

|  |
| --- |
| **Simulation des PIC16F84** |
| Von Marc Lepold und Timo Kern |
|  |
|  |







**Inhaltsverzeichnis**

Seite 0 – Deckblatt

Seite 1 – Inhaltsverzeichnis

**Grundsätzliches über Simulatoren:**

Seite 2 – Definition eines Simulators

Seite 2 – Vor- und Nachteile eines Simulators

**Realisation:**

Seite 3 – Programmoberfläche

Seite 3 – Beschreibung der Grundidee

Seite 3 - Verwendete Programmiersprache

Seite 4 – Verwendete Variablen

Seite 5 – Beschreibung SUBWF

Seite 6 – Beschreibung XORLW

Seite 7 – Beschreibung MOVF

Seite 8 – Beschreibung DECFSZ

Seite 9 – Beschreibung CALL

Seite 10 – Beschreibung RRF

Seite 11 – Beschreibung BTFSC

Seite 12 – Beschreibung BTFSS

Seite 13 – Verwendete Methoden

**Zusammenfassung:**

Seite 14 - Kann die Simulation den Controller gut simulieren?

Seite 15 – Fazit

**Definition eines Simulators:**

Simulatoren werden heutzutage in vielen Bereichen eingesetzt. Sie haben einige Hauptmerkmale die bei allen Simulatoren vorkommen. Zunächst sollte der Begriff „Simulator“ grundlegend erklärt werden. Ein Simulator stellt eine ungefähre Nachahmung eines Ablaufs da. Dabei können diese Abläufe aus der realen Welt stammen oder eine rein hypothetische Annahme sein. Grundsätzlich lässt sich festhalten, dass ein Simulator den Anspruch hat den Ablauf nahezu echt darzustellen oder den Ablauf soweit zu vereinfachen, um ihn überhaupt darstellbar machen zu können.

**Vor- und Nachteile eines Simulators:**

|  |  |
| --- | --- |
| **Vorteile** | **Nachteile** |
| * Günstiger in der Herstellung und im Betrieb im Vergleich zu realen Abläufen | * Eine Simulation ist nie perfekt. Oft können nicht alle Faktoren berücksichtigt bzw. alle Fälle abgedeckt werden. |
| * Änderungen sind einfach umzusetzen | * Möglicherweise kann eine Simulation nie real umgesetzt werden, da sie zu abstrakt ist. |
| * oft sind bestimmte Umstände leichter zu generieren als in der realen Welt |  |
| * Simulationen lassen sich reproduzieren |  |

**Programmoberfläche:**

Unsere Programmstruktur bestehen im Wesentlichen aus zwei Komponenten. Die erste Komponente stellt das Userinterface da. Dabei haben wir auf die Standard-Ui von Herr Lehmann zurückgegriffen. Die zweite Komponente bildet die Eclipse Entwicklungsumgebung. Sie beinhaltet die Logik für die Simulation, während die UI die Benutzerinteraktion mit dem Programm sicherstellt. So soll die UI sowohl den Programmstatus darstellen als auch eine Interaktion und Beeinflussung des Programmes möglich machen. Die Steuerung erfolgt recht intuitiv durch Drücken von Knöpfen und ähnliches. Um das Programm zu starten muss zuerst die Entwicklungsumgebung ausgeführt werden und in der UI ein sogenanntes Testfile ausgewählt werden. Danach läuft das Programm selbständig ab und zeigt entsprechende Parameter auf der UI an. Des Weiteren soll während der Laufzeit eine Interaktion mit dem Backend möglich sein, um den Ablauf aktiv zu beeinflussen.

**Beschreibung der Grundidee:**

Die grundlegende Idee der Projekts-Durchführung liegt darin die Funktionen des PIC16F84 nahezu vollständig in einer Simulation abzubilden. Die Simulation soll aber nur die logische Ebene darstellen, die elektrischen Bereiche sollen vernachlässigt und nicht simuliert werden. Aufgrund der verwendeten Testdateien soll ein Pegelwechsel der I/O-Pins möglich sein. Dafür soll durch Benutzereingabe entweder ein HIGH- oder LOW-Signal gesetzt werden können. Die wesentlichen Eingabemöglichkeiten innerhalb der verwendeten UI sollen die Knöpfe Reset, Einzelschritt, Start und Stop sein. Außerdem sollen die Abfrageintervalle der UI eingestellt und eine Einstell-Möglichkeit des Timing-Verhalten ermöglicht werden. Darüber hinaus soll die aktuell ausgeführte Testdatei dem Anwender angezeigt werden und alle weiteren Informationen wie Inhalt des Fileregisters, Stack-Inhalte, Zustände der einzelnen Statusbits und auch Werte die z.B. im W-Register abgelegt wurden. Um die Funktionalität und Richtigkeit des Simulators bewerten zu können, sollen diverse Testdateien die Logik prüfen.

**Verwendete Programmiersprache:**

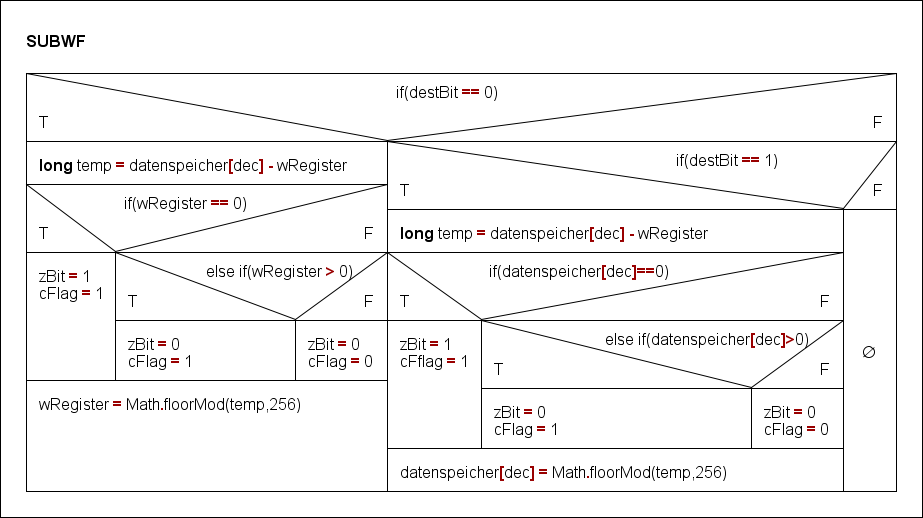
Das Projekt wurde in der Hochsprache Java geschrieben, da diese zum einen vorgeschrieben wurde und zum anderen auch schon mit einer gewissen Praxiserfahrung einher geht.

**Verwendete Variablen:**

|  |  |  |
| --- | --- | --- |
| **Variablenbezeichnung:** | **Datentyp:** | **Funktion:** |
| stack | Long-Array | Ein Long-Array der Größe 8, welches Rückkehradressen speichert, falls Unterprogramme aufgerufen wurden. |
| befehlspeicher | Long-Array | Ein Long-Array der Größe 1024, welches sämtliche Adressen (welche auch Parameter enthalten) einer Testdatei speichert. |
| datenspeicher | Long-Array | Ein Long-Array der Größe 256, welches Werte abspeichern kann, die während einer Berechnung auftreten. |
| stPointer | Integer | Eine Integer-Variable, die den Stackpointer darstellt und entweder inkrementiert oder dekrementiert wird. Sie zeigt so den „Füllstand“ des Stacks an. Und ermöglicht damit FIFO-Prinzip. |
| cFlag | Integer | Eine Integer-Variable, die das Carry-Flag darstellt. Sie wird benötigt um einen Zahlen-überlauf anzuzeigen. Wird 1, wenn ein Überlauf stattgefunden hat, sonst 0. |
| zBit | Integer | Eine Integer-Variable, die das Zeroflag darstellt. Sie wird 1, wenn das Ergebnis eines Vorgangs 0 ist. Sonst wird sie 0. |
| wRegister | Long | Eine Long-Variable, die das W-Register darstellt. Das W-Register ist 8-bit groß und wird benötigt für ALU-Operationen und kann einen Wert abspeichern. |
| teilerFaktor | Integer | Eine Integer-Variable die den aktuellen Teiler-Faktor beinhaltet. |
| counter | Integer | Wird nach jedem Befehl inkrementiert. Für die Berechnung des Timer0 ist sie von Bedeutung. |

**Beschreibung SUBWF:**

**SUBWF**: Der Wert aus der Variable wRegister wird von einem Wert im Datenspeicher subtrahiert. Welcher Wert im Datenspeicher das sein soll bestimmen die Literale. Zusätzlich wird die siebte Ziffer der Adresse genau betrachtet, denn diese stellt das sogenannte Destination-Bit da. Das Destination-Bit (destBit) entscheidet dabei wohin das Ergebnis der obigen Rechenoperation gespeichert wird. Außerdem wird C, DC und Z gesetzt. Am Ende wird das Ergebnis Modulo gerechnet um einen Überlauferkennung zu ermöglichen. Die Variable dec repräsentiert die Literale als Dezimalzahl.



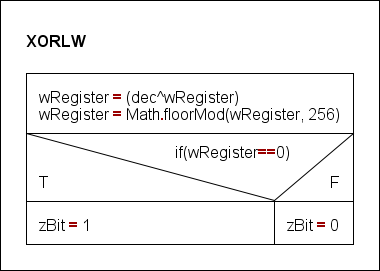
**Beispiel-SUBWF:**

|  |  |
| --- | --- |
| **Vor der Ausführung:** | **Nach der Ausführung:** |
| SUBWF 20Dh  -------------------------------  Wert1(gespeichert im datenspeicher) = 15  wRegister= 15  cFlag =?  zBit = ? | Wert1=0  wRegister = 0  cFlag = 0  zBit = 1 |

**Beschreibung XORLW:**

**XORLW:** Der Wert aus wRegister wird mit dem 8bit-Literal XOR-weise verrechnet. Das Exklusiv-Oder ist genau dann wahr, wenn genau einer der beiden Parameter wahr ist. Außerdem wird Z gesetzt und am Ende wird das Ergebnis Modulo gerechnet um einen Überlauferkennung zu ermöglichen. Der Erlebniswert wird wieder im wRegister abgelegt. Das zBit (Zero-Bit) wird =1, wenn wRegister = 0 ist. Ansonsten setzt es zBit = 0. Die Variable dec repräsentiert die Literale als Dezimalzahl.

|  |  |  |
| --- | --- | --- |
| **a** | **b** | **a ^ b** |
| 0 | 0 | 0 |
| 0 | 1 | 1 |
| 1 | 0 | 1 |
| 1 | 1 | 0 |

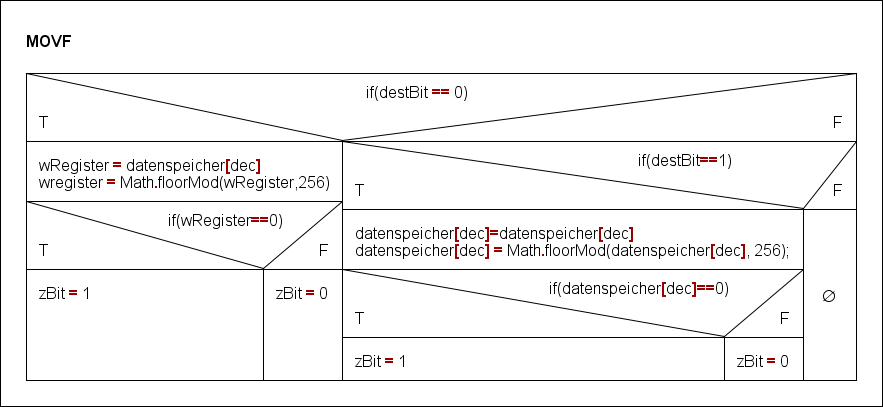


**Beispiel-XORLW:**

|  |  |
| --- | --- |
| **Vor der Ausführung:** | **Nach der Ausführung:** |
| XORLW AFh  ----------------------  wRegister= B5h  zBit = ? | wRegister = 2  zBit = 0 |

**Beschreibung MOVF:**

**MOVF:** Ein Wert aus datenspeicher wird abhängig vom destBit abgespeichert. Bei destBit=0 wird der Wert ins wRegister gespeichert, bei destBit=1 wird der Wert wieder in den datenspeicher gespeichert. Am Ende wird das Ergebnis Modulo gerechnet um einen Überlauferkennung zu ermöglichen. Das zBit wird =1, wenn wRegister/datenspeicher = 0 ist. Ansonsten setzt es zBit = 0. Die Variable dec repräsentiert die Literale als Dezimalzahl.

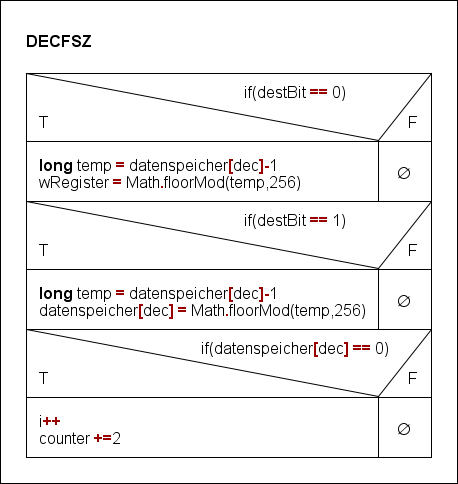


**Beispiel-MOVF:**

|  |  |
| --- | --- |
| **Vor der Ausführung:** | **Nach der Ausführung:** |
| MOVF 800h  ----------------------  dec= 0  wRegister= ? | wRegister = 0  zBit = 1 |

**Beschreibung DECFSZ:**

**DECFSZ:** Ein Wert aus datenspeicher wird um eins dekrementiert und abhängig vom destBit abgespeichert. Bei destBit=0 wird der Wert ins wRegister gespeichert, bei destBit=1 wird der Wert wieder in den datenspeicher gespeichert. Zusätzlich wird das Ergebnis Modulo gerechnet um einen Überlauferkennung zu ermöglichen. Außerdem wird bei einem Ergebnis = 0 der nächste Befehl übersprungen und ein nop wird ausgeführt. Falls das Ergebnis = 1 ist, wird der nächste Befehl planmäßig ausgeführt.

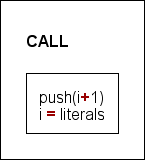


**Beispiel-DECFSZ:**

|  |  |
| --- | --- |
| **Vor der Ausführung:** | **Nach der Ausführung:** |
| DECFSZ B8Ch  ----------------------  destBit = 1  datenspeicher[dec]= 15 | datenspeicher[dec]=14 |

**Beschreibung CALL:**

**CALL:** Die nachfolgende Adresse wird in den Stack geladen und das Programm wird auf die Startadresse eines Unterprogramms eingestellt.

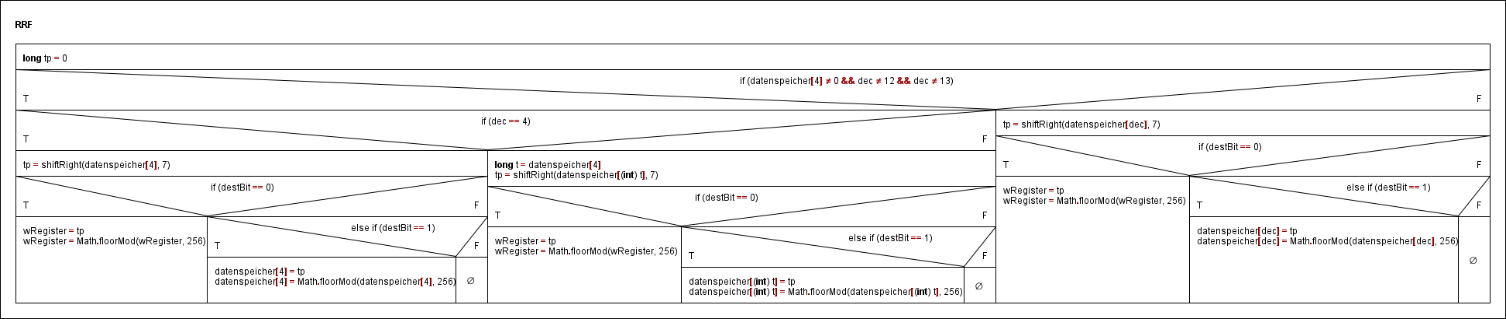


**Beispiel-CALL:**

|  |  |
| --- | --- |
| **Vor der Ausführung:** | **Nach der Ausführung:** |
| CALL 201Fh  ----------------------  stack[stPointer] = -1  i=literals | stack[stpointer] = Rückkehradresse  i=31 |

**Beschreibung RRF:**

**RRF:** Ein Wert aus datenspeicher wird um eins nach rechts geshiftet. Das Element welches rechts rausgedrückt wird, landet in der Variable cFlag. Außerdem wird bei destBit=0 wird der Wert ins wRegister gespeichert, bei destBit=1 wird der Wert wieder in den datenspeicher gespeichert. Zusätzlich wird das Ergebnis Modulo gerechnet um einen Überlauferkennung zu ermöglichen.

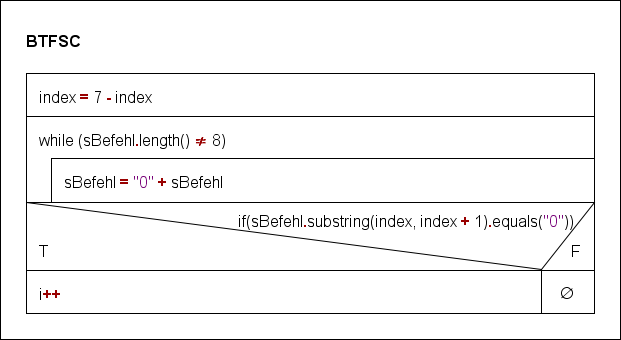


**Beispiel-RRF:**

|  |  |
| --- | --- |
| **Vor der Ausführung:** | **Nach der Ausführung:** |
| RRF DE6h  ----------------------  datenspeicher[dec] = 1110 0110  cFlag = 0 | datenspeicher[dec]= 1110 0110  wRegister = 0111 0011  cFlag = 0 |

**Beschreibung BTFSC:**

**BTFSC:** Falls die b-bits in einer Adresse =1 sind wird der nächste Befehl planmäßig ausgeführt, falls sie = 0 sind wird der nächste Befehl übersprungen und anstatt dessen ein NOP ausgeführt.

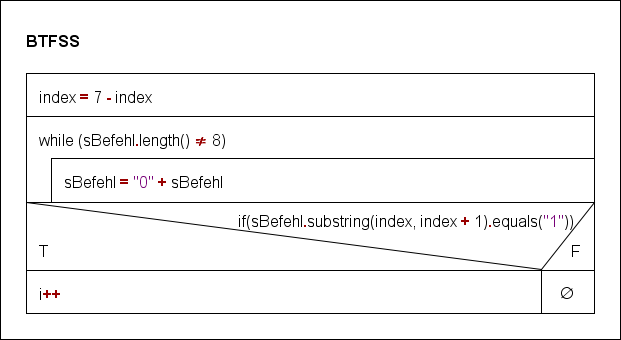


**Beispiel-BTFSC:**

|  |  |
| --- | --- |
| **Vor der Ausführung:** | **Nach der Ausführung:** |
| DECFSZ 1B80  ----------------------  b-Bits= ? | b-Bits=1 -> nächster Befehl wird planmäßig ausgeführt |

**Beschreibung BTFSS:**

**BTFSS:** Ist im Prinzip das genaue Gegenteil von BTFSC. Falls die b-bits in einer Adresse =0 sind wird der nächste Befehl planmäßig ausgeführt, falls sie = 1 sind wird der nächste Befehl übersprungen und anstatt dessen ein NOP ausgeführt.



**Beispiel-BTFSS:**

|  |  |
| --- | --- |
| **Vor der Ausführung:** | **Nach der Ausführung:** |
| DECFSZ BTFSS  ----------------------  b-Bits=? | b-Bits =0 -> nächster Befehl wird planmäßig ausgeführt. |

**Verwendete Methoden:**

|  |  |  |
| --- | --- | --- |
| **Methodenname:** | **Übergabeparameter:** | **Methodenbeschreibung:** |
| getDBit | (long binary, int position) | Extrahiert das Destination-Bit aus der Adresse „binary“ welches an der Stelle „position“ steht. |
| dateiSchreiben | () | Erstellt die Datei „gui\_set.dat“ und befüllt diese mit Werten aller Variablen, damit die GUI diese aufnehmen kann. |
| getLiterals | (long binary, int length) | Extrahiert die letzten legth+1 Stellen aus der Adresse „binary“. Diese Stellen sind die Literale. |
| swapNibble | (long dec) | Vertauscht das erste Halb-Byte mit dem zweiten Halb-Byte der Variable „dec“. |
| negator | (long value, int length) | Negiert den Wert „value“ mit der Länge „length“. Null wird zu Eins. Eins wird zu Null. |
| shiftLeft | (long value, int length) | Verschiebt alle Ziffern der Variable „value“ (mit der Länge „length“) um eins nach links.  Die rausgeschobene Ziffer wird ins C-Flag gespeichert und der bisherige Wert aus dem C-Flag wird rechts eingefügt. |
| shiftRight | (long value, int length) | Verschiebt alle Ziffern der Variable „value“ (mit der Länge „length“) um eins nach rechts.  Die rausgeschobene Ziffer wird ins C-Flag gespeichert und der bisherige Wert aus dem C-Flag wird links eingefügt. |
| push | (long adresse) | Schreibt die Rückkehradresse „adresse“ an die oberste Stelle des Stacks. Falls pointer>7 wird „pointer“ = 0 gesetzt um beim nächsten Push das älteste Element zu überschreiben. Falls nicht wird „pointer“ inkrementiert. |
| pop | () | Löscht Inhalt aus dem Stack an der Stelle „stPointer“ und dekrementiert den pointer um eins. |
| getDCFlag | (long wert1, long wert2) | Verkleinert wert1 und wert2 auf jeweils vier Stellen und addiert disese. Wenn dabei ein Überlauf stattfindet wird DCFlag = 1 gesetzt, andernfalls wird DCFlag = 0 gesetzt. |

**Kann die Simulation den Controller gut simulieren?**

Man kann die grundsätzlichen Funktionen des Controllers nachbilden, allerdings fallen bestimmte Tiefendetails heraus. Je nachdem sind diese Details vernachlässigbar, manchmal wäre aber eine Implementierung sicher interessant. Zum Beispiel, welche Temperaturen der Controller auf Maximallast erreichen würde. Die Simulation reicht also aus um die grundlegenden Prinzipe des Controllers zu verstehen und anschaulich zu machen.

**F****azit:**

Der Gesamteindruck zu unserem Projekt ist relativ unterschiedlich. Einerseits haben wir viel gelernt und vor allem konnten wir bereits Gelerntes anwenden. Andererseits hat uns dieses Projekt vor Herausforderungen gestellt die oft ausweglos schienen und erst nach intensiver Analyse einen Lösungsweg bereithielten. Manchmal waren die Probleme banal ein anderes Mal waren sie so komplex, dass man eine genaue Strategie benötigte um das Problem zu lösen. Allerdings war es jedes Mal ein Erfolgserlebnis, wenn man ein Problem lösen konnte und so seinem Ziel ein Stück nähergekommen ist. Was uns auf jeden Fall interessiert hätte, wäre wie man die Logik außerhalb der IDE hätte betreiben können. Außerdem fehlte uns das Wissen wie man eine eigene UI in Java erstellt, so haben wir uns letztendlich auch für die Standard-UI entschieden. Ein Problem das tatsächlich immer wieder auftrat war, dass wenn man einen neuen Befehl hinzugefügt hat, dass dann alte bereits implementierte Funktionen fehlerhaft wurden. So musste man nach fast jeder Änderung das Programm debuggen um auf Fehler zu prüfen. Dies beanspruchte wahrscheinlich die meiste Zeit, trotzdem sind wir der Meinung das bei der Entwicklung einer Software debuggen einfach unabdingbar ist. Außerdem ist uns klar geworden, dass man bei größeren Projekten nicht einfach los programmieren kann, sondern man benötigt eine genaue Strategie. Gerade bei Codeänderungen muss man sämtliche Funktionen testen. Denn wenn man dies nicht beachten sollte, können später bei Fehlfunktionen die Fehlerquellen nur noch unter großem Aufwand ausfindig gemacht werden oder im schlimmsten Fall könnte der gesamte Code unbrauchbar werden. Ein weiterer Punkt der uns schwer gefallen ist war ganz klar der Anfang, wir hatten keinen Plan wie wir das Projekt angehen sollten, so ging uns auch wertvolle Zeit verloren. Erst nach Gesprächen mit den Tutoren und dem Dozenten wussten wir wo wir anfangen konnten. Auch der Austausch mit Kommilitonen hat uns im Laufe des Projekts immer wieder weitergeholfen. Wenn wir das Projekt nochmal machen müssten, dann würden wir auf jeden Fall dafür sorgen das der Code übersichtlicher wird, am besten mit Auslagern von Code in Methoden oder Klassen. Außerdem würden wir wahrscheinlich innerhalb des Programmes ein anderes Zahlensystem verwenden, da wir in unserem Projekt oft zwischen den einzelnen Zahlensystemen umher rechnen mussten. Um unser Fazit zusammenzufassen würden wir sagen, dass dieses Projekt uns auf jeden Fall herausgefordert hat, aber auch Spaß gemacht hat. Einige Dinge würden wir beim nächsten Mal sicher anders umsetzten. Wir haben einiges gelernt und konnten bereits Erlerntes anwenden. Unser Gesamteindruck würden wir als positiv bewerten.