

Eine Einführung in die Mikroelektronik*

(*) für Schüler der HTBLuVA Salzburg

1. Ziel des Tutorials

- Kennenlernen der Entwurfsmethodik von Microelektronik nach heutigem Industriestandards
- Microelektronik mit Hilfe der Hardwarebeschreibungssprache VHDL selbst entwerfen.
- Ein Verständnis für Hard- und Software Co-Development und die Komplexität von Microelektronik-Projekten aufbauen.

2. Vorgangsweise

- Kennenlernen der Hardwarebeschreibungssprache VHDL.
- Kennenlernen der Entwicklungsumgebung für Microelektronik-Projekte.
- Lesen von Artikeln und Fachliteratur sowie user manuals im Unterricht und zu Hause, in Englisch und in Deutsch.
- Erstellen einer eigenen Mitschrift.
- Installieren einer IDE am Laptop.
- Erstellen eigener Microelektronik-Projekte in VHDL.
- Simulation/Verifikation von VHDL Programmen.
- Jede Woche eine Mitarbeitskontrolle in schriftlicher Form bezüglich der Theorie.

Ziel:

Anhand von kleinen Beispielen sollte vermittelt werden:

- Wie VHDL aufgebaut ist und angewendet wird,
- Wie Microelektronik entwickelt wird,
- Welche Probleme lösbar sind,
- Wo die technischen Grenzen liegen.

3. Was ist VHDL?

http://de.wikipedia.org/wiki/Very_High_Speed_Integrated_Circuit_Hardware_Description_Language

4. Fachliteratur und Kurse

Im WWW findet sich bereits eine nicht mehr zu überblickende Menge an Kursen und Literatur zum Thema VHDL. Fast jede Universität und viele Firmen bieten entsprechende Kurse und auch Training zum Erlernen der Programmiersprache VHDL an.

Anbei seien nur einige Quellen aufgezeigt um einen raschen Einstieg zu vermitteln.

- ❖ Erste Einführung: FH-München (**Crashkurs VHDL**)
http://www.uni-ulm.de/fileadmin/website_uni_ulm/iui.inst.050/vorlesungen/sose09/lrob/Crashkurs_VHDL.pdf
- ❖ VHDL Wissensdatenbasis der UNI Hamburg
<http://tams-www.informatik.uni-hamburg.de/vhdl/>
- ❖ Altera: IDE Umgebung und Bausteine
<http://www.altera.com/>

5 Anmerkungen

hier steht ein schlauer Motivationsspruch

Programmierbare Logikbausteine

PLD (ROM, PLA, PAL)

Abb 1 Zeigt die vereinfachte Darstellung von Gattern in programmierbaren Logikbausteinen, wie sie in den nachfolgenden Abbildungen dargestellt werden.

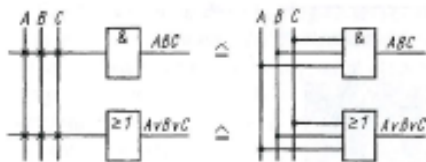


Abb 1: Vereinfachte Darstellung der Verknüpfungen von AND und OR Gattern.

Die Verknüpfungen sind bei diesen Bausteinen fix verdrahtet, und werden bei der Programmierung durchgebrannt. Daraus folgt, dass diese Bausteine nur einmal verwendbar sind. In der heutigen Zeit haben diese Bausteine ihre Bedeutung verloren, und wurden schon vor einiger Zeit durch elektrisch programmierbare und löschbare Bausteine abgelöst (siehe EPLD)

Begriff „fusen“

Zum näheren Verständnis eines programmierbaren Logikbausteins wird ein Siebensegment-Decoder als Beispiel betrachtet. Dieser hat vier Eingangsleitungen, sowie 7 Ausgangsleitungen. Somit kann man auch von einem ROM mit entsprechend vier Adressleitungen und sieben Datenleitungen ausgehen.

Dieser Adressdecoder wurde im Rahmen des Konstruktionsunterrichtes des Öfteren mit Hilfe eines PLD's realisiert.

Beispiel:

Entwurf eines Siebensegmentdecoders von Gray Code auf Siebensegmentansteuerung.

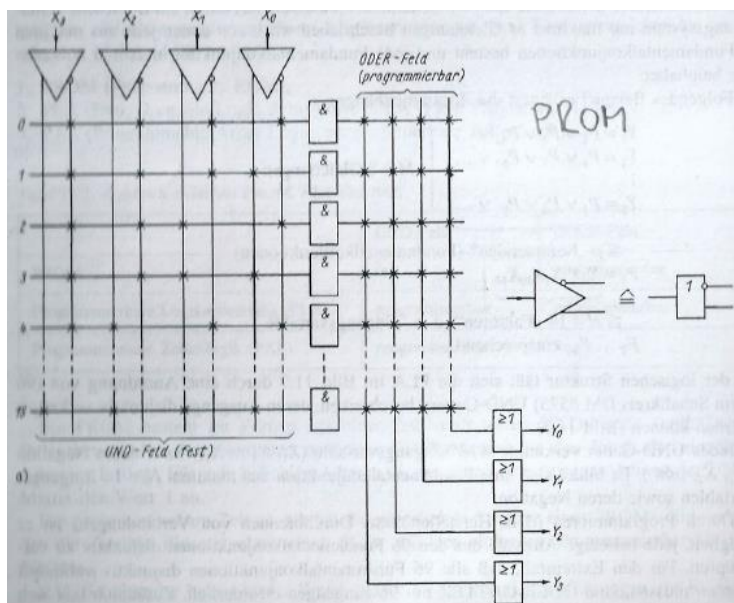


Abb 2: Innere Struktur eines ROM Bausteins

	UND Matrix	ODER Matrix	Anmerkung
ROM	fest verdrahtet	programmierbar	kanonische Normalform (alle Adressen)
PLA	programmierbar	programmierbar	minimierte Realisierung eines ROM, jedoch hoher Platzbedarf durch zwei programmierbare Arrays
PAL	programmierbar	fest verdrahtet	Kompromiss von geringerer Fläche zu weniger Flexibilität

Tabelle 1: Zusammenfassung und Unterscheidung zwischen PLD's.

Makrozellenstruktur

Neben der Matrixstruktur von programmierbaren UND und fest verdrahteten ODER Gattern besitzt eine Makrozelle auch einen Speicher (D-FF), sowie einen Tri-State Ausgang.

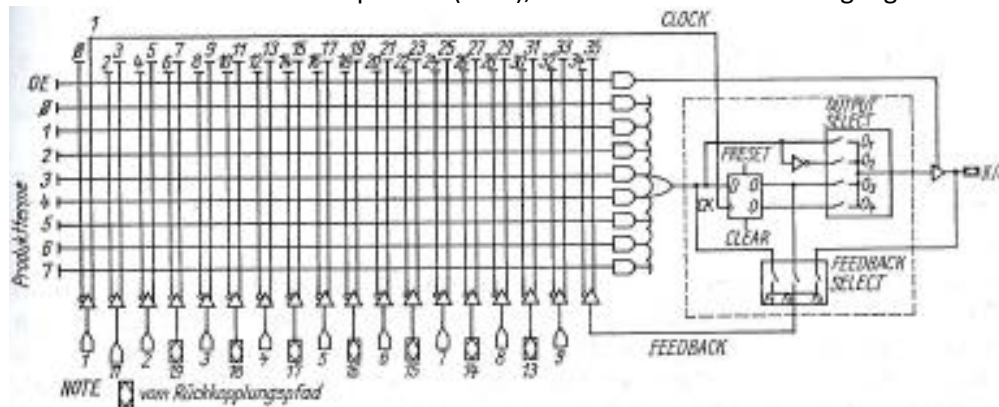


Abb 3: Makrozellenstruktur mit vorgeschalteter Programmiermatrix.

Tri-state Ausgang:

Diskutiere mit Deinem Nachbarn die folgende Schaltung eines Tri-State Treibers (Abb 4) und beschreibe die Funktion mittels einer Wahrheitstabelle.

<http://www.elektronik-kompodium.de/public/schaerer/tristate.htm> (Quelle: Elektronik Kompodium, 18.03.2014, 21:30)

Funktionsbeschreibung:

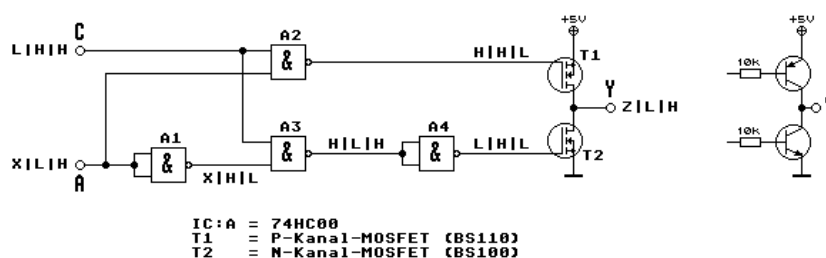


Abb 4: Implementierung eines Tri-State Treibers

EPLD (Erasable PLD)

Im Gegensatz zu PLA und PAL Bausteinen enthält diese Bausteintechnologie keine fix verdrahteten Verknüpfungen, sondern elektrische Schalter zur Herstellung der logischen Verknüpfungen. Diese können auf elektrischem Weg programmiert und gelöscht werden.

Ein EPLD Baustein enthält meist mehrere Makrozellen. In der heutigen Zeit haben auch diese Bausteine aufgrund ihrer begrenzten Funktionsdichte ihre Bedeutung verloren.

GAL

Eine Variante programmierbarer PLD's sind sog. GAL Bausteine (Generic Array Logic). Die Verknüpfungen sind in einer zusätzlichen EEPROM Einheit am Baustein gespeichert, und erlauben damit ca. 10.000 Programmier- und Löschzyklen.

Der innere Aufbau besteht wie bei PAL aus einer programmierbaren UND Matrix mit einer nachfolgenden besonders flexibel konfigurierbaren Ausgangslogik Makrozelle (OLMC Output Logic MacroCell).

Auch diese Bausteine sind heutzutage nicht mehr im industriellen Einsatz. Vielfach wurden sie durch Kombination von mehreren PAL Bausteinen zu einem komplexen PLD abgelöst, um der Forderung nach steigender Komplexität und Integrationsdichte Rechnung zu tragen.

CPLD

Bei CPLD's sind im Allgemeinen mehrere PAL über ein gemeinsames Kontaktfeld (switch matrix) miteinander verbunden. Der Vorteil ist immer die Beibehaltung der PLD Entwurfsprinzipien (Optimierung Boolescher Gleichungen) und die Berechenbarkeit und Konstanz der Verbindungen zwischen den Gattern. Entsprechend den verschiedenen Herstellern gibt es unterschiedliche Architekturvarianten, die sich in der Anzahl der Makrozellen, der direkten Ein- und Ausgänge, oder auch in der Anzahl der Taktsignale voneinander unterscheiden. Geringe Verzögerungszeiten (ns-Bereich) sowie eine hohe Anzahl von Programmierzyklen zeichnen moderne CPLD Bausteine darüber hinaus aus.

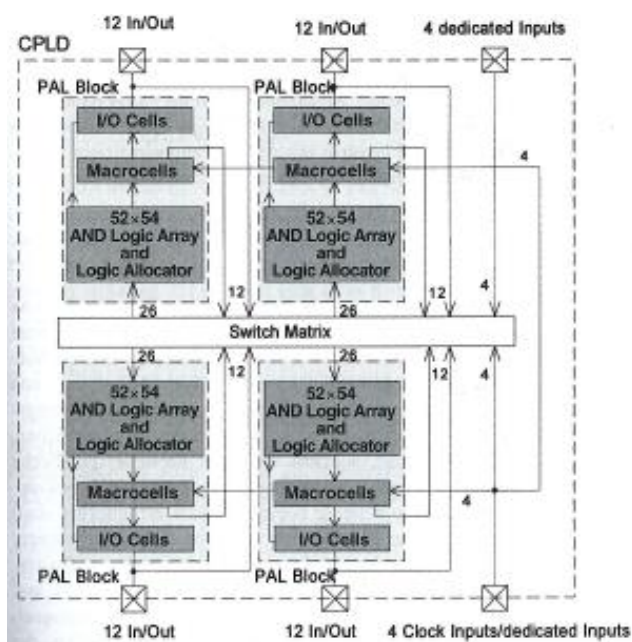


Abb 5: Innerer Aufbau eines CPLD Bausteins (z.B. MAX7000/EP7064)

FPGA (Field Programmable Gate Arrays)

Die Funktionsweise von FPGAs stützt sich prinzipiell auf das Vorbild der bisher besprochenen Bausteine. Das heißt, der innere Aufbau orientiert sich im Kernbereich an einer matrixförmigen

Anordnung von Logikblöcken (Logic Element, LE/Configurable Logic Block, CLB). Die Logikblöcke unterscheiden sich jedoch von der aus der CPLD Technik bekannten Anordnung von Makrozellen. Der Aufbau von Logikzellen besteht meist aus der Verbindung einer programmierbaren Look Up Table (LUT) sowie einem nachgeschalteten Speicher.

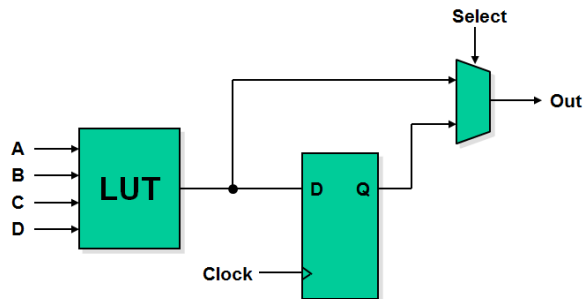


Abb 6: Schematische Darstellung eines FPGA Logic Blocks

Auf Abbildung (Abb 7) erkennt man den Aufbau eines Logic Elements wie es in einem FPGA von Altera implementiert ist. Grundsätzlich lassen sich dieselben Elemente wie sie in Abb 6 dargestellt sind erkennen. Zusätzliche Erweiterungen, wie der Block carry-chain, diverse Multiplexer und bypass Pfade dienen der Erhöhung der Flexibilität des LE in Verbindung mit anderen LE's.

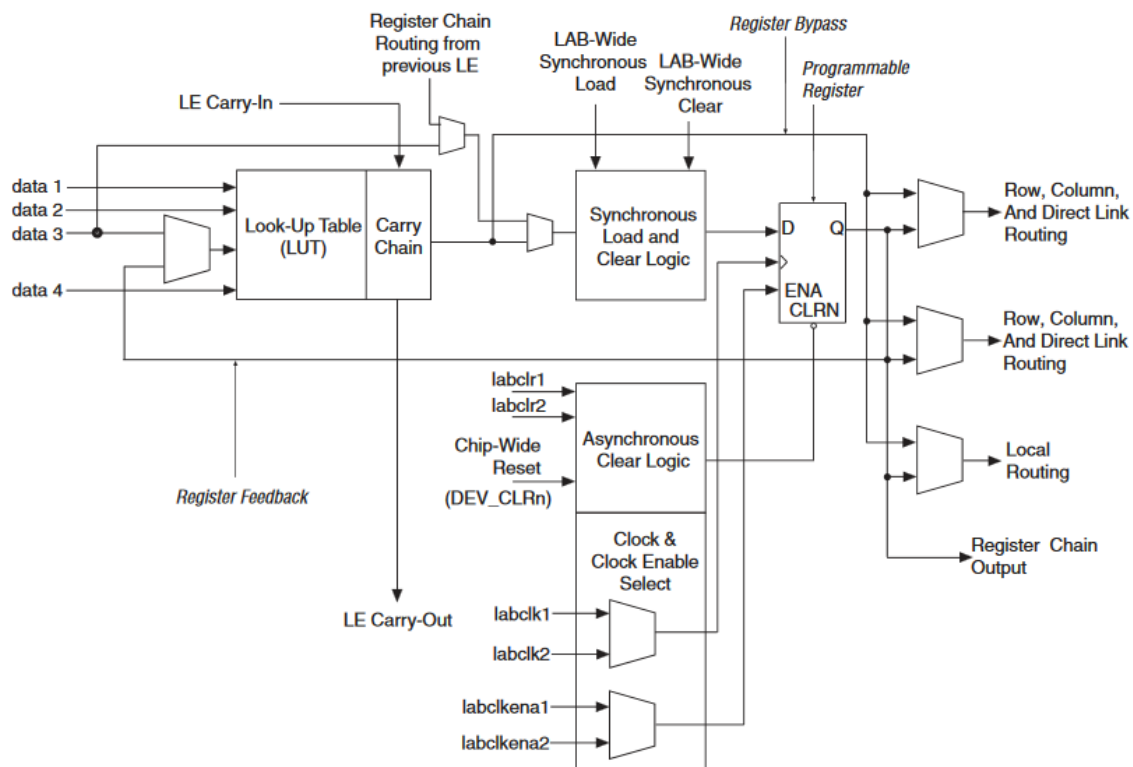


Abb 7: Ansicht eines Logic Elements aus dem Altera Cyclone IV Device Handbook

Das nachfolgende Blockschaltbild zeigt den prinzipiellen Aufbau eines FPGA Bausteins bestehend aus ein Logic Elementen, I/O-Blöcken, sowie den Verbindungspfaden welche in eine Schaltmatrix führen um die einzelnen LE miteinander zu verknüpfen. Alle drei genannten Blöcke sind flexibel programmierbar entsprechend der vorgegebenen Hardwarebeschreibung in VHDL oder Verilog.

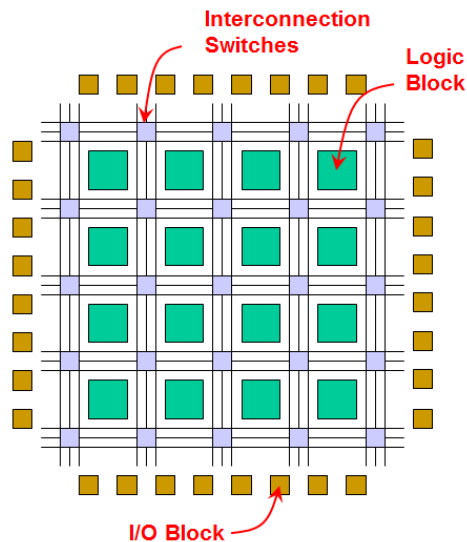


Abb 8: Prinzipieller Aufbau eines FPGA Bausteins

Aufgabenstellung1:

Zur genaueren Studie des inneren Aufbaus von FPGAs ist in Einzelarbeit das folgende „White Paper“ von National Instruments zur Funktionsweise aufzuarbeiten. Der innere Aufbau sowie die Funktionsbestandteile LUT, Flip-Flop sind in die Mitschrift zu übernehmen.

<http://www.ni.com/white-paper/6983/de/> (Quelle: National Instruments, 18.03.2014; 21:50)

Aufgabenstellung2:

Aufbau moderner FPGAs: https://www.altera.com/content/dam/altera-www/global/en_US/pdfs/literature/hb/cyclone-iv/cyiv-5v1.pdf (Altera, 06.03.2017, 17:10)

Was sind die wesentlichen Merkmale der Altera Cyclone IV Familie.

Fragestellungen:

- 1) Wie ist ein ROM (PAL, PLA) aufgebaut?
- 2) Beschreibe die disjunktive Normalform.
- 3) Wie ist eine Makrozelle aufgebaut?
- 4) Wie funktioniert ein Tri-State Treiber.
- 5) Was sind die Eigenschaften eines CPLD?
- 6) Unterschiede FPGA, CPLD....
- 7) Was ist eine LUT und gib dafür ein Beispiel.
- 8) Innerer Aufbau eines FPGA.
- 9) Wie sind Moderne FPGAs aufgebaut? Wie viele LEs können implementiert sein, und nenne mind drei weitere Funktionseinheiten.

Die Hardwarebeschreibungssprache VHDL

Aufbau einer VHDL Beschreibung

VHDL Beschreibungen einer Schaltung (design, block, unit) bestehen im Wesentlichen aus drei Elementen, nämlich der entity, der architecture und der configuration. Diese können hierarchisch aufgebaut sein, dh ausgehend von einem toplevel, zB der Microchip Außenhülle gibt es einen baumartige Verzweigung in Untereinheiten.

Schnittstellenbeschreibung (entity)

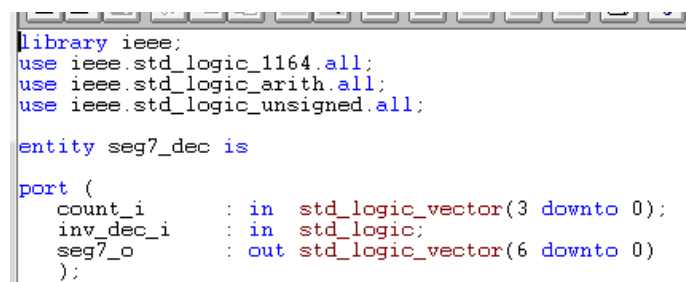
In der Entity wird die Schnittstelle des zu modellierenden Systems beschrieben, also die Ein- und Ausgänge der dieser Entity zugeordneten Schaltungs-Architektur.

Die formale Syntax einer Entity wird in Abb 9 gezeigt.

```
entity <blockname> is
port
(
  <signalnamen>: <richtung> <typ>; -- bei weiteren Signalen: `;`
  <signalnamen>: <richtung> <typ>   -- beim letzten Signal KEIN `;`
);
end;
```

Abb 9: Struktur einer Entity

Zur näheren Erklärung dieser Struktur wird die Entity eines Siebensegment Decoders verwendet, welcher im Rahmen eines Schülerprojektes implementiert wurde.



```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
use ieee.std_logic_unsigned.all;

entity seg7_dec is
port (
  count_i      : in  std_logic_vector(3 downto 0);
  inv_dec_i    : in  std_logic;
  seg7_o       : out std_logic_vector(6 downto 0)
);
```

Abb 10: Entity eines Siebensegment Decoders

Weiters ist in Abb 10 der eingefügte Headerkopf ersichtlich. Er ist vergleichbar mit der #include Anweisung aus C, und implementiert spezielle nur für die Schaltungsbeschreibung geltende Definitionen, wie zB Rechenoperatoren und Dualzahlenarithmetik.

Architektur (architecture)

Die Architektur ist die Beschreibung der Schaltungsfunktionalität eines Systems. Hierzu gibt es grundsätzlich zwei unterschiedliche Ansätze. Das System kann über eine Verhaltensmodellierung beschrieben werden, oder es kann ein strukturaler Ansatz (Netzliste) implementiert werden. Es können auch beide Ansätze gemischt implementiert werden.

```
architecture <beschreibungsnamen> of <blockname> is
  -- hier können lokale Signale deklariert werden
begin

  -- hier steht die Funktionsbeschreibung (nebenläufig)

end;
```

Abb 11: Rahmen einer Architecture

Zu jeder Entity können auch mehrere Architekturvarianten bestehen, von denen jedoch nur eine ausgewählt werden kann. Die Auswahl der jeweils optimalsten Architekturvariante passiert über eine sogenannte Konfiguration.

Konfiguration (configuration)

Abb 12 zeigt den Zusammenhang von Entity, Architecture und Configuration.

Sofern nur eine Architekturvariante erstellt wurde, kann die Configuration auch entfallen.

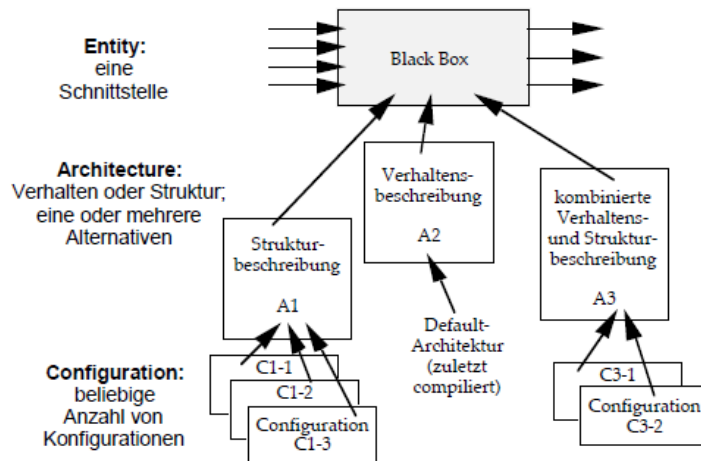


Abb 12: Aufbau einer VHDL Beschreibung

Aufgabenstellungen:

- 1) Erstelle eine VHDL Beschreibung eines einzelnen Grundgatters mit zwei und nachfolgend mit mehreren Eingängen und einem Ausgang.
- 2) Erstelle eine VHDL Beschreibung eines Standard CMOS Bausteins (74HCTxx). Realisiere eine der gegebenen Grundfunktionen wie NAND, NOR, NOT, AND, OR, EXOR.

Signale, Vektoren und zugehörige Datentypen

Die einzelnen units einer hierarchischen Designstruktur werden über sogenannte Signale miteinander verbunden.

Signale sind neben **Konstanten** und **Variablen** spezielle Objekte, welche durch die Angabe eines Identifiers (= Name), eines Datentyps und ggf eines Defaultwertes deklariert werden.

Nachfolgend gezeigt ist die Syntax zur Deklaration eines Signals und einige Beispiele.

Weitere Beispiele sind dem Unterricht zu entnehmen.

Syntax:

signal <signalnamen>: typ;

Beispiele:

```
signal X0, X1, X2, X3: bit;    -- vier Signale X0 bis X3 vom Typ bit
signal EN: std_logic;         -- signal EN hat den Typ std_logic
signal ein_aus: boolean;      -- ein boolesches Signal ein_aus
```

Abb 13: Verschiedene Signaldeklarationen

Weitere Beispiele:

```
signal hcus_1t          : std_logic      := '1';
```

```
signal al_del      : std_logic;
signal count      : integer    := 0;
```

Ein weiteres wichtiges Objekt sind Vektoren. Dabei handelt es sich um zusammengesetzte Signale beliebiger Bitbreite. Vektoren dienen zur Definition von Bussen, Zählern, Schieberegistern uva.

Syntax:

```
signal <signalnamen>: typ( <lower> to <upper>);
signal <signalnamen>: typ( <upper> downto <lower>);
```

Abb 14: Declaration eines Vektors.

Beispiele:

```
signal data_1t      : std_ulogic_vector(data_width_g-1 downto 0);
signal h_ctr        : unsigned((hmax_par_i'length)-1 downto 0);
```

Weitere Beispiele:

```
CONSTANT pplop_lim_c : integer := 32; -- minimal linelength/2
SIGNAL pplop_int     : integer RANGE 0 TO (2**14)-1; -- pplop
```

Unbedingte / bedingte /selektive Signalzuweisung

Quelle: Reichardt / Schwarz

Die Signalzuweisung erfolgt mit Hilfe eines Zuweisungsoperators: „<=“

Unbedingte Signalzuweisung

Nach erfolgter Deklaration eines Signals kann diesem dann ein anderes Signal, ein logischer Wert oder das Signal selbst einem Port zugewiesen werden. Es ist jedoch auf die entsprechende Bitbreite zu achten.

```
signal temp1      : std_logic;
signal temp2      : std_logic;
```

```
temp1 <= '1';
temp2 <= temp1;
```

```
-- Zuweisung auf den Ausgang port_a_o
port_a_o <= temp1 and temp2;
```

Aufgabenstellungen:

1) Löse das Übungsblatt: 02_VHDL_Signale_Ubungsaufgaben.

Bedingte Signalzuweisung

Die bedingte Signalzuweisung führt zu einer Verschachtelung entsprechend der auscodierten Priorität der Steuersignale.

Die Anweisung ist vergleichbar mit der if – then – else Anweisung wie wir sie in sequentiellen Umgebungen noch kennenlernen werden.

Die in Abb 15 gezeigte Syntax zeigt, daß beliebige Bedingungen gesetzt werden können. Der n-te Zweig ohne Bedingung entspricht einer Defaultbelegung für die Zuweisung.

```
<ergebnis> <=      <Verknüpfung_1> when <Bedingung_1>
                    else <Verknüpfung_2> when <Bedingung_2>
                    else <Verknüpfung_n>;
```

Abb 15: Syntax der bedingten Signalzuweisung when/else

```
58  -- Variation 4: bedingte Signalzuweisung
59  architecture rtl_when of mux_variation is
60  L-- enter signal declarations here
61  begin
62      y_o <= a_i when (sel_i = '1') else
63          b_i when (sel_i = '0') else
64          b_i;
65  end rtl_when; -- of mux_simple
66
```

Abb 16: Bedingte Signalzuweisung zur Beschreibung eines Multiplexers.

```
69  -- solution 2: concurrent statement
70  seg7_int <= "111110" when (count_i = "0000") else
71      "0110000" when (count_i = "0001") else
72      "1101101" when (count_i = "0011") else
73      "1111001" when (count_i = "0010") else
74      "0110011" when (count_i = "0110") else
75
76      "1011011" when (count_i = "0111") else
77      "1011111" when (count_i = "1000") else
78      "1110000" when (count_i = "1001") else
79      "1111111" when (count_i = "1010") else
80      "1110011" when (count_i = "1011") else
81      "1000111"; -- fail
82
```

Abb 17: Bedingte Signalzuweisung zur Realisierung eines Siebensegment Decoders

Selektive Signalzuweisung

Die Syntax (siehe) einer selektiven Signalzuweisung entspricht der Auswahl aus einer Reihe gleichberechtigter Möglichkeiten. Die Verwendung dieser Implementierungsform führt nach der Synthese häufig zu einer Multiplexerstruktur.

```
with <auswahlsignal> select
ergebnis <=      <Verknüpfung_1> when <auswahlwert_1>,
                <Verknüpfung_2> when <auswahlwert_2>,
                <Verknüpfung_n> when others;
```

Abb 18: Syntax der selektive Signalzuweisung with/select

```
47  -- Variation 3: selektive Signalzuweisung
48  architecture rtl_with of mux_variation is
49  L-- enter signal declarations here
50  begin
51      with sel_i select
52          y_o <= a_i when '1',
53              -- b_i when '0',
54              b_i when others;
55  end rtl_with; -- of mux_simple
56
```

Abb 19: Selektive Signalzuweisung zur Beschreibung eines Multiplexers

Nebenläufige und sequentielle Umgebungen

Der wesentliche Unterschied einer Hardwarebeschreibungssprache zu einer sequentiellen Programmiersprache wie „C“ besteht in der zwingenden Notwendigkeit parallel ablaufende Ereignisse zu beschreiben, wie dies auch in echter Hardware zum Tragen kommt.

Man unterscheidet daher zwischen nebenläufigen und sequentiellen Umgebungen. Eine Umgebung wird mit den Schlüsselwörtern **begin** eröffnet und mit **end** abgeschlossen. Die erste, bei der Erstellung der architecture eröffnete Umgebung, ist logischerweise nebenläufig.

Nebenläufige Umgebungen

Nebenläufige Umgebungen beschreiben das parallele Verhalten von Hardware.

Alle Zuweisungen werden parallel ausgeführt. Die Reihenfolge der Zuweisung in der architecture hat auf das Ergebnis keinen Einfluß.

Aufgabenstellungen:

Verwende jeweils eine unbedingte, bedingte bzw selektive Signalzuweisung.

- 1) Implementierung eines 1002 Multiplexers
- 2) Implementiere einen 4002 Demultiplexer
- 3) Implementierung eines Volladdierers
- 4) Implementiere eine Füllstandsüberwachung
- 5) Implementiere einen Adressdecoder der aus einer 8bit Adresse vier Segmente zu je 6bit ansteuert.

Benötigtes Grundwissen, welches im Crash Kurs zu finden ist:

Nebenläufige Anweisungen, logische Operatoren

Sequentielle Umgebungen

Sequentielle Umgebungen ermöglichen die Beschreibung von hintereinander ablaufenden Ereignissen, zB von Prozessen, Funktionen und Prozeduren. Dazu stehen programmiersprachenartige Befehle (Verzweigungen, Schleifen Variablenzuweisungen) zur Verfügung.

Die in VHDL zur Beschreibung von sequentiellen Umgebungen zur Verfügung stehende Grundstruktur ist der Prozess. Abb 20 zeigt die dazu notwendige grundsätzliche Syntax.

```
process <empfindlichkeitsliste>
-- hier können lokale Signale oder Variablen deklariert werden
begin
-- hier ist eine sequentielle Umgebung
end process;
```

Abb 20: Grundstruktur eines Processes in VHDL

Ein process kann, besser sollte, einen process-label besitzen.

Im Folgenden werden drei unterschiedliche Anwendungsfälle der Beschreibung von sequentieller VHDL Implementierung besprochen.

Erzeugung von Signalvektoren in einer Testbench

Eine Anwendung einer sequentiellen Umgebung ist die einer main-loop in einer testbench (siehe Kap Testbenches), welche eine zeitliche Abfolge von Testvektoren an die Eingänge einer Schaltung anlegt.

```

-- *** Test Bench - User Defined Section ***
tb : PROCESS
BEGIN
    wait for 500us;
    inv_dec_i <= '0';
    count_i <= "0000";

    wait for 1ms;
    count_i <= "0001";

    wait for 1ms;
    count_i <= "0110";

    wait for 1ms;
    inv_dec_i <= '1';
    count_i <= "0000";

    -- extend for all input vectors
    wait for 2ms;
    wait; -- will wait forever

END PROCESS;

-- *** End Test Bench - User Defined Section ***
END behavior;

```

Abb 21: Prozess als sequentielles Steuerelement einer Testbench

Beschreibung von kombinatorischer Logik

Ein process kann jedoch auch zur Beschreibung kombinatorischer Logik verwendet werden. Dies ist eine in der industriellen Praxis durchaus übliche Vorgangsweise.

Eine „goldene Regel“ in VHDL besagt hierzu folgendes:

To synthesize combinational logic using a process, all inputs to the design must appear in the sensitivity list.

Diese Regel sollte an einem Programmbeispiel zu einem einfachen multiplexer, welcher über einen Prozess beschreiben wird, demonstriert werden.

Ln#	
15	entity mux_simple is
16	port (
17	a_i : in std_logic; -- input
18	b_i : in std_logic; -- input
19	sel_i : in std_logic; -- input
20	y_o : out std_logic -- output
21);
22	end mux_simple;
23	
24	architecture rtl of mux_simple is
25	
26	-- enter signal declarations here
27	
28	begin
29	
30	mux: process(a_i, b_i, sel_i)
31	begin
32	if (sel_i = '1') then
33	y_o <= a_i;
34	else
35	y_o <= b_i;
36	end if;
37	end process mux;
38	
39	end rtl; -- of mux_simple
40	
41	

Abb 22: Strukturelle Beschreibung eines einfachen 1oo2 Multiplexers

Beschreibung von sequentieller Logik

Für diesen Fall ist auf das Kapitel Latch und Register, Unterkapitel Register zu verweisen.

Kontrollstrukturen in sequentiellen Umgebungen

if/then/else

Diese Anweisung ist nur in einer sequentiellen Umgebung zulässig. Sie ist daher dementsprechend in einem process zu verwenden.

Mit dieser Anweisung beliebige Blöcke von sequentiellen Anweisungen in Abhängigkeit von beliebigen Bedingungen implementiert werden.

```

if      <Bedingung_1>      then <Sequentielle Anweisungen 1>;
elsif   <Bedingung_2>      then <Sequentielle Anweisungen 2>;
elsif   <Bedingung_2>      then <Sequentielle Anweisungen 3>;
else                                <Sequentielle Anweisungen n>;
end if;

```

Abb 23: Syntax der if – then – else Anweisung

Ausgehend vom Beispiel eines simplen 1oo2 - Multiplexers (siehe Abb 22) sollte dieser auf einen Multiplexer mit mehreren Eingängen (≥ 3) erweitert werden. Dies kann mittels mehrfacher „if-end if (multiple if)“ sowie mittels „if – elsif – endif (single if)“ Kontrollstrukturen erfolgen, mit unterschiedlichen Auswirkungen auf die Schaltung welche man erhält.

Vergleiche die beiden strukturellen Modellierungen mit „multiple if“ statements und einem „single if“ statement.

Aufgabenstellungen:

Implementiere die beiden Designs in jeweils einem process in einer architecture. Mittels einer configuration sollte dann jeweils eine Implementierung ausgewählt werden können. Synthetisiere das Design auf einem CPLD/FPGA Baustein, und untersuche es im RTL viewer zur Beantwortung der nachfolgenden Fragen.

Welche strukturellen Unterschiede sind erkennbar?

Beschreibe die Unterschiede der beiden Modellierungsarten.

Warum spricht man bei diesen Designs von „priority encodern“?

Wie kann die „latency“ auf den einzelnen Eingängen gesteuert werden?

case/is

Diese Anweisung entspricht der with/select Anweisung wie sie in nebenläufigen Umgebungen zulässig ist, kann jedoch nur in einer sequentiellen Umgebung verwendet werden.

Mit dieser Anweisung beliebige Blöcke von sequentiellen Anweisungen in Abhängigkeit von einem Testsignal implementiert werden.

```

case   <testsignal> is
  when <Wert_1>    => <Sequentielle Anweisungen 1>;
  when <Wert_2>    => <Sequentielle Anweisungen 2>;
  when <Wert_2>    => <Sequentielle Anweisungen 3>;
  when others      => <Sequentielle Anweisungen n>;
end case;

```

Abb 24: Syntax der case / is Struktur

Der Vorteil der case-is Struktur ist, dass sich mit ihrer Hilfe Strukturen beschreiben lassen, welche in ihrer Laufzeit balanciert sind.

Am einfachen Beispiel eines Multiplexers kann dies sehr rasch selbst verifiziert werden. Es zeigt sich dabei, dass alle drei Eingangssignale dieselbe Signallaufzeit aufweisen.

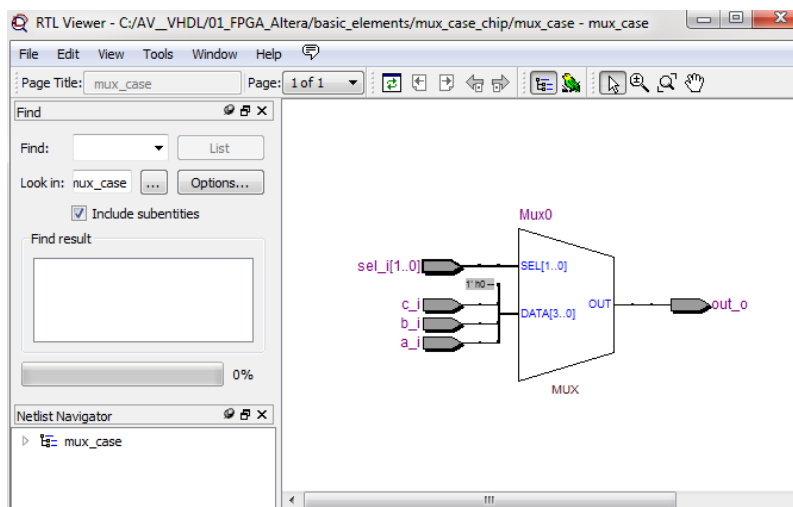


Abb 25: Implementierung eines MUX mittels case-is Anweisung

Latch und Register

Latch und Register dienen zum Zwischenspeichern von Daten.

Latch

Ein Latch (D-Latch) ist ein taktzustandgesteuertes Flipflop. Es besitzt einen enable input oder gate, den data input, sowie einen data output.

Neben der gewollten Struktur eines Latches (siehe Abb 26), gibt es den häufigeren Fall der ungewollten Implementierung eines Speicherelements. Dies ist der Fall, wenn ein Signal nicht zu jedem Zeitpunkt getrieben wird.

```

15 entity latch is
16   port (
17     d_i      : in  std_logic;      -- input
18     ena_i    : in  std_logic;      -- input
19     y_o      : out std_logic       -- output
20   );
21 end latch;
22
23 architecture rtl of latch is
24   -- enter signal declarations here
25
26 begin
27
28   latch_p: process(d_i, ena_i)
29   begin
30     if (ena_i = '1') then
31       y_o <= d_i;
32     end if;
33   end process latch_p;
34
35 end rtl; -- of latch
36

```

Abb 26: Implementierung eines taktzustandgesteuerten Flip-flop („Latch“)

Warum ist ein Latch in der Microelektronikentwicklung mit Vorsicht zu behandeln?

Latches verursachen Probleme bei automatischer Testvector-Generierung, und müssen daher eigens behandelt werden, was nicht immer eine einfache Aufgabenstellung ist.

Aufgabe: Vergleiche die unit latch mit mux_simple. Was fällt dir auf?

Register

Die Darstellung eines Flip-flops (zB DFF, Register) erfolgt in der sgn Normaldarstellung. Diese sollte in ihrer Syntax auch strikt eingehalten werden, damit die Synthesewerkzeuge fehlerfrei arbeiten können.

Die nachfolgenden Beispiele dienen zur Einarbeitung in die Thematik der sequentiellen Prozesse.

DFF mit Reset

Implementierung des Elements dff_simple (ohne enable Funktion) als unit in der library basic_elements (siehe Abb 27).

DFF mit Reset und Enable

Erweiterung des Elements dff_simple mit einem enable (ena_i) Eingang (siehe advanced topics: keyword "clock gating")

```

12 library ieee;
13 use ieee.std_logic_1164.all;
14
15 entity dff_simple is
16   port (
17     clk_i      : in  std_logic;    -- input
18     res_n      : in  std_logic;    -- input
19     ena_i      : in  std_logic;    -- input
20     d_i        : in  std_logic;    -- input
21     q_o        : out std_logic;    -- output
22   );
23 end dff_simple;
24
25 architecture rtl of dff_simple is
26   -- enter signal declarations here
27   -- signal test : std_logic;
28
29 begin
30
31   dff_simple_p: process(clk_i, res_n)
32   begin
33     if (res_n = '0') then
34       q_o <= '0';
35     elsif (clk_i'event and clk_i = '1') then
36       if (ena_i = '1') then
37         q_o <= d_i;
38       end if;
39     end if;
40   end process;
41
42 end rtl; -- of dff_simple

```

Abb 27: VHDL Implementierung einer DFF Basisstruktur

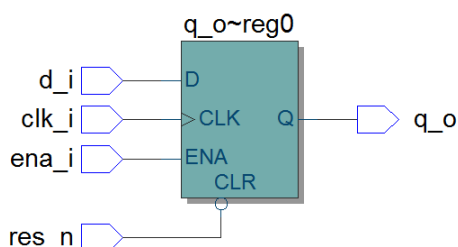


Abb 28: Syntheseresult des DFF aus der Quartus-toolchain (RTL viewer)

Anmerkung: Der Entityname DFF für das FF wäre ungeeignet, da es sich hierbei um den Namen eines Design-Primitive aus der Quartus Library handelt. Es kommt in bei einer solchen Namensgebung zu einer Doppeldeutigkeit, da es zwei unterschiedliche units mit gleichem Namen gibt, und daraus resultierend zu einem Compile-Error.

Registerarray (regbank)

Das nachfolgende Beispiel zeigt die Implementierung einer 8bit Reisterbank, als Erweiterung zum obigen Beispiel eines DFF.

```

15 entity regbank is
16   port (
17     clk_i      : in  std_logic;    -- input
18     res_n      : in  std_logic;    -- input
19     ena_i      : in  std_logic;    -- input
20     in_i       : in  std_logic_vector(7 downto 0); -- input
21     out_o      : out std_logic_vector(7 downto 0) -- output
22   );
23 end regbank;
24
25 architecture rtl of regbank is
26
27   -- enter signal declarations here
28   -- signal test : std_logic;
29
30 begin
31
32   regbank_p: process(clk_i, res_n)
33   begin
34     if (res_n = '0') then
35       -- out_o <= ("00000000"); -- reset vector
36       out_o <= (others => '0'); -- improved coding style
37     elsif (clk_i'event and clk_i = '1') then
38       if (ena_i = '1') then
39         out_o <= in_i;
40       end if;
41     end if;
42   end process;
43
44 end rtl; -- of regbank

```

Abb 29: Implementierung einer 8-bit Registerbank

Schieberegister (shiftreg) mit preload Funktion

```

15 entity shiftreg is
16   port (
17     clk_i      : in  std_logic;    -- input
18     res_n      : in  std_logic;    -- input
19     ena_i      : in  std_logic;    -- input
20     ser_in_i   : in  std_logic;    -- input
21     par_out_o   : out std_logic_vector(7 downto 0) -- output
22   );
23 end shiftreg;
24
25 architecture rtl of shiftreg is
26
27   -- enter signal declarations here
28   signal shiftreg : std_logic_vector(7 downto 0);
29
30 begin
31
32   shift_p: process(clk_i, res_n)
33   begin
34     if (res_n = '0') then
35       shiftreg <= (others => '0');
36     elsif (clk_i'event and clk_i = '1') then
37       if (ena_i = '1') then
38         shiftreg <= ser_in_i & shiftreg(7 downto 1);
39       end if;
40     end if;
41   end process;
42
43   par_out_o <= shiftreg;
44
45 end rtl; -- of shiftreg

```

Abb 30: Implementierung eines Schieberegisters mit „shift to right“ Funktion.

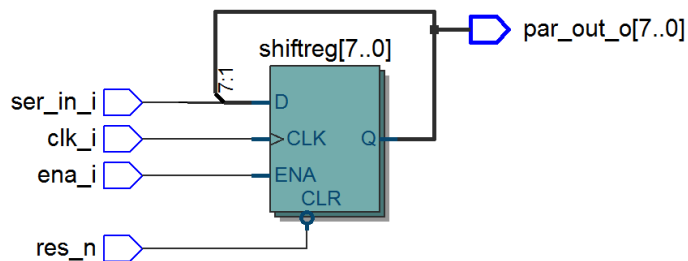


Abb 31: Quartus Syntheseergebnis des Schieberegisters

Aufgabenstellungen: Die nachfolgenden Schaltungsvarianten sind in Form von Projekten in den Übungen zu implementieren.

Takteiler

Implementierung eines DIV2 und eines DIV3 Takteilers.

Counter

Implementierung eines 6bit Zählers, mit verschiedenen Erweiterungsfunktionen.

- Der Zähler sollte bei Erreichen eines bestimmten Zählerstandes auf Null zurückspringen.
- Die Zählrichtung sollte veränderbar sein.
- Der Zähler sollte eine Preload Funktion haben.
- Der Zähler sollte bei Erreichen eines bestimmten Zählerstandes ein Bit (cnt_int) ausgeben.

```

23 architecture rtl of counter_6bit is
24
25   -- signal declaration
26   signal count : unsigned(5 downto 0);
27
28 begin
29
30   count_p: process(res_n, clk_i)
31   begin
32     if (res_n = '0') then
33       --count <= (others => '0');
34       count <= "000000";
35     elsif (clk_i'event and clk_i = '1') then
36       if (enable_i = '1') then
37         if (load_i = '1') then
38           count <= par_in_i;
39         else
40           if (up_down_i = '1') then
41             count <= count + 1;
42           else
43             count <= count - 1;
44           end if;
45         end if;
46       end if;
47     end if;
48   end process;
49
50   count_o <= count;
51
52 end rtl;

```

Abb 32: Implementierungsvariante (b) / (c) des Zählers.

Synchronizer (Metastability von Signalen)

Siehe Kopie des White Papers von Alterra.

„edge detection“

Fallstudie einer Flankenerkennung als Bausteine eines SPI Interfaces (siehe Synchronizer)

```

15 entity edge_det is
16 port (
17     clk_i      : in  std_logic;      -- input
18     res_n      : in  std_logic;      -- input
19     ena_i      : in  std_logic;      -- input
20     in_i       : in  std_logic;      -- input
21     out_o      : out std_logic;      -- output
22     rise_o     : out std_logic;      -- output
23 );
24 end edge_det;
25
26 architecture rtl of edge_det is
27     -- enter signal declarations here
28     signal del1 : std_logic;
29     signal del2 : std_logic;
30
31 begin
32     FF_p: process(clk_i, res_n)
33     begin
34         if (res_n = '0') then
35             del1 <= '0';
36             del2 <= '0';
37         elsif (clk_i'event and clk_i = '1') then
38             if (ena_i = '1') then
39                 del1 <= in_i;
40                 del2 <= del1;
41             end if;
42         end if;
43     end process;
44
45     out_o <= del2;
46     rise_o <= del1 and (not del2);
47
48 end rtl; --

```

Abb 33: Implementierung eines Synchronizers.

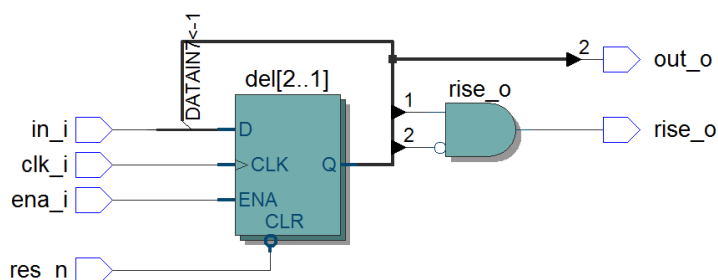


Abb 34: Syntheseresultat der Synchronizer unit.

Advanced topics:

Es ist ein in der Bitbreite konfigurierbaren SPI Interface (client) zu konfigurieren. Nachfolgend ist eine Logik zu entwerfen, welche einfache Befehle aus dem SPI Interface decodieren kann.

Mittelwertrechner (FIR Filterstruktur)

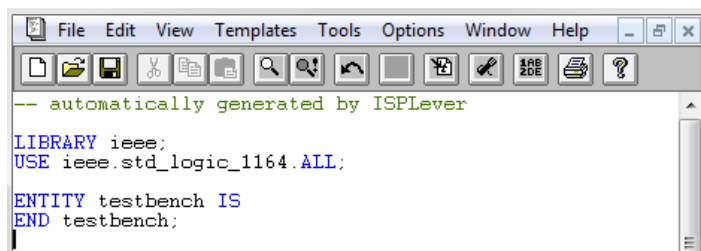
Implementierung einer FIR Filterstruktur als Mittelwertrechner. Gegeben ist die Konzeptstruktur aus Matlab.

Testbenches

Eine testbench dient der Verifikation eines designs. Ziel der testbench ist es möglichst automatisch eine Reihe von Testsignalen an ein design anzulegen, um zu bewerten ob ein design die Zielkriterien erfüllt.

Es kommt hierbei zu einer PASS/FAIL Aussage, welche ebenfalls meist automatisch ausgewertet wird.

Der Aufbau einer Testbench besteht aus einer leeren entity, da im Allgemeinen keine weiteren Signale in die testbench hinein- oder herausgeführt werden.



```

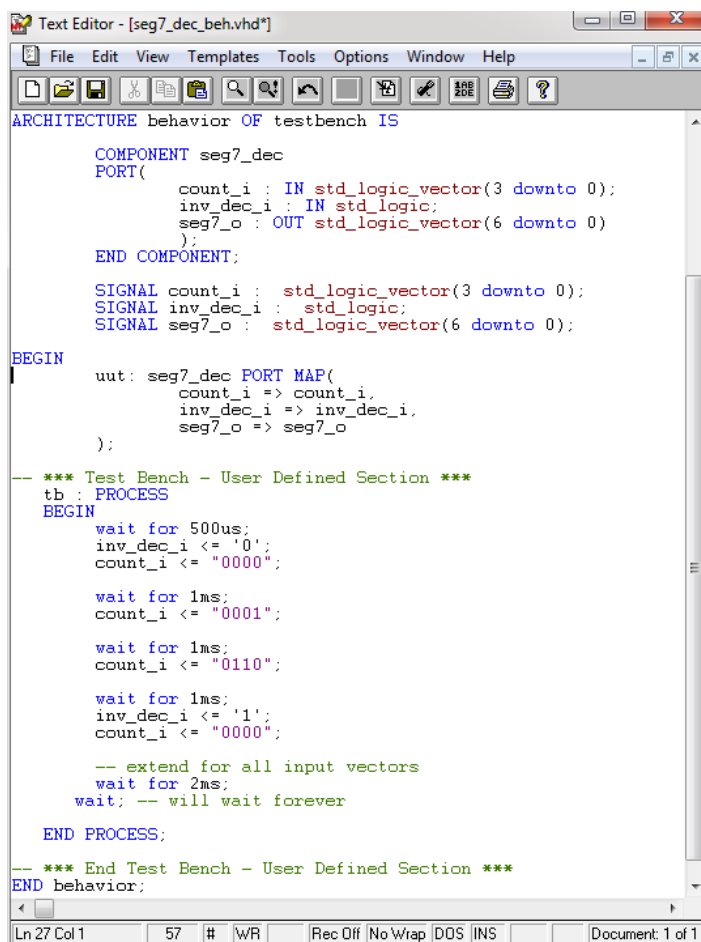
-- automatically generated by ISPLever

LIBRARY ieee;
USE ieee.std_logic_1164.ALL;

ENTITY testbench IS
END testbench;

```

Abb 35: Leere Entity einer Testbench



```

Text Editor - [seg7_dec_beh.vhd]

ARCHITECTURE behavior OF testbench IS
    COMPONENT seg7_dec
    PORT(
        count_i : IN std_logic_vector(3 downto 0);
        inv_dec_i : IN std_logic;
        seg7_o : OUT std_logic_vector(6 downto 0)
    );
    END COMPONENT;

    SIGNAL count_i : std_logic_vector(3 downto 0);
    SIGNAL inv_dec_i : std_logic;
    SIGNAL seg7_o : std_logic_vector(6 downto 0);

BEGIN
    uut: seg7_dec PORT MAP(
        count_i => count_i,
        inv_dec_i => inv_dec_i,
        seg7_o => seg7_o
    );

    -- *** Test Bench - User Defined Section ***
    tb : PROCESS
    BEGIN
        wait for 500us;
        inv_dec_i <= '0';
        count_i <= "0000";

        wait for 1ms;
        count_i <= "0001";

        wait for 1ms;
        count_i <= "0110";

        wait for 1ms;
        inv_dec_i <= '1';
        count_i <= "0000";

        -- extend for all input vectors
        wait for 2ms;
        wait; -- will wait forever

    END PROCESS;

    -- *** End Test Bench - User Defined Section ***
END behavior;

```

Abb 36: Architecture with declaration and instantiation of the component "seg7_dec".

Testbenchelemente

Unter Testbenchelementen (TBE) versteht man units, welche in die Testbench mit eingebunden werden, und welche den Ablauf der Verifikation automatisieren sollen. Im Allgemeinen versteht man darunter units zur Takterzeugung, Filereader, Filewriter oder Businterfaces.

Als Beispiel eines Testbenchelementes sollte nachfolgend eine Clock-Generation Unit (CGU) erstellt werden.

Die Aufgabe einer CGU besteht in der Erzeugung eines Taktsignals, meist in Verbindung mit einem Reset-Signal, welches mit dem zu testenden Design verbunden wird.

```

10 generate_clock: process
11 |
12 begin
13     clk_signal <= '1';
14     wait for clk_period_i * 0.5;
15     clk_signal <= '0';
16     wait for clk_period_i * 0.5;
17 end process ;
18
19 clk <= clk_signal;
20

```

Abb 37: Architecture einer sgn Clock Generation Unit.

Die entity der CGU muss nun in der Testbench als Component declariert und miteingebunden werden.

```

13
14 architecture rtl of clk_res_ctrl is
15
16     component CLOCK_GEN
17
18         port (
19             reset_length_i : in  integer := 3;
20             clk_period_i   : in  time   := 40 ns;
21             clk             : out std_ulogic;
22             reset_n         : out std_ulogic
23         );
24     end component;
25
26     signal def_reset_length : integer := 3;
27
28     begin
29
30     def_reset_length <= 3;
31
32     clk_gen_1 : clock_gen
33
34         port map(
35             reset_length_i => clk_reset_length_i,
36             clk_period_i   => clk_1_period_i,
37             clk             => clk_1,
38             reset_n         => open
39         );

```

Abb 38: Component declaration und instantiation einer CGU.

Aufgabenstellungen:

- 1) Erstelle einen 1004 Multiplexer (a_i,..., d_i) mit jeweils 2bit Signalbreite über eine „single“ / „multiple“ if-then-else Struktur. Es sollte dabei das Signal a_i am schnellsten Pfad liegen.
- 2) Erstelle einen einfachen 8bit Addierer mit zwei Eingängen. Beachte dabei die resultierende Bitbreite.
Erstelle den adder mit und ohne registered outputs.

Hierarchisches Design

Teststrategien für PLD und FPGA

Anhand einer 6bit ALU sollte die Implementierung von verschiedenen Teststrategien auf eine intuitive Art erlernt werden.

Die ALU wird mit den folgenden Funktionen realisiert:

- ADD
- SHR/SHL
- INC/DEC
- AND/OR/NOT/NAND/NOR/EXOR

Die ALU wird in VHDL beschrieben, simuliert und auf ein FPGA der entsprechenden Größe implementiert (Labor bzw Übungsteil).

Schon bei Designs dieser geringen Komplexität erkennt man in der Praxis sehr leicht, dass im Fall eines Fehlers die Diagnosemöglichkeiten nur sehr eingeschränkt möglich sind. Schließlich ist es nicht möglich das „Innenleben“ des FPGA auszumessen. Für diesen Fall müssen nun spezielle Vorkehrungen getroffen werden um das Innenleben des FPGA, dh die implementierten Schaltungsteile nach außen hin sichtbar zu machen.

Dazu stehen nun verschiedene Möglichkeiten zur Verfügung, welche im unter dem Begriff Teststrategie in der Microelektronik zusammengefasst sind.

Teststrategien:

Einführen eines Test (TST) pins, welcher den Baustein in den sgn Testmodus schaltet. Im Testmodus können nun weitere Schaltungsteile implementiert werden, die nur in diesem speziellen Testmodus ausgewählt werden können. Eine solche Teststruktur kann zum Beispiel mit einer if-then-else Kontrollstruktur implementiert werden.

- 1) Bypass von logischen Funktionen
- 2) Output-Multiplexer
- 3) Testoutputs

Simulation

Themen der Simulation

- Umgang mit Simulatorwerkzeugen im Allgemeinen
- Wave Window, Trigger Punkte einstellen
- Breakpoints setzen
- Single step
- Beobachtung von Variablen (Objects)
-

Fortgeschritten

- Verification metrics: Code coverage vs functional coverage
- Assertions
- Assertion based verification (ABV)
- Delta delays
-

Synthese

Spezialthemen

Metastability

Clock Domain Crossing

State Machines

Die Theorie zu den state machines ist im Unterricht präsentiert worden. Der Fokus liegt im Folgenden in der praktischen Realisierung in VHDL.

Entsprechend der Schaltungsskizze aus dem Theorieunterricht zum Thema Automatentheorie existieren verschiedene Ausführungsformen von state machines.

Man spricht hierbei von „single-process“, „two-process“ und sogar von „three-process“ Implementierungen.

Industriell ist (*meiner Erfahrung nach*) die Variante einer „single-process“ Implementierung am verbreitetsten.

Aus didaktischen Gründen macht es jedoch durchaus Sinn, „two-process“ Varianten vorzuziehen, und damit die nötigen Werkzeuge und Fähigkeiten zur Implementierung zu erarbeiten.

Eine Variante einer „two process“ machine trennt die Zustandsspeicherung, welche in einem Prozess abgebildet wird von der Bildung der Ausgangssignale, und der Folgezustände welches in einem weiteren Prozess erfolgt.

Der zugehörige Code wird im Unterricht besprochen und erarbeitet.

Mitschrift: *Erarbeitung der two-process machine anhand der Zustandsfolgeerkennung.*

- 1) Start mit fr, fc1 und fc2
- 2) Reduzierung des output process in den input process
- 3) Reduzierung zu einer single-process state machine

Implementierung einer state machine („Zustandsautomat“) entsprechend dem Altera Tutorial. Ausführungsform ist eine „two process implementation“. Die Eingangsschaltlogik und die Zustandsspeicherung sind in einem Prozess zusammengefasst. Die Ausgangssignale werden in einem zweiten Prozess aus den states gebildet.

http://quartushelp.altera.com/13.0/mergedProjects/hdl/vhdl/vhdl_pro_state_machines.htm

```

ENTITY state_machine IS
  PORT(
    clk      : IN    STD_LOGIC;
    input    : IN    STD_LOGIC;
    reset    : IN    STD_LOGIC;
    output   : OUT   STD_LOGIC_VECTOR(1 downto 0));
END state_machine;
ARCHITECTURE a OF state_machine IS
  TYPE STATE_TYPE IS (s0, s1, s2);
  SIGNAL state      : STATE_TYPE;
BEGIN
  PROCESS (clk, reset)
  BEGIN
    IF reset = '1' THEN
      state <= s0;
    ELSIF (clk'EVENT AND clk = '1') THEN
      CASE state IS
        WHEN s0=>
          IF input = '1' THEN
            state <= s1;
          ELSE
            state <= s0;
          END IF;
        WHEN s1=>
          IF input = '1' THEN
            state <= s2;
          ELSE
            state <= s1;
          END IF;
        WHEN s2=>
          IF input = '1' THEN
            state <= s0;
          ELSE
            state <= s2;
          END IF;
        END CASE;
      END IF;
    END PROCESS;

    PROCESS (state)
    BEGIN
      CASE state IS
        WHEN s0 =>
          output <= "00";
        WHEN s1 =>
          output <= "01";
        WHEN s2 =>
          output <= "10";
        END CASE;
      END PROCESS;

```

Abb 39:Two-process Implementierung einer state machine, Altera tutorial

Aufgabenstellungen:

- 1) Implementiere einen Glixon/Gray/Exzess-3, etc. Counter.
- 2) Implementiere ein Drehkreuz.
Es gibt den Zustand geschlossen und offen. Bei Einwurf einer Münze erfolgt der Zustandswechsel.

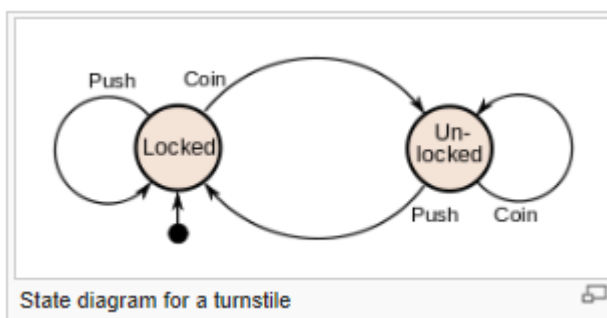


Abb 40: Quelle Wikipedia, state machine.

3) Impulsfolgeerkennung

Es soll die Zustandsfolge „01-11-10“ erkannt werden.

Die Details werden im Theorieunterricht besprochen.

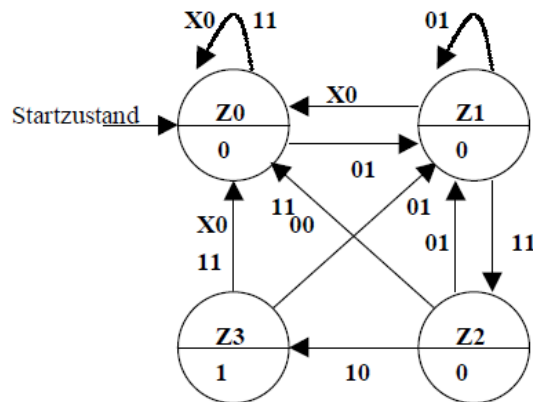


Abb 41: Moore Implementierung einer Zustandsfolgeerkennung.

- 4) Implementiere den „Fragomat“ aus dem VHDL crashkurs, in Form einer two-state machine. Die Ausgangslogik sollte von dem jeweiligen Zustand in einem separierten Prozess abgeleitet werden.

5) Ampelsteuerung

Implementiere die Ampelsteuerung einer Verkehrskreuzung.

Die Zustandsfolge ist entsprechend der deutschen Norm mit rt / rt_ye / gn / ye festgelegt.

Anhang

Altera DE0 nano board

Unklar ist die Beschreibung der sgn DIP switches am Nano-board, welche im Allgemeinen als CSQ bzw Reset Quelle verwendet werden könnten.

Abhängig von ihrer Stellung liefern die switches ein HIGH bzw LOW Signal, wie dies in Abb 42 zu sehen ist.

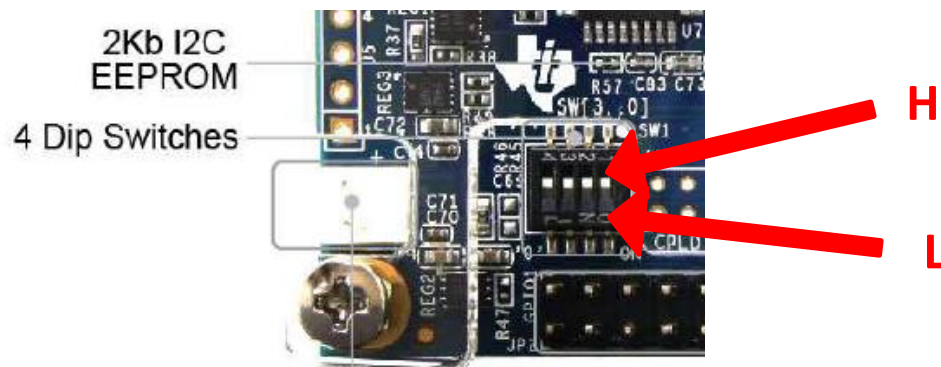


Abb 42: DIP switches am Nano-board