Einführung in die Rechnerarchitektur Praktikum

ANWENDERDOKUMENTATION

Projekt:

VHDL - AM2901

Projektleiter	Martin Zinnecker
Dokumentation	Sandra Grujovic
Formaler Vortrag	Ivan Chimeno

1 Einleitung

Mit diesem Programm werden das Speicherwerk des AM2901, bestehend aus Q-Register und den 16 adressierbaren Registern (RAM), und die ALU-Quelloperandenauswahl mithilfe von VHDL simuliert. Ziel ist die Modularisierung dieser Elemente des AM2901. Es besteht aus 3 Einzelprogrammen, welche jeweils das Q-Register, die RAM und die ALU Quelloperandenauswahl realisieren.

2 Beschreibung des Programms

2.1 RAM – Grobe Funktionsweise

Das RAM-Programm realisiert die 16 adressierbaren Register des AM2901. Zu den Grundfunktionen dieses Moduls gehören das Schreiben eines Wertes in die Speichereinheit, und das Lesen daraus. Die Interne Funktionsweise des Moduls lässt sich der Entwicklerdokumentation entnehmen. Für den Anwender lediglich wichtig ist, dass die Registerstruktur mithilfe eines Arrays realisiert ist. Dieser wird als mem deklariert und in den nachfolgenden Erklärungen benötigt. Das Programm wird primär dazu benutzt, die Speicherstruktur des AM2901 mithilfe einer anderen Sprache zu realisieren, um sie leichter verständlich und später synthetisierbar zu machen, um (vielleicht) mithilfe der VHDL Realisierung einen Bau der entsprechenden Schaltungen möglich zu machen.

Vorhandene Eingänge und ihre Bedeutung

Zu den vorhandenen Eingängen des Moduls gehören die folgenden:

- `d : 16 Bit langer std_logic_vector`
  Dieser Eingang beschreibt unser eigentliches Datenwort, welches wir in die Register schreiben möchten. Bei einem Lesevorgang ist das Datenwort zu vernachlässigen. Wichtig ist hierbei, dass es sich um ein 16-Bit langen std_logic_vector handeln muss.
- `a_address : 16 Bit langer std_logic_vector`
  Dies ist das binär kodierte Register, welches die Adresse enthält, an der man das gesuchte gespeicherte Wort findet. Hierbei ist zu beachten, dass die niederwertigsten 4 Bit der a_address binär kodiert die richtige Speicherzelle adressieren. Beispiel: wir möchten das a_address das Register 5 adressiert, das heißt unsere niederwertigsten 4 Bit sind 0101. Der Rest der Bits der a_address ist wieder zu vernachlässigen und in diesem Programm nicht relevant.

- `b_address : 16 Bit langer std_logic_vector`
  Dies ist das binär kodierte Register, welches die Adresse enthält, an der man das gesuchte gespeicherte Wort findet. Hierbei ist zu beachten, dass die niederwertigsten 4 Bit der b_address binär kodiert die richtige Speicherzelle adressiert. Beispiel: wir möchten das b_address das Register 5 adressiert, das heißt unsere niederwertigsten 4 Bit sind 0101. Der Rest der Bits der b_address ist wieder zu vernachlässigen und in diesem Programm nicht relevant.
- `ce_write & ce_read : std_logic`
  Wenn ce_write/ce_read gesetzt ist, wird der Schreib/Lesevorgang aktiviert. Ist ce_write/ce_read nicht gesetzt, werden die Module und ihre eigentliche Funktionalität nicht aktiviert.
  Darüber hinaus besitzt das Modul auch einen `rst` Eingang, der, wenn gesetzt, alle Register auf undefined zurücksetzt. Außerdem ist dieses Modul getaktet, folglich besitzt es einen `clk` Eingang, der eine Periode von 20ns besitzt. Diese Frequenz war der Aufgabenstellung zu entnehmen. Jedwede Art von Vorgang wird nur bei `rising_edge(clk)` ausgeführt.

Vorhandene Ausgänge und ihre Bedeutung

Zu den vorhandenen Ausgängen des Moduls gehören die folgenden:

- `a_out & b_out : 16 Bit langer std_logic_vector`
  Diese beiden Ausgänge geben nach einem Lesevorgang die Inhalte des Registers an den Adressen a_address und b_address aus. Wenn der Schreibvorgang eingeleitet wird, gibt b_out das gespeicherte Datenwort d aus, während a_out mit einem default Wert gefüllt wird, da es bei einem Schreibvorgang keine konkrete Bedeutung besitzt.

Funktionsweise – Schreiben

Der Schreibvorhang des RAM-Programm Moduls funktioniert wie folgt:

- Lege das zu speichernde Datenwort an den Eingang d.
- Spezifiziere b_address und a_address, achte hierbei auf die Richtlinien, die in Vorhandene Eingänge und ihre Bedeutung genannt worden sind
- Setze ce_write = 1

- Nun wird das Datenwort in das Register in b_address gespeichert. (z.B. sei b_address das binär kodierte 4, dann wird das Datenwort in mem(4) geschrieben.
- In b_out wird nun das zuvor gespeicherte Datenwort ausgegeben, a_out besitzt einen default Wert.
- Setze ce_write wieder auf 0.

Dieser Vorgang lässt sich so oft wiederholen wie man möchte. Ist der Array voll und man möchte wieder einen leeren Array zum Beschreiben haben, reicht es, einen rst Vorgang einzuleiten.

Funktionsweise – Lesen

Der Lesevorgang des RAM-Programm Moduls wird wie folgt realisiert:

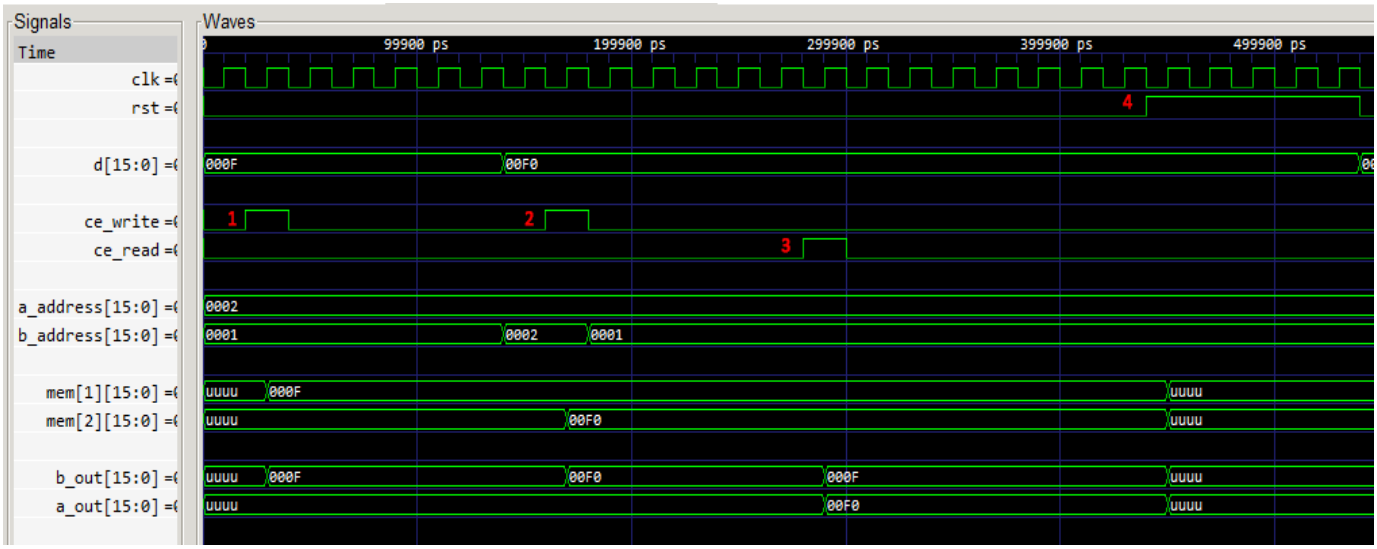
- a_address und b_address genau spezifizieren, nach den oben genannten Richtlinien.
- ce_read auf 1 setzen.
- Die entsprechenden Datenworte werden aus unserem mem-Array gelesen und jeweils auf a_out und b_out gelegt. (z.B. sei b_address das binär kodierte 4, dann wird das Datenwort in mem(4) in b_out ausgegeben.)
- ce_read auf 0 setzen

Was ist zu beachten?

- ce_write und ce_read können nicht gleichzeitig gesetzt werden, auch nicht direkt aufeinanderfolgend im nächsten Takt. Es muss ein Takt Pause dazwischen sein.
- Es ist darauf hinzuweisen, dass beim Lesevorgang tatsächlich etwas an den a_address und b_address spezifizierten Registern steht. Ein Tipp wäre, den Speicherarray erst mit mindestens zwei Werten zu beschreiben bevor man einen Lesevorgang einleitet.
- Das CLK Signal ist momentan auf 20ns gesetzt, kann aber beliebig verändert werden.
- Wie immer gilt es, auf die Datentypen und –längen der einzelnen Eingänge zu achten, bevor man das Modul benutzt.

Beispiel für zwei Schreibvorgänge und einen Lesevorgang des RAM - Moduls

Hiermit wird eine typische Benutzung des RAM-Moduls simuliert.



1) ce_write ist auf 1 gesetzt, die A und B Adressen sind jeweils mit 2 und 1 initialisiert worden. Unser Datenwort ist 000F. Bei der nächsten rising_edge(clk) wird nun unser Datenwort 000F in mem(b_address), also mem(1) gespeichert. Nach dem vollzogenen Schreiben wird ce_write wieder auf 0 gesetzt.
2) Der zweite Schreibvorgang wird eingeleitet. Nun wird ein anderes Datenwort, nämlich 00F0, geschrieben. Die b_address wurde zuvor auch zu 2 geändert. Bei der nächsten rising_edge(clk) wird das Datenwort 00F0 in mem(b_address), in diesem Falle mem(2) gespeichert, was man an der Grafik erkennen kann. Ist der Schreibvorgang beendet, wird das ce_write wieder auf 0 gesetzt.
3) Jetzt erfolgt das Lesen aus den Registern. B_address wurde wieder zu 1 geändert. Nun wird mem(a_address), also mem(2) und mem(b_address), also mem(1) auf a_out und b_out gelegt. Ist der Speichervorgang beendet, wird ce_read wieder auf 0 gesetzt.
4) Zum Schluss erfolgt ein synchroner Reset. rst wird auf 1 gesetzt, der gesamte Array wird wieder mit undefined beschrieben. Der Zyklus beginnt von neuem.

2.2 QREG – Grobe Funktionsweise

Das QREG – Programm simuliert das tatsächliche Q-Register des AM2901. Analog zur RAM bestehen die Grundfunktionen des Moduls aus dem Lese- und Schreibvorgängen. Allerdings besitzt das QREG im Gegensatz zur RAM keinen Speicherarray, sondern lediglich ein einzelnes Signal welches den Speicher des QREG imitiert. Für die genauere interne Funktionalitätsbeschreibung wird auf die Entwicklerdokumentation verwiesen. Auch hier ist das Ziel des Programms, eine einfachere und simplere Art aufzuzeigen, um das AM2901 QREG zu synthetisieren und es später in eine tatsächliche Schaltung umbauen zu können. Das eigentliche Speichern wird durch ein einzelnes Signal, das q_mem, realisiert. Wird ein Schreibvorgang eingeleitet, wird das Datenwort in q_mem gespeichert, wird ein Lesevorgang eingeleitet, wird das der q_out Ausgang mit q_mem beschrieben.

Vorhandene Eingänge und ihre Bedeutung

Zu den vorhandenen Eingängen des Moduls gehören die folgenden:

- `f : 16 Bit langer std_logic_vector`
  Dieser Eingang beschreibt unser eigentliches Datenwort, welches wir in das Register schreiben möchten. Bei einem Lesevorgang ist das Datenwort zu vernachlässigen. Wichtig ist hierbei, dass es sich um ein 16-Bit langen std_logic_vector handeln muss.
- `ce_write & ce_read : std_logic`
  Wenn ce_write/ce_read gesetzt ist, wird der Schreib/Lesevorgang aktiviert. Ist ce_write/ce_read nicht gesetzt, wird das Modul und seine eigentliche Funktionalität nicht aktiviert.
  Darüber hinaus besitzt das Modul auch einen rst Eingang, der, wenn gesetzt, das Speichersignal auf undefined zurücksetzt. Außerdem ist dieses Modul getaktet, folglich besitzt es einen clk Eingang, der eine Periode von 20 ns besitzt. Diese Frequenz war der Aufgabenstellung zu entnehmen. Jedwede Art von Vorgang wird nur bei rising_edge(clk) ausgeführt.

Vorhandene Ausgänge und ihre Bedeutung

Zu den vorhandenen Ausgängen des Moduls gehören die folgenden:

- `q_out – 16 Bit langer std_logic_vector`

Dieser Ausgang gibt nach einem Lesevorgang die Inhalte des Speichersignals q_mem aus. Wenn der Schreibvorgang eingeleitet wird, gibt q_out das gespeicherte Datenwort f aus.

Funktionsweise – Schreiben

Der Schreibvorhang des QREG-Programm Moduls funktioniert wie folgt:

- Lege das zu speichernde Datenwort an den Eingang f.
- Setze ce_write = ‚1‘
- Nun wird das Datenwort in das q_mem Signal geschrieben.
- In q_out wird nun das zuvor gespeicherte Datenwort ausgegeben.
- Setze ce_write wieder auf 0.

Dieser Vorgang lässt sich so oft wiederholen wie man möchte. Möchte man aber das q_mem Signal komplett zurücksetzen, reicht es, einen Reset Vorgang einzuleiten.

Funktionsweise – Lesen

Der Lesevorgang des RAM-Programm Moduls wird wie folgt realisiert:

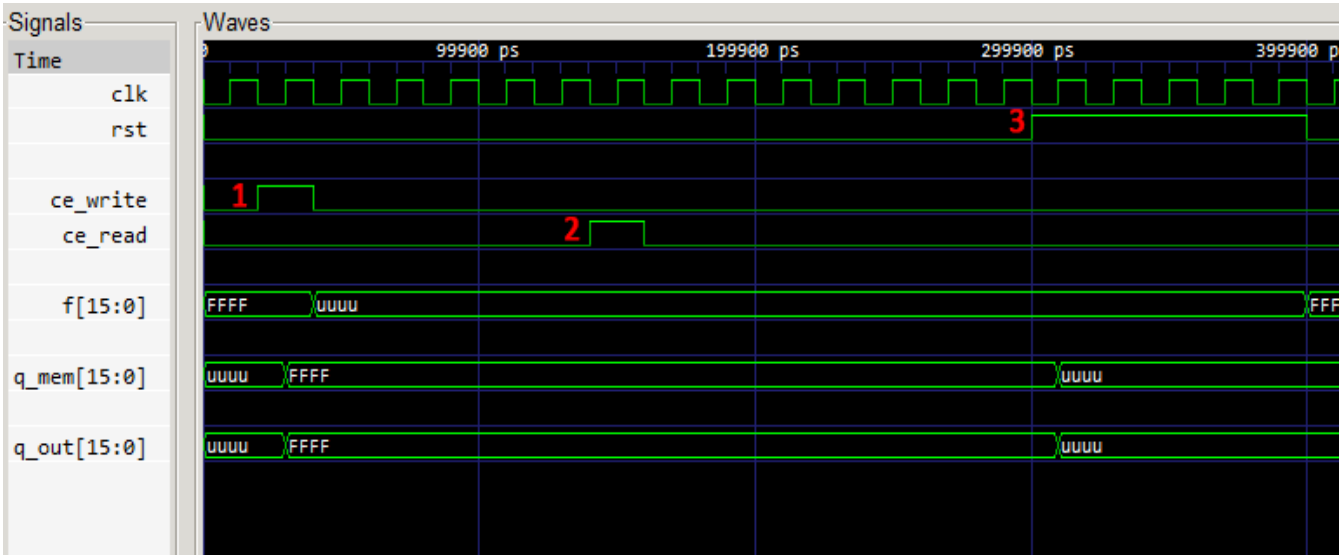
- ce_read auf 1 setzen.
- Die entsprechenden Datenworte werden aus unserem q_mem gelesen und auf q_out gelegt.
- ce_read auf 0 setzen

Was ist zu beachten?

- ce_write und ce_read können nicht gleichzeitig gesetzt werden, auch nicht direkt aufeinanderfolgend im nächsten Takt. Es muss ein Takt Pause dazwischen sein.
- Es ist darauf hinzuweisen, dass beim Lesevorgang tatsächlich etwas in dem q_mem des QREG steht. Ein Tipp wäre, das QREG erst mit mindestens einem Wert zu beschreiben bevor man einen Lesevorgang einleitet.
- Das CLK Signal ist momentan auf 20ns gesetzt, kann aber beliebig verändert werden.
- Wie immer gilt es, auf die Datentypen und –längen der einzelnen Eingänge zu achten, bevor man das Modul benutzt.

Beispiel für zwei Schreibvorgänge und einen Lesevorgang des QREG - Moduls

Hiermit wird eine typische Benutzung des QREG-Moduls simuliert.



1) ce_write ist auf 1 gesetzt, das Datenwort ist bereits mit FFFF initialisiert worden. Bei der nächsten rising_edge(clk) wird unser Datenwort FFFF in q_mem gespeichert. Nach dem vollzogenen Schreiben wird ce_write wieder auf 0 gesetzt.
2) Jetzt erfolgt das Lesen aus dem QREG. Nun wird q_mem auf q_out gelegt. Ist der Speichervorgang beendet, wird ce_read wieder auf 0 gesetzt.
3) Zum Schluss erfolgt ein synchroner Reset. Rst wird auf 1 gesetzt, q_mem wird wieder mit undefined beschrieben. Der Zyklus beginnt von neuem.

2.1 ALU – Operandenauswahl – Grobe Funktionsweise

Das ALU-Operandenauswahl-Programm realisiert die Operandenauswahl des AM2901. Hierbei wird die Logik und Auswahl der richtigen Operanden mithilfe Concurrent Statements und Bedingungsabfragen realisiert. Auch hierbei wird angestrebt, durch die Übersetzung in VHDL das Modul leichter verständlich und letztendlich synthetisierbar und in Schaltung umsetzbar zu machen.

Vorhandene Eingänge und ihre Bedeutung

Zu den vorhandenen Eingängen des Moduls gehören die folgenden:

- `i : 9 Bit langer std_logic_vector`

Dieser Eingang beschreibt unsere Instruktionswort. Hierbei sind die niederwertigsten 3 Bit am wichtigsten, da sie die Logik der Operandenauswahl beschreiben. Diese Bit werden anhand der unten angeführten Tabelle gesetzt.

I_0	I_1	I_2	r_out	s_out	Beschreibung
0	0	0	a	q	**a** wird auf **r_out** gelegt, **q** wird auf **s_out** gelegt
0	0	1	a	b	**a** wird auf **r_out** gelegt, **b** wird auf **s_out** gelegt
0	1	0	zero	q	**zero** wird auf **r_out** gelegt, **q** wird auf **s_out** gelegt
0	1	1	zero	b	**zero** wird auf **r_out** gelegt, **b** wird auf **s_out** gelegt
1	0	0	zero	a	**zero** wird auf **r_out** gelegt, **a** wird auf **s_out** gelegt
1	0	1	d	a	**d** wird auf **r_out** gelegt, **a** wird auf **s_out** gelegt
1	1	0	d	q	**d** wird auf **r_out** gelegt, **q** wird auf **s_out** gelegt
1	1	1	d	zero	**d** wird auf **r_out** gelegt, **zero** wird auf **s_out** gelegt

- `d,a,b,q,zero : 16 Bit langer std_logic_vector`
  Dies sind alles Eingänge der ALU-Quelloperandenauswahl, die der AM2901 besitzt und wir modellieren müssen.
- `ce: std_logic`
  Wenn ce gesetzt ist, wird das Modul aktiviert. Ist ce nicht gesetzt, wird das Modul und eine eigentliche Funktionalität nicht aktiviert.

Vorhandene Ausgänge und ihre Bedeutung

Zu den vorhandenen Ausgängen des Moduls gehören die folgenden:

- `r_out & s_out : 16 Bit langer std_logic_vector`
  Die Ausgänge der AM 2901 Alu-Quelloperandenauswahl. Diese werden weiter an die Arithmetisch-logische Einheit weitergeleitet.

Funktionsweise

Das ALU-Quelloperandenauswahl Modul funktioniert wie folgt:

- die niederwertigsten 0-2 Bit des Instruktionswortes i nach oben genannter Tabelle spezifizieren
- alle sonstigen Eingängen im Rahmen der Richtlinien beliebig initiieren
- setze ce = 1

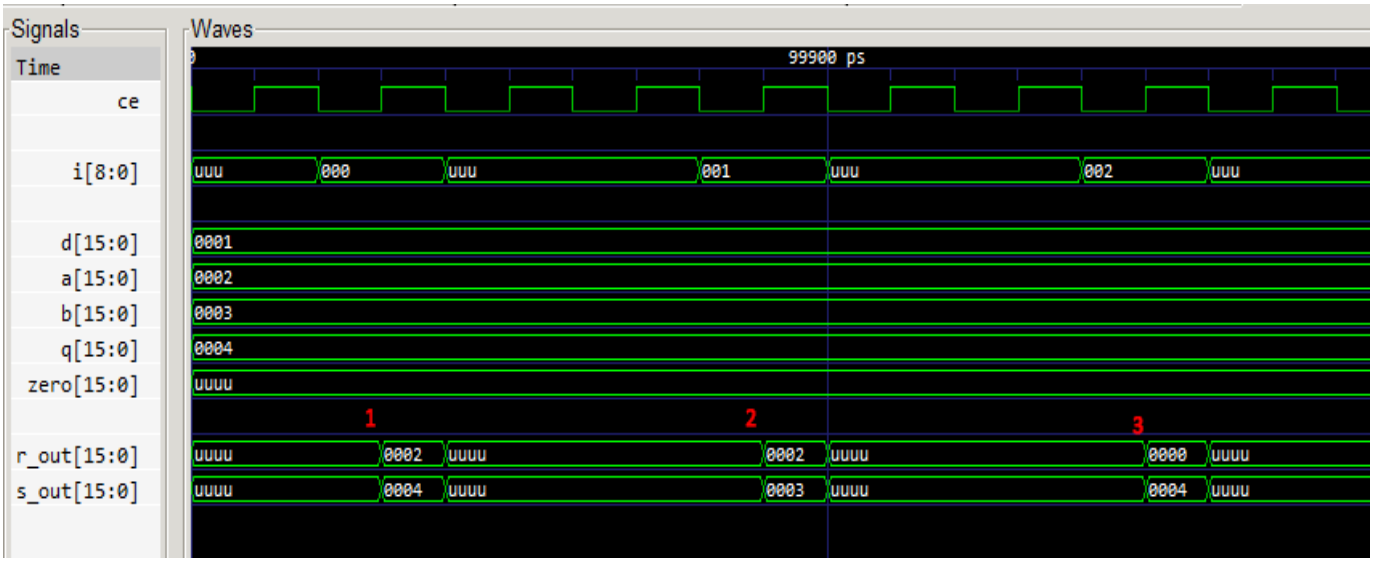
- r_out und s_out geben nun die entsprechenden korrekt ausgewählten zwei Eingänge aus.

Was ist zu beachten?

- Wie immer gilt es, auf die Datentypen und –längen der einzelnen Eingänge zu achten, bevor man das Modul benutzt.
- Der Zero Eingang wird in diesem Programm nur der Vollständigkeit wegen aufgeführt. Intern wird dies mit einer Zuweisung von 0 realisiert.

Beispiel für unterschiedliche Instruktionswörter des ALU-Operandenauswahl - Moduls

Hiermit wird eine typische Benutzung der ALU-Operandenauswahl simuliert. Hierbei ist zu beachten, dass aus Platzgründen nicht alle 8 Tests visualisiert sind.



1) Das Instruktionswort wurde mit 000 als den niederwertigsten Bits initialisiert. Zusätzlich wurden d,a,b,q als 1,2,3,4 initialisiert. Bei ce = 1 werden nun a auf r_out und q_auf s_out gelegt. Nach diesem Vorgang wird das Instruktionswort auf undefined gesetzt um eine lesbarere Simulation zu ermöglichen und die r_out und s_out Ausgänge mit undefined zu beschreiben.
2) Das Instruktionswort wurde nun mit 001 als den niederwertigsten Bits initialisiert. Bei ce = 1 wird nun a auf r_out und b auf q_out gelegt. Auch nach diesem Vorgang wird das Instruktionswort modifiziert um eine lesbarere Simulation zu ermöglichen.
3) Das Instruktionswort wurde nun mit 010 als den niederwertigsten Bits initialisiert. Bei ce = 1 wird nun 0 auf r_out und q auf s_out gelegt. Auch nach diesem Vorgang

wird das Instruktionswort modifiziert um eine lesbarere Simulation zu ermöglichen.

4) Analog dazu wären die restlichen 5 Testfälle.