

Versuch EP9
Digitalelektronik
Teil 2:
Programmierbare Logikbausteine (FPGA)

I. Zielsetzung des Versuches

Im vorherigen Versuch haben Sie einfache logische Grundsaltungen und ihre Realisierung mit einfachen Logikbausteinen (Digital-ICs) kennenlernt.

Im heutigen Versuch soll es um programmierbare Logikbausteine gehen (FPGA). Sie sind heute sehr preisgünstig und erlauben die einfache Realisierung auch komplexer Steuerungsaufgaben, wo früher sehr viele einfache Logikbausteine verschaltet wurden.

Die Programmierung solcher Logikbausteine (die Definition des logischen Verhaltens) kann auf zwei Arten geschehen:

- Bei Problemen, die vollständig mit bereits vorgegebenen Logikblöcken gelöst werden können, kann man zunächst einen Schaltplan zeichnen, in dem diese Logikblöcke miteinander verbunden werden. Dies kann den Entwurf deutlich beschleunigen und komplexe Lösungen abstrahieren, ist aber weniger flexibel.
- Bei allen anderen Problemen ist es besser, eine sogenannte Hardware-Beschreibungssprache wie VHDL oder Verilog zu verwenden. Mit dieser können Lösungen deutlich flexibler umgesetzt werden, resultiert aber meist in deutlich mehr Aufwand.

Wir werden in unserem Versuch das Programmieren mit VHDL kennenlernen.

II. Vorkenntnisse

Grundlagen der Digitaltechnik (Versuch EP8)

Informationen zum Umgang mit der Programmierumgebung finden Sie in dieser Versuchsanleitung.

III. Theorie zum Versuch

1. Logikbausteine

1.1. Einfache und programmierte Logikbausteine

Die klassischen Logikbausteine, die es seit Anfang der 1960er Jahre gibt, beherrschen immer nur bestimmte einfache Funktionen. Der Aufbau komplexerer Schaltungen (und sei es nur eine 6stellige Digitaluhr) führt schnell zu einer großen Ansammlung solcher Bausteine, die viel Platz braucht und letztlich hohe Kosten verursacht.

Ein erhebliches Problem tritt auf, wenn eine solche Schaltung modifiziert werden soll. Da alle Details der Schaltung durch die festen Leitungsverbindungen definiert sind, müssen diese entweder mit Drahtbrücken geändert oder die gesamte Schaltung neu aufgebaut werden.

Aus diesem Grund wurden Anfang der 1980er Jahre Bausteine entwickelt, deren genaues Verhalten durch eine Programmierung festgelegt werden kann. Ändern sich im Laufe einer Schaltungsentwicklung bestimmte Anforderungen, so können diese leicht durch Umprogrammierung umgesetzt werden.

Grundsätzlich ist zu unterscheiden zwischen:

- der programmierbaren Logik (FPGA), mit der wir uns in diesem Versuch befassen, und
- den Mikrocontrollern und Mikroprozessoren, die im nächsten Versuch behandelt werden.

1.2. Programmierbare Logik

Ein programmierbarer Logikbaustein (FPGA = Field Programmable Gate Array) ist vereinfacht betrachtet eine Ansammlung von vielen Gattern und Flipflops. Zusätzlich zu diesen kommen noch zahlreiche komplexe Funktionen hinzu: große Speicherblöcke (je nach Typ mehrere Millionen Bits), komplexe Logikfunktionen (z.B. Multiplizierer), Frequenzsynthese (PLL) usw.

Im unprogrammierten Baustein haben die Gatter und Flipflops noch keine Verbindung zueinander. Erst durch die Programmierung wird festgelegt, welche Gatter und Flipflops wie miteinander und mit den Anschlüssen des Bausteins verbunden werden.

Dabei kann die Beschreibung auf zwei Arten entwickelt werden:

- Als Schaltplan: Die Programmierumgebung auf dem PC stellt die Symbole für vorgegebene Bausteine bereit, die dann miteinander verbunden werden.
- Als Beschreibungssprache: Das Verhalten der Schaltung wird durch ein Text (die Beschreibung) mit logischen Gleichungen usw. beschrieben. Solche Sprachen (Hardware Description Languages, HDLs) wie VHDL oder Verilog haben aber nur auf den ersten Blick eine große Ähnlichkeit mit Programmiersprachen für Computer wie z.B. C/C++, sollten aber mit diesen nicht verwechselt werden.

Ist die Beschreibung geschrieben kann das Verhalten der Schaltung am PC simuliert werden, um die korrekte Funktion zu testen.

Schließlich wird eine Binärdatei erzeugt (Synthese/Implementierung) und vom PC in den Logikbaustein übertragen.

FPGAs haben aus technischen Gründen in der Regel nur einen SRAM-Speicher, d.h. sie verlieren ihre Programmierung beim Verlust der Versorgungsspannung. Sie können sich aber ihre Beschreibung sofort nach Anlegen der Versorgungsspannung automatisch von einem externen Flash-Speicherbaustein holen. In diesem Versuch werden wir diesen aber nicht verwenden.

2. Unterschiede zwischen programmierbarer Logik und Mikrocontrollern

Programmierbare Logik arbeitet „in Echtzeit“. Die Beschreibung schaltet die Leitungen wie eine „normale“, diskrete Schaltung zusammen, so dass die Eingangssignale sofort von den Gattern und Flipflops in der gewünschten Weise verknüpft und an die Ausgänge gegeben werden. Die Beschreibung stellt also die Weichen für die Signale, welche sich dann wie gewohnt über die Leitungen ausbreiten. Somit können alle Verknüpfungen *gleichzeitig* ausgeführt werden.

Diese Bausteine können oft Taktfrequenzen von mehreren 100 MHz erreichen und somit sehr schnell viele Daten verarbeiten.

Komplexe Rechenfunktionen (z.B. Division, trigonometrische Funktionen usw.) sind aber — vor allem bei den einfacheren Bausteinen — nicht oder nur mühsam zu realisieren.

Mikrocontroller arbeiten nach dem Prinzip eines Computers. Sie arbeiten ein bestimmtes Programm *nacheinander* (sozusagen Zeile für Zeile) ab, nehmen Eingangssignale entgegen, verknüpfen und vergleichen sie, geben sie danach als Ausgangssignale aus oder speichern etwas ab.

Daher sind diese Bausteine relativ langsam und können nur Signalfrequenzen von einigen MHz verarbeiten. Das liegt auch an den geringen Taktfrequenzen, die auch aus Gründen geringer Stromaufnahme oft nur bei höchstens einigen 10 MHz liegen.

Komplexe Rechenfunktionen sind aber leicht zu realisieren.

Das Programm kann aus elementaren Programmierbefehlen (d.h. im Assemblercode) aufgebaut werden. Üblich ist aber die Verwendung einer Hochsprache wie C/C++, da dies wesentlich einfacher und bequemer ist.

Heutzutage werden in FPGAs oft Mikrocontroller realisiert, da mit letzteren Steuerungsaufgaben einfacher und flexibler umgesetzt werden können.

3. Programmierbare Logik am Beispiel des FPGA vom Typ Spartan6

Als FPGA verwenden Sie den Typ Spartan6 (XC6SLX9) des Herstellers Xilinx.

Es handelt sich um einen der einfacheren Bausteine. Sein 144poliges Gehäuse bietet 102 Anschlüsse, die frei als Ein- oder Ausgänge definiert werden können. Die übrigen Anschlüsse dienen zur Spannungsversorgung und zur Programmierung.

Für die Programmierung des FPGA ist die genaue Kenntnis des Innenlebens oft nicht wichtig. Dennoch sollte man sie in groben Zügen kennen, um die Grenzen des Bausteins abschätzen zu können. So hat der XC6SLX9 11.440 Flipflops und 5.720 Wahrheitstabellen, welche die Verknüpfungen umsetzen. Sollten diese nicht reichen, muss ein anderer Baustein mit mehr Ressourcen verwendet werden, welcher in der Regel nicht nur teurer ist, sondern oft auch ein anderes Gehäuse hat.

Im Anhang finden Sie weitere Details zum Innenleben (Architektur) des FPGAs.

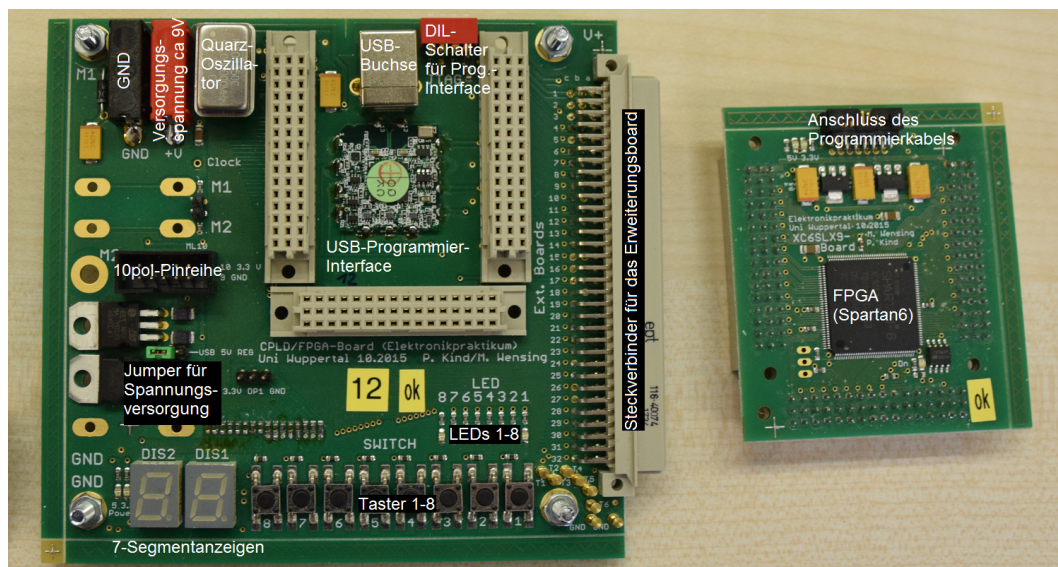
4. Hardware: FPGA-Board und Trägerboard

Der FPGA-Baustein befindet sich auf einer Platine (ca. 6 cm x 7 cm), der FPGA ist das schwarze quadratische Bauelement etwa in der Mitte. Oben an diesem Board ist ein Steckverbinder für Programmierkabel, falls ein externes Programmiergerät benutzt wird.

Dieses Board steckt auf einer Trägerplatine (ca. 12 cm x 11 cm), die neben der Spannungsregelung auch die Peripheriebauteile enthält, nämlich:

- 8 Taster als Eingabelemente. Wenn Sie den Taster drücken, ist das jeweilige Signal logisch 1, sonst 0.
- 8 Leuchtdioden (LEDs, leuchten bei logisch 1)
- 2 Siebensegmentanzeigen (Segmente leuchten bei logisch 1)
- Taktgenerator (Quarzoszillator, auswechselbar, d.h. verschiedene Frequenzen möglich)
- Bananenbuchsen für Spannungsversorgung (V+ und GND, etwa 9 Volt)
- 10polige Pinreihe und Pins für Oszilloskop-Tastkopf

Alle diese Elemente sind fest mit bestimmten Pins des FPGA verbunden.



Zwei Bananenbuchsen (M1 und M2, im Bild nicht bestückt) sind für Messungen vorgesehen und gehen über je einen 10-k Ω -Widerstand an zwei kleinere Pins, von wo aus sie über kleine Buchsenkabel mit Kontakten der 10poligen Pinreihe verbunden werden können.

An einer der unteren GND-Schrauben können Sie die Tastkopf-Krokodilklemme befestigen.

Links unten neben den Siebensegmentanzeigen und oben auf dem FPGA-Board befinden sich noch je 2 LEDs für die Versorgungsspannungen (3,3 V und 5 V), sie müssen beide leuchten!

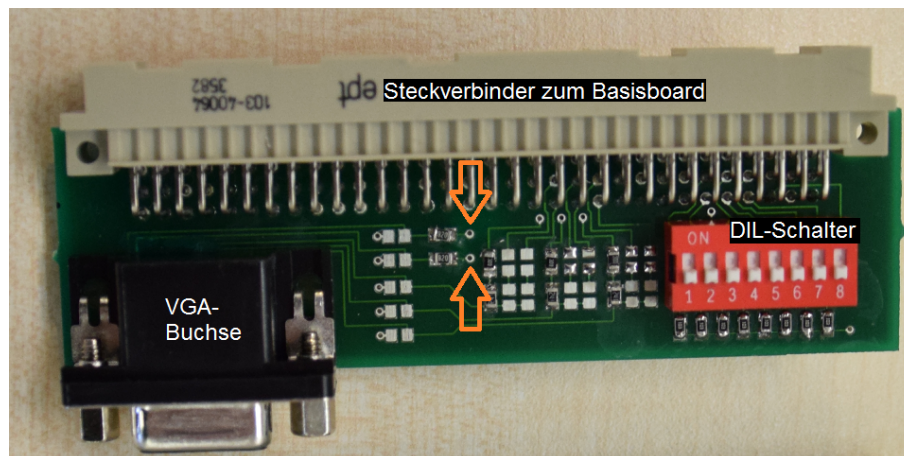
5. Hardware: USB-Programmierinterface

Wenn man das FPGA-Board entfernt, sieht man darunter eine USB-Buchse und ein USB-Programmierinterface. Schließt man das Board über diese USB-Buchse an den PC an, so kann man

- den FPGA direkt über das USB-Kabel programmieren, d.h. ohne externes Programmiergerät und
- das ganze Board und die Peripherie auch aus der USB-Spannung versorgen, d.h. es ist kein externes Versorgungsspannung erforderlich. (Dazu muß ein Jumper richtig gesetzt werden, der sich etwas unterhalb der 10pol-Pinreihe befindet.)
- Wichtig: Um das USB-Programmierinterface verwenden zu können, müssen die 4 DIL-Schalter neben der USB-Buchse auf ON (oben) stehen.

6. Hardware: Erweiterungsboard

Für die nachfolgenden Versuche wird außerdem noch ein Erweiterungsboard benötigt, welche zusätzliche Funktionen bereit stellt.



- Die DIL-Schalter bieten weitere 8 Eingänge (neben den Tastern auf dem Basisboard) für den FPGA.
- Über die VGA-Buchse kann ein Videosignal ausgegeben werden.
- An den Durchkontaktierungen, die mit den beiden orangenen Pfeilen markiert sind, können die Synchronisationssignale mit dem Oszilloskop nachgemessen werden.

IV. Versuchsdurchführung

1. Anleitung Entwurf und Test einer Verhaltens-Beschreibung

1.1. Vorbereitung

Die folgende Anleitung zeigt Ihnen Schritt für Schritt, wie Sie mit dem Programm „Xilinx ISE 14.7“ (im folgenden kurz „ISE“) eine Beschreibung eines einfachen Projekts realisieren.

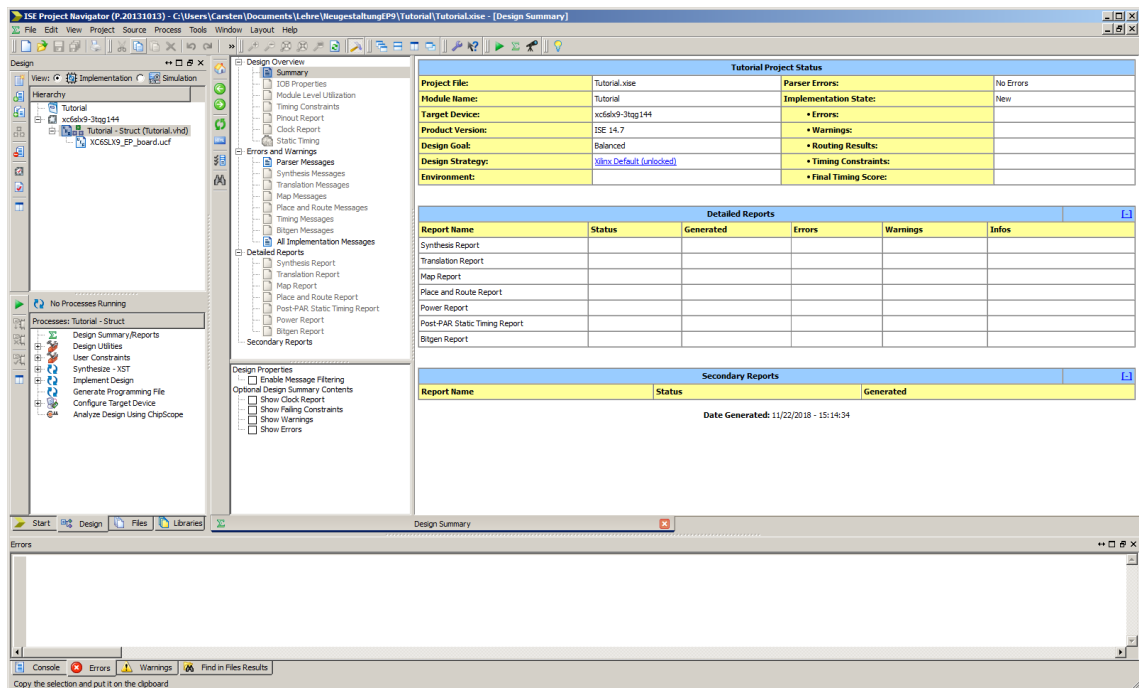
Nachdem Sie das Projekt in ISE geöffnet haben, können Sie dieses in eine Binärdatei übersetzen lassen, mit welcher der FPGA programmiert werden kann. Wenn das erfolgreich verlaufen ist, wird Ihr FPGA sich so verhalten, wie es in der Beschreibung vorgegeben wurde. Sie müssen nun aber überprüfen, ob das Verhalten (und damit die tatsächliche Beschreibung) auch dem gewünschten Verhalten entspricht.

Im Anschluss werden Sie dann einige einfache Änderungen an der Beschreibung vornehmen, diese erneut übersetzen auf den FPGA übertragen. Auch hier muss wieder überprüft werden, ob das tatsächliche Verhalten mit dem gewünschten übereinstimmt.

Das Öffnen der Design Suite: Entweder: Desktop → Xilinx ISE Design Suite 14.7 (ISE Project Navigator)
Oder: Start → Xilinx Design Tools → Project Navigator

Ein Projekt öffnen: Sie sehen nun die Entwicklungsumgebung und möglicherweise ein älteres Projekt.

- Schließen Sie *Tip of the day*, falls dieses Fenster erscheint.
- Öffnen Sie das Projekt unter *File* (oben links) → *Open Project*
- Ein Fenster zur Auswahl der Projekt-Datei öffnet sich.
- Gehen Sie in den Ordner unter *Desktop/Vorgaben_EP9/Vorgaben/Tutorial* und wählen die Datei *Tutorial.xise* aus. Klicken sie auf den *Open*-Knopf um das Projekt zu öffnen.
- Im oberen, linken Fenster sollte nun der Projektname (*Tutorial*), der FPGA-Typ (*xc6slx9-3tqg144*) sowie mehrere weitere Dateien zu sehen sein:



WICHTIGER HINWEIS: Vermeiden Sie, unter ISE auf Ihrem USB-Stick abzuspeichern! Es kann sonst leicht passieren, dass Sie das „programming file“ (das bit-File) aus einer falschen Datei (VHDL-Modul, alte Kopie) erzeugen, d.h. immer wieder eine völlig veraltete Beschreibung übertragen. Kopieren Sie Dateien auf Ihren USB-Stick nur über den normalen Windows-Explorer. Achten Sie außerdem bei den verschiedenen Versuchsteilen auf eine übersichtliche Verzeichnisstruktur. Kopieren Sie die Vorgaben bevor Sie diese bearbeiten, z.B. wenn Sie Müller und Meier heißen nach:

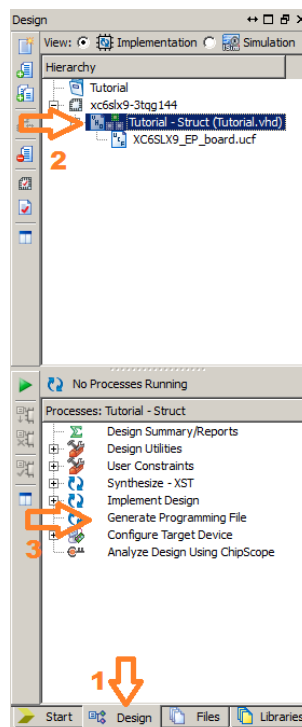
Desktop/Vorgaben_EP9/mueller_meier/

1.2. Übersetzen und Übertragen

Im folgenden werden Ihnen die wichtigsten Funktionen von ISE anhand eines Beispiels gezeigt.

Dabei werden Sie nun die Beschreibung übersetzen (implementieren) und in den Baustein übertragen.

- Ändern Sie die Einstellungen der linken Taskleiste:



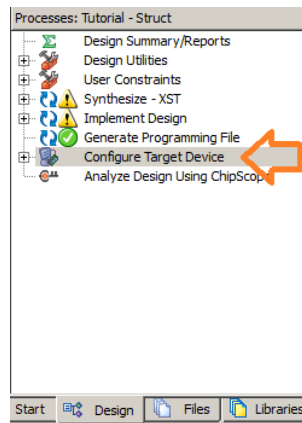
Wählen Sie die mit orangenen Pfeilen markierten Optionen der Reihenfolge nach aus, also:

1. Klicken Sie auf das Register *Design*

2. Klicken Sie auf Ihren Dateinamen (die vhd-Datei)

3. und doppelklicken danach *Generate Programming File*. Achtung: Sollte ISE nicht erkannt haben, dass die Beschreibung geändert wurde (es werden orangene Kreise mit weißem Fragezeichen angezeigt), ist es besser dort ein Rechts-Klick zu machen und dann *Rerun all* zu wählen.

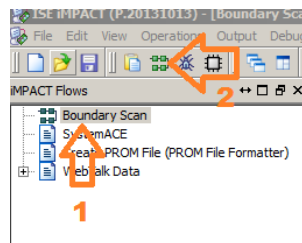
- Es erscheinen evtl. einige Warnungen, weil noch nicht alle definierten Signale (z.B. DIS1 und DIS2) benutzt werden. Warnungen können Sie erstmal ignorieren. Echte Fehler (Error) müssen Sie beseitigen bzw. Ihren Assistenten ansprechen.
- Wenn die Berechnungen fertig sind, erscheint rechts ein großes neues Fenster mit der *Device Utilization Summary*. Hier sehen Sie, wieviel Ressourcen des FPGAs benötigt werden. Das Tutorial belegt z.B. 4 LUTs (Wahrheitstabellen, engl. Look-Up Table) der 11.440 LUTs des FPGAs.
- Sie müssen jetzt das Programm *iMPACT* starten, um Ihre Programmierung in den Baustein zu übertragen. Dazu gehen Sie wie folgt vor:



Doppelklicken Sie links auf *Configure Target Device* und bestätigen Sie die auftauchende Warnung mit einem Klick auf *OK*.

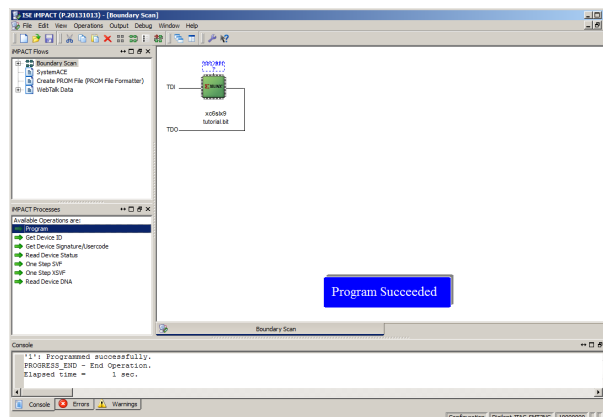
Spätestens jetzt sollten Sie das Board über die USB-Buchse an den PC anschließen.

- Nun erscheint das Fenster des Programms *iMPACT*. In diesem wird die Übertragung zum FPGA konfiguriert:



1. Doppelklicken Sie links auf *Boundary Scan*.
2. Klicken Sie dann oben auf *Initialize Chain*.
3. Anschließend werden Sie gefragt, ob sie ein *configuration file* auswählen wollen, bestätigen Sie dies mit einem Klick auf *yes*.
4. Gehen Sie ggf. in das richtige Verzeichnis und wählen Sie die Binärdatei (*tutorial.bit*) aus. Klicken Sie dann auf *Open*.
5. Nun werden Sie gefragt, ob sie ein *PROM* hinzufügen wollen, was Sie mit einem Klick auf *No* ablehnen.
6. Zum Abschluss bekommen Sie ein weiteres Fenster mit den *Device Programming Properties* angezeigt, welches sie einfach schließen können.

- Links sehen Sie jetzt eine Liste: *Available Operations are:* → *Program*. Doppelklicken Sie auf → *Program*.
- Zum Schluß können Sie mitverfolgen, wie der Baustein programmiert wird.
- Ist der Baustein fertig programmiert erscheint *Programing Succeeded*. Glückwunsch.



WICHTIGER HINWEIS: In diesem Versuch wird die Beschreibungssprache VHDL verwendet. Machen Sie sich mit dieser vertraut! Eine Einführung zu dieser können sie in der folgenden PDF finden:

https://www.uni-ulm.de/fileadmin/website_uni_ulm/iui.inst.050/vorlesungen/sose08/LaborpraktikumEingebetteteSysteme/Material/Crashkurs_VHDL.pdf

1.3. Änderungen der Beschreibung

Die Beschreibung für das Tutorial ist eine Kombination aus einfachen, logischen Verknüpfungen. Es werden jeweils zwei nebeneinander liegende Taster sowohl über *UND* als auch *ODER* verknüpft und auf zwei LEDs ausgegeben:

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

-- Uncomment the following library declaration if using
-- arithmetic functions with Signed or Unsigned values
use IEEE.NUMERIC_STD.ALL;

entity Tutorial is
  port (
    SW      : in  std_logic_vector(7 downto 0);
    LED     : out std_logic_vector(7 downto 0)
  );
end Tutorial;

architecture Struct of Tutorial is
  signal s_LED      : std_logic_vector( 7 downto 0);
begin

  LED <= s_LED;

  s_LED(0) <= SW(0) and SW(1);
  s_LED(1) <= SW(0) or SW(1);
  s_LED(2) <= SW(2) and SW(3);
  s_LED(3) <= SW(2) or SW(3);
  s_LED(4) <= SW(4) and SW(5);
  s_LED(5) <= SW(4) or SW(5);
  s_LED(6) <= SW(6) and SW(7);
  s_LED(7) <= SW(6) or SW(7);

end Struct;
```

Eine ähnliche, sehr häufig verwendete Schaltung ist der Halbaddierer, welcher zwei 1-bit-Signale addiert und sowohl die Summe als auch den Übertrag (engl.: *carry*) ausgibt. Die beiden Gleichungen lauten:

```
Sum <= a xor b;
Carry <= a and b;
```

Aufgabe: Ändern Sie das Tutorial so ab, dass die ungeraden LEDs (1,3,5,7 auf dem Board) die Summe und die geraden LEDs (2,4,6,8 auf dem Board) den Übertrag ausgeben. Generieren Sie das bit-File neu und übertragen Sie dies auf den FPGA. Achten Sie dabei darauf, dass die Nummerierung im Quellcode von 0 bis 7 anstatt von 1 bis 8 läuft!¹

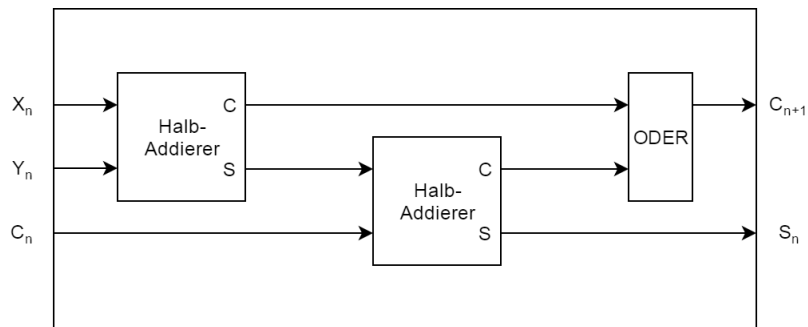
TIPP: Nachdem das bit-File neu generiert wurde, reicht es aus, in *iMPACT* erneut auf \rightarrow *Program* doppelt zu klicken. Dabei kommt nun ein Fenster, dass sich die Datei geändert hat und ob die Datei neu geladen werden soll. Dies bestätigen Sie mit einem Klick auf *yes*.

¹d.h.: Die Zuweisung `LED(0) <= '1'`; bringt die LED 1 auf dem Board zum Leuchten.

2. Logische Verknüpfungen

2.1. Logik-Gleichungen: Der Volladdier

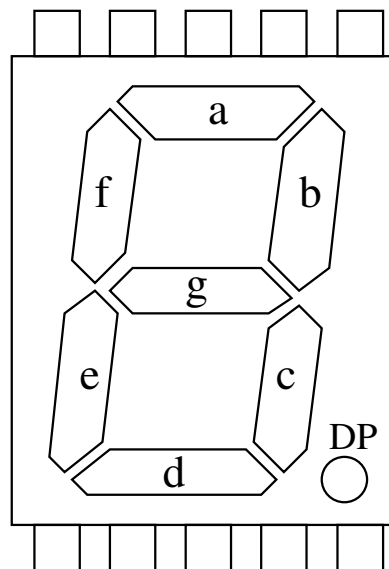
Da der Halbaddierer nur 2 Bit addieren kann, wir aber noch den Übertrag aus der vorherigen Stelle mit berücksichtigen müssen, bauen wir uns nun den Volladdierer. Dieser besteht aus 2 Halbaddierern und einem ODER Gatter:



Aufgabe: Leiten Sie die Gleichungen für S_n und C_{n+1} her. Erweitern Sie die Vorgaben so, dass die Summe der beiden Zahlen (gegeben durch die Eingänge SW und DILSW) berechnet und über die LEDs ausgegeben wird. Die LED 8 soll dabei den höchsten Übertrag anzeigen.

2.2. Wahrheitstabellen: Die 7-Segment-Anzeige

Auf dem Board sind zwei Siebensegmentanzeigen DIS1 und DIS2. Die 7 Segmente² werden wie folgt bezeichnet:



Wir wollen nun die Summe auf den Siebensegmentanzeigen sichtbar machen. Wir haben in der VHDL-Datei eine Funktion vorbereitet, die aber noch aktiviert werden muss. Dazu muss der Aufruf der Funktion in die Beschreibung eingefügt werden.

Beispiel: Mit dieser Zeile werden die unteren 4 Bits von *counter* der Funktion übergeben und das Ergebnis dem Signal *output* zugewiesen:

```
output <= segment7(counter(3 downto 0));
```

²Es werden nur die 7 Segmente a-g verwendet, der Dezimalpunkt wird in den hiesigen Aufgaben nicht verwendet.

Die Funktion ist wie folgt definiert:

```
function segment7 ( bits : in std_logic_vector(3 downto 0))
    return std_logic_vector is
    variable ret : std_logic_vector(6 downto 0);
begin
    case bits is          -- gfedcba
        when x"0" => ret := "01111111";
        when x"1" => ret := "00001110";
        when x"2" => ret := "00000000";
        when x"3" => ret := "00000000";

        when x"4" => ret := "00000000";
        when x"5" => ret := "00000000";
        when x"6" => ret := "00000000";
        when x"7" => ret := "00000000";

        when x"8" => ret := "00000000";
        when x"9" => ret := "00000000";
        when x"a" => ret := "00000000";
        when x"b" => ret := "00000000";

        when x"c" => ret := "00000000";
        when x"d" => ret := "00000000";
        when x"e" => ret := "00000000";
        when x"f" => ret := "00000000";

        -- die nachfolgende Zeile müsst ihr nicht anpassen.
        when others => ret := (others => '0');
    end case;

    return ret;
end function;
```

Beachten Sie: In dem 7-Bit-Ausdruck "01111111" steuert die Ziffer ganz links das Segment g an, die Ziffer ganz rechts steuert Segment a an. Im Beispiel leuchten also alle Segmente außer g und man sieht auf dem Display die Zahl 0. Bei "00001110" leuchten nur die Segmente b und c und man sieht auf dem Display die Zahl 1.

Aufgabe: Erweitern Sie den Code so, dass die unteren 4 Bits der Summe neben der Ausgabe auf den LEDs ebenfalls auf der Anzeige DIS1 und die oberen 4 Bits auf der Anzeige DIS2 ausgegeben werden. Überlegen Sie sich, wie die Muster 2 bis f aussehen müssen, damit auch die entsprechenden Zahlen bzw. Buchstaben angezeigt werden und ändern Sie die Funktion entsprechend ab. Achten Sie darauf, dass die Muster unterscheidbar sind. Eine 8 und ein großes B lassen sich z.B. nicht unterscheiden.

Übersetzen Sie diese Datei und programmieren Sie sie in den FPGA!

3. Verhaltensbeschreibung

Die Beschreibung der Schaltung mittels logischer Verknüpfungen wird sehr schnell kompliziert und unübersichtlich. Daher versucht man, die Beschreibung etwas zu abstrahieren und statt der konkreten Struktur das gewünschte Verhalten zu beschreiben. Die genaue Umsetzung der Beschreibung ist dabei (meist³) uninteressant und wird dem Übersetzer überlassen.

3.1. Binärzähler

Als Beispiel für eine komplexere Verhaltensbeschreibung soll ein Zähler dienen. Dieser besteht aus einem Addierer und Flipflops zum Speichern des aktuellen Wertes. Anstelle der vollständigen Beschreibung des Volladdierers aus dem ersten Versuchsteil soll nun lediglich das Verhalten *addiere 1 dazu* beschreiben werden, was mit der folgenden Zuweisung erreicht wird:

```
counter <= counter + 1;
```

Damit der Zähler aber nicht unkontrolliert zählt, brauchen wir noch Flipflops, die mit einem vorgegebenen Takt die neuen Zählerwerte übernehmen. Dies erreichen wir mit einem getaktetem Prozess, wie ihn die folgende Beschreibung zeigt:

```
name : process(clk)
begin
  if rising_edge(clk) then
    counter <= counter + 1;
  end if;
end process;
```

In diesem Versuchsteil haben wir zwei Zähler, die ineinander verschachtelt sind und mit einem Takt von 20 MHz angesteuert werden. Der erste Zähler *r_counter* wird jeden Takt um 1 erhöht und auf 0 zurück gesetzt, sobald ein vorgegebener Wert *c_prescaler - 1* erreicht wird. Der zweite Zähler *r_LED* wird um 1 erhöht, sobald der erste Zähler zurückgesetzt wird. Somit skaliert *r_counter* die Zählrate von *r_LED*. Der Zählerstand von *r_LED* wird auf die LEDs auf dem Board ausgegeben und kann dort kontrolliert werden.

```
LED <= std_logic_vector(r_LED);

p_counter: process(s_clk20)
begin
  if rising_edge(s_clk20) then
    if s_reset = '1' then
      r_counter <= to_unsigned(0,17);
      r_LED <= to_unsigned(0,8);
    else
      r_counter <= r_counter + 1;
      if r_counter = c_prescaler - 1 then
        r_counter <= to_unsigned(0,17);

        r_LED <= r_LED + 1;
      end if;
    end if;
  end if;
end process;
```

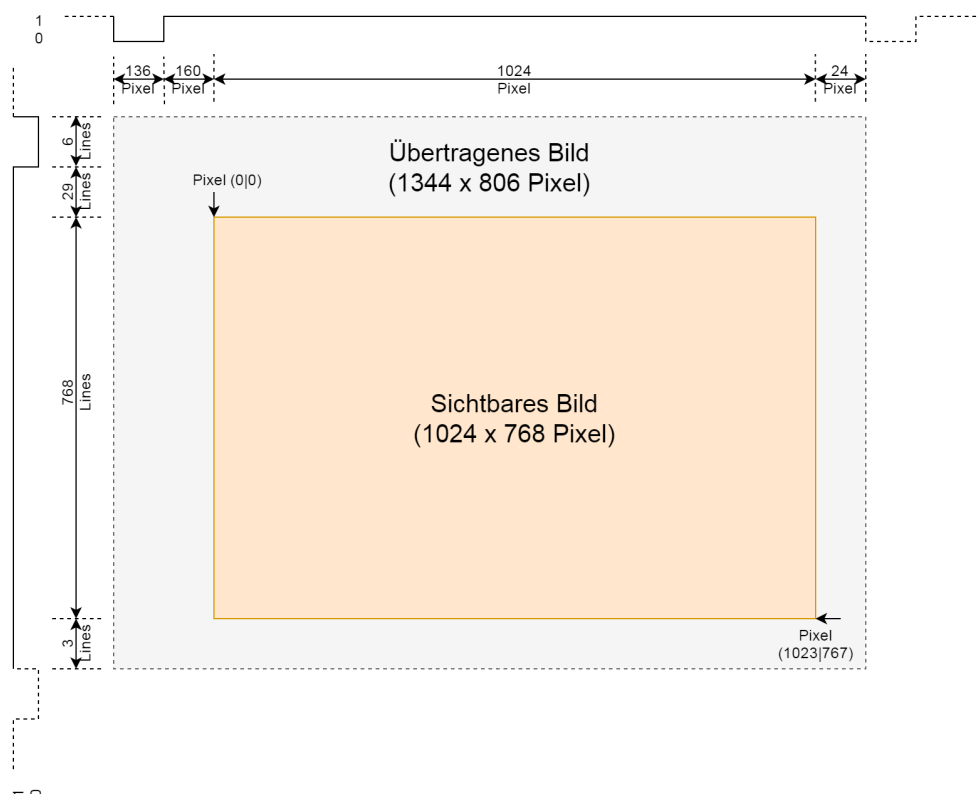
Aufgabe: Übersetzen Sie die Beschreibung und testen Sie diese auf dem FPGA-Board. Was sehen Sie? Messen Sie die Frequenz, mit welcher der Zähler *r_LED* erhöht wird, mit dem Oszilloskop nach. Dazu können Sie das Signal am besten an den Vorwiderständen der LEDs mit dem Tastkopf abgreifen. Wie muss die Beschreibung abgeändert werden, damit die *LED8* mit 1 Hz blinkt?

³Verschiedene Umschreibungen können in unterschiedlichen Verbrauch von Ressourcen enden. Da diese Ressourcen bei modernen FPGAs zwar recht zahlreich, aber noch immer endlich sind, sollte man den Verbrauch im Auge behalten. Zusätzlich kann für eine kleinere Schaltung auch ein kleinerer FPGA ausreichend sein, was sich im Preis niederschlägt.

3.2. Videoausgabe mittels VGA-Schnittstelle

Das Signal für die Videoausgabe besteht aus zwei Synchronisationssignalen sowie der Farbinformation. Die Synchronisationssignale steuern dabei, wann das Ende einer Zeile (horizontale Synchronisation, kurz *hSync*) bzw. das Ende des Bildes (vertikale Synchronisation, kurz *vSync*) erreicht ist. Beide Signale benutzen inverse Logik, sind also aktiv, wenn sie den Wert 0 haben.

Die Technik mit den Synchronisationssignalen stammt noch aus der Zeit der Bildröhren. Da die Elektronenstrahlen noch Zeit zum Stabilisieren brauchten, gibt es neben dem Bereich für Synchronisationspulse noch die so genannte *Schwarzschulter*. Dies ist ein Bereich des Bildes, der zwar mit gesendet wird, aber nicht sichtbar ist. Eine Übersicht über die Synchronisationssignale sowie den Unterschieden zwischen übertragenem und sichtbarem Bild zeigt die folgende Abbildung:



Die Synchronisationssignale können mit Hilfe zweier Zähler erzeugt werden. Dabei zählt der Zähler für den horizontalen Synchronisationspuls mit jedem Takt um 1 hoch und wird beim Erreichen des Zeilenendes wieder auf 0 gesetzt. Der Zähler für den vertikalen Synchronisationspuls wird jeweils am Ende einer Zeile (d.h.: wenn der horizontale Zähler zurück auf 0 gesetzt wird) um 1 erhöht bzw. beim Erreichen der maximalen Zeilenzahl wieder auf 0 gesetzt.

Hinweis: Es gibt eine Reihe von vordefinierten Konstanten, die Sie für die Umsetzung dieser Aufgabe benutzen können. Diese sind in der nachfolgenden Tabelle aufgeführt:

Name	Wert	Beschreibung
<code>c_totalPixelPerLine</code>	1344	Anzahl der Pixel (Takte) pro Zeile des übertragenen Bildes
<code>c_activePixelPerLine</code>	1024	Anzahl der Pixel (Takte) pro Zeile des sichtbaren Bildes
<code>c_syncPixelPerLine</code>	136	Dauer des horizontalen Synchronisationspulses in Pixel (Takten)
<code>c_totalLinesPerFrame</code>	806	Anzahl der Zeilen des übertragenen Bildes
<code>c_activeLinesPerFrame</code>	768	Anzahl der Zeilen des sichtbaren Bildes
<code>c_syncLinesPerFrame</code>	6	Dauer des vertikalen Synchronisationspulses in Zeilen

Aufgabe: Erweitern Sie die Vorgaben so, dass das Signal `r_hSync` den horizontalen Synchronisationspuls erzeugt. Der Puls soll sich dabei am Anfang der Zeile befinden. Verwenden Sie `r_hCounter` als Zähler. Mit welcher Frequenz kommt der Puls (Taktfrequenz beträgt 65 MHz)? Messen Sie die Pulsdauer und Frequenz mit dem Oszilloskop nach!

Aufgabe: Erweitern Sie nun ihre Beschreibung so, dass das Signal `r_vSync` den vertikalen Synchronisationspuls erzeugt. Der Puls soll sich auch hier am Anfang des Bildes befinden. Verwenden Sie `r_vCounter` als Zähler. In wie weit ähnelt dies der Beschreibung zur Erzeugung des horizontalen Synchronisationspulses? Welche Bildrate erwarten Sie?









Sobald die Synchronisationssignale korrekt erzeugt werden, können Sie das FPGA Board mit einem Monitor verbinden. Der Monitor sollte nun das Signal erkennen und ein Bild anzeigen. In diesem gibt es jeweils eine rote Zeile am linken und rechten Rand sowie jeweils eine blaue Zeile am oberen und unteren Rand. Zusätzlich kann durch drücken der Tasten 6-8 die Farbe geändert werden. Die Taste 6 steht dabei für blau, die Taste 7 für grün und die Taste 8 für rot.

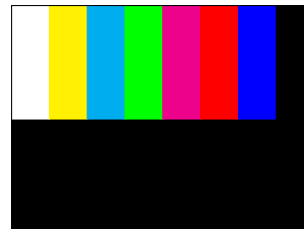
Aufgabe: Wie viele Farbkombinationen lassen sich generieren?

3.3. Bilderzeugung

Nachdem nun die Ansteuerung des Monitors funktioniert, wollen wir uns um den Inhalt des Bildes kümmern. Dazu erweitern wir die Beschreibung aus der letzten Teilaufgabe.

Als erstes soll ein Testbild generiert werden, anhand welchem die richtige Zuordnung der Farben kontrolliert werden kann. Dazu sollen in der oberen Bildhälfte Rechtecke mit den entsprechenden Farbkombination so erstellt werden, dass deren Helligkeit von links nach rechts abnimmt. Die folgende Tabelle zeigt die Farben entsprechend der Helligkeit sortiert:

Farbe	Rot	Grün	Blau	
Weiss	1	1	1	
Gelb	1	1	0	
Cyan	0	1	1	
Grün	0	1	0	
Magenta	1	0	1	
Rot	1	0	0	
Blau	0	0	1	
Schwarz	0	0	0	



Es stehen vordefinierte Funktionen⁴ zur Verfügung, die als Rückgabewert liefern, ob man sich innerhalb eines Rechtecks oder Kreises mit vorgegebenem Mittelpunkt und Größe befindet. Diese können zum Zeichnen von farbigen Blöcken genutzt werden. Ein Aufruf der Rechteckfunktion könnte dabei so aussehen:

```

-- x      y      l      h
if withinRectangle(getPixel(r_hCounter), getLine(r_vCounter), 512, 384, 64, 48) then
-- setze die Farbe, die das Rechteck haben soll, zum Beispiel weiß:
  r_red   <= '1';
  r_green <= '1';
  r_blue  <= '1';
end if;
```

Dieser Aufruf zeichnet in der Mitte des Bildes ein weißes Rechteck mit der Breite von 64 Pixel und der Höhe von 48 Zeilen.

Aufgabe: Fügen Sie die Beschreibung zum Zeichnen der Farbböcke ein und überprüfen Sie dies auf dem Bildschirm. Die Blöcke sollen alle die gleiche Breite haben. Was passiert nun mit der unteren Bildhälfte?

3.4. Bewegte Bilder

Abschließend soll noch ein sich bewegendes Ball (Kreis) hinzugefügt werden. Dazu verwenden wir 2 weitere Zähler(`r_posX`, `r_posY`) anstelle eines festen Mittelpunktes, welche pro Bild um eine feste Schrittweite (für den Anfang erstmal um 1) weiterzählen. Die Richtung (ob die Zähler erhöht oder verringert werden) soll dabei von den beiden Signalen `r_dirX` und `r_dirY` abhängen. Hat `r_dirX` den Wert 1, so soll `r_posX` erhöht werden. Hat `r_dirX` dagegen den Wert 0, so soll `r_posX` entsprechend verringert werden. Gleiches gilt für `r_dirY` und `r_posY`.

⁴Deren Definition kann im Anhang auf Seite 15 nachgeschlagen werden.

Aufgabe: Erweitern Sie ihre Beschreibung aus dem vorherigen Versuchsteil so, dass sich ein weißer Kreis über das Bild bewegt und scheinbar an den Bildrändern abprallt. Dieser soll sich diagonal mit 1 Pixel pro Bild bewegen, also seine Position gleichzeitig um 1 Pixel horizontal und um 1 Pixel vertikal verändern. Lassen Sie diesen in der Mitte des Bildes starten. Wann sollte die Position des Kreises aktualisiert werden? Wann muss die Richtung geändert werden?

V. Anhang

1. Übersicht Konstanten und Funktionen für die Videoausgabe

Konstanten:

Name	Wert	Beschreibung
c_totalPixelPerLine	1344	Anzahl der Pixel (Takte) pro Zeile des übertragenen Bildes
c_activePixelPerLine	1024	Anzahl der Pixel (Takte) pro Zeile des sichtbaren Bildes
c_syncPixelPerLine	136	Dauer des horizontalen Synchronisationspulses in Pixel (Takten)
c_totalLinesPerFrame	806	Anzahl der Zeilen des übertragenen Bildes
c_activeLinesPerFrame	768	Anzahl der Zeilen des sichtbaren Bildes
c_syncLinesPerFrame	6	Dauer des vertikalen Synchronisationspulses in Zeilen

Funktionen:

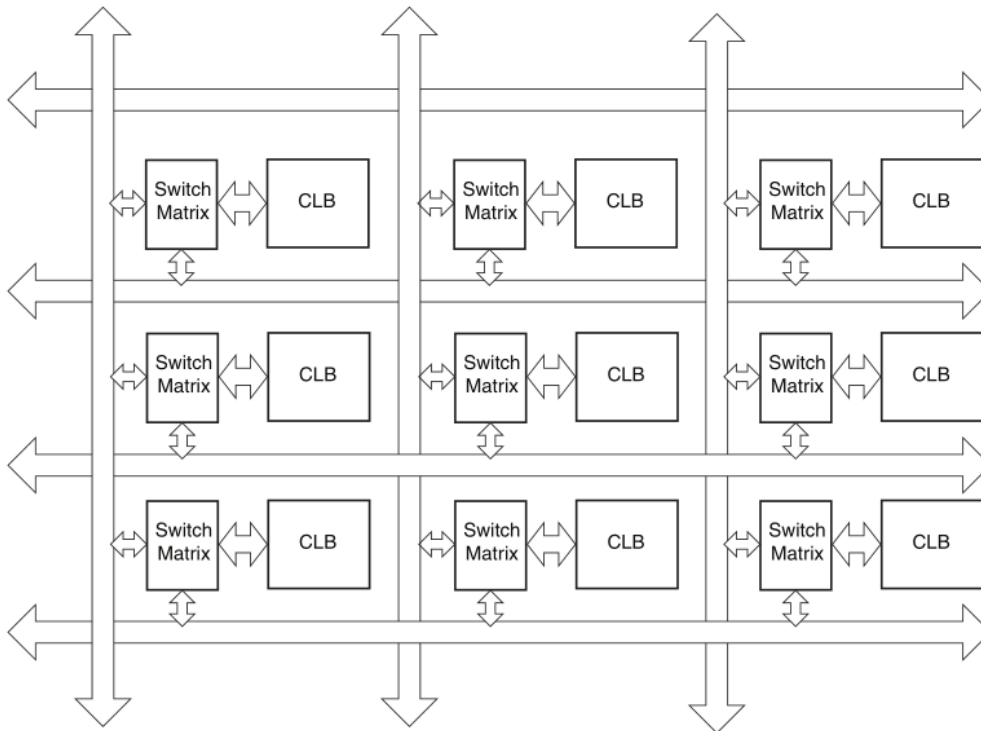
Name	# Parameter	Parameter Name	Parameter Typ	Beschreibung
getPixel	1		integer	Berechnet die Position(Pixel) im sichtbaren Bild aus der Position im übertragenem Bild
		counter	unsigned	Zähler für den horizontalen Synchronisationspuls
getLine	1		integer	Berechnet die Position(Zeile) im sichtbaren Bild aus der Position im übertragenem Bild
		counter	unsigned	Zähler für den vertikalen Synchronisationspuls
withinRectangle	6		boolean	Prüft, ob die übergebenen Koordinaten (currentX currentY) innerhalb eines Rechtecks mit dem Mittelpunkt (centerX centerY) sowie der angegebenen Breite und Höhe liegt
		currentX	integer	aktueller Pixel, vorzugsweise die Ausgabe von getPixel()
		currentY	integer	aktueller Zeile, vorzugsweise die Ausgabe von getLine()
		centerX	unsigned ⁵	Pixel des Mittelpunktes
		centerY	unsigned ⁵	Zeile des Mittelpunktes
		width	integer	Breite des Rechtecks in Pixeln
withinCircle	5		boolean	Prüft, ob die übergebenen Koordinaten (currentX currentY) innerhalb eines Kreises mit dem Mittelpunkt (centerX centerY) sowie dem angegebenen Durchmesser liegt
		currentX	integer	aktueller Pixel, vorzugsweise die Ausgabe von getPixel()
		currentY	integer	aktueller Zeile, vorzugsweise die Ausgabe von getLine()
		centerX	unsigned ⁵	Pixel des Kreismittelpunktes
		centerY	unsigned ⁵	Zeile des Kreismittelpunktes
		diameter	integer	Durchmesser des Kreises in Pixeln

⁵Es gibt auch eine Variante der Funktion, bei der diese Parameter den Typ *integer* haben

2. FPGA-Datenblätter und Architektur

Ausführliche Datenblätter zu den FPGAs findet man auf den Internetseiten des Herstellers: www.xilinx.com

Die folgende Abbildung⁶ zeigt die prinzipielle Architektur des FPGAs:



ug384_29_012710

Im Bild sieht man die regelmäßige Struktur, wie sie für FPGAs üblich sind. Diese teilt sich auf in:

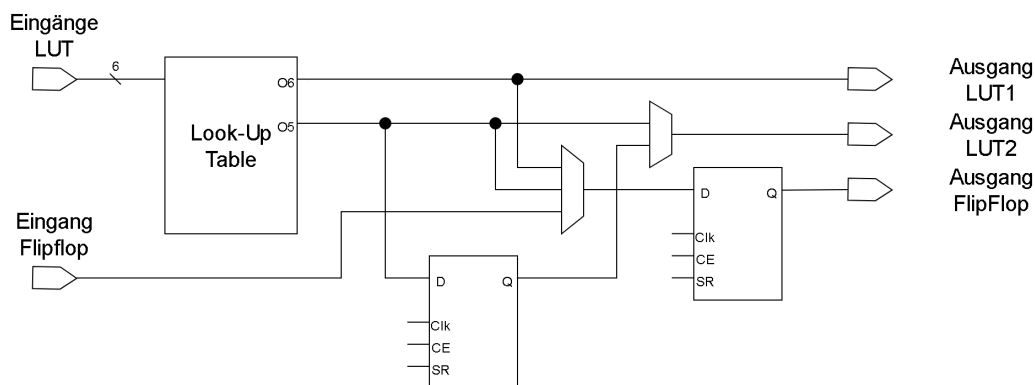
- Verbindungswege, welche den ganzen FPGA vertical und horizontal durchziehen,
- Verbindungspunkte (*Switching Matrix*), welche verticale und horizontale beliebig Leitungen verbinden können und
- Funktionseinheiten (hier als Beispiel *Configurable Logic Block*, kurz CLB), in welchen das Verhalten umgesetzt wird.

Diese Funktionseinheiten können dabei auf ganz unterschiedliche Aufgaben spezialisiert sein:

- Ein-/Ausgabe: Pins mit entsprechenden Treiberstufen bzw. Verstärkern, De-/Serialisierer, ...
- Takterzeugung/-verbreitung: Frequenzsynthese, Jitter-Reduktion, Treiberstufen zur Verteilung auf dem FPGA
- Logik (CLBs): Wahrheitstabellen, Flipflops, zusätzliche Elemente zum Umsetzen von erweiterten Logikfunktionen sowie einfachen Speicherelemente
- Speicher: Blöcke an zusammenhängendem Speicher, mit denen auch größere Speicher gebaut werden können
- Berechnungen: Auf die Verarbeitung von digitalen Signalen spezialisierte Blöcke, welche Addition und Multiplikation beschleunigen.

⁶Diese entstammen dem User Guide Nr. 384, welcher hier zu finden ist:
https://www.xilinx.com/support/documentation/user_guides/ug384.pdf

Die *Configurable Logic Block* sind dabei aus 2 identischen Blöcken aufgebaut, welche *Slice* genannt werden. Diese Slices bestehen wiederum aus 4 Gruppen mit jeweils einer Wahrheitstabelle (*LookUp Table*, LUT) und 2 Flipflops. Der Aufbau einer solchen Gruppe ist hier vereinfacht dargestellt:



- Von links kommen die Eingangssignale. Die Daten werden in Gruppen gebündelt, während die Steuersignale für das gesamte Slice gemeinsam sind.
- Die Daten können entweder in einer Wahrheitstabelle (*LookUp Table*, LUT) mit 6 Eingängen (alternativ auch als 2 Tabellen mit jeweils 5 Eingängen nutzbar) verknüpft oder an diesen vorbei geführt werden.
- Hinter den Wahrheitstabellen folgen diverse Multiplexer, welche die Wege der Daten umschalten können. Diese können genutzt werden, um Funktionen mit mehr als 6 Eingängen zu bilden (z.B.: 16:1 Multiplexer).
- An den Ausgängen stehen jeweils 2 Flipflops pro Wahrheitstabelle zur Verfügung, in welchen die Ergebnisse gespeichert werden können. Für den Fall, dass eine Wahrheitstabelle nicht ausreicht, um die Logik umzusetzen, können Zwischenergebnisse auch an den Flipflops vorbei zu weiteren Slices geführt werden