DHBW
Duale Hochschule
Baden-Württemberg
Karlsruhe

Dokumentation PIC Simulator

Abgabedatum	12. Mai 2025
Kurs	TINF23B3
Autoren	Luca Müller
	Leander Gantert

Inhaltsverzeichnis

1 Einleitung **1**
- 1.1 Über dieses Dokument . . . 1
- 1.2 Projektziele . . . 1

2 Grundlagen **2**
- 2.1 Grundsätzliche Arbeitsweise eines Simulators . . . 2
- 2.2 Vor- und Nachteile einer Simulation . . . 3

3 Benutzeroberfläche **5**
- 3.1 Hauptfenster . . . 5
- 3.2 Menüleiste . . . 6
- 3.3 Debugging-Tools . . . 6
- 3.4 Visualisierung und Interaktivität . . . 6
- 3.5 Zusammenfassung . . . 6

4 Technische Details und Funktionsweise **7**
- 4.1 Warum Python gewählt wurde . . . 7
- 4.2 Konzept des PIC Simulators . . . 7
- 4.3 Ausführung eines Befehlszyklus . . . 8
- 4.4 Detailierte Funktionsweise . . . 9
  - 4.4.1 subwf . . . 9
  - 4.4.2 call . . . 10
  - 4.4.3 readRegister . . . 13

5 Fazit **15**
- 5.1 Herausforderungen im Entwicklungsprozess . . . 15
  - 5.1.1 Integration von UI und Backend . . . 15
  - 5.1.2 Kontinuierliche Weiterentwicklung . . . 16
- 5.2 Lessons Learned . . . 16

Abbildungsverzeichnis

4.1 Programmablauf einer Befehlsausführung 9

Liste der Algorithmen

4.1 Methode subwf aus dem Arithmetic Logic Unit (Arithmetisch-logische Einheit) (ALU) Modul 10
4.2 Ausschnitt der CALL-Befehlsimplementierung im Central Processing Unit (Zentrale Verarbeitungseinheit) Modul 11
4.3 Ausschnitt der Decode-Methode im Decode-Modul 12
4.4 Ausschnitt der readRegister-Methode im Speichermodul 13

Abkürzungsverzeichnis

ALU Arithmetic Logic Unit (Arithmetisch-logische Einheit) iii
CPU Central Processing Unit (Zentrale Verarbeitungseinheit) 7
GUI Graphical User Interface (Grafische Benutzeroberfläche) 7

Kapitel 1

Einleitung

1.1 Über dieses Dokument

Dieses Dokument handelt von einem PIC Simulator und seinen Funktionen sowie deren Implementierung. Die Dokumentation richtet sich an Benutzer und Entwickler, die den Simulator verstehen möchten.

1.2 Projektziele

Der PIC Simulator wurde entwickelt, um auch in Abwesenheit von physischer Hardware Programme ausführen zu können. Die Hauptfunktionen umfassen:

- Simulation von PIC-Mikrocontroller-Programmen
- Visualisierung des Programmablaufs
- Debugging-Funktionalitäten

Kapitel 2

Grundlagen

2.1 Grundsätzliche Arbeitsweise eines Simulators

Ein Simulator ist eine Software, welche das Verhalten eines realen Systems nachbildet. Im Fall eines Mikrocontrollers wie dem PIC wird die Hardwarearchitektur – bestehend aus Central Processing Unit (CPU), Speicher, Registern und Peripheriekomponenten – virtuell nachgebildet, um Programme auszuführen, als würden sie auf der physischen Hardware laufen. Die grundsätzliche Arbeitsweise eines solchen Simulators umfasst typischerweise die folgenden, eng miteinander verzahnten Schritte:

- **Laden des Programms**: Der erste Schritt besteht darin, das zu testende Programm in den simulierten Speicher des Mikrocontrollers zu laden. Dieses Programm liegt in Form einer .LST-Datei vor, die sowohl den Maschinencode als auch die ursprünglichen Assemblerinstruktionen enthält.

- **Dekodieren der Instruktionen**: Nachdem das Programm geladen ist, beginnt der Simulator mit der Ausführung. Dazu liest die simulierte CPU die erste Instruktion aus dem Programmspeicher, auf die der Programmzähler (Program Counter, PC) zeigt. Jede dieser Instruktionen ist ein HEX Code, der vom Simulator dekodiert werden muss. Das Dekodieren bedeutet, den HEX Code zu analysieren, um festzustellen, welcher Befehl ausgeführt werden soll und welche Operanden dafür benötigt werden.

- **Ausführen der Instruktionen**: Nach der Dekodierung führt die simulierte CPU die entsprechende Operation aus. Dies beinhaltet die Manipulation von Daten in den simulierten Registern, das Lesen oder Schreiben von Werten im simulierten Speicher und die Aktualisierung von Statusregistern (Flags), wie z.B. dem Zero-Flag, Carry-Flag oder Overflow-Flag, basierend auf dem Ergebnis der Operation. Nach der Ausführung einer Instruktion wird der Programmzähler in der Regel inkrementiert, um auf die nächste Instruktion zu zeigen, es sei denn, es wurde ein Sprung- oder Verzweigungsbefehl ausgeführt.

- **Interaktion mit Peripherie**: Moderne Mikrocontroller verfügen über eine Vielzahl von integrierten Peripheriegeräten wie Timer/Counter, Interrupt-Controller und digitale Ein-/Ausgabe-Ports (I/O-Ports). Wenn das Programm mit diesen Peripheriegeräten interagiert (z.B. einen Timer startet, einen Wert von einem I/O-Port liest oder einen Interrupt auslöst), muss der Simulator dieses Verhalten ebenfalls nachbilden. Dies kann bedeuten, interne Zustände der simulierten Peripherie zu ändern, simulierte Zeitabläufe zu verwalten oder externe Ereignisse (z.B. ein simuliertes Signal an einem I/O-Pin) zu verarbeiten.
- **Visualisierung und Debugging-Unterstützung**: Ein wesentlicher Aspekt vieler Simulatoren ist die Fähigkeit, den internen Zustand des simulierten Systems dem Benutzer darzustellen. Dies umfasst die Anzeige der aktuellen Werte von CPU-Registern, Speicherinhalten, dem Zustand von Peripheriegeräten und des Stacks.

2.2 Vor- und Nachteile einer Simulation

Simulationen bieten zahlreiche Vorteile für den Entwicklungsprozess von Embedded Systems, bringen jedoch auch spezifische Einschränkungen mit sich, die berücksichtigt werden müssen.

Vorteile

- **Kosteneffizienz**: Einer der größten Vorteile ist die erhebliche Reduktion der Kosten. Es entfallen Ausgaben für die Anschaffung teurer Entwicklungsboards, spezifischer Messgeräte oder Prototypen-Hardware. Auch Kosten für Reparatur oder Ersatz bei Beschädigung durch fehlerhafte Programme oder Experimente entfallen. Dies ermöglicht insbesondere kleineren Teams oder Einzelentwicklern den Zugang zu Entwicklungswerkzeugen.
- **Flexibilität und schnelle Iteration**: Änderungen am Programmcode oder an der simulierten Hardwarekonfiguration, wie beispielsweise die Taktfrequenz oder der angeschlossene Peripherieumfang, können softwareseitig schnell und ohne physischen Umbau vorgenommen werden. Dies beschleunigt den Entwicklungszyklus erheblich, da verschiedene Szenarien und Konfigurationen in kurzer Zeit getestet werden können.
- **Umfassende Debugging-Möglichkeiten**: Simulatoren bieten oft weitreichendere Einblicke in das Systemverhalten als dies mit physischer Hardware möglich wäre. Entwickler können den Programmablauf schrittweise verfolgen (Single-Stepping), Haltepunkte (Breakpoints) an beliebigen Stellen im Code setzen oder den Inhalt von Registern und Speicherbereichen in Echtzeit inspizieren und modifizieren. Auch komplexe Zustände und Zeitabläufe lassen sich detailliert analysieren, was die Fehlersuche und -behebung (Debugging) signifikant erleichtert.

- **Sicherheit und Risikominimierung**: Kritische oder potenziell destruktive Programmzustände können in einer simulierten Umgebung gefahrlos getestet werden. Es besteht kein Risiko, reale Hardware durch fehlerhaften Code, falsche Spannungen oder Kurzschlüsse zu beschädigen. Dies ist besonders wertvoll beim Testen von Grenzfällen oder Fehlerbehandlungsroutinen.

- **Frühzeitige Softwareentwicklung**: Die Softwareentwicklung kann parallel zur oder sogar vor der Fertigstellung der physischen Hardware beginnen. Dies verkürzt die Gesamtentwicklungszeit (Time-to-Market) und ermöglicht eine frühere Validierung von Softwarekonzepten.

- **Automatisierte Tests**: Simulatoren lassen sich gut in automatisierte Testumgebungen integrieren. Testskripte können eine Vielzahl von Szenarien durchlaufen lassen und die Ergebnisse protokollieren, was die Qualitätssicherung verbessert und Regressionstests vereinfacht.

Nachteile

- **Eingeschränkte Genauigkeit und Realitätsnähe**: Eine Simulation ist immer eine Abstraktion der Realität. Insbesondere timing-kritisches Verhalten, analoge Komponenten oder sehr spezifische Hardwareeigenschaften (z.B. exakte Interrupt-Latenzzeiten, Stromverbrauch) können oft nur angenähert oder gar nicht simuliert werden.

- **Abweichungen zur realen Hardware**: Trotz sorgfältiger Modellierung können immer subtile Unterschiede zwischen dem Verhalten des Simulators und der realen Hardware bestehen. Dies kann an unvollständiger Dokumentation der Hardware, Vereinfachungen im Simulationsmodell oder Fehlern im Simulator selbst liegen.

- **Entwicklungs- und Wartungsaufwand für den Simulator**: Die Erstellung und Pflege eines genauen und umfassenden Simulators ist selbst ein komplexes Softwareprojekt, das Ressourcen bindet. Insbesondere bei neuen oder sehr speziellen Mikrocontrollern ist möglicherweise kein passender Simulator verfügbar oder muss erst entwickelt werden.

Kapitel 3

Benutzeroberfläche

Die Benutzeroberfläche des PIC Simulators ist so gestaltet, dass sie eine intuitive und effiziente Nutzung ermöglicht. Im Folgenden werden die wichtigsten Elemente der Benutzeroberfläche und deren Handhabung detailliert beschrieben:

3.1 Hauptfenster

Das Hauptfenster des Simulators ist in mehrere Bereiche unterteilt, die jeweils spezifische Informationen und Funktionen bereitstellen:

- **Code-Editor**: Der integrierte Code-Editor ermöglicht das Laden von Assemblerdateien. Breakpoints können durch einfaches Anklicken der Checkbox am Zeilenanfang gesetzt werden, um den Programmablauf gezielt zu analysieren.
- **Steuerelemente**: Buttons wie `Go` und `Reset` steuern die Programmausführung. Diese Steuerelemente sind übersichtlich angeordnet und leicht zugänglich.
- **Register- und Speicheranzeige**: Diese Bereiche zeigen den aktuellen Zustand der Register, Flags und des Speichers in Echtzeit an. Änderungen werden sofort visualisiert, um den Programmablauf nachvollziehbar zu machen und das Debugging zu erleichtern.
- **I/O-Visualisierung**: Die I/O-Pins des Mikrocontrollers werden interaktiv dargestellt. Benutzer können die Zustände der Pins durch Anklicken ändern, um verschiedene Szenarien zu simulieren. Ebenfalls wird angezeigt, ob ein Pin als Eingang oder Ausgang konfiguriert ist.
- **Peripheriegeräte**: Eine Übersicht der simulierten Peripheriegeräte, wie Timer und Interrupts, wird bereitgestellt. Diese Bereiche ermöglichen eine einfache Konfiguration und Überwachung der Peripheriefunktionen.
- **Statusbereich**: Der Statusbereich zeigt wichtige Informationen wie den aktuellen Program Counter (PC), den Stackpointer und die Ausführungszeit an.

3.2 Menüleiste

Die Menüleiste bietet Zugriff auf grundlegende Funktionen des Simulators:

- **Datei**: Optionen zum Laden von Quelldateien.
- **Hilfe**: Zugriff auf die Dokumentation.

3.3 Debugging-Tools

Die Benutzeroberfläche enthält leistungsstarke Debugging-Tools, die die Fehlersuche erleichtern:

- **Breakpoints**: Benutzer können Breakpoints setzen, um die Programmausführung an bestimmten Stellen zu pausieren und schrittweise fortzusetzen.
- **Speicherüberwachung**: Der Speicherinhalt kann in Echtzeit überwacht und bei Bedarf manuell geändert werden.
- **Registeranzeige**: Alle Register des Mikrocontrollers werden angezeigt, einschließlich ihrer aktuellen Werte. Die Register können ebenfalls manuell bearbeitet werden.
- **Statusflags**: Die Statusflags (z. B. Carry, Zero) werden in Echtzeit aktualisiert und angezeigt, um den aktuellen Zustand des Mikrocontrollers zu verdeutlichen.

3.4 Visualisierung und Interaktivität

Die Benutzeroberfläche bietet eine visuelle Darstellung des Mikrocontrollers und seiner Peripherie:

- **Simulation von Peripheriegeräten**: Timer, Interrupts und andere Peripheriegeräte können simuliert und überwacht werden.
- **Echtzeit-Updates**: Änderungen im Programm oder in der Hardwarekonfiguration werden sofort in der Benutzeroberfläche reflektiert.
- **Interaktive Elemente**: Benutzer können direkt mit der Simulation interagieren, z. B. durch das Ändern von I/O-Pin-Zuständen.
- **Konfiguration**: Benutzer können die Taktfrequenz des Simulators anpassen.

3.5 Zusammenfassung

Die Benutzeroberfläche des PIC Simulators kombiniert Funktionalität und Benutzerfreundlichkeit. Sie bietet alle notwendigen Werkzeuge, um Programme effizient zu testen und zu debuggen. Durch die klare Struktur und die interaktiven Elemente wird die Arbeit mit dem Simulator erheblich erleichtert.

Kapitel 4

Technische Details und Funktionsweise

4.1 Warum Python gewählt wurde

Zur Umsetzung des Projekts wurde Python als Programmiersprache gewählt, da es mehrere Vorteile bietet:

- **Einfache Syntax**: Python ermöglicht eine klare und übersichtliche Implementierung, was die Entwicklung erleichtert.
- **Schnelle Entwicklungszeit**: Dank der hohen Abstraktionsebene können Prototypen schnell erstellt und getestet werden.
- **Umfangreiche Bibliotheken**: Python bietet zahlreiche Bibliotheken, wie `PyQt` für die Graphical User Interface (Grafische Benutzeroberfläche) (GUI), welche die Entwicklung beschleunigen.
- **Plattformunabhängigkeit**: Python-Programme können auf verschiedenen Betriebssystemen ausgeführt werden.

4.2 Konzept des PIC Simulators

Der PIC Simulator ist modular aufgebaut, um eine klare Trennung der Verantwortlichkeiten zu gewährleisten. Das Herzstück des Simulators bildet die **Central Processing Unit (Zentrale Verarbeitungseinheit) (CPU)** . Sie ist für die gesamte Steuerung des Simulationsablaufs zuständig. Ihre Hauptaufgabe besteht darin, die Befehle aus dem Programmspeicher zu holen und die entsprechenden Methoden zum Dekodieren und Ausführen dieser Befehle in den anderen Modulen aufzurufen.

Die **ALU** ist spezialisiert auf die Durchführung aller arithmetischen und logischen Operationen. Wenn die CPU einen Befehl dekodiert, der eine solche Operation erfordert (z.B. Addition, Subtraktion, logisches UND), delegiert sie die Ausführung an die ALU.

Das **Speichermodul** (Memory) verwaltet die verschiedenen Speicherbereiche des simulierten PIC-Mikrocontrollers. Dazu gehören der Programmspeicher, in dem der auszuführende Code abgelegt ist, der Datenspeicher, der die Register und Variablen enthält und der Stack auf dem die Rücksprung Adressen abgelegt werden. Das Speichermodul stellt Methoden zum Lesen und Schreiben dieser Speicherbereiche bereit, die von der CPU und anderen Komponenten genutzt werden.

Der **Decoder** ist dafür zuständig, die aus dem Programmspeicher gelesenen hexadezimalen Maschinenbefehle in eine für die CPU verständliche Form zu übersetzen. Er analysiert den Opcode und die Operanden eines jeden Befehls und gibt eine strukturierte Repräsentation zurück, die der CPU mitteilt, welche Aktion ausgeführt und welche Daten dafür verwendet werden sollen.

Für die GUI wird das Framework **Qt** (über PySide6) eingesetzt. Qt ist eine plattformübergreifende C++-Bibliothek, die umfangreiche Werkzeuge zur Erstellung interaktiver Oberflächen bereitstellt. Ein fundamentales Konzept in Qt ist der **Signal-Slot-Mechanismus**. Dabei können Objekte Signale aussenden, wenn bestimmte Ereignisse eintreten (z.B. ein Button-Klick oder die Aktualisierung eines Registerwerts durch die CPU Simulation). Andere Objekte können ihre Slots – spezielle Funktionen – mit diesen Signalen verbinden. Wird ein Signal emittiert, werden automatisch alle verbundenen Slots aufgerufen. Dies ermöglicht eine entkoppelte und flexible Kommunikation zwischen den GUI Elementen und der darunterliegenden Simulatorlogik, beispielsweise um Änderungen im Zustand der CPU oder des Speichers in Echtzeit in der Oberfläche darzustellen.

4.3 Ausführung eines Befehlszyklus

Der Befehlszyklus im PIC Simulator folgt einem klar strukturierten Ablauf. Zunächst wird die Assemblerdatei analysiert, und die darin enthaltenen Befehle in ein Array geladen, das als Programmspeicher dient.

Zu Beginn eines Befehls Zyklus wird überprüft, ob sich der Mikrocontroller im Sleep-Modus befindet, die Ausführung pausiert wurde oder der aktuelle Befehl ignoriert werden soll. Wenn nicht wird der nächste Befehl aus dem Programmspeicher eingelesen und dekodiert. Der Programmzähler (Program Counter) wird daraufhin inkrementiert, um auf den nächsten Befehl zu zeigen. Basierend auf der Dekodierung wird die entsprechende Funktion ausgeführt, welche den Befehl implementiert und dabei alle relevanten Statusflags (z. B. Carry, Zero) setzt. Nach der Ausführung des Befehls wird die Laufzeit des Programms entsprechend der Befehlslänge inkrementiert. Gleichzeitig wird der Watchdog-Timer erhöht, um dessen Überwachung zu simulieren. Zum Abschluss eines Zyklus werden anstehende Interrupts geprüft und verarbeitet. Schließlich wird die Benutzeroberfläche aktualisiert, um den aktuellen Zustand des Mikrocontrollers visuell darzustellen. (Vergleich siehe Abbildung 4.1)

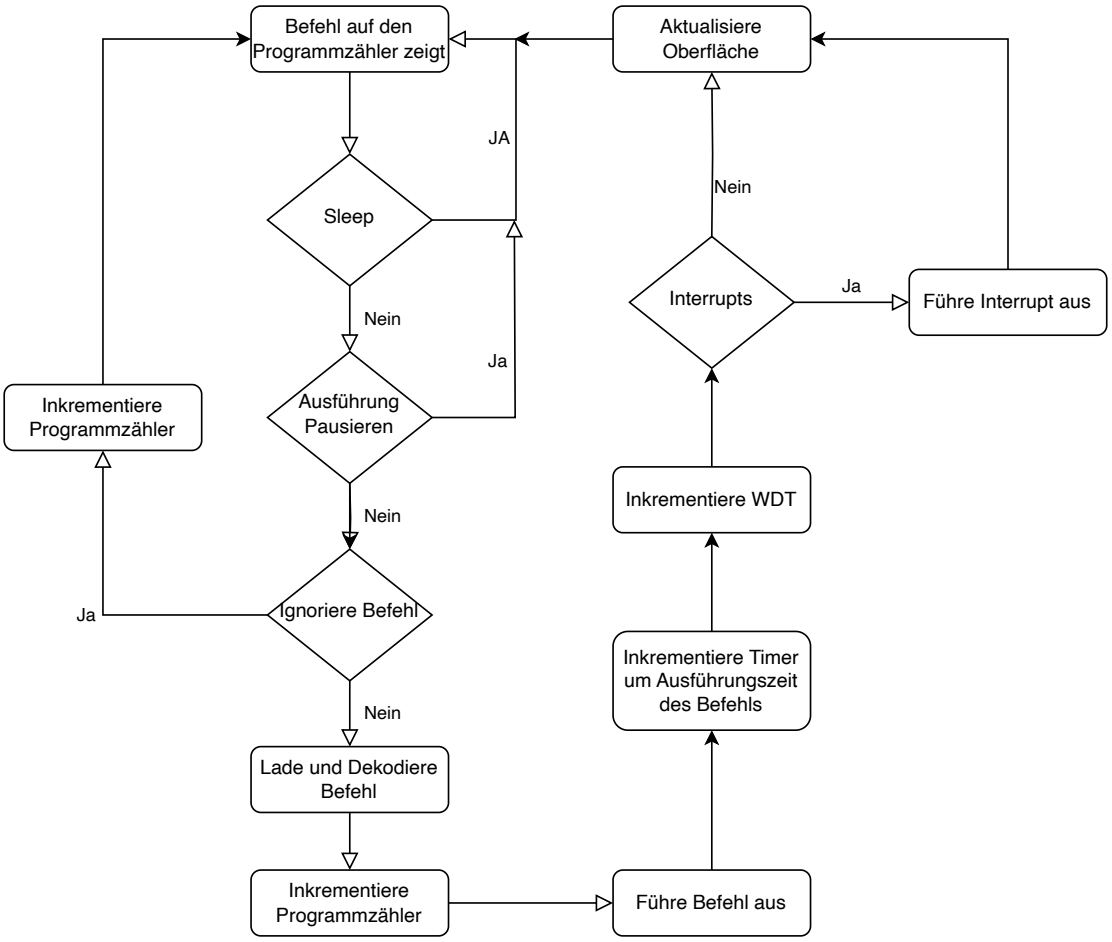

Abbildung 4.1: Programmablauf einer Befehlsausführung

4.4 Detailierte Funktionsweise

In diesem Abschnitt soll die genauere Funktionsweise verschiender Funktionen des Simulators anhand von einzelnen Beispielen erläutert werden. Dabei wird auf die einzelnen Module eingegangen und deren Funktionsweise erklärt.

4.4.1 subwf

Der Befehl `subwf f,d` (Subtract W from f) subtrahiert den Inhalt des W-Registers (Arbeitsregister) vom Inhalt des Registers `f`. Das Ergebnis dieser Subtraktion wird dann, abhängig vom Wert des Destination-Bits `d`, entweder zurück in das W-Register (wenn `d=0`) oder in das Register `f` (wenn `d=1`) geschrieben. Bei dieser Operation werden die Statusbits C (Carry), DC (Digit Carry) und Z (Zero) im STATUS-Register entsprechend

dem Ergebnis der Subtraktion beeinflusst. Wichtig zu beachten ist, dass das Carry Flag bei der Subtraktion invertiert ist.

```
1  def subwf(self, dest, reg_num):
2      value = self.memory.readRegister(reg_num)
3      result = value - self.memory.getW()
4      self.memory.setBit(0x03, 0, 0 if result < 0 else 1)
5      if result < 0:
6          result = (result * (-1)-1) ^ 0xFF
7      self.memory.setBit(0x03, 1, 0 if (value & 0x0F) - (self.
           memory.getW() & 0x0F) < 0  else 1)
8      result = int(int(bin(result),2) & int('0b11111111',2))
9      self.memory.setBit(0x03, 2, 1 if result == 0 else 0)
10
11     self.memory.writeRegister(reg_num if dest else 'w', result)
```

Algorithmus 4.1: Methode subwf aus dem ALU Modul

Erläuterung des Codes: Die Methode `subwf` (siehe Listing 4.1) simuliert den gleichnamigen PIC-Befehl:

- Zeile 2: Liest den Wert des Registers `f` (Operand `reg_num`).
- Zeile 3: Subtrahiert den Inhalt des W-Registers vom gelesenen Wert `value`.
- Zeile 4: Setzt das Carry-Flag (C). Die Logik prüft, ob das `result` negativ ist.
- Zeile 5-6: Ist das Ergebnis `result` negativ, wird es in das 8-Bit-Zweierkomplement umgewandelt (z.B. -1 zu 255).
- Zeile 7: Setzt das Digit-Carry-Flag (DC) korrekt, basierend auf einem Borrow bei der Subtraktion der unteren 4 Bits.
- Zeile 8: Maskiert `result` auf 8 Bit, um das Verhalten eines 8-Bit-Registers sicherzustellen.
- Zeile 9: Setzt das Zero-Flag (Z), falls das 8-Bit-`result` Null ist.
- Zeile 11: Schreibt das 8-Bit-`result` in das Zielregister: Register `f` (wenn `dest=1`) oder W-Register (wenn `dest=0`).

4.4.2 call

Der Befehl `call k` (Call Subroutine) ruft ein Unterprogramm auf, das sich an der Adresse `k` befindet. Bevor der Sprung zur Adresse `k` ausgeführt wird, wird die Adresse des nächsten Befehls (der aktuelle Programmzähler + 1) auf dem Stack gespeichert. Dies ermöglicht die Rückkehr aus dem Unterprogramm mit einem `return`, `retlw` oder `retfie` Befehl. Die Zieladresse `k` ist eine 11-Bit-Adresse. Die oberen Bits des 13-Bit-Programmzählers werden aus dem PCLATH-Register geladen.

```
1 case 'call':
2     self.stack.push(self.dMemory.getPCounter())
3     pclath = int(("".join([str(x) for x in self.dMemory.memory
          [1][0x0A]])[3:5]),2) << 3
4     val = bin(inst[1])[2:]
5     pc = (pclath << 8) + int(val,2)
6     self.dMemory.setPCounter(pc)
7     self.dMemory.setPCL(self.dMemory.getPCounter() & 0xFF)
8     self.addClockCycle()
```

Algorithmus 4.2: Ausschnitt der CALL-Befehlsimplementierung im Central Processing Unit (Zentrale Verarbeitungseinheit) Modul

Erläuterung des Codes: Die Methode simuliert die Ausführung des `call`-Befehls (siehe Listing 4.2):

- Zeile 2: Die aktuelle Adresse des Programmzählers (`self.dMemory.getPCounter()`), welche auf den Befehl nach dem `call` zeigt, wird auf den Stack gelegt. Dies ist die Rücksprungadresse.
- Zeile 3: Die Bits 4 und 3 des PCLATH-Registers (Register an Adresse 0x0A, hier in Bank 1 angenommen) werden ausgelesen, zu einem Integer konvertiert und um 3 Bits nach links verschoben. Diese bilden einen Teil der oberen Bits der Zieladresse. Die komplexe String-Konstruktion dient dazu, die Bitwerte aus der Speicherrepräsentation zu extrahieren.
- Zeile 4: Der 11-Bit-Operandenwert `k` des `call`-Befehls (gespeichert in `inst[1]`) wird in seine binäre Stringrepräsentation ohne das Präfix `0b` umgewandelt.
- Zeile 5: Die vollständige 13-Bit-Zieladresse (`pc`) wird zusammengesetzt. Der in Zeile 3 berechnete Wert aus PCLATH wird um weitere 8 Bits nach links verschoben (insgesamt also um 11 Bits) und mit dem 11-Bit-Wert `val` (konvertiert zu Integer) addiert (bzw. OR-verknüpft, da `val` die unteren 11 Bits darstellt).
- Zeile 6: Der Programmzähler wird auf die neu berechnete Zieladresse `pc` gesetzt.
- Zeile 7: Das PCL-Register (Program Counter Low Byte, typischerweise Teil des Datenspeichers und Spiegel des PC<7:0>) wird mit den unteren 8 Bits des neuen Programmzählers aktualisiert.
- Zeile 8: Die Anzahl der Taktzyklen wird erhöht, da ein `call`-Befehl üblicherweise zwei Taktzyklen benötigt.

Die `decode`-Methode ist ein zentraler Bestandteil des Simulators und dafür zuständig, einen 14-Bit Maschinenbefehl (Opcode), der als Hexadezimalwert `cmd` übergeben wird, zu analysieren und in eine für die CPU verständliche Struktur zu übersetzen. Diese Struktur besteht typischerweise aus dem Namen des Befehls (z.B. "bcf"), dem ersten Operanden

(oft eine Registeradresse oder ein Literalwert) und dem zweiten Operanden (oft ein Bit-Index oder das Ziel-Bit).

```
1  # Opcode-Format: xxxx xxbb bfff ffff (b=Bit, f=Register)
2  # Beispiel: bcf f,b -> Opcode 01 00bb bfff ffff
3  # self.mask2 = 0x3C00 (11 1100 0000 0000) -> Isoliert die Bits 13-10
4  # self.fmask = 0x007F (00 0000 0111 1111) -> Isoliert die unteren 7
       Bits (Register f)
5
6      # cmd ist der 14-Bit Opcode als Integer
7      masked2 = cmd & self.mask2 # Maskiert den Opcode, um die
           Befehlsgruppe zu identifizieren
8      match masked2:
9          #bcf (Bit Clear f) -> Opcode-Gruppe 0100 xxxx xxxx xxxx
10         case 0x1000: # 0x1000 entspricht 0100 0000 0000 0000 (maskiert)
11             return ("bcf", cmd & self.fmask, self.getBit(cmd))
12         #bsf (Bit Set f) -> Opcode-Gruppe 0101 xxxx xxxx xxxx
13         case 0x1400: # 0x1400 entspricht 0101 0000 0000 0000 (maskiert)
14             return ("bsf", cmd & self.fmask, self.getBit(cmd))
15         #btfsc (Bit Test f, Skip if Clear) -> Opcode-Gruppe 0110 xxxx
               xxxx xxxx
16         case 0x1800: # 0x1800 entspricht 0110 0000 0000 0000 (maskiert)
17             return ("btfsc", cmd & self.fmask, self.getBit(cmd))
18         #btfss (Bit Test f, Skip if Set) -> Opcode-Gruppe 0111 xxxx
               xxxx xxxx
19         case 0x1C00: # 0x1C00 entspricht 0111 0000 0000 0000 (maskiert)
20             return ("btfss", cmd & self.fmask, self.getBit(cmd))
21         #movlw (Move Literal to W) -> Opcode-Gruppe 1100 xxxx xxxx xxxx
22         case 0x3000: # 0x3000 entspricht 1100 0000 0000 0000 (maskiert)
23             return ("movlw", cmd & self.mask8, None) # self.mask8 = 0
                   x00FF
24         #retlw (Return with Literal in W) -> Opcode-Gruppe 1101 xxxx
               xxxx xxxx
25         case 0x3400: # 0x3400 entspricht 1101 0000 0000 0000 (maskiert)
26             return ("retlw", cmd & self.mask8, None) # self.mask8 = 0
                   x00FF
27         case _:
28             pass
```

Algorithmus 4.3: Ausschnitt der Decode-Methode im Decode-Modul

Erläuterung des Codes: Das Prinzip der `decode`-Methode (siehe Listing 4.3) beruht darauf, den übergebenen Maschinenbefehl `cmd` schrittweise zu analysieren, um die spezifische Operation und deren Operanden zu identifizieren.

- Zeile 5: Der Opcode `cmd` wird mit einer Bitmaske (`self.mask2`, hier z.B. `0x3C00` bzw. `11110000000000`) bitweise UND-verknüpft. Diese Maske ist so gewählt, dass sie einen bestimmten Teil des Opcodes isoliert, der für eine Gruppe von Befehlen charakteristisch ist. Das Ergebnis `masked2` repräsentiert diese Befehlsgruppe.

- Zeile 6-23: Eine `match-case`-Struktur vergleicht `masked2` mit vordefinierten Werten, die den einzelnen Befehlsgruppen entsprechen.

- Zeile 8 (Beispiel `bcf`): Trifft der `case 0x1000` zu (was bedeutet, dass die oberen relevanten Bits des Opcodes dem Muster für `bcf` entsprechen), wird ein Tupel zurückgegeben.
  - Das erste Element ist der Name des Befehls als String: `"bcf"`.
  - Das zweite Element ist der erste Operand. Für `bcf f,b` ist dies die Registeradresse `f`. Sie wird extrahiert, indem `cmd` mit einer weiteren Maske (`self.fmask`, z.B. `0x007F`) verknüpft wird, die nur die Bits für die Registeradresse isoliert.
  - Das dritte Element ist der zweite Operand, hier der Bitindex `b`. Dieser wird durch die Methode `self.getBit(cmd)` extrahiert, welche die entsprechenden Bits aus `cmd` isoliert und als Integer zurückgibt.
- Zeilen 11-20: Analog werden andere Befehle wie `bsf`, `btfsc`, `btfss`, `movlw` und `retlw` dekodiert. Bei `movlw k` beispielsweise wird das 8-Bit-Literal `k` mit `cmd & self.mask8` extrahiert, und der dritte Operand ist `None`, da dieser Befehl nur einen Operanden hat.
- Zeile 22: Wenn keine der vorherigen Bedingungen zutrifft, wird `pass` ausgeführt, was bedeutet, dass der Code zu weiteren `match-case`-Blöcken oder einer Fehlerbehandlung übergehen würde (nicht im Ausschnitt gezeigt).

Diese Methode ist notwendig, damit der Simulator die rohen, numerischen Befehlscodes, die aus der LST-Datei oder dem Programmspeicher gelesen werden, in eine Form umwandeln kann, mit der die CPU Logik arbeiten kann, um die entsprechenden Aktionen (wie Registermanipulationen, Speicherzugriffe oder Sprünge) auszuführen.

4.4.3 readRegister

Die Methode `readRegister` ist eine Funktion des Speichermoduls, die den Inhalt eines bestimmten Registers im Datenspeicher des simulierten Mikrocontrollers liest. Diese Methode ist entscheidend für die Interaktion zwischen der CPU und dem Speicher, da sie es der CPU ermöglicht, auf die Werte in den Registern zuzugreifen, die für die Ausführung von Befehlen benötigt werden.

```
1  def readRegister(self, register: int, bank=None) -> int:
2      if register == 'w':
3          return self.WREG
4      elif register in range(0x00, 0x80):
5          if register == 0:
6              register = self.readRegister(4)
7          bank = self.getActiveBank() if bank == None else bank
8          return int(("".join([str(x) for x in self.memory[bank][
               register]])),2)
9      else:
10         raise ValueError("Invalid register address")
```

Algorithmus 4.4: Ausschnitt der readRegister-Methode im Speichermodul

Erläuterung des Codes: Die Methode `readRegister` (siehe Listing 4.4) dient zum Auslesen von Werten aus den Registern des simulierten PIC-Mikrocontrollers.

- Zeile 2-3: Wenn als `register` der String `'w'` übergeben wird, gibt die Methode direkt den Inhalt des W-Registers (`self.WREG`) zurück.
- Zeile 4: Prüft, ob die angeforderte Registeradresse im gültigen Bereich einer Speicherbank liegt (hier `0x00` bis `0x7F`).
- Zeile 5-7: Behandelt den Sonderfall der indirekten Adressierung. Wenn Register 0 (INDF - Indirect File) gelesen werden soll, wird stattdessen der Inhalt des FSR-Registers (File Select Register, Adresse 0x04) als die eigentliche Zieladresse interpretiert. Die Methode ruft sich also rekursiv mit der Adresse des FSR auf, um dessen Inhalt zu erhalten, welcher dann als neue `register`-Adresse verwendet wird.
- Zeile 9: Bestimmt die zu verwendende Speicherbank. Wenn der optionale Parameter `bank` nicht explizit übergeben wurde (`None`), wird die aktuell aktive Bank über `self.getActiveBank()` ermittelt. Andernfalls wird die übergebene `bank` verwendet.
- Zeile 13: Greift auf den Speicher zu. `self.memory[current_bank][register]` adressiert das spezifische Register in der gewählten Bank. Wichtig ist hierbei, dass jedes Register als ein Array von 8 Elementen (Bits, z.B. `[0,1,0,1,0,1,0,1]`) gespeichert ist. Dieser Array wird zuerst in einen String aus Nullen und Einsen umgewandelt (z.B. `"01010101"`) und dann mittels `int(..., 2)` als Binärzahl interpretiert und in einen Integer-Wert konvertiert.
- Zeile 16-17: Wenn die angegebene Registeradresse ungültig ist (außerhalb des erwarteten Bereichs und nicht 'w'), wird ein `ValueError` ausgelöst.

Diese Funktion ist fundamental, da viele PIC-Befehle Registerinhalte lesen müssen, um Operationen auszuführen oder Entscheidungen zu treffen. Die korrekte Bankauswahl ist dabei entscheidend für die Korrektheit der Simulation.

Kapitel 5

Fazit

Nach Abschluss des PIC Simulator-Projekts können wir mit Stolz festhalten, dass alle ursprünglich definierten Anforderungen erfolgreich umgesetzt wurden.

Der entwickelte Simulator bietet eine umfassende und benutzerfreundliche Lösung zur Simulation von PIC-Mikrocontroller-Programmen und stellt damit eine wertvolle Alternative zur physischen Hardware dar. Folglich konnten die zu begin des Projekts definierten Ziele entsprechend umgesetzt werden. Der PIC Simulator ermöglicht es Entwicklern, Programme zu testen und zu debuggen, ohne auf physische Hardware angewiesen zu sein.

5.1 Herausforderungen im Entwicklungsprozess

Trotz des erfolgreichen Abschlusses des Projekts gab es während der Entwicklung einige Herausforderungen:

5.1.1 Integration von UI und Backend

Eine der größten Herausforderungen war die nahtlose Integration der Benutzeroberfläche mit dem Backend. Die konstante Kommunikation zwischen diesen beiden Komponenten erwies sich als komplexer als zunächst angenommen:

- **Asynchrone Aktualisierungen**: Die Notwendigkeit, die UI in Echtzeit zu aktualisieren, während das Backend die Simulation durchführt, erforderte eine sorgfältige Implementierung von asynchronen Mechanismen.
- **Datenfluss**: Die effiziente Übertragung von Daten zwischen Backend und Frontend musste optimiert werden, um nicht unnötig Ressourcen zu verbrauchen.
- **Ereignisbehandlung**: Die Verarbeitung von Benutzereingaben und deren korrekte Weiterleitung an die enstprechenden Backend Komponenten erfordete eine detailierte Einarbeitung in das Signal-Slot-System von Qt.

Die Lösung dieser Herausforderungen wurde durch den Einsatz von Python und Qt erreicht, die ein robustes Signal-Slot-System für die Kommunikation zwischen den Komponenten bereitstellen.

5.1.2 Kontinuierliche Weiterentwicklung

Ein weiterer anspruchsvoller Aspekt war die kontinuierliche Weiterentwicklung des Projekts. Während der Implementierung entstanden immer wieder neue Anforderungen und Herausforderungen:

- **Erweiterung der Funktionalität**: Mit fortschreitender Entwicklung wurden zusätzliche Funktionen identifiziert, die integriert werden mussten, ohne die bestehende Codebasis zu beeinträchtigen. Teilweise mussten bestehende Module umgeschrieben werden, um neue Funktionalitäten zu unterstützen.
- **Fehlerbehandlung**: Mit zunehmender Komplexität des Systems musste ein robustes Fehlerbehandlungssystem entwickelt werden, um mit unerwarteten Situationen umgehen zu können.

Der modulare Aufbau des Simulators erwies sich hier als entscheidender Vorteil, da er die schrittweise Erweiterung und Anpassung des Systems ermöglichte, ohne die Gesamtstabilität zu gefährden.

5.2 Lessons Learned

Aus diesem Projekt haben wir wertvolle Erkenntnisse gewonnen:

- **Vorausschauende Planung**: Eine gründliche Anforderungsanalyse und architektonische Planung zu Beginn des Projekts zahlen sich in späteren Phasen aus.
- **Modularität**: Ein modularer Ansatz ermöglicht flexiblere Anpassungen und Erweiterungen während der Entwicklung.
- **Dokumentation**: Eine umfassende Dokumentation erleichtert die Wartung und Weiterentwicklung des Systems.