0000 0101 (dezimal: 5) handelt es sich um diesen Befehl.

Im InstructionDecoder wird dies anhand eines Switches realisiert.

Ein Bild, das Text, Schrift, Screenshot enthält.

Automatisch generierte Beschreibung

Abbildung 10: Ausschnitt InstructionDecoder

# Implementierung

## Befehlsumsetzung

### MOVF

MOVF speichert den Wert in Adresse-f Destinationbit abhängig ins w Register oder wieder in die Adresse-f. Das Zeroflag wird gesetzt, wenn der Wert gleich 0 ist.

Der Opcode dieses Befehls lautet: **00 1000 dfff fff**.

Die Adresse f kann durch eine UND Verknüpfung mit der Maske 00 0000 0111 1111 isoliert werden. Das Desinationbit mit 00 0000 1000 000.

Die Funktion CheckZFlag() erhält einen Wert, und prüft ob das Ergebnis 0 ist. Sollte das Ergebnis 0 sein, wird das Z-Flag gesetzt.

Die Funktion BankAddressResolution() bekommt eine Adresse und gibt entsprechend dem RP0 Bit, welches darüber entscheidet, ob auf Bank0 oder Bank1 zugegriffen wird, die korrekte Adresse des Registers auf das zugegriffen werden will, zurück.

Ein Bild, das Text, Screenshot, Visitenkarte, Schrift enthält.

Automatisch generierte Beschreibung

Abbildung 11: Ablaufdiagramm Befehl MOVF

Ein Bild, das Text, Screenshot, Schrift enthält.

Automatisch generierte Beschreibung

Abbildung 12: Implementierung MOVF-Befehl

### CALL

Dieser Befehl ruft ein Unterprogramm auf und legt den aktuellen Programmcounter auf den Stack. Sobald das Unterprogramm durchgeführt wurde, wird anhand des PC wieder an die Ursprungsstelle zurückgekehrt.

Der Opcode ist wie folgt aufgebaut: **10 0kkk kkkk kkkk**. K beschreibt die Adresse im Programmspeicher, an die beim CALL gesprungen werden soll.

Bevor der CALL durchgeführt wird, wird jedoch noch das High-Byte der Zieladresse (k2) durch das 3. und 4. Bit des PCLATH erweitert. Dies ermöglicht das Springen an höhere Adressen.

Die Funktion **SetPCFromBytes(byte high, byte low)** setzt den Programmcounter aus den zwei Bytes high und low. Hier ist zu beachten, dass die Zusammensetzung LittleEndian ist. Also wird das Low-Byte an der Stelle des High-Bytes positioniert und umgekehrt.

Die Funktion SetPCLfromPC() setzt das Low-Byte des Programmcounter wieder in das PCL-Register, sodass der PC und das PCL wieder synchron sind.

Am Ende wird noch ein Cycle übersprungen, da es sich bei einem CALL um einen 2-Cycle Befehl handelt.

Ein Bild, das Text, Screenshot, Software, Schrift enthält.

Automatisch generierte Beschreibung

Abbildung 13: Codeausschnitt Implementierung CALL

### GOTO

Der Befehl GOTO springt an eine Adresse im Programm, speichert sich aber im Gegensatz zu dem CALL-Befehl nicht die Rücksprungadresse.

Der Opcode dieses Befehls sieht wie folgt auf: **10 1kkk kkkk kkkk**.

Auch hier wird noch das High-Byte der Zieladresse (k2) durch das 3. und 4. Bit des PCLATH erweitert. Dies ermöglicht das Springen an höhere Adressen.

Am Ende wird noch ein Cycle übersprungen, da es sich bei einem GOTO um einen 2-Cycle Befehl handelt.

Ein Bild, das Text, Screenshot, Schrift, Software enthält.

Automatisch generierte Beschreibung

Abbildung 14: Codeausschnitt Implementierung GOTO

### ANDWF/ADDWF

## BankAddressResolution

Rasufinden von Adresse in Register abhängig von RP0 Bit => Adresse wird zurückgegeben +80 bei RPO (Bank0 oder Bank1)

## DirectionalWrite

* Schrieben in Register abhängig von destination bit

### Watchdog

### Interrupts

# Entwurfsmuster

## MVVM

Bei der Umsetzung des Simulators wurde auf die Einhaltung des MVVM-Entwurfsmuster geachtet.

Dieses besagt, dass UI-Elemente, sowie die Werte, die in der UI angezeigt werden, nicht direkt im ausführbaren Code (dem Model) verändert werden, sondern jegliche in der UI (dem View) angezeigten Informationen in einer zusätzlichen Klasse (dem ViewModel) gespeichert und bei Bedarf verändert werden.

Die Nutzung von Bindings ermöglicht das dynamische reagieren auf Änderungen im ViewModel, was eine Synchronisation, der in der UI angezeigten Daten mit denen im ViewModel ermöglicht.

# Fazit