

Projektdokumentation

PIC16F84-Simulator

Daniel Rittershofer

Valerio Cocco

Kurs

TINF20B3

Datum:

30.05.2022

Inhaltsverzeichnis

1	Vorwort	1
2	Simulatoren	1
3	Funktionsweise	1
3.1	Gewählte Technologien	1
3.2	Einlesen von Programmen	1
3.3	Pic-Objekt	2
3.4	Befehle	2
3.4.1	Befehlsdecodierung	2
3.4.2	Befehlsmodellierung	3
3.5	Hauptspeicher	3
3.6	Programmzähler	4
3.7	Timerfunktion	4
3.8	Interrupts	4
3.9	Sleep	5
3.10	Watchdog	5
3.11	EEPROM	5
3.12	I/O Ausgangslatch	7
3.13	Hardwareanbindung	7
4	Zusammenfassung	7
4.1	Ergebnis	7
4.2	Persönliches Fazit (Valerio Cocco)	8
4.3	Persönliches Fazit (Daniel Rittershofer)	9

1 Vorwort

Die Dokumentation orientiert sich an der Reihenfolge der Aufgaben im Bewertungsschema. Alle Testprogramme funktionieren und die Hardwareansteuerung wurde umgesetzt.

2 Simulatoren

Entwickelt wurde ein Simulator für den PIC16F84: Also eine Software, die das Verhalten dieses Mikrocontrollers nachbildet, um PIC16F84 Assemblerprogramme auf einem x86 Computer in einer grafischen Oberfläche Ausführen zu können. Hierbei wurde nicht das genaue Verhalten implementiert, mit allen internen Zuständen, sondern das Ergebnis nach außen.

3 Funktionsweise

3.1 Gewählte Technologien

Die Software ist als Webanwendung umgesetzt. Es gibt also zwei Teile, in die die Anwendung unterteilt werden kann: Front- und Back-End. Das Back-End ist in der Programmiersprache C# geschrieben, mit der ~90 % des Codes geschrieben wurde. Die Logik im Front-End in JavaScript. Zum Beschreiben der grafischen Oberfläche wird HTML verwendet. Es wird das Blazor¹ Web-Framework verwendet. Der Grund für die Wahl von Blazor ist, diese (für uns unbekannte Technologie) beim Entwickeln dieses Projekts kennenzulernen.

3.2 Einlesen von Programmen

Das Einlesen der LST-Dateien funktioniert über einen Parser im Front-End, der mit dem Parsergenerator Tree-sitter² erstellt wurde. Tree-sitter generiert aus einer LR(1) Grammatik C-Code für einen Parser, der zu WebAssembly kompiliert wird, ein Bytecode-Standard der W3C zum Ausführen von Programmen innerhalb des Webbrowsers. Die Anbindung an die JS-API des Browsers funktioniert über eine Bibliothek³ des Tree-sitter Projekts. Das ist der Grund dafür, dass dieser Teil der Logik im Front-End ist.

```
1 rules: {  
2     source_file: $ => repeat($.row),  
3     row: $ => seq(  
4         optional($.instruction),  
5         $.row_number,  
6         repeat(choice($.mnemonic, $.other)),  
7         optional($.comment),
```

¹<https://dotnet.microsoft.com/en-us/apps/aspnet/web-apps/blazor>

²<https://tree-sitter.github.io/tree-sitter/>

³https://github.com/tree-sitter/tree-sitter/tree/master/lib/binding_web

```

8      /\n/
9      ),
10     instruction: $ => seq(
11         $.instruction_number,
12         $.instruction_code
13     ),
14     instruction_number: $ => /[0-9a-fA-F]{4}/,
15     // ...
16 }

```

Auszug der Grammatik für LST-Dateien (tree-sitter-pic/grammar.js).

Der Parser baut einen konkreten Syntaxbaum auf. Durch Traversieren des Baumes wird der LST-Text (zum Anzeigen mit Syntaxhervorhebung) in HTML-Elemente verpackt, weiter werden durch das Traversieren die relevanten Informationen für den Simulator extrahiert:

- Position im Programmspeicher
- Opcode
- Zeilennummer im Quellcode (für Breakpoints)

Diese Informationen werden an das Back-End (den C# Teil) übergeben, der damit ein Pic-Objekt initialisiert, das Hauptobjekt des Simulators.

3.3 Pic-Objekt

Das Hauptobjekt im Simulator ist vom Typ Pic. Es enthält alle simulierten Elemente des PIC16F84, wie z. B. den Hauptspeicher, Programmspeicher und das W-Register.

3.4 Befehle

3.4.1 Befehlsdecodierung

Die Befehlsdecodierung funktioniert über das Interpretieren der geparsen Opcode-Zeichenketten als Zahlen. Diese werden wieder als Strings dargestellt mit Binärdarstellung (der Zahlen), für einen Vergleich vom Stringanfang, um den Befehl zu bestimmen.

```

1  if (binaryString.StartsWith("000111"))
2      return new ADDWF(binaryString, pic);
3  if (binaryString.StartsWith("000101"))
4      return new ANDWF(binaryString, pic);
5  if (binaryString.StartsWith("0000011"))

```

Logik zum Decodieren der Opcode-Strings (Simulation/InstructionDecoder.cs).

3.4.2 Befehlsmodellierung

Befehle werden durch Klassen modelliert (z. B. ADDWF). Sie erben alle von der abstrakten Klasse Instruction.

Programmspeicher Der Programmspeicher ist ein Array vom Typ Instruction, das alle decodierten Befehle in Reihenfolge der LST-Datei enthält.

Instruction-Objekte haben eine Referenz auf ihr zugehöriges Pic-Objekt und verändern durch Aufruf ihrer Execute-Methode den Zustand von diesem; z. B. die Flags im Statusregister.

```

1 public override int Execute()
2 {
3     Pic.WRegister = k;
4     Pic.IncreaseProgramCounter();
5     return 0;
6 }

```

Execute-Methode vom MOVLW-Befehl (Simulation/Instructions/LiteralInstructions/MOVLW.cs)

Befehlsabarbeitung Die Befehlsabarbeitung funktioniert über eine Schleife, die mit dem Programmzähler (als Index) den aktuellen Befehl aus dem Programmspeicher-Array holt. In dieser Schleife werden auch die Cycles gezählt. Zwei Cycle Anweisungen werden beachtet. Die Cycles werden mit der einstellbaren Frequenz für die Laufzeitberechnung verwendet.

3.5 Hauptspeicher

Der Hauptspeicher ist als Klasse Memory umgesetzt, die ein Array vom Typ uint für die Register enthält. Das Lesen und Schreiben in den Hauptspeicher funktioniert über spezielle Methoden: ReadRegister und WriteRegister. In diesen Methoden werden Bedingungen wie die Adressspiegelung für spezielle Register umgesetzt.

```

1 case 0: // Indirect addr
2 case 0x80:
3     if (FSR == 0) break; // prevent infinite recursion
4     WriteRegister(Registers[4], value);
5     break;
6 case 1: // TIMER0
7     if (!ReadRegister(0x81).IsBitSet(3))
8     {
9         // IF Prescaler is assigned to TMR0 (PSA = 0)
10        _pic.ResetScaler();
11    }

```

Teil der Switch-Case in der WriteRegister-Methode

Weiter gibt es zwei extra Methoden für das Lesen und Schreiben, die das RP0-Bit beachten.

3.6 Programmzähler

Der Programmzähler wird in der Variable `_programCounter` in `Pic` gespeichert. Das Schreiben funktioniert über einen Setter, der immer das `Pcl-Register` aktualisiert. Falls in das `Pcl-Register` geschrieben wird, wird die `_programCounter` Variable aktualisiert (von der `Memory-Klasse` aus).

PCLATH Der Wert im `PCLATH-Register` wird bei jedem `GOTO` und `CALL` Befehl beachtet. Hier wird in der `Execute-Methode` die `_programCounter` Variable aktualisiert. Die Beachtung für z. B. ein `ADDWF` wird auch in der `Memory-Klasse` umgesetzt, wenn in das `PCL-Register` geschrieben wird.

3.7 Timerfunktion

Der Timer wurde zum Teil in der `Pic Klasse` und zum Teil in der `PortA Klasse` realisiert. Zunächst musste eine `Prescaler Variable` erstellt werden, welche sich der `Timer` und `Watchdog` teilen. Hierzu wurde zunächst eine Art `decoder Funktion` mit dem Namen `GetScalerRate()` erstellt. Diese Wertet automatisch die Werte welche in `PS0`, `PS1`, `PS2` und `PSA` stehen aus und gibt den entsprechenden Wert für den `Prescaler` zurück. Immer wenn die `Simulation` einen `Cycle` ausführt, wird das `T0CS Bit` überprüft und gegebenenfalls die `TimerStep()` Funktion aufgerufen. Diese wiederum überprüft anhand des `PSA -Bit` ob der `Timer` den `Prescaler` zugewiesen hat oder nicht. Wenn der `Timer` den `Prescaler` zugewiesen hat, so wird dieser zunächst dekrementiert, bis dieser den Wert 0 erreicht. Danach wird der `Prescaler` zurückgesetzt und die Funktion `IncreaseTimer()` aufgerufen. Sollte der `Prescaler` nicht dem `Timer` zugewiesen sein wird diese Methode sofort aufgerufen. `IncreaseTimer()` holt sich zunächst den aktuellen Wert aus dem `Speicher des Pics`. Danach wird dieser inkrementiert und es wird überprüft, ob es einen Überlauf gab. Ist dies gegeben, so wird der `Timerwert` auf die untersten 8 Bit maskiert. Außerdem wird überprüft, ob der `Timer0 Interrupt` aktiv ist durch das Bit `T01E`. Ist auch dies gegeben wird die `Interrupt Routine`, also `Interrupt()`, ausgelöst und das entsprechende Flag in `T0IF` gesetzt. Unabhängig davon, ob es einen Überlauf gab oder nicht, wird der neue `Timer-Wert` wieder in den `Speicher des Pics` geschrieben. Natürlich werden die aktuellen Werte des `Prescalers` und des `Timers` auch noch in der `UI` angezeigt. Unabhängig von all dem wird in der `PortA Klasse` beim togglen des `RA4 Pins` über das `T0CS Bit` geprüft, ob `RA4` als `Clock Source` verwendet wird. Ist dies der Fall so wird über das `T0SE Bit` geprüft, ob es sich um eine steigende oder eine fallende Flanke handeln soll. Demensprechend wird dort ebenfalls die `TimerStep()` Funktion des `Pics` aufgerufen.

3.8 Interrupts

Abgesehen von dem gerade beschriebenen `Timer Interrupts` gibt es noch 3 weitere `Interrupts`. Zunächst gibt es noch den `EEPROM Interrupt`. Dieser ist implementiert, indem am Ende des `EEPROM Schreibvorganges` das entsprechende `EEIE Bit` überprüft und gegebenenfalls die `Interrupt()` Methode aufgerufen wird. Weiterhin gibt es den `RB0/INT Interrupt`. Hierbei wird bei einem `Toggle` an `RB0` zunächst durch

das INTE Bit überprüft, ob dieser aktiv ist. Danach wird über das NTEDG Bit der Flankenmodus ausgelesen. Danach wird überprüft ob die aktuelle Flanke dieser entspricht und gegebenenfalls die Interrupt() Methode aufgerufen. Außerdem wird das INTF Flag gesetzt, um zu signalisieren um welche Art von Interrupt es sich handelt.

Dasselbe wird auch bei den Pins RB4-7 getan jedoch ohne die Selektierung der Flanke. Hier wird jedoch das RBIF Bit als Flag gesetzt.

Die Interrupt() Methode pusht zunächst den aktuellen Programcounter wert auf den Stack und setzt ihn dann auf den Wert 4. Im Falle das der Pic Schläft werden stattdessen das TO und PD Bit entsprechend gesetzt und die IsSleeping Variable auf false gesetzt.

3.9 Sleep

Sleep wurde zunächst anhand von der Dokumentation implementiert, um die Entsprechenden Bits in den Registern zu setzen. Zusätzlich wird noch die interne Variable des Pics mit dem Namen IsSleeping gesetzt. In der Step() Funktion des Pics wird diese Variable überprüft und nur falls der Pic nicht schläft wird die nächste Instruction ausgeführt. Die IsSleeping Variable kann nur durch einen Interrupt oder einen Reset wieder auf false gesetzt werden.

3.10 Watchdog

Der Watchdog hat eine eigene wdtCheck() Funktion in der Pic Klasse welche wie die Cycle() Methode beim durchlauf jedes Steps ausgerufen wird. Während in der Cycle() Methode die entsprechende Zählvariable des Watchdos hochgezählt wird, überprüft die wdtCheck() Methode nur ob die Laufzeitsumme der bisherigen Cycles einen Wert über 18ms ergibt. Ist dies der Fall wird über das PSA Bit überprüft, ob der Watchdog den Prescaler zugewiesen bekommen hat. Ist dies der Fall wird dieser dekrementiert. Sobald dieser den Wert 0 erreicht oder falls der Prescaler nicht dem Watchdog zugewiesen ist, wird die WatchDogReset() Methode aufgerufen. Die WatchDogReset() Methode überprüft zunächst, ob der Pic schläft. In diesem Fall wird der Pic aufgeweckt, das PD und TO Bit entsprechend gesetzt und der Programcounter inkrementiert. Schläft der Pic nicht so wird stattdessen ein Reset ausgeführt und mögliche EEPROM Schreibvorgänge werden terminiert.

3.11 EEPROM

Die EEPROM wird mit der Klasse EEPROM dargestellt. Wie bei Memory sind die Speicherstellen über ein uint-Array abgebildet. Das Lesen aus EEPROM wird bei jedem setzen des Read Control Bits (im EECON-Register) ausgelöst.

Lesen Das Prüfen erfolgt über die Akzessoren für das EECON-Register in der EEPROM Klasse, die von der Klasse Memory immer verwendet werden, wenn auf das Register zugegriffen wird.

Schreiben Das Schreiben wird ähnlich wie das Lesen über das »Write Control Bit« ausgelöst. Zusätzlich muss eine »write sequence« davor erfolgt sein. In dieser Implementierung löst das Setzen des »Write Control Bit« nur direkt nach der »write sequence« ein Schreiben aus. Die Prüfung für diese Bedingung erfolgt in der Schleife für die Befehlsabarbeitung. In der Schleife wird immer `EEPROM.CheckInstruction` Methode aufgerufen.

```

1 private static readonly Instruction[] _requiredInstructionSequenceForWrite =
2 {
3     new MOVWF(9, 1), // EEC0N2
4     new MOVLW(0xAA),
5     new MOVWF(9, 1), // EEC0N2
6     new BSF(8, 1), // Set WR bit → begin write
7 };
8
9
10 public void CheckInstruction(Instruction? instruction)
11 {
12     if (instruction == null)
13     {
14         _nextRequiredInstructionForWrite = 0;
15     }
16     else
17     {
18         if (_nextRequiredInstructionForWrite ==
19 → _requiredInstructionSequenceForWrite.Length)
20         {
21             _nextRequiredInstructionForWrite = 0;
22         }
23         Instruction expected =
24 → _requiredInstructionSequenceForWrite[_nextRequiredInstructionForWrite];
25         if (instruction.Equals(expected))
26         {
27             _nextRequiredInstructionForWrite++;
28         }
29         else
30         {
31             _nextRequiredInstructionForWrite = 0;
32         }
33     }
34 }

```

Code zum Prüfen der »write sequence«

Die Bedingung zum Auslösen des Schreibens entspricht damit `_nextRequiredInstructionForWrite == 4`.

Programmierzeit Die Programmierzeit wird sehr einfach über die Laufzeitberechnung umgesetzt. Wenn geschrieben wird, wird die aktuelle Laufzeit in einer Variable gespeichert. In der Befehlsabarbeitungsschleife wird geprüft, ob die Differenz zwischen aktueller und gespeicherter Laufzeit größer gleich 1 ms ist.

3.12 I/O Ausgangslatch

Die Wirkung des Ausgangslatch wird in der Klasse Port umgesetzt. Sie bietet Akzessoren für einen internen Zustand, der dem Zustand widerspiegelt der per Software (im Assembler-Code) gelesen wird und einem externen Zustand, der dem physikalischen Wert am Pin entspricht. Der »physikalische« Zustand der Pins wird in einer extra Variable gespeichert.

TRIS Output Wenn im Tris-Register ein Pin auf Output eingestellt ist, sind interner und externer Zustand gleich. Bei jedem Aufruf eines Akzessor wird für Output-Pins der Wert im Port-Register auf die Pin-Variable (»physikalisch«) geschrieben. So wird bei einem Umschalten auf Output auch direkt der Wert im Register auf die Pins ausgegeben.

TRIS Input Für einen Pin im Input Modus löst das Setzen eines Werts über den Akzessor für den internen Zustand – ein Schreiben auf das Port-Register aus. Gelesen wird jedoch immer von der Variable für den physikalischen Zustand (Pin). Das Setzen über den Akzessor für den externen Zustand verwendet genau diese.

3.13 Hardwareanbindung

Zur Hardware Anbindung wird nach jedem Cycle die Methode serialHandler.Write() aufgerufen. Der Serialhandler baut über die aus der Doku gegebenen Parameter (Baudrate 4800, Parity None, Databits 8, Stopbits 1) eine Verbindung zur externen Hardware auf. Der Write Methode wird ein zuvor per GenerateSerialPayload() generiertes Payload übergeben. Das Payload besteht entsprechend der Doku aus High- und Lowbytes des Tris A/B und Port A/B Registers und wird mit einem CarriageReturn abgeschlossen. Nachdem das Payload abgeschickt wurde, wird überprüft, ob eine mindestens 5 Byte lange Antwort entgegenkam. Falls dies der Fall ist, werden die Halbbytes wieder zu vollen Bytes verknüpft und die entsprechenden Register geschrieben.

4 Zusammenfassung

4.1 Ergebnis

Das Ergebnis dieses Projekts ist ein Simulator, der jedes Testprogramm korrekt ausführen kann. Über die grafische Oberfläche wird jeder Zustand, die der Nutzer am PIC16F84 nachvollziehen kann angezeigt. Verhalten wie der »weak pull-up« an PP0RTB Pins wurden nicht per Software nachgebaut.

The screenshot displays the 16F84-Simulator interface. The main window shows assembly code for a PIC16F84. The code includes comments in German and assembly instructions like `list c=132`, `indirect equ 0`, `status equ 83h`, `fsr equ 84h`, `ra equ 85h`, `rb equ 86h`, `rp0 equ 5`, `device 16F84`, `org 0`, `start`, `btfsc status,rp0`, `goto links`, `goto rechts`, `goto links`, `goto ende`, and `goto ende`. The right panel shows the File Register (SFR & GPR) with Bank 0 and Bank 1 registers. The Timing section shows Quartz at 4 MHz, Prescaler Rate at 1:128, and TMR0 at 00h. The Stack section shows the current pointer at 0000h. The SFR (Bits) section shows various status and control bits. The Port A and Port B sections show the TRIS and Pin configurations.

Erwähnenswert ist, dass der Simulator auch komplett ohne Server lauffähig ist (als standalone WebAssembly), wenn das Projekt entsprechend kompiliert wird.

4.2 Persönliches Fazit (Valerio Cocco)

Die Entwicklung am Simulator hat mir persönlich gut gefallen. Zuvor habe ich nichts Vergleichbares programmiert. Durch dieses Projekt konnte ich mein Verständnis über die Arbeitsweise von Mikrocontrollern und damit Computern verbessern und festigen.

Weiter ist mir noch mal klar geworden, dass Computer aus einer sehr abstrakten Sichtweise als Zustandsautomat betrachtet werden können. In diesem Fall der Pic mit all seinen Komponenten, die den Zustand widerspiegeln (Register, Timer, Watchdog, usw.) und Bedingungen/Eingaben wie, Zyklen, Befehle, Pin Signale, etc. die Zustandsänderungen bewirken.

Besonders interessant/neu für mich war der sehr hardwarenahe Teil wie Interrupts, Data Latches und EEPROM-Programmierzzeit.

Neben diesen Aspekten habe ich auch viel über die verwendeten Werkzeuge wie C# und Blazor gelernt.

Mit der Zusammenarbeit zwischen meinem Partner und mir, bin ich sehr zufrieden. Wir konnten die Aufgaben gut aufteilen, selbstständig arbeiten, fertiggestellte Teile zusammen besprechen und uns bei Problemen gegenseitig unterstützen.

Der zeitliche Umfang des Projekts war sehr groß. Tatsächlich hätten wir auch Aufgaben weglassen können für eine 1.0. Ich denke aber, die Tatsache, dass wir alle Aufgaben umgesetzt haben, zeigt, dass wir nicht ungern am Projekt gearbeitet haben. Für das nächste Mal nehme ich mir vor mich

auf die wichtigsten Aspekte eines solchen Projekts zu fokussieren, um Zeit zu sparen. Z. B. hätte ich Angelegenheiten wie das Aussehen der Benutzeroberfläche weiter hinten anstellen können.

4.3 Persönliches Fazit (Daniel Rittershofer)

Ich persönlich finde das Projekt eine sehr gute Aufgabe, um die internen Funktionsabläufe und das Arbeiten des besser zu verstehen. Durch das Projekt habe ich nicht nur gelernt wie die einzelnen Befehle, welche der PIC beherrscht funktionieren, sondern auch, wie der PIC Interrupts und Timer realisiert und handhabt. Zu Beginn des Projekts war mir noch gar nicht so richtig klar, was wir alles benötigen, um den PIC vollständig zu simulieren aber durch die Aufeinander aufbauenden Aufgaben konnte unser Projekt immer um die fehlenden Bausteine erweitert werden. Beispielsweise der Stack, welcher als Ringspeicher realisiert werden musste. Für die ersten Programme konnte dieser ignoriert werden, doch als er dann benötigt wurde konnten wir mithilfe eines selbst erstellten RingBuffer Datentyps, da C# diesen Datentyp nicht bereitstellt, einfach realisieren und einbauen. Ebenfalls hilfreich fand ich es, dass einem im Laufe der Programme neue Fehler auffallen, welche man zuvor übersehen hat. Beispielsweise haben wir uns Anfangs keine größeren Gedanken gemacht, was passiert, wenn es einen Overflow beim addieren oder anderen Operationen gibt. Erst als es bei einem Programm zu einem Überlauf kam und das erwartete Ergebnis nicht übereinstimmte fiel uns dies auf und wir konnten es durch eine Maskierung auf die niedrigsten 8 Bit beheben. Beim nächsten mal eines solchen Projekts würde ich nicht viel verändern, da ich sehr zufrieden mit unserem Endergebnis und unseren Arbeitsprozessen war. Lediglich zu Beginn des Projekts würde ich mehr Zeit in einen genaueren Plan investieren.