Hochschule Bremen
Fakultät Elektrotechnik und Informatik

Embedded Systems

Laboraufgabe 2

Nachweis harter Echtzeiteigenschaften

Semester:	SoSe 2015
Laboranten:	Marteyn Weidenbach Artur Schmidt Tobias Wenzig
Datum:	21.05.2015

Inhaltsverzeichnis

Marteyn Weidenbach,
Artur Schmidt,
Tobias Wenzig

In diesem Laborversuch soll ein System auf dessen Echtzeiteigenschaften Untersucht werden. Hierzu werden mittels des Messtools von Keil µVision die Ausführungszeiten gemessen und daraufhin überprüft, ob sie kleiner als die obere Schranke von 400 µs sind. Als Anwendung dient hierbei die in der Laboraufgabe 1 entwickelte Fahrstuhlsteuerung.

1 Einleitung

In diesem Laborversuch soll ein Nachweise echter Echtzeiteigenschaften erfolgen. Dies ist deshalb so wichtig, weil es bei eingebetteten Systemen nicht ohne weiteres Möglich ist neue Software aufzuspielen und so nachträglich Softwarefehler zu beheben. Weiterhin macht die in der Regel hohe Stückzahl ein nachträgliches Austauschen der Komponente sehr kostspielig.
Dies alles zeigt wie wichtig das Testen der Software ist.
Eine wichtige Rolle spielt hierbei der Nachweis der Echtzeiteigenschaften. Das bedeutet, dass die eingehenden Signale schnell genug verarbeitet werden um rechtzeitig am Ausgang, für z.B. eine andere Komponente, anzuliegen.
In diesem Zusammenhang wichtig sind die folgenden Abdeckungen:

- vollständige Zustandsübergangsabdeckung
- vollständige Anweisungsabdeckung
- Verzweigungsabdeckung
- Pfadabdeckung
- Worst-Case-Ausführungszeit
- obere Schranke

Mit Hilfe des Zustandsgraphen (Siehe Anhang Abbildung 6.1) kann zunächst eine vollständige Zustandsübergangsabdeckung erreicht werden. Dabei wird jeder mögliche Zustandsübergang durchlaufen. Ein testen mittels einer Testfolge die eine vollständige Zustandsübergangsabdeckung gewährleistet, weißt lediglich eingeschränkt die Funktionalität des Systems nach. Die Funktionalität ist deswegen nur eingeschränkt nachweisbar, weil über das zeitliche Verhalten des Systems keine Aussagen getroffen werden können. So kann es passieren, dass das System zwar seine Aufgabe richtig erfüllt, aber so langsam, dass die richtigen Werte nicht rechtzeitig an den Ausgängen anliegen.

Um das nachzuweisen, muss der Code getestet werden. Hierzu eignen sich die vollständige Anweisungs-, Verzweigungs-, und Pfadabdeckung.
Mit Hilfe der Anweisungsabdeckung kann festgestellt werden, wie viel Prozent aller Anweisungen ausgeführt werden. Dadurch kann beispielsweise toter Code aufgespürt und entfernt werden, um das Programm so klein wie möglich zu halten.
Bei der vollständigen Verzweigungsabdeckung soll jede Abfrage einmal die Werte Wahr und Falsch annehmen. Dieser Test ist schwächer als die vollständige Pfadabdeckung. Bei dieser soll nämlich jeder mögliche Pfad durchlaufen werden. Eine vollständig durchgeführte Pfadabdeckung beinhaltet auch eine immer eine vollständige Anweisungs-, sowie Verzweigungsabdeckung und stellt somit den bestmöglichen Test dar.
In der Praxis ist eine vollständige Pfadabdeckung aber selten möglich, da das austesten zu lange dauern würde.

2 Codeanalyse

Der Programmcode der Fahrstuhlsteuerung kann in 2 Phasen gegliedert werden : Deklarationsteil und die Funktion *main*.
Im Deklarationsteil werden einzelnen Bits aus dem Special Function Register die Eingänge (Port P1.1, P1.2 und P1.7) sowie die Ausgänge (Port P5.1 und P5.2) zugewiesen, sowie einzelne bits deklariert mit welchen im Folgenden die Berechnungen erfolgen werden. Weiterhin werden die Zustände mittels Enum erzeugt. Hierbei wurde auch zugleich ein neuer Typ *zustaende_t* definiert.

Der zweite Teil des Codes, die *main()*-Funktion, kümmert sich um die Ein- und Ausgänge, sowie um die Berechnung der Signale.
Die *main()*-Funktion lässt sich wiederum in 3 Phasen aufteilen:

- Eingabephase
- Berechnungsphase
- Ausgabephase

Wichtig zu erwähnen ist, dass alle drei Phasen innerhalb einer Endlosschleife (while(true)) stattfinden und die Funktion somit niemals terminiert.

Die erste Phase ist die „Eingabephase“ und ist in dem Programm als *readInputVars()* implementiert. In dieser Funktion werden die an den Ports anliegenden Signale gelesen und internen Variablen zugewiesen.
Die nächste Phase ist die Berechnungsphase. Die Methode *computeOutputVars()* dient als Implementierung dieser Phase. Diese ist die aufwändigste der drei Phasen, denn hier werden die, den Variablen zuvor zugewiesenen, Werte mittels If-Abfragen ausgewertet und dementsprechend der Zustand geändert.
Weiterhin ist die Zustandsübergangsfunktion hier definiert und durch eine Switch-Case-Anweisung realisiert.

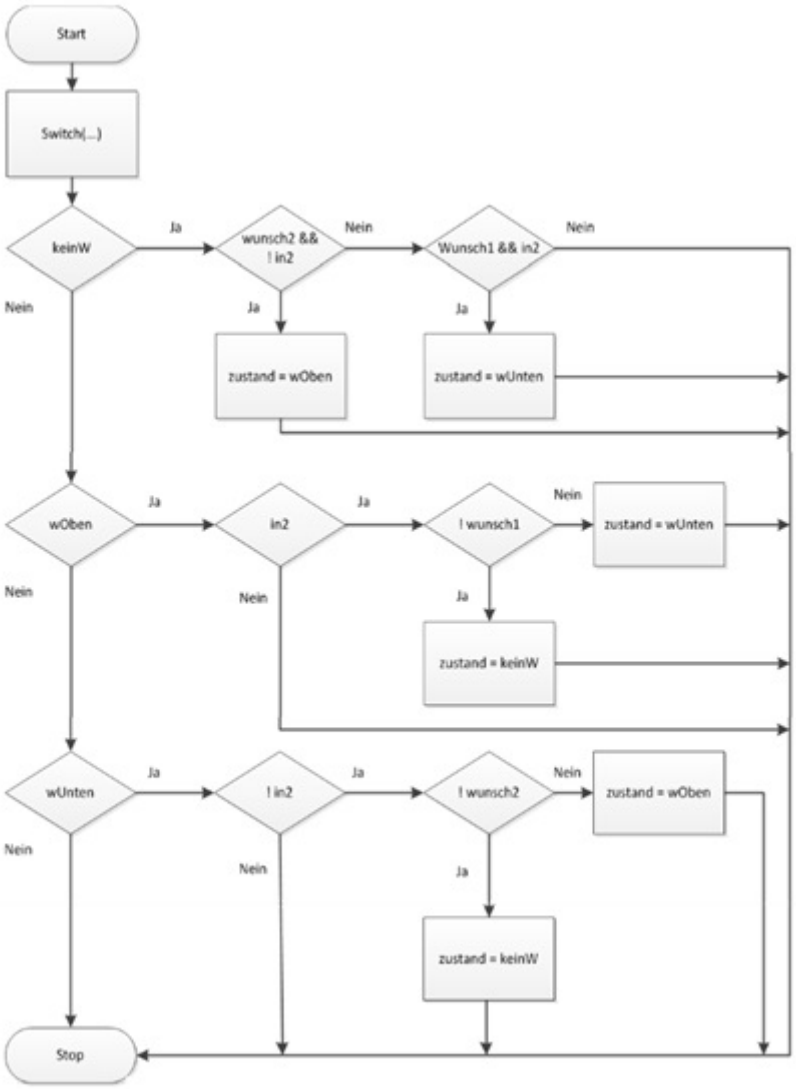

Abbildung 2.1: Zustandsübergangsfunktion per switch-case-Anweisung

Außerdem ist die Ausgabefunktion ebenfalls mit einer „switch-case“ umgesetzt.

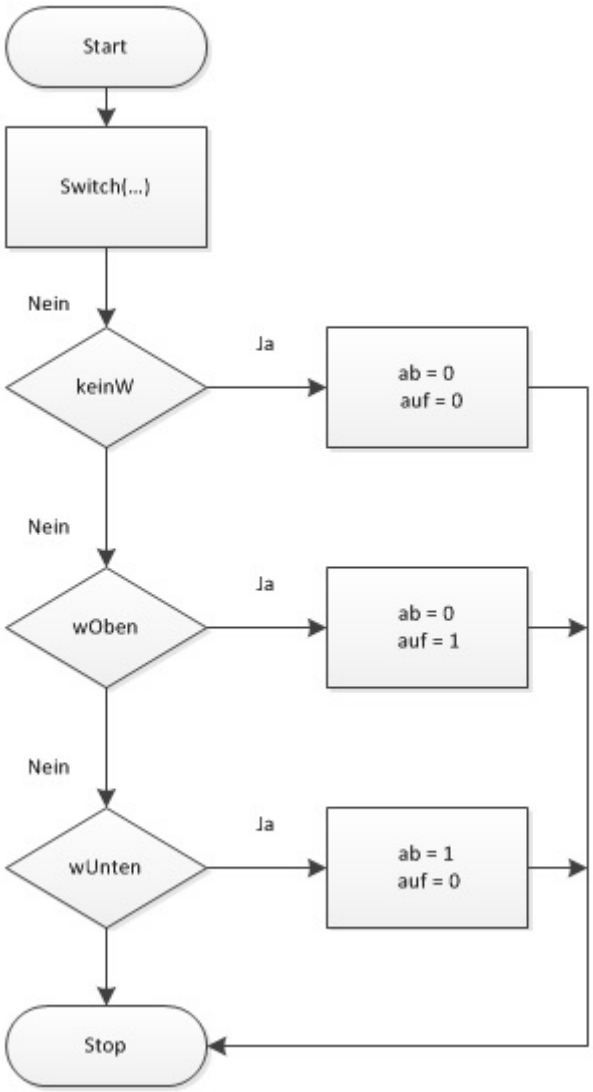

Abbildung 2.2: Ausgabefunktion der Berechnungsphase per switch-case-Anweisung

Als letzte Phase ist die Ausgabephase in der Methode *writeOuputVars()* implementiert worden. In dieser Methode werden die in der *computeOutputVars()* berechneten Signale an die Ausgänge angelegt.

Das Gesamtkonstrukt entspricht einem Moore-Automaten.

3 Testfolge

Der Funktionstest erfolgte mit einer Testfolge die als Ziel die vollständige Zustandsübergangsabdeckung hatte.(Siehe hierzu Tabelle **??**. Mit Hilfe dieser war gewährleistet, dass das Programm ordnungsgemäß funktionierte. In der Regel folgt aus einer Testfolge für die vollständige Zustandsübergangfunktion nicht automatisch eine vollständige Pfadabdeckung, in diesem Fall ist das jedoch so. Dennoch wurde eine kürzere und kompaktere Testfolge speziell für die vollständige Pfadabdeckung entwickelt. Siehe hierzu Tabelle 1. Dabei ist anzumerken, dass die Testfolge im Zustand *keinW* beginnt.

Tabelle 1: Testfolge für die vollständige Pfadabdeckung

Nr.	**in2_i**	**wunsch2_i**	**wunsch1_i**	**ab_o**	**auf_o**
1	1	1	0	0	0
2	1	**0**	0	0	0
3	**0**	0	0	0	0
4	0	0	**1**	0	0
5	**1**	0	1	1	0
6	1	0	**0**	1	0
7	**0**	0	0	0	0
8	0	**1**	0	0	1
9	0	**0**	0	0	1
10	**1**	0	0	0	0
11	1	0	**1**	1	0
12	1	0	**0**	1	0
13	1	**1**	0	1	0
14	**0**	1	0	0	1
15	0	**0**	0	0	1
16	0	0	**1**	0	1
17	**1**	0	1	1	0

4 Performance-Messung

4.1 Erste Messung

Nach dem Ausführen des Performance Analyzers wird die Ausführungszeit jeder aufgerufenen Funktion anhand eines Balkendiagramms angezeigt (siehe Abbildung 1). Es ist zu erkennen, dass die Funktion *computeOutputVars()* die längste Ausführungszeit vorweist.

Abbildung 4.1: Balkendiagramm zur prozentualen Zeitdauer der Funktionen

Wenn auf die einzelnen Funktionen geklickt wird, werden nähere Informationen zur jeweiligen Laufzeit angezeigt (siehe Abbildung 2).

- min time: Gibt die minimale Zeit an, die das Programm in der Funktion bzw. im Adressbereich verbracht hat.
- max time: Gibt die maximale Zeit an, die das Programm in der Funktion bzw. im Adressbereich verbracht hat.
- avg time: Gibt die durchschnittliche Zeit an, die das Programm in der Funktion bzw. im Adressbereich verbracht hat. Diese berechnet sich wie folgt:

$$avg = \frac{totalTime}{count}$$

- total time: Gibt die gesamte Zeit an, die das Programm in der Funktion bzw. im Adressbereich verbracht hat. Bei zu langer Laufzeit werden an dieser Stelle Zahlen abgeschnitten.
- %: Gibt den Prozentsatz der gesamten Zeit an, die das Programm in der Funktion bzw. im Adressbereich verbracht hat.
- count: Gibt die Häufigkeit an, mit der die Funktion bzw. der Adressbereich betreten wurde.

Die Abbildung 4.2 zeigt die Analyse für die main-Funktion auf.

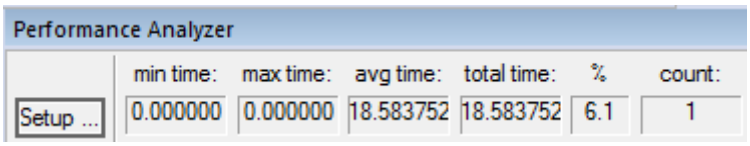

Abbildung 4.2: Messdaten der *main*

Die Abbildung 4.3 zeigt die Analyse für die readInputVars-Funktion auf.

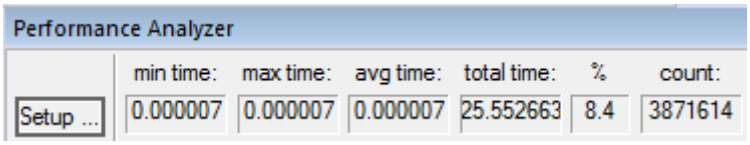

Abbildung 4.3: Messdaten der Funktion *readInputVars()*

Die Abbildung 4.4 zeigt die Analyse für die computeOutputVars-Funktion auf.

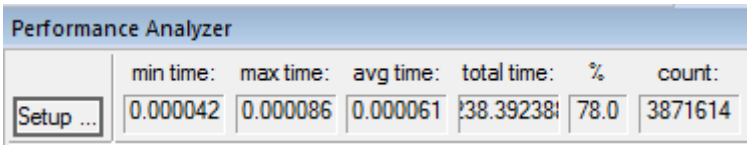

Abbildung 4.4: Messdaten der Funktion *ComputeOutputVars()*

Die Abbildung 4.5 zeigt die Analyse für die writeOutputVars-Funktion auf.

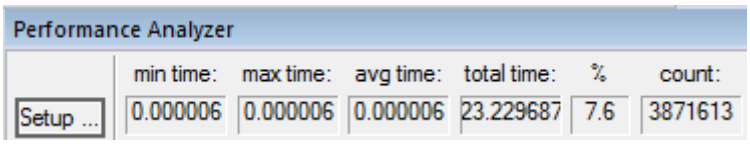

Abbildung 4.5: Messdaten der Funktion *writeOutputVars*

4.2 Zweite Messung

Zur Berechnung der Ausführungszeit eines Schleifendurchlaufs (while-Schleife) in der main-Funktion, wurde eine erneute (kürzere) Messung durchgeführt. Grund dafür ist, dass die 1. Messung eine zu lange Laufzeit vorweist und dadurch die „total time“nicht richtig dargestellt wird. Die Abbildung 4.6 zeigt die Analyse für die main-Funktion.

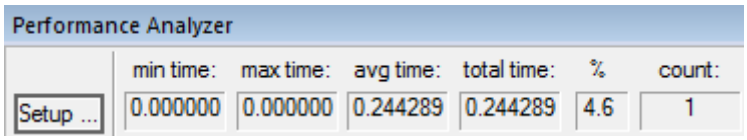

Abbildung 4.6: Messdaten der 2. Messung der main

Die Abbildung 4.7 zeigt die Analyse für die writeOutputVars-Funktion auf.

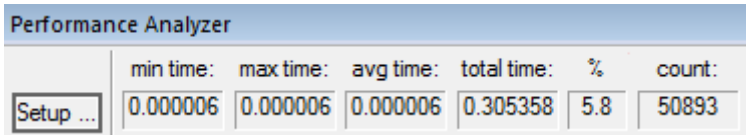

Abbildung 4.7: Messdaten der 2. Messung der Funktion *writeOutputVars()*

5 Auswertung

Im Folgenden wird nachgewiesen, dass das Gesamtsystem innerhalb von 400µs nach einer beliebigen Eingabe mit Sicherheit die zugehörige Ausgabe gemacht hat. Zur Ermittlung einer oberen Schranke der Reaktionszeit wird die maximale Ausführungszeit jeder Funktion betrachtet. Aus den Messergebnissen lassen sich folgende Werte (max time) ablesen:

- readInputVars-Funktion: 7µs
- writeOutputVars-Funktion: 6µs
- computeOutpurVars-Funktion: 86µs

Zur Ermittlung der Ausführungszeit der main-Funktion wurde eine zweite Messung durchgeführt, da die total time einen zu hohen Wert erreicht hatte und dadurch abgeschnitten wurde.
Die maximale Ausführungzeit der main-Funktion kann nicht wie bei den anderen Funktionen direkt aus den Messergebnissen abgelesen werden. Der Wert für die max time wird mit 0µs angegeben. Der Grund dafür ist, dass die while-Schleife nie verlassen wird. Die Ausführungszeit der main-Funktion kann aber mit Hilfe der total time bestimmt werden. Dazu wird bei eine der drei Funktionen die in der main-Funktion aufgerufen wird, der count Messwert ausgelesen. Dieser beträgt beispielsweise bei der writeOutputVars -Funktion 50893. Es lässt sich also sagen, dass die while-Schleife 50893 durchlaufen wurde, bevor das Programm beendet wurde.
Die Ausführungszeit der main-Funktion berechnet sich also wie folgt:

$$t_main = \frac{totalTimeMain}{countderwriteOutputVars}$$
$$t_main = \frac{0.244289\mu s}{50893}$$
$$t_main \approx 48\mu s$$

Die gesamte Ausführungszeit berechnet sich also wie folgt:

$$t \approx 7\mu s + 6\mu s + 8\mu s + 48\mu s \approx 99\mu s$$

Da nur einmal pro while-Schleifendurchlauf die Eingabe ausgelesen wird, kann es im schlimmsten Fall zu einer Verzögerung von fast einem ganzen Schleifendurchlauf kommen, wie Abbildung 5.1 zeigt.

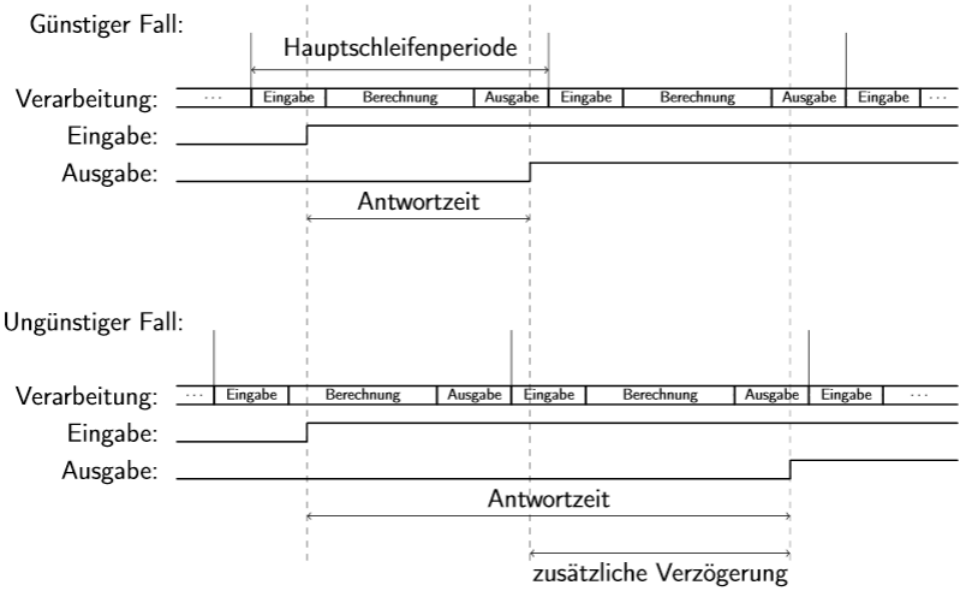

Abbildung 5.1: Darstellung der Verzögerung

Die berechnete Ausführungszeit muss also noch mit zwei Multipliziert werden, um auch diesen Fall ausschließen zu können. Dadurch ergibt sich eine obere Schranke von ≈198µs. Diese liegt unter dem geforderten Wert von 400µs und damit ist die harte Echtzeiteigenschaften nachgewiesen.

6 Anhang

6.1 Code

Listing 1: Quellcode

```
#include <REG515C.H>

// Laboraufgabe 2
// Tobias Wenzig, Marteyn Weidenbach, Artur Schmidt

// Eingaenge
sbit wunsch1_i = P1^1;
sbit wunsch2_i = P1^2;
sbit in2_i = P1^7;
bit wunsch1;
bit wunsch2;
bit in2;

// Ausgaenge
sbit ab_o = P5^1;
sbit auf_o = P5^2;
bit ab;
```

```
bit auf;

// functionen
static void readInputVars();
static void computeOutputVars();
static void writeOutputVars();

typedef enum {keinW, wOben, wUnten} zustaende_t;
zustaende_t zustand = keinW;

void main(){
        while(1){ // Endlosschleife
                // Eingabephase
                readInputVars();
                // Berechnungsphase
                computeOutputVars();
                // Ausgabephase
                writeOutputVars();
        }
}

static void readInputVars(){
        wunsch1 = wunsch1_i;
        wunsch2 = wunsch2_i;
        in2 = in2_i;
        return;
}

static void computeOutputVars(){
        // Zustandsuebergangsfunktion
        switch(zustand) {
                case keinW:
                        if(wunsch2 && !in2){
                                zustand = wOben;
                        }
                        if(wunsch1 && in2){
                                zustand = wUnten;
                        }
                        break;
                case wOben:
                        if(in2)
                        {
                             if(!wunsch1){
                                zustand = keinW;
                             }
                             else{
                                zustand = wUnten;
                                }
                        }
                        break;
```

```
                case wUnten:
                        if(!in2)
                        {
                            if(!wunsch2){
                                zustand = keinW;
                            else{
                                zustand = wOben;
                            }
                        }
                        break;
        }
        // Ausgabefunktion
        switch(zustand){
                case keinW:
                        ab = 0;
                        auf = 0;
                        break;
                case wOben:
                        ab = 0;
                        auf = 1;
                        break;
                case wUnten:
                        ab = 1;
                        auf = 0;
                        break;
        }
        return;
}

static void writeOutputVars(){
        ab_o = !ab;
        auf_o = !auf;
        return;
}
```

6.2 Zustandsautomat

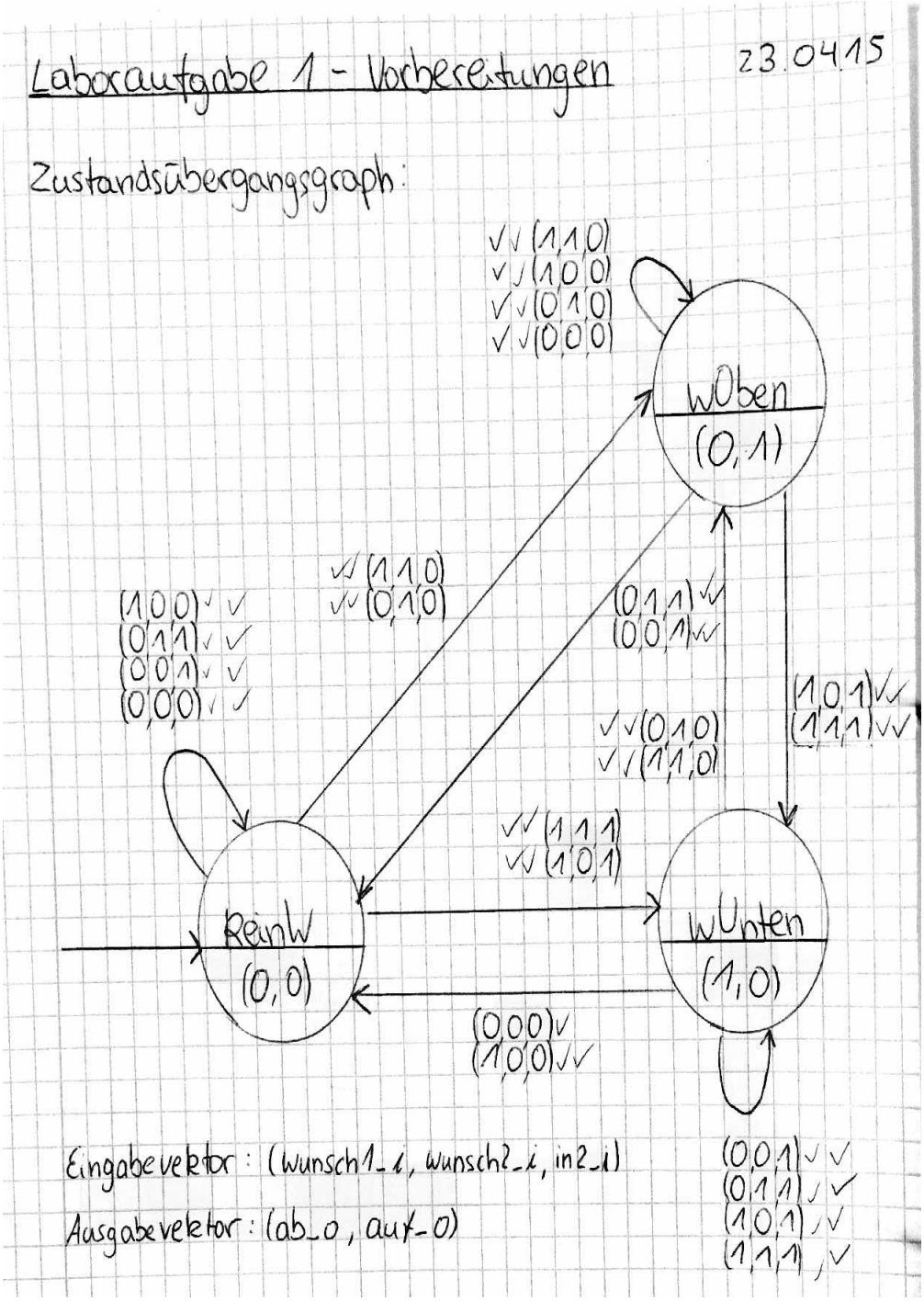

Abbildung 6.1: Zustandsautomat der Fahrstuhlsteuerung

6.2.1 Testfolge für die vollständige Zustandsübergangsabdeckung

HIER DIE TABELLE mit dem Label Zustand EINFÜGEN! DANKE!