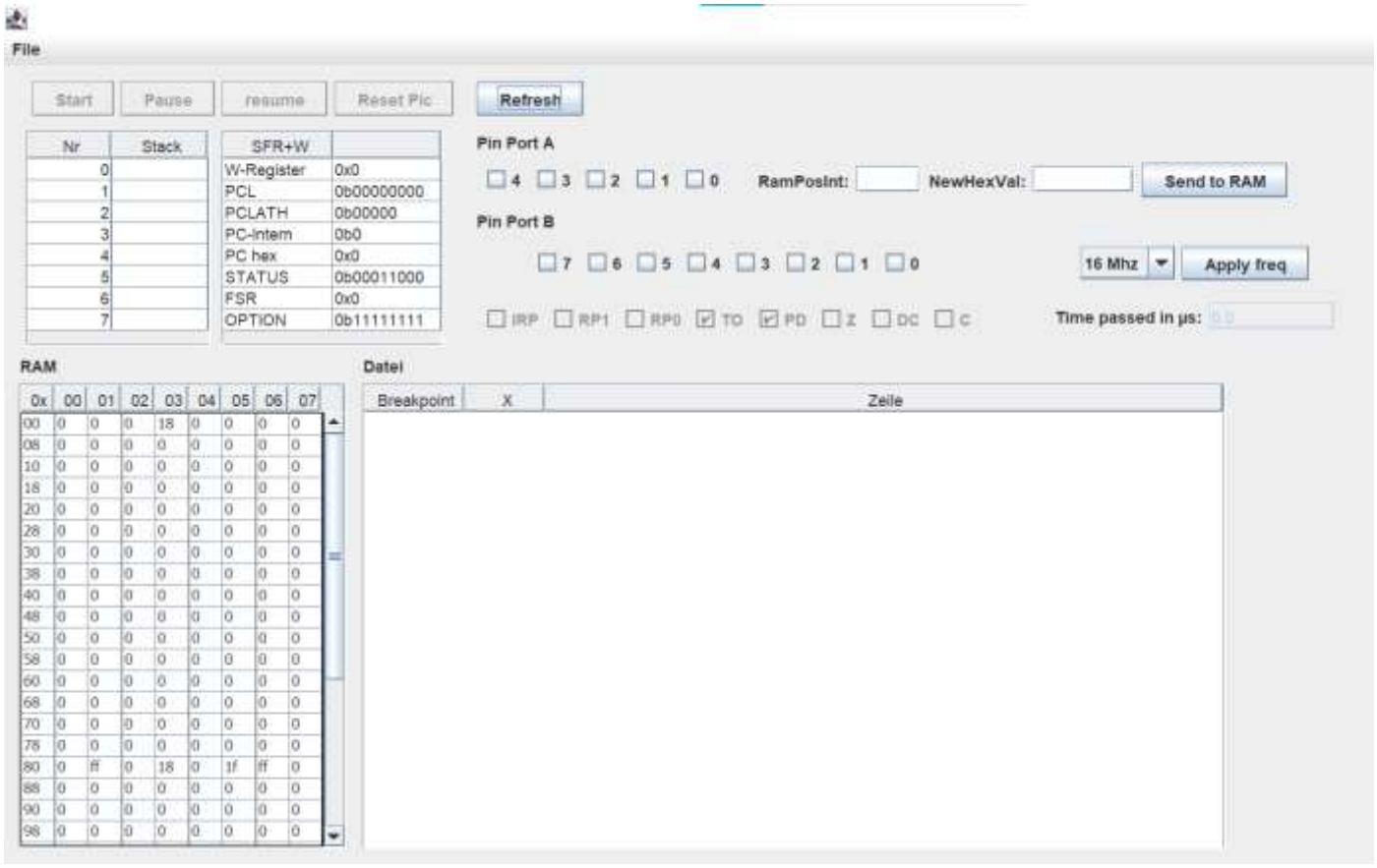
Fazit Projekt Pic16f84 Simulator

Rechnerarchitektur



Übersicht

Einführung

Die Aufgabenstellung des Rechnerarchitekturen-Labors war ein Simulator für den Microcontroller PIC16F84 zu entwickeln. Der Simulator umfasst eine Vielzahl von implementierten Features, die es dem Nutzer ermöglichen, verschiedene Funktionen des Mikrocontrollers zu simulieren.

Im Code intern des Simulators können verschiedene Befehle ausgeführt werden, darunter einfache Literal Befehle wie MOVLW, ADDLW, SUBLW, CALL, GOTO, MOVWF, MOVF, SUBWF, DCFSZ, INCFSZ, RLF, RRF, BSF, BCF, BTFSC und BTFSS. Zusätzlich können diese Befehle auch mit indirekter Adressierung ausgeführt werden. Auch Bytebefehle sind verfügbar. Des Weiteren ist eine Timmer Funktion implementiert, die jedoch keinen Counter-Modus mit RA4-Pin von extern unterstützt. Diese Funktion kann auch mit Prescaler genutzt werden. Es gibt auch ein Interrupt für Timer 0, der PCL mit Berücksichtigung von PCLATH bearbeitet.

Die GUI des Simulators umfasst viele nützliche Funktionen wie einstellbare Breakpoints im Programm und einen visualisierten Laufzeitzähler. Auch können die I/O-Pins per Maus gesteuert werden und es gibt die Möglichkeit, die Quarzfrequenz frei zu wählen und im Zusammenhang mit dem Laufzeitzähler zu verwenden. Der aktuelle oder nächste Befehl im LST-Fenster kann markiert werden und es gibt separate Fenster für LST (Eingangsdatei), SFR und GPR (Speicher des Prozessors). Es besteht auch die Möglichkeit, SFR und GPR zu bearbeiten. Auch der Stack ist visualisiert.

Insgesamt bietet der Simulator eine Vielzahl von Funktionen, die dem Nutzer ein umfassendes Verständnis des PIC16F84 ermöglichen. Durch die Möglichkeit, den eigenen Code zu bearbeiten (sofern im richtigen Format) und jeweils die Abarbeitung zu sehen, können Studenten und Entwickler den Simulator nutzen, um verschiedene Szenarien zu testen und zu verstehen, wie der PIC16F84 funktioniert. Der Simulator ist ein wertvolles Tool und kann dazu beitragen, das Verständnis der Studenten für die Entwicklung von Mikrocontrollern zu verbessern.

Erfassung der Aufgaben/Problematiken

In unserem Projekt zur Entwicklung eines Simulators für den Microcontroller PIC16F84 sind wir anfangs auf einige Herausforderungen gestoßen. Zum einen war uns zu Beginn nicht klar, wie groß das Projekt tatsächlich sein würde und wie wir es am besten gemeinsam verwalten könnten. Wir haben uns für die Git-Erweiterung für Eclipse entschieden auf Rat eines Tutors, jedoch stellte sich später heraus, dass externe Programme wie GitKraken hier deutlich angenehmer gewesen wären. Wir haben in dieser Hinsicht jedoch erst im 3. Semester dazu gelernt.

Wir sind mit den Basics gestartet und haben uns wenig Gedanken über eine ausführliche Planung mit Roadmap gemacht. Wir haben jedoch darauf geachtet, dass die verschiedenen Aspekte des Projekts unabhängig voneinander sind. Allerdings hat uns diese Herangehensweise später dazu gezwungen, Lückenfüller-Methoden zu verwenden, die keine vollständige Implementierung hatten. Das hat beispielsweise dazu geführt, dass die Anzahl der Zyklen für Dinge wie Laufzeit oder Timer außerhalb der Befehle berechnet werden mussten, was suboptimal war. Hier wäre es besser gewesen, die Funktionalität als Rückgabewert der Methoden zu implementieren.

Was den RAM angeht, haben wir uns für eine einfache Lösung entschieden, bei der wir get- und set-Methoden für die einzelnen Speicherstellen implementiert haben. Diese Methode war praktisch, da wir so Funktionen erweitern konnten, ohne anfangs eine vollständige Implementierung vornehmen zu müssen.

Im Verlauf des Projekts haben wir die einzelnen Bereiche mehr miteinander verknüpft. Dies hat jedoch dazu geführt, dass am Ende wenig Zusammenarbeit möglich war, da wir jeweils auf die komplexe

Weiterentwicklung der anderen Bereiche warten mussten. Insgesamt hätten wir von Anfang an eine bessere Planung und eine ausführlichere Roadmap machen sollen, um diese Probleme zu vermeiden.

Aufgabenverteilung

Die Aufgabenverteilung für das Projekt war von Anfang an klar, Moritz und ich haben uns abgesprochen und entschieden, wer welchen Teil des Simulators übernehmen würde. Moritz hat sich hauptsächlich um die Implementierung der einzelnen Befehle und die Basis des Simulators gekümmert, während ich mich hauptsächlich auf die GUI und die Implementierung der Code Abarbeitung konzentriert habe. Es gab eine Art harte Trennung zwischen unseren jeweiligen Aufgabenbereichen, die wir im Laufe des Projekts beibehalten haben.

Obwohl diese klare Aufgabenteilung Vorteile hatte, wie die Möglichkeit, sich voll auf den eigenen Teil zu konzentrieren, hat sie auch zu einigen Problemen geführt. Zum einen gab es kurz vor der ursprünglichen Abgabefrist im zweiten Semester ein Problem, als ich krank wurde und nicht weiterarbeiten konnte. Wir hatten noch nicht dazu gekommen, unsere jeweiligen Bereiche dem anderen zu erklären, so dass Moritz nicht in der Lage war, meine Arbeit fortzusetzen, und das Projekt wurde auf unbestimmte Zeit verschoben.

Ein weiteres Problem war, dass die harte Trennung der Aufgaben zu einem Mangel an Zusammenarbeit zwischen uns führte. Wir mussten jeweils auf die Fertigstellung des anderen warten, bevor wir unsere Arbeit fortsetzen konnten, was zu Verzögerungen führte. Wenn wir von Anfang an eine umfassendere Planung und eine klare Roadmap

erstellt hätten, hätten wir besser zusammenarbeiten und unsere Arbeit besser koordinieren können.

Trotz dieser Herausforderungen hatte die klare Aufgabenteilung auch Vorteile. Wir konnten uns auf unsere jeweiligen Stärken und Interessen konzentrieren und effektiver arbeiten. Insgesamt würde ich sagen, dass die Aufgabenteilung ein zweischneidiges Schwert war - es hatte sowohl Vor- als auch Nachteile, aber wenn man die Arbeitsteilung richtig plant und koordiniert, kann sie zu einer erfolgreichen Projektumsetzung führen.

Umsetzung Lst Parser

Die Umsetzung des Lst Parsers, der dazu dient, Befehlsdateien im Lst-Format zu lesen und zu interpretieren, war eines der Ersten Bausteine, welche entwickelt wurden, da diese späteres testen deutlich vereinfachen sollte und die Basis für die Automatisierung war. Als Teil meines Beitrags zum Projekt habe ich mich mit der Entwicklung dieses Parsers befasst.

Um den Parser zu implementieren, habe ich zuerst die Lst-Dateien analysiert, um ihre Struktur zu verstehen. Anschließend habe ich die Implementierung in Java begonnen. Die Herausforderung bestand darin, die Dateien zeilenweise einzulesen und dabei die Informationen zu extrahieren, die für die weitere Verarbeitung benötigt wurden.

Im Parser-Code habe ich verschiedene Methoden entwickelt, die die extrahierten Informationen in geeigneter Weise speichern und für den Befehlsdecoder zugänglich machen. Ein wichtiger Aspekt war dabei auch die Fehlerbehandlung: Wenn der Parser auf unerwartete

Einträge stößt, muss er in der Lage sein, wäre es ratsam hier direkt einen Fehler zu werfen. Dies wurde bei unserem Projekt hier jedoch übersprungen, um das Testen zu vereinfachen, mit jedoch dem Nachteil das für eine korrekte Ausführung die Lst-Datei genau das richtige Format einhalten muss.

Hier ein Bild der essenziellen Stelle, welche eine beliebige Datei einliest:

```
File fp = new File(filepath);
FileReader fr = new FileReader(fp);
BufferedReader br = new BufferedReader(fr);

ArrayList<String> lines = new ArrayList<>();
ArrayList<Integer> code = new ArrayList<>();
ArrayList<Integer> lineswithcode = new ArrayList<>();
ArrayList<String> codeString = new ArrayList<>();
ArrayList<String> codeStringZeile = new ArrayList<>();

String line;
int counter =0;
while((line = br.readLine()) != null) {
    lines.add(line);
    if (Character.isDigit(line.charAt(0))) {
        code.add(Integer.decode("0x"+line.substring(5, 9)));
        codeString.add(line.substring(5, 9));
        lineswithcode.add(counter);
        codeStringZeile.add(line);
    }

    counter++;
}

ArrayList[] x = new ArrayList[]{lines,code,lineswithcode,codeString,codeStringZeile};
fr.close();
linesCodeLineswithcodeCodestring=x;
return x;
```

1. Jeweils jede Zeile der Datei wird in einer Liste gespeichert
2. Es wird erkannt, ob das erste Zeichen der jeweiligen Zeile eine Zahl ist, wenn ja werden folgende Dinge gespeichert
3. Der Befehl wird zu einem Integer decodiert und abgespeichert
4. Der Befehl wird als String beibehalten und gespeichert
5. Es wird gespeichert in welcher Zeile des Dokuments sich vermeintliche Befehle befinden
6. Es wird jeweils diese Zeile selbst gespeichert

Hier ist zu erkennen, dass deutlich mehr Dinge aus der Lst-Datei heraus gespeichert werden als eigentlich nötig, jedoch war diese Entscheidung vor allem für das Debuggen sehr nützlich und wurden für weiteres behalten, falls zusätzliche Funktionen implementiert werden sollten.

Umsetzung Befehl Decoder

Der Befehl Decoder war eine meiner Hauptaufgaben im Rahmen des Microcontroller PIC16F84 Simulators. Meine Aufgabe war es, die Befehlsanweisungen, die übergeben wurden, in tatsächliche Befehle zu übersetzen, die der PIC16F84 verarbeiten kann.

Dazu habe ich eine Klasse namens "decoder" erstellt, die für die Übersetzung der Befehle zuständig war. Die Klasse war in der Lage, jeden Befehl aus dem Befehlssatz des PIC16F84 zu decodieren und die entsprechende Methode auszuführen.

Um die Befehle zu decodieren, habe ich verschiedene Techniken verwendet, wie z.B. Bitmaskierung, um die verschiedenen Teile des Befehls zu isolieren, und Switch-Statements, um die verschiedenen Befehle zu unterscheiden und die entsprechenden Aktionen auszuführen.

Die Implementierung des Befehl Decoders war eine Herausforderung, da ich mich intensiv mit der Architektur des PIC16F84 und seinen Befehlen auseinandersetzen musste und jeweils einzeln die passende Bitmaskierung für jeden Befehl zu wählen. Aber es war auch eine sehr lohnende Erfahrung, da ich dadurch ein tieferes Verständnis für die Funktionsweise des Microcontrollers erlangt habe.

Insgesamt war der Befehl Decoder eine wichtige Komponente des Simulators und hat dazu beigetragen, dass der PIC16F84 korrekt emuliert wurde, indem die richtigen Befehle gewählt wurden.

Umsetzung Gui

Die Umsetzung der Gui war ebenfalls meine Aufgabe im Rahmen des Projekts. Da ich bereits Erfahrung im Bereich der Gui-Programmierung hatte, war ich zuversichtlich, dass ich diese Aufgabe erfolgreich umsetzen konnte.

Als GUI-Toolkit habe ich mich für Java Swing entschieden, da es eine sehr umfangreiche Bibliothek ist, jedoch vor allem, weil ich bereits leichte Erfahrungen damit hatte und Eclipse direkt ein Add-on bietet welches das ganze um eine Drag and Drop Umgebung erweitern konnte namens „Window Builder“. Ich habe damit die Oberfläche des Simulators entworfen, die die Anzeige von Registerinhalten, des RAM-Speichers, der Programmausführung und der verschiedenen Statusbits ermöglicht. Außerdem habe ich dafür gesorgt, dass der Benutzer über die Gui Einstellungen vornehmen und Befehlsdateien laden kann.

Ein wichtiger Aspekt der Gui war es, sie intuitiv und benutzerfreundlich zu gestalten. Daher habe ich mich auf eine klare und übersichtliche Darstellung der Informationen konzentriert und dafür gesorgt, dass die wichtigsten Funktionen leicht zugänglich sind. Dazu gehören beispielsweise Buttons zur Ausführung des Programms und zum Zurücksetzen des Speichers.

Zusätzlich habe ich die Gui an verschiedenen Stellen um interaktive Elemente erweitert. So können beispielsweise die Werte der Register

und Speicherstellen direkt in der Gui geändert werden, um die Wirkung der Befehle auf die Hardware zu überprüfen. Dies ist jedoch sehr händisch und simpel realisiert durch die Auswahl einer der stellen 0-255 und dem neuen Hex-Wert, was man eventuell etwas schöner hätte lösen können.

Insgesamt war die Umsetzung der Gui eine interessante Herausforderung, da sie unabhängig von dem eigentlichen Backend entwickelt werden konnte.

Umsetzung des "CodeRunner"

Die Implementierung des "CodeRunner" war einer der wichtigsten Bestandteile unseres PIC16F84-Simulators. Mit diesem Modul war es möglich, den eingegebenen Maschinencode abzuarbeiten und den simulierten Zustand des Mikrocontrollers zu verändern. Es gab jedoch einige Herausforderungen bei der Implementierung, auf die ich im Folgenden eingehen möchte.

Eine wichtige Funktion des CodeRunners waren die sogenannten Breakpoints. Diese ermöglichten es dem Benutzer, den Code an einer bestimmten Stelle zu pausieren und den Zustand des Simulators zu überprüfen. Die Umsetzung war relativ einfach, da wir jeweils eine Abfrage an die Gui machen ob bei dem aktuellen Befehl die Checkbox für den Breakpoint gesetzt ist oder nicht und die Ausführung daraufhin pausierten. Dies erlaubte es dem Benutzer, den Code Schritt für Schritt auszuführen und den Zustand des Simulators bei jedem Schritt zu überprüfen.

Ein weiteres wichtiges Feature war die Verarbeitung des PCL (Programm Counter Low Byte). Da der PCL nur 8 Bit groß ist, wird er

oft mit dem PC (Programm Counter) kombiniert, um die Adresse des nächsten Befehls zu bestimmen. Wir mussten sicherstellen, dass der PCL korrekt verarbeitet wurde und dass der PC immer korrekt inkrementiert wurde, wenn nötig, um den nächsten Befehl auszuführen. Hierbei hatten wir einige Schwierigkeiten, da es einige besondere Fälle gab, die wir berücksichtigen mussten.

Ein weiteres wichtiges Element war die Unterstützung des Timers und Interrupts. Der PIC16F84 verfügt über einen Timer und mehreren Interrupts, wovon jedoch bloß der Timer Interrupt in unserem Simulator implementiert wurde jedoch mit zuweisbarem Prescaler. Wir mussten sicherstellen, dass die Timerverarbeitung korrekt implementiert wurde und dass die Interrupts korrekt ausgelöst wurden, wenn sie auftraten.

Schließlich mussten wir die Anzahl der Zyklen für jeden Befehl berechnen, um die Laufzeit des Programms zu bestimmen. Hierbei hatten wir einige Schwierigkeiten, da wir einige Befehle hatten, die mehrere Zyklen benötigten, um ausgeführt zu werden. Wir mussten sicherstellen, dass die Zyklen Berechnung korrekt implementiert wurde, um sicherzustellen, dass die Laufzeit des Programms korrekt simuliert wurde.

Insgesamt war die Umsetzung des CodeRunners eine Herausforderung, aber auch eine sehr lohnende Erfahrung. Wir mussten sicherstellen, dass alle Funktionen des Mikrocontrollers korrekt simuliert wurden, um sicherzustellen, dass unser Simulator korrekt funktionierte. Hier war es besonders schwer jeweils zu erkennen, woher Fehlerquellen kamen, da an diesem Punkt der Entwicklung alle Bereiche aufeinandertrafen.