Entwicklung von digitaler Logik mit programmierbaren Logikbausteinen.

Juni 2005 A.Schnell DG6KBU

Bei der Entwicklung von einfachen digitalen Interfacesystemen auch im Bereich des Amateurfunks wurde bislang üblicherweise aus dem Sortiment einfacher Logikbausteine ausgewählt und mit Gattern, Flipflops, Zählern, Multi/Demultiplexer und Schieberegistern die entsprechende Schaltung aufgebaut. Eine Vielzahl entsprechender Bauanleitungen findet sich in den einschlägigen Fachzeitschriften. Der Vorteil dieser Vorgehensweise ist, das die Schaltung mit einfachen Mitteln (Spannungsmessgerät, Oszilloskop) an jedem Punkt überprüft werden kann, insbesondere da die Arbeitsfrequenzen meist nur bis in den 100KHz - 1MHz Bereich reichen. Der Nachteil ist natürlich, das eine größere Platine für die zahlreichen einzelnen Bausteine entworfen werden muß und jeder Fehler, der sich beim Testen ergibt, normalerweise mit der Erstellung einer neuen Platine einhergeht. Je umfangreicher die Schaltung wird, umso unübersichtlicher wird dann auch die Funktion des Systems, dies ist vergleichbar mit der sogenannten „Spagettiprogrammierung“ bei der Softwareentwicklung. Lösen läßt sich diese Problematik durch eine Modularisierung des Systems mit klaren Schnittstellen zwischen den Teilmodulen, wie es im Bereich der objektorientierten Programmierung auch realisiert wird.

Eine andere Vorgehensweise besteht darin, die notwendige Logik nicht mit Hardware sondern mit Software zu realisieren. Hierfür können einfache, preiswerte aber dennoch recht leistungsfähige Mikrocontroller wie zB. die AVR Serie der Firma Atmel eingesetzt werden. Diese Mikrocontroller sind inzwischen weitgehend mit Flash Programmspeichern ausgestattet und können selbst im fertigen System mit der notwendigen Software geladen werden. Die erforderliche Entwicklungssoftware zur Programmerstellung und zum Download gibt es kostenfrei vom Hersteller, selbst Entwicklungsumgebungen die eine Programmierung in C erlauben sind meist kostenlos oder zu geringen Preis erhältlich. Eine einfache, aber typische Entwicklungsaufgabe (Abb.1) soll dies aufzeigen. Die Abfrage eines 3x4 Tastaturfeldes mit der Ermittlung der gedrückten Taste, je nach gedrückter Taste soll einer der 12 Ausgänge auf Low gehen und zB. eine angeschlossene Leuchtdiode aufleuchten lassen.

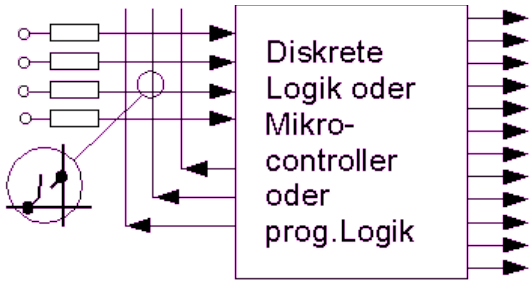

Abb.1

Eine mögliche Lösung mit diskreter Logik ist schon nicht ganz trivial, da doch eine ganze Reihe von unterschiedlichen Logikkomponenten benötigt werden. Sie soll hier nicht weiter diskutiert werden.

Mikrocontroller

Der zweite Lösungsansatz mit einem Mikrocontroller erscheint einfach: ein kurzes C Programm welches die Tastaturspalten abscannt und prüft ob eine Taste gedrückt ist, aus Spalte und Reihe läßt sich die Taste ermitteln und der entsprechende Ausgang des Mikrocontrollers auf Low setzen. (Listing.1). Dies Listing zeigt ein C Programm, welches mit der freien Version einer C Entwicklungsumgebung[1] erstellt worden ist und sich zum Download in einen Atmel AVR Mikrocontroller eignet. Wir benötigen hier schon insgesamt 19 Portleitungen. Falls der Controller neben der Tastaturabfrage auch noch weitere Dinge, wie eine Displayansteuerung, serielle Schnittstelle, PWM etc. behandeln soll, wird die notwendige Software gegebenfalls mit Interuptroutinen schnell recht komplex und das zeitliche Zusammenspiel der einzelnen Softwarekomponenten muß sehr genau behandelt werden, da der Controller ja nur alle Aufgaben sequentiell bearbeiten kann.

Programmierbare Logik

An dieser Stelle kommt nun der dritte Lösungsansatz ins Spiel, der von Funkamateuren bislang noch nicht so häufig benutzt wird. Allerdings gibt es Bereiche im Amateufunk, wie D-ATV, wo für die umfangreichen Fehlerkorrekturmaßnahmen

1 http://www.hpinfotech.ro/cvavre.zip CodeVisionAVR Evaluation

bereits komplexe programmierbare Systeme eingesetzt werden[2]. In den letzten Jahren sind programmierbare Logikbausteine auf den Markt gekommen, die neben einer hohen Anzahl von möglichen Logikverknüpfungen (mehrere 1000 bis 100,000 de) insbesondere wie die modernen Mikrocontroller die nichtflüchtige Programmierung der Logik im fertigen System erlauben (In System Programmable ISP). Außerdem ist die dazugehörende Entwicklungssoftware, die vor wenigen Jahren nur auf teuren Workstations lief und von den Herstellern zu sehr hohen Preisen veräußert wurde, inzwischen bei den meisten Herstellern von Komplexen (Complex) bzw. in Feld (Field) programmierbaren Bausteinen CPLD u. FPLA kostenfrei auf den entsprechenden Webseiten verfügbar. Siehe [3] [4]. Auch gibt es unter Anderem eine Art Programmiersprache, mit der sich Hardware und Logikverknüpfungen relativ einfach beschreiben lassen: VHDL, Very high Hardware Description Language. Diese Hardwarebeschreibungssprache ist von den Strukturen an übliche Programmiersprachen angelehnt und recht einfach zu verstehen. Ich hoffe, das das Beispiel den einen oder anderen Funkamateur ermuntert kleinere Logikprojekte auf die gezeigte Weise zu realisieren und sich mit den programmierbaren Logiksystemen zu befassen. Die weiteren Möglichkeiten reichen bis zur komplexen Basisbandaufbereitung von digitalen Videosignalen. Hier soll allerdings nur der Einstieg in diese moderne Technik aufgezeigt werden.

Mit VHDL ergibt sich also die Möglichkeit, die notwendige Logik einer Schaltung auf einer abstrakten Ebene zu beschreiben und einen programmierbaren Logikbaustein entsprechend zu programmieren. Der Preis eines für unsere Aufgabe geeigneten Bausteins liegt in der gleichen Größenordnung wie der eines einfachen Mikrocontrollers (3-4 Euro)[5]. Der Vorteil einer solchen Lösung liegt aber darin, das die entsprechenden CPLD Bausteine in der Regel recht viele Input/Outputleitungen zur Verfügung stellen und alle Komponenten innerhalb des Bausteins unabhängig voneinander parallel arbeiten, wenn nötig bis zu Taktraten in den 20-50 MHz Bereich!

Um einen Einstieg in die neue Technik zu finden, eignen sich insbesondere die Entwicklungskits der verschiedenen Hersteller. Diese sind auf den Webseiten aufgeführt und können dort auch gleich bestellt werden. Zwei sehr günstige und leistungsfähige Kits, die auch bei einem deutschen Distributor bestellt werden können, habe ich bei XILINX gefunden[6] [7]. Der erste Kit kostet ca. 50 Euro und enthält alles (Software und Hardware einschließlich Downloadkabel für Parallelschnittstelle), um zwei CPLD Bausteine auf dem Board programmieren zu können. Das zweite Board ist wesentlich komplexer, enthält einen recht großen FPLA Baustein und ist schon für sehr umfangreiche Projekte geeignet, es können hier sogar komplette Mikrocontroller in den Baustein geladen werden. Ein simples VGA Interface, PS2 Anschluß, diverse Tasten/ Schalter, eine 4 fach 7Segmentanzeige sowie externes RAM vervollständigen das Board. Mit ca. 150 Euro ist aber dieses Board aber auch deutlich teurer.

Für die ersten Versuche ist das preisgünstige Board gut geeignet. Zusätzliche Peripherie, die durch den Baustein angesteuert werden kann, kann auf einer recht großen Lochrasterfläche ergänzt werden, zB. eine Ergänzung mit LEDs, Tastern oder 7 Segmentanzeigen. Am flexibelsten ist man aber, wenn man an den Steckerleisten eine zweite Platine mit allen notwendigen Komponenten anschließt. Für ein anderes Projekt läßt sich dann diese Zusatzplatine entsprechend neu gestalten. Wir haben eine solche Zusatzplatine entworfen, die 10 LEDs, einen PS2 Anschluß, 2 7Segmentanzeigen, ein 3*4 Keypad, einen 8 poligen DILSchalter, und einen Rotary Encoder (digitales Drehpoti) enthält (Abb.2). Diese Komponenten können entweder von dem kleineren XC9572XL Baustein oder nach Umstecken der Zusatzplatine von dem größeren CoolrunnerII XC2C256 Baustein angesteuert werden, wobei im letzteren Fall noch sehr viele ungenutzte I/Os für weitere Aufgaben zur Verfügung stehen.

2 Thomas Sailer, HB9JNX, Uwe Kraus, DJ8DW

3 www.xilinx.com: ISE7 Web Edition

4 www.altera.com: QuartusII Web Edition

5 www.reichelt.de: XC9572XL PC44

6 www.hot-electronic.de: Coolrunner II Starter Board

7 www.hot-electronic.de: Spartan-3 Starter Board

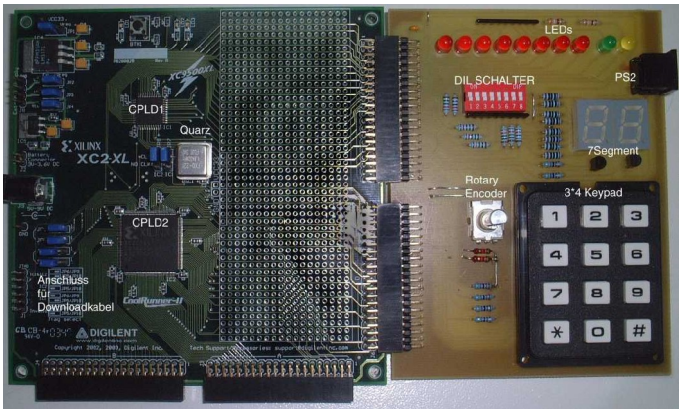

Abb.2 Starterkit mit Peripherieboard

Ein weitere Vorteil des kleinen Entwicklungsboard besteht übrigens darin, das die CPLD Bausteine bei Taktraten im 1-2 Mhz Bereich nur sehr wenig Strom benötigen und daher das ganze System mit zwei AA Batterien versorgt werden kann. Das verringert beim Arbeiten mit einem Notebook auf dem Schreibtisch unnötigen Kabelwust.

CPLD- FPLA

Nun zur Erklärung der beiden bisher benutzten Bezeichnungen CPLD und FPLA. CPLD steht für Complex Programmable Logic Device und benutzt eine Technologie, die schon seit vielen Jahren für programmierbare Logikbausteine wie zB. PALs 22V10 etc. verwendet wird. Die Eingangsleitungen werden über programmierbare Matrixkreuzungspunkte einer UND Matrix zugeführt und UND verknüpft, die Ausgänge der UND Matrix werden dann ODER Gattern zugeführt, deren Ausgänge wiederum zB in FlipFlops zwischengespeichert werden können (Abb.3).

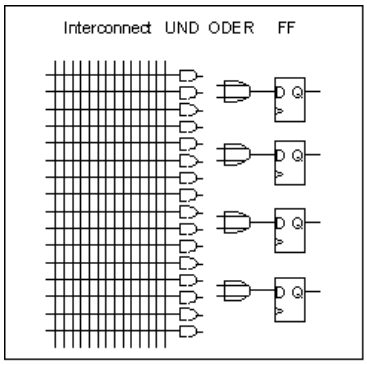

Abb.3 Stuktur CPLD

In den Grundlagen der digitalen Logik lernt man, das jeder beliebige logische Zusammenhang zwischen digitalen Eingängen und Ausgängen über eine solche UND / ODER Verknüpfung realisiert werden kann. Die programmierbaren Matrixkreuzungspunkte werden in der Technologie von Flashspeichern als nichtflüchtige und löschbare Verbindungen realisiert.

FPLA steht für Field Programmable Logic Array. Hier werden die Matrixkreuzungspunkte nicht in der Flashtechnologie erstellt, sondern durch Schalttransistoren die mit einer RAM Speicherzelle verbunden sind. Die logische Information in einem FPLA ist also flüchtig, muß bei jedem Einschalten des Systems wiederhergestellt werden. Dafür sind heute üblicherweise kleine Flashspeicher vorgesehen, in welche die logischen Verknüpfungsinformationen beim Download gespeichert werden und die beim Einschalten des Systems automatisch die notwendige Neuprogrammierung des Bausteins erledigen. FPLA können daher unbegrenzt neu beschrieben werden, während bei CPLDs wie auch bei Flash Speichern eine Grenze bei 10-20 Tausend Schreibzyklen liegt. (Bei 10 Neuprogrammierungen jeden Tag ist man da aber schon bei 3-4 Jahren).

Die interne Struktur der FPLA ist ebenfalls anders, man hat eine große Anzahl von kleineren Matrixverbindungsbereichen mit Logik/ FlipFlops. Damit lassen sich FPLAs flexibler mit komplexer Logik programmieren (Abb.4).

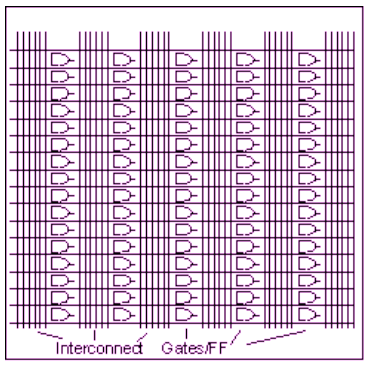

Abb.4 Struktur FPLA

Glücklicherweise braucht man sich aber über die Details der internen Struktur keine großen Gedanken zu machen, da die Entwicklungsumgebungen sich beim Erstellen der Downloaddateien automatisch an die interne Struktur des Bausteins anpassen (sie "Fitten" den Entwurf in den Baustein). Man muß nur im Kopf behalten, das die interne Struktur der CPLDs nicht ganz so flexibel ist wie die der FPLAs und die Anzahl der zur Verfügung stehenden FlipFlops begrenzt ist. Wir werden aber sehen das selbst mit dem

kleinen XC9572XL schon ganz nette Designs, wie unsere Keypadabfrage, ohne Probleme umgesetzt werden können.

"Programmiersprache" VHDL
Wenden wir uns jetzt der Programmierung der Bausteine zu. Wie erwähnt kann man hierfür VHDL verwenden. Am einfachsten versteht man diese Sprache indem ein Beispiel analysiert wird. In dem ersten Beispiel soll ein synchroner Binärzähler realisiert werden, der das Systemtaktsignal in 20 Stufen herunterteilt und nach außen führt. Als Erstes müssen dafür die 20 Zähler FlipFlops definiert werden, die für einen Binärzähler benötigt werden. Dies geschieht mit

```
signal clk_divider :
           std_logic_vector(19 downto 0);
```

signal ist das Schlüsselwort (im folgendem immer kursiv) für die Verwendung interner Register FlipFlops. `clk_divider` ist die Bezeichnung dieser Zählerkette, die einzelnen Stufen werden mit zB. mit `clk_divider(5)` angesprochen. `std_logic_vector( .. )` definiert, das alle 20 Stufen, durchnummeriert von 19 bis 0 zum kompletten Binärzähler zusammengefaßt werden.

Die Verbindung zur Außenwelt muß natürlich auch hergestellt und definiert werden, dies geschieht mit dem Schlüsselwort *Port*:

```
port( clock : in std_logic;
      divider_out : out std_logic_vector
      (19 downto 0);
```

`clock` kennzeichnet die Eingangsleitung, an der der Systemtakt angeschlossen ist, `divider_out` bezeichnet die Anschlüsse, an denen das heruntergeteilte Taktsignal zur Verfügung steht.

Dies sind die wesentlichen syntaktischen Vorbereitungen die gemacht werden müssen, jetzt kommt der eigentliche Zähler. Dieser wird als eigenständiger Prozess definiert:

```
process(clock)
begin
  if(clock'event and clock = '1') then
    clk_divider <= clk_divider + 1;
  end if;
end process;
```

if(clock'event *and* clock = '1') legt fest, daß nur bei einer Änderung des clocksignals, dh. einer Taktflanke (`event`) und dem high (`'1'`) Zustand des Taktsignals (damit wird die ansteigende Flanke definiert!) der Inhalt der `if` Konstruktion ausgeführt wird. In unserem Fall soll ja der synchrone Zähler um eins incrementiert werden, dies wird ganz einfach durch die Anweisung

```
clk_divider <= clk_divider + 1;
```

realisiert. Der Operator <= ist keine Zuweisung im Sinne üblicher Programmiersprachen, sondern beschreibt eine elektrische Verbindung. In unserem Fall wird bei jeder ansteigenden Taktflanke der um 1 erhöhte binäre Wert der Zählerkette wieder in die 20 (D)FlipFlops übernommen.

Wichtig ist, das bei diesem synchronen Zähler alle Ausgangssignale gleichzeitig ihren Wert verändern, im Gegensatz zu einfachen asynchronen Binärzählern, die wegen der Gatterlaufzeiten leicht zeitversetzt ihren Zustand ändern. Grundsätzlich werden Prozesse von den Signalen gesteuert, die in der Parameterliste von `process( .. )` (sogenannte process sentivity list) angegeben sind.

```
--          simple counter for ELEKTOR
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

entity counter is
  Port ( clock : in std_logic;
         divider_out : out
         std_logic_vector(19 downto 0));
end counter;

architecture Behavioral of counter is
  signal clk_divider : std_logic_vector(19
  downto 0);
begin
  process(clock)
  begin
    if(clock'event and clock ='1') then
      clk_divider <= clk_divider + 1 ;
    end if;
  end process;

  divider_out <= clk_divider;
end Behavioral;
```

Listing 2 Simple Counter in VHDL

Listing 2 zeigt nun das komplette Programm für den 20 stufigen synchronen Binärzähler. Syntak-

tisch sind noch einige zusätzliche Dinge aufgeführt, die aber durch die Entwicklungsumgebung teilweise selbständig erzeugt werden.

Nach `--` ist immer Kommentar zu finden (1. Zeile). Der *`library`* `IEEE;` Block definiert einige Bibliotheksmodule, die standardmäßig eingebunden werden. Der Block mit dem Schlüsselwort *`entity`* enthält die Definition der Verbindungen zur Außenwelt (`port ...`), der Block mit dem Schlüsselwort *`architecture`* enthält die Hardwarebeschreibung des Projekt, zunächst mit der Deklaration der benötigten Signale (die als D FlipFlops realisiert werden) und nach dem *`begin`* dann alle Hardwareprozesse. Elektrische Verknüpfungen können auch außerhalb von Prozessen realisiert werden, sie sind dann dauerhaft und hängen nicht von Taktsignalen ab:

```
divider_out <= clk_divider;
```

alle 20 Verbindungen der Zählerkette werden hier direkt mit den Ausgangsleitungen verbunden.

Dies einfache Beispiel zeigt wie logisch und einfach in VHDL Hardware beschrieben werden kann. Andererseits zeigt dieses Beispiel nur einen Bruchteil der syntaktischen Möglichkeiten die von VHDL abgedeckt werden. In seiner Gesamtheit ist VHDL eine äußerst mächtige und leistungsfähige Programmiersprache, heute werden selbst komplexe Telekommunikationsbausteine wie sie in Handies verwendet werden, komplett in VHDL beschrieben. Die eigentlichen Vorteile werden deutlich, wenn statt des 20 stufigen synchronen Zählers jetzt ein 25 stufiger Zähler, der bei jeder Taktflanke um 2 dekrementieren soll, realisiert werden soll. Hierfür sind lediglich einige Zahlenwerte zu ändern. Die Struktur bleibt gleich. Damit können einmal realisierte und ausgetestete Teilprozesse sehr leicht wiederverwendet und an andere Randbedingungen angepasst werden.

Durch die Entwicklungsumgebung wird automatisch auch ein Report über die verwendeten Resourcen des Bausteins erzeugt. Ein Auszug daraus:

```
***  Resource Summary  XC9572XL VQ44 ****

Macrocells     Product Terms    Registers
Used           Used             Used
20 /72 (28%)   19 /360 (5%)     20 /72 (28%)

Pins           Function Block
Used           Inputs Used
21 /34 (62%)   54 /216 ( 25%)
```

Die Anzahl der Macrozellen wie auch der benutzten Register gibt an, wieviel FlipFlops verwendet wurden. Insgesamt sind in diesem Baustein 72 FFs vorhanden, 20 davon wurden für die Zählerstufen verwendet. Die Anzahl der Produktterme gibt die maximale Anzahl der verfügbaren ODER Verknüpfungen an, die verfügbar sind. Für den synchronen Zähler sind nur 5% davon verwendet worden. Wir sehen, das nur ein geringer Teil der Resourcen für diesen Zähler verwendet wurden und damit noch einige Erweiterungsmöglichkeiten in diesem Baustein stecken.

Dem Leser ist vielleicht schon aufgefallen, das wir den I/O Pins zwar Namen gegeben haben, aber noch nirgendwo festgelegt haben an welchen physikalischen Anschlussbeinchen des Bausteins die entsprechenden Signale zur Verfügung stehen sollen. Hierfür ist mit der Entwicklungsumgebung eine Zuordnungsdatei zu erstellen, die genau diese Zuordnung festlegt. Diese Zuordnung definiert eine Einschränkung für das Entwicklungssystem, daher wird sie als Constraints Datei bezeichnet. Man kann neben den Pinzuordnungen insbesondere auch bestimmte zeitliche Einschränkungen, (wie zB. maximale Schaltzeiten zwischen I/O Pins) definieren, die besonders bei komplexen Entwicklungen bei hohen Taktraten wichtig sein können.

Keypadabfrage in VHDL

Wenden wir uns dem Ausgangsproblem zu. Die Keypadabfrage ist etwas komplizierter, da wir das Tastenfeld in 3 Schritten abscannen müssen, dh. wir haben es auch mit 3 verschiedenen Abfragezuständen zu tun. Für solche Aufgabenstellungen eignen sich sogenannte Statemachines hervorragend, die mit Zustandsdiagrammen sehr leicht beschrieben werden können. Die Darstellung aus einem Zustandsdiagramm kann sehr einfach in VHDL dargestellt werden, die Entwicklungsumgebung enthält sogar einen eigenen grafischen Eingabemodus (StateCAD) in dem Zustandsdiagramme gezeichnet werden können und die dann automatisch zur Programmierung des Bausteins verwendet werden können!

Zustandsdiagramm
Die Zustände eines Systems werden in einem Zustandsdiagramm einfach durch Kreise dargestellt. In unserem Fall benötigen wir also drei Kreise, die die drei Abfragezustande kennzeichnen, wir können also jedem Kreis den zugehörenden binären Outputzustand der drei Keypadspalten zuordnen. Pfeile zwischen den Zustandskreisen beschreiben, wie und wann von einem Zustand in den nächsten gewechselt wird. Diese Pfeile können auch auf den gleichen Zustandskreis zurückverweisen, wenn ein Ereignis nicht zu einem Zustandswechsel führen soll. In unserem Fall soll durch ein Taktsignal zyklisch durch die drei möglichen Abfragezustände gewechselt werden, wenn keine Taste gedrückt ist. Damit sieht das Zustandsdiagramm für unsere Keypadabfrage wie folgt aus:

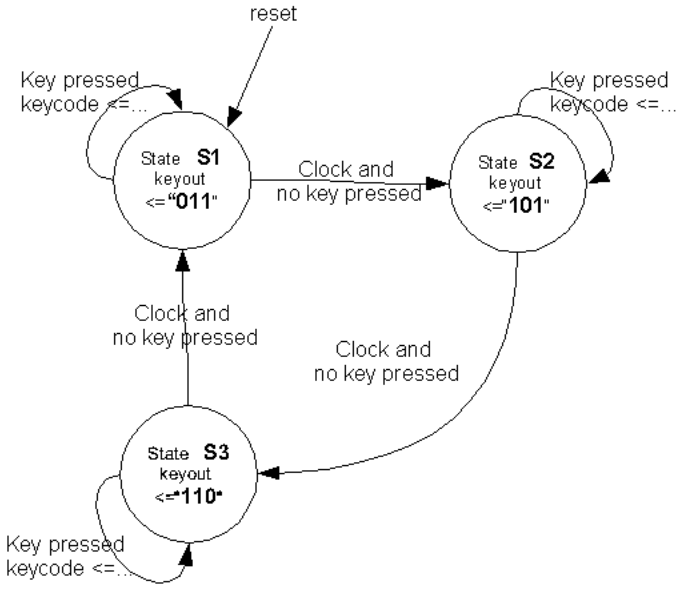

Abb.5 Zustandsdiagramm für Keypad

Mit S1, S2 und S3 sind die drei Abfragezustände bezeichnet, `keyout` bezeichnet die drei Ausgangsleitungen für die Reihenabfrage des Keypads, diese werden in den drei Zuständen der Reihe nach auf low gesetzt. Der Zustandswechsel erfolgt immer zum Taktevent, wenn keine Taste gedrückt ist. Dies läßt sich leicht feststellen, indem die 4 Eingangsleitungen `keyin`, die mit den Zeilen des Keypads verbunden sind, daraufhin überprüft werden, ob ihr Wert gleich "`1111`" ist, dh. alle high sind. Beim Drücken einer Taste wird eine der vier Eingangsleitungen auf low gehen, und die Kombination mit der gerade abgefragten Spalte (dem `keyout` Wert) ergibt den Tastencode, der eine der angeschlossenen LEDs aufleuchten läßt.

Generell kann man sagen, das die Beschreibung eines Systemverhalten durch diese Zustandsdiagramme sehr eindeutig auch komplexe Systeme beschreiben kann. Überall da wo Echtzeitprozesse und Steuerungsfunktionen ausgeführt werden sollen, die von Ereignissen ausgelöst werden ist diese Art der Systembeschreibung ideal.

Sehen wir uns die Umsetzung dieser Statemachine in den entsprechenden VHDL Code mal an. Es werden zwei Prozesse definiert, ein Synchronisationprozess, der festlegt wann der Zustandswechsel erfolgt (mit Label SYNC_PROC) und ein Kombinatorischer Prozess (COMB_PROC), der die eigentliche Abfragelogik festlegt:

Zusätzlich sind noch folgende Deklarationen für die Statemaschinen im architecture Block: zu machen:

```
type STATE_TYPE is (S1,S2,S3);
CurrentState, NextState : STATE_TYPE;
```

Hiermit werden die drei Zustände festgelegt und zwei Zustandscounter, die den aktuellen und den nächsten Zustand festhalten.

```
SYNC_PROC: process(clk, reset)
begin
  if (reset='0') then
    CurrentState <= S1;
  elsif (clk'event and clk = '1') then
    CurrentState <= NextState;
  end if;
end process;
```

Dieser Prozess ist einfach zu verstehen, nach einem reset wird der Zustand S1 als Ausgangszustand eingestellt, bei einer ansteigenden Flanke des Taktsignals wird der in Nextstate vorliegende Zustand in den aktuellen Zustandszähler übernommen.

Der Kombinatorische Prozess bereitet nun jeweils vor, welches der nächste Zustand sein soll, indem in `NextState` jeweils der nächste Zustandswert abgelegt wird, zB.

```
NextState <= S2;
```

```
COMB_PROC: process (CurrentState)
```

```
begin
  case CurrentState is
    when S1 => keyout <= "011";
      if (keyin = "0111") then keycode
                    <= "011111111111";
      elsif(keyin = "1011") then keycode
                    <= "101111111111";
      elsif(keyin = "1101") then keycode
                    <= "110111111111";
      elsif(keyin = "1110") then keycode
                    <= "111011111111";
      else
        keycode <= "111111111111";
        NextState <= S2; -- if no key
      end if;

    when S2 => keyout <= "101";
      if (keyin = "0111") then keycode
                    <= "111011111111";
      elsif(keyin = "1011") then keycode
                    <= "111110111111";
      elsif(keyin = "1101") then keycode
                    <= "111111011111";
      elsif(keyin = "1110") then keycode
                    <= "111111101111";
      else
        keycode <= "111111111111";
        NextState <= S3; -- if no key
      end if;

    when S3 => keyout <= "110";
      if (keyin = "0111") then keycode
                     <= "111111110111";
      elsif(keyin = "1011") then keycode
                     <= "111111111011";
      elsif(keyin = "1101") then keycode
                     <= "111111111101";
      elsif(keyin = "1110") then keycode
                     <= "111111111110";
      else
        keycode <= "11111111";
        NextState <= S1; -- if no key
      end if;
  end case;
end process;
```

Listing 3 State Machine in VHDL

Syntaktisch neu für uns ist die *case* Anweisung mit den Auswahlschlüsselwort when ... Je nachdem, welchen Wert currentstate hat (S1, S2 oder S3) wird der entsprechende when Block ausgewählt. Innerhalb dieses when Blocks wird dann mit dem *if* Konstrukt der anliegende keyin Wert mit den möglichen Eingangskombinationen verglichen und entsprechend mit dem keycode Wert eine der AusgangsLEDS eingeschaltet. Falls keine Taste gedrückt ist werden alle LEDs ausgeschaltet und NextState auf den nächsten Zustand geschaltet.

Damit ist unsere Hardware, die wir für unsere Keypadabfrage brauchen komplet beschrieben. Das ganze muss noch ergänzt werden durch die notwendigen library und Port Deklarationen. Listing 4 zeigt das komplette Programm.

Entwicklungsumgebung

Zum Schluss wollen wir einen kurzen Blick auf die von XILINX angeboten integrierte Entwicklungsumgebung (IDE) werfen. Alle Details zu erklären würde den Rahmen dieses Artikels sprengen. Es handelt sich um ein sehr umfangreiches und leistungsfähiges Gesamtpacket, welches kostenfrei von der XILINX Webseite heruntergeladen werden kann bzw. bei den Starterkits auf CD mitgeliefert wird. Abb. 6 gibt einen Eindruck von der Oberfläche:

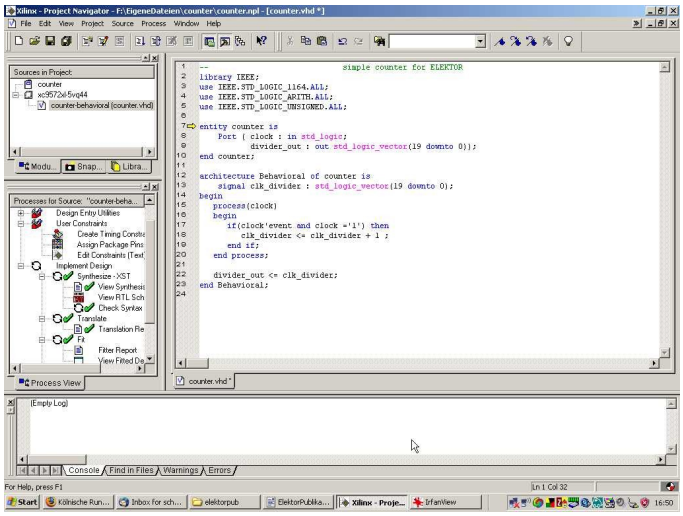

Abb. 6 XILINX IDE

Sämtliche Arbeitsschritte sind logisch in einem Prozessfenster angeordnet und werden auf Mausklick ausgeführt. Auch der endgültige Download in den Baustein des Starterkits wird auf diese Weise interaktiv gestartet. Hinter einem Button mit dem Symbol einer Glühbirne versteckt sich ein für den Anfänger äußerst nützliches Feature dieser IDE. Nach dem Klick darauf erscheint ein sogenanntes Language Template, indem zwischen den verschiedenen Hardwarebeschreibungssprachen wie ABEL, UCF, VERILOG und VHDL ausgewählt werden kann. Nach Klick auf VHDL erscheint eine Liste aller syntaktischen Konstruktionen, nach einer weiteren Auswahl zB. case wird beispielhaft die Syntax angezeigt, sie kann direkt in das aktuelle

Editorfenster der IDE übernommen werden. Selbst komplexere Komponenten wie Statemachines und Schieberegister werden so anschaulich erklärt. Insgesamt ist diese Entwicklungsumgebung meiner Meinung nach hervorragend in die Windowsoberfläche integriert, nach einigen Stunden Beschäftigung mit ihr steht eigenen einfachen Designs mit CPLD Bausteinen nichts mehr im Wege.

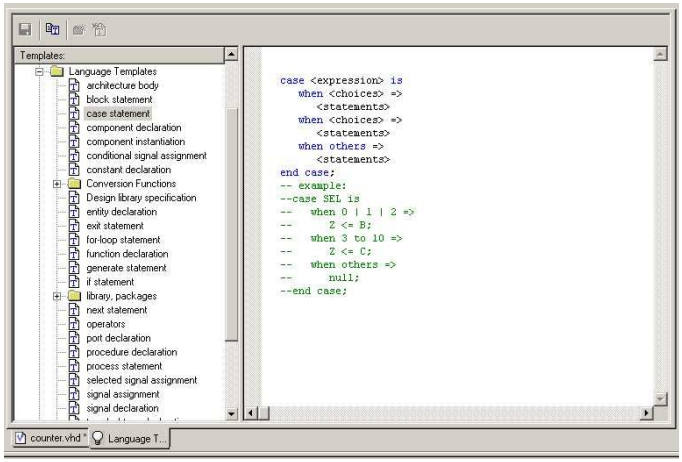

Abb. 6 Screenshot von Language Template

Beim professionellen Entwurf von programmierbaren Schaltungen wird typischerweise immer noch eine sogenannte Testbench programmiert, die das Verhalten des gerade erstellten Hardwareentwurfs mit dem erwarteten Verhalten überprüft. Insbesondere die unterschiedlichen Gatterlaufzeiten innerhalb des Bausteins können zu fehlerhaften Verhalten führen. Die Erstellung einer Testbench ist oft sogar noch komplexer wie das eigentliche Hardwaredesign und soll hier nicht behandelt werden. Für einfache Logiksysteme ist ein Test direkt in der fertigen Hardware viel einfacher zu realisieren. Man probiert nach dem Download in den Baustein einfach aus ob alles funktioniert. Im Zweifelsfall programmiert man den Baustein einfach neu.

Anlagen:
Listing 1 AVR Mikrocontrollercode
Listing 4 VHDL Code komplett
Schaltplan, Stückliste und Layout von Experimentierplatine
Literatur/Links VHDL:
http://redhat.regent.e-technik.tu-muenchen.de/forschung/vhdl/schaltungsdesignVHDL.pdf
http://tech-www.informatik.uni-hamburg.de/vhdl/doc/kurzanleitung/vhdl.pdf

```
// 3*4 keypad connected to PortD, for each pressed
// key one of the 12 Outputs of PortB and PortC
// will be low

#include <90s4433.h>
void displaykey(void);

void main (void)
{  PORTB=0x3F;
   DDRB=0x3F;  //Output  for 6 LEDs
   PORTC=0x3F;
   DDRC=0x3F;  //Output  for 6 LEDs

   PORTD=0x7F; // Port D initialization: xOOOIIII
   DDRD=0x70;  // for keypad: 3 rows and 4 lines

   while(1)    // main loop
   {  displaykey(); }
}

void displaykey(void)
{     char keyin;

      PORTD = 0b11101111;    // "110" to keyout
      keyin = PIND & 0b00001111;
      if (keyin == 0b00001110) PORTB = 0b11110111;
      else
      if (keyin == 0b00001101) PORTC = 0b11111110;
      else
      if (keyin == 0b00001011) PORTC = 0b11110111;
      else
      if (keyin == 0b00000111) PORTC = 0b11101111;
      else {PORTB =0b11111111; PORTC = 0b11111111;}

      PORTD = 0b11011111;    // "101" to keyout
      keyin = PIND & 0b00001111;
      if (keyin == 0b00001110) PORTB = 0b11111101;
      else
      if (keyin == 0b00001101) PORTB = 0b11011111;
      else
      if (keyin == 0b00001011) PORTC = 0b11111011;
      else
      if (keyin == 0b00000111) PORTB = 0b11111110;
      else {PORTB =0b11111111; PORTC = 0b11111111;}

      PORTD = 0b10111111;    // "011" to keyout
      keyin = PIND & 0b00001111;
      if (keyin == 0b00001110) PORTB = 0b11111101;
      else
      if (keyin == 0b00001101) PORTB = 0b11110111;
      else
      if (keyin == 0b00001011) PORTC = 0b11111101;
      else
      if (keyin == 0b00000111) PORTC = 0b11011111;
      else {PORTB =0b11111111; PORTC = 0b11111111;}

      return;
}
```

Listing 1