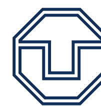
ANLEITUNG ZUM LABORPRAKTIKUMSVERSUCH



Elektronische Detektorauslese und digitale Datenverarbeitung mit FPGA

Institut für Kern- und Teilchenphysik

Dirk Duschinger

Version vom: 7. Januar 2013

Inhaltsverzeichnis

1 Kurzeinführung in die elektronische Auslese von Kalorimetern 2

2 Experimenteller Aufbau 7

3 Field Programmable Gate Arrays (FPGA) 8
3.1 Funktionsweise von FPGAs . . . 8
3.2 FPGA Logik-Bausteine . . . 10
3.2.1 Look-Up Tabellen (LUT) . . . 10
3.2.2 Register . . . 10
3.2.3 Multiplexer . . . 11
3.3 Der Xilinx Spartan-3 FPGA . . . 11

4 Programmierung von FPGAs 13
4.1 ISE: Xilinx Design-Oberfläche . . . 13

5 VHDL-Syntax 14
5.1 Einbindung der Bibliotheken . . . 14
5.2 Schnittstelle . . . 14
5.3 `architecture`-Implementierungen . . . 14
5.3.1 Deklarationsteil . . . 15
5.3.2 Anweisungsteil . . . 17

6 Aufgaben 21
6.1 Abschnitt 1 . . . 21
6.2 Abschnitt 2 . . . 22

7 Anhang 24
7.1 Logische Verknüpfungen . . . 24

1 Kurzeinführung in die elektronische Auslese von Kalorimetern

Moderne, komplexe Teilchendetektoren erfordern die Verarbeitung einer Vielzahl von digitalen Signalen in Echtzeit, wobei Datenraten von mehr als 100 TBit/s erreicht werden. Die dabei auftretenden Anforderungen, wie schneller Datentransfer, Signalfilterung- und prozessierung mit kurzer Verarbeitungszeit, geordnete Datenspeicherung und Vernetzung von Detektor-Auslesemodulen, sind ein idealer Einsatzbereich für digitale Elektronik. Dabei sind vor allem sogenannte Field Programmable Gate Arrays (FPGAs) von Interesse, da sie flexibel programmiert werden können. Mikroprozessoren, wie sie in Computern verwendet werden, sind den FPGAs in Verarbeitungsgeschwindigkeit und Datendurchsatz unterlegen und werden daher meist erst eingesetzt, wenn die Signale nicht mehr in Echtzeit verarbeitet werden müssen.

Im Forschungsbereich Elementarteilchenphysik werden verschiedenste Detektoren zum Nachweis von geladenen und neutralen Teilchen eingesetzt. Die Signale moderner Detektoren werden elektronisch ausgelesen und verarbeitet. Ein Beispiel ist das Flüssig-Argon-Kalorimeter des ATLAS-Experiments, mit welchem die Energie von Elektronen und Photonen gemessen werden. Der Detektor ist aus Schichten von Blei-Absorbern und flüssigem Argon als sensitivem Material aufgebaut (siehe Abbildung 1).

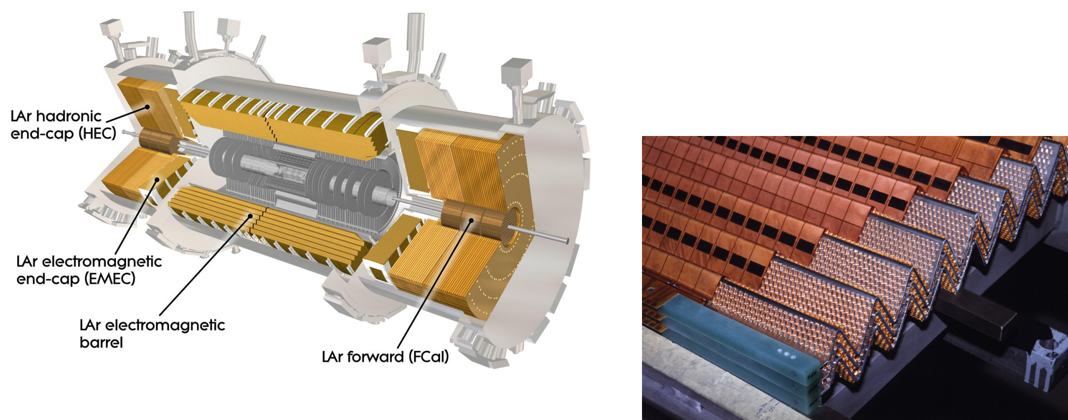


Abb. 1: Aufbau des Flüssig-Argon-Kalorimeter des ATLAS-Detektors (links), welches dem Nachweis von Elektronen und Photonen dient. Das Kalorimeter besteht aus akkordeonförmig gefalteten Blei-Absorberstrukturen (rechts), welche mit Stahlplatten stabilisiert sind. Zwischen den Absorberplatten ist flüssiges Argon als Absorber eingefüllt. Um die Ladungen, die bei der Ionisation des Argon entstehen, aufzusammeln, wird eine Hochspannung an Kupferelektroden angelegt, welche auf beiden Seiten des Absorbers montiert sind. Diese können separat angesteuert und ausgelesen werden und bilden eine Detektorzelle. Das Flüssig-Argon-Kalorimeter hat etwa 180.000 solcher Zellen. Die Ladungen driften so zu den Elektroden und erzeugen einen Strompuls, welcher linear mit der Zeit abnimmt. Die Amplitude des Pulses ist proportional zur Zahl der erzeugten Ladungsträger und somit zur in jeder Auslesezelle deponierten Energie.

Elektronen oder Photonen wechselwirken mit dem Absorbermaterial, so dass ein elektromagnetischer Schauer entsteht. Dieser Schauer besteht aus einer Kaskade von Photonen und Elektronen, welche schließlich im Detektor absorbiert werden. Die energiereichen Elektronen des Schauers ionisieren die Argon-Atome. Die dabei entstehenden Elektronen und Ionen driften zu Elektroden aus Kupfer, auf welchen ein elektronischer Puls induziert wird. Die Amplitude des elektronischen Signals ist proportional zur deponierten Energie des Teilchens, welches im Detektor absorbiert wird. So kann schließlich die Energie des Schauers aufsummiert und die Energie der Elektronen und Photonen gemessen werden. Die relative Energieauflösung eines Kalorimeters kann durch 3 Terme beschrieben werden, welche unterschiedliche Energieabhängigkeit zeigen [1]:

$$\frac{\sigma(E)}{E} = \frac{a}{\sqrt{E}} + b + \frac{c}{E} \tag{1}$$

Der erste Term ist der *stochastische Term*. Die Genauigkeit, mit welcher die Gesamtenergie gemessen wird, hängt von der Zahl N der ionisierten Atome ab, wobei diese wiederum proportional zur insgesamt deponierten Energie ist: $N \propto E$. Die statistische Genauigkeit, mit der man N bestimmen kann, beträgt entsprechend der Poisson-Statistik $\sqrt{N}$, so dass sich die relative Messgenauigkeit verhält wie:

$$\frac{\sigma(E)}{E} \propto \frac{\sigma(N)}{N} = \frac{\sqrt{N}}{N} = \frac{1}{\sqrt{N}} \propto \frac{1}{\sqrt{E}} \tag{2}$$

Bei nicht homogenen Kalorimetern, die aus mehreren Schichten Absorber und sensitivem Material bestehen (Sampling-Kalorimeter), verschlechtert sich die Auflösung entsprechend dem relativen Anteil des Absorbermaterials, in dem keine Energiedepositionen gemessen werden.

Der zweite Term ist der sogenannte *konstante Term*. Dieser hängt ab von Inhomogenitäten des Absorbermaterials, nicht-linearem Ansprechverhalten des Detektors, Verlusten durch nicht im Detektor nachgewiesene Schaueranteile (z.B. bei Beginn oder Auslaufen des Schauers außerhalb des Kalorimeters), und Inter-Kalibration der einzelnen Detektorsegmente.

Der dritte Term ist der *Rausch- oder Noise-Term*. Dieser wird verursacht durch elektronisches Rauschen der Ausleseelektronik, welches unabhängig von der Energie ist und somit die relative Auflösung proportional zu 1/E verschlechtert. Weiterhin können auch die Radioaktivität des Detektors selbst (z.B. bei Uran als Absorber) sowie eine Überlagerung des nachzuweisenden Teilchens (Elektron oder Photon) mit weiteren im Detektor absorbierten Teilchen (Pile-Up) Beiträge zum Rausch-Term liefern.

Ziel der Detektorauslese ist also unter anderem, einen gut kalibrierbaren Signalpuls zu liefern, welcher nur einen geringen Rauschanteil hat. Dies geschieht durch elektronisches Filtern der Signale, welches sowohl analog als auch digital erfolgen kann. Digitale Filteralgorithmen lassen sich leicht mit FPGA-Chips umsetzen, welche mit den Algorithmen programmiert werden können. Das elektronische Signal ist ein elektrischer Strom welcher ein spezielles zeitliches Verhalten hat. Es fällt linear von einem Maximalwert auf Null ab. Ein solches Signal hat jedoch den Nachteil, dass bei einer Überlagerung mehrerer,

zeitlich schnell aufeinander folgender Pulse die Amplitude immer weiter anwächst und die elektrische Null-Linie zu positiven Werten verschoben wird:

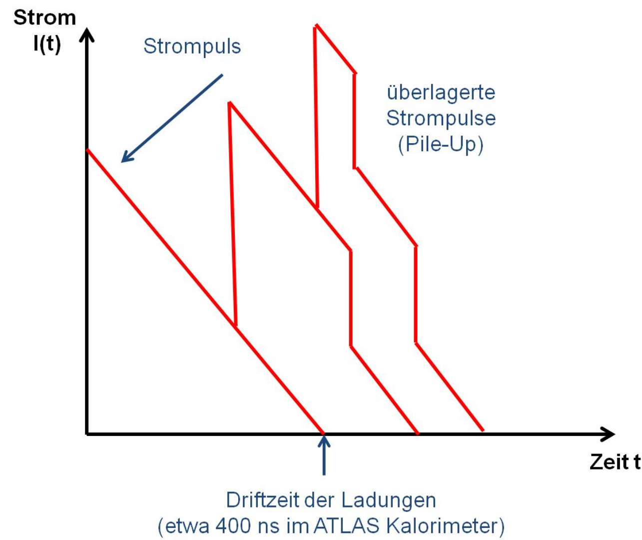


Abb. 2: Idealisierte Abfolge mehrerer Strompulse, welche durch ionisierende Teilchen erzeugt werden, welche die Detektorzellen durchqueren. Da die Pulslänge länger ist als die Zeitabstände, mit denen Teilchen den Detektor kreuzen, überlagern sich die Pulse (sogenanntes Pile-Up).

Da dies zu einem permanent anwachsenden Strom führen würde, wird der Puls elektronisch neu geformt. Mit elektronischen RC- und CR-Schaltungen wird der Puls "differenziert" bzw. "integriert". Gleichzeitig wirken diese Schaltungen als Frequenzfilter (Hochpass bzw. Tiefpass). Ein Beispiel [4] ist in Abbildung 3 gegeben.

Bei ATLAS wird eine CR-RC^2 Schaltung verwendet aus einer Folge eines CR-Gliedes mit zwei RC-Gliedern. Dabei entsteht ein sogenannter bi-polarer Puls [3, 2], wie in Abbildung 4 gezeigt. Der Puls hat die Eigenschaft, dass das Integral exakt Null ist. Daher ist auch nach Summierung vieler Pulse das Integral immer Null. Die Nulllinie bleibt also auch bei einer Vielzahl von überlagerten Signalen konstant.

Im nächsten Schritt wird der Puls digitalisiert, d.h. die Strommessung wird in diskrete (Spannungs-)Werte umgewandelt, welche später in FPGA-Chips verarbeitet werden können. Beim Kalorimeter von ATLAS wird der Puls in 4096 diskrete Werte zerlegt ($4096 = 2^{12}$, also mit 12 Bit Genauigkeit). Der Puls wird in Zeitschritten von 25 Nanosekunden (ns) abgetastet und digitalisiert. Man kann zeigen, dass man durch Multiplikation der Digitalwerte mit speziellen Konstanten das elektronische Rauschen reduzieren kann. Dies nennt man *digitale Filterung*. Gleichzeitig werden die Messwerte in Energie umgerechnet, so dass die Ausgabe des FPGA direkt für physikalische Messungen verwendet werden kann:

$$\text{Energiewert } E = a_1 S_1 + a_2 S_2 + a_3 S_3 + a_4 S_4 + a_5 S_5 \ , \tag{3}$$

wobei a_i die Filterkonstanten sind und S_i die digitalisierten Signalhöhen ($i = 1, \ldots, 5$).

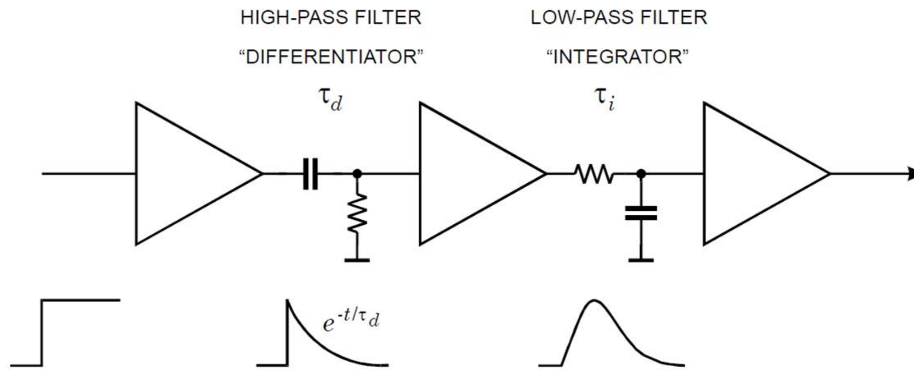
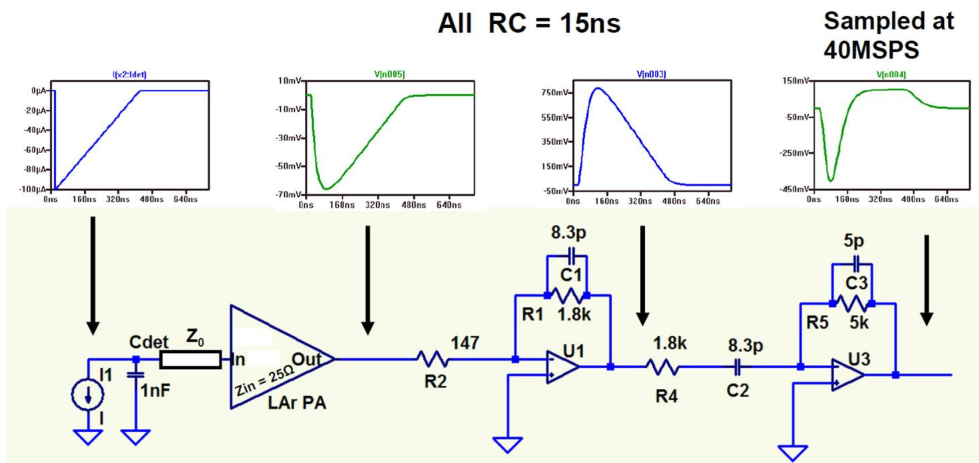


Abb. 3:
Nach Verstärkung des Signals durchläuft der Strompuls einen "Differenzierer" (CR) und einen "Integrierer" (RC). Danach hat der Puls eine neue Form und hohe sowie niedrige Frequenzanteile sind herausgefiltert. Damit erhält man einen Puls, der eine endliche Anstiegszeit hat, und dessen Maximum man einfacher bestimmen kann als das Maximum des ursprünglichen Dreieckspulses.

Bei ATLAS werden immer 5 Werte aufsummiert. Das reicht aus, um das Rauschen zu unterdrücken und die Energie ausreichend genau zu messen. Dieser Algorithmus ist in programmierbare Elektronikchips implementiert, z.B. in FPGAs. Diese können viele Signale gleichzeitig parallel berechnen, so dass eine große Datenmenge schnell verarbeitet werden kann. Die Programmierung solcher Chips erfolgt in einer speziellen Programmiersprache, VHDL, welche für die Programmierung elektronischer Signalverarbeitung optimiert ist.



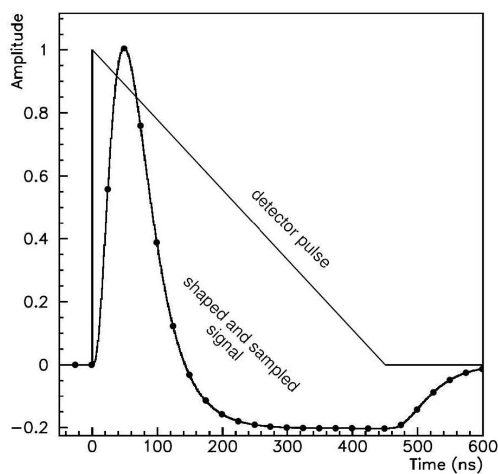


Abb. 4: Oben:
idealisiertes Schaltbild der Verstärker- und Pulsformungsstufen des Flüssig-Argon-Detektors von ATLAS. Unten: Dreieckspuls, welcher durch ionisierende Teilchen im Detektor erzeugt wurde, und geformter, sogenannter bi-polarer Puls. Der geformte Puls wird mit einer bestimmten Frequenz (alle 25 ns) abgetastet und die Werte digitalisiert. Aus diesen digitalen Werten lässt sich durch einfache Linearkombination die Pulshöhe berechnen, wenn man die Form des Pulses kennt. So kann aus diesen Digitalwerten die in jeder Detektorzelle deponierte Energie berechnet werden.

2 Experimenteller Aufbau

Der Messplatz besteht aus einem ELVIS II Entwicklungs-Board der Firma National Instruments, welches in Abbildung 5 gezeigt ist. Es ist mit Spannungsversorgungen, Strom- und Spannungsmessgeräten, einem 2-Kanal Oszilloskop zur elektronischen Signalmessung und einem Pulsgenerator ausgestattet. Die Bedienung erfolgt mit Hilfe eines PCs und einer LABVIEW Software-Umgebung. Auf einem Steckbrett können Elektronik-Bauteile verschaltet werden. Weiterhin steht eine Entwicklungsumgebung für FPGAs zur Verfügung. Der hier verwendete FPGA ist das Modell *XILINX Spartan 3*.

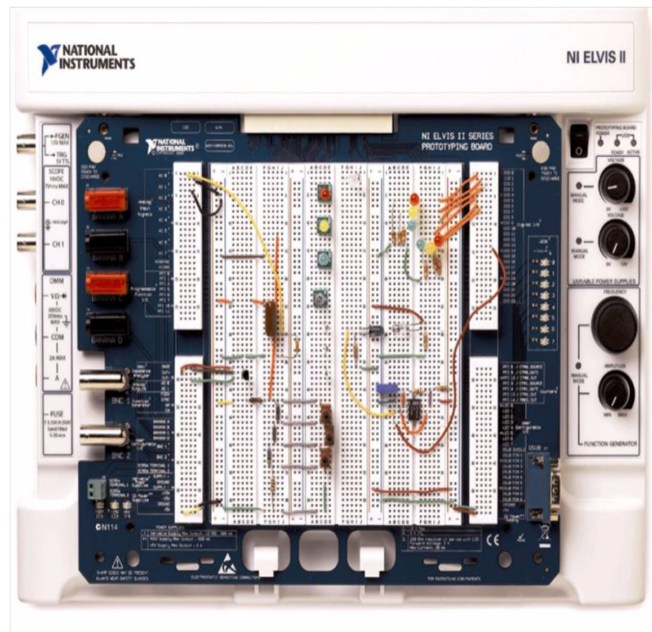


Abb. 5: ELVIS II Entwicklungs-Board.

3 Field Programmable Gate Arrays (FPGA)

3.1 Funktionsweise von FPGAs

Ein sogenanntes Field Programmable Gate Array (FPGA) is ein hochintegrierter Schaltkreis (Integrated Circuit = IC), der vom Nutzer nach der Herstellung des Chips frei konfiguriert werden kann. Er gehört somit zur Familie der programmierbaren Logikbausteine (Programmable Logic Device = PLD). FPGAs sind Alternativen zu benutzerspezifischen integrierten Schaltkreisen (Application Specific Integrated Circuit = ASIC), welche nach Herstellung nicht neu programmiert werden können und somit immer nur die gleiche Funktionionalität abrufen können. FPGAs haben somit folgende Vorteile:

- Rekonfigurierbarkeit
- Schnelle Implementierung neuer Funktionen
- Niedrige Kosten bei kleinen Stückzahlen
- Geringes Risiko eines Fehl-Designs

Aber sie haben auch Nachteile:

- Niedrigere Taktfrequenzen bei Ausführung von digitalen Operationen
- Geringere Dichte der logischen Hardwarebausteine
- Höherer Stromverbrauch bei gleicher Funkionalität
- Geringere Strahlenhärte

Wie der IC-Baustein hat ein FPGA Eingangs- und Ausgangssignale (I/O ports), Logik-Blöcke und Verbindungssignale zwischen den Logik-Blöcken. Abbildung 6 zeigt verschiedene FPGA-Architekturen bzw. Anordnungen von Logik-Blöcken und deren Verbindungen.

Die Rekonfigurierbarkeit wird durch "Schalter" zwischen den Logik-Verbindungen erreicht, die durch Anlegen von Spannungen programmiert werden können. Abbildung 7 zeigt eine solche vereinfachte Schaltmatrix. Hier gibt es verschiedene Technologien der Schalter-Programmierung, wie SRAM-basierte FPGAs (muss bei jeden Start neu programmiert werden), anti-fuse-Technologie (strahlenhart, aber nicht neu programmierbar) oder EEPROM-Technologie. Der im Versuch verwendete Spartan-3 FPGA ist ein SRAM-basierter FPGA.

Um die Logikblöcke in gewünschter Weise zu verknüpfen und zu konfigurieren, wird eine Hardware-Programmiersprache verwendet, die "Hardware Description Language" oder HDL. Die Konfigurierung des FPGA beschreibt hierbei nicht den schrittweisen Ablauf von Funktionen, wie es bei der Programmierung von Mikroprozessoren oder Mikrocontrollern geschieht, sondern ist eine abstrahierte Darstellung des im FPGA umgesetzten elektronischen Schaltplans. Es werden also elektronische Signale und deren logische Verknüpfung programmiert. Somit können auch viele Funktionen gleichzeitig

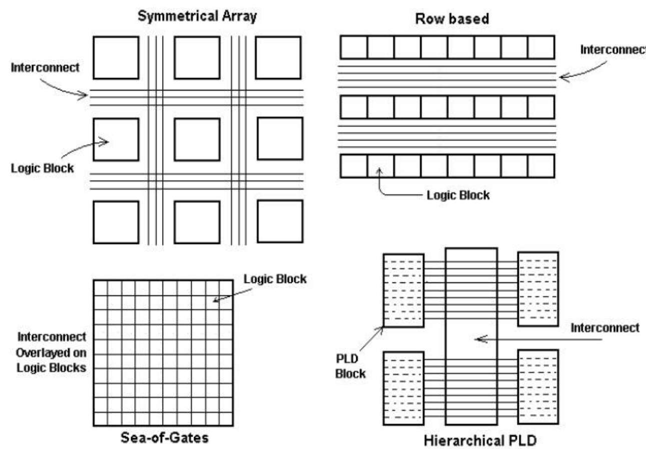


Abb. 6: FPGA-Architekturen
mit verschiedenen Anordnungen von Logik-Blöcken (Logic Block) und deren Verbindungen (Interconnect): eine symmetrisch-quadratische Anordnung, eine Reihen-Anordnung, ein "Logik-Gatter-See" (sea of gates) und hierarchisch organisierte Programmierbausteine (PLD).

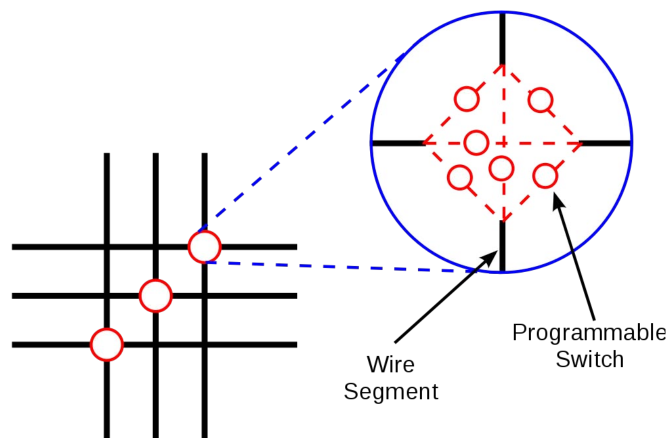


Abb. 7: Schaltmatrix als Verbindungsstruktur zwischen Logikblöcken.

und parallel ablaufen. Die Geschwindigkeit der Signalverarbeitung ist daher nur durch die Zahl der möglichen Logik-Bausteine des FPGA, die Signallaufzeit im FPGA-Chip und durch die Taktfrequenz bei digitalen Operationen begrenzt.

Die parallele Signalverarbeitung ist die Stärke des FPGA. Ein weiterer Vorteil ist die hohe Geschwindigkeit und die große Anzahl von digitalen Signal-Ein- und Ausgängen. FPGAs werden daher z.B. in der digitalen Signalverarbeitung, Prototypen-Entwicklung von ASICs, medizinischer Bildgebung, Spracherkennung, Kryptographie und Echtzeit-Bildverarbeitung verwendet.

3.2 FPGA Logik-Bausteine

Die wichtigsten logischen Komponenten eines FPGA werden hier kurz vorgestellt. Weitere Information bietet z.B. [5].

3.2.1 Look-Up Tabellen (LUT)

Look-up Tabellen sind kleine Speicherstrukturen mit n Eingängen (a_i), einem Ausgang (O) und 2^n Einträgen. Jeder Speicherblock kann mit einer beliebigen logischen Funktion der n Eingänge programmiert werden.

Beispiel einer LUT mit 3 Eingängen ist folgende Verknüpfung ($\wedge$ = UND = AND, $\vee$ = Oder = OR):

$$O = ((\bar{a}_0 \wedge a_1) \vee (\bar{a}_0 \wedge a_2)) \tag{4}$$

Die LUT kann durch ihre logische Funktion (Gl. 4) oder eine Wahrheitstabelle definiert werden (Tabelle 1). Etwas kompliziert, aber ökonomischer, kann die LUT durch die als Binärzahl interpretierte Reihenfolge der Ausgabewerte definiert werden, hier z.B. `"0010 1110"` bzw. im Hexadezimalformat "2E".

a_0	a_1	a_2	O
0	0	0	0
0	0	1	1
0	1	0	1
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	0
1	1	1	0

Tab. 1: Wahrheitstabelle der logischen Funktion $O = ((\bar{a}_0 \wedge a_1) \vee (\bar{a}_1 \wedge a_2))$.

3.2.2 Register

Register sind Speicher-Elemente, die den Datenfluss kontrollieren (Abbildung 8). Diese Kontrolle geschieht mit Hilfe eines Clock-Signals, also einem Signal, das mit einer festen Frequenz zwischen den Zuständen 0 und 1 wechselt. Anders als Look-up-Tabellen dient bei Registern dieses Clock-Signal als Schalter, um das Eingangssignal an den Ausgang weiterzuleiten. Ein Register kann noch weitere Kontrollfunktionen haben, z.B. eine Rücksetzfunktion (Reset), eine Aktivierungsfunktion (Enable) oder Initialisierungsfunktion (Initialisation), welche jeweils synchron oder asynchron mit dem Clock-Signal erfolgen können.

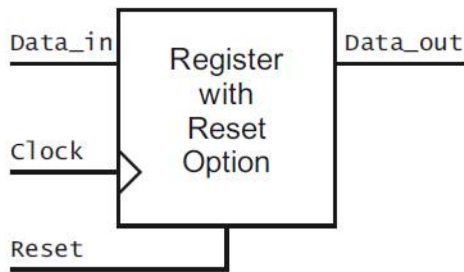


Abb. 8: Register mit Reset-Funktion. Wenn das Clock-Signal seinen Zustand ändert (z.B. $0 \rightarrow 1$), wird das an *Data_in* anliegende Signal an *Data_out* übertragen. Sonst bleibt *Data_out* im bisherigen Zustand, der z.B. durch *Reset* auf 0 gesetzt werden kann.

3.2.3 Multiplexer

Eine sehr nützliche Logik-Struktur, besonders für FPGAs, ist der Mulitplexer. Er wählt anhand eines Steuersignals die verschiedenen Eingangssignale aus, welche an den Ausgang weitergeleitet werden (Abbildung 9). Es entspricht also in einer höheren Programmiersprache der Auswahl "if then else" bei zwei Eingängen bzw. einer "case"-Anweisung bei mehr als zwei Eingängen. Man kann z.B. dadurch erreichen, dass digitale Eingangssignale, die mit einer bestimmten Takt-Frequenz anliegen, von der nachfolgenden Verarbeitungsstufe mit einem Vielfachen der Frequenz abgearbeitet werden können. So kann z.B. eine hohe Daten-Bandbreite erzielt werden.

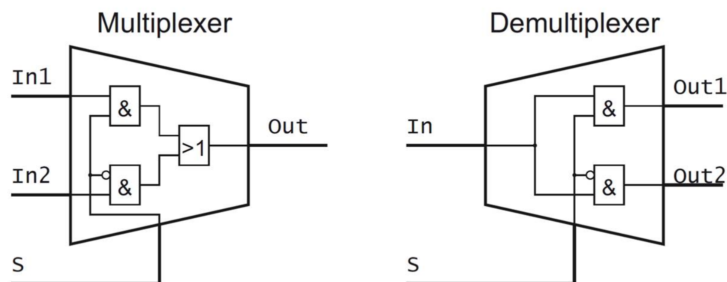


Abb. 9: Multiplexer und Demultiplexer. Mit dem Steuersignal *S* kann des jeweilige Eingangssignal oder Ausgangssignal ausgewählt werden.

3.3 Der Xilinx Spartan-3 FPGA

Im Laboraufbau wird ein Xilinx Spartan-3 FPGA verwendet (Abbildung 10). Diesen gibt es in verschiedenen Ausführungen und mit unterschiedlichen Konfigurationen. Abbildung 11 zeigt die verschiedenen Typen und Abbildung 12 den typischen Aufbau eines Spartan-3 FPGA.

Abb. 10: Xilinx Spartan 3.

Device	System Gates	Equivalent Logic Cells(1)	CLB Array (One CLB = Four Slices)			Distributed RAM Bits (K=1024)	Block RAM Bits (K=1024)	Dedicated Multipliers	DCMs	Maximum User I/O	Maximum Differential I/O Pairs
			Rows	Columns	Total CLBs						
XC3S50(2)	50K	1,728	16	12	192	12K	72K	4	2	124	56
XC3S200(2)	200K	4,320	24	20	480	30K	216K	12	4	173	76
XC3S400(2)	400K	8,064	32	28	896	56K	288K	16	4	264	116
XC3S1000(2)	1M	17,280	48	40	1,920	120K	432K	24	4	391	175
XC3S1500	1.5M	29,952	64	52	3,328	208K	576K	32	4	487	221
XC3S2000	2M	46,080	80	64	5,120	320K	720K	40	4	565	270
XC3S4000	4M	62,208	96	72	6,912	432K	1,728K	96	4	633	300
XC3S5000	5M	74,880	104	80	8,320	520K	1,872K	104	4	633	300

Abb. 11: Modell-Ausführungen von Spartan-3 FPGAs. Logic Cell = 4-input Look-Up Tabellen (LUT) plus 1 D-flip-flop. CLB = Configurable Logic Block = 8 Logic Cells. RAM=Random Access Memory (Speicherblock). DCM=Digital Clock Manager. Einer "Equivalent Logic Cell" entspricht der Anzahl der CLBs $\times$ 8 Logic Cells/CLB $\times$ 1.125 Effektivitätsfaktor.

Im Praktikum wird das Modell XC3S500E-4FTG256C mit 10476 Logikzellen verwendet. Der FPGA wird beim Start über ein PROM Flash-Memory programmiert, auf den das FPGA-Bit-File über eine USB-Schnittstelle eingespeichert werden kann. Der FPGA wird mit einer 50 MHz-Clock betrieben, welche die Basisfrequenz der logischen Operationen vorgibt. Weiterhin sind an den FPGA 32 digitale Ein- und Ausgänge angeschlossen, mit dem man Signale einspeisen oder abgreifen kann. Der FPGA ist direkt an zwei Digital-Analog- und Analog-Digital-Wandler angeschlossen. Diese können Analogsignale mit einer Taktfrequenz bis zu 10 MHz in digitale Werte umwandeln. Die Genauigkeit der Wandler beträgt 14 Bit mit einem Eingangsspannungsbereich von 2.5 V.

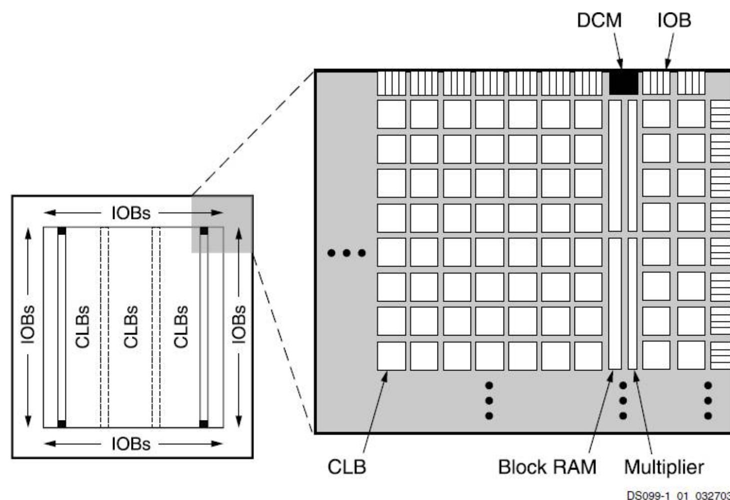


Abb. 12: Architektur des Spartan-3 mit Clock-Manager (DCM), konfigurierbaren Logikblöcken (CLB), Datenspeicher (RAM), Multiplizier-Einheiten (Multiplier) und Input/Output-Einheiten (Input/Output Block = IOB).

4 Programmierung von FPGAs

4.1 ISE: Xilinx Design-Oberfläche

Um ein FPGA-Programm (auch "Firmware") zu entwickeln, werden meist Design-Werkzeuge der FGPA-Hersteller verwendet. Die Design-Oberfläche, die zur Programmierung des Xilinx Spartan-3 FPGA verwendet wird, heißt "ISE Design Suite". Diese enthält einen Projekt-Navigator, mit dem man Programm-Projekte erstellen und bearbeiten kann. Das Programm wird dann in Hardware-Bausteine umgesetzt, synthetisiert, und schließlich in ein sogenannten Bit-File umgewandelt, mit dem der FPGA programmiert werden kann. Weiterhin kann das Verhalten von FPGA-Code simuliert werden, so dass man Programmierung und logischen Ablauf der programmierten Schaltung prüfen und korrigieren kann.

Zunächst wird in der Programmiersprache VHDL ein Programm-Code entwickelt, welcher auf Syntax und Hardware-Kompatibilität zum FPGA getestet und kompiliert wird. Dabei werden auch Probleme der Signalführung erkannt und evtl. Warnungen ausgegeben, wenn z.B. Ambiguitäten auftreten.

Im nächsten Schritt, der Implementierung des Designs, werden drei Unterprozesse ausgeführt: "Translate", "Map", "Place and Route". Dabei wird die Schaltungslogik auf die Hardware-Bausteine des FPGA übertragen und die Logik- und Speicherzellen miteinander verbunden. Schließlich wird mit "Generate Programming File" das Bit-File erzeugt zur Programmierung des FPGA.

5 VHDL-Syntax

In VHDL wird Grundsätzlich nicht zwischen Groß- und Kleinschreibung unterschieden. Kommentare werden mit "--" eingeleitet. Ein VHDL-Modul kann man in 3 Abschnitte unterteilen.

5.1 Einbindung der Bibliotheken

Im ersten werden die verwendeten Bibliotheken aufgeführt, z.B.:

```
libary IEEE;
use IEEE.STD_LOGIC_1164.ALL
```

5.2 Schnittstelle

Der zweiten Abschnitt enthält die sogenannte Schnittstelle, beispielsweise:

```
entity TOP is
    Generic(
        a : STD_LOGIC := '1';
        b : STD_LOGIC := '0';
        c : STD_LOGIC := '1'
    );
    Port(
        in1, in2, in3 : in STD_LOGIC;
        out1 : out STD_LOGIC
    );
end TOP
```

Hier sind `a`, `b`, `c`, `in1`, `in2`, `in3` und `out1` Signale vom Typ `STD_LOGIC`. Innerhalb von `Generic(...);` können allgemeine Signale stehen, die Ein- und Ausgangssignale. Zu beachten ist, dass hinter jedem Signal ein Semikolon steht, nicht jedoch hinter dem letzten in dem Generic- und dem Port-Abschnitt.

5.3 `architecture`-Implementierungen

Im dritten Abschnitt stehen eine oder mehrere Implementierungen, z.B.:

```
architecture Behavioural of TOP is
    Signal x : STD_LOGIC := '0';
begin
    x <= in1 and in2;
    out1 <= x or in3;
```

```
end Behavioural;
```

5.3.1 Deklarationsteil

Im Bereich zwischen `architecture Behavioural of TOP is` und `begin` können interne Zuweisungen, Komponenten- und Funktionen-Deklarationen erfolgen. In dem Beispiel war das das Signal `x` vom Typ `STD_LOGIC` mit dem vordefinierten Wert von `'0'`.

`STD_LOGIC` bezeichnet einen Datentyp der die in Tabelle 2 angegeben Werte annehemen kann.

Tab. 2: Mögliche Werte des Datentyps `STD_LOGIC`

Wert	Beschreibung
`0`	Logischer Wert 0
`1`	Logischer Wert 1
`U`	Uninitialisierter Wert
`X`	Unbekannter logischer Wert
`Z`	Hochohmig, Tristate-Ausgang
`W`	Schwaches Signal, unbekannt ob 0 oder 1
`L`	Schwaches Signal, könnte 0 sein
`H`	Schwaches Signal, könnte 1 sein
`-`	Don't care

Weitere wichtige Datentypen sind in Tabelle 3 aufgelistet:

Tab. 3: Auflistung einiger Datentypen in VHDL

Datentyp	Beschreibung
`STD_LOGIC_VECTOR`	Array mehrerer `STD_LOGIC`-Typen
`UNSIGNED`	Wie `STD_LOGIC_VECTOR`, nur sind mathematische Operationen möglich
`SIGNED`	Wie `UNSIGNED`, mit postiven und negativen Zahlen
`INTEGER`	Ganze Zahl

Die Länge der Arrays werden werden folgendermaßen festgelegt:

```
signal x : STD_LOGIC_VECTOR (7 downto 0);
signal y : UNSIGNED (7 downto 0);
```

Das Signal `y` hat also eine Länge von 8 Bit was den Zahlenbereich von 0 bis $2^8 - 1$ abdeckt. Wäre der Datentyp vom Typ `SIGNED`, dann wäre der Zahlenbereich von -2^7 bis $2^7 - 1$.

Zur Erleichterung für viele Programme besitzen Arrays Attribute, welche in Tabelle 4 aufgelistet sind.

Tab. 4: Attribute von Arrays am Beispiel `Signal x : STD_LOGIC_VECTOR (7 downto 0);`

Befehl	Beschreibung	Beispielausgabe
`x'length`	Feldlänge	`8`
`x'range`	Indexbereich	`7 downto 0`
`x'reverse_range`	umgekehrter Indexbereich	`0 to 7`
`x'left`	linke Grenze des Indexbereich	`7`
`x'right`	rechte Grenze des Indexbereich	`0`
`x'high`	größter Feldindex	`7`
`x'low`	kleinster Feldindex	`0`

Komponenten sind wichtige Bausteine, wenn es darum geht, bestehende Lösungen wiederzuverwenden. Sie werden mit folgender Syntax deklariert:

```
component Modul
    Generic(
        ...
    );
    Port(
        ...
    );
end component;
```

Dabei müssen die Signale innerhalb von `Generic` und `Port` mit den Abschnitten aus dem jeweiligen Modul übereinstimmen.

Die Funktionen in VHDL kann man wie Funktionen aus bekannten Programmiersprachen, wie C oder Python, betrachten. Allerdings kann jede Funktion nur einen Rückgabewert besitzen, im Gegensatz zu Python. Zur Beschreibung der Syntax soll hier wieder ein Beispiel dienen:

```
function invert_vector(x : STD_LOGIC_VECTOR) return
  STD_LOGIC_VECTOR is
    variable result : STD_LOGIC_VECTOR (x'length-1 downto 0);
begin
    for i in 0 to x'length-1 loop
        result(x'length-1-i) := x(i);
    end loop;
    return result;
end;
```

In der ersten Zeile steht der Funktionsname hinter `function`, dahinter in Klammern die Variablen die an die Funktion übergeben werden. Darauf folgen `return`, der Datentyp des Rückgabewertes und letztlich `is`. Nach dieser Zeile können verschiedene Variablen deklariert werden, wie hier die Variable `result`. Zwischen `begin` und `end` stehen wieder verschieden Anweisungen und der Befehl `return ...;` der den Rückgabewert zurückgibt.

5.3.2 Anweisungsteil

Zwischen `begin` und `end;` können Zuweisungen, Prozesse oder die Instantiierung von Komponenten stehen.

Signal-Zuweisungen erfolgen in diesem Bereich mittels mittels `<=`, Variablen-Zuweisungen mittels `:=`. Dabei sind die verschiedenen Zuweisungen vollkommen parallel zueinander. Es spielt also keine Rolle welche Zuweisung zuerst steht. Die Zuweisungen aus dem obigen Beispiel hätten also auch lauten können:

```
...
    out1 <= x or in3;
    x <= in1 and in2;
...
```

In VHDL existieren die in Tabelle 5, nach aufsteigender Priorität geordneter, Operatoren

Tab. 5: Auflistung der Operatoren in VHDL

Operatorart	Operatoren
Logisch binär	`and, or, xor, nand, nor, xnor`
Vergleich	`=, /=, <, >, <=, >=`
Additiv	`+, -`
Vorzeichen	`+, -`
Multiplikativ	`*, /, mod, rem`
unär	`not, abs, **`

Zuweisungen

Zusätzlich zu den unbedingten Zuweisungen wie `x <= in1 and in2;` gibt es noch bedingte Zuweisungen wie:

```
out1 <= '1' when in1 = '1' else
        '1' when in2 = '1' else
```

```
          '0';
```

Bei der bedingten Zuweisung wird der erste Treffer dem Wert zugewiesen. In dem Beispiel würde also, wenn `in1` oder `in2` den Wert `'1'` haben, `out1` der Wert `'1'` zugewiesen, wenn keiner von beiden `'1'` ist, dann bekommt `out1` den Wert `'0'`.

Des Weiteren gibt es noch Auswahlzuweisungen wie zum Beispiel (`in` sei vom Typ `STD_LOGIC_VECTOR (3 downto 0)`):

```
with in select out1 <=
    in1 when "0000" | "0001",
    in2 when "1100",
    in3 when others;
```

Mit Hilfe der Auswahlzuweisung kann eine Fallunterscheidung durchgeführt werden. Hier wird `out1` der Wert `in1` zugewisen, wenn `in` den Wert `"0000"` oder `"0001"` hat. Ist der Wert von `in` gleich `"1100"`, dann wird `out1` der Wert `in2` zugewiesen. Für alle anderen Fälle wird `out1` der Wert von `in3` zugewiesen. Allerdings sollten alle nicht aufgezählten Fälle zum Schluss mittels `others` abgedeckt werden.

Prozesse

Wie schon erwähnt, können zwischen `begin` und `end` auch Prozesse stehen. Ein Beispiel:

```
process(in1, in2)
    Signal x : STD_LOGIC := '0';
    variable v : STD_LOGIC := '0';
begin
    v := in1 xor in2;
    x <= in1 nand v;
end process;
```

In den Klammern nach `process` stehen alle Eingänge, auf die der Prozess sensitiv sein soll, das heißt, dass der Prozess nur ausgeführt wird, wenn eine oder mehrere der darin stehenden Signale sich verändert. Danach können wieder mehrere interne Signal/Variablen-Deklarationen erfolgen. Zwischen `begin` und `end` befinden sich die Zuweisungen. Hier ist zu beachten, dass die Reihenfolge der Zuweisungen hier im Gegensatz zu der `architecture`-Umgebung eine Rolle spielt. Innerhalb der Prozesse können auch bedingte Zuweisungen und Auswahlzuweisungen gemacht werden, allerdings mit etwas anderer Syntax.

Beispiel für eine bedingte Zuweisung:

```
if in1 = '1' then
    ...
```

```
elsif in2 = '1' then
    ...
else
    ...
end if;
```

Auswahlzuweisung:

```
case in1 is
    when "0000" | "0001" =>
        ...
    when "1100" =>
        ...
    when others =>
        ...
end case;
```

Zusätzlich kann man noch Schleifen programmieren. Diese haben die Form:

```
for i in 0 to 3 loop
    in(i) <= '0';
end loop;
```

Instantiierung von Komponenten

Die Instantiierung von Komponenten soll wieder an einem Beispiel erklärt werden. Sei die Komponente im Deklarationsteil folgendermaßen definiert:

```
component Modul is
    Generic (
        x : integer := 5;
    );
    Port (
        in1 : in STD_LOGIC_VECTOR (3 downto 0);
        in2 : in STD_LOGIC_VECTOR (7 downto 0);
        out1 : out STD_LOGIC;
    );
end component;
```

Dann lautet die Syntax für die Instantiierung:

```
name : Modul
```

```
    Generic map(
        x => 10
    )
    Port map(
        in1 => a,
        in2 => b,
        out1 => c
    );
```

Dabei ist `name` eine beliebige Bezeichnung, die allerdings keinem deklarierten Signal oder einer deklarierten Variable entsprechen darf. Darauf folgt die `Generic Map`, welche fakultativ ist, und die `Port Map`. Darin müssen alle Signale des Moduls mit Signalen verknüpft werden, welche den gleichen Typ besitzen.

Eine weitere nützliche Funktion könnte diese sein (im Hinblick auf den Aufgabenabschnitt 2):

```
name : for i in x to y generate
 begin
  ...
 end generate;
```

Mit dieser Schreibweise kann man mehrere Anweisungen in Abhängigkeit vom Parameter `i` Parallel erstellen. Dabei läuft `i` sukzessive von `x` bis `y`. `x` und `y` müssen allerdings konstante Größen sein.

6 Aufgaben

Der Praktikumsversuch ist in zwei Abschnitte aufgeteilt:

- Einarbeitung in die Programmierumgebung für FPGAs mit Beispielen.
- Aufbau eines analogen Pulsformers und eines Digitalfilters zur Signalprozessierung mit Hilfe eines FPGA. Auswertung von eingespeißten Messdaten und Optimierung der Energieauflösung für eine Detektorzelle.

6.1 Abschnitt 1

Problemstellung 1:

- Die LED 0 soll leuchten, wenn die Schalter 0 bis 2 angeschaltet sind
- Die LED 1 soll leuchten, wenn mindestens 2 der Schalter 0 bis 2 angeschaltet sind
- Die LED 2 soll leuchten, wenn der Schalter 2 angeschaltet ist und der Button 0 nicht gedrückt wird

Aufgabenstellungen:

1. Überlegen Sie sich im Vorhinein wie die Schaltung aussehen könnte und zeichnen Sie diese auf!
2. Übertragen Sie die Schaltung auf das Board und testen Sie diese! Dazu stehen die in Abbildung 13 dargestellten logischen Verknüpfungen (siehe Kapitel 7.1) zur Verfügung!
3. Lassen sie die Schaltung durch den Betreuer überprüfen!
4. Lösen Sie nun die Problemstellung in VHDL-Code und übertragen die .bit Datei auf das FPGA-Board!
5. Schauen Sie sich unter dem Punkt "Synthesize - XST -> View RTL Schematic" die vom Programm erstellte Schaltung an!

Problemstellung 2:

- Programmieren Sie einen Sekundenzähler mit Hilfe der 7-Segment-Anzeigen
- Testen Sie jedes ihre Module jeweils mit geeigneten Beispielen

Aufgabenstellungen:

1. Schreiben Sie ein VHDL-Modul `single_disp`, welches einen `STD_LOGIC_VECTOR` als Eingabe hat und auf einer 7-Segment-Anzeige die Zahlen von 0 bis 9 darstellen kann.

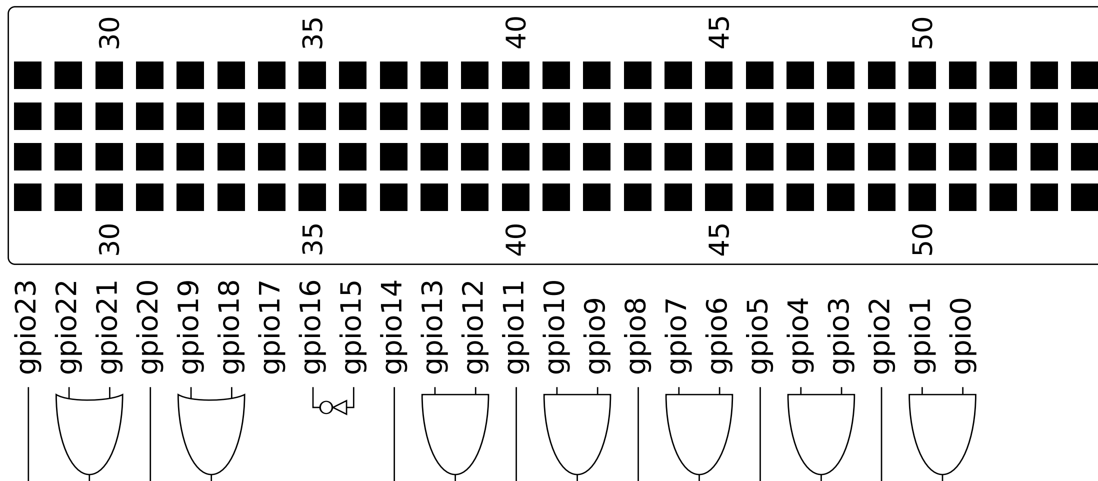


Abb. 13: Logische Verknüpfungen

2. Schreiben Sie ein weiteres VHDL-Modul `double_disp`, welches einen `STD_LOGIC_VECTOR` als Eingabe hat und mit dem vorher programmierten Modul `single_disp` und dem zur Verfügung gestellten Modul `n-m-bit_divider` die Zahlen von 00 bis 99 auf beiden 7-Segment-Anzeigen darstellen kann.

3. Schreiben Sie ein drittes Modul, welches die System-Clock benutzt um einen `STD_LOGIC_VECTOR` kontinuierlich jede Sekunde um 1 zu addieren und letztlich auf den 7-Segment-Anzeigen darzustellen. Entspricht der `STD_LOGIC_VECTOR` einer Zahl größer als 99, so soll er wieder auf 0 zurückgesetzt werden.

6.2 Abschnitt 2

Problemstellung 3:

- Programmieren Sie einen Addierer
- Programmieren Sie einen Multiplizierer

Aufgabenstellungen:

1. Schreiben Sie ein VHDL-Modul `add` welches zwei `STD_LOGIC_VECTOR`en beliebiger Längen `m` und `n` als Eingabe bekommt und binär die Summe der beiden berechnet. Überlegen Sie sich dazu wie lang der `STD_LOGIC_VECTOR` für das Ergebnis mindestens sein muss um alle Möglichkeiten abzudecken und was sie zusätzlich für Signale deklarieren müssen.

2. Schreiben Sie ein VHDL-Modul `mult` welches zwei `STD_LOGIC_VECTOR`en beliebiger Längen `m` und `n` als Eingabe bekommt und binär das Produkt der beiden berechnet. Nutzen Sie dazu das vorher geschriebene Modul `add`.

Überlegen Sie sich wieder wie lang der `STD_LOGIC_VECTOR` für das Ergebnis sein muss und was sie zusätzlich für Signale deklarieren müssen.

Bemerkungen:

Binäre Addition: Das binäre Addieren funktioniert genauso, wie das Addieren im Dezimalsystem, nämlich mit Hilfe von Merkbits. Man addiert zunächst die beiden ersten Bits, sollten diese eine Zahl größer als Eins im Dezimalsystem ergeben, dann wird im darauf folgenden Merkbit eine `1` geschrieben und in das zugehörige Ergebnisbit eine `0`, bzw. eine `1`. Danach werden die nachfolgenden Bits nach dem gleichen Schema abgearbeitet, nur dass jetzt auch noch das Merkbit addiert werden muss. Ein Beispiel ($430 + 43 = 473$) soll das verdeutlichen:

```
a = 110101110
b =    101011
m = 001011100
-------------
s = 111011001
```

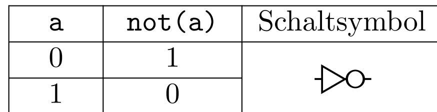
Binäre Multiplikation: Wieder ist die Multiplikation im Dualsystem analog zu der im Dezimalsystem. Zur Veranschaulichung soll ein Beispiel ($430 \cdot 41 = 17630$) dienen:

```
110101110  x  101001=
---------------------
    110101110
 +   000000000
 +    110101110
 +     000000000
 +      000000000
 +       110101110
---------------------
 = 100010011011110
```

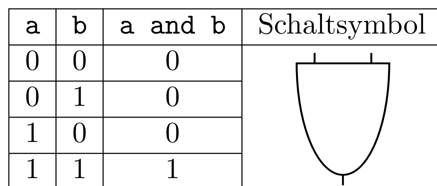
7 Anhang

7.1 Logische Verknüpfungen

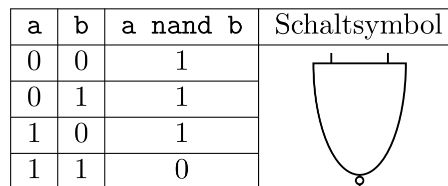
Tab. 6: Logische Verknüpfungen mit Schaltsymbolen nach Ansi-Standard



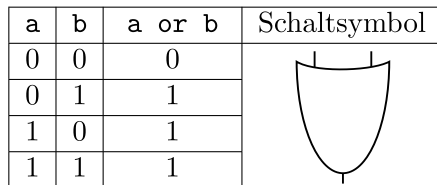
a	not(a)	Schaltsymbol
0	1	
1	0	



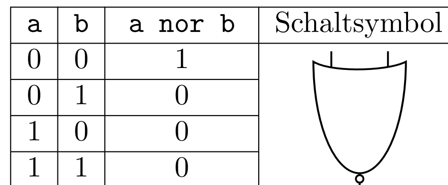
a	b	a and b	Schaltsymbol
0	0	0	
0	1	0	
1	0	0	
1	1	1	



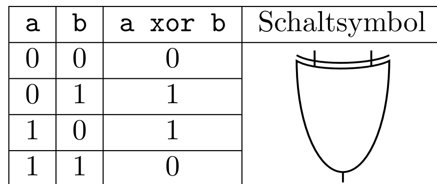
a	b	a nand b	Schaltsymbol
0	0	1	
0	1	1	
1	0	1	
1	1	0	



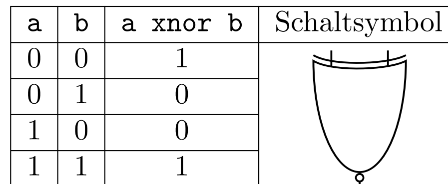
a	b	a or b	Schaltsymbol
0	0	0	
0	1	1	
1	0	1	
1	1	1	



a	b	a nor b	Schaltsymbol
0	0	1	
0	1	0	
1	0	0	
1	1	0	



a	b	a xor b	Schaltsymbol
0	0	0	
0	1	1	
1	0	1	
1	1	0	



a	b	a xnor b	Schaltsymbol
0	0	1	
0	1	0	
1	0	0	
1	1	1	

Literatur

[1] Grupen, Shwartz, “Particle Detectors”, Cambridge University Press, Auflage 2, 2008.

[2] Steffen Stärz, “Development of Digital Signal Processing with FPGAs for the Readout of the ATLAS Liquid Argon Calorimeter at HL-LHC”, Diplomarbeit, CERN-THESIS-2010-174-1.

[3] Mitch Newcomer, “LAPAS: A SiGe Front End Prototype for the Upgraded ATLAS LAr Calorimeter”, TWEPP Conference, 2009.

[4] Helmut Spieler, “Analog and digital electronics”, LBNL, `www-physics.lbl.gov/~spieler/ICFA_Rio_2003/text/Analog_and_Digital_Electronics_for_Detectors.pdf`

[5] P. Zainalabedin Navabi, “Embedded Core Design with FPGA”, The McGraw-Hill Companies, 1. Auflage, 2007.