ulm university universität
uulm

Universität Ulm
Institut für Eingebettete Systeme und Echtzeitsysteme

Übungen Architektur Eingebetteter Systeme

Dipl.-Ing. Tobias Bund

SS 2010

Inhaltsverzeichnis

Kapitel 1

Ziel der Übungen

Begleitend zur Vorlesung „Architektur Eingebetteter Systeme“ werden in den Übungen die Grundlagen des Hardwaredesign praktisch vermittelt. Dabei sollen auch die Unterschiede zwischen einer Hardwarebeschreibungssprache und der bekannten Softwareprogrammierung herausgestellt werden. Beide Implementierungsmöglichkeiten bieten bestimmte Vor- und Nachteile. Für viele Probleme in eingebetteten Systemen bietet sich eine Hardware/Software Co-Design Lösung an. Dabei entsteht die Möglichkeit bestimmte (oft rechenintensive) Komponenten in Hardware auszulagern ohne dabei auf die Vorteile einer Softwareimplementierung verzichten zu müssen.

Um dieses anschaulicher zu machen, soll im Laufe der Übungen ein zyclische Redundanzprüfung und -generierung (engl. cyclic redundancy check - CRC) realisiert werden. Zuerst soll, nach einer grundlegenden Einführung in die Hardwarebeschreibungsprache, ein CRC als reine Hardwareimplementierung realisiert werden. Nach dem Erstellen eines so genannten Softcoreprozessors soll dann ein CRC als reine Softwareimplementierung realisiert werden. Da das Generieren und Prüfen von CRC-Werten meist im Zusammenhang mit der Übertragung oder Speicherung von Daten zusammenfällt, ist es wünschenswert die Hardwarekomponente auch vom Softcoreprozessor aus zu benutzen. Als Abschluss der Übungen ist daher ein Hardware/Software Co-Design eines CRC-Wert Generators und Prüfers für einen Datenblock zu realisieren.

Innerhalb der Übungen wird zur Umsetzung auf Altera gesetzt. Als Hardwareplattform wird das DE2-Board (Development and Education) von Altera verwendet. Als Kern enthält dieses den FPGA (Field Programmanle Gate Array) Chip “Cyclone

II". Dieser bietet ausreichend viele Logikeinheiten um alle geplanten Vorhaben zu realisieren. Als Hardwarebeschreibungssprache wird VHDL (Very High Speed Integrated Circuit Hardware Description Language) verwendet. Als Implementierungsplattform für VHDL wird das Altera Tool Quartus2 verwendet. Dieses ermöglicht auch die direkte Programmierung des DE2-Boards über den USB-Blaster. Weiterhin kann innerhalb des Tools mit dem sogenannten SOPC-Builder ein Softcoreprozessor (NIOS2) gebaut werden. Dieser wird aus den Logikbausteinen des FPGAs zusammengesetzt. Um diesen nun zu programmieren wird das Altera Tool NIOS2 verwendet, welches auf der Eclipse Plattform basiert. Als Programmiersprache wird C verwendet.

Die Übungen sind in folgende Kapitel aufgegliedert:

1. Einführung in VHDL
2. Das erste Projekt in Quartus
3. Grundlagen VHDL 1
4. Grundlagen VHDL 2
5. Grundlagen VHDL 3
6. Implementierung eines CRC-Prüfers in Hardware
7. Einbindung des Hardware CRC-Prüfers als Prozessorkomponente
8. Erstellen eine NiosII Softcore-Prozessors
9. Anbindung von SDRAM-Speihers an den NiosII
10. Implemeniteren eines Software CRC-Prüfers
11. Anbinden des Hardware CRC-Prüfers an den NiosII

Für die grundlegenden Einführungen werden jeweils Teile des Altera Tutorials genutzt. Die Tutorials finden Sie auf dem Netzlaufwerk unter `\\whitehouse\Groups\Lehrveranstaltungen\aes10`.

Weiterhin können die Tutorials auch direkt von der Altera Seite heruntergeladen werden: `http://www.altera.com`. Dort finden Sie auch Literatur und Dokumentationen über Quartus und NIOS.

Kapitel 2

Einführung in VHDL

Wie bereits in der Einleitung erwähnt ist VHDL eine Hardwarebeschreibungssprache, die sich im Gegensatz zu Softwaresprachen dadurch auszeichnet, dass Abarbeitungen paralell ablaufen können. Dieses beutetet wiederum, dass eine sequentielle Beschreibung, wie sie in den meisten Software-Programmiersprachen stattfindet, nicht direkt in VHDL umgesetzt werden kann. Hierzu muss selbst eine Taktung eingeführt werden. Wie dieses im einzelnen aussieht, hängt von der Situation ab. Typische Konzepte sind die State-Machine oder die Benutzung der Clock in Zusammenhang mit einem Counter. Als Grundregel können Sie zuerst einmal davon ausgehen, dass alle innerhalb eines Prozesses beschriebenen Aktionen parallel ausgeführt werden und die Belegung der Signale sowie Ein- und Ausgänge erst am Ende des Prozesses stattfindet. Somit ist folgende Beschreibung in VHDL möglich:

```
1 ARCHITECTURE Tauschen OF Beispiel IS
2 SIGNAL a,b: STD_LOGIC;
3 BEGIN
4     PROCESS(a,b)
5     BEGIN
6         a <= b;
7         b <= a;
8     END PROCESS;
9 END Tauschen;
```

VHDL bietet die Möglichkeit sowohl eine strukturelle Beschreibung als auch eine Verhaltensbeschreibung eines Bausteins zu implementieren, um dessen Ein- und Ausgabeverhalten zu spezifizieren. Die strukturelle Beschreibung eines Bausteines gibt an, wie viele Instanzen der einzelnen Komponenten zur Realisierung dieses Bausteines benutzt werden, wie diese Komponenten untereinander verdrahtet sind

und wie diese mit den Ein- und Ausgängen des Bausteines verdrahtet sind. Die Verhaltensbeschreibung dagegen beschreibt einen Baustein nicht durch seine Komponenten sondern durch gleichzeitig nebeneinander laufende Prozesse, welche die Belegung der Signale des Bausteines funktional bestimmen. Prozesse beschreiben damit wie sich Signale in Abhängigkeit anderer Signale verhalten.

Bevor Sie im nächsten Kapitel beginnen sich mit dem DE2-Board und Quartus2 zu beschäftigen, sollen im Folgenden einige Grundlegenden Konzepte von VHDL erklärt werden. Durch die weite Verbreitung von VHDL finden Sie auch ausreichend Literatur in der Bibliothek oder online.

Zu jeder (simulierbaren und synthetisierbaren) VHDL Beschreibung exisitiert eine sogenannte Top-Level-Entity. Das Hauptmerktmal dieser Komponente besteht darin, dass diese den Projektnamen trägt und dass alle nach externen geführten Signale aufgelistet sind. In unserem Fall sind dies die PINs des FPGA.

2.1 Bibliotheken

Jeder Baustein beginnt mit dem Einbinden der Bibliotheken (Libraries). Folgende zwei Zeilen werden am Anfang von allen Ihren Bausteinen stehen:

```
1 LIBRARY ieee;
2 USE ieee.std_logic_1164.all;
```

Der Datentyp **std_logic** ist eine nach dem IEEE Stadard 1164-1993 definiert 9-wertige Logik. Dabei können Signale folgende Werte annehmen:

'U' *nicht inizialisiert*
'X' *undefiniert*
'0' *starke logische Null*
'1' *starke logische Eins*
'Z' *Hochohmig*
'W' *schwach Unbekannt*
'L' *schwach logische Null*
'H' *schwach logische Eins*
'-' *nicht beachten (don't care)*

Im Gegensatz zu dem Datentyp **bit**, der nur die Werte ’0’ und ’1’ annehmen kann, hat **std_logic** den Vorteil alle Zustände die auch auf der Zielplattform auftreten können zu beschreiben. Innerhalb der Übungen werden Sie jedoch fast ausschließlich mit starken logischen Nullen und Einsen (’0’ bzw. ’1’) arbeiten. Jedoch sollten sie trotzdem alle Signale immer als **std_logic** Datentypen deklarieren, da innerhalb des Syntheseprozesses von Quartus2 auch andere Zustände der Signale betrachtet werden können, beispielsweise wenn Ihr Signal noch nicht initialisiert wurde (’U’) oder wenn der Zustand undefiniert (’X’) ist.

Sobald sie elementare Rechenoperationen benötigen (wie beispielsweise Plus ’+’ oder Minus ’-’) müssen Sie zusätzlich aus der **ieee** Bibliothek folgendes benutzen:

```
1 USE ieee.std_logic_unsigned.all;
```

2.2 Schnittstellenspezifikation

Die Schnittstellen eines Bausteines werden über die Schnittstellenspezifikation (**entity** Konstrukt) beschrieben. Diese stellt die externe Sicht der Komponente dar. Den Namen des Bausteines können Sie selbst bestimmen (natürlich sollten keine reservierten Worte, Zahlen, Umlaute, Leerzeichen ... verwendet werden). Jedoch muss Ihr oberster Baustein den Namen des Projektes tragen. Beispielhaft sieht dieses folgendermassen aus:

```
1 ENTITY baustein IS
2     PORT (
3         portname : INOUT    datentyp
4         );
5 END baustein;
```

```
1 ENTITY most_used IS
2     PORT (
3         CLOCK_50      :   IN  STD_LOGIC;
4         KEY       :   IN  STD_LOGIC_VECOTR(3 downto 0);
5         SW        :   IN  STD_LOGIC_VECTOR(17 downto 0);
6         LEDG          :   OUT STD_LOGIC_VECTOR(7 downto 0);
7         LEDR          :   OUT STD_LOGIC_VECTOR(17 downto 0)
8     );
9 END most_used;
```

Für die Bezeichnungen der externen Schnittstellen sollten Sie auf die im Handbuch angegebenen Bezeichnungen zurückgreifen. Dieses macht Ihnen die Arbeit leichter, da die jeweiligen Hardwarekomponenten auf dem Board auch mit den gleichen Bezeichnungen beschriftet sind. Die in diesen Übungen am Meisten benutzten Ein- und Ausgaben sind im Beispiel *most_used* aufgelistet. Hierzu kommen noch die PINs welche Sie für den Softcoreprozessor benötigen werden. Diese sind dann im jeweiligen Tutorial angegeben.

Technisch gesehen sind die PINs des FPGA direkt über Leiterbahnen mit den jeweiligen elektronischen Bauteilen verbunden. Jedoch erscheint es sehr umständlich die PINs als Schnittstellen anzugeben. Daher ist es wesentlich einfacher, den PINs Namen und Vektoren zuzuordnen, so dass beispielsweise alle grünen LEDs einen gemeinsamen Vektor LEDG haben. Dieser ist nun 8-bit Breit und läuft damit von 7 bis 0.

Um Ihnen die Arbeit abzunehmen, gibt es von Altera schon die Datei, DE2pin-assignments.csv welches die Zuordnung zwischen den FPGA PINs und den auf dem Board angegebenen Namen enthält. Im Tutorial des nächstes Kapitels werden Sie lernen, wie diese Datei eingebunden wird.

Zusätzlich soll noch darauf hingewiesen werden, dass alle nicht verwendeten PINs auf tri-state gesetzt werden sollen. Dieses verhindert, dass ungewollt Spannungspegel an Bauteile und PINs gelegt werden, die nicht verwendet werden.

Vor der Programmierung des FPGAs sollten alle offenen Pins noch auf Tri-state gesetzt werden. Unter `Assignments` → `Settings` finden Sie unter der Kategorie `Device` einen Button `Device and Pin Options...`. Unter der Registerkarte `Unused Pins` können sie nun alle unbenutzten Pins auf `As input tri-stated` setzten.

2.3 Beschreibung

Nachdem Sie Ihre Biliotheken eingebunden haben und die Schnittstellen definiert haben, können Sie die Komponente durch sein Verhalten und/oder Struktur beschreiben. Der grundsätzliche Aufbau sieht folgendermassen:

```
1 ARCHITECTURE behavior OF baustein IS
```

```
-- Hier k nnen Signale definiert werden
BEGIN
--Code
END behavior;
```

Sie sollten sich selbständig über die Unterschiede zwischen einer Verhaltens- und Strukturbeschreibung informieren. Im folgenden finden Sie einige oft verwendete Routinen. Es soll noch auf folgendes hingewiesen werden: Sie können Sich innerhalb der Komponente Hilfssignale definieren. Diese sollten auch für die Schnittstellensignale verwendet werden. Sollten sich die Signalnamen der Komponente ändern (beispeilsweise weil sie andere LEDs benutzen oder reservierte Namen verwendet haben) müssen sie nur die Verdrahtung am Anfang Ihrer Komponente ändern und nicht alle Namen innerhalb der Komponente ersetzten. Sieses könnte beispielhaft für die Clock so aussehen:

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;
USE ieee.std_logic_unsigned.all;

ENTITY signaluse IS
    PORT (
        CLOCK_50    : IN    STD_LOGIC;
    );
END singaluse;

ARCHITECTURE behavior OF signaluse IS

    SIGNAL clk      : STD_LOGIC_VECTOR;

    BEGIN
        clk <= CLOCK_50;

    PROCESS (clk)
        -- ...
    END PROCESS;

END behavior;
```

2.4 Hilfreiche Routinen

- Bedingungen

```
1  ARCHITECTURE behavior OF Bedingung IS
2  SIGNAL a: STD_LOGIC_VECTOR (1 DOWNTO 0);
3  SIGNAL b: STD_LOGIC_VECTOR (2 DOWNTO 0);
4  BEGIN
5      PROCESS(a,b)
6      BEGIN
7          IF (a="01") THEN
8              b<="001";
9          ELSIF (a="11") THEN
10             b<="000";
11         ELSE
12             b<="111"';
13         END IF;
14     END PROCESS;
15 END behavior;
```

- Fallunterscheidung

```
1  ARCHITECTURE behavior OF Fallunterscheidung IS
2  SIGNAL a,b: STD_LOGIC_VECTOR (2 DOWNTO 0);
3  BEGIN
4      PROCESS(a,b)
5      BEGIN
6          CASE a IS
7              WHEN "001" => b <= "000";
8              WHEN "011" | "110" => b <= "100";
9              WHEN OTHERS => b <= "111";
10         END CASE;
11     END PROCESS;
12 END behavior;
```

- Bedingung mit „WITH ... SELECT“

```
1  ARCHITECTURE structure OF Verzweigung IS
2  SIGNAL a              :STD_LOGIC_VECTOR(1 DOWNTO 0);
3  SIGNAL out,c,d,e      : STD_LOGIC;
4  BEGIN
5      WITH a SELECT b <=  c WHEN "10",
6                          d WHEN '01',
7                          e WHEN OTHERS;
```

```
8
9  END structure;
```

- Bedingung für Signale außerhalb eines Prozesses

```
1  ARCHITECTURE structure OF Signalbedingung IS
2  SIGNAL a,b,c,d: STD_LOGIC;
3  BEGIN
4      a<= NOT a WHEN b = '1' ELSE (c XOR d);
5  END structure;
```

- Clock-Signale

```
1  ARCHITECTURE behavior OF Clocksignal IS
2  SIGNAL clk: STD_LOGIC;
3  SIGNAL count: STD_LOGIC_VECTOR(4 DOWNTO 0);
4  BEGIN
5      clk <= clock_50;
6      PROCESS(clk)
7          IF (clk='1' AND clk'event) THEN -- wenn clk steigende Flanke
8              count <= count + 1;
9          ELSE
10             count <= count;
11         END IF;
12     END PROCESS;
13 END behavior;
```

- Variablen und Schleifen

```
1  ARCHITECTURE behavior OF Schleife IS
2  SIGNAL clk,reset: STD_LOGIC;
3  SIGNAL register: STD_LOGIC_VECTOR(6 DOWNTO 0);
4  BEGIN
5      clk <= clock_50;
6      reset <= NOT KEY(0);
7
8      PROCESS(clk)
9          VARIABLE i : INTEGER := 0;
10         IF reset ='1' THEN
11             FOR i IN 0 TO 5 LOOP
12                 IF i = 0 OR i = 1 THEN
13                     register(i) <= NOT register(i+1);
14                 ELSE
15                     register(i) <= register(i+1);
16                 END IF;
```

```
17                END LOOP;
18            END IF;
19        END PROCESS;
20  END behavior;
```

2.5 Simulation und Synthese

Ein wichtiger Unterschied besteht zwischen simmulierbarem (oder funktionellem) und synthetisierbaren Code. Wie der Name schon sagt, lässt sich simmulierbarer Code nicht unbedingt auf der Hardware ausführen. In den Übungen wird ausschließlich synthetisierbarer Code benötigt. Dieser zeichnet sich dadurch aus, dass er vom Übersetzungsprogramm/Synthesetool (in unserem Fall Quartus2) in eine Netzliste übersetzt werden kann und somit auf den FPGA programmiert werden kann. Der simmulierbarer Code dagegen kann auch Elemente enthalten welche nur für die Simmulation übersetzt werden können, woraus aber keine Synthese und damit keine Netzliste erstellt werden kann. Dieses soll an folgendem, abstrakten Beispiel anschaulicher gemacht werden:

```
1  ARCHITECTURE Oszillator OF Simmulationsbeispiel IS
2  CONSTANT clk: time := 1000 ns;
3  BEGIN
4      PROCESS(clk, input)
5      BEGIN
6          out <= REJECT 10 ns INERTIAL NOT input AFTER 10 ns;
7          WAIT FOR clk/2;
8          out <= REJECT 10 ns INERTIAL input AFTER 10 ns;
9          WAIT FOR clk/2;
10     END PROCESS;
11 END Oszillator;
```

Mit den Befehlen **REJECT**, **INERTIAL** und **AFTER** kann das (zeitliche) Verhalten von realen Bausteinen beschrieben werden. Soll der Code synthetisiert werden, ist dieses nicht mehr nötig bzw. möglich, da die Beschreibung auf den realen Bauelementen läuft und daher die Zeitangaben nicht mehr zum tragen kommen. Bei der **WAIT** Anweisung handelt es sich ebenfalls um eine nicht synthetisierbare Anweisung. Die reale Hardware kann nicht wie eine Simmulation oder ein Softwareprozess einfach “anhalten”. Soll ein Warten implementiert werden, muss dieses indirekt

über einen Counter realisiert werden. Auch das angeben von Konstanten (sowie von Variablen) ist nur sehr eingeschränkt möglich. Das Zuweisen von direkten Zeiten, wie im Beispiel, ist nicht möglich. Daher kann auf diese Weise kein Taktgeber realisiert werden. Allgemein können Variablen wie Integer nur dazu verwendet werden, den “Schreibaufwand” zu minimieren, jedoch nicht als Variablen, die innerhalb der Laufzeit benutzt werden können. Einer der wenigen Fälle, wo Integer zur Anwedung kommt:

```
1  PROCESS(clk)
2  VARIABLE i : INTEGER := 0;
3  BEGIN
4      IF (clk = '1' AND clk'event) THEN
5          FOR i IN 0 TO 10 LOOP
6              IF i = 0 OR i = 2 THEN
7                  shift_reg(i + 1) <= shift_reg(i);
8              ELSE
9                  shift_reg(i + 1) <= NOT shift_reg(i);
10             END IF;
11         END LOOP;
12     ELSE
13         shift_reg <= shift_reg
14     END IF;
15 END PROCESS;
```

Dieses wäre nun äquvalent mit folgendem Code. Das Synthesetool wird auch bevor die Netzliste erstellt wird, das obige Beispiel zu dem folgenden Code umbauen, da die **LOOP** Schleife in der oben gegeben Form nicht synthetisierbar ist.

```
1  PROCESS(clk)
2  BEGIN
3      IF (clk = '1' AND clk'event) THEN
4          shift_reg(1) <= shift_reg(0);
5          shift_reg(2) <= NOT shift_reg(1);
6          shift_reg(3) <= shift_reg(2);
7          shift_reg(4) <= NOT shift_reg(3);
8          shift_reg(5) <= NOT shift_reg(4);
9          shift_reg(6) <= NOT shift_reg(5);
10         shift_reg(7) <= NOT shift_reg(6);
11         shift_reg(8) <= NOT shift_reg(7);
12         shift_reg(9) <= NOT shift_reg(8);
13         shift_reg(10) <= NOT shift_reg(9);
14         shift_reg(11) <= NOT shift_reg(10);
15     ELSE
```

```
16            shift_reg <= shift_reg
17        END IF;
18    END PROCESS;
```

Kapitel 3

Das erste Projekt in Quartus2

Innerhalb dieser Übungsstunde wollen wir uns mit dem Entwurfswerkzeug Quartus2 beschäftigen. Da der Hersteller Altera hierfür bereits Tutorials bereit stellt, soll auf diese Zurückgegriffen werden.

Achtung: Bitte beachten Sie bei der Programmierung der Boards unbedingt, dass der kleine Schalter links des LCD-Displays auf **RUN** steht, dass Board mit Strom versorgt wird und **angeschaltet** ist. Bitte stecken Sie erst dann das USB-Kabel in die USB-BLASTER-Buchse. Nun können Sie das Board programmieren. Ist der Schalter nicht auf run gestellt und/oder das Baord beim programmieren ausgeschaltet wird das Board beschädigt!

Aufgabe 1

Bearbeiten Sie das Tutorial *Quartus II Introduction Using VHDL Design*. Das PIN-Assigment File finden Sie auf dem Netzlaufwerk. Überspringen Sie das Kapitel 7.2 (Active Serial Mode Programming) des Tutorials. Vergessen Sie dabei nicht, wie in Abschnitt 2.2 beschrieben, alle nicht benutzten PINs auf tri-state zu setzen. Dieses müssen Sie bei allen folgenden Projekte nun immer selbstständig machen.

Aufgabe 2

Ziel dieser Aufgabe ist es, zu lernen, wie einfache eingabe und ausgabe Bausteine an einen FPGA angebunden werden. Hierzu sollen Sie die 18 Eingabeschalter SW_{17-0} mit den 18 roten LEDs $LEDR_{17-0}$ diekt verbunden werden. Dabei gibt Ihnen das Handbuch *DE2 User Manual* vor, dass beispielsweise der Schalter SW_0 mit dem PIN $N25$ und die $LEDR_0$ mit dem PIN $AE23$ verbunden ist. Um nun nicht alles PINS suchen zu müssen, bietet es sich an, diesen PINS über das sogegannte Assigment Namen zu geben. Dabei bietet es sich weiterhin an, die Namen aus dem *DE2 pin assignments.csv* File durch importieren zu verwenden.

Nun können Sie eine Zuordnung in folgender Form vornehmen:

```
1 LEDR(17) <= SW(17);
2 LEDR(16) <= SW(16);
3  ...
4 LEDR(0) <= SW(0);
```

Dabei scheint es sinnvoll, beide Schnittellen als Standard Logik Vektoren zu implementieren. Damit kann die Zurodnung wie folgt erfolgen:

```
1 LEDR(17 downto 0) <= SW(17 downto 0);
```

Oder, da beide Vektoren die gleiche Größe haben:

```
1 LEDR <= SW;
```

Erstellen Sie nun ein Projekt und testen Sie dieses auf dem Board, dass die 18 Schalter mit den 18 LEDs so verbindet, dass jeweils ein Schalter eine LED ein- bzw. ausschaltet.

Erweitern Sie nun Ihren Code so, dass folgendes für alle Schalter funktioniert. Dabei sollen Sie auch den Vorteil der Vektoren nutzen, so dass am Ende nicht für jede LED eine eigene Zeile dasteht.

```
1 LEDR(0) <= SW(0) AND SW(9);
2 LEDR(1) <= SW(1) AND SW(10);
3   ...
```

Kapitel 4

VHDL Grundlagen - Part I

In diesen Übungen sollen Sie den Umgang in der Hardwarebeschreibung besser kennen lernen. Dazu sind Ihnen einige Aufgaben vorgegeben, welche jeweils bestimmte Grundlagen und Aspekte vermitteln sollen.

Aufgabe 1

Wir wollen uns noch weiterhin etwas mit den Lichtern und Schaltern auf dem Board beschäftigen. Zum Anfang verwenden Sie einen der roten LEDs und drei der Schalter um einen einfachen Multiplexer aufzubauen. Ein Multiplexer hat die Eigenschaft, dass ein Eingang bestimmt, welcher der beiden weiteren Eingänge zum Ausgang durchgeschalten werden soll. Sie können dieses als eine VHDL Aussage in der Form `ausgang <= ...` ausdrücken. Bauen Sie somit einen 2-1 Multiplexer. SW_0 und SW_1 dienen dabei als die 2 Dateneingänge und SW_2 als Umschalter. Die Ausgabe erfolgt über die $LEDR_0$.

Erweitern Sie Ihren VHDL-Code nun so, dass es Ihnen möglich ist einen 8-bit breiten, 2-zu-1 Multiplexer zu realisieren. Benutzen Sie SW_{17} als Umschalter und die Schalter SW_{0-15} als die 2 Blöcke mit jeweils 8 Eingängen. Geben Sie die Eingaben auf den roten LEDs $LEDR_{0-17}$ aus und das Ergebnis auf den grünen LEDs $LEDG_{7-0}$.

Aufgabe 2

Schreiben Sie einen VHDL Code, der einen 3-bit breiten, 5-zu-1 Multiplexer realisiert. Benutzen Sie SW_{17-15} als Umschalter und die restlichen Schalter als Eingabeschalter. Geben Sie Ihre Eingabe wieder auf die roten LEDs aus und benutzen Sie für die Ausgabe die grünen LEDs ($LEDG_{2-0}$). Um den logischen Operationen besser folgen zu können, sollten Sie Zwischensignale verwenden.

Aufgabe 3

Eine 7 Segmentanzeige besteht im wesentlichen aus 7 LEDs die einzeln angesteuert werden können. Um das Darstellen von Zahlen auf einer 7 Segmentanzeige zu vereinfachen wird oft ein 7-Segment Decoder vorgeschalten. Dieser soll in unserem Fall 4 Eingänge haben (4 Bit) und damit die Zahlen 0-9 darstellen können. Als Eingabe verwenden Sie 4 Schalter und als Ausgabe die 7 Segmentanzeige.

```
1 HEX0: OUT STD_LOGIC_VECTOR(0 TO 6)
```

Bedenken Sie bei der Erstellung, dass diese Entity später wieder verwendet werden soll, wenn der Counter erstellt wird. Hilfreich könnte die Verwendung des *CASE* oder *WITH...SELECT* Statements sein.

Aufgabe 4

Beschreiben Sie zunächst einen Volladdierer mit den Eingängen `a,b,ci` und den Ausgängen `s,co`. Bevor sie nun mit Hilfe des Volladdierers einen Carry-Ripple-Addierer realisieren, sollten Sie diesen auf Korrektheit testen, indem sie die Eingänge mit Schaltern verdrahten und die Ausgänge mit LEDs.

Mit der beschriebene Volladdiererkomponente können Sie nun recht einfach einen Carry-Ripple-Addierer realisieren. Verwenden Sie hierzu SW_{7-4} und SW_{3-0} um die Eingänge A und B zu realisieren. SW_8 gibt Ihnen den In-Carry. Geben Sie wieder

Ihre Eingaben auf den roten LEDs aus und verbinden Sie *S*, und *Out*-Carry mit den grünen LEDs. Die Verwendung sollte ungefähr so aussehen:

```
1 bit0 : voll_addierer PORT MAP(A(0),B(0),C(0),S(0),C(1))
```

Kapitel 5

VHDL Grundlagen - Part II

In diesen Übungen werden Sie einen elementaren Teil von VHDL kennen lernen. Zum Speichern von Bits haben Sie verschiedene Möglichkeiten. Diese Übung wird Ihnen vermitteln, wie Sie Latches, Flip-Flops und Register selbst implementieren können.

Aufgabe 1

Hierbei soll auf das Altera Tutorial zurückgegriffen werden. Bearbeiten Sie in Tutorial drei *Latches, Flip-flops, and Registers* die Teile eins bis drei. Um eine neue vwf Datei im ersten Teil zu erstellen müssen Sie unter `File` $\rightarrow$ `New..` gehen.

Kapitel 6

VHDL Grundlagen - Part III

Ein weiterer elementarer Teil von VHDL sind Counter. Diese werden sehr häufig benutzt und weisen oft eine ähnliche Struktur auf. Grundlegend dabei ist, dass das hochzählen dabei über eine clock funktioniert und meist alle Prozesse immer Taktsynchron arbeiten.

Aufgabe 1

Erstellen Sie einen VHDL Code, der die Zahlen 0 bis 9 auf der 7-Segmentanzeige `HEX0` hochzählt. Jede Zahl soll dabei für etwas eine Sekunden dargestellt werden. Ihr Zähler erhält als Eingabe das 50-MHz Signal des DE2- Boards (`CLOCK_50`). Verwenden Sie bei der Ausgabe ihren Code von der 7-Segmentanzeige aus Aufgabe 3. Als Grundstruktur können Sie folgendes verwenden:

```
1  LIBRARY ieee;
2  USE ieee.std_logic_1164.all;
3  USE ieee.std_logic_unsigned.all;
4
5  ENTITY counter IS
6      PORT (
7          CLOCK_50      : IN     STD_LOGIC;
8          HEX0          : OUT    STD_LOGIC_VECTOR(6 TO 0)
9          );
10 END counter;
11
12 ARCHITECTURE behavior OF counter IS
```

```
13 COMPONENT htb   -- Hex to binary decoder
14     PORT (
15         input   : IN    STD_LOGIC_VECTOR(3 DOWNTO 0);
16         output  : OUT   STD_LOGIC_VECTOR(6 DOWNTO 0)
17         );
18 END COMPONENT;
19
20 SIGNAL clk       : STD_LOGIC_VECTOR;          -- Clock
21 SIGNAL count     : STD_LOGIC_VECTOR(24 DOWNTO 0);    -- Counter
22 SIGNAL number    : STD_LOGIC_VECTOR(3 DOWNTO 0);     -- Output number in hex
23 SIGNAL output    : STD_LOGIC_VECTOR(6 DOWNTO 0);     -- Output for the hex display
24
25 BEGIN
26
27     -- Connect the signals here
28     -- ...
29     Digit0: htb PORT MAP (number, output);
30
31
32     PROCESS (clk)
33     BEGIN
34         IF () THEN -- Syncronize with the clock
35             -- First of all an implementation for "counting" 1 second is needed
36             -- Thnik about overflow
37         END IF;
38     END PROCESS;
39
40     PROCESS (clk)
41     BEGIN
42         IF () THEN -- Syncronize with the clock
43             IF () THEN -- Trigger the action to count
44                 IF () THEN -- Is the number grater then 9?
45                     -- reset it
46                 ELSE
47                     -- increase it
48                 END IF;
49             END IF;
50         END IF;
51     END PROCESS;
52 END behavior;
```

```
1 LIBRARY ieee;
2 USE ieee.std_logic_1164.all;
3
```

```
4  ENTITY htb IS
5      -- Define ports
6  END htb;
7
8  ARCHITECTURE behavior OF htb IS
9  BEGIN
10     -- Behavior of the hex to binary converter
11     -- You can copy it from your 7-segment display project
12 END behavior;
```

Aufgabe 2

Erweitern Sie nun Ihren Code von Aufgabe 1 so, dass Ihr Zähler nicht nur bis 9 sondern bis 999 zählen kann. Passen sie die Taktung dabei so an, dass die Zählfrequenz ca. 100ms beträgt.

Kapitel 7

Hardware CRC-Checker

Im Gegensatz zum Parity-Check, der nur einfache Bit-Fehler erkennen kann, können mit dem CRC-Check auch viele Mehrfachfehler sicher oder doch zumindestens mit einer hohen Wahrscheinlichkeit erkannt werden. Das Verfahren basiert auf Polynomarithmetik. Eine n-stellige Nachricht wird als Polynom $(k-1)$-ten Grades aufgefasst, wobei jedes Bit der Nachricht ein Koeffizient des Polynoms darstellt. Vorteil ist, das dafür eine Modulo-2-Arithmetik verwendet werden kann. D.h. nach jedem Rechenschritt wird Modulo 2 gerechnet. Dadurch sind bei Additionen und Subtraktionen im Binären keine Überträge zu berücksichtigen. Beispiel:

$$\begin{array}{rr} 10011011 & 10011011 \\ +\ \underline{11001010} & -\ \underline{11001010} \\ 01010001 & 01010001 \end{array}$$

Additionen und Subtraktionen führen somit zum gleichen Ergebnis. Die Verknüpfung der einzelnen Bits kann mit xor erfolgen. Die Division erfolgt korrespondierend zur klassischen Division, mit dem Unterschied das bei den einzelnen Subtraktionen modulo 2 gerechnet wird. Dadurch fallen Überträge weg und die Berechnung wird wesentlich einfacher. Beispiel:

$$
\begin{array}{l}
111010100/101 = 1101110 \\
\text{-}\underline{101} \\
\quad 100 \\
\quad \text{-}\underline{101} \\
\quad\quad 011 \\
\quad\quad \text{-}\underline{000} \\
\quad\quad\quad 110 \\
\quad\quad\quad \text{-}\underline{101} \\
\quad\quad\quad\quad 111 \\
\quad\quad\quad\quad \text{-}\underline{101} \\
\quad\quad\quad\quad\quad 100 \\
\quad\quad\quad\quad\quad \text{-}\underline{101} \\
\quad\quad\quad\quad\quad\quad 010 \\
\quad\quad\quad\quad\quad\quad \text{-}\underline{000} \\
\quad\quad\quad\quad\quad\quad\quad 10 = \text{Rest}
\end{array}
$$

Für den CRC-Check wird ein Bit-String (oder Polynom) der Länge k gewählt. Dabei muss das erste und das letzte Bit eine 1 sein. Ein Beispiel ist 10011. Dieser Bit-String wird als Generator G bezeichnet. Die eigentliche Nachricht n (bzw. ihr Polynom $N(x)$) wird um $(k-1)$ Stellen nach links geshiftet, also n wird mit2^{k-1} multipliziert (bzw. $N(x) \cdot x^{k-1}$). Diese geshiftete Polynom wird mit Modulo 2 Arithmetik durch G geteilt. Durch die einfache Berechnung der Subtraktionen ist die Division auch einfach zu berechnen. Bei der Division bleibt ein Rest $R < G$ übrig. Von der geshifteten Nachricht wird dieser ermittelte Rest abgezogen (was in modulo 2 gleichbedeutend ist mit addiert). Damit wird die Nachricht durch G teilbar. Dabei werden nur die zugefügten Stellen modifiziert, die Bitfolge der eigentlichen Nachricht bleibt unangetastet. Diese Nachricht kann gesendet werden. Zur Verifikation der Nachricht auf der Empfängerseite ist diese durch G zu teilen. Bleibt ein Rest, ist die Übertragung fehlerhaft.

Hier nochmals zusammengefasst die einzelnen Schritte zur Berechnung der Checksumme:

1. Gegeben eine Nachricht M und ein CRC-Generator G mit Länge k.
2. Shifte die Nachricht M um $k-1$ Bits nach links (füge $k-1$ Nuller an)
3. Dividiere die neue Nachricht durch den Generator (mit Modulo 2 Arithmetik)

4. Addiere (bzw. Subtrahiere, ist hier das gleiche) den Rest zur Nachricht dazu (Modulo 2 Arithmetik!)
5. Ergebnis ist die zu sendende modifizierte Nachricht inklusive Checksumme

Diese Nachricht kann folgendermaßen geprüft werden:

1. Dividiere modifizierte Nachricht durch den Generator
2. Ist Rest ungleich Null, ist die Nachricht fehlerhaft.

Das Verfahren soll zunächst in VHDL umgesetzt werden. Dazu sind zwei Routinen zu schreiben, eine zum Erzeugen der um die Prüfsequenz erweiterte Nachreicht, eine um eine gegebene Nachricht zu verifizieren. Ein Ansatz basiert auf einfachen logischen Operation. Abbildung 7.1 zeigt das Prinzip. Dabei wird die obige Division im Modulo 2 System nachgebildet. Interessant ist, dass das eigentliche Ergebnis der Division nicht interessiert. Die Nachricht ist schrittweise Bit für Bit abzuarbeiten.

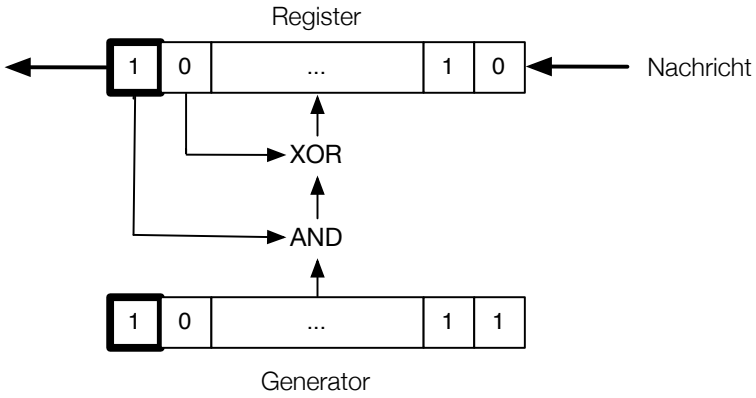

Abb. 7.1 Prinzip des CRC-Checkers mit Hilfe eines Schieberegisters

Um die Implementierung zu optimieren bietet sich an ein Schieberegister zu verwenden bzw. in VHDL nachzubilden, welches die Länge der Nachricht und des Polynoms hat, anstatt eines Registers mit nur der Länge des Polynoms. In dieses Register wird nun die Nachricht Bit für Bit geschoben. Immer dann, wenn das oberste Bit des Registers 1 ist, werden die ersten $k-1$ Bit des Registers mit dem Generatorpolynom bitweise mit XOR verknüpft.

Da das erste Bit des Registers bein nächsten Schieben herausgeschoben wird, kann das erste Bit des Generators weggelassen werden. Dann muss allerdings auch das Register um das oberste Bit gekürzt werden. Bb eine xor-Verknüpfung stattfindet entscheidet somit nicht mehr das oberste Bit im Register, sondern das Bit was gerade aus dem Register herausgeschoben wurde.

Nach der Nachricht ist noch die entsprechende Anzahl Nullen in das Register zu schieben (siehe Schritt 2 des Verfahrens). Wurde eine Nachricht geprüft, muss das Register nur Nullen enthalten. Wurde ein CRC-generiert, so enthält das Register die Prüfsumme.

Aufgabe 1

Schreiben Sie nun als erstes eine VHDL Implementierung die das oben genannte Verfahren realisiert. Als Polynomlänge gehen sie von 8 Bit aus, für die Nachricht (inkl. CRC-Summe) sehen Sie eine Länge von 32 Bit vor. Zusätzlich sollten Sie ein Enable-Signal vorsehen. Sobald Ihre Komponente mit der Berechnung fertig ist, sollte das Enable-Bit wieder von dieser zurückgesetzt werden. Da der Ablauf genau der gleiche für die Generierung als auch für die Prüfung der Nachricht ist, ist es nicht nötig, zwischen Generator und Prüfer umzuschalten. Durch das Anhängen der Nullen im Falle des Generators hat die Nachricht die gleiche Länge und muss gleich oft geschiftet werden. Als Ergebnis steht einmal die Prüfsumme im Register und das anderemal der Rest, der bei einer positiven Prüfung Null ist.

Um Ihre implementierung zu testen, überlegen Sie sich eine geeignete Beschaltung auf dem Board und verbinden Sie die Signale. Hifreich hierbei wird ein Signal für die Addresse, sowie ein Write Signal sein. Sie können schonmal einen Blick auf die nächste Übung werfen. Dort sehen Sie, wie Ihre Komponente dann fertig funktionieren sollte.

Kapitel 8

CRC-Checker als NIOS2-Komponente

Bevor wir uns dem Softcoreprozessor NIOS2 widmen, wollen wir unsere CRC-Komponente in VHDL so vorbereiten, dass wir diese in einer späteren Übung als Prozessorkomponente an den NIOS2 anbinden können.

Die CRC-Komponente soll frei programmierbar werden. Dabei müssen die Werte für das Generatorpolynom, die Statusbits und die Nachricht in Registern zwischengespeichert werden. Der Zugriff auf die Komponente soll deshalb über einen 32 Bit breiten Datenbus erfolgen, wobei die Datenleitungen für den Lese- und den Schreibzugriff getrennt sein dürfen. Eine Übersicht finden Sie in Abbildung 8.1.

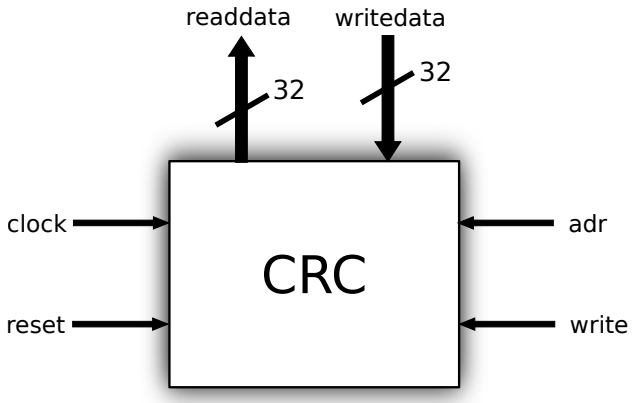

Abb. 8.1 CRC-Komponente

Über den Datenbus sollen zwei verschiedene Register angesteuert werden, die über die Adressleitung auswählbar sind. Die Registertabelle soll folgendermaßen aufgebaut sein:

Adresse	**Funktion**
0	Nachricht bzw. Ergebnis
1	Polynom (8 Bit) | 0 | $\cdots$ | 0 | Enable |

Über das Enable-Bit an der Stelle 0 an Adresse 1 soll die CRC-Komponente gestartet werden können, damit die Komponente weiß, wann eine Nachricht bereit steht. Sobald die Komponente fertig ist, soll dieses Bit wieder zurückgesetzt werden. Über das write-Signal soll gesteuert werden, wann die Eingabe auf dem Datenbus in das momentan adressierte Register übernommen wird. Das Reset soll asynchron erfolgen und die ganze Komponente (alle Register) auf Null zurücksetzen.

Aufgabe 1

Implementieren Sie die oben vorgestellte Komponente.

Binden Sie die Komponente in Ihre Top-Level-Entity ein. Als Taktsignal soll wieder direkt der 50 MHz Takt verwendet werden. Als Dateneingabe dienen die Schiebeschalter 14 bis 0, als Ausgabe die roten LEDs 17 bis 0. Dabei müssen Sie sich eine sinnvolle Zuordnung überlegen, da weniger Schalter und LEDs zur Verfügung stehen, wie die 32-Bit Register benötigen. Benutzen Sie SW17 für den Reset, SW16 für das write-Signal und SW15 für das Adressbit.

Kapitel 9

NIOS2

Innerhalb dieser Übungen wollen wir einen einfachen Softcoreprozessor bauen. Hierfür stellt Altera ein Tutorial bereit. Als Grundlage benötigen Sie das Tutorial *Timing Considerations with VHDL-Based Designs*.

Aufgabe 1

Bearbeiten Sie das Tutorial *Introduction to the Altera SOPC Builder Using VHDL Design* bis einschließlich Abschnitt 3.1. Vor der Programmierung des FPGAs setzten Sie alle offenen Pins noch auf Tri-state. Unter `Assignments` → `Settings` finden Sie unter der Kategorie `Device` einen Button `Device and Pin Options...`. Unter der Registerkarte `Unused Pins` können sie nun alle unbenutzten Pins auf `As input tri-stated` setzten.

Nun können sie mit Abschnitt 3.2, der Programmierung des FPGAs fortfahren. Nach Beendigung der Programmierung beenden Sie Quartus II (Projekt speichern) und öffnen Sie das Programm Nios II, um mit der Programmierung der Software für den Softcoreprozessor zu beginnen.

Aufgabe 2

Für die Programmierung des Softcoreprozessors wird in Zukunft das Programm Nios2 von Altera benutzt. Erstellen Sie ein neues Projekt (`File` → `New` → `Project`). Wählen Sie den obersten Eintrag aus (`Nios II C/C++ Application`) und klicken Sie auf `Next`. Als `Project Template` wählen Sie nun `Hello World Small` aus. Nun müssen Sie noch die `Target Hardware` auswählen. Öffnen sie hierzu Ihr SOPC Builder File (`nios_system.ptf`) . Klicken Sie nun auf `Finish`. Nun können Sie unter `Run` → `Run...` und der Auswahl von „NiosII Hardware“ das Projekt kompilieren lassen. Es wird anschliessend automatisch auf das FPGA gespielt (dieses Dauert etwa 2-3 Minuten).

Sollte das Projekt erfolgreich erstellt und überspielt worden sein, müssten Sie in der Konsole eine Meldung `Hello from NIOS II!` bekommen. Der prozessor kann über den Taster KEY0 Zurückgesetzt (”Reset”) werden. Da in diesem Fall das Programm nochmals abläuft, bekommen Sie erneut eine Meldung in der Konsole.

Aufgabe 3

Erstellen Sie sie nun ein neues Projekt analog der vorherigen Aufgabe, nur dass Sie dieses mal als Template `Blank Projekt` wählen. Erstellen Sie nun in Ihren neuen Projekt eine neue Quelldatei (Source File) mit dem Namen `lights.c`. Fügen Sie in das File nun folgenden Quelltext ein:

```
1 #define Switches (volatile char *) 0x0003000
2 #define LEDs (char *) 0x0003010
3
4 void main()
5 {
6 while (1)
7 *LEDs = *Switches;
8 }
```

Das Programm definiert sich einen Zeiger ”Switches”, der auf die Adresse (0x0003000) des in SOPC-Builder definierten PIOs zeigt. Weiterhin wird ein Zeiger auf die Adresse der LEDs (0x0003010) definiert. ”Volatile” verhindert, dass die C-Code Optimierung das Auslesen der Registereinträge in denen der Status der Schalter ge-

speichert ist nur einmalig bei der Initialisierung durchgeführt wird. Die Optimierung geht von dem C-Code aus, in dem während des Programmablaufes softwareseitig keine neue Zuordnung der Registerinhalte erfolgt. Innerhalb von ”main” wird bei jedem Durchlauf der while Schleife der Inhalt der Switch Register in den Inhalt der LED Register geschrieben.

Da bei der Implementierung des Nios II nur ein on-board-memory verwendet wurde, muss noch Angegeben werden, dass nur eine abgespeckte Version der C-Libraries verwendet wird. Hierzu öffnen sie die `Properties` der Syslib Ihres Projektes. Dort finden Sie unter `System Library` die Möglichkeit folgende Optionen zu aktivieren: `Program never exit`, `Lightweight device driver API`, `Reduced device drivers` und `Small C Library`. Zusätzlich schalten Sie die Option `Support C++` und `Clean exit` aus.

Nun können Sie ihr Projekt kompilieren lassen (`Run` $\rightarrow$ `Run`). Achten Sie darauf, dass unter `Project` Ihr gerade erstelltes Projekt ausgewählt ist. Es sollte Ihnen nun möglich sein, die grünen LEDs (LEDG0 - LEDG7) über die Schalter (SW0 - SW7) zu schaltern.

Kapitel 10

NIOS2 mit SDRAM

Erstellen Sie ein neues System auf der Grundlage eines Nios II Prozessors. Um mehr Platz für den kompilierten Programmcode sowie die Laufzeitdaten zu haben, soll als Speicher der auf dem DE2-Board vorhandene SDRAM verwendet werden. Im SOPC-Builder gibt es bereits eine fertige IP-Komponente für den SDRAM-Controller. Die physikalische Verdrahtung der Taktleitung zum RAM auf dem Board führt jedoch zu einer Taktverzögerung (clock skew), so dass die Taktflanke in Relation zu den anderen Signalen zu spät am RAM-Baustein ankommt. Es ist in diesem Fall notwendig, die Flanke für den SDRAM um 3 ns nach vorn (früher) zu verschieben. Dies wird über einen Phase-Locked-Loop (PLL) realisiert, der ein Taktsignal mit der selben Frequenz aber zusätzlicher negativer Phasenverschiebung liefern kann. Der Takt für die restlichen Komponenten wird dabei unverändert weiterpropagiert. Abbildung 10.1 verdeutlicht diese Zusammenhänge.

Im Dokument *„Using the SDRAM Memory on Altera's DE2 Board with VHDL Design“ (`tut_DE2_sdram_vhld.pdf`)* finden sie eine entsprechende Anleitung.

Aufgabe 1

Um einen SDRAM in den Softcoreprozessor, der in den letzten Übungen erstellt wurde, einzubinden folgen Sie dem Tutorial *Using the SDRAM Memory on Altera's DE2 Board with VHDL Design*. Entfernen Sie beim Zusammenstellen der Komponenten im SOPC-Builder die beiden PIO Komponenten (Switches und LEDs).

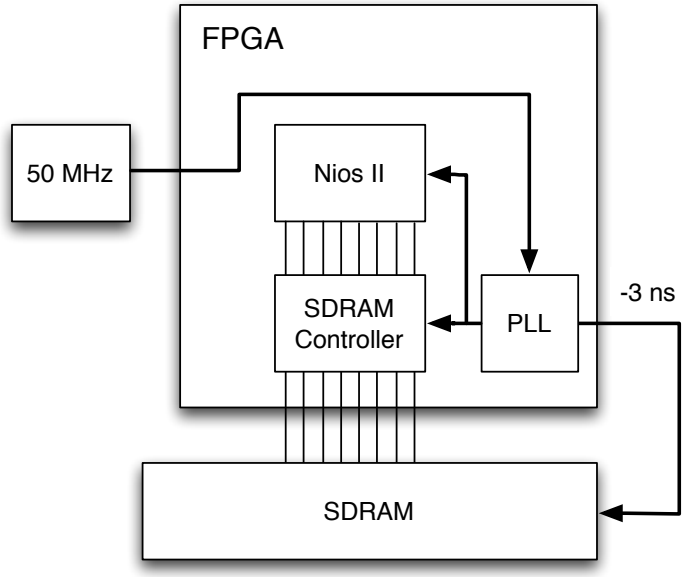

Abb. 10.1 PLL für die Phasenverschiebung des SDRAM-Taktes

Lesen Sie bitte Abschnitt 4 durch, aber überspringen Sie die Anweisungen in Abschnitt 4 des Tutorials (sowohl das Anlegen der Datei `lights.vhd`, als auch das Experiment). Fahren Sie mit Abschnitt 5 wieder normal fort.

Versuchen Sie nun, die im Tutorial angegebene `lights.vhd` entsprechend anzupassen, da die PIO Module entfernt wurden. Beachten Sie auch, dass der VHDL-Code Fehler enthält. Sie können das Quartus II Projekt beliebig oft kompilieren lassen um die Fehler zu lokalisieren. (Hinweis: es muss nur die Datei `lights.vhd` verbessert werden).

Aufgabe 2

Nachdem Sie Ihren neuen Softcoreprozessor auf den FPGA programmier haben, können Sie nun unter Nios II ein neues Projekt erstellen. Dabei können Sie nun entweder selbst ein ”Hello World”-Programm schreiben oder das gegebene Template

verwenden. Beachten Sie, dass es nun nicht mehr nötig ist, die ”small” Variante zu wählen, da der SDRAM Ihnen genügend Speicherplatz zur Verfügung stellt. Falls Sie das Programm selbst schreiben, können Sie ”normalen” C-Code für ein ”Hello World”-Programm verwenden:

```
1 #include <stdio.h>
2
3 int main()
4 {
5 printf(''Hello World again!\n'');
6
7 return 0;
8 }
```

Unter den `syslib` Eigenschaften müssten sie nun unter `System Library` den SDRAM anstelle des on-chip-memory auswählen.

Kapitel 11

Software-CRC

Nun soll analog zu der Hardware-CRC-Komponente ein C-Programm erstellt werden. Bilden Sie daher die Schieberegister in C nach.

Aufgabe 1

Schreiben Sie ein C-Programm, dass eine binär-codierte Nachricht einliest und die CRC Prüfsumme generiert. Außerdem soll das Programm eine Nachricht mit angefügter Prüfsumme auf Fehler prüfen können, d.h. ob die Nachricht fehlerfrei übertragen wurde. Für die Ein- bzw. Ausgabe können Sie die Bibiothek `stdio.h` und die Befehle `scanf` sowie `printf` verwenden.

Laden Sie Ihr Programm nun auf den Softcoreprozessor und verifizieren Sie dessen Korrektheit. Im Internet finden die genügend Beispiele mit CRC-Prüfsummen um Ihr Programm zu testen.

Kapitel 12

Hardware/Software Co-Design

In dieser Übung soll das bereits von Ihnen entwickelte programmierbare CRC-Modul an den Nios II Prozessor angebunden und dann per Software gesteuert werden. Altera stellt dafür ein Konzept bereit, um per Memory-Mapped-IO auf die Register von Komponenten eines SOPC zugreifen zu können.

Das Avalon-MM System-Interconnect-Fabric ist ein Bus-ähnliches Konstrukt, das die Verbindung von Komponenten innerhalb eines mit dem SOPC-Builder erstellten Systems ermöglicht. Für die angeschlossenen Komponenten erscheint die Anbindung und Adressierung wie ein gewöhnlicher gemeinsamer System-Bus, intern arbeitet das Avalon-MM allerdings mit Eins-zu-Eins-Verbindungen, genauer gesagt mit Slave-seitiger Arbitrierung. Bei dieser Verbindungsart wird auf der Seite des Slaves entschieden, mit welchem Master er kommunizieren soll, so dass zur selben Zeit verschiedene Master mit verschiedenen Slaves Daten austauschen können.

Im Folgenden wird das Avalon-MM System-Interconnect-Fabric der Einfachheit halber als „Avalon-Bus“ oder schlicht „Avalon“ bezeichnet werden, auch wenn dies (wie oben dargestellt) technisch gesehen nicht ganz korrekt ist.

Um mit einer Komponente am Avalon-Bus teilnehmen zu können, ist es notwendig, das entsprechende Avalon-Interface zu implementierten. Dabei gibt es einige elementare Signale, die jedes Interface beherrschen muss, sowie darüber hinaus gehende Erweiterungen, die optional sind.

Die Idee des Interconnect-Fabrics wird in Kapitel 2 von Volume 4, Sektion I des Quartus II Handbuchs beschrieben (Datei `qts_qii54003.pdf`). Die Spezifikation des gesamten Avalon-Systems findet sich in der Datei `mnl_avalon_spec.pdf`.

Aufgabe 1

Erstellen Sie ein neues System mit einem Nios II Prozessor, der Zugriff auf eine Instanz Ihres CRC-Moduls hat.

Dank der von Ihnen bereits implementierten Register kann eine Anbindung an den Avalon-Bus relativ leicht erfolgen. Die vorhandene Trennung von `readdata` und `writedata` passt direkt zum Avalon-Interface.

Benutzen Sie Ihre selbst entwickelte CRC-Komponente als Basis. Achten Sie darauf, dass nur die angegebenen Signale (und evtl. zusätzlich `read`) oder ihre Komplemente in der Schnittstelle nach außen geführt werden. Falls nötig, passen Sie Ihre Komponente noch einmal an.

Wenn Sie zuvor die Komponente wie angegeben erstellt haben, sollte sie bereits den Anforderungen eines Slave-Interfaces des Avalon-Busses genügen. Überprüfen Sie anhand des Timing-Diagramms für Lese- und Schreibzugriffe mit festen Wartezyklen, ob das auch für Ihre Implementierung gilt. Vorgesehen sind jeweils **null Wartezyklen**.

Anschließend können Sie beginnen, das CRC-Modul in den SOPC-Builder einzubinden. Dazu legen Sie die VHDL-Datei in Ihrem Projektverzeichnis ab (Alteras Entwurfsfluss sieht dafür das Unterverzeichnis `hw` vor, um den selbst erstellten Code von den automatisch erzeugten VHDL-Dateien unterscheiden zu können) und wählen im SOPC-Builder den Menüpunkt „File“ und dann „New Component“. Dort können Sie nun Ihren VHDL-Code auswählen und die Signale des Avalon-Busses mit Ihren eigenen verbinden. Sie benötigen neben dem Slave-Interface einen Clock-Input.

Beim Timing sollten Sie alle Werte vorerst auf **null** stellen. Geben Sie im „Component Wizard“ (rechte Registerkarte) als „Component Class Name“ `CRC` ein.

Sie können nun eine Instanz ihres CRC-Moduls dem System hinzufügen. Der Name dieser Instanz muss sich allerdings aus technischen Gründen vom Namen des Moduls unterscheiden. Nun können Sie das System generieren.

Aufgabe 2

Erstellen Sie in der Nios II IDE ein neues Projekt, dass das CRC-Modul ansteuert.

Ein „Mini-Treiber“ für den Registerzugriff ist bereits mit der Datei `crc_regs.h` gegeben, die Sie im Übungsverzeichnis finden. In dieser Header-Datei werden einige Makros definiert, die etwas von dem direkten Zugriff auf die IO-Ports der Komponente abstrahieren. So existieren für jedes Register eigene Lese- und Schreibbefehle und es muss nur noch die Basisadresse übergeben werden, der Offset zum gewünschten Register jedoch nicht. Die Form dieser Makros ist bereits von Altera für die Treiberentwicklung vorgesehen.

Erstellen Sie einen „richtigen“ kleinen Treiber in Form einer Auswahl an Funktionen, denen Sie die Werte für die Nachricht sowie das Polynom übergeben können. Achten Sie bei diesen Funktionen auch auf die korrekte bzw. sinnvolle Benutzung des „Enable“.

Schreiben Sie dann ein Hauptprogramm, dass unter Verwendung Ihrer Treiberroutinen eine Nachricht übergeben kann und durch geeignetes Abfragen des Enable-Bite weiß, wann die Hardwareberechnung fertig ist. Denken Sie auch daran, die Header-Dateien `system.h` und bei Bedarf `alt_types.h` (wenn Sie die Altera-Datentypen verwenden wollen) mit einzubinden.

Aufgabe 3

Ermöglichen Sie nun als Abschluss Ihrer Übungen das Benutzen des erstellten Hardware/Software-Co-Designs. Es sollte möglich sein, über die Konsole eine Nachricht sowie Polynom einzugeben und das CRC Prüfen bzw. generieren zu lassen.