Dokumentation PIC16F84-Simulator

Einleitung

Das vorliegende Dokument beschreibt die Umsetzung des Projektes der Erstellung einer Simulationssoftware des Microcontrollers des Typs PIC16F84 der Firma Microchip. Das Projekt im Rahmen der Vorlesung Systemnahe Programmierung 2 erfordert das Anwenden bereits erlernter Kenntnisse aus den vorigen Semestern wie beispielsweise binäre Schaltoperationen aus der Digitaltechnik (Bool'sche Algebra, binäre Rechenverfahren, Grundgatter, etc.). Zusätzlich sind Vorkenntnisse der Rechnerarchitekturen notwendig, um die Funktionsweise eines Microcontrollers zu verstehen und im Simulator umsetzen zu können.

Das Projekt wird in einem Zwei-Personen-Team umgesetzt, zur Implementation ist Java mit der Entwicklungsumgebung Eclipse die Programmiersprache der Wahl; Hauptgrund für diese Entscheidung waren die vielen vorgefertigten und einfach zu importierenden Tools und Bibliotheken zur Variablen- und GUI-Bearbeitung. Für die Konstruktion des GUI wird das Eclipse-Plugin Windowbuilder verwendet, das mit einem Baukastensystem die GUI-Erstellung erheblich vereinfacht.

Was ist eine Simulation?

Der Verein Deutscher Ingenieure definiert eine Simulation folgendermaßen:

> *„[Eine Simulation ist die] Nachbildung eines Systems mit seinen dynamischen Prozessen in einem experimentierfähigen Modell, um zu Erkenntnissen zu gelangen, die auf die Wirklichkeit übertragbar sind"*
> *[VDI 3633, (2013)]*

Übersetzt bedeutet das in etwa „In einer Simulation wird die reale Welt so nachgestellt, dass man durch Experimente Wissen über Prozesse der realen Welt erlangen kann." Sie werden meistens dann eingesetzt, wenn Experimente und Messungen in der Realität zu langwierig (z.B. Klimamodelle), zu schnell (Explosionen), zu gefährlich (u.A. Crashtests), unmöglich (Urknallforschung) oder einfach zu teuer (z.B. Stadtplanung) wären.

Pro/Kontra Simulation

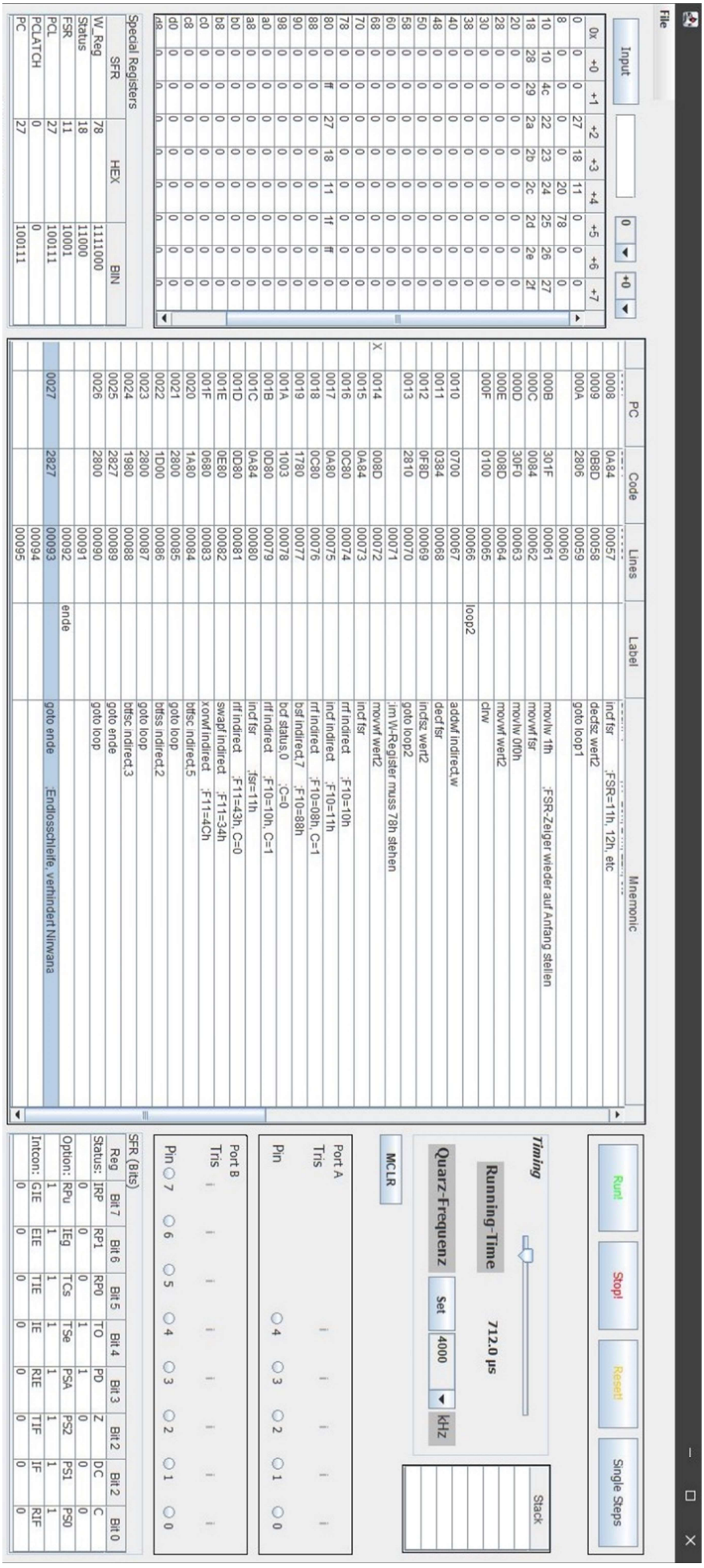
Das wichtigste Argument für eine Simulation wurden bereits angesprochen: sie sind ein sehr gutes Mittel, wenn Messungen in der echten Welt aus verschiedenen Gründen nicht möglich sind. Dafür allerdings unterliegen sie der Problematik. dass sie die Realität nicht exakt abbilden können und daher deutlich anfälliger für Fehler und Ungenauigkeiten sind. Zusätzlich bergen Simulationen die Gefahr, dass anfängliche Fehler wie falsch gewählte Parameter sich im Laufe der Simulation fortsetzen und verstärken, sodass am Ende ein vollkommen verfälschtes Ergebnis steht.

Unsere Implementation

Da bei der Umsetzung des Projektes die Funktionsweise des originalen Controllers möglichst exakt nachgebildet werden soll, ist dieses Projekt keine Emulation, sondern eine vollwertige Simulation, äquivalent zu Flug- oder Fahrschulsimulatoren. Bei einer Emulation steht lediglich das Ergebnis im Vordergrund, die Funktionsweise muss nicht unbedingt mit dem emulierten System übereinstimmen, wie zum Beispiel bei der Emulation von Smartphone-Betriebssystemen auf dem PC oder zu Emulation nicht mehr verfügbarer Spielekonsolen für Retro-Videospiele.

Autoren: Mohammad Mehjazi, David Perk

Interface



Autoren: Mohammad Mehjazi, David Perk

Abbildung 1: GUI des Simulators

Das Interface besteht aus mehreren Teilen zur Darstellung der einzelnen Simulator-Funktionsbereiche. Direkt unterhalb der Menüleiste am oberen Rand des Fensters, mit der sich die Simulation beenden oder eine LST-Datei einlesen lässt, befindet sich die Übersicht über die General Purpose Register (GPR) mit Bearbeitungsfunktion für einzelne Register. Das Textfeld dient zur Eingabe des Wertes, der in das gewählte Register geschrieben werden soll, die beiden Dropdown-Menüs dienen zur Auswahl der Registeradresse über Tabellenzeile und -spalte. Direkt darunter befindet sich eine kleine Tabelle, die die Special Function Register in hexadezimal sowie binär anzeigt; führende Nullen werden dabei ausgeblendet.

Der markanteste Teil des grafischen Nutzerinterface (engl. Graphical User Interface, GUI) ist die große Tabelle in der Mitte, in die der eingelesene Code zeilenweise angezeigt wird. Dabei wird jede Codezeile aufgeteilt in Program Counter, den Befehl (in Hexadezimal-Schreibweise), Codezeile, Label (falls vorhanden) und Mnemonic, also das für Menschen lesbare Kürzel des Befehls. In der ganz linken Spalte der Tabelle lassen sich (sofern es eine Zeile mit gültigem Code ist) durch Anklicken Breakpoints für die jeweilige Zeile setzen.

Im rechten Drittel des Fensters findet sich der Teil, bei dem der User den Simulator steuern kann. Mit je einem Knopf am oberen Rand lässt sich die Simulation Starten/Fortsetzen, Stoppen/Pausieren, auf Start zurücksetzen und zeilenweise durchlaufen. Im Bereich darunter findet sich rechts eine kleine Tabelle mit der Virtualisierung des Stacks, hier allerdings in Dezimalcodierung. Links von dieser Tabelle befindet sich ein mit „Timing“ betitelter Slider zur Anpassung der Verzögerung zwischen der Ausführung von Befehlen; er ist also für die Simulations-„Geschwindigkeit“ verantwortlich. Im selben Feld direkt unter dem Slider ist zum einen der Laufzeitzähler und zum anderen eine Anpassungsmöglichkeit per Dropdown für die Quarz-Frequenz. Auch der Button für den Master Clear (MCLR) befindet sich hier.

Direkt unterhalb der Simulatorsteuerung findet sich schließlich die I/O-Funktion über Port A und Port B mit einer Visualisierung des TRIS-Registers mit einem i für Input und einem o für Output. Wenn Output für das entsprechende Bit ausgewählt ist, wird der Radiobutton ausgegraut und ist nicht mehr klickbar. Wenn am entsprechenden Pin eine 1 ausgegeben wird, wird der Radiobutton als ausgewählt angezeigt.
Ganz unten rechts befindet sich erneut die Special Function Register, hier allerdings auf jedes Bit einzeln aufgeteilt und mit kurzem Erklärungstext für jedes einzelne Bit.

Autoren: Mohammad Mehjazi, David Perk

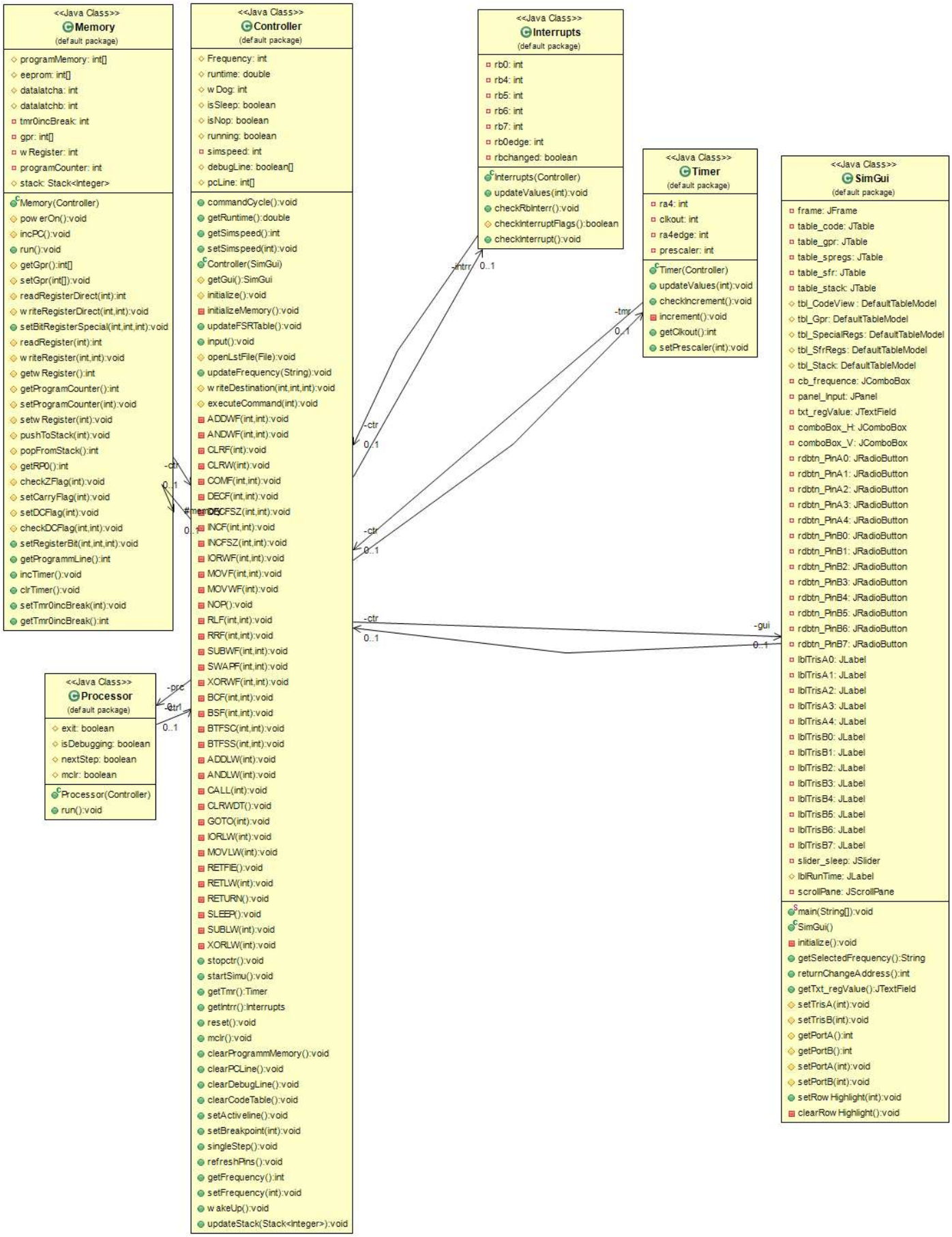


Abbildung 2: Vollständiges Klassendiagramm des Simulators

Die Hauptklasse des Projekts ist SimGui, da sie die main()-Methode enthält; sie ist außerdem für Aufbau und Handling des GUI verantwortlich. Für die Funktionsweise allerdings wurden die verschiedenen Aspekte des PIC auf

mehrere Klassen aufgeteilt. Für Speicherfunktionen ist Memory verantwortlich, um die richtige Ausführung der Simulatorsteuerung (z.B. Befehlsausführung starten oder Breakpoints/Debugmodus) kümmert sich Processor, die Klassen Timer und Interrupts tun exakt das, was ihre Namen andeuten (Interrupt- und Timerhandling).
Die optisch größte und auch die für den Simulator wohl funktionell wichtigste Klasse ist der Controller. Sie ist unter Anderem verantwortlich für das Dekodieren und ausführen der Befehle, das Handling der Codetabelle im GUI und das Berechnen der Runtime mithilfe der Quarzfrequenz.

Die Realisierung des General Purpose Registers erfolgt mithilfe eines Arrays vom Typ Integer in der Memory-Klasse. Die Adresse eines Registers entspricht der Adresse im Array und sein Wert dem Wert an der entsprechenden Stelle im Array. Über eine dedizierte Schreibemethode wird sowohl die Plausibilitätsprüfung des zu schreibenden Werts ($0 \leq x \leq 255$) als auch die Berücksichtigung des RP0-Bits und einer eventuellen indirekten Adressierung gewährleistet. Ebenfalls gibt es eine Funktion, die auf Basis der Registeradresse, die als Parameter übergeben werden muss, den dazugehörigen Registerwert zurückgibt.
Ebenfalls in der Klasse Memory befinden sich die Methoden, die bei Bedarf die Status-Bits 0, 1 und 2 setzen (Carry, Digit-Carry und Zero-Flag). Wenn eines dieser Bits bei der Ausführung eines Befehls berücksichtigt werden muss, werden diesen Methoden das Ergebnis beziehungsweise die beiden Ausgangswerte per Methodenparameter zur Verfügung gestellt und auf Basis dessen werden die Bits entsprechend gesetzt.

Die Befehle des PIC-Mikrocontrollers werden in einem 14-Bit Opcode codiert, der neben dem Befehl an sich auch die Parameter wie Destinationbits und Speicher-Adressen beinhaltet. Für den Decoder innerhalb des Controllers muss dieser Opcode in seine Bestandteile aufgeteilt. Laut Spezifikation des PIC16F84 wird dabei der Befehl mit den höchstwertigsten Bits codiert, diese sind also bei erster Analyse die wichtigsten Bits. Der Opcode wird bitweise aufgeteilt und analysiert:
Anhand der zwei höchstwertigsten Bits (dem Precommand) wird eine Vorsortierung vorgenommen, mithilfe derer die Bedeutung der folgenden zwölf Bits erfasst werden kann, da diese sich abhängig vom Precommand unterscheiden.
Bei Befehlen aus der Gruppe der bitweisen Adressierung, also jenen mit dem Precommand 01, wird der Befehl anhand der Bits 11 und 10 identifiziert und per Fallunterscheidung ausgewählt; die Adresse des Bits ist codiert in den Bits 9-7, im letzten Teil des 14-stelligen Opcodes (also den Bits 6-0) steht die Adresse des Registers, in dem das Bit gesetzt werden soll, beide Adressen werden der jeweiligen Java-Methode als Parameter mitgegeben.
Äquivalent zu dieser Bit-Aufteilung wird auch für die anderen Precommands eine Aufteilung vorgenommen. Falls es, wie beim Bit 7, das sowohl ein Destinationbit als auch eine Unterscheidung zwischen CLRF und CLRW bedeuten kann, eine Bedeutungsüberschneidung bei einzelnen oder mehreren Bits gibt, werden mehrere Variablen mit Werten bestückt. Eine leicht verkürzte Übersicht über die gesamte Dekoderstruktur bietet die Code-Zusammenfassung auf der nächsten Seite.

```
int precommand = (line >> 12) & 0x0003;
if (precommand == 0){
    int command = (line >> 8) & 0x000f;
    int payload = line & 0x00ff;
    int d = payload >> 7 & 0x0001;
    int f = payload & 0b01111111;
    if (f == 0x00 | f == 0x80) {f = this.memory.readRegisterDirect(0x04);}
    int w = this.memory.getwRegister();
    if (command == 7) {this.ADDWF(f, d);}
    else if (command == 5) {this.ANDWF(f, d);}
    else if (command == 1)
    {
        if (d == 1) {this.CLRF(f);}
        else        {this.CLRW(w);}
    }
    else if (command == 9)  {this.COMF(f, d);}
    else if (command == 3)  {this.DECF(f, d);}
    else if (command == 11) {this.DECFSZ(f, d);}
    else if (command == 10) {this.INCF(f, d);}
    else if (command == 15) {this.INCFSZ(f, d);}
    else if (command == 4)  {this.IORWF(f, d);}
    else if (command == 8)  {this.MOVF(f, d);}
    else if (command == 0)
    {
        if (d == 1) {this.MOVWF(f);}
        else if (payload == 0b01100100) {this.CLRWDT();}
        else if (payload == 0b00001001) {this.RETFIE();}
        else if (payload == 0b00001000) {this.RETURN();}
        else if (payload == 0b01100011) {this.SLEEP();}
        else                            {this.NOP();}
    }
    else if (command == 13) {this.RLF(f, d);}
    else if (command == 12) {this.RRF(f, d);}
    else if (command == 2)  {this.SUBWF(f, d);}
    else if (command == 14) {this.SWAPF(f, d);}
    else if (command == 6)  {this.XORWF(f, d);}
} else if (precommand == 1) {
    int command = (line >> 10) & 0x0003;
    int payload = line & 0x03ff;
    int b = (payload >> 7) & 0x0007;
    int f = payload & 0x007f;
    if (f == 0x00 | f == 0x80) {f = this.memory.readRegisterDirect(0x04);}
    if      (command == 0) {this.BCF(f, b);}
    else if (command == 1) {this.BSF(f, b);}
    else if (command == 2) {this.BTFSC(f, b);}
    else if (command == 3) {this.BTFSS(f, b);}
} else if (precommand == 2) {
    int command = (line >> 11) & 0x0001;
    int k = line & 0x07ff;
    if      (command == 0) {this.CALL(k);}
    else if (command == 1) {this.GOTO(k);}
    else                    System.out.println("Neither CALL nor GOTO");
} else if (precommand == 3) {
    int command = (line >> 8) & 0x000f;
    int k = line & 0x00ff;
    if (command == 8) {this.IORLW(k);}
    else if ((command >> 1) == 7) {this.ADDLW(k);}
    else if (command == 9)        {this.ANDLW(k);}
    else if ((command >> 2) == 0) {this.MOVLW(k);}
    else if ((command >> 2) == 1) {this.RETLW(k);}
    else if (command == 10)       {this.XORLW(k);}
    else if ((command >> 1) == 6) {this.SUBLW(k);}
    else                          {this.NOP();}
}
```

Autoren: Mohammad Mehjazi, David Perk

Abbildung 3: Gekürzte Version des Decoders

Konkrete Implementation

Da bei der Implementation der einzelnen Befehle die für den Befehl nötigen Parameter wie zum Beispiel die Registeradresse oder die Bit-Position per Methoden-Parameter direkt aus dem Decoder übergeben werden, war es möglich, die Methoden ohne Eingabeüberprüfung zu schreiben. Ein Beispiel dafür ist die Methode BTFSS, die die Registeradresse f und die Bitposition b übergeben bekommt. Eine in der Speicherklasse integrierte Methode erlaubt das Abrufen des Registerwertes. Durch gezieltes shiften eines einzelnen gesetzten Bits wird eine Maske erzeugt, mithilfe derer bei Verundung das zu prüfende Bit isoliert werden kann. Durch Zurückshiften wird dieses isolierte Bit an die Position 0 gebracht, von wo aus ein simpler numerischer Vergleich (=1?) die Bitüberprüfung ermöglicht. Im Falle eines gesetzten Bits wird hier der Programcounter inkrementiert und ein NOP ausgeführt. In jedem Falle, unabhängig ob eine Anweisung übersprungen wird oder nicht, wird zum Abschluss ein erneuter Programcounter-Inkrement durchgeführt, sodass am Ende entweder der nächste oder der übernächste Befehl ausgeführt wird.

```
private void BTFSS(int f, int b) {
        System.out.println("BTFSS");
        int fValue = this.memory.readRegister(f);
        int bitMask = 0x01;
        bitMask = bitMask << b;
        fValue = fValue & bitMask;
        if ((fValue >> b) == 1) {
                this.memory.incPC();
                this.isNop = true;
        }
        this.memory.incPC();
}
```

Abbildung 4: Code-Ausschnitt Methode BTFSS

Eine weitere wichtige Funktion des PIC sind Interrupts, für die eine eigene Klasse angelegt wurde. In dieser werden zum einen die Interrupt-Flags überprüft (ob ein Interrupt überhaupt ausgegeben wird), zum anderen aber auch die entsprechenden Interrupts an den dazugehörigen Pins ausgegeben. Mit verschiedenen Überprüfungs- und Update-Methoden und -Funktionen wird so ein zuverlässiges Interrupthandling ermöglicht.

Die I/O-Funktionen der Port-Register wurden wie oben bereits beschrieben mit Radiobuttons ermöglicht, das grafische Handling derer wird von der Klasse SimGui übernommen, die Wertsteuerung erfolgt allerdings über die Controller-Klasse. In einer Refresh-Methode liest dieser die TRIS- und PORT-Register sowie die Radiobuttons des GUI aus und durchläuft diese Bit für Bit, bis am Ende der neue Wert feststeht und am GUI ausgegeben werden kann. Der Code für diese Refresh-Methode befindet sich auf der nächsten Seite.

```
// Refresh I/O Pins
public void refreshPins() {
        int trisa = this.memory.readRegisterDirect(0x85);
        int trisb = this.memory.readRegisterDirect(0x86);
        int dataa = this.memory.datalatcha;
        int datab = this.memory.datalatchb;
        this.getGui().setTrisA(trisa);
        this.getGui().setTrisB(trisb);
        int ra = getGui().getPortA();
        int rb = getGui().getPortB();
        for (int i = 0; i < 8; i++) {
                if ((trisa & 0x01) == 1) {
                        this.memory.setBitRegisterSpecial(0x05, i, ra & 0x01);
                } else {
                        this.memory.setBitRegisterSpecial(0x05, i, dataa & 0x01);
                }
                dataa = dataa >> 1;
                trisa = trisa >> 1;
                ra = ra >> 1;
                if ((trisb & 0x01) == 1) {
                        this.memory.setBitRegisterSpecial(0x06, i, rb & 0x01);
                } else {
                        this.memory.setBitRegisterSpecial(0x06, i, datab & 0x01);
                }
                datab = datab >> 1;
                trisb = trisb >> 1;
                rb = rb >> 1;
        }
        getGui().setPortA(this.memory.readRegisterDirect(0x05));
        getGui().setPortB(this.memory.readRegisterDirect(0x06));
}
```

Abbildung 5: Refresh-Methode für TRIS- und PORT-Register

Konklusion

Rückblickend haben sich vor allem die Wahl der Programmiersprache und der Programmierstil negativ auf den Projektverlauf ausgewirkt. Unter anderem die „else if“-Struktur im Dekoder, aber auch viele ähnliche Strukturen an anderen Stellen, hätten im Nachhinein betrachtet einfacher und besser mit switch case Anweisungen abgearbeitet werden können. Solche Fehler lassen sich in erster Linie auf mangelnde Erfahrung mit der Programmiersprache zurückführen.

Ebenfalls ist an vielen Stellen aufgefallen, dass ein wenig mehr Arbeit in die Planung und Konzeptionierung hätte fließen müssen, da viele Methoden und Funktionsweisen an falschen Stellen im Code oder in den falschen Klassen gemanagt wurden. So waren Speicherfunktionen im Controller zu finden und eigentlich beim Prozessor angesiedelte Aufgaben wurden direkt in der GUI-Klasse bearbeitet. Ebenfalls gab es vor allem beim Timer und den Interrupts Rückschläge, die sich mit vorheriger Planung zum größten Teil hätten vermeiden lassen können.

Ebenfalls war eine Problematik das Nicht-Verwenden von Alias-Variablen, sodass eine Adresse stets als Integer-Literal notiert und nicht per Ersatztext direkt ersichtlich war. Eine externe Wartbarkeit ist dadurch sehr erschwert, weil zum Beispiel eine Statusregister-Adressierung nicht mit „STATUS“ und einem hinterlegten Integer-Wert erfolgt, sondern immer direkt mit 0x03. Dies hat einige vermeidbare Verwirrung beim Code-Schreiben und vor allem -Debuggen verursacht.

Autoren: Mohammad Mehjazi, David Perk