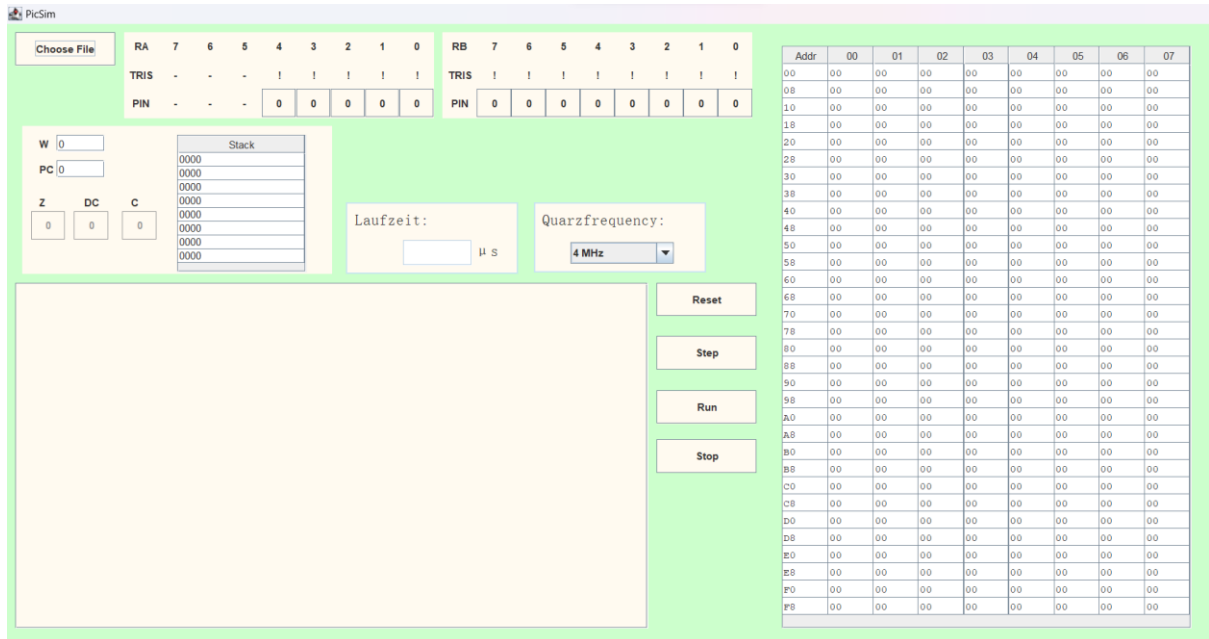


Projektdokumentation

Simulator für den Microcontroller

PIC16F84



Eingereicht von:

Jinfei Cao

Matrikelnummer:

193861

Studiengang:

Angewandte Informatik

Inhaltsverzeichnis

Einleitung -----	1
------------------	---

1. Allgemeines

1.1 Arbeitsweise eines Simulators-----	1
1.2 Vor-und Nachteile -----	1
1.3 Programmoberfläche -----	2

2. Realisierung

2.1 Grundkonzept -----	3
2.2 Gliederung und Programmstruktur -----	4
2.3 Programmiersprache -----	5
2.4 Implementierung ausgewählter Befehle-----	6
2.5 Flags -----	7
2.6 Interrupts -----	8
2.7 TRIS-Register -----	9

3. Zusammenfassung

3.1 Projektergebnis-----10

3.2 Persönliche Reflexion -----11

4. Anhang

Einleitung

- Dieses Projekt befasst sich mit der Entwicklung eines Simulators für den Mikrocontroller PIC16F84.
- Ziel ist es, die grundlegenden Funktionen eines Mikrocontrollers softwareseitig nachzubilden, um dessen Arbeitsweise besser zu verstehen und zu analysieren.

1. Allgemeines

1.1 Arbeitsweise eines Simulators

Ein Simulator ahmt das Verhalten eines realen Mikrocontrollers nach, indem er dessen Funktionen softwareseitig implementiert. Der Mikrocontroller wird nicht physisch ausgeführt, sondern jede Operation (z. B. Befehlsausführung, Speicherzugriff, Interruptverarbeitung) wird Schritt für Schritt in einem Programm simuliert. Dadurch kann man das Verhalten analysieren, debuggen und testen, ohne die reale Hardware zu benötigen.

1.2 Vor- und Nachteile

Vorteile:

- Keine physische Hardware nötig → einfaches Testen
- Fehler können besser gefunden und analysiert werden
- Gute Lernumgebung zur Veranschaulichung von Mikrocontroller-Funktionen
- Wiederholbar und kontrollierbar

Nachteile:

- Zeitlich nicht exakt wie echte Hardware
- Simuliert nur bekannte Bedingungen – reale Probleme (z. B. elektrische Störungen) fehlen
- Kann unvollständig sein, wenn nicht alle Funktionen exakt nachgebildet sind

1.3 Programmoberfläche

In meinem Simulator wurde eine einfache grafische Benutzeroberfläche (GUI) mit mehreren interaktiven Komponenten entwickelt. Diese ermöglicht es dem Benutzer, das Verhalten des Mikrocontrollers bequem zu beobachten und zu steuern.

Folgende Funktionen sind enthalten:

• Button:

- Step – führt eine einzelne Instruktion aus.
- Run – startet die kontinuierliche Programmausführung.

- Reset – setzt den Zustand des Simulators zurück.
- ChoseFile- Zeigt die ursprüngliche Datei mit allen geladenen Befehlen.

- **Taktfrequenz-Einstellung:**

- Der Benutzer kann einstellen, wie schnell das Programm abläuft (Tick-Frequenz).

- **Anzeige der Laufzeit:**

- Zeigt die vergangene Simulationszeit in Mikrosekunden an.

- **Programmmonitor:**

- Eine Tabelle mit allen gelesenen Instruktionszeilen (Adresse, Maschinenbefehl, Assemblerbefehl, Kommentar).
- Die aktuell ausgeführte Zeile wird farblich hervorgehoben.

- **SFR-Fenster:**

- Spezialregister (wie W, STATUS usw.) sollen hier angezeigt werden.

- **RAM-Anzeige:**
 - Übersicht über den aktuellen RAM-Zustand.
-
- **Ziel der GUI:**

Die Oberfläche dient dazu, dem Benutzer nicht nur die internen Zustände des Mikrocontrollers zugänglich zu machen, sondern auch gezielt einzelne Schritte zu kontrollieren, Interrupts zu testen und Registerwerte visuell zu überprüfen.

2. Realisierung

2.1 Grundkonzept

Das Grundkonzept meines Simulators basiert auf einer modularen Struktur. Für jede wichtige Komponente eines Mikrocontrollers gibt es eine eigene Klasse, z. B. eine Klasse für den Speicher (DataMemory), für Ports (Port), für Timer0 (Timer0), für das Hauptprogramm (ProgramMemory) usw.

Der Simulator liest eine Liste von Maschinenbefehlen ein und führt diese Schritt für Schritt aus. Dabei wird geprüft, ob bestimmte Bedingungen erfüllt sind (z. B. ein gesetzter Interrupt oder ein Timerüberlauf). Das Verhalten des Mikrocontrollers wird durch Methodenaufrufe und Objektinteraktionen nachgebildet.

Die Steuerung des Gesamtablaufs übernimmt die Klasse `PicSimulator`, die das zentrale Bindeglied zwischen Decoder, Executor, Speicher und anderen Modulen darstellt. So wird die Arbeitsweise eines Mikrocontrollers wie der PIC16F84 realitätsnah simuliert.

2.2 Gliederung und Programmstruktur

Klasse	Funktion
DataMemory	Zentrale Speicherklasse-verwaltet Register, Ports, Flags, Interrupts usw.
InstructionExcutor	Führt einzelne Instruktionen aus

DecoderWithBitshift	Wandelt Maschinencode in verständliche Instruktionen um
SimulatorInterface	Interface für den Simulator, z.B Methoden wie step(),reset().
PicSimulator	Implementiert das Interface SimulatorInterface, in den grundlegenden Methoden wie step, reset definiert sind. Damit stellt sie die Verbindung zwischen der GUI und der internen Logik des Simulators her.
ProgramMemory	Speichert und verwaltet den Programmcode
Timer0	Simuliert den internen Timer0 mit Zählfunktion.

Port	Repräsentiert PORTA und PORTB mit TRIS, Latch, Value, Update etc.
InstructionFileReader	Liest den gelisteten Maschinen Code aus einer Datei ein
InstructionLine	Repräsentiert eine gelesene Zeile mit Adresse, Instruktion, Machinecode, Kommentar
GUISim	Grafische Oberfläche -z.B. Buttons für Step, reset, Anzeige von Registern

2.3 Programmiersprache, warum?

Java, weil Java es ermöglicht, die verschiedenen Komponenten wie Speicher, Register und Instruktionen in klar getrennte Klassen zu organisieren. So kann man die Logik des Controllers strukturiert umsetzen- etwa wie der W-Register, der Programm Counter oder das Statusregister bei jeder

Instruktion aktualisiert werden.

Außerdem bietet Java mit seiner GUI-Bibliothek eine einfache Möglichkeit, den aktuellen Zustand des Controllers visuell darzustellen. Man kann Registerwerte, Programmschritte, Laufzeit oder gesetzte Breakpoints direkt im Interface verfolgen. Über Buttons wie „Step“, „Run“ oder „Reset“ lässt sich der Ablauf interaktiv steuern-sogar die Taktfrequenz kann angepasst werden. Das macht die gesamte Simulation viel verständlicher und benutzerfreundlicher-besonders zum Lernen oder Debuggen.

2.4 Implementierung ausgewählter Befehle

- MOVF

Beschreibung:

Die MOVF-Anweisung liest den Inhalt des Registeradressplatzes f aus und

speichert diesen Wert abhängig vom Wert des Zielbits d an einem bestimmten Ort:

- Wenn d = 0, wird der Wert in das Arbeitsregister WREG geschrieben.
- Wenn d = 1, wird der Wert zurück in das Register f selbst geschrieben.

Flag-Effekte:

- Z (Zero Flag):

Wird gesetzt (Z = 1), wenn der gelesene Wert gleich 0 ist.

Ansonsten wird das Zero-Flag gelöscht (Z = 0).

```
public void movf(int instruction) {
    int f = instruction & 0x007F; // address
    int d = (instruction >> 7) & 0x01;

    int value = memory.read(f); // value

    memory.setZeroFlag(value == 0); // false

    if (d == 0) {
        memory.setW(value);
    } else {
        memory.write(f, value);
    }

    memory.incrementPC();
    memory.tickTimer0();
}
```

- DECFSZ

Beschreibung:

Die DECFSZ-Anweisung dekrementiert (verringert um 1) den Inhalt des Registers f. Das Verhalten danach hängt vom Ergebnis des Dekrements ab:

- Wenn das Ergebnis 0 ist, wird die nächste Anweisung übersprungen.
- Wenn das Ergebnis ungleich 0 ist, wird mit der nächsten Anweisung normal fortgefahren.

Die Speicherung des Ergebnisses hängt vom Wert von d ab:

- $d = 0 \rightarrow$ Ergebnis wird in das Arbeitsregister WREG geschrieben.
- $d = 1 \rightarrow$ Ergebnis wird in f selbst gespeichert.

```

public void DECFSZ(int instruction) {
    int f = instruction & 0x007F;
    int d = (instruction >> 7) & 0x1;

    int value = memory.read(f);
    int result = (value - 1) & 0xFF;

    if (d == 0) {
        memory.setW(result);
    } else {
        memory.write(f, result);
    }
    if (result == 0) {
        memory.setPC(memory.getPC() + 2);
        memory.tickTimer0();
    } else {
        memory.incrementPC();
    }

    memory.tickTimer0();
}

```

- XORLW

Beschreibung:

Die XORLW-Instruktion führt eine bitweise Exklusiv-Oder-Operation (XOR) zwischen dem aktuellen Inhalt des Arbeitsregisters WREG und einer 8-Bit-Literal-Konstante k durch.

Das Ergebnis wird in das Arbeitsregister WREG geschrieben.

Diese Instruktion eignet sich insbesondere zur Bitmanipulation und zur logischen Verknüpfung mit festen Werten.

Flag-Effekte:

- Z (Zero-Flag):

Wird gesetzt ($Z = 1$), wenn das Ergebnis gleich 0 ist.

Wird gelöscht ($Z = 0$), wenn das Ergebnis ungleich 0 ist.

```
public void xorlw(int instruction) {  
    int k = instruction & 0x00FF;  
    int w = memory.getW();  
  
    int result = w ^ k;  
    memory.setW(result);  
  
    memory.setZeroFlag(result == 0);  
    memory.incrementPC();  
    memory.tickTimer0();  
}
```

- SUBWF

Beschreibung:

Die SUBWF-Instruktion subtrahiert den Inhalt des Arbeitsregisters WREG vom Inhalt des Registers f nach dem Zweierkomplement-Verfahren. Das Ergebnis wird – abhängig vom Zielbit d – entweder in WREG oder in das Register f geschrieben.

- $d = 0 \rightarrow$ Ergebnis wird in WREG gespeichert
- $d = 1 \rightarrow$ Ergebnis wird in f selbst gespeichert

Flag-Effekte:

- C (Carry-Flag):

Wird gesetzt, wenn kein Übertrag notwendig war (d.h. $f \geq WREG$)

- DC (Digit-Carry-Flag):

Wird gesetzt bei Übertrag von Bit 3 zu Bit 4

- Z (Zero-Flag):

Wird gesetzt, wenn das Ergebnis gleich 0 ist

```
public void subwf(int instruction) {
    int f = instruction & 0x007F;
    int d = (instruction >> 7) & 0x1;

    int w = memory.getW();
    int fVal = memory.read(f);
    int result = (fVal - w) & 0xFF;

    if (d == 0) {
        memory.setW(result);
    } else {
        memory.write(f, result);
    }

    memory.setZeroFlag(result == 0);

    memory.setCarryFlag(fVal >= w);

    memory.setDigitatCarryFlag((fVal & 0x0F) >= (w & 0x0F));

    memory.incrementPC();
    memory.tickTimer0();
}
```

- BTFSS

Beschreibung:

Die BTFSS-Instruktion überprüft das Bit b im Register f. Wenn dieses Bit gesetzt ist (Wert = 1), wird die nächste Instruktion übersprungen (Program

Counter wird um 2 erhöht). Andernfalls wird der normale Ablauf fortgesetzt.

```
public void BTFSS(int instruction) {
    int f = instruction & 0x007F;
    int b = (instruction >> 10) & 0x07;

    int value = memory.read(f);
    if (((value >> b) & 1) == 1) {
        memory.setPC(memory.getPC() + 2);
        memory.tickTimer0();
    } else {
        memory.incrementPC();
    }
    memory.tickTimer0();
}
```

- BTFSC

Beschreibung:

Die BTFSC-Instruktion überprüft das Bit b im Register an der Adresse f.

Wenn dieses Bit nicht gesetzt ist (Wert = 0), wird die nächste Instruktion übersprungen (Program Counter wird um 2 erhöht). Andernfalls wird der normale Programmablauf fortgesetzt.

```
public void BTFSC(int instruction) {
    int f = instruction & 0x007F;
    int b = (instruction >> 10) & 0x07;

    int value = memory.read(f);
    if ((value & (1 << b)) == 0) {
        memory.setPC(memory.getPC() + 2);
        memory.tickTimer0();
    } else {
        memory.incrementPC();
    }
    memory.tickTimer0();
}
```

- RRF

Beschreibung:

Die RRF-Instruktion führt eine Rechtsrotation des 8-Bit-Wertes im Register f um ein Bit durch.

Dabei wird das niederwertigste Bit (bit 0) in das Carry-Flag (C) verschoben und das vorherige Carry-Flag wird als höchstwertiges Bit (bit 7) in das Ergebnis übernommen.

Der Zielort des Ergebnisses hängt vom Wert d ab:

- d = 0 → Ergebnis wird in WREG geschrieben
- d = 1 → Ergebnis wird in f selbst geschrieben

Flag-Effekte:

- C (Carry-Flag):

Erhält den Wert von bit 0 aus dem Originalwert von f

```

public void RRF(int instruction) {
    int f = instruction & 0x007F;
    int d = (instruction >> 7) & 0x1;

    int value = memory.read(f);
    int carryIn = memory.getCarryFlag();
    int bit0 = value & 0x01;

    int result = (value >> 1) | (carryIn << 7);

    memory.setCarryFlag(bit0 == 1);

    if (d == 1) {
        memory.setW(result);
    } else {
        memory.write(f, result);
    }
    memory.incrementPC();
    memory.tickTimer0();
}

```

- CALL

Beschreibung:

Die CALL-Instruktion ruft eine Unterprozedur an der Adresse k auf. Dabei wird die Rücksprungadresse (PC + 1) auf den Hardware-Stack gelegt. Anschließend wird der Program Counter (PC) mit der neuen Zieladresse geladen.

Die Zieladresse ist 13 Bit breit und setzt sich aus folgenden Teilen zusammen:

- Bits 0–10: aus dem k
- Bits 11–12: aus PCLATH<4:3>

```

public void call(int instruction) { // with PCLath
    int k = instruction & 0x07FF;

    int pclath = memory.getPclath();
    int pclathHighBits = (pclath >> 3) & 0x03; // only bit 3 and bit 4

    int targetAddress = (pclathHighBits << 11) | k;

    memory.writeInstack(memory.getPC() + 1); // remember next instruction after CALL

    memory.setPC(targetAddress); // (go to) target-address

    memory.tickTimer0();
    memory.tickTimer0();
}

```

2.5 Flags

Im Rahmen dieses Projekts wurden drei zentrale Status-Flags des PIC16F84-Mikrocontrollers implementiert: **Carry-Flag (C)**, **Digit-Carry-Flag (DC)** und **Zero-Flag (Z)**. Diese Flags befinden sich im STATUS-Register und spielen eine entscheidende Rolle bei der Steuerung von Programmabläufen, insbesondere bei Sprungbefehlen und bedingten Ausführungen.

▪ Umsetzung

Zur Manipulation einzelner Bits innerhalb des STATUS-Registers wurden Bitmasken und bitweise Operationen eingesetzt. Jedes Flag wurde durch eine spezifische Maske identifiziert:

- C_Mask = 0x01 für das Carry-Flag (Bit 0)
- DC_Mask = 0x02 für das Digit-Carry-Flag (Bit 1)
- Z_Mask = 0x04 für das Zero-Flag (Bit 2)

Diese Masken ermöglichen es, gezielt einzelne Bits zu setzen oder zu löschen, ohne andere Bits im Register zu verändern. Für jedes Flag wurde eine eigene Methode implementiert, z. B. `setCarryFlag(boolean value)`, die intern entweder `setStatusBit()` oder `clearStatusBit()` mit der entsprechenden Maske aufruft.

▪ **Carry-Flag (C)**

Das Carry-Flag zeigt an, ob bei einer arithmetischen Operation ein Übertrag aus dem höchsten Bit (Bit 7) erfolgt ist. Es wird bei Operationen wie ADDWF oder SUBWF verwendet.

Typische Situation:

- ADDWF: Wenn die Summe > 255 → C = 1
- SUBWF: Wenn kein Übertrag notwendig → C bleibt 1, sonst 0

▪ **Digit-Carry-Flag (DC)**

Das Digit-Carry-Flag zeigt einen Übertrag von Bit 3 auf Bit 4 an und wird bei binär-kodierten Dezimaloperationen verwendet.

Typische Situation:

- ADDWF, wenn untere 4 Bits (Nibble) einen Übertrag erzeugen

▪ **Zero-Flag (Z)**

Das Zero-Flag zeigt an, ob das Ergebnis einer Operation gleich Null ist. Es wird oft zur Steuerung von Schleifen und Bedingungen verwendet, z. B. bei DECFSZ, MOVF, XORLW.

Typische Situation:

- DECFSZ: Wenn Ergebnis = 0 → Überspringe nächste Instruktion
- MOVF: Wenn gelesener Wert = 0 → Z wird gesetzt

2.6 Interrupts

▪ Implementierung der Interrupts

Die Interrupts wurden durch eine Kombination von Bitmasken, Statusabfragen und einer zentralen Interruptbehandlungsfunktion implementiert. Die Hauptlogik befindet sich in der Methode `checkAndHandleInterrupt()`.

Zunächst wurden für das `INTCON`-Register Bitmasken definiert, um gezielt einzelne Bits (wie z.B. `GIE`, `TMR0IF`, `INTF`, `RBIF`) zu überprüfen oder zu verändern.

Die Methode `checkAndHandleInterrupt()` prüft nacheinander drei Arten von Interrupts:

1. Timer0-Interrupt

Wenn `GIE`, `TMR0IE` und `TMR0IF` gesetzt sind, wird der Interrupt ausgelöst.

Es werden folgende Aktionen durchgeführt:

- Der aktuelle Program Counter (PC) wird in den Stack geschrieben
- Der PC wird auf Adresse `0x004` gesetzt

- Das Flag TMR0IF wird zurückgesetzt
- Der globale Interrupt GIE wird deaktiviert

2. RB0-Interrupt (externer Interrupt)

Wenn GIE, INTE und INTF gesetzt sind, erfolgt die gleiche Prozedur wie oben, jedoch mit Rücksetzung des INTF-Flags.

3. PortB-Interrupt (RB4-RB7)

Wenn GIE und RBIE gesetzt sind, wird über eine Schleife ($i = 4$ bis 7) geprüft, ob sich ein Bit geändert hat (`portB.hasChanged()`).

Falls ja:

- Stack speichern
- PC auf 0x004 setzen
- RBIF zurücksetzen
- GIE deaktivieren

Die Methode `writelnStack()`, `setPC()`, sowie spezifische Methoden zum Setzen und Zurücksetzen der Flag-Bits (`clearTMR0OverflowFlag()`, `clearExternalInterruptFlag()`, `clearPortBInterruptFlag()`) wurden separat

definiert, um den Code modular zu halten.

```
// INTCON bit mask
private static final int GIE_MASK = 0b10000000;
private static final int T0IE_MASK = 0b00100000;
private static final int T0IF_MASK = 0b00000100;
private static final int INTE_MASK = 0b00010000;
private static final int INTF_MASK = 0b00000010;
private static final int RBIE_MASK = 0b00001000;
private static final int RBIF_MASK = 0b00000001;

// --- INTCON --- (Addr: 0x0B & 0x8B)
public boolean isGlobalInterruptEnabled() {
    return (read(ADDR_INTCON) & GIE_MASK) != 0;
}

public void disableGlobalInterrupt() {
    int val = read(ADDR_INTCON);
    write(ADDR_INTCON, val & ~GIE_MASK); // clear GIE
}

public boolean isTMR0InterruptEnabled() {
    return (read(ADDR_INTCON) & T0IE_MASK) != 0;
}

public boolean isTMR0Overflowed() {
    return (read(ADDR_INTCON) & T0IF_MASK) != 0;
}

public void setTMR0OverflowFlag() { // set T0IF=1
    int value = read(ADDR_INTCON);
    write(ADDR_INTCON, value | T0IF_MASK);
}
}
```

```

public void checkAndHandleInterrupt() {

    // Timer0 Interrupt
    if (isGlobalInterruptEnabled() && isTMR0InterruptEnabled() && isTMR0Overflowed()) {
        interruptTriggered = true;
        writeInstack(getPC());
        setPC(0x004);
        clearTMR0OverflowFlag();
        disableGlobalInterrupt();
    }

    // RB0 Interrupt
    if (isGlobalInterruptEnabled() && isExternalInterruptEnabled() && isExternalInterruptFlagSet()) {
        interruptTriggered = true;
        writeInstack(getPC());
        setPC(0x004);
        clearExternalInterruptFlag();
        disableGlobalInterrupt();
    }

    // RB4-RB7 Interrupt
    if (isGlobalInterruptEnabled() && isPortBInterruptEnabled()) {
        for (int i = 4; i <= 7; i++) {
            if (portB.isInput(i) && portB.hasChanged(lastPortB, i)) {
                interruptTriggered = true;
                setPortBInterruptFlag();
                writeInstack(getPC());
                setPC(0x004);
                clearPortBInterruptFlag();
                disableGlobalInterrupt();
                break;
            }
        }
    }
}

```

2.7 TRIS-Register

Das TRIS-Register bestimmt, ob ein bestimmter Pin als Eingang (1) oder Ausgang (0) fungiert. In unserer Implementierung wird dieses Register intern durch eine Variable tris dargestellt. Für jeden Pin wird überprüft, ob das entsprechende Bit in tris gesetzt ist. Dies erfolgt zum Beispiel durch die Methode isInput(int bit).

```
public boolean isInput(int bit) {  
    return (tris & (1 << bit)) != 0;  
}
```

In meinem Simulator habe ich zwei verschiedene Ports implementiert:

PortA mit 5 Bit Breite und PortB mit 8 Bit.

Beide Ports werden durch dieselbe Port-Klasse repräsentiert, die in ihrem Konstruktor eine width und einen name erhält:

```
private Port portA = new Port("A", 5);  
private Port portB = new Port("B", 8);
```

▪ Latch-Funktion

Im realen Mikrocontroller kann ein Ausgang nicht einfach durch ein externes Signal überschrieben werden – das ist gefährlich. Deshalb muss unser Simulator diese Trennung exakt nachbilden.

Die Latch-Funktion ist wichtig, um den Zustand von Ausgängen stabil zu halten. Wenn ein Ausgangspin einmal einen Wert bekommen hat (z. B. 1), soll er diesen behalten, bis das Programm ihn bewusst ändert. Eingänge dagegen können sich jederzeit durch externe Signale ändern.

```
public void updateValue(int newValue) {  
    // only update the state of input  
    this.value = (this.value & ~tris) | (newValue & tris);  
    // keep output  
}
```

- (this.value & ~tris) hält den Zustand der Ausgangspins stabil – sie bleiben wie sie sind.
- (newValue & tris) aktualisiert nur die Eingangspins – denn dort dürfen neue externe Werte ankommen.
- Das | kombiniert beides zu einem neuen value.

Die Kombination aus TRIS- und Latch-Mechanismus ermöglicht es unserem Simulator, realistische I/O-Vorgänge nachzubilden – sowohl was Stabilität als auch Datenrichtung angeht.

3. Zusammenfassung

3.1 Projektergebnis

Im Laufe dieses Projekts konnte ich viele grundlegende Funktionen des Mikrocontrollers PIC16F84 in Software erfolgreich nachbilden. Dazu gehören unter anderem:

- das Einlesen und Ausführen von Maschinenbefehlen (z. B. MOVF, CALL, RRF usw.),
- die Aktualisierung der Register (z. B. STATUS, Timer0, PORTA/B),
- die Simulation der I/O-Pins inklusive TRIS- und Latch-Funktion,

- das Setzen und Überprüfen von Status-Flags,
- das Triggern und Verarbeiten von Interrupts (z. B. durch RB0, Timer0 oder PORTB),
- sowie die Möglichkeit, Breakpoints im Programm zu setzen.

Darüber hinaus wurde eine einfache Benutzeroberfläche implementiert, mit der die Programme ausgeführt, gestoppt oder schrittweise verarbeitet werden können. Dabei kann die aktuelle Laufzeit in Mikrosekunden verfolgt und die Ausführungsgeschwindigkeit über die Taktrate angepasst werden.

Trotz dieser Erfolge gibt es noch einige Einschränkungen und offene Punkte. Die Laufzeitanzeige funktioniert z. B. nach einem Dateiwechsel nicht immer zuverlässig. Auch die Hervorhebung (Syntax-Highlighting) von aktiven Programmzeilen funktioniert nicht in jedem Fall korrekt. Einige Funktionen wie das automatische Stoppen der Zeit nach Programmende oder das Laden und Zurücksetzen verschiedener Dateien sind ebenfalls noch nicht vollständig stabil.

Insgesamt bildet mein Simulator jedoch bereits einen großen Teil der Mikrocontroller-Funktionalität nach. Der Aufbau bietet eine gute

Grundlage für künftige Verbesserungen, Tests und Erweiterungen.

3.2 Persönliche Reflexion

Während der Entwicklung des Projekts dachte ich zunächst, dass die schwierigste Aufgabe die Implementierung der Befehlssatz-Logik sein würde. Tatsächlich stellte sich jedoch heraus, dass es viel herausfordernder war, die vorhandene Logik so in das GUI zu integrieren, dass sie korrekt und sichtbar ausgeführt wird.

Diese Erfahrung hat mir gezeigt, dass es nicht reicht, wenn die Logik im Backend korrekt funktioniert – es ist ebenso wichtig, dass sie im Interface klar und zuverlässig dargestellt wird. Für mich persönlich war diese Synchronisation zwischen Logik und Anzeige ein Aspekt, den ich vorher kaum beachtet habe, jetzt aber deutlich besser verstehe.

- **Problem und Lösung**

In diesem Java-Simulator-Projekt war eine der größten

Herausforderungen für mich die Verbindung zwischen dem grafischen Benutzerinterface (GUI) und dem Backend. Da ich mich vorher noch nie systematisch mit der Architektur solcher Verbindungen beschäftigt hatte, wusste ich zunächst nicht, wie ich das lösen sollte. Ich habe überlegt, ob ich alle Logik-Klassen wie DataMemory, InstructionExecutor oder Decoder direkt im GUI verwenden sollte – aber das hätte zu einer unübersichtlichen und schwer wartbaren Struktur geführt.

Um dieses Problem zu lösen, habe ich mich für die Nutzung von Interfaces entschieden. Ich habe ein Interface definiert und dann eine Implementierungsklasse erstellt, die dieses Interface vollständig implementiert. Diese Herangehensweise wurde auch durch die allgemeine Empfehlung im Unterricht unterstützt, bei der Verbindung zwischen GUI und Backend auf saubere Schnittstellen zu achten. In dieser Klasse habe ich alle benötigten Backend-Komponenten zusammengeführt. So konnte ich im GUI lediglich diese Implementierungsklasse einbinden, ohne alle einzelnen Logikmodule direkt im GUI verarbeiten zu müssen. Diese Herangehensweise hat nicht nur den Code modularer und übersichtlicher gemacht, sondern mir auch ein tieferes Verständnis für die Bedeutung und den Nutzen

von Interfaces in der Softwarearchitektur vermittelt. Für mich war das einer der größten Lerneffekte in diesem Projekt.

Beim Schreiben des InstructionExecutor-Moduls habe ich den Aufbau und die Ausführung von Befehlen besser verstanden. Dabei habe ich auch gelernt, wie man in Java mit Bitoperationen wie `>>` und `&` arbeitet, um Opcodes und Operanden zu verarbeiten. Das hat mein Verständnis für Low-Level-Logik und Bitmanipulation in Java deutlich verbessert.

• Verbesserung

Wenn ich diesen Simulator noch einmal von Grund auf entwickeln könnte, würde ich mehr Wert auf eine saubere Trennung der Verantwortlichkeiten legen. Zum Beispiel übernimmt die Decoder-Klasse aktuell nicht nur das Dekodieren, sondern auch die Ausführung der Befehle – das widerspricht dem Prinzip der Einzelverantwortung. In einer neuen Version würde ich das Dekodieren und Ausführen strikt voneinander trennen.

Außerdem würde ich verschiedene Register wie `TIMER0` oder `STATUS`

in eigene Klassen auslagern, damit sie jeweils nur ihren eigenen Zustand und ihre Logik verwalten. Die DataMemory-Klasse sollte lediglich als Speicher dienen – nicht als Steuerlogik. Auch wenn ich momentan alle Register über den gemeinsamen RAM realisiert habe, hat mir dieser Prozess geholfen, die Wechselwirkungen der einzelnen Komponenten besser zu verstehen – zum Beispiel, welche Instruktionen welchen Einfluss auf den Systemstatus haben.

Ein Verbesserungspunkt, den ich nach dem Feedback unseres Tutors erkannt habe, betrifft die Strukturierung meines GUI-Codes. Der Konstruktor in der GUI-Klasse ist aktuell sehr lang und enthält viele Initialisierungen direkt hintereinander, was die Lesbarkeit erschwert.

Eine sinnvolle Verbesserung wäre, die Initialisierungen in mehrere Methoden wie ``initStackPanel()`` oder ``initRegisterView()`` aufzuteilen und diese dann im Konstruktor aufzurufen. Dadurch würde der Code übersichtlicher werden und man müsste nicht so weit nach unten scrollen, um zu sehen, was alles initialisiert wird.

Außerdem wurde auch angemerkt, dass manche Variablennamen – wie

z. B. ``label`` für das Breakpoint-Label – nicht besonders aussagekräftig sind. Diese Art von Benennung kann zu Verwirrung führen, vor allem bei größeren Projekten. In zukünftigen Versionen würde ich darauf achten, sprechendere und eindeutigere Namen zu verwenden.

Insgesamt hat mir dieses Feedback geholfen, meine Programmierweise zu reflektieren, und ich sehe nun deutlicher, wie wichtig saubere Code-Struktur und Lesbarkeit – besonders im GUI-Bereich – sind.

4. Anhang

Anhang 1: Programmlisting-Timer0

Diese Methode simuliert das Ticken des Timer0. Bei Erreichen eines Maximalwertes wird das Overflow-Flag gesetzt und der Zähler zurückgesetzt.

```
public class Timer0 {  
  
    private int value = 0;  
    private int prescalerCount = 0;  
    private final DataMemory memory;  
    private int tickCount = 0;  
  
    public Timer0(DataMemory memory) {  
        this.memory = memory;  
    }  
  
    public int getTickCount() {  
        return tickCount;  
    }  
  
    //  
    public void tick() {  
        tickCount++;  
        if (memory.isInternalClock()) {  
            prescalerCount++;  
            if (prescalerCount > memory.getPrescalerRate()) {  
                prescalerCount = 0;  
                incrementTMR0();  
            }  
        } else {  
            incrementTMR0();  
        }  
    }  
}
```

```

private void incrementTMR0() {
    value = value + 1;
    memory.write(0x01, value);
    if (value == 0x00) {
        memory.setTMR0overflowFlag();
    }
}

public void reset() {
    value = 0;
    prescalerCount = 0;
    memory.write(0x01, 0);
    memory.clearTMR0overflowFlag();
}

```

Anhang 2: Programmlisting-Option Register

Diese drei Methoden ermöglichen die Konfiguration des Timers TMR0 durch das OPTION-Register. Es wird abgefragt:

- ob die interne Clock als Quelle dient (`isInternalClock()`)
- ob der Prescaler dem Timer zugewiesen ist (`isPrescalerAssignedToTMR0()`)
- welches Prescaler-Verhältnis eingestellt ist (`getPrescalerRate()`)

Die Methoden greifen über `read(ADDR_OPTION)` auf den Wert des OPTION-Registers zu.

```

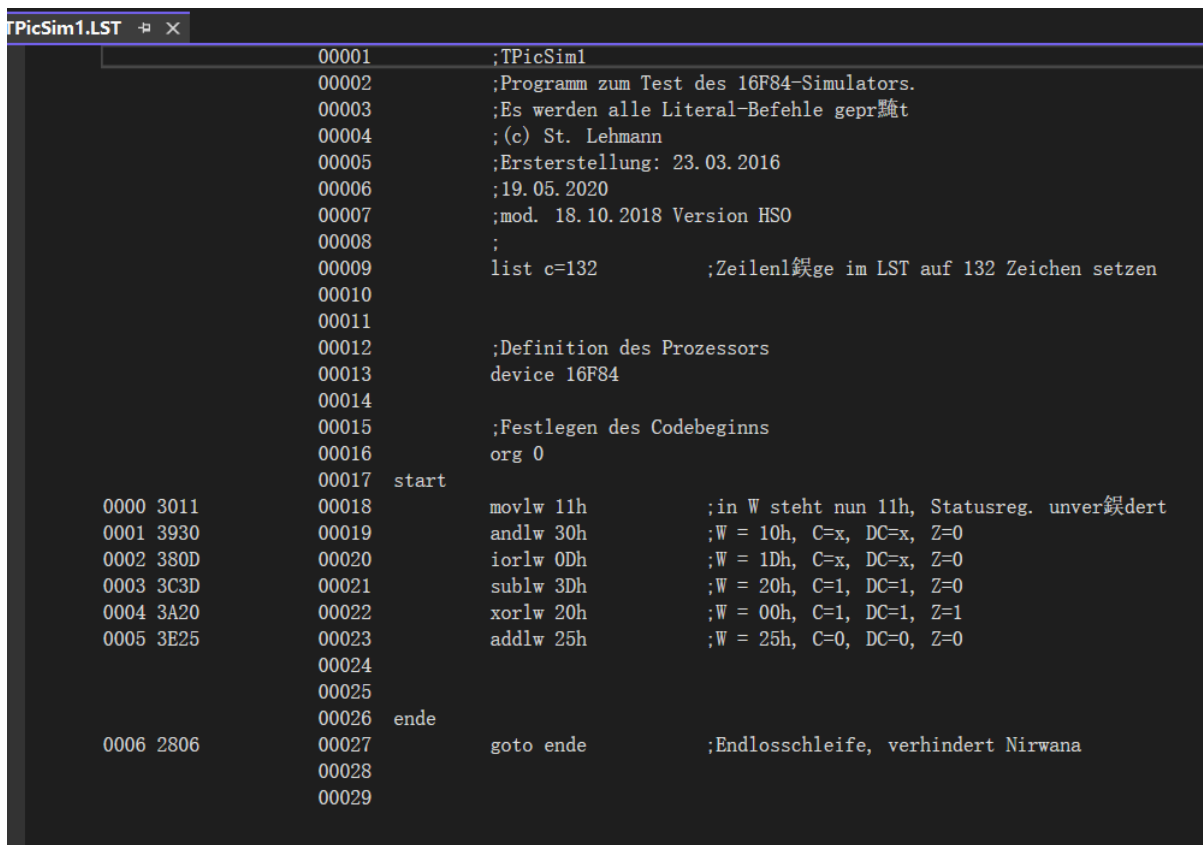
// ---- OPTION ---(Addr: 0x81)
public boolean isInternalClock() { // T0CS = 0
    int value = read(ADDR_OPTION);
    return (value & 0b00100000) == 0;
}

public boolean isPrescalerAssignedToTMR0() { // PSA = 0
    int value = read(ADDR_OPTION);
    return (value & 0b00001000) == 0;
}

public int getPrescalerRate() { // PS2-0
    int value = read(ADDR_OPTION);
    int ps = value & 0b00000111;
    return switch (ps) {
        case 0 -> 2;
        case 1 -> 4;
        case 2 -> 8;
        case 3 -> 16;
        case 4 -> 32;
        case 5 -> 64;
        case 6 -> 128;
        case 7 -> 256;
        default -> 1; // only for short version needed, not possible to get
    };
}

```

Anhang 3: Eingabedatei



```
TPicSim1.LST  ▢ ×
00001      ;TPicSim1
00002      ;Programm zum Test des 16F84-Simulators.
00003      ;Es werden alle Literal-Befehle geprüft
00004      ;(c) St. Lehmann
00005      ;Ersterstellung: 23.03.2016
00006      ;19.05.2020
00007      ;mod. 18.10.2018 Version HS0
00008      ;
00009      list c=132      ;Zeilenlänge im LST auf 132 Zeichen setzen
00010
00011
00012      ;Definition des Prozessors
00013      device 16F84
00014
00015      ;Festlegen des Codebeginns
00016      org 0
00017      start
0000 3011      00018      movlw 11h      ;in W steht nun 11h, Statusreg. unverändert
0001 3930      00019      andlw 30h      ;W = 10h, C=x, DC=x, Z=0
0002 380D      00020      iorlw 0Dh      ;W = 1Dh, C=x, DC=x, Z=0
0003 3C3D      00021      sublw 3Dh      ;W = 20h, C=1, DC=1, Z=0
0004 3A20      00022      xorlw 20h      ;W = 00h, C=1, DC=1, Z=1
0005 3E25      00023      addlw 25h      ;W = 25h, C=0, DC=0, Z=0
00024
00025
00026      ende
0006 2806      00027      goto ende      ;Endlosschleife, verhindert Nirwana
00028
00029
```

Anhang 4: Laufendes GUI mit aktiver Instruktionsausführung

Diese Abbildung zeigt den Zustand des Simulators während der Ausführung des Befehls `andlw 30h`.

Die grafische Benutzeroberfläche stellt die aktuelle Instruktionszeile farblich

hervor,

zeigt den W-Registerinhalt, den PC-Wert sowie den Stack und den Statusregister korrekt an.

Zudem ist ersichtlich, dass der Benutzer die Quarzfrequenz und die Laufzeit anpassen kann

und dass die Steuerung über Step-, Run- und Stop-Buttons funktioniert.

