6. Blinkende LEDs

In diesem Projekt geht es nicht bloß um eine einfache Blinkschaltung, wie man das zum Beispiel vom Arduino-Blink-Sketch als klassisches Hello World-Programm kennt. Es wird weitaus spannender. Wir starten mit einem weiteren sehr wichtigen Teilbereich, bei dem es um einen Prozess (Process) geht.

Was ist ein Prozess?

Bisher wurden im Architecture-Block lediglich Anweisungen hinterlegt, die dafür sorgten, dass die logischen Strukturen gleichzeitig (englisch concurrent) ausgeführt wurden. In VHDL gibt es jedoch zusätzlich die Möglichkeit, bestimmte Anweisungen hintereinander, also sequenziell abzuarbeiten. Das VHDL-Konstrukt, das dieses Verhalten erlaubt, wird Process genannt.

Ein Prozess ist der klassischen Programmiersprache sehr ähnlich, wo der Code innerhalb der Prozessanweisung sequenziell ausgeführt wird. Die Prozessanweisung wird im Architecture-Block deklariert, sodass zum Beispiel zwei verschiedene Prozesse gleichzeitig ausgeführt werden können. Die allgemeine Syntax lautet wie folgt:

```
[process_label :] process[(sensitivity_list)]
  -- declarations
begin
  -- sequential statements
end process [process_label];
```

Man kann den Process mit einem Label, also einem aussagekräftigen Namen versehen, was jedoch optional ist. Hinter dem Schlüsselwort process wird eine optionale Parameterliste genannt, die Sensitivity List genannt wird. Dort werden Signale aufgeführt, die bei einer Zustandsänderung des jeweiligen Signals der Liste den Prozess aufruft. Jeder Prozess tut solange nichts, bis er durch eine Änderung angestoßen wird. Es handelt sich in diesem Fall um einen getakteten Prozess. Diese Änderungen stellen eine Art Trigger für den Prozess dar, um zum Beispiel eine neue Berechnung durchzuführen. Es ist möglich, mehrere Prozesse zu definieren, die auch gleichzeitig aktiv sein können. Beginnen wir dieses Thema mit einem einfachen Beispiel, mit der Definition eines einzigen Prozesses.

Der Prozess wird angestoßen

Was könnte für das Anstoßen eines Prozesses im einfachsten Fall verwendet werden und was kommt als Trigger für einen Pegelwechsel in Frage? Genau, ein Taster, der entweder gedrückt oder auch wieder losgelassen wird:

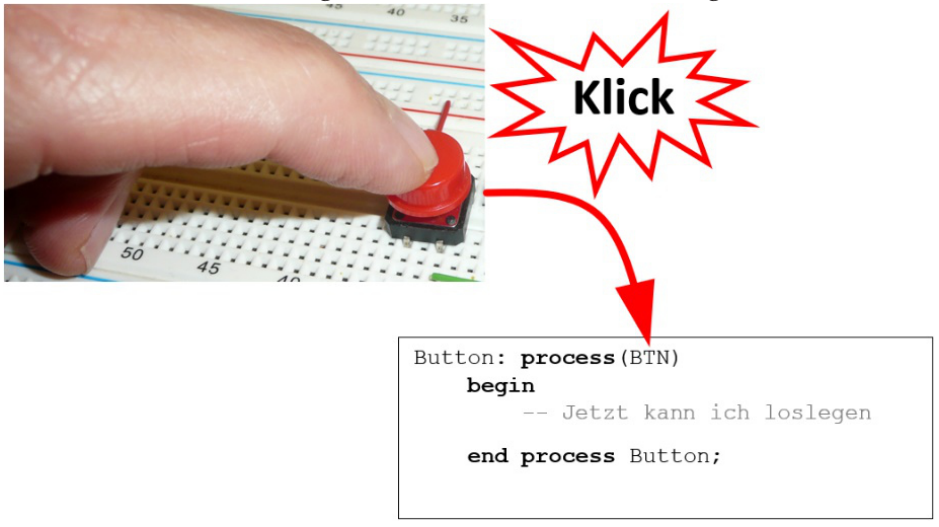

Abbildung 1: Das Triggern des Prozesses

Also sehen wir uns dazu ein einfaches Beispiel an, bei dem ich den Taster auf dem MAX1000-Board nutzen möchte, um damit die acht LEDs in einem bestimmten Muster abhängig vom Tasterstatus leuchten zu lassen. Es soll der Einfachheit halber zwischen den beiden Mustern (Patterns) hin und her geschaltet werden:

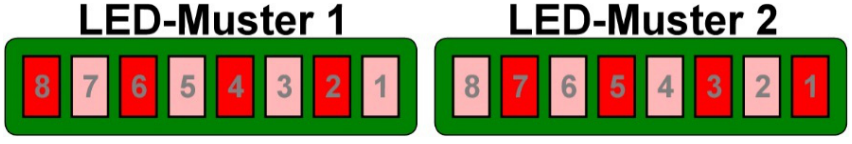

Abbildung 2: Die beiden LED-Patterns

Nun reicht es aber im einfachsten Fall nicht aus, dass der Prozess beim Drücken des Tasters aufgerufen wird, sondern es muss innerhalb der Prozessverarbeitung ermittelt werden, welchen Status der Tasten besitzt. Er kann die beiden Werte 0 oder 1 haben, was über eine bestimmte Codestruktur ausgewertet werden muss. Wer sich mit der Programmierung schon ein wenig auskennt, dem ist die if-then-else-Struktur bekannt, was übersetzt so viel wie „wenn-dann-sonst“ bedeutet. Der else-Zweig ist hierbei optional, wird aber dennoch benötigt. In einem Flussdiagramm würde diese Abfrage wie folgt aussehen:

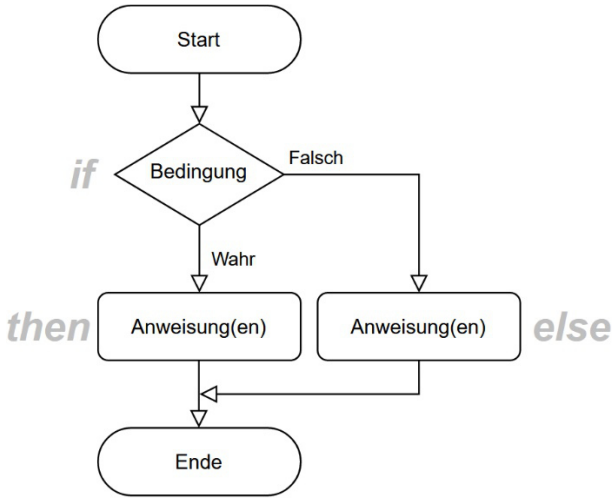

Abbildung 3: Das Flussdiagramm für if-then-else

Eine if-Anweisung kann nur innerhalb eines Prozesses angewendet werden, weil es innerhalb von if-then-else zur sequenziellen Abarbeitung von Befehlen kommt. Im Architecture-Block ohne einen Prozess könnte eine if-Anweisung niemals einen Sinn ergeben, weil dort alles gleichzeitig erfolgt. Sehen wir uns jetzt den entsprechenden Code zur Realisierung des Vorhabens an. In VHDL läuft wieder eine Blockbildung ab:

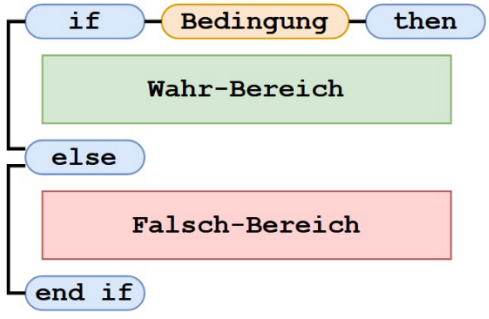

Abbildung 4: Die if-then-else Blockbildung in VHDL

Anfangs erfolgt eine Bewertung des Wahrheitsgehalts des Ausdrucks. Liefert die Bewertung ein logisches wahr (true) zurück, wird der Code zwischen then und else ausgeführt. Kommt es zu einer Bewertung, die ein logisches falsch (false) zurückliefert, wird der Code zwischen else und end if ausgeführt. Ich wende diese Form der if-Abfrage im kommenden Beispiel an. Zur Vervollständigung der Thematik möchte ich an dieser Stelle noch andere if-Abfragen kurz zeigen.

Die if-then-Anweisung

Bei der if-then-Anweisung wird bei der Erfüllung der Bedingung wahr nur ein Block mit Anweisungen ausgeführt. Wird die Bedingung nicht erfüllt (falsch), kommt es zu keiner Ausführung von Anweisungen innerhalb dieser Struktur:

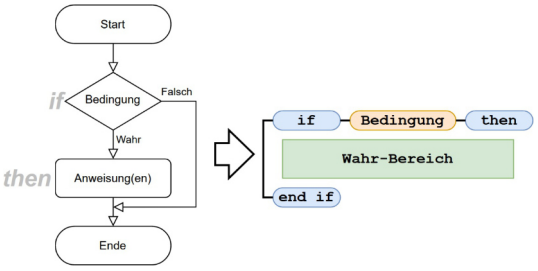

Abbildung 5: Die if-then-Anweisung

Die if-then-elsif-Anweisung

Bei der if-then-elsif-Anweisung (Achtung: else wird ohne e am Ende geschrieben) kommt es zu einer kaskadierenden Abfrage von Bedingungen. Wird die erste nicht erfüllt, kommt es zur Bewertung der zweiten und so weiter und so fort. Wird keine Bedingung erfüllt, kommt es zur Ausführung des else-Blocks. Wird jedoch eine Bedingung erfüllt, werden die entsprechenden Anweisungen ausgeführt und der if-then-elsif-Block unmittelbar verlassen, sodass keine weiteren folgenden Bedingungen auf ihren Wahrheitsgehalt hin untersucht werden:

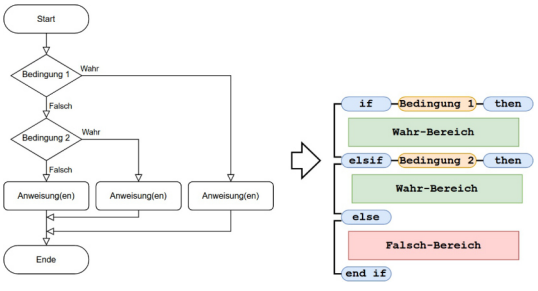

Abbildung 6: Die if-then-elsif-Anweisung - mit zwei Bedingungen

Abschließend eine Zusammenfassung der Merkmale einer if-Anweisung:

- Sie darf nur innerhalb eines Prozesses zum Einsatz kommen.
- Sie kann mehrere Bedingungen verarbeiten.
- Sie kann als Ergebnis mehrere Anweisungen ausführen.

Der Code zur Process-Steuerung

Ich verwende zur Ansteuerung der acht LEDs wieder das Array, weil es darüber viel einfacher ist. Und hier wird die if-then-else-Anweisung genutzt, denn es muss entschieden werden, was passieren soll, wenn der Taster gedrückt und wieder losgelassen wird:

```
01 library IEEE;
02 use IEEE.std_logic_1164.all;
03 use IEEE.numeric_std.all;
04
05 entity Kap06_Button_Pattern is
06     port(
07         BTN : in     std_logic;
08         LED : out std_logic_vector(7 downto 0)
09     );
10 end entity Kap06_Button_Pattern;
11
12 architecture rtl of Kap06_Button_Pattern is
13 begin
14     Button: process(BTN)
15     begin
16         if BTN = '1' then
17             LED <= "10101010"; -- LED-Pattern 1
18         else
19             LED <= "01010101"; -- LED-Pattern 2
20         end if;
21     end process Button;
22 end architecture rtl;
```

Listing: Kap06_Button_Pattern.vhd - Die Process-Steuerung der LEDs

Die Einstellungen im Connect and Compile-Dialogfenster sehen wie folgt aus:

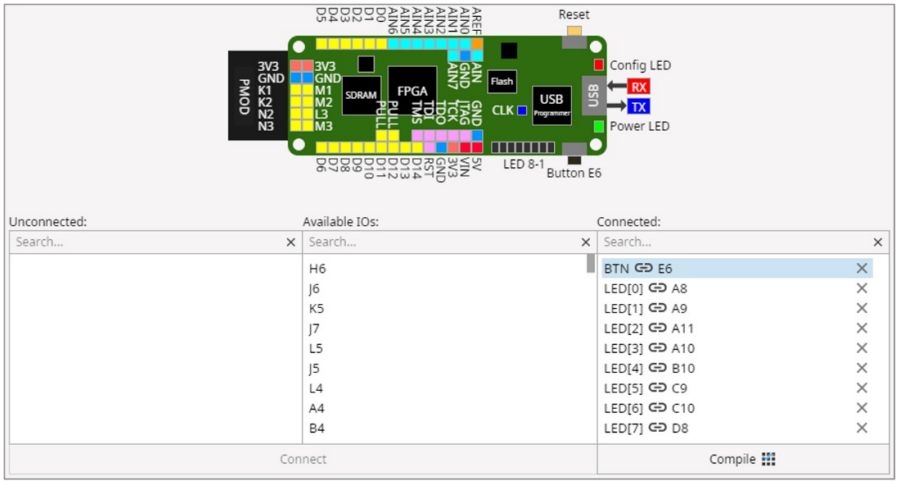

Abbildung 7: Das Mapping für die Process-Steuerung der LEDs

Nach dem Download des Codes sollte nach unserem Wissenstand also welches LED-Muster zu sehen sein? Muster 1 oder Muster 2? Wir erinnern uns, wie das mit dem Pullup-Widerstand am besagten Taster ist: Bei nicht gedrücktem Taster wird eine logische 1 zurückgeliefert, die in der if-then-Abfrage dazu führt, dass das LED-Muster 1 zu sehen ist, wie es im Bit-String "10101010" hinterlegt ist. Wird der Taster losgelassen, kommt es zur Ausführung des else-Blocks und das LED-Muster 2 "01010101" wird angezeigt. Möchte man das Verhalten umkehren, um den Pullup-Widerstand mit seinem vorgegebenen Logikpegel entgegenzuwirken, muss man ... stopp, ich hatte die Lösung schon einmal genannt.

Ein Lauflicht

Im nächsten Schritt möchte ich die genannten Grundlagen dazu nutzen, ein Lauflicht zu implementieren. Zuerst werden wir mit einer manuellen Version über das Drücken eines Tasters starten und dann im nächsten Schritt das Ganze automatisch ablaufen lassen. Dieser Automatismus ist möglich, weil sich auf dem Board ein Clock-Generator befindet, den wir anzapfen können. Es wird also wieder spannend.

Ein manuelles Lauflicht

Ein manuelles Lauflicht ist zwar nicht der Bringer, doch an diesem Beispiel lassen sich sehr gut ein paar essenzielle Dinge verdeutlichen. Ziel ist es, die acht LEDs auf dem MAX1000-Board so geschickt anzusteuern, dass es aussieht, als würde sich beim Druck auf den kleinen Taster eine leuchtende LED bewegen. Ich möchte zu Beginn die LED1 rechts außen aufleuchten lassen. Solange der Taster nicht gedrückt wird, soll auch nichts weiter geschehen. Wenn er dann gedrückt wird, soll LED1 erlöschen und LED2 links davon aufleuchten. Bei jedem weiteren Tastendruck soll die Ansteuerung der LED eine Position nach links wandern.

Was ist ein Shift-Register?

Ein Shift-Register, deutsch Schieberegister, ist eine Schaltung aus mehreren in Reihe geschalteten Flipflops, die bei jedem Takt ihren Speicherinhalt ein Flipflop weiterschieben.

Dazu sind ein paar neue Grundlagen erforderlich. Da wir das Standardsignal für die LEDs, das wir bisher im Entity-Block als out deklariert haben, nicht so ändern können, um da die Bits zu verschieben, nutzen wir ein zweites Signal,

das ich Shift-Register (im Code: shift_reg) nenne. Was ist eigentlich ein Shift-Register?

Ich sprach gerade von einem zusätzlichen Signal, dessen Inhalt sich wie eine Reihe von hintereinander geschalteten Flipflops verhalten soll. Also wird dieses Signal am besten wieder ein Array sein, das aus genau so vielen Elementen besteht wie unser altbekanntes LED-Array. Die allgemeine Syntax einer Signal-Deklaration sieht wie folgt aus, was wir schon im Entity-Block gesehen haben:

```
signal signal_name : <Type> [:= init_value];
```

Nun sollten wir uns überlegen, wo denn dieses Signal im Code am besten unterzubringen ist. Wir erinnern uns an das VHDL-Kapitel mit der Aufzählung der unterschiedlichen Strukturen. Hier eine kurze Auffrischung. Ich habe den für uns relevanten Bereich rot markiert:

```
architecture architecture_name of entity_name is
-- Declarations region
-- ...
begin
    -- Architecture region
    -- ...
end [architecture] [architecture_name];
```

Im Architecture-Block zwischen den beiden Schlüsselwörtern is und begin können die Signal-Deklarationen eingefügt werden und so machen wir es auch mit unserem Signal:

```
signal shift_reg : std_logic_vector (7 downto 0) := "00000001";
```

Somit liegt also ein Array mit den folgenden Elementen und deren Inhalten vor, die 1 an Index-Position 0 ist rot markiert:

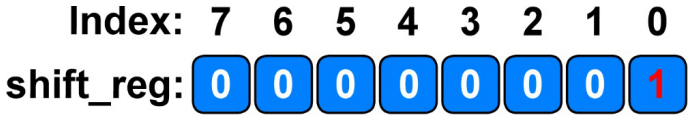

Abbildung 8: Das Schieberegister mit seiner Initialisierung

Jetzt ist es ein Leichtes, den Inhalt dieses Signals dem LED-Signal zuzuweisen, das über die gleiche Dimensionierung verfügt. Man könnte Folgendes schreiben:

```
LED <= shift_reg;
```

Wenn wir jetzt die 1 im Schieberegister an Indexposition 0 nach links zu Indexposition 1 wandern lassen und dann eine erneute Zuweisung an das LED-Signal vornehmen, erkennt man, wie das Lauflicht realisiert werden kann. Das Ganze müsste durch irgendeine Steuerung in Gang gesetzt werden. Wir verwenden dafür den Taster in Verbindung mit dem schon gezeigten Prozess. Der Befehl, der das Wandern der 1 bewirkt, könnte wie folgt aussehen; wir werden sehen, ob er korrekt arbeitet:

```
shift_reg <= shift_reg(6 downto 0) & '0';
```

Auf den ersten Blick sieht das etwas merkwürdig aus, doch sehen wir uns das wieder im Detail an. Dem Schieberegister shift_reg wird sein eigener Inhalt zugewiesen, doch nicht komplett, sondern nur von Indexposition 6 bis 0, sozusagen ein verkleinerter und verschobener Bereich. Damit aber die ursprüngliche Größe des Arrays mit acht Elementen wieder erreicht wird, wird durch den &-Operator (nicht zu verwechseln mit einer UND-Verknüpfung!) eine 0 am rechten Ende angehängt, sodass in Summe wieder acht Elemente erreicht werden. Auf der folgenden Abbildung ist das Verhalten etwas deutlicher zu sehen:

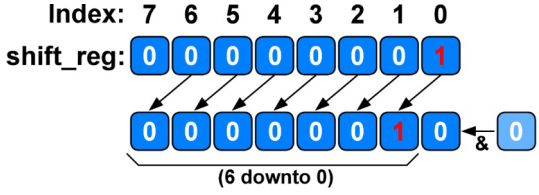

Abbildung 9: Die Array-Zuweisung und Erweiterung

Würde man die einzelnen Elemente schrittweise neu zuweisen, könnte das auch wie folgt aussehen:

```
shift_reg(7) := shift_reg(6);
shift_reg(6) := shift_reg(5);
shift_reg(5) := shift_reg(4);
shift_reg(4) := shift_reg(3);
shift_reg(3) := shift_reg(2);
shift_reg(2) := shift_reg(1);
shift_reg(1) := shift_reg(0);
shift_reg(0) := '0';
```

Doch was würde passieren, wenn die letzte 1 links aus dem Array-Index 7 rausfliegen würde? Sie wäre weg, und da auf der rechten Seite nur Nullen angefügt werden, hätten wir es mit einem Einmal-Lauflicht zu tun, was im Anschluss dunkel wäre. Diese Funktion würde einer sogenannten Shift-Operation gleichkommen. Das wollen wir aber nicht! Der Code müsste also so ange-

passt werden, dass die auf der linken Seite hinausgeschobene 1 wieder auf der rechten Seite hereingeschoben wird. Die gewünschte Funktion würde einer sogenannten Rotations-Operation gleichkommen. Der angepasste Code sieht wie folgt aus:

Jetzt wird auf der rechten Seite keine 0 angefügt, sondern immer der Inhalt des Array-Werts auf der linken Seite mit dem Index-Wert 7. An dieser Stelle

```
shift_reg <= shift_reg(6 downto 0) & shift_reg(7);
```

sollte ich noch kurz auf den Verkettungsoperator durch das kaufmännische Und & eingehen. Dieser kann verwendet werden, um zwei oder mehrere Elemente miteinander zu verbinden beziehungsweise zu verketten (concatenate). Da VHDL stark typisiert ist, müssen alle involvierten Werte für eine Verkettung vom gleichen Typ sein. Außerdem muss das Ergebnis der Verkettung genau der Breite der verketteten Eingangssignale entsprechen, wie wir das mit den unterschiedlich breiten Arrays gesehen haben und durch das Anhängen eines zusätzlichen Elements am rechten Rand wieder ausgeglichen und korrigiert werden.

Um das nun zu implementieren, muss ein weiterer wichtiger Aspekt beachtet werden. Würde man den Code in der jetzigen Form in den Process-Block schreiben, käme es zum Aufruf der Rotationsfunktionalität, solange der Taster gedrückt würde, was aber nicht erwünscht ist. Die LED soll je Tastendruck unabhängig von dessen Dauer nur um eine Position wandern. Hier kommt die Flankensteuerung ins Spiel. Was können wir uns darunter vorstellen? Zuerst müssen wir uns darüber klar sein, was überhaupt eine Flanke in Bezug auf ein Signal ist.

Was ist eine Flanke?

Bei digitalen Signalen sind Signalflanken die Übergänge zwischen den beiden Signalzuständen HIGH und LOW.

Das bringt uns zu der eigentlich gestellten Frage.

Was ist eine Flankensteuerung?

Bei der Bewertung einer Flanke wird ermittelt, ob ein Signal den Zustand zum vorherigen Programmzyklus verändert hat. Wenn das Signal den Pegel von LOW auf HIGH wechselt, liegt eine positive Flanke vor. Bei einer Signalveränderung von HIGH auf LOW liegt eine negative Flanke vor.

Sehen wir uns dazu die genannten Pegelwechsel noch einmal in der folgenden Abbildung an:

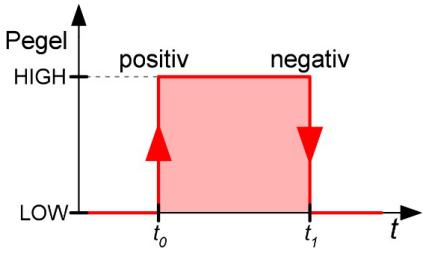

Abbildung 10: Eine positive und negative Flanke

Zum Zeitpunkt t0 wechselt der Pegel von LOW auf HIGH und stellt damit eine positive Flanke dar. Etwas später, zum Zeitpunkt t1 wechselt der Pegel von HIGH auf LOW, was eine negative Flanke bedeutet. Diese beiden Ereignisse können im Code berücksichtigt werden. Hier die allgemeine if-then-Abfrage, ob es sich um eine steigende Flanke (rising_edge) oder eine fallende Flanke (falling_edge) handelt.

```
Button: process(BTN)
    begin
        if rising_edge|falling_edge (BTN) then
            -- Tue etwas
        end if;
    end process Button;
```

Mit diesen Informationen ist es nun möglich, den endgültigen Code zu schreiben:

```
01 library IEEE;
02 use IEEE.std_logic_1164.all;
03 use IEEE.numeric_std.all;
04
05 entity Kap06_Button_Schiftregister is
06     port(
07         BTN : in  std_logic;
08         LED : out std_logic_vector(7 downto 0)
09     );
10 end entity Kap06_Button_Schiftregister;
11
12 architecture rtl of Kap06_Button_Schiftregister is
13     signal shift_reg : std_logic_vector (7 downto 0) := "00000001";
14 begin
15     Button: process(BTN)
16     begin
17         if rising_edge(BTN) then
18             shift_reg <= shift_reg(6 downto 0) & shift_reg(7);
19         end if;
20     end process Button;
21     LED <= shift_reg;
22 end architecture rtl;
```

Listing: Kap06_Button_Shiftregister.vhd - Die Schieberegister-Steuerung

Nun ist wiederum zu beachten, dass der Taster über einen Pullup-Widerstand im nicht gedrückten Zustand einen HIGH-Pegel liefert und im gedrückten einen LOW-Pegel. Wie verhält es sich dann in meinem Fall mit der Pegelabfrage in Zeile 17?

Doch Stopp! Beim Testen der Schaltung prellt der Taster, wie auch im Taster-Kapitel erwähnt. Beim Drücken des Tasters springt die leuchtende LED manchmal nicht nur um eine, sondern gleich um mehrere Positionen. Ein eindeutiges Zeichen für ein sogenanntes Bouncing. Da muss noch ein Debouncing her, das wir uns etwas später ansehen werden.

Der RTL-Viewer für das Schieberegister

Sehen wir uns den RTL-Viewer in Quartus Prime Lite an:

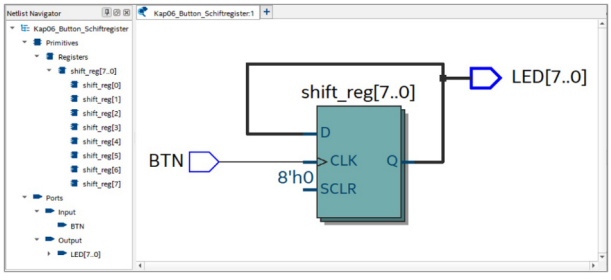

Abbildung 11: Das Ergebnis des Schieberegisters im RTL-Viewer von Quartus Prime Lite

Hier ist wunderbar das taktgesteuerte D-Flipflop zur Realisierung eines Schieberegisters zu sehen, auf das im Flipflop-Kapitel detaillierter eingegangen wird. Ich möchte an dieser Stelle nur so viel sagen, dass bei jeder ansteigenden Taktflanke, die durch den BTN-Taster ausgelöst wird, das Signal von einem zum nächsten Schieberegister weitergeleitet wird.

Ein manuelles Lauflicht mit Reset

Im nächsten Beispiel möchte ich den vorherigen Code mit einer Reset-Funktionalität ausstatten. Dazu wird ein zweiter Taster benötigt und ich möchte diesen mit einem Taster des Discoveryboards verbinden. Doch zuvor einige Grundüberlegungen. Der bisherige Prozess besitzt in seiner Sensitivity List lediglich den Taster, der das Schieberegister beeinflusst. Nun soll noch ein weiterer hinzukommen, denn der Taster für den Reset soll ebenfalls den Prozess aufrufen, um dort entsprechend abgefragt zu werden. Die Sensitivity List kann erweitert werden, indem man ein Komma hinter das letzte Signal setzt und dann den Namen des neuen Signals anfügt. Werfen wir zunächst einen Blick auf den Schaltplan:

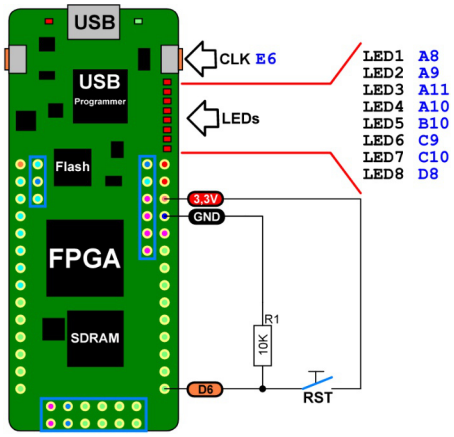

Abbildung 12: Der Schaltplan zur Ansteuerung des Schieberegisters mit Reset-Taster

Folgender GPIO-Pin wird angewendet:

GPIO-Pin	MAX1000-Signal	Modus	Signalbezeichnung
D6	L12	in	CLR

Tabelle: Der verwendete GPIO-Pin

Und nun der Code, der den zusätzlichen Taster RST abfragt, um darüber das Schieberegister mit seinem Start-Pattern zu initialisieren:

```
01 library IEEE;
02 use IEEE.std_logic_1164.all;
03 use IEEE.numeric_std.all;
04
05 entity Kap06_Shiftregister_Reset is
06     generic(BITS : integer := 8); -- Anzahl der Bits
07     port(CLK, CLR : in std_logic := '0';
08         LED : out std_logic_vector (BITS-1 downto 0));
09 end entity Kap06_Shiftregister_Reset;
10
11 architecture rtl of Kap06_Shiftregister_Reset is
12     signal shift_reg  : std_logic_vector (BITS-1 downto 0):=
"00000001";
13 begin
14     process(CLR, CLK)
15     begin
16         if CLR = '1' then
17             shift_reg  <= "00000001";
18         elsif rising_edge(CLK) then
19             shift_reg  <= shift_reg (6 downto 0) & shift_reg (7);
20         end if;
21     end process;
22     LED <= shift_reg ;
23 end architecture rtl;
```

Listing: Kap06_Button_Shiftregister_Reset.vhd - Schieberegister-Steuerung mit Reset-Taster

In Zeile 14 ist die Sensitivity List mit den beiden Signalen CLR und CLK zu sehen, die dann innerhalb des Process-Blocks innerhalb der if-then-elsif-Abfrage entsprechend ausgewertet werden. Wird der CLR-Taster gedrückt, wird Zeile 17 ausgeführt und das Schieberegister auf seinen Startwert gesetzt. Wird der CLK-Taster gedrückt (steigende Flanke), wird das Schieberegister in Zeile 19 um eine Position nach links verschoben.

Die Taktgenerierung

Kommen wir zur Realisierung eines voll automatischen Lauflichts, das über den internen Takt (Clock) getriggert wird. Zur Einführung in die Clock-Thematik wollen wir erst einmal eine einzige LED in unterschiedlichen Taktraten blinken lassen. Haben wir das verstanden, können wir fortfahren mit der eigentlichen Aufgabe der Realisierung eines automatischen Lauflichts. Werfen wir noch einmal einen Blick auf das MAX1000-Board und sehen, was sich dort noch für Komponenten befinden, die noch nicht besprochen wurden. Wir sehen, dass sich etwas rechts oberhalb des Flash-Speichers eine Oszillatorkomponente befindet, die für die Taktgenerierung zuständig ist:

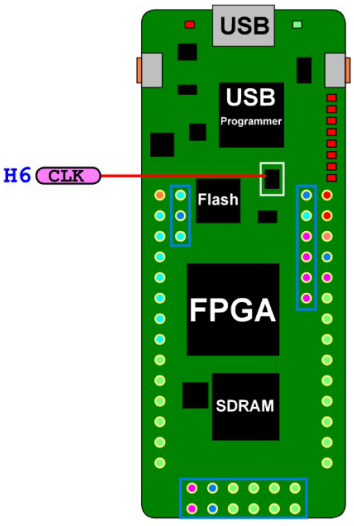

Abbildung 13: Die Taktgenerierung auf dem MAX1000-Board

Wenn man einen digitalen Signalpegel immer zwischen LOW und HIGH wechseln lassen möchte, was auch Toggeln genannt wird, dann kommt in den meisten Fällen ein NOT-Operator zum Einsatz. Dieser kehrt den aktuellen logischen Pegel einfach um. Man könnte meinen, dass der folgende Code funktionieren wird, um eine LED, wenn auch sehr schnell (6MHz), bei der ansteigenden Flanke des Takts blinken zu lassen:

```
01 library IEEE;
02 use IEEE.std_logic_1164.all;
03 use IEEE.numeric_std.all;
04
05 entity Blink_Problem is
06     port(
07         CLK : in  std_logic;
08         LED : out std_logic := '0'
09     );
10 end entity Blink_Problem;
11
12 architecture rtl of Blink_Problem is
13 begin
14     blink: process(CLK)
15     begin
16         if rising_edge(CLK) then
17             LED <= not LED;
18         end if;
19     end process blink;
20 end architecture rtl;
```

Listing: Das Toggeln der LED - Fehlerhafter Code!

Doch bevor es losgehen kann, möchte ich noch zeigen, wie Clock auf dem Board in das Mapping mit eingebunden wird. Wir klicken also auf die Compile-Schaltfläche und wählen dort das CLK-Signal in der Unconnected-Liste aus:

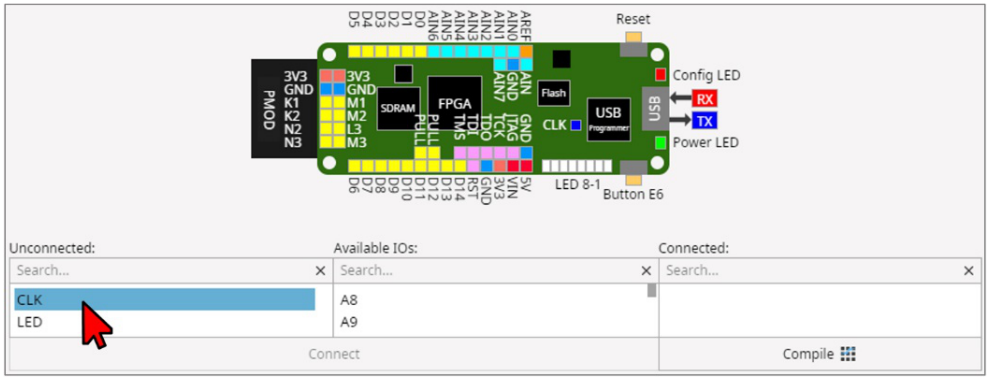

Abbildung 14: Das Auswählen des CLK-Signals

Nun erfolgt die Zuordnung zum Oszillator mit der Beschriftung CLK auf dem Board:

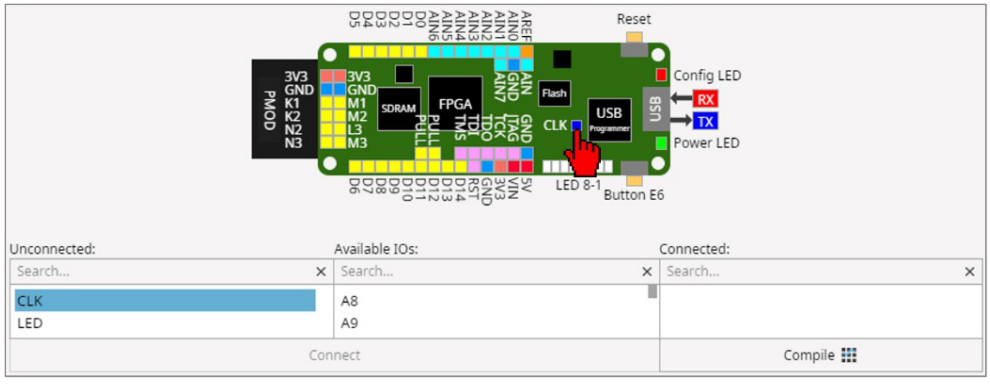

Abbildung 15: Das Auswählen des Oszillators

Um später leicht auf das Ausgangssignal zugreifen zu können, um es an ein Oszilloskop zu versenden, habe ich den digitalen Ausgang D5 für die LED verwendet. Der Dialog sieht wie folgt aus:

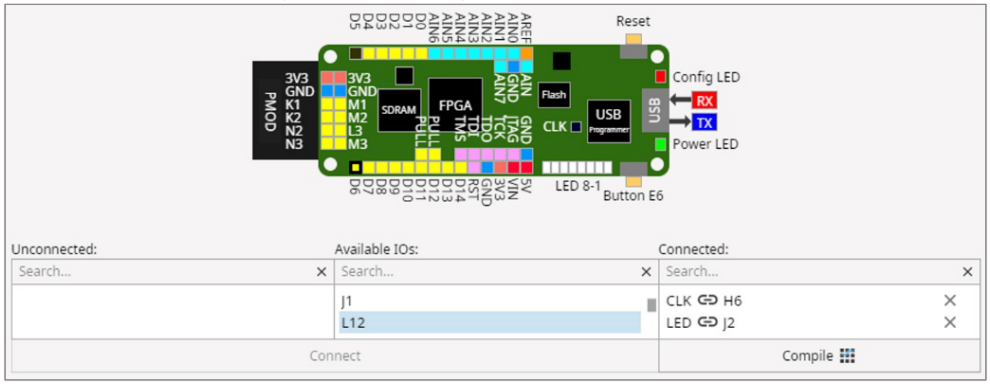

Abbildung 16: Das fertige Mapping

Nach dem Compiling erscheint folgende Fehlermeldung:

```
Interface object "LED" of mode out cannot be read. Change object
mode to buffer.
```

Ist diese Meldung verständlich? Sie besagt, dass das Signal LED nicht gelesen werden kann, da der Mode auf out steht. Es ist in dem Fall nur möglich, das Signal über eine Zuweisung zu beeinflussen, zum Beispiel so:

```
    LED <= '1';
```

Warum funktioniert also unsere Zeile nicht?

```
17  LED <= not LED;
```

Der NOT-Operator muss zur Umkehr des LED-Pegels diesen erst einmal ermitteln, also lesen, was aus genannten Gründen nicht möglich ist. Was also tun? Die Lösungsmöglichkeiten sind vielfältig. Ich möchte zwei davon zeigen. Zum einem zeigt das Standard-Template beim Öffnen eines VHDL-Projektes einen Signaltypen, der sich Buffer nennt. Wird dieser anstelle von out verwendet, klappt auch das Compiling. Die Modifikation in Zeile 08 sieht wie folgt aus:

```
04  ...
05  entity Blink_Problem is
06      port(
07          CLK : in     std_logic;
08          LED : buffer std_logic := '0'
09      );
10  end entity Blink_Problem;
11  ...
```

Ports, die als *buffer* deklariert sind, werden verwendet, wenn ein bestimmter Anschluss gelesen und auch geschrieben werden muss. Dieser Modus unterscheidet sich vom Inout-Modus. Die Quelle des Ports kann dabei nur intern sein. Ich möchte nicht weiter in die Thematik einsteigen. An dieser Stelle sei nur so viel gesagt, dass dieser Typ vom Entwickler und Hersteller Xilinx nicht empfohlen wird, weil bei es bei der Synthese zu Problemen führen kann. Darum zeige ich eine zweite Möglichkeit über die Verwendung eines zusätzlichen Signals, um dieses Problem zu beseitigen. Sehen wir uns dazu den Code weiter unten an. Mein Ziel ist es, ein Signal mit einer Frequenz von 10Hz am Pin D5 zu generieren, um dort eine LED und auch den Eingang meines Oszilloskops anzuschließen. Nun müssen wir herausfinden, mit welcher Frequenz der Oszillator auf dem MAX1000-Board arbeitet. Dort steht unter anderem:

Oscillator (DSC6011ME2A): - 12 MHz

Es sind also 12 MHz, was 12x106 Hz entspricht. Das sind demnach 12 Millionen Pulse in der Sekunde. Ich möchte mit dem folgenden Code eine Frequenz von 10 Hz erzeugen. Der Process-Block wird gemäß der vorherrschenden Taktrate durchlaufen. Zur Taktreduzierung verwendet man am besten einen Zähler, der zum Beispiel bei 0 beginnt und immer um den Wert 1 erhöht wird und beim Erreichen eines bestimmten Grenzwerts ein Ereignis auslöst. Anschließend muss der Zähler wieder auf 0 gesetzt werden, um das Spiel von

vorne beginnen zu lassen. Das Toggeln des Ausgangs für den Frequenzabgriff erreichen wir über das schon angesprochene zusätzliche Signal, das über den NOT-Operator zwischen 0 und 1 hin und her schwankt. Gehen wir das Vorhaben Schritt für Schritt durch.

Schritt 1: Die Schnittstellensignale deklarieren

Im ersten Schritt muss innerhalb des Entity-Blocks der Clock des Oszillators als in deklariert werden, da sein Signal in die Schaltung hineingeleitet wird. Des Weiteren wird das LED-Signal als out deklariert und mit dem Wert 0 initialisiert. An diesem Ausgang wird später der gewünschte Takt abgegriffen:

```
01 library IEEE;
02 use IEEE.std_logic_1164.all;
03 use IEEE.numeric_std.all;
04
05 entity Kap06_LED_Blink is
06     port(
07         CLK : in  std_logic;
08         LED : out std_logic := '0'
09     );
10 ...
```

Schritt 2: Architecture-Block mit Signalen hinzufügen

Im zweiten Schritt werden im Architecture-Block die zusätzlichen Signale definiert, wobei clk_counter für den Zähler und flag für das Toggeln zuständig ist:

```
01 library IEEE;
02 use IEEE.std_logic_1164.all;
03 use IEEE.numeric_std.all;
04
05 entity Kap06_LED_Blink is
06     port(
07         CLK : in   std_logic;        -- Clock-Signal-Oszillator
08         LED : out  std_logic := '0' -- Clock-Signal-Ausgang
09     );
10 end entity Kap06_LED_Blink;
11
12 architecture rtl of Kap06_LED_Blink is
13     signal clk_counter : integer range 0 to 6000000 := 0;
14     signal flag : std_logic := '0';
15 ...
```

Warum habe ich für den Zähler den Bereich 6.000.000 angegeben? Das hängt mit der Taktgeschwindigkeit von 12 MHz zusammen, wobei der Wert noch einmal halbiert wurde, um die An- und Ausphase des Rechtecksignals zu berücksichtigen. Würde ich diesen Wert später auch bei der Festlegung des Grenzwerts nutzen, um zu Toggeln und den Wert wieder mit 0 zu initialisieren, ergäbe sich eine Taktfrequenz von 1 Hz. Wir wollen jedoch eine höhere

Frequenz von 10 Hz erreichen. Dazu muss der Grenzwert kleiner sein, damit ein schnelleres Toggeln erreicht wird. In Zeile 13 wird ein Zählersignal (clk_counter) deklariert, das vom Typ Integer ist, den genannten Bereich von 0 bis 6.000.000 abdeckt und mit dem Wert 0 initialisiert wird. Das Flag-Signal (flag) wird in Zeile 14 vom Type std_logic deklariert und mit dem logischen Wert 0 initialisiert.

Schritt 3: Prozess definieren

Im dritten Schritt wird der Prozess mit seiner Logik definiert, um den Zähler zu beeinflussen und die LED anzusteuern. Hier zeige hier nur den hinzugekommenen Code:

```
14  ...
15  begin
16      blink: process(CLK)
17      begin
18          if rising_edge(CLK) then
19              clk_counter <= clk_counter + 1;
20              if clk_counter >= 600000 then
21                  flag <= not flag;
22                  clk_counter <= 0;
23              end if;
24          end if;
25      end process blink;
26      led <= flag;
27  end architecture rtl;
```

Der Prozess in Zeile 16 wird hinsichtlich der Sensitivity-List (CLK) nur bei einer Änderung des Clocks aufgerufen. Die Blockbildung über begin und end sorgt dafür, dass alle Anweisungen, die sich dazwischen befinden, sequenziell ausgeführt werden. In Zeile 19 wird der Zähler immer um den Wert 1 erhöht. Die if-Abfrage sorgt dafür, dass über die formulierte Bedingung schon beim Erreichen des Wertes 600.000 das Flag toggelt (Zeile 21) und anschließend der Zähler (Zeile 22) wieder auf den Wert 0 zurückgesetzt wird. Anschließend kommt es dann in Zeile 26 zur Zuweisung des Flag-Signals an die LED. Der gewünschte Takt wird generiert. Nachfolgend ist das Oszillogramm zu sehen:

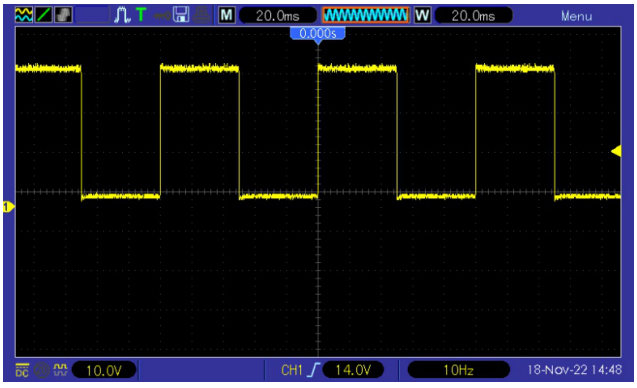

Abbildung 17: Das Oszillogramm des Ausgangssignals – Frequenz = 10Hz

Stimmt denn die Frequenz? Zum einen sieht man rechts unten, dass die Software des Oszilloskops das Signal analysiert hat und gibt die Frequenz mit 10 Hz an. Das passt also sehr gut. Man kann sich die Frequenz auch anhand des Oszillogramms ausrechnen. Hier ein kleiner Exkurs dazu: Die Zeitbasis des Oszilloskops ist auf 20ms eingestellt, was bedeutet, dass jeder Teilstrich in der Vertikalen einen zeitlichen Abstand von 20ms zum nächsten besitzt. Die Periodendauer T erstreckt sich genau über fünf Teilstriche je 20ms, was insgesamt eine einzige Periodendauer von 100ms ergibt. Die Frequenz f berechnet sich aus der gezeigten Formel ist der Kehrwert der Periodendauer:

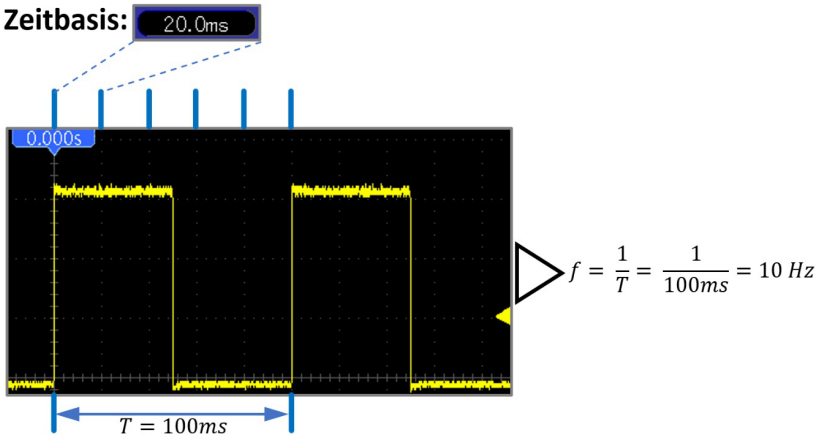

Abbildung 18: Die Berechnung der Frequenz

Das Ergebnis der Berechnung ergibt die erwarteten 10 Hz, die am Ausgang von Pin D5 anliegen. Nun kommen wir zum eigentlichen Ziel dieses Themenbereichs, nämlich der automatischen Ansteuerung des Lauflichts über den generierten Takt.

Ein automatisches Lauflicht

Jetzt müssen wir bestimmte Codebereiche aus der manuellen Steuerung des Lauflichts und der Taktgenerierung geschickt miteinander kombinieren. Die Weiterschaltung der einzelnen LEDs soll ja nun nicht mehr durch den Taster, sondern durch den Takt erfolgen. An der Stelle der einzelnen LED aus der vorigen Taktgenerierung muss wieder das LED-Array stehen. Zudem fällt das Flag für das Toggeln der einzelnen LED weg und an diese Stelle muss der Aufruf des Schieberegisters in Zeile 21 stehen, um dieses um eine Position nach links zu verschieben. Im Grunde genommen wäre es das schon. In Zeile 26 wird dann das Schieberegister dem LED-Array zugewiesen. Ich zeige nachfolgend den

```
01 library IEEE;
02 use IEEE.std_logic_1164.all;
03 use IEEE.numeric_std.all;
04
05 entity Kap06_Lauflicht is
06     port(
07         CLK : in  std_logic;
08         LED : out std_logic_vector(7 downto 0)
09     );
10 end entity Kap06_Lauflicht;
11
12 architecture rtl of Kap06_Lauflicht is
13     signal clk_counter : integer range 0 to 6000000 := 0;
14     signal shift_reg : std_logic_vector (7 downto 0) := "00000001";
15 begin
16     blink: process(CLK)
17     begin
18         if rising_edge(CLK) then
19             clk_counter <= clk_counter + 1;
20             if clk_counter >= 600000 then
21                 shift_reg <= shift_reg(6 downto 0) & shift_reg(7);
22                 clk_counter <= 0;
23             end if;
24         end if;
25     end process blink;
26     LED <= shift_reg;
27 end architecture rtl;
```

Listing: Kap06_Lauflicht.vhd – Das automatische Lauflicht

Sehen wir uns abschließend das zeitliche Verhalten der LED-Ansteuerung in einem Logic-Analyzer an. Es ist wunderbar zu erkennen, dass bei jedem Taktimpuls der HIGH-Pegel von einem zum nächsten Ausgang wandert und am Schluss alles wieder von vorne beginnt:

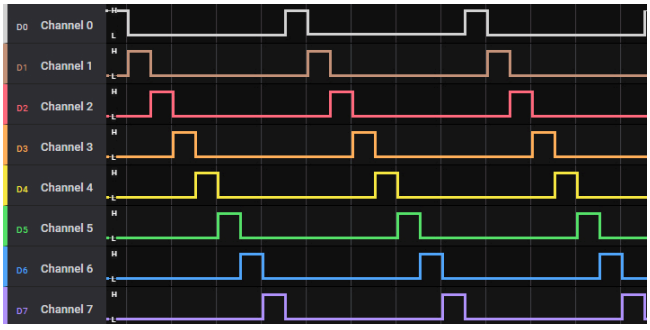

Abbildung 19: Der zeitliche Verlauf der LED-Ansteuerung

Allgemeine Taktgenerierung

Abschließend zum Thema blinkende LEDs und Lauflichter, wo es ja um die Nutzung des Takts auf dem MAX1000-Board ging, hier ein kurzer Code, der die Taktgenerierung mit einstellbarer Frequenz zeigt:

```
01 library IEEE;
02 use IEEE.std_logic_1164.all;
03 use IEEE.numeric_std.all;
04
05 entity Kap06_Taktgenierung is
06     generic(
07         CLK_Freq : integer := 12000000; -- 12MHz (Board-Takt)
08         MUX_Freq : integer := 100       -- 100Hz (gewünschte Freq.)
09     );
10     port(
11         CLK : in  std_logic;
12         LED : out std_logic := '0'
13     );
14 end entity Kap06_Taktgenierung;
15
16 architecture rtl of Kap06_Taktgenierung is
17     signal clk_counter : integer range 0 to (CLK_Freq/MUX_Freq/2)
   :=0;
18     signal flag : std_logic := '0';
19 begin
20     proc: process(CLK)
21     begin
22         if rising_edge(CLK) then
23             clk_counter <= clk_counter + 1;
24             if clk_counter = CLK_Freq/MUX_Freq/2 then
25                 clk_counter <= 0;  -- Zähler-Reset
26                 flag <= not flag;  -- Flag toggeln
27             end if;
28         end if;
29     end process proc;
30     LED <= flag;  -- Flag-Signal an Ausgang schicken
31 end architecture rtl;
```

Listing: Kap06_Taktgenierung.vhd

Es ist eine Frequenz von 100Hz gewünscht und diese wird an Pin D5 geleitet, an dem normalerweise auch eine LED angeschlossen ist. Die in den Generics definierten Konstanten (Zeilen 07 und 08) legen fest, wie weit später ein Zähler inkrementiert werden soll. Beim Erreichen des Grenzwerts (er muss zudem innerhalb der if-then-Abfrage noch durch 2 dividiert werden) erfolgen zwei Dinge:

- Das Rücksetzen des Zählers auf 0 (Zeile 25)
- Die Umschaltung (Toggeln) des Flags (Zeile 26)

Über das Oszillogramm sehen wir, dass der Frequenzwert des Rechtecksignals korrekt angezeigt wird:

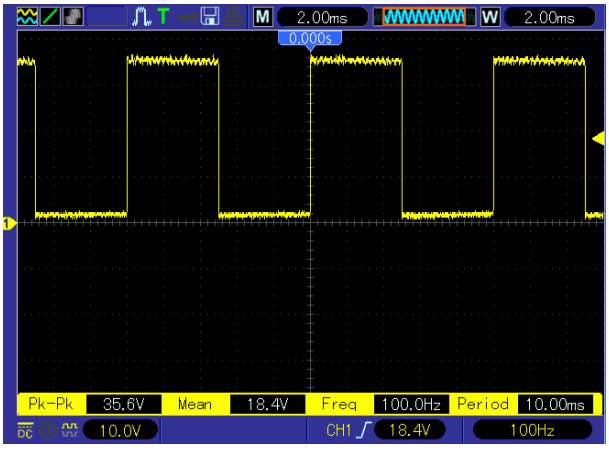

Abbildung 20: Das Oszillogramm des Ausgangssignals – Frequenz = 100Hz

Diesen Code zur Taktgenerierung habe ich aus dem Grund einmal separat gezeigt, weil er später im Kapitel zur Ansteuerung von mehrstelligen Siebensegmentanzeigen verwendet wird. Es geht im Speziellen darum, zwischen den einzelnen Stellen der Anzeige im Multiplexverfahren zu wechseln. Doch dazu später mehr.

Im kommenden Projekt geht es um das wichtige Thema Variablen und Signale.