VHDL Praktikum

Aufgabe 1: Ein- / Ausgabemodul

Bertram, Alexander (B_TInf 2616, 6. Fachsemester)

Inhaltsverzeichnis

1. Aufgabenstellung

Die Aufgabe besteht aus zwei Teilaufgaben.

1.1. Teilaufgabe 1

In der ersten Teilaufgabe sollen zwei Zähler implementiert werden, die per Tastendruck in- bzw. dekrementiert werden können. Zur Darstellung der Zähler werden zwei vierstellige 7-Segment-Anzeigen verwendet.

1.2. Teilaufgabe 2

Die Zähler aus der ersten Teilaufgabe sollen so modifiziert bzw. erweitert werden, dass das In- bzw. Dekrementieren jeden Zählers über jeweils einen Drehencoder möglich ist. Das Drehen des Encoders soll den Zähler je nach Drehrichtung in- bzw. dekrementieren. Die Wirkrichtung soll sich mittels jeweils eines Schalters umkehren lassen.

Zusätzlich ist ein Encodersimulator zu implementieren, der das Verhalten des Drehencoders möglichst detailliert abbildet.

2. Bedienungsanleitung

2.1. Komponenten und ihre Funktionen

Auf der folgenden Grafik und in der darauf folgenden Tabelle sind die Komponenten und ihre Funktionen dargestellt. Die Zahl hinter den einzelnen Funktionen bedeutet, dass diese Komponente für die Anzeige mit der entsprechenden Zahl zuständig ist.

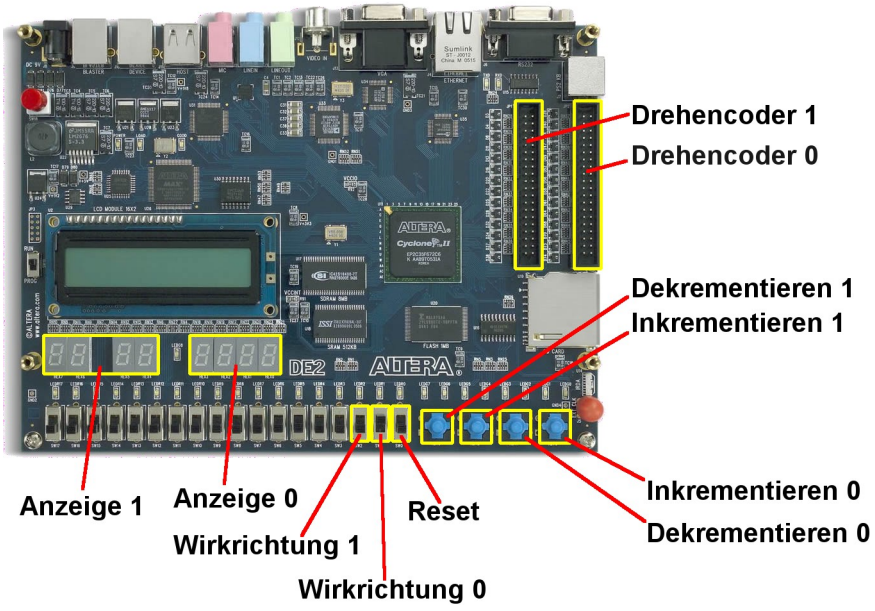

Abbildung 1: Übersicht der Komponenten

Komponente	Funktion
SW0	Reset
HEX0-3	Anzeige 0
HEX4-7	Anzeige 1
KEY0	Inkrementieren der Anzeige 0
KEY1	Dekrementieren der Anzeige 0
KEY3	Inkrementieren der Anzeige 1
KEY4	Dekrementieren der Anzeige 1
JP2	Erweiterungsstecker für Drehencoder 0
JP1	Erweiterungsstecker für Drehencoder 1
SW1	Wirkrichtungsschalter für Anzeige 0
SW2	Wirkrichtungsschalter für Anzeige 1

2.2. Reset

Sobald das Design auf das Board heruntergeladen wurde, sollte es zunächst resettet werden. Dies geschieht in dem der Schalter SW0 in die obere und dann wieder in die untere Position bewegt wird.

2.3. In- / Dekrementieren

Es gibt zwei Wege die Zähler auf den Anzeigen zu ändern. Der minimale Wert, der angezeigt werden kann, ist 0 und der maximale 9999.

Der erste Weg sind die Tasten KEY0 bis KEY3. Die Tastenbelegung ist der oberen Grafik bzw. Tabelle zu entnehmen.

Der zweite Weg sind die Drehencoder. Je nach Lage der Wirkrichtungsschalter werden die Anzeigen in- bzw. dekrementiert. Die verschiedenen Möglichkeiten sind der folgenden Tabelle zu entnehmen:

Schalterposition	Drehrichtung	Funktion
unten	im Uhrzeigersinn	inkrementieren
unten	gegen den Uhrzeigersinn	dekrementieren
oben	im Uhrzeigersinn	dekrementieren
oben	gegen den Uhrzeigersinn	inkrementieren

3. Entwicklungskonfiguration

3.1. Hardware

- Windows-kompatibler PC
- Altera DE2 Development & Education Board
- Zwei Drehencoder der Fachhochschule Wedel

3.2. Software

- Quartus II 7.2sp3 Web Edition
- ModelSim-Altera 6.1g

4. Designstruktur

Dieser Abschnitt beschreibt das Design so wie es entworfen wurde, nach dem Top Down Prinzip. Es wird zuerst die oberste Einheit erklärt und dann die Einheiten, aus denen sie besteht.

4.1. Ein- / Ausgabemodul: EIOModul

4.1.1. Beschreibung

EIOModul ist die Top Level Entity. Sie besteht aus zwei Komponenten: EInput und EOutput. Da es zwei Anzeigen mit gleichem Verhalten gibt, die bedient werden sollen, werden sowohl von EInput als auch von EOutput zwei Instanzen erzeugt und über Signale miteinander verknüpft.

4.1.2. Blockstruktur

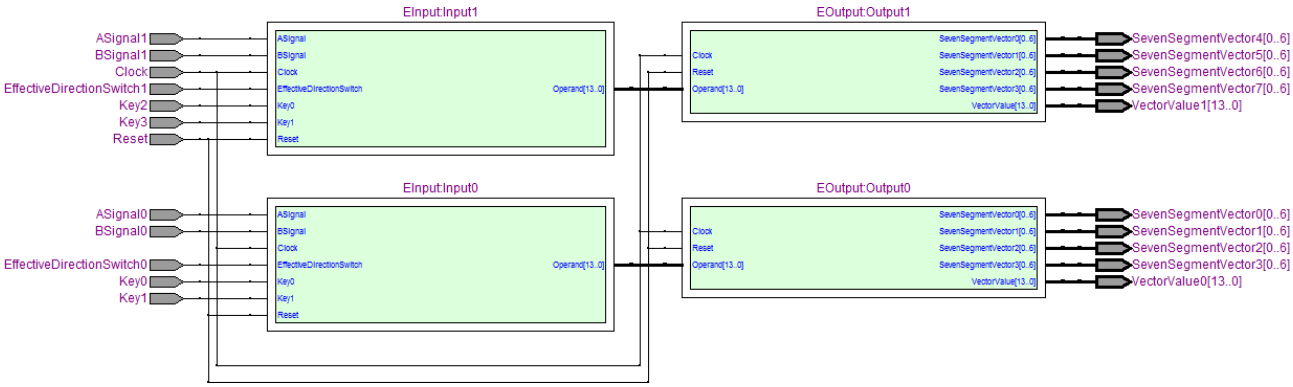

Abbildung 2: EIOModul Blockstruktur

4.2. Allgemeine Datentypen: Types

In der Package Types werden Datentypen und Konstanten definiert, die in mehreren Entities gebraucht werden. Dies erleichtert die Arbeit und schließt Fehlerquellen aus.

4.3. Eingabemodul: EInput

4.3.1. Beschreibung

Das Eingabemodul besteht aus zwei Komponenten: EKeyInput und EEncoderInput. Beide Komponenten werden instantiiert und berechnen als Ausgangssignal je einen Operanden: KeyOperand und EncoderOperand. Anhand dieser Ausgangssignale wird in einer nebenläufigen Anweisung der endgültige Operand bestimmt, der das Ausgangssignal der Entity EInput darstellt.

Der endgültige Operand ist gleich dem KeyOperanden wenn der EncoderOperand gleich Null ist. Ist das nicht der Fall, wird der KeyOperand überprüft. Sollte dieser gleich Null sein, ist der endgültige Operand gleich dem EncoderOperanden. In allen anderen Fällen ist der endgültige Operand gleich Null.

```
Operand <=
      KeyOperand when EncoderOperand = TVectorValue(to_signed(0,
TVectorValue'Length)) else
      EncoderOperand when KeyOperand = TVectorValue(to_signed(0,
TVectorValue'Length)) else
      TVectorValue(to_signed(0, TVectorValue'Length));
```

4.3.2. Blockstruktur

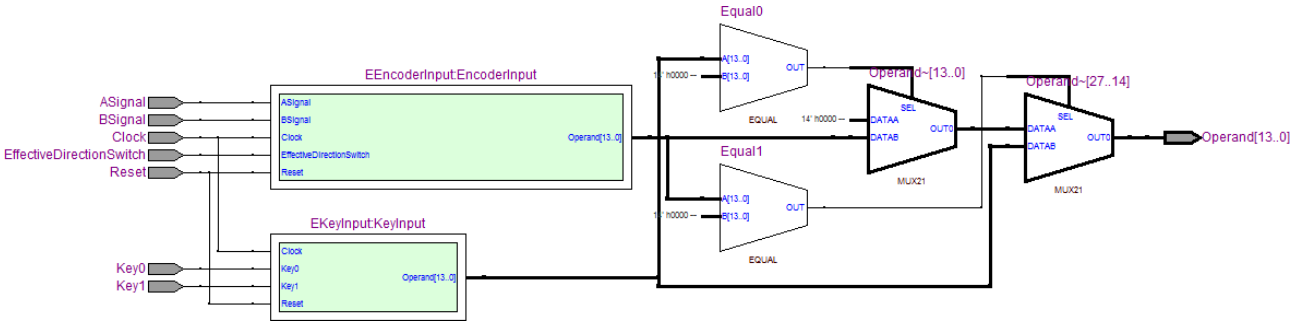

Abbildung 3: EInput Blockstruktur

Links sind die beiden eingebundenen Komponenten, und rechts die Bestimmung des Operanden zu finden.

4.3.3. Ansatzanalyse

Ein anderer Ansatz, das In- bzw. Dekrementieren nach außen bekannt zu geben, wäre es zwei Ausgangssignale zu definieren. Jeweils ein Signal für eine Operation.

Der entscheidende Nachteil dieses Ansatzes ist die fehlende Flexibilität. Es ist im Gegensatz zum gewählten Ansatz nicht möglich, für jede Instanz der Komponente die Operanden fest zu legen.

4.4. Ausgabemodul: EOutput

Die Komponente EOutput bindet zwei weitere Komponenten ein: EAdder und EBinaryToSevenSegmentConverter. Beide werden instantiiert und über Signale verknüpft. Die

Ausgangssignale dieser Komponenten werden unverändert nach außen geführt.

4.5. Eingabe per Tastendruck: EKeyInput

4.5.1. Beschreibung

EKeyInput erzeugt den Operanden, der durch einen Tastendruck entsteht. Die Einheit erhält am Eingang zwei Tastensignale, Key0 und Key1. Diese Tastensignale werden mit dem Systemtakt synchronisiert und ausgewertet. Ein Druck auf eine Taste wird durch eine negative Flanke dargestellt, das Loslassen durch eine positive Flanke.

Das Ausgangssignal dieser Entity ist ein std_logic_vector, der einen Operanden in Binärformat darstellt. Der Operand kann sowohl positiv als auch negativ sein und wird mit Hilfe von zwei Generics bestimmt, PlusOperand und MinusOperand. Dies hat den Vorteil, dass jede Instanz dieser Entity unterschiedliche Operanden zur Verfügung stellen kann.

4.5.2. Testbench: TBKeyInput

Dies ist eine Testbench sowohl für EKeyInput als auch für das gesamte Design EIOInput. Sie simuliert folgende Fälle:

Simulierter Fall	Erwartetes Verhalten	Beobachtetes Verhalten
Alle Tasten beginnend bei KEY0 drücken und wieder loslassen	Beide Anzeigen werden inkrementiert und gleich wieder dekrementiert	Wie erwartet
KEY2 drücken und festhalten	Anzeige 1 wird inkrementiert	Wie erwartet
KEY2 und KEY3 gleichzeitig drücken und wieder loslassen	Anzeige 1 wird inkrementiert	Wie erwartet
KEY2 und KEY3 drücken und festhalten	Anzeige 1 wird inkrementiert	Wie erwartet
KEY2 drücken, festhalten, KEY3 drücken und dann beide nacheinander loslassen	Anzeige 1 wird inkrementiert und gleich wieder dekrementiert	Wie erwartet
Key2 drücken, festhalten, Key3 drücken und dann beide gleichzeitig loslassen	Anzeige 1 verändert sich nicht	Wie erwartet
minimalen Wertebereich unterschreiten	Anzeige bleibt beim minimalen Wert stehen	Wie erwartet
maximalen Wertebereich überschreiten	Anzeige bleibt beim maximalen Wert stehen	Wie erwartet

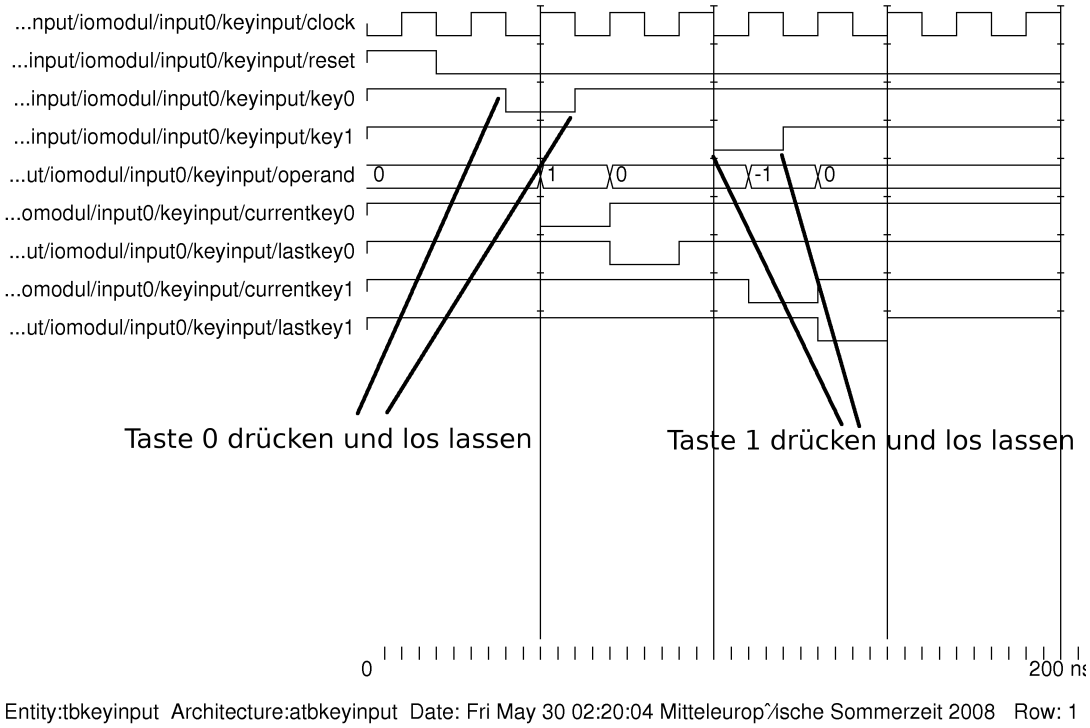

Abbildung 4: Auszug aus der Testbench TBKeyInput

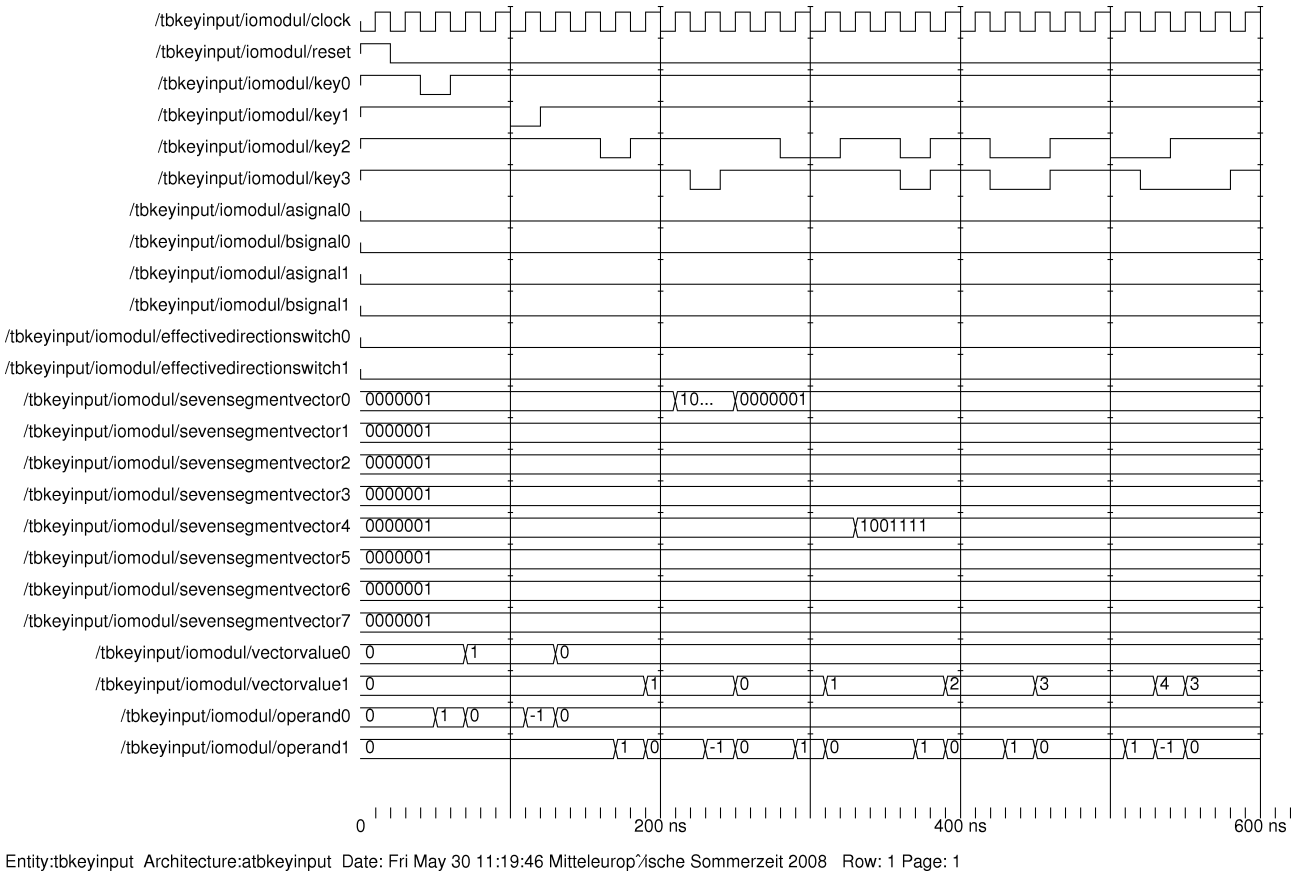

Abbildung 5: Auszug aus der Testbench TBKeyInput, zusätzlich wird der Verlauf der Signale aus EIOModul dargestellt

4.6. Eingabe per Drehencoder: EEncoderInput

4.6.1. Beschreibung

EEncoderInput verhält sich analog zu EKeyInput. Die Eingangssignale entstehen aber nicht durch Tasten, sondern werden von einem Drehencoder und einem Wirkrichtungsschalter erzeugt. EEncoderInput besteht aus einer Komponente und einer nebenläufigen Anweisung. Die Komponente, ERotationDirectionDecoder, wertet die Signale vom Drehencoder aus und bestimmt die Drehrichtung. Sie wird weiter unten detaillierter beschrieben. Die nebenläufige Anweisung bestimmt anhand der Drehrichtung und des Wirkrichtungsschalters das Ausgangssignal, also den Operanden.

Drehrichtung	Wirkrichtungsschalter	Operand
Uhrzeigersinn	unten („low“)	PlusOperand
Uhrzeigersinn	oben („high“)	MinusOperand
gegen den Uhrzeigersinn	unten („low“)	MinusOperand
gegen den Uhrzeigersinn	oben („high“)	PlusOperand

4.6.2. Testbench: TBEncoderInput

Dies ist eine Testbench sowohl für die Komponente EEncoderInput als auch für das gesamte Design EIOModul. Sie simuliert folgende Fälle:

Drehrichtung	Wirkrichtungsschalter	Erwartetes Verhalten	Beobachtetes Verhalten
Uhrzeigersinn	Unten	Anzeige wird inkrementiert	Wie erwartet
Gegen den Uhrzeigersinn	Unten	Anzeige wird dekrementiert	Wie erwartet
Uhrzeigersinn	Oben	Anzeige wird dekrementiert	Wie erwartet
Gegen den Uhrzeigersinn	Oben	Anzeige wird inkrementiert	Wie erwartet

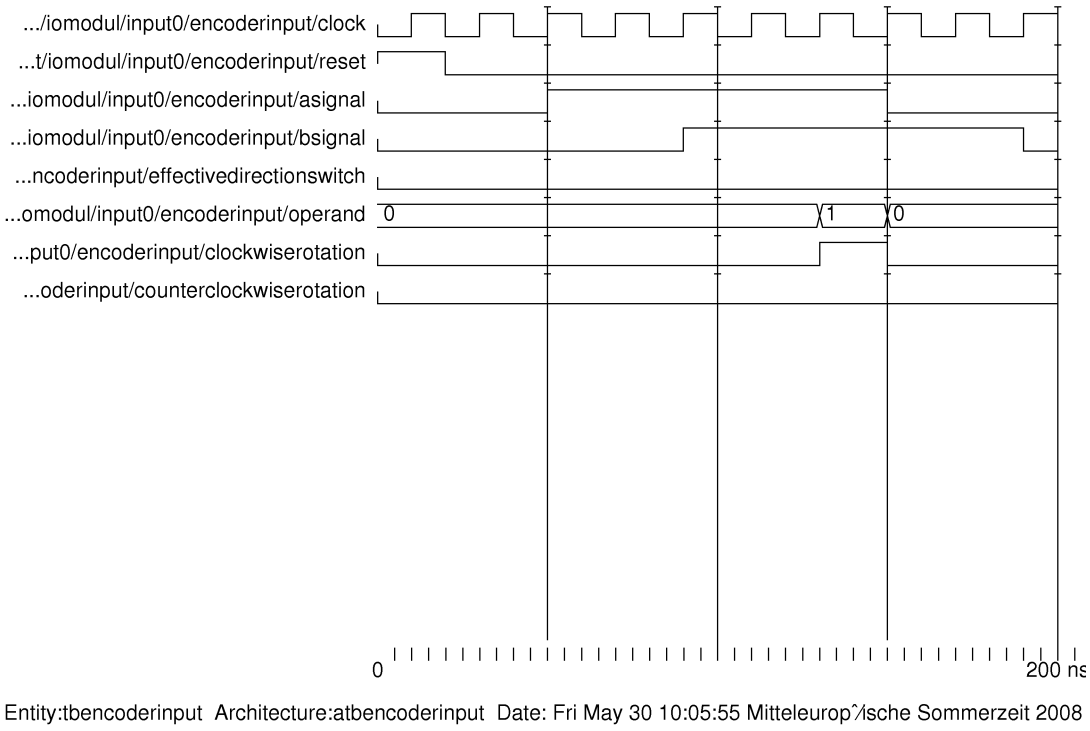

Abbildung 6: Auszug aus der Testbench TBEncoderInput

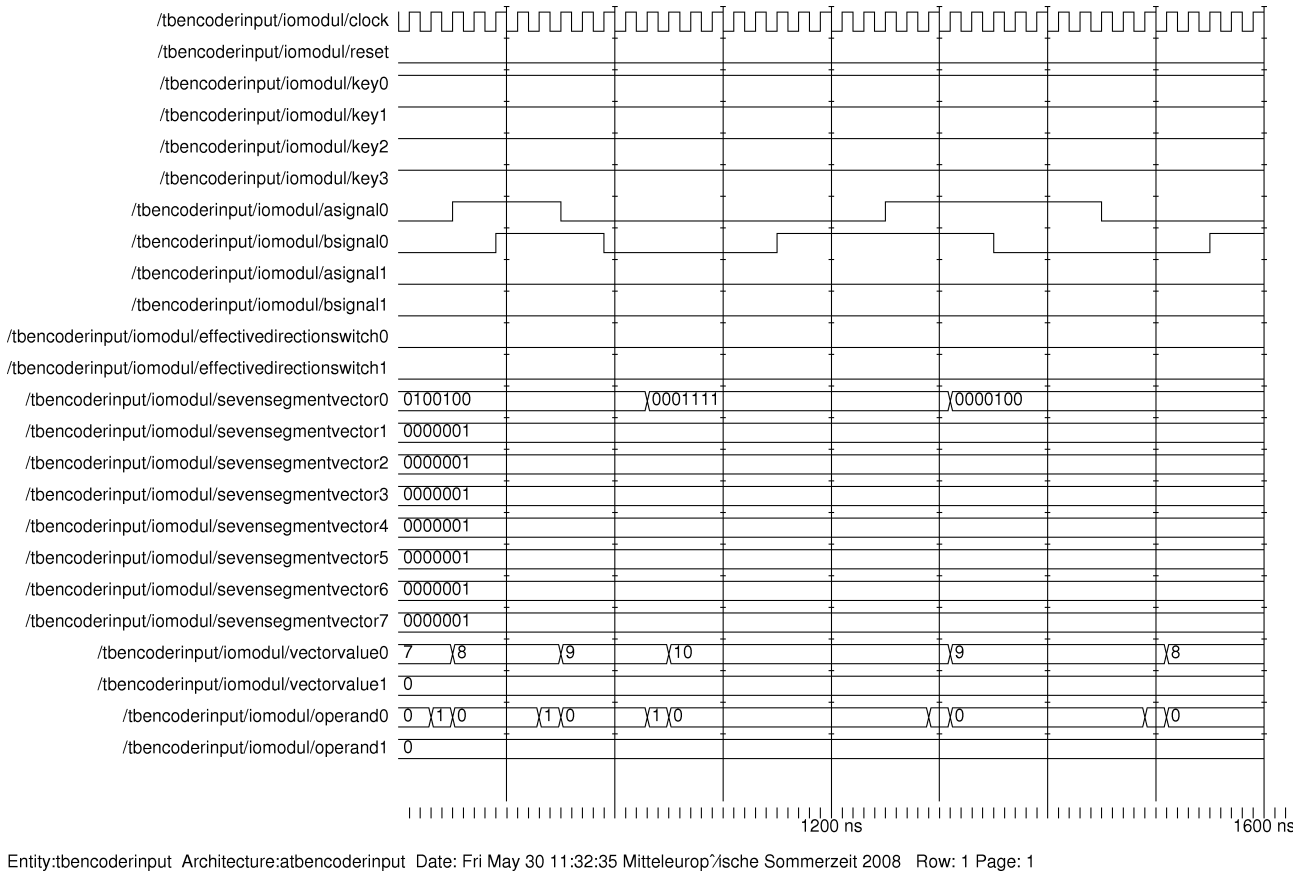

Abbildung 7: Auszug aus der Testbench TBEncoderInput, in dem zusätzlich der Signalverlauf des Moduls EIOModul dargestellt ist

4.7. Drehrichtungsdecoder: ERotationDirectionDecoder

Der Drehrichtungsdecoder verwendet zwei Komponenten, einen Schieberegister für die Flankenerkennung und einen Entpreller, die wichtigste Komponente für den Decoder. Von jeder Komponente werden zwei Instanzen kreiert, da der Drehencoder zwei Signale erzeugt. Der Decoder berechnet aus den beiden entprellten Signalen die Drehrichtung.

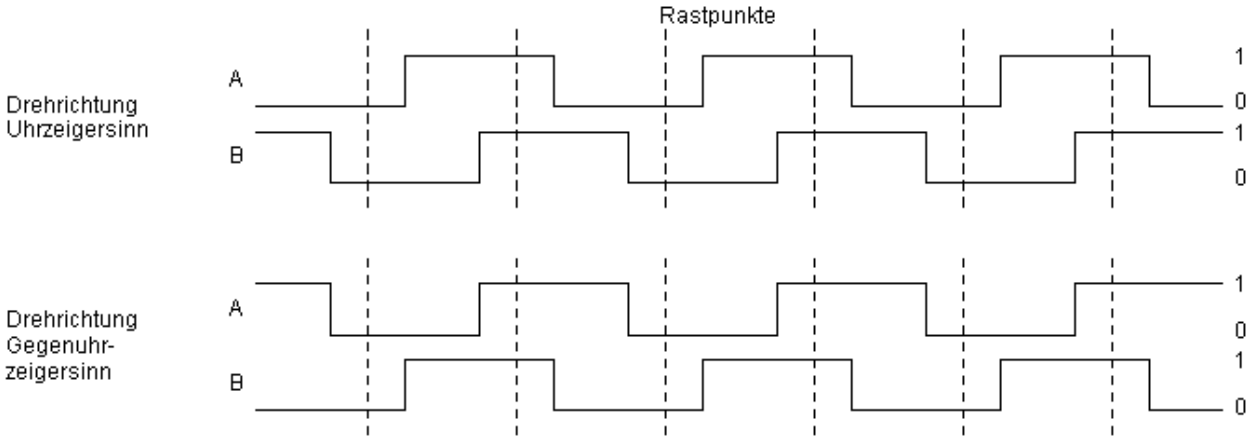

Abbildung 8: Drehencodersignale

Die Erkennung der Drehrichtung ist durch Flankenerkennung in einem Zustandsautomaten realisiert worden.

4.7.1. Zustandsautomat

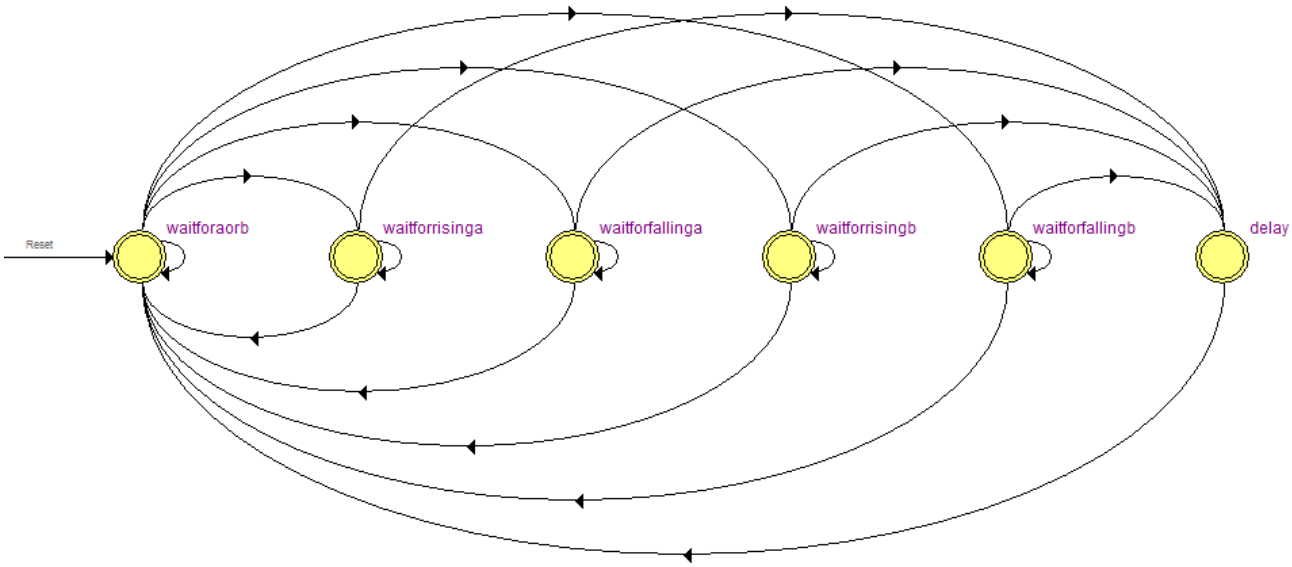

Abbildung 9: Zustandsautomat des Drehdecoders

Der Automat wartet auf eine beliebige Flanke eines Eingangssignal. Wird eine Flanke auf einem der Signale entdeckt, wechselt der Automat den Zustand und wartet auf die dazugehörige Flanke auf dem anderen Signal. Wird auch auf diesem Signal eine Flanke erkannt, wird der Wartezustand aktiv. Dieser macht nichts anderes als in den Ausgangszustand zu wechseln.

4.7.2. Bestimmung der Drehrichtung

Der Drehencoder wird im Uhrzeigersinn gedreht, wenn der Automat auf eine Flanke des Signals B wartet (es muss also eine Flanke auf dem Signal A vorausgegangen sein) und als nächstes in den Wartezustand wechselt.

```
ClockwiseRotation <=
      '1' when (CurrentState = WaitForRisingB or CurrentState = WaitForFallingB)
and NextState = Delay else
      '0';
```

Sollte der Automat im Gegensatz dazu auf eine Flanke auf dem Signal A warten und als nächstes in den Wartezustand wechseln, wir der Drehencoder gegen den Uhrzeigersinn gedreht.

```
CounterClockwiseRotation <=
      '1' when (CurrentState = WaitForRisingA or CurrentState = WaitForFallingA)
and NextState = Delay else
      '0';
```

4.8. Schieberegister: EShiftRegister

Das Schieberegister ist eine aus der Informatik sehr bekannte Komponente und wird hier nicht näher erläutert.

Die Breite des Registers und die Schiebeweite werden bei der Instantiierung der Komponente per Generics festgelegt. Durch ein Eingangssignal kann das Schieberegister ausgeschaltet werden und somit einen Wert dauerhaft speichern.

4.9. Zähler: EDelayCounter

Der Zähler tut nichts anderes als Systemtakte zu zählen. Hat er einen per Generic gesetzten Wert erreicht, wird ein Ausgangssignal gesetzt und der Zähler zurück gesetzt.

4.10. Entpreller: EDebouncer

4.10.1. Beschreibung

Der Entpreller besteht aus einem zweistufigen Schieberegister und einem Zähler. Er sorgt dafür, wie der Name schon sagt, für das Entprellen eines Signals.

Sobald im Schieberegister eine Flanke erkannt wurde, wird der Zähler eingeschaltet und wieder ausgeschaltet, wenn eine bestimmte Anzahl an Takten, die per Generic gesetzt wird, erreicht ist. Für diese Zeit wird das Schieberegister ausgeschaltet.

```
DelayEnable <= (OutputVector(1) xor OutputVector(0)) and not CountLimitReached;
ShiftEnable <= not DelayEnable;
```

4.10.2. Blockstruktur

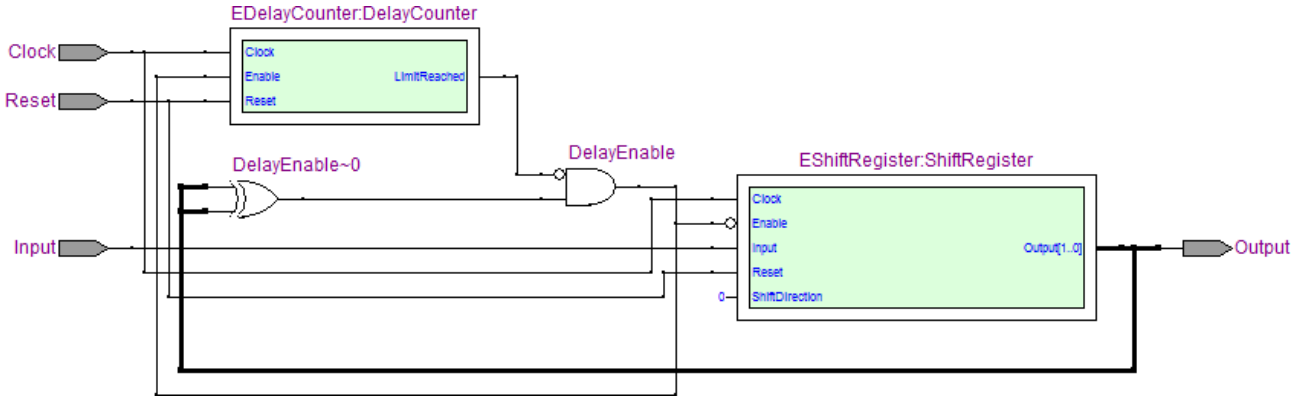

Abbildung 10: EDebouncer Blockstruktur

4.10.3. Testbench: TBDebouncer

Die Testbench erzeugt ein periodisches Signal, das eine niedrigere Frequenz als der Systemtakt hat. Dieses Signal wird nach dem Erkennen einer Flanke für mehrere Takte stabil gehalten.

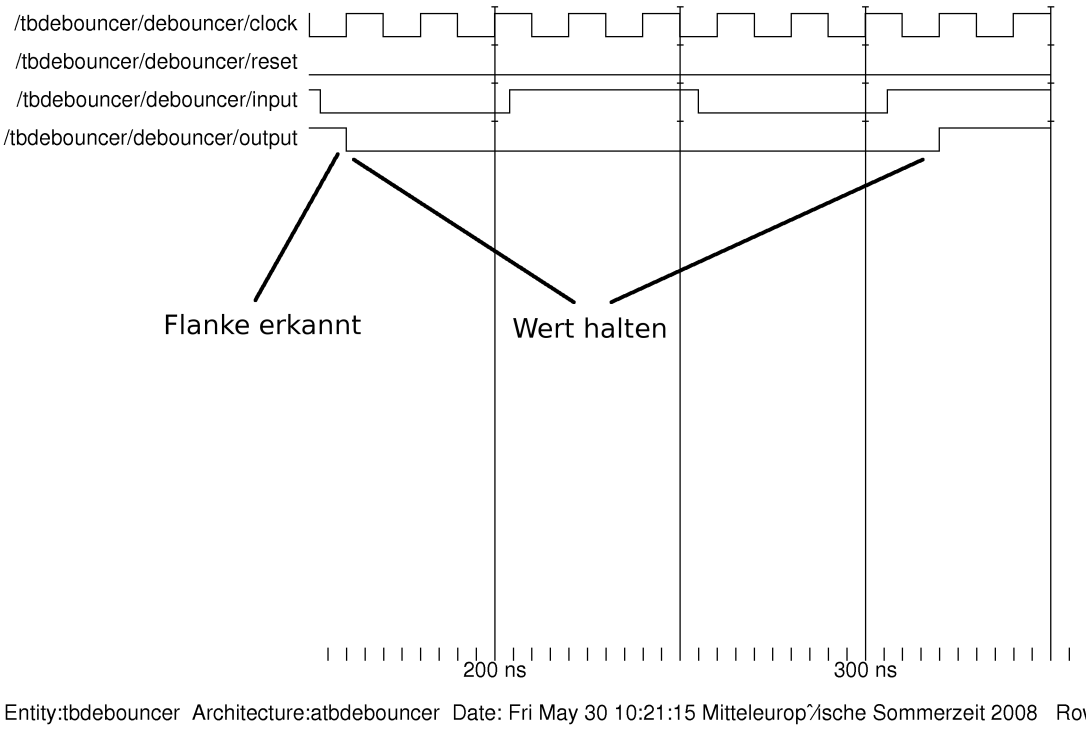

Abbildung 11: Auszug aus der Testbench TBDebouncer

4.11. Addierer: EAdder

EAdder verknüpft einen Operanden mit einer internen Binärzahl. Liegt das Ergebnis dieser Operation innerhalb der Grenzen, die bei der Instantiierung durch zwei Generics festgelegt werden, wird es als Ausgangssignal nach außen geführt.

4.12. Binär zu 7-Segment Konverter: EBinaryToSevenSegmentConverter

Diese Komponente konvertiert eine Binärzahl so, dass sie auf einer vierstelligen 7-Segment-Anzeige im Dezimalformat dargestellt werden kann. Die Ansteuerung der sieben Segmente ist für jede Ziffer in einer Lookup Tabelle in der Package Types hinterlegt.

4.12.1. Zustandsautomat

Die Architektur enthält einen Zustandsautomaten, der für die Umrechnung benötigt wird. Der Automat besteht aus zwei Zuständen: Init und ComputeDigits. Im Zustand Init wird die aktuelle Zahl für die Konvertierung zwischengespeichert und im Zustand ComputeDigits wird sie konvertiert.

4.12.2. Konvertierungssalgorithmus

Die zwischengespeicherte Zahl wird in eine auf der vierstelligen 7-Segment-Anzeige darstellbare Form gebracht, indem so lange der höchste Stellwert abgezogen wird, bis die Zahl kleiner als dieser Stellwert ist. Danach wird die gleiche Prozedur mit dem nächstkleineren Stellenwert wiederholt. Dies wird so lange fortgesetzt, bis die Zahl gleich Null ist. Wenn es soweit ist, wechselt der Automat wieder in den Zustand Init und es wird die nächste Zahl umgewandelt.

4.12.3. Realisierung

Die Stellenwerte sind in einem konstanten Array, das zu einem ROM synthetisiert wird, deklariert.

```
type TPlaceValues is array(0 to 3) of positive range 1 to 1000;

constant cPlaceValues: TPlaceValues := (1000, 100, 10, 1);
```

Die einzelnen Ziffern des Ergebnisses werden in einem Array gespeichert.

```
type TDigits is array(0 to 3) of natural range 0 to 9;
```

Durch diese Konstrukte ist es sehr einfach den oben erläuterten Algorithmus in einer Schleife umzusetzen. Da Schleifen aber nicht bzw. nur in bestimmten Fällen synthesefähig sind, übernimmt der Zustandsautomat diese Rolle.

4.12.4. Analyse

Dieser Algorithmus hat den Nachteil, dass er für die Berechnung einer vierstelligen Zahl relativ viel Zeit, bezogen auf den Systemtakt, braucht. Dies kann vernachlässigt werden, da sich diese Zeiten immer noch in dem Bereich befinden, der für das menschliche Auge zu schnell ist und in dem die Anzeige nicht flackert.

Ein anderer Ansatz, die Zahl umzuwandeln, ist es z.B. mit den Operationen Ganzzahldivision und Modulo zu arbeiten. Da die Division aber sehr viel Logikbausteine verbraucht, ist dieser Ansatz nicht zu gebrauchen.

4.13. Drehencodersimulator: EEncoderSimulator

4.13.1. Beschreibung

Der Simulator versucht so gut wie möglich den Drehencoder zu simulieren. Er besteht aus einer Komponente, der EEncoderSignalGenerator. Diese wird zwei Mal instantiiert und wird weiter unten näher erläutert.

4.13.2. Testbench: TBEncoderSimulator

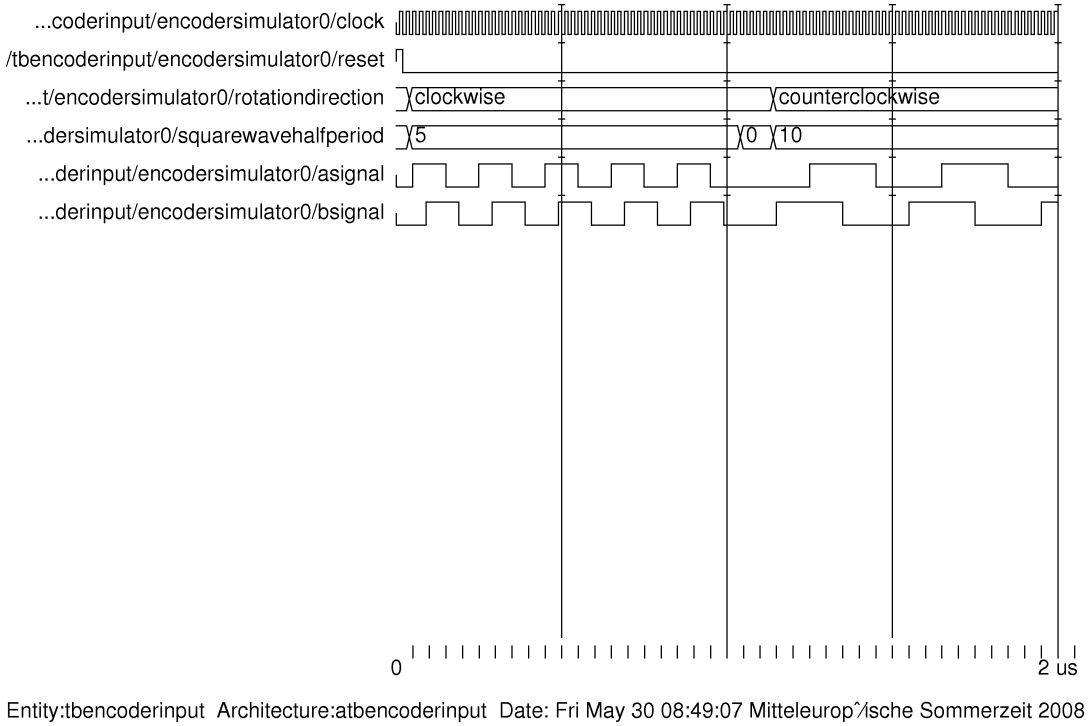

Abbildung 12: Erzeugte Impulse des Simulators

4.14. Rechteckimpulserzeuger: EEncoderSignalGenerator

4.14.1. Beschreibung

Diese Komponente erzeugt ein Rechtecksignal mit einer eventuellen Verzögerung. Zur Erzeugung der Impulse sind in dieser Entity zwei wichtige Eingangssignale deklariert: die Drehrichtung und die Periodenlänge eines halben Rechteckimpulses.

4.14.2. Realisierung

Die Realisierung der Komponente erlaubt ein einfacher Zustandsautomat mit vier Zuständen: Init, Delay, HighLevel und LowLevel.

Im Zustand Init wird anhand der Drehrichtung und der Periodenlänge überprüft, ob der Simulator

eine Drehung simulieren soll. Ist das der Fall, wird die Drehrichtung mit einer bei der Instantiierung per Generic gesetzten Drehrichtung verglichen. Stimmen diese überein, wird zunächst eine Verzögerung des Signals erzeugt. Sollte das nicht zutreffen, wird der letzte Zustand überprüft. Wurde in diesem Zustand ein High-Pegel erzeugt, so wird nun in den LowLevel Zustand gewechselt und umgekehrt.

```
if (RotationDirection = NoRotation) or (SquareWaveHalfPeriod = 0) then
      NextState <= CurrentState;
elsif RotationDirection = GenericRotationDirection then
      NextState <= Delay;
else
      if LastState = LowLevel then
            NextState <= HighLevel;
      else
            NextState <= LowLevel;
      end if;
end if;
```

Im Zustand HighLevel wird ein Zähler überprüft. Hat dieser die Periodenlänge erreicht, wechselt der Zustandsautomat in den Zustand LowLevel.

```
if HighLevelClockCounter = TmpSquareWaveHalfPeriod - 1 then
      NextState <= LowLevel;
else
      NextState <= CurrentState;
end if;
```

In dem Zustand LowLevel wird auch ein Zähler überprüft, der die aktuelle Periodenlänge darstellt, und, sobald es nötig ist, in den Zustand HighLevel gewechselt. Zusätzlich wird aber auch die aktuelle Drehrichtung und die aktuelle Periodenlänge mit den zugehörigen Eingangssignalen verglichen. Dies ist nötig, damit der Automat in den Ausgangszustand wechselt und gegebenenfalls eine neue Verzögerung erzeugt.

```
if LowLevelClockCounter = TmpSquareWaveHalfPeriod - 1 then
      if (TmpSquareWaveHalfPeriod = SquareWaveHalfPeriod) and
(TmpRotationDirection = RotationDirection) then
            NextState <= HighLevel;
      else
            NextState <= Init;
      end if;
else
      NextState <= CurrentState;
end if;
```

Das Zählen der Takte, sowie das Erzeugen des Ausgangssignals, übernehmen eigene Prozesse, die so einfach sind, dass sie hier nicht weiter beschrieben werden.

Da die Impulserzeugung nicht mitten im Ablauf unterbrochen werden kann, **muss** eine bestimmte Zeit vergehen, bevor eine neue Drehrichtung bzw. Periodenlänge eingestellt werden kann. In dieser

Wartezeit, die mindestens eine Perodenlänge lang ist, muss die Drehrichtung des Simulators auf NoDirection oder die Periodenlänge auf Null gesetzt sein. Somit hat der Zustandsautomat genug Zeit, den aktuellen Ablauf zu beenden und erzeugt die erwarteten Signale.