# PIC16F8X Simulator

Inhaltsverzeichnis

[PIC16F8X Simulator 1](#_Toc164256713)

[Einleitung Simulator 1](#_Toc164256714)

[Funktionen der Benutzeroberfläche 1](#_Toc164256715)

[Registeraufbau 1](#_Toc164256716)

[Stack 1](#_Toc164256717)

[Programmspeicher 1](#_Toc164256718)

[Auswertung Opcode 2](#_Toc164256719)

[Instructiondecoder 2](#_Toc164256720)

[Befehlsumsetzung 3](#_Toc164256721)

[Ablauf einer Simulation 3](#_Toc164256722)

[Einlesen der LST-Datei 3](#_Toc164256723)

[Abbildung 1: Codeausschnitt SetRegister() 2](#_Toc164256724)

[Abbildung 2: Opcode Encoding 2](#_Toc164256725)

[Abbildung 3: Isolation von Desitinationbit und f-Register 2](#_Toc164256726)

[Abbildung 4: 14Bit Opcode ANDWF Befehl 3](#_Toc164256727)

[Abbildung 5: Ausschnitt InstructionDecoder 3](#_Toc164256728)

## 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

## 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 1: Codeausschnitt SetRegister()

## Stack

## Programmspeicher

Der Programmspeicher wurde durch eine Liste aus Objekten vom Typ Command realisiert. Ein Command ist 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).

## 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.

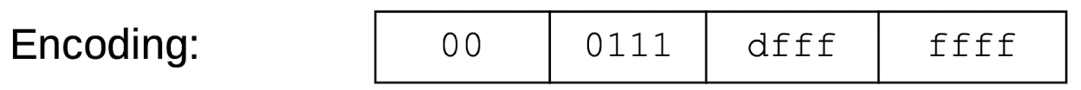


Abbildung 2: Opcode Encoding

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 3: 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 alleine 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 4: 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 5: Ausschnitt InstructionDecoder

## Befehlsumsetzung

ADDWF

## Ablauf einer Simulation

## Einlesen der LST-Datei

Fazit