Dokumentation

des TuxSim Simulators

für den

MicroController PIC16F84

Von Karolin Edigkaufer und Lukas Essig
aus dem Kurs TINF11B3
der Dualen Hochschule Baden-Württemberg
im Mai/Juni 2013

Inhaltsverzeichnis

Was ist ein Simulator?

Ein Simulator bildet ein System aus der Wirklichkeit nach. Dies ist hilfreich um meist dynamische Sachverhalte zu analysieren. In einer Simulation können (im Unterschied zu einem Emulator) Problemfälle langsamer dargestellt und somit nachvollzogen werden, um so zu einer Lösung zu kommen. Diese Lösung kann anschließend auf die Realität übertragen werden.

Vor- und Nachteile einer Simulation

Vorteile bei einer Simulation liegen beispielsweise in den Kosten und der Sicherheit. So trainieren unter anderem Piloten ihre Fähigkeiten an Simulatoren, da im Falle eines Absturzes niemand zu Schaden kommt. Auch bei einem Microcontroller ist eine Simulation sinnvoll. So können Programme auf dem Simulator getestet werden und so umständliche Fehlersuche am Microcontroller vermieden werden.

Nachteile liegen in der Begrenztheit der Simulatoren. Es kann lediglich das getestet werden, was auch implementiert ist. Dies ist natürlich stark von Zeit und Geld abhängig.

Programmiersprache

Unseren Simulator haben wir in der Programmiersprache Java geschrieben, da diese plattformunabhängig ist. Zudem bietet Java einen hohen Grad an Geschwindigkeit um prozeduralen Programmcode auszuführen.

Der Simulator wird als .jar- Datei herausgebracht, was im Gegensatz zu einer .exe den Vorteil hat, dass die .jar auf jedem System auf dem java installiert ist lauffähig ist. Dies ist nicht nur sinnvoll im Bezug auf den Benutzer, der unabhängig von seinem Betriebssystem unseren Simulator nutzen kann, sondern auch notwendig gewesen, da wir auf unterschiedlichen Betriebssystemen (Linux und Windows) entwickeln.

Entwicklungsumgebung Eclipse und GitHub

Die eigens für Java entwickelte und angepasste Entwicklungsumgebung Eclipse haben wir sowohl für die funktionale Programmierung, als auch für die GUI Programmierung genutzt. Dies war über ein Plug – In relativ leicht möglich.

Mittlerweile ist Eclipse durch hinzufügen von Plugins auch im Stande C, C++ oder Perl-Code auszuführen. Vorteil von Eclipse sind die umfangreichen integrierten Funktionen, wie git- Unterstützung, Debugging und Source-Funktionen (Automatische Generierung von Getter- und Setter Funktionen) derer wir uns reichlich bedient haben, sowie die Autovervollständigung beim programmieren.

Als Versionsverwaltung wurde git gewählt, dass im Gegensatz zu SVN oder CVS, ein lokales sowie ein Remote- Repository (hier Github) nutzt. Eine Versionsverwaltung bietet die bequeme Möglichkeit zusammenzuarbeiten und gegebenenfalls zu einer früherern Version zurückzukehren. Dabei wird das Remote- Repository lokal „ausgecheckt“ und erst nach einem *commit* im lokalen Repository kann dieses ins Remote- Repository *gepusht* werden.

Git hat allerdings den Nachteil, dass klar getrennt sein muss, wer welche Bereiche bearbeitet, da es zu Fehlern kommt, wenn mehrere Personen an der gleichen Datei etwas verändert haben.

Vorraussetzungen

Damit der Simulator ausgeführt werden kann muss eine Quelltext- Datei im Format LST vorliegen. Diese Datei wurde schon von einem Assemblierer assembliert und enthält die für den Simulator essentiellen Befehle in hexadezimaler Darstellung.

Zudem wird zur Ausführung eine Java- Laufzeitumgebung benötigt (empfohlene Version Java 7), da der Simulator in Java programmiert wurde.

Konzept

Zu Beginn des Projektes haben wir ein Lastenheft verfasst. In diesem wird beschrieben welche Muss – und Kann- Funktionen implementiert werden. Dabei wurde Schwerpunkt auf eine einfach zu bedienende Oberfläche und die Korrektheit der PIC- Befehle gelegt.

Funktionalitäten wie Schalter, Funktionsgenerator oder 7- Segment- Anzeige werden niedrig priorisiert. Zudem wird abgegrenzt, dass der eingegebene Programmcode korrekt sein muss und der Simulator keine Funktionen eines Assemblers übernimmt.

Oberfläche und Benutzerhandbuch

Die Oberfläche des TuxSim ist, wie in Abbildung 1 zu sehen, in vier Bereiche untergliedert.

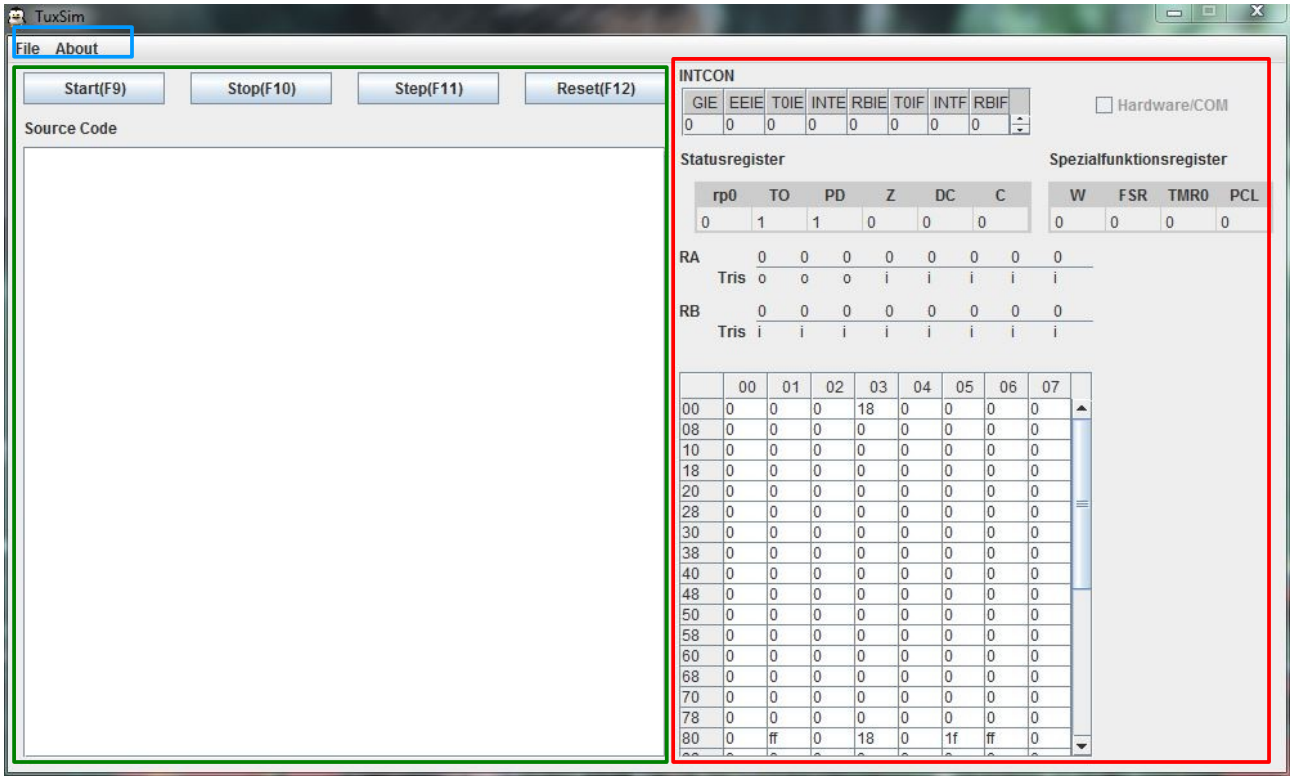


Abbildung 1: Oberfläche des TuxSim

Der blaue Bereich spiegelt das Menü wieder. Dort kann ein neues Programm zur Ausführung geöffnet werden, die *Hilfe (F1)* angezeigt werden oder das *About* – Fenster angezeigt werden.

Im grünen Bereich wird der Quelltext angezeigt und mit den Buttons kann der Programmablauf gesteuert werden.

Die wesentlichen internen Strukturen des PIC- Simulator werden im roten Bereich nachgebildet. Das sind das INTCON-Register (Interrupt-Bits), das Statusregister, das Spezialfunktionsregister (W-Register, FSR, TMR0 und Programm Counter). Darunter werden die Ports mit den Tris – Registern angezeigt. Die Tris – Register zeigen an, welche Datenrichtung (input|output) das jeweilige Port-Bit besitzt. Unter den Ports sind die kompletten Registerinhalte abgebildet.

Alle Werte der Register werden als Hexadezimal- Zahl dargestellt.

Klassendiagramm

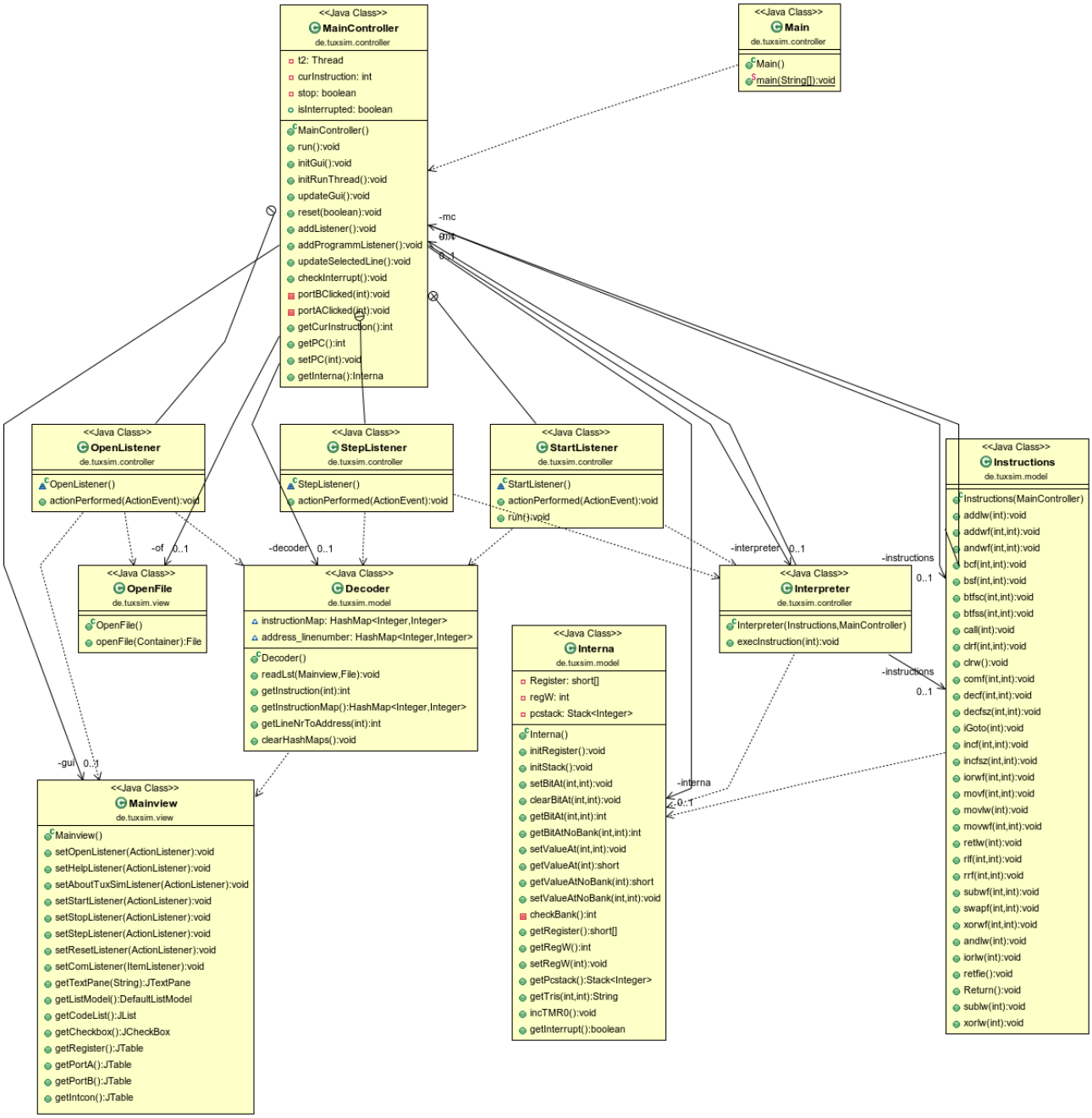


Abbildung 2: Klassendiagramm

Das Klassendiagramm zeigt alle elementaren Klassen und Abhängigkeiten. Die Architektur wurde als Model-View-Controller gewählt. Die View beinhaltet alle Klassen, die mit der Darstellung der Daten für den Anwender zu tun haben. Im Model werden die Daten gespeichert, in diesem Fall die Register, die Ports, der Stack sowie das geparste Programm. Im Controller werden dann die Daten abgerufen und verwertet. Hier ist der *MainController* das Herz des Simulators, da er alle anderen Klassen kennt und die Funktionen dieser aufruft und das Ergebnis an andere Klassen übergibt.

Programmablauf

Der wesentliche Programmablauf gliedert sich in folgende Punkte auf :

1. Anwender wählt LST- Datei zur Ausführung aus
2. LST-Datei wird decodiert, Opcodes werden geladen, PC = 0
3. Start (Ende = Stop) / Step (Ende = PC+1)
    1. Befehl von PC-Adresse holen
    2. Anhand Opcode entscheiden was für Befehl (Byteops, Bitops or Literal/Control-Ops)
    3. Befehl ausführen
    4. PC erhöhen

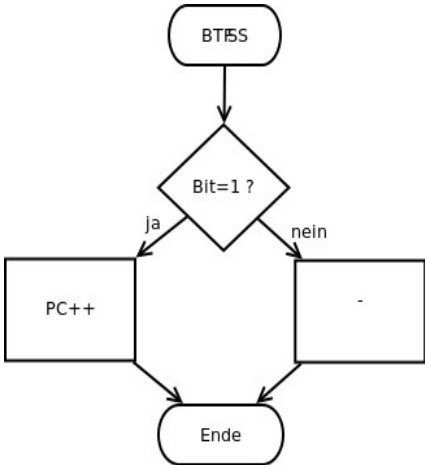
Der Opcode wird durch reguläre Ausdrücke entschlüsselt. Wenn dieser mit 00 beginnt handelt es sich um ByteOps (oder returns), bei 01 um BitOps und ansonsten um Literal/Control-Ops. Falls der Opcode kürzer als 14bit ist, wird dieser mit führenden Nullen aufgefüllt. Zudem wird vor jeder Ausführung eines Befehls überprüft, ob ein Interrupt aufgetreten ist.

Befehle

Im Folgenden wird die Funktionsweise anhand einiger essentieller Befehle erläutert.

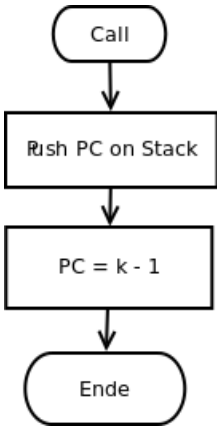
BTFSx

Die bitorientierten Befehle (bcf,bsf,btfsc,btfss) betreffen immer nur ein einzelnes Bit im Register. Der Opcode dazu wird in der *Interpreter*-Klasse entschlüsselt. Dazu wird dieser mit einem Literal verundet und gegebenfalls geshiftet. Die daraus entstandenen Variablen werden den bitorientierten Befehlen mitgegeben.
Beispielshaft der Programmablaufplan für BTFSS:



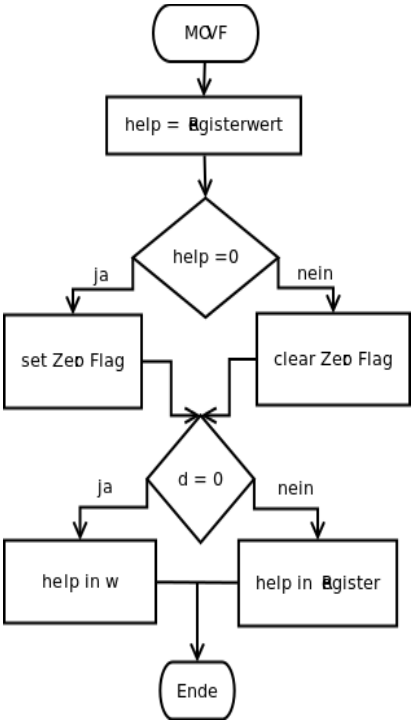
CALL

Bei Call wird der aktuelle PC auf den Stack gepusht und der neue (übergebenes Literal k) PC wird zum aktuellen gesetzt. Dabei wird der PC = k-1, da nach jeder Ausführung eines Befehlt der PC erhöht wird.

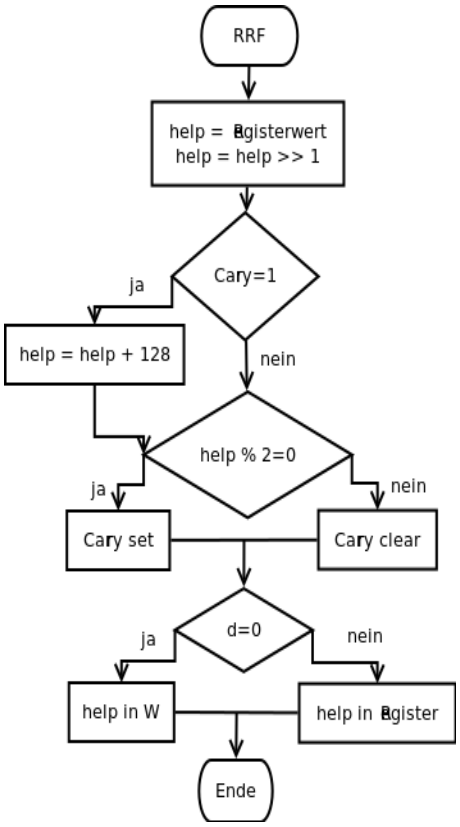


MOVF

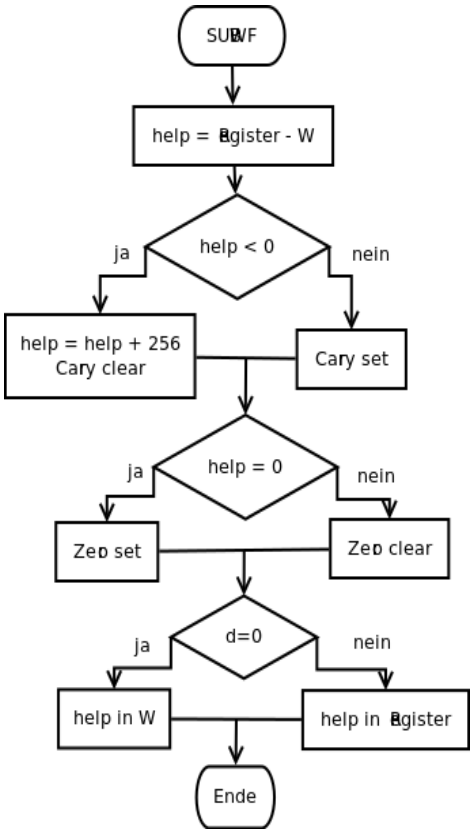
MOVF gehört, ebenso wie RRF, SUBWF und DECFSZ, zu den byteorientierten Befehlen, die immer ein ganzen Register betreffen. Dabei werden den Funktionen das Register f und die Destination d übergeben, welche angibt wohin das Ergebnis der Operation gespeichert werden soll.



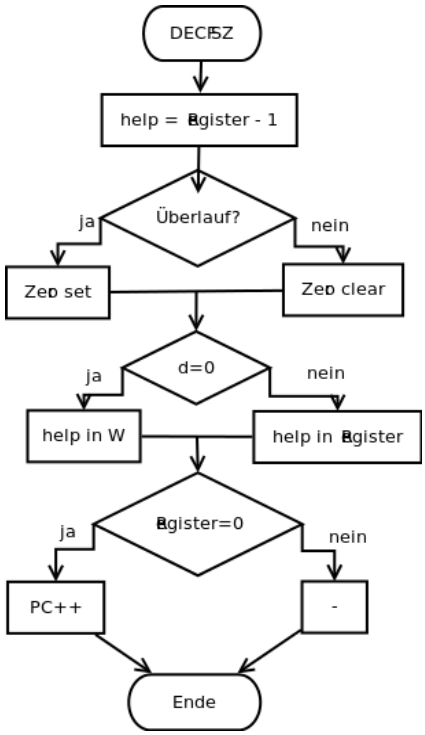
RRF


RRF
help = Registerwert
help = help >> 1
Cary=1
ja
help = help + 128
nein
help % 2=0
ja
nein
Cary set
Cary clear
d=0
ja
nein
help in W
help in Register
Ende

SUBWF

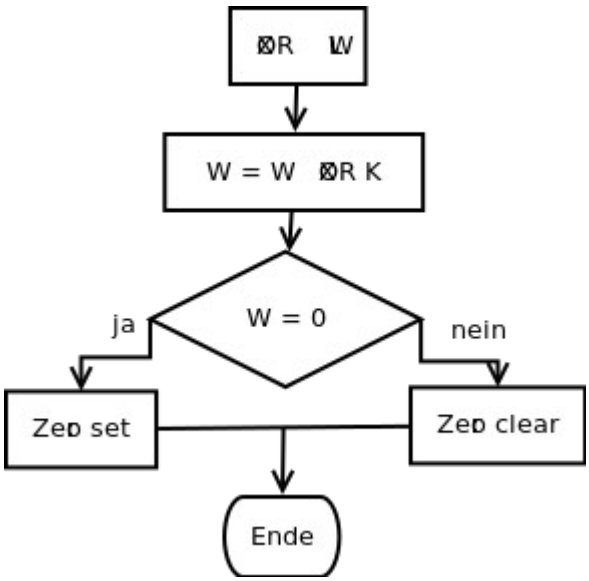

SUBWF
help = Register - W
help < 0
ja
nein
help = help + 256
Cary clear
Cary set
help = 0
ja
nein
Zero set
Zero clear
d=0
ja
nein
help in W
help in Register
Ende

DECFSZ



XORLW

XORLW gehört zu den Literal & Control Operationen, bei denen den Funktionen ein Literal k übergeben wird.



Flags

Die Flags sind in der GUI sichtbar und werden in den essentiellen Befehlen überprüft und gesetzt. Diese sind: Carry (Überlauf), Digit Carry (Überlauf der unteren 4 Bits), Zero (Ergebnis einer arithmetischen Operation war null), Powerdown, Timeout, rp0 (Register Bank Select).

Interrupts

Implementiert wurden der TMR0- Interrupt (Überlauf) und der RB- Interrupt. Sowohl beim TMR0, als auch beim RB, wurde die Flankenerkennung realisiert. Wichtig dabei ist, dass die Interrupts auch asynchron zum Prozessortakt auftreten können, aber das Anspringen der Interrupt-Adresse synchron mit dem folgenden Zyklus erfolgt. Zu Beginn eines jeden Befehlszyklus wird dann das Intcon – Register überprüft und gegebenenfalls die Interrupt-Adresse angesprungen. Es wurden also Funktionen für den Port B und den TMR0 implementiert, die bei Änderungen an den entsprechenden Bits die Interruptbedingungen überprüfen. Ist eine erfüllt, werden die entsprechenden Bits gesetzt.

TRIS – Register

Das TRIS – Register ist auch in der GUI sichtbar. Ein *i* steht für input, ein *o* für Output. Die Ports, die die Datenrichtung Output besitzen, sind nicht vom Anwender veränderbar, also können nicht angeklickt werden.

Fazit

Durch das Betreuen eines Projektes von Beginn an, wurden viele Fähigkeiten gerade im organisatorischen Teil erlernt. Durch die Implementierung und gerade auch durch Suchen und Finden von gemachten Fehlern, konnte ein weitreichendes Verständnis für die Funktionsweise des Microcontrollers erworben werden.

Durch die Verwendung von Threading für die Ausführung von Einzelbefehlen, wurde ein relativ realitätsnahes Zeitverhalten des Simulators erzeugt. Zwar laufen die Befehle wesentlich langsamer ab als auf dem realen Prozessor, dies ist jedoch durch die Einschränkung der Programmiersprache Java bedingt, die hier keine kürzeren Wartezeiten zuließ. Durch die Anpassung (Verlangsamung) der restlichen äußeren Einflüsse lässt sich mit Hilfe des Simulators also trotzdem eine Aussage über das Verhalten des PIC16C84 bei bestimmten Programmen als Eingabe machen.

Zudem sind einige Funktionen des realen PIC16F84 noch nicht implementiert. Weiteres dazu unten.

Nicht implementierte Funktionen

- EEPROM Funktionalität
- Prescaler, Watchdog
- Externe Hardware über serielle Schnittstelle
- 7- Segment Anzeige
- Signalgenerator für externen Takt an RA4

Anhang

Datenblatt

Datenblatt.pdf

LST – Files

BA_Test.LST

BA_Test2.lst

Lastenheft

Lastenheft.pdf