TECHNISCHE UNIVERSITÄT CHEMNITZ 175 Jahre 1836-2011

Rechnerorganisation
Praktikum
Prof. Dr.-Ing. W. Rehm

Praktikum 2
Bearbeiter: Ralph Schlosser
Letzte Änderung: René Oertel, 25.11.2010

RECHNER ARCHITEKTUR

1 Ziel

Hier zunächst ein Überblick über die Ziele des 2. Praktikums:

- Sequentielle Systeme: Grundlagen und Realisierung in VHDL
- Bausteine von Digitalrechnern: Register
- Der Taktgenerator des Digilent Nexys 2
- Besonderheiten der Schalter des Digilent Nexys 2

2 Sequentielle Systeme: Grundlagen und Realisierung in VHDL

2.1 Grundlagen

Die bisher betrachteten kombinatorischen Systeme hatten die Eigenschaft, dass das Ausgangssignal y zu jedem Zeitpunkt nur vom Zustand der Eingangssignale x *zu diesem Zeitpunkt* abhängt.

Systeme, die eine Rückkopplung des Ausgangs auf die Eingänge besitzen, weisen ein anderes zeitliches Verhalten auf. Betrachtet man die Änderungen der Ein- und Ausgänge solcher Schaltungen, so ist eine Abhängigkeit des Ausgangssignales y vom aktuellen Zustand x und von *vorherigen* Zuständen x' der Eingänge erkennbar. Solche Systeme nennt man auch *sequentielle Systeme*. Die wichtigste Eigenschaft der sequentiellen Systeme ist ihre Fähigkeit, Informationen zu speichern.

2.2 Automaten

Wie schon aus der Digitaltechnik bekannt ist, bieten sich endliche Automaten (engl. *FSM – Finite State Machines*) zur Beschreibung von sequentiellen Systemen an. Da diese eine große Bedeutung haben, wollen wir nochmal kurz auf die theoretischen Grundlagen von endlichen Automaten und dann auf deren Realisierung als Verhaltensbeschreibung in VHDL eingehen.

Mathematisch ist ein endlicher Automat als 5-Tupel definiert:

$$A = (X, Y, Z, f, g)$$

Dabei ist X die Menge der Eingangsbelegungen, Y die Menge der Ausgangsbelegungen, Z die Menge der Zustände, f die Überführungsfunktion und g die Ergebnisfunktion. Die Mengen X, Y und Z sind jeweils Potenzmengen über $B = \{0, 1\}$. Die Überführungsfunktion f und die Ergebnisfunktion g sind bei einem *Mealy-Automaten* definiert als

$$f : Z \times X \rightarrow Z$$
$$g : Z \times X \rightarrow Y$$

Bei einem *Moore-Automaten* hingegen hängt das Ergebnis nicht von der Eingangsbelegung X ab. Bei diesem ist die Ergebnisfunktion demnach definiert als

$$g : Z \rightarrow Y$$

Neben den beiden Automatentypen Moore und Mealy klassifiziert man Automaten noch danach, ob sie taktgesteuert (*synchron*) oder ungetaktet (*asynchron*) arbeiten. Wir werden im weiteren Verlauf der Praktika ausschließlich mit getakteten Automaten arbeiten. Dass heißt, wenn Zustandsübergänge stattfinden, dann nur im Zuge einer steigenden oder fallenden Taktflanke (bei uns in der Regel mit der steigenden).

2.3 Realisierung in VHDL

Bei der Realisierung eines Automaten in VHDL ist es sinnvoll, für Zustandsüberführungs- und Ergebnisfunktion jeweils einen eigenen Prozess zu schreiben, der die entsprechende kombinatorische Logik enthält. Daneben erstellt man noch einen dritten Prozess, der die Speicherung des aktuellen Zustandes übernimmt. Dieser Prozess ist in der Regel an ein Taktsignal gebunden, welcher mit steigender oder fallender Taktflanke das Ergebnis der Zustandsüberführungsfunktion übernimmt. Defacto wird hierbei nichts anderes realisiert als eine Menge von D-Flipflops. Der Zustandsspeicherprozess enthält üblicherweise gleichzeitig auch noch ein (synchrones oder asynchrones) Reset, um den Automaten in einen definierten Ausgangszustand zu bringen. Diese Art der Realisierung nennt man auch *3-Prozessnotation*.

In der Praxis kann es allerdings aus Gründen der Übersichtlichkeit manchmal günstiger sein, diese strikte Aufteilung in drei Prozesse aufzuheben, und alle drei Funktionen in einem VHDL-Prozess unterzubringen. Der VHDL-Neuling sollte jedoch zu Beginn von der 3-Prozessnotation Gebrauch machen. Wir wollen nun diese Art der Notation eines endlichen Automaten in VHDL an einem einfachen Beispiel nachvollziehen.

Beispiel 1:

Es soll eine VHDL-Verhaltensbeschreibung eines (zugegebenermaßen eher theoretischen ;-) Bremssystems mit ABS für ein Auto erstellt werden. Dazu seien folgende Vereinbarungen getroffen:

- Das Bremssystem ist sehr primitiv. Es hat lediglich die Eingänge `bremsen` (Bremspedal gedrückt) und `blockiert` (Räder blockieren).
- Das ABS ist entweder aktiviert oder nicht aktiviert, d. h. es gibt keine Abstufung der Stärke der ABS-Wirkung.
- Der Automat erzeugt Ausgaben für den Bremskreislauf (`bremse_aktiv`) und für eine ABS-LED auf dem Armaturenbrett (`abs_aktiv`).
- Zustandsübergänge finden nur mit steigenden Taktflanken am Signal `clk` statt, und der Automat werde mit dem Signal `einschalten` unabhängig vom Taktsignal (also asynchron) in den Ausgangszustand versetzt.

Ausgehend von dieser Beschreibung könnte man den Automatengraphen in Abbildung 1 angeben. Aus Gründen der Übersichtlichkeit lässt man dabei den Takt sowie sämtliche Zustandsübergänge bei einem Reset (`einschalten`) weg.

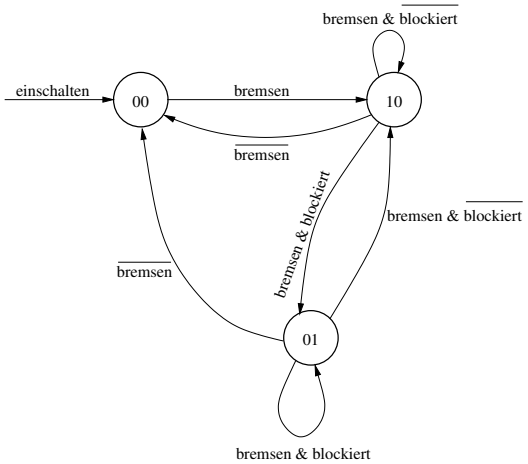

Abbildung 1: Automatengraph für Bremssystem mit ABS

Die Kodierung der Zustände des Automaten ist dabei so gewählt worden, dass die Zustandsbits selbst als Ausgabe herausgeführt werden können. Das heißt die Ergebnisfunktion besteht bei diesem Automaten aus einem „Stück Draht“. Dies ist eine spezielle Form eines Moore-Automaten, und wird auch als *Medvedev-Automat* bezeichnet. Bei uns steht dabei das linke Bit (MSB) für „Bremse aktiviert“ und das rechte Bit (LSB) für „ABS aktiviert“. Der VHDL-Quellcode des Bremssystems ist in Listing 1 dargestellt.

Listing 1: Bremssystem in VHDL

```
5  entity bremssystem is
6      port
7      (
8          bremsen, blockiert, clk, einschalten: in bit;
9          abs_led, bremse_aktiv: out bit
10     );
11 end;
12
13 architecture arch_brems of bremssystem is
14
15     -- Definition der 3 Zustaende
16     constant ZUSTAND1: bit_vector(1 downto 0) := "00";
17     constant ZUSTAND2: bit_vector(1 downto 0) := "10";
18     constant ZUSTAND3: bit_vector(1 downto 0) := "01";
19
20     signal akt_zustand, naechst_zustand: bit_vector(1 downto 0);
21
22 begin
23
24 f:  process(akt_zustand, bremsen, blockiert)        -- Ueberfuehrungsfunktion
25     begin
26         case akt_zustand is
27             when ZUSTAND1 =>
28                 if bremsen = '1' then
29                     naechst_zustand <= ZUSTAND2;
30                      else
31                     naechst_zustand <= ZUSTAND1;
32                 end if;
33             when ZUSTAND2 =>
34                 if bremsen = '0' then
35                     naechst_zustand <= ZUSTAND1;
36                 elsif bremsen = '1' and blockiert = '0' then
37                     naechst_zustand <= ZUSTAND2;
38                 elsif bremsen = '1' and blockiert = '1' then
39                     naechst_zustand <= ZUSTAND3;
40                 end if;
41             when ZUSTAND3 =>
42                 if bremsen = '0' then
43                     naechst_zustand <= ZUSTAND1;
44                 elsif bremsen = '1' and blockiert = '1' then
45                     naechst_zustand <= ZUSTAND3;
46                 elsif bremsen = '1' and blockiert = '0' then
47                     naechst_zustand <= ZUSTAND2;
48                 end if;
49             when others =>                           -- Reset (Anti-Latch)
50                     naechst_zustand <= ZUSTAND1;
51         end case;
52     end process f;
53
54 g:  process(akt_zustand)                            -- Ergebnisfunktion
55     begin
56         abs_led <= akt_zustand(0);
57         bremse_aktiv <= akt_zustand(1);
58     end process g;
59
60 zw: process(clk, einschalten)                       -- Taktung, Reset (asynchron)
61     begin
62         if einschalten = '1' then
63             akt_zustand <= ZUSTAND1;
64         elsif clk'event and clk = '1' then
65             akt_zustand <= naechst_zustand;
66         end if;
67     end process zw;
68
69 end architecture arch_brems;
```

Wesentlich beim Verständnis der Architekturbeschreibung ist das oben behandelte Automatenmodell. Im Prozess mit dem Namen `f` wird in Abhängigkeit des aktuellen Zustands `akt_zustand` die Überführung in den nächsten `naechst_zustand` festgelegt, aber noch nicht vollzogen. Dies geschieht durch eine `case`-Auswahl und weitere `if`-Abfragen. Damit werden die im Automatengraphen an den Pfeilen angegebenen Bedingungen überprüft sowie der entsprechende nächste Zustand ermittelt (`naechst_zustand`).

Die eigentliche Zustandsüberführung wird dann im Prozess `zw` mit jeder steigenden Taktflanke (`clk'event and clk = '1'`) vollzogen. Dabei ist aber auch asynchron (also unabhängig vom Takt) die Möglichkeit gegeben, den Automaten wieder in den Anfangszustand (`ZUSTAND1`) zu versetzen. Dies entspricht allgemein einem Reset des Automaten.

Ein wichtiger Teil der Prozesse, den sie sich unbedingt genau anschauen sollten, ist die Sensitivitätsliste die in Klammern direkt hinter der `process`-Anweisung steht. Hier wird festgelegt, auf welche Signale der Prozess reagiert. Wann immer sich eines der Signale auf der Sensitivitätsliste ändert, wird der Prozess neu ausgeführt. Wichtig ist dabei Folgendes: Für getaktete Prozesse ist dieser nur vom Takt abhängig, nicht aber von den anderen Eingabesignalen. Dies wird von der ISE Design Suite immer in eine sequentielle Schaltung umgesetzt, die also Register und Flipflops enthält. Wenn alle Eingangssignale auf der Sensitivitätsliste auftauchen, wird der Prozess als einfache kombinatorische Schaltung realisiert. Vergessen Sie für die Überführungs- und Ergebnisfunktion also nicht alle Eingangssignale auf die Sensitivitätsliste zu setzen, sonst wird der Compiler unnötige Speicher verwenden. Dies führt zu schlechterer Performance oder gar zu ungewollten Seiteneffekten.

Im Prozess `g` schließlich wird die Ausgabe des Automaten ermittelt. Da, wie oben schon angedeutet, die Ausgabe im Zustand des Automaten kodiert ist, wird keine weitere kombinatorische Logik zur Ermittlung der Ausgabe benötigt.

Bei Simulation des Designs erhält man eine Waveform-Ausgabe wie in Abbildung 2, die die Änderung der Signale während der Simulation darstellt.

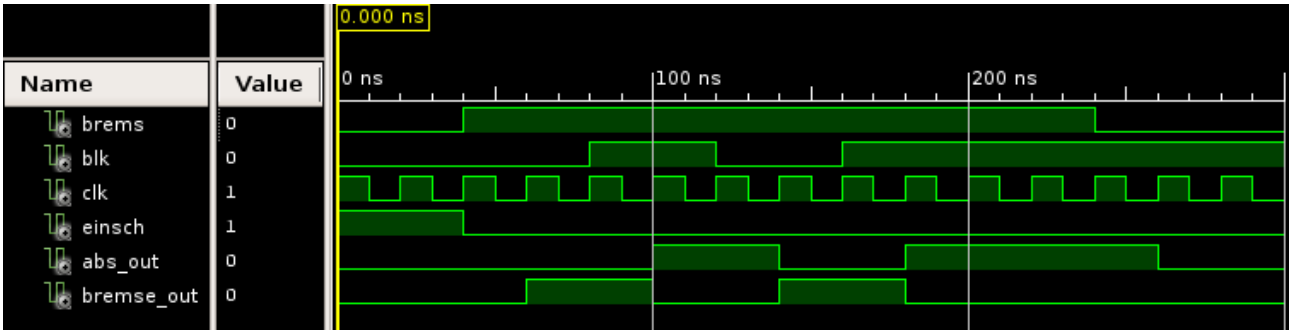

Abbildung 2: Simulationsergebnis des Bremssystems

Wie man sieht, werden der Reihe nach verschiedene Werte an die Eingänge `brems` (Bremspedal gedrückt), `blk` (Räder blockiert) und `einsch` (Fahrzeug einschalten) angelegt, um die Funktionsweise des Automaten zu testen. Die Ausgangssignale `abs_out` (ABS aktiv) und `bremse_out` (Bremswirkung aktiv) lassen den Schluss zu, dass der Automat korrekt arbeitet.

Sie können die Simulation mit der vorbereiteten Testbench selbst nachvollziehen. Fügen Sie dazu die Dateien `bremssystem.vhdl` und die Testbench `brems_tb.vhdl` aus dem Archiv mit den Vorgabedateien wie gewohnt einem neuen Projekt hinzu und starten Sie dann die Simulation mit ISim.

3 Ein wichtiger Baustein von Digitalrechnern: Register

Nachdem wir nun die Grundlagen von kombinatorischen und sequentiellen Systemen wiederholt und deren prinzipielle Realisierung in VHDL kennengelernt haben, wollen wir uns nun mit einem wichtigen Baustein von Digitalrechnern beschäftigen: Das Register.

Register gehören zu den grundlegenden Speichern, die meist in Mikroprozessoren oder Peripherie-Chips verwendet werden, um kleine Mengen an Information zu speichern. Prinzipiell sind solche Register nichts weiter als eine Aneinanderreihung von D-Flipflops.

Zunächst halten wir fest, welche Dinge ein Register besitzen bzw. können muss:

- Es gibt ein Ein- und Ausgangssignal (`reg_in`, `reg_out`), die eine gewisse Bit-Breite besitzen. Diese Bit-Breite ist natürlich am Eingang und am Ausgang gleich.
- Das Register soll rücksetzbar sein, d. h. wir möchten seinen Inhalt initialisieren können (z. B. asynchrones Resetsignal `reset`).
- Zur Synchronisation führen wir einen Takt (`clk`) an das Register und verlangen, dass das Register mit steigender Taktflanke den am Eingang anliegenden Wert übernimmt, wenn ...
- ... zusätzlich das Schreiben in das Register erlaubt ist. Dazu benutzen wir ein Enable-Signal (`reg_en`), das an das Register herangeführt wird.

In Abbildung 3 sind noch einmal die Schnittstellen des eben spezifizierten Registers zu sehen.

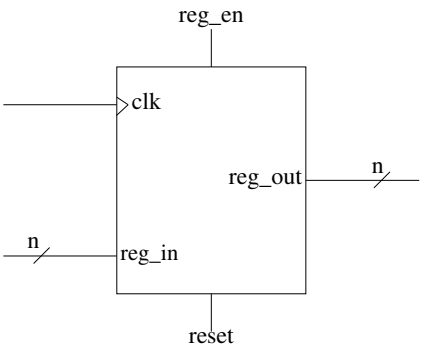

Abbildung 3: Schnittstellen eines Registers

Beispiel 2:

Als Beispiel sei in Listing 2 eine VHDL-Verhaltensbeschreibung eines Registers der Breite 2 Bit angegeben, das die obige Spezifikation erfüllt.

Listing 2: Verhaltensbeschreibung eines 2 Bit Registers

```
1  library ieee;
2  use ieee.std_logic_1164.all;
3  use ieee.std_logic_unsigned.all;
4
5  entity register_2bit is
6      port
7      (
8          clk:     in  std_logic;
9          reset:   in  std_logic;
10         reg_en:  in  std_logic;
11         reg_in:  in  std_logic_vector(1 downto 0);
12         reg_out: out std_logic_vector(1 downto 0)
13      );
14 end register_2bit;
15
16 architecture behave of register_2bit is
17 begin
18 P_reg: process(clk, reset)
19        begin
20            if reset = '1' then
21                reg_out <= "00";
22            elsif clk'event and clk = '1' then
23                if reg_en = '1' then
24                    reg_out <= reg_in;
25                end if;
26            end if;
27        end process P_reg;
28 end behave;
```

Wir wollen nun anhand eines Impulsdiagramms (Abbildung 4) den Signalverlauf der Signale `clk`, `reg_in` und des Registerzustandes `reg_out` der obigen Architektur noch einmal genauer nachvollziehen. Zur Vereinfachung sei angenommen, dass `reg_en` immer auf `1` gesetzt ist und außerdem keine Verzögerungszeiten auftreten, welche in realen technischen Systemen immer vorhanden sind.

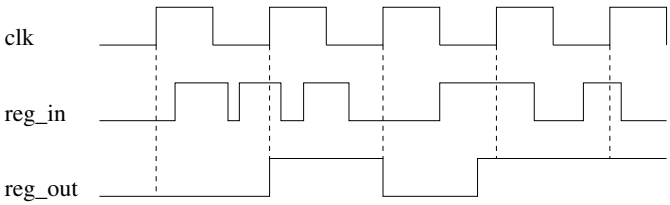

Abbildung 4: Impulsdiagramm (vereinfacht) für das Register

Aus Abbildung 4 wird also ersichtlich, dass Änderungen an `reg_in` nur übernommen werden, wenn eine steigende Taktflanke (d. h. `clk'event and clk = '1'`) an `clk` anliegt. Diese Stellen sind in der Abbildung mit einer gestrichelten Linie markiert worden.

4 Der Taktgenerator des Nexys 2 Boards

Bei den eben behandelten Designs wird jeweils eine Taktquelle benötigt, um gewisse Schaltvorgänge bzw. Zustandswechsel auszulösen. Nun stellt sich die Frage, von wo diese Taktquelle genommen werden soll.

Für diesen Zweck verfügt das Nexys 2 Board über einen eigenen Taktgenerator. Dieser Taktgenerator erzeugt ein Taktsignal mit einer Frequenz von 50 MHz. D. h. dieses Signal ändert seinen Zustand von `0` zu `1` und zurück 50.000.000 mal pro Sekunde. Dieses Taktsignal wird dem Spartan-3E über Pin B8 zugeführt.

Werden andere (z. B. kleinere) Taktfrequenzen als die bereitgestellte benötigt, so muss über einen geeigneten Frequenzteiler der Ausgangstakt auf die gewünschte Frequenz heruntergeteilt werden. Zur Erzeugung höherer Frequenzen werden die DCMs (Digital Clock Manager) eingesetzt, was wir hier aber nicht benötigen werden. Diese Bauteile sind auch speziell dafür ausgelegt, sehr „saubere“ Taktsignale, das heißt mit wenig Störungen und steilen Flanken zu generieren und das mit beliebigen Taktfrequenzen.

5 Besonderheiten der Schalter des Nexys 2 Boards

5.1 Die Problematik

Die auf dem Nexys 2 vorhandenen Schalter (Schiebeschalter, Taster) sind nicht entprellt. Das bedeutet, dass bedingt durch die mechanische Konstruktion der Schalter bei einem Schaltvorgang das Signal am Ausgang noch einige Zeit zwischen den beiden Pegeln für Schalterstand 1 (P1) und Schalterstand 2 (P2) schwingt. Abbildung 5 soll diesen Sachverhalt am Beispiel eines Push-Buttons verdeutlichen. Für viele Anwendungen hat dieses Verhalten sehr schlechte Auswirkungen. Man denke zum Beispiel an einen Zähler, der mit jedem Tastendruck um 1 weiterzählen soll. Dieser würde dann bei einem Tastendruck nicht nur einmal, sondern je nach Verhalten sehr viel öfter weiterzählen.

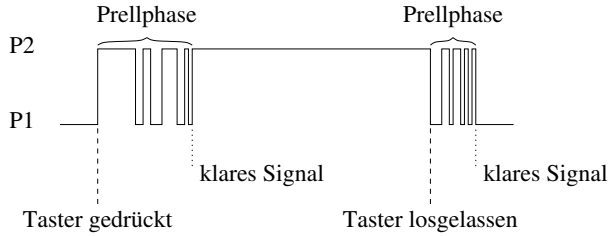

Abbildung 5: Prellphasen beim Schaltvorgang

Aus diesem Grund muss durch entsprechende Hardware dafür gesorgt werden, dass dieses „Zittern“ des Signales herausgefiltert wird.

Wir wollen hier noch einen Schritt weitergehen. Und zwar soll durch das Drücken eines Tasters ein entsprechendes Signal lediglich für eine Taktperiode lang eine `1` liefern. Eine solche Funktionalität werden wir später oft benötigen. Dies bietet sich z. B. dann an, wenn Einzelschrittfunktionalitäten gefordert sind. Wie etwa bei dem oben angesprochenen Zähler, der bei jedem Tastendruck um genau einen Schritt hochzählen soll.

5.2 Eine Lösungsvariante

Die Vorgabedatei `single_step.vhd` enthält eine fertige Lösung. Zur Überbrückung der kritischen Zeit (also unmittelbar nach dem ersten festgestellten Umschaltvorgang) wird ein Zähler verwendet. Dieser wir einmal bis 262143 gezählt, bevor das Signal des Schalters erneut ausgewertet wird. Bei 50 MHz Taktfrequenz entspricht dies einer Zeit von ca. 5,2 ms.

Hinweis: Eine Zusatzaufgabe wäre die experimentelle Bestimmung einer minimalen Wartezeit, bei der der Mechanismus noch zuverlässig arbeitet.

Abbildung 6 gibt einen schematischen Überblick über die Verwendung des Modules.

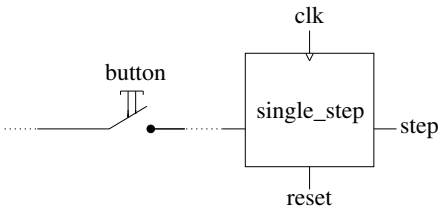

Abbildung 6: Struktur der Single-Step-Logik

Neben dem Signal des Schalters muss natürlich noch der Takt `clk`, sowie ein Reset-Signal `reset` herangeführt werden, welches zur Initialisierung des dort enthaltenen Automaten verwendet wird.

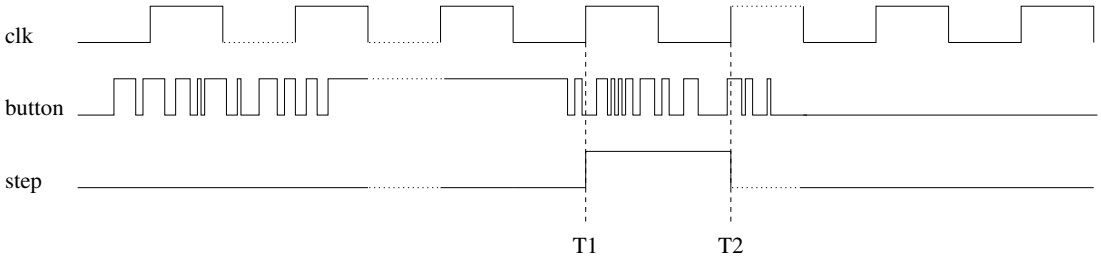

Abbildung 7: Beispiel für Impulsdiagramm der Single Step-Logik

In Abbildung 7 ist beispielhaft der Signalverlauf für einen Schaltvorgang mit Prellphasen aufgezeichnet. Wie man sieht, ist das Ausgangssignal `step` der Single-Step-Logik nur für eine Taktphase T1 bis T2 auf `1`, wenn der Taster wieder losgelassen wurde. Die gepunkteten Linien stellen dabei längere Zeiträume dar, in denen sich (außer dem Takt) nichts ändert – viele tausend Takte, die nicht auf das Blatt passen würden :-)

Hinweis: Möchte man das Verhalten des Single-Step-Modules simulieren, so sollte man die Entprellzeit drastisch reduzieren (Simulation dauert sonst etwas lange...).

6 Versuchsvorbereitung

Hinweis:
In Vorbereitung auf den Versuch ist es empfehlenswert, dass die folgenden Vorarbeiten von jeder Versuchsgruppe durchgeführt und die schriftlich ausgearbeiteten stichpunktartigen Antworten zu diesen Fragen und ggf. vorbereitete VHDL-Dateien zur praktischen Übung mitgebracht werden!

1. Worin unterscheiden sich bei einer 3-Prozessnotation die VHDL-Realisierungen eines Mealy- und eines Moore-

Automaten? Es geht dabei nicht um die Funktionalität der Automaten, sondern um einen grundlegenden Unterschied in der Notation (besonderes Augenmerk ist hier auf die Ergebnisfunktion zu legen)!

2. In Beispiel 1 wurde die Ergebnisfunktion im Zustand des Automaten kodiert. Was sind dabei Vorteile, was evtl. Nachteile?

3. In Abschnitt 3 wurde eine VHDL-Beschreibung eines 2 Bit Registers angegeben. Wie müsste diese Beschreibung abgewandelt werden, wenn das Register mit der fallenden Taktflanke getriggert werden soll und das Reset nur synchron (d. h. mit dem Takt) erfolgen kann?

4. Analysieren Sie die Vorgabedatei `single_step.vhd`! Zeichnen Sie den Automatengraphen des darin enthaltenen Automaten auf! Notieren Sie an die Kanten die notwendigen Bedingungen für einen Zustandsübergang und die jeweilige Ausgabe des Automaten. Ziehen Sie zur Hilfe auch Abbildung 7 heran, und zeichnen Sie ein, wann sich der Automat in welchem Zustand befindet.

7 Aufgaben

1. Es soll ein elektronischer Würfel als VHDL-Zustandsautomat entworfen werden. Dazu ist folgender Automatengraph gegeben (Abbildung 8).

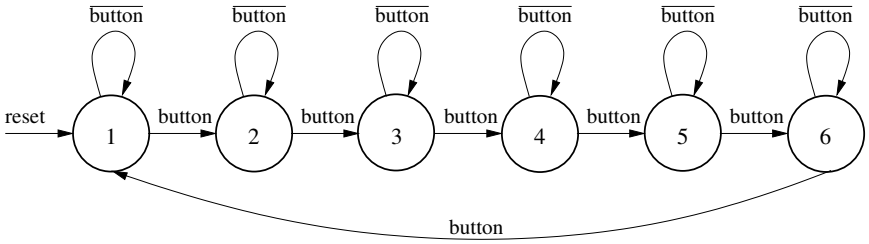

Abbildung 8: Würfel

Das Signal `button` sei ein Signal von einem Taster. Wie aus Abbildung 8 ersichtlich, wechselt der Automat mit jedem Takt den aktuellen Zustand, wenn der Taster gedrückt ist. Sobald man allerdings den Taster loslässt, soll der Automat im aktuellen Zustand verharren und ihn erst wieder verlassen, wenn der Taster wieder gedrückt wird. Der aktuelle Würfelzustand soll dezimal in einer Siebensegmentanzeige, z. B. *AN0* angezeigt werden.

Die Kodierung der Zustände kann frei gewählt werden. Durch ein asynchrones Reset-Signal `reset` soll der Würfel wieder in den Anfangszustand wechseln (diese Zustandsüberführung ist im obigen Automatengraphen aus Gründen der Übersichtlichkeit nicht mit angegeben).

Das Reset kann z. B. unter Verwendung eines weiteren Tasters realisiert werden. Als externe Taktquelle soll der in Abschnitt 4 angesprochene Taktgenerator benutzt werden.

Hinweis: Da es sich hierbei ohnehin um eine Art Zufallsgenerator handelt, ist es nicht unbedingt notwendig das Eingangssignal `button` zu entprellen. Aus technischen Gründen sollte das Signal vom Schalter dennoch mit Hilfe eines einfachen D-Flipflops mit dem Takt synchronisiert werden bevor es dem Würfel-Automaten zugeführt wird. D. h. man legt das Signal des Tasters an den D-Eingang eines D-Flipflops und verwendet den Ausgang des Flipflops anstatt das direkte Signal. Tut man dies nicht, so wird man unter Umständen ein merkwürdiges Verhalten feststellen. Eine detaillierte Begründung dieser Maßnahme würde hier zu weit führen und soll uns nicht weiter interessieren.

2. In Abschnitt 3 wurde eine Verhaltensbeschreibung eines 2 Bit Registers angegeben. Entwerfen Sie ausgehend von dieser Variante ein 4 Bit Register mit den in Abbildung 3 angegebenen Eingängen. Über die Schiebeschalter soll ein entsprechender 4 Bit Vektor voreingestellt werden können, der durch einen Druck auf einen Taster in das Register übernommen wird. Der Inhalt des Registers soll zur Kontrolle auf geeigneten LEDs angezeigt werden. Verwenden Sie einen zweiten Taster, um die asynchrone Rücksetzfunktion des Registers auszulösen. Dabei soll der aktuelle Registerinhalt mit `0000` überschrieben werden.

Hinweis: Verwenden Sie das vorgegebene Single-Step-Modul, um das Signal zur Datenübernahme zu erzeugen. Integrieren Sie dazu das Modul im Sinne einer Strukturbeschreibung!

Überprüfen Sie bei einem Funktionstest auch, ob das Register gemäß der Single-Step-Funktionalität erst beim Loslassen des Tasters den neuen Wert übernimmt.

3. *Zusatzaufgabe:* In der vorherigen Übung wurde als Zusatzaufgabe ein Encoder für die Siebensegmentanzeige angefertigt. Mit dem Wissen aus dieser Übung können wir die Siebensegmentanzeige nun komplett in Betrieb nehmen. Schreiben Sie ein Modul, welches folgende Funktionalität bereitstellt: Es sollen 4 Eingangsvektoren von jeweils 8 Bit Breite auf die Siebensegmentanzeige gelegt werden. Jeder Eingangsvektor enthält jeweils die 7 Segmente und den Dezimalpunkt einer Anzeige. Durch *AN0* bis *AN3* werden die einzelnen Anzeigen ausgewählt. Sie müssen also der Reihe nach alle 4 Anzeigen mit dem entsprechenden Eingangsvektor belegen. Die Anzeigen sollten zwischen 1 und 16 ms angesteuert werden, bevor zur nächsten gewechselt wird. Verwenden Sie z. B. einen Taktteiler um dies zu erreichen. Ist die Geschwindigkeit des Wechsels zu langsam, so flackert die Anzeige, ist sie zu schnell, wird kein lesbares Bild erzeugt. Binden Sie das Modul im Sinne einer Strukturbeschreibung ein und testen Sie es. Sie können den Encoder aus der vorherigen Übung benutzen, um 4-Bit-Vektoren direkt als Hexadezimalzahl anzuzeigen. Testen Sie auch dies.

4. *Zusatzaufgabe:* Wir wollen jetzt betrügen! Sei X das Ereignis, dass eine bestimmte Zahl gewürfelt wird. Normalerweise gilt bei einem Würfel für die Wahrscheinlichkeit eine Zahl i zu würfeln:

$$P(X = i) = \frac{1}{6} \quad \forall i = 1, ..., 6$$

   Wie könnte nun das Design des oben zu entwerfenden Würfels abgeändert werden, so dass ein Betrug möglich ist? Es soll also nach Möglichkeiten gesucht werden, damit die Gleichverteilung der Ereignisse (Zahl i wird gewürfelt) nicht mehr gegeben ist, also dass z. B. eine 6 häufiger vorkommt als andere Zahlen.
   (In der Realität müsste man dies dann anpassen – je nachdem welches Spiel und wer an der Reihe ist ;-)

   Implementieren und testen Sie Ihr neues Design!

5. *Zusatzaufgabe:* Das Eingangsbeispiel des ABS lässt sich auch wesentlich einfacher realisieren, wenn wir eine unendlich hohe Taktfrequenz annehmen.
   Wie könnte hier eine andere, einfachere Realisierung bei gleicher Funktionalität aussehen?

Hinweise, Berichtigungen und Kritik zu den Übungsunterlagen bitte an:

- René Oertel <rene.oertel@cs.tu-chemnitz.de>

Literatur und wichtige Links

[1] *The VHDL Cookbook, First Edition*
Peter J. Ashenden
http://www.vhdl.org/misc/VHDL-Cookbook.ps.tar.Z
http://tams-www.informatik.uni-hamburg.de/vhdl/doc/cookbook/VHDL-Cookbook.pdf

[2] *Schaltungsdesign mit VHDL*
Gunther Lehmann, Bernhard Wunder, Manfred Selz
Eine ausführliche Einführung in VHDL und nebenbei ein gutes Referenzhandbuch. Ein „must-have“ ;-) Das Buch, bestehend aus mehreren .pdf-Dateien und Beispielen, ist kostenlos im Internet verfügbar:
http://www.itiv.kit.edu/653.php

25. November 2010