

Du wirst



VHDL lieben!

Auszug Lehrplan

Die Schülerinnen und Schüler können im III. Jahrgang im Bereich Embedded Systems...

kombinatorische Systeme mit Hilfe einer Hardwarebeschreibungssprache spezifizieren und simulieren.

1 Rekonfigurierbare Logik

Während der Design-Phase eines eingebetteten Systems ist ein vollständiger Hardware-Nachbau (Aufbau) in Form von anwendungsspezifischen integrierten Schaltkreisen (ASICs) meist zu teuer. Die Nachbildung mit reiner Software (z.B. mittels Simulation oder durch den Einsatz von SW-Emulatoren) benötigt häufig zu lange Rechenzeiten, die nicht tolerierbar sind. In der Industrie wird daher meist während der Prototyping-Phase oder bei kleinen Stückzahlen mit rekonfigurierbaren Logikbausteinen gearbeitet.

Ein Prototyp darf in der Regel mehr kosten (und wird es auch!) als das endgültige System. In einem frühen Prototyping-Stadium sind meist die Anforderungen an Stromverbrauch oder Zeitbedingungen noch geringer, auch kann der Prototyp vom endgültigen System noch abweichen (so kann ein späterer ASIC sowohl analog- als auch digitale Komponenten beinhalten, die während der Prototyping-Phase noch getrennt aufgebaut/realisiert sind).

Der Entwurfsprozess eines ASICs ist langwierig und teuer – wenn das zu erwartende Marktvolumen die ASIC-Entwicklungskosten nicht amortisiert, da z.B. die Stückzahlen zu gering sind, kann rekonfigurierbare Logik auch im endgültigen System verbleiben. Ein weiterer wesentlicher Vorteil im Verbleib von rekonfigurierbarer Logik in einem Produkt besteht darin, dass durch Firmware-Update alle Freiheitsgrade in der Funktionsänderung/Modifikation bestehen bleiben (so kann z.B. eine komplette Softcore-Architektur ausgetauscht werden)

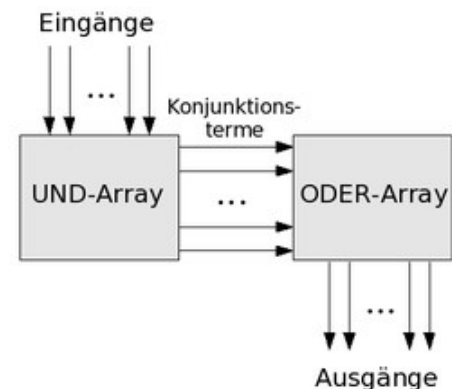
Der FPGA ist die am häufigsten eingesetzte Form rekonfigurierbarer Logik.

Über die Jahre wurden verschiedene Typen/Arten von programmierbaren Bausteinen entwickelt und eingesetzt – der Oberbegriff PLD (*Programmable Logic Device*) wurde geboren. Auch heute bestehen viele PLD-Bausteine intern aus matrixförmigen Koppelfeldern, wobei die Anzahl an verfügbaren Koppelpunkte über die Jahre ständig erhöht wurde, so dass sich heutzutage umfangreiche digitale Schaltungen in einem einzigen PLD realisieren lassen.

Die Grundstruktur der meisten „einfachen PLDs“ besteht aus einem Array aus UND-Verknüpfungen als Eingangsfeld und einem Array aus ODER – Verknüpfungen als Ausgangsfeld. Dieser Aufbau hat sich aus der Tatsache entwickelt, dass sich jede logische Gleichung als UND-Verknüpfung der Eingangsvariablen und ODER-Verknüpfung dieser Terme darstellen lässt (→ disjunktive Normalform):

$$Y1 = X0 \cdot \overline{X1} \cdot X2 \vee \overline{X0} \cdot X1 \cdot X2 \vee X3$$

$$Y2 = \overline{X0} \cdot \overline{X1} \cdot X2 \vee X1 \cdot X2 \cdot X3 \vee \dots$$



Grundprinzip (Quelle: wikipedia)

Beispiel: Aufbau eines UDB (universell programmierbarer Block) bei Cypress / PSoC

Figure 21-1. UDB Block Diagram

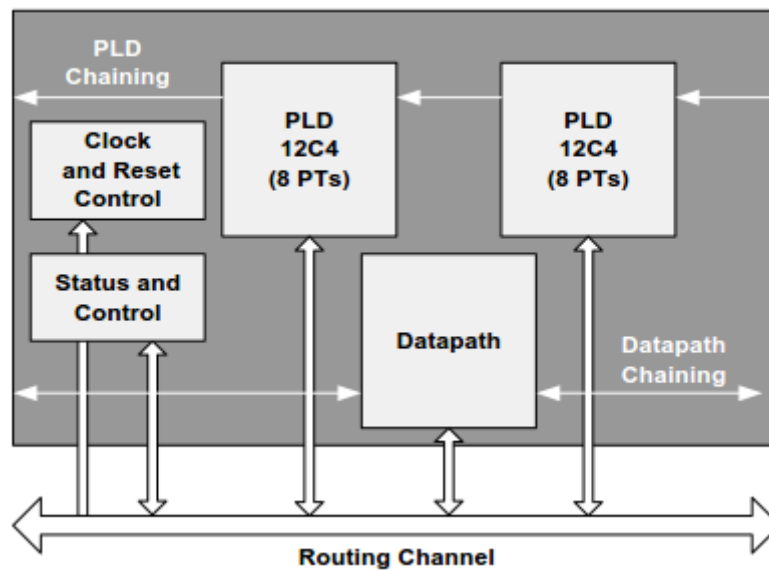
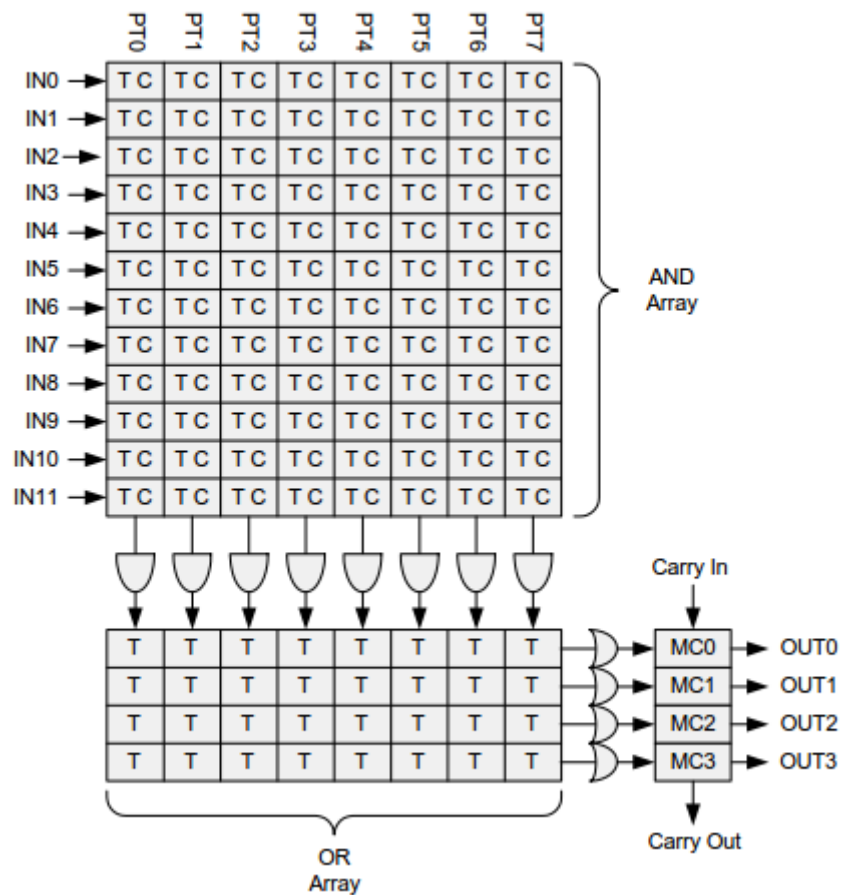
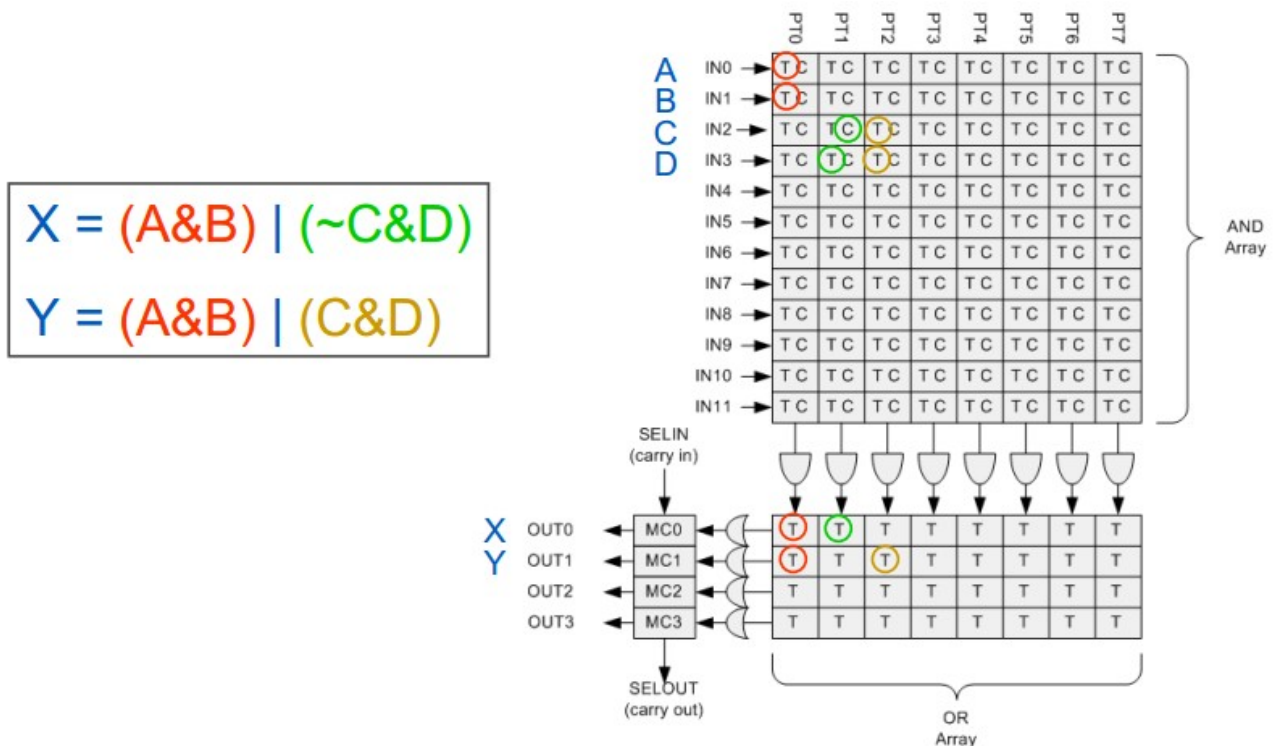


Figure 21-2. PLD 12C4 Structure





Ein weiteres, wichtiges Unterscheidungsmerkmal von PLDs ist Löscharkeit bzw. die Möglichkeit einer Neuprogrammierung. Es gibt PLDs, die als OTP angeboten werden (*OTP: One-Time-Programmable*). Die Preise für diese PLDs sind in der Regel sehr niedrig. Andere PLDs basieren auf Flash-Speicher-Architekturen und haben nur eine begrenzte Anzahl an Schreibzyklen (d.h. sie können nicht beliebig oft neu programmiert werden; meist wird von den Herstellern ein Wert von 1.000 – 10.000 Schreibzyklen angegeben). Wiederum andere PLD Bausteine basieren auf SRAM-Zellen und müssen vor jeder Inbetriebnahme erst geladen werden. Diese Bausteine sind beliebig oft rekonfigurierbar, benötigen jedoch zur Konfiguration einen (meist) externen (Flash-) Speicher oder uC.

- **Programmierbare Festwertspeicher (PROM)**

... besteht aus einem festen UND-Array und einem programmierbaren ODER-Array. Es ist der wohl am längsten bekannte anwenderprogrammierbare Baustein. Die Eingänge werden als Adress-Eingänge bezeichnet und nach Anlegen einer Adresse erscheint am Ausgang meist in paralleler Form das (meist 8bit breite) Codewort. Daher kann man alle PROMs als Codewortwandler verwenden – jedem n-bit-breitem Codewort am Eingang (Adresse) kann ein m-bit-breites Codewort am Ausgang fest zugeordnet werden.

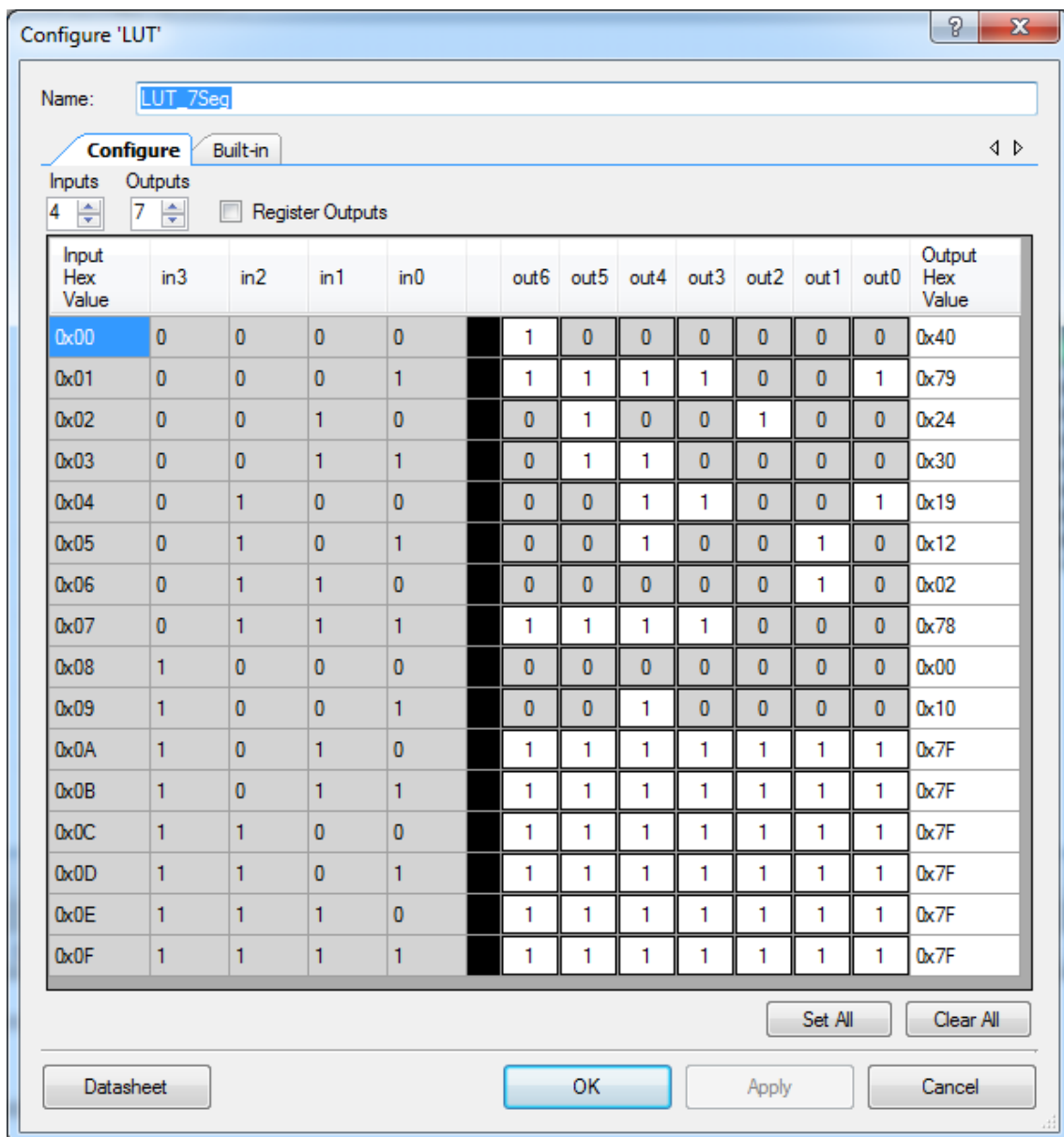
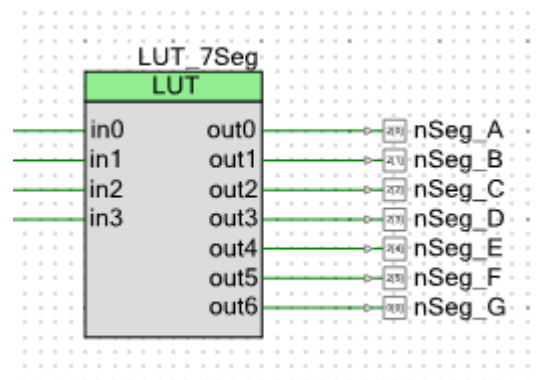
Prinzipiell lässt sich eine Wahrheitstabelle in einem PROM direkt ablegen. Aufgrund fehlender Flip-Flops kann allerdings nur kombinatorische Logik realisiert werden.

PROMs können daher als eine der ersten LUT (*Look-Up-Tables*) bezeichnet werden.

PROMs als *programmable read-only-memory* sind per Definition nur einmalig programmierbar, wohingegen EPROMs (*erasable PROM*) und EEPROMs (*electrically erasable PROM*) mehrmals wiederbeschreibbar sind.

Anwendungsbeispiel im heutigen Kontext:

BCD zu 7-Segment-Dekoder als LUT Realisierung (7-Segment-Anzeige mit gemeinsamer Anode):

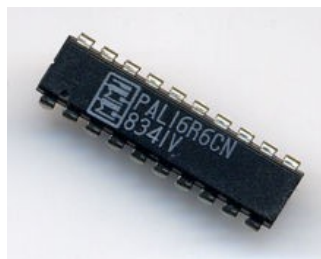


Screenshot: PSoC Creator

- **Programmable Array Logic (PAL) bzw. Generic Array Logic (GAL)**

Der Begriff PAL ist ein eingetragenes Warenzeichen der Firmen AMD und MMI (Monolithic Memories Inc.), GAL ist ein eingetragenes Warenzeichen von Lattice Semiconductor Corporation.

Ein PAL besteht aus einem programmierbaren UND Array als Eingangsfeld und einem festverdrahteten ODER-Array als Ausgangsfeld. Ein GAL besitzt eine PAL-ähnliche Struktur und ist wiederbeschreibbar (wohingegen die ersten PALs nur einmalig programmiert werden konnten). MMI hat 1978 die PAL-Struktur entwickelt und über einen langen Zeitraum waren PALs und GALs die wichtigsten Vertreter programmierbarer Logikbausteine und wurden zum Beispiel gerne für Adress-Decodierungs-Aufgaben eingesetzt. Es waren vielfältige Varianten erhältlich, deren Unterschiede vor allem in der Ausführung des Ausgangsfeldes bestanden. Bei den Low (L) - Typen wurde ein NOR-Array verwendet, d.h. die Ausgänge waren gegenüber den High (H) – Typen negiert. Es wurden auch PAL Bausteine mit internen Registern (R) angeboten (in der Regel D-FFs). Diese FFs hatten einen gemeinsamen Takt-Anschluss und die meisten Ausgangssignale waren intern in die programmierbare UND-Matrix zurückgeführt. Dadurch sparte man sich die externen Rückkopplungsleitungen und musste für diese Aufgaben auch keine (wertvollen) Eingänge verschwenden. Damit konnten synchrone Schaltwerke ohne zusätzliche Komponenten realisiert werden.



PAL vom Typ 16R6 von MMI (Quelle: wikipedia)

Ein GAL beinhaltet -im Unterschied zu den PALs- programmierbare Makrozellen, die dem Baustein ermöglichten, sämtliche PAL Architekturen (H(igh), L(ow), R(egister), B(idirektional), S(haring), ...) nachzubilden.

Da die Begriffe PAL und GAL markenrechtlich geschützt sind, konnten andere Firmen diese Bausteine mit diesen Originalbezeichnungen nur auf Lizenz-Basis fertigen. AMD hat daher ihre programmierbaren Logikbausteine umbenannt und diese trugen die Bezeichnung PALCE - die Unterschiede lagen nur im Programmieralgorithmus, nicht in der Pinbelegung und Funktion¹.

Zu diesem Zeitpunkt wurden auch die ersten PLD-Programmiersprachen und -Compiler entwickelt. Die Tools und Beschreibungssprachen waren in der Regel herstellerabhängig, jedoch konnte meist eine Logikbeschreibung auf mindestens eine der drei unterschiedlichen Arten durchgeführt werden:

- mittels Wahrheitstabelle (Funktionstabelle)
- mittels Bool'schen Gleichungen
- mittels Finite-State-Maschine (FSM) - Beschreibung

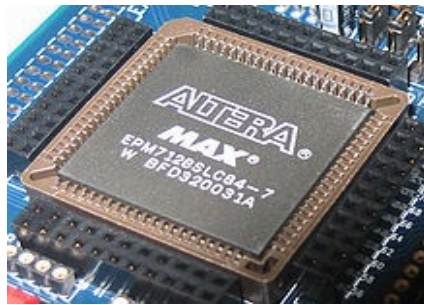
Bausteine, deren UND- als auch ODER-Matrix programmierbar waren, wurden als PLA (*Programmable Logic Array*) bezeichnet.

¹ http://de.wikipedia.org/wiki/Generic_Array_Logic

Seit Mitte der 1990er Jahre werden PALs/GALs in der Neuentwicklung kaum mehr eingesetzt und wurden fast vollständig durch CPLDs (*Complex Programmable Logic Arrays*) abgelöst.

- **Complex Programmable Logic Array (CPLD)**

CPLDs bestehen im Prinzip aus mehreren Standard-PLDs (diese werden hier meist Makrozellen genannt), die über interne Busse miteinander verbunden sind.



CPLD vom Typ EPM7128 von Altera (jetzt Intel; Quelle: wikipedia)

CPLDs bestehen im Wesentlichen aus

- programmierbaren Makrozellen
- Ein-/Ausgabeblocken

Je nach CPLD-Hersteller und CPLD-Familie sind die Makrozellen und Ein-/Ausgabeblocke unterschiedlich aufgebaut. In der Regel besitzt jede Makrozelle ein Flip-Flop sowie eine Look-Up-Table zur Realisierung kombinatorischer Logikterme und für die FF-Ansteuerung. Mehrere Ein-/Ausgabepins sind zu Ein-/Ausgabeblocken zusammengefasst und meist ebenfalls einstellbar – d.h. TTL oder CMOS Pegel, programmierbare Pull-Up/Down Widerstände, Slew-Rate Einstelloptionen, Tristate- und Schmitt-Trigger-Optionen, ...

Einige CPLD-Modelle besitzen auch einen internen Speicher und/oder einen internen Taktgenerator. Meist gibt es einige CPLD-Anschlüsse mit Sonderfunktionen wie zum Beispiel dedizierte Takteingänge, Globale Reset-Eingänge oder JTAG-/ISP-Anschluss.

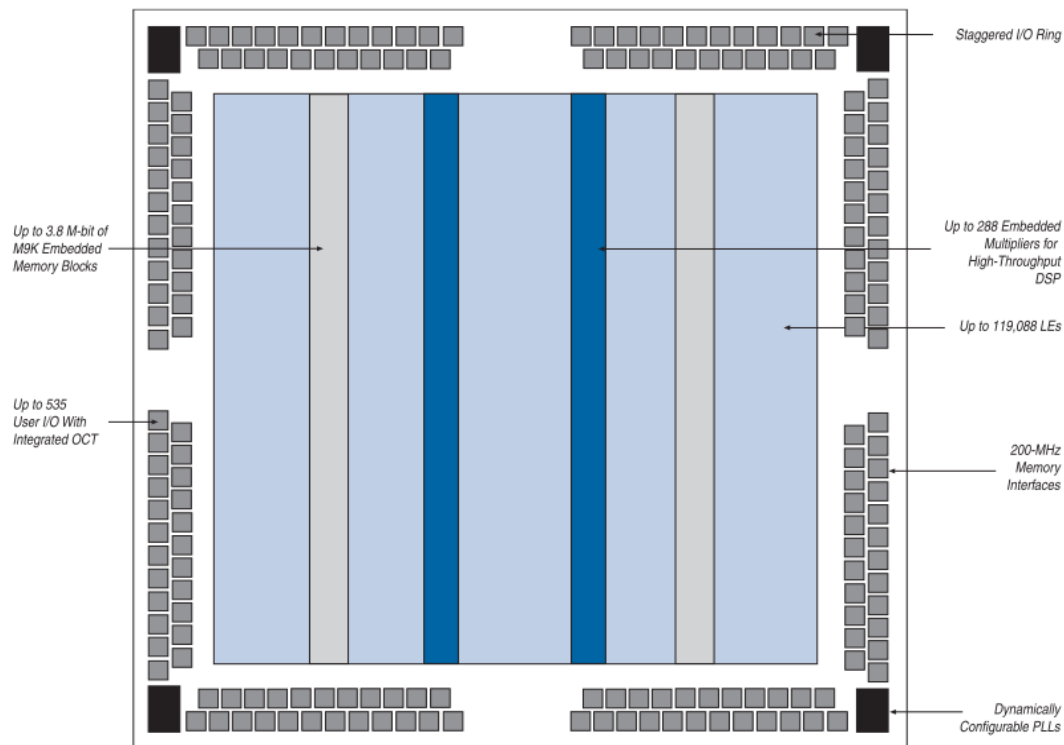
Der homogene Aufbau von CPLDs ermöglicht eine ziemlich genaue Bestimmung von Durchlaufzeiten. CPLDs speichern ihre Konfiguration im internen EEPROM-ähnlichen Speicher und müssen daher bei Inbetriebnahme nicht extern neu geladen werden. Durch die begrenzte Anzahl dieses Flash-Speichers können CPLDs jedoch nicht beliebig oft neu programmiert werden. Dies ist ein Unterschied von CPLDs zu FPGAs.

- **Field-Programmable Gate Arrays (FPGA)**

Ein FPGA besteht –ähnlich eines PLDs– aus einer Vielzahl untereinander vernetzter Blöcke. In der Regel befinden sich auch Speicherblöcke, PLLs oder Prozessorkerne in einem FPGA. Auch die Möglichkeiten diese Blöcke miteinander zu verbinden sind im Vergleich zu CPLDs viel umfangreicher.

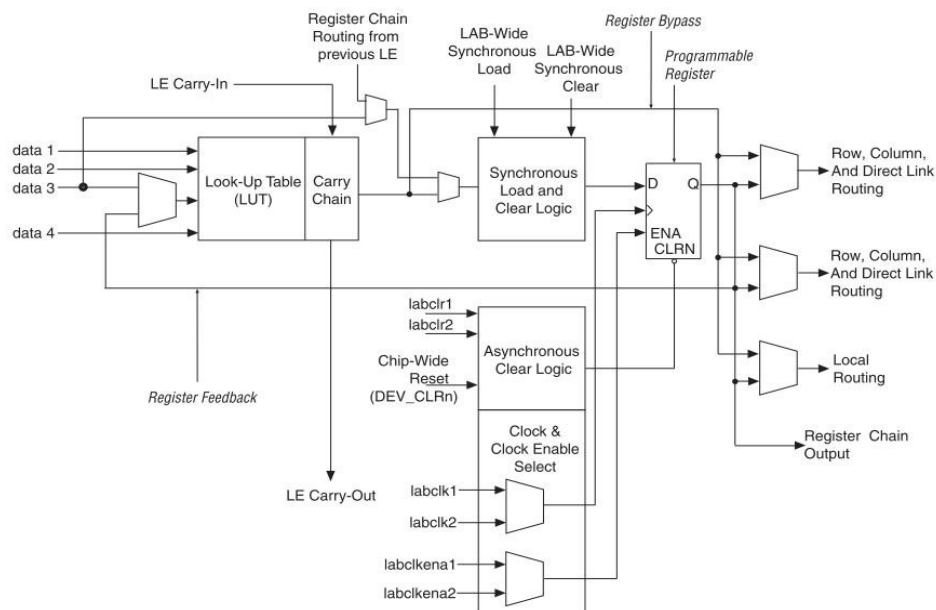
Für die DIC Übungseinheiten wird ein FPGA des Herstellers Intel (früher: Altera) verwendet – der EP3C16F484 aus der Cyclone III Serie. Auf diesem Chip vereint befinden sich

- ✓ insgesamt 15 408 LEs (Logic Elements; ein LE entspricht in etwa einer Makrozelle)
- ✓ 516 096 Bit RAM (organisiert als 56 9k-Blöcke)
- ✓ 56 Hardware-Multiplizierer
- ✓ 4 PLLs
- ✓ ... und insgesamt 346 I/O-Pins.



Struktur EPC3 FPGA (Quelle: Intel)

Kernelemente des Logic Elements / der Makrozelle sind die Look-Up Tabelle und das 1-Bit-Register.



EPC3: Aufbau eines LE (Logic Elements) (Quelle: Intel)

Mit Hilfe der SRAM basierten LUTs kann jede beliebige n-stellige Binärfunktion realisiert werden – vorausgesetzt, es sind entsprechend viele LUTs vorhanden. Die Programmierung der gewünschten Funktion, die ein LE ausführen sollte, erfolgt im Wesentlichen durch die Hinterlegung einer Wahrheitstabelle in der LUT, die hier 4 binäre Eingänge besitzt. LUT-Eingang 3 ist ein Multiplexer vorgeschaltet, um ggf. auf das Register (FF) – Ausgangssignal zuzugreifen, um Schaltwerke oder ähnliche Aufgaben zu realisieren. Eine Vielzahl an Multiplexer erlauben unterschiedliche Konfigurationen – das LUT Ausgangssignal kann mit / ohne Umgehung des Flip-Flops an weitere Blöcke weitergeleitet werden, Rückkopplungen von Ausgang auf Eingang, Ankopplung an sehr schnelle, lokale Signalbusse, ... sind somit möglich.

Da die LUTs des EPC3 SRAM-basiert sind, müssen sie bei jeder Inbetriebnahme neu konfiguriert werden – d.h. mit Ausschalten der Versorgungsspannung geht die hinterlegte Wahrheitstabelle verloren. Das FPGA kann daher typischerweise von einem seriellen oder parallelen externen Speicher seine Konfigurationsdaten lesen oder diese über JTAG oder ISP Anschlüsse von einem μ C oder angeschlossenem Programmer erhalten. Das muss aber zwangsläufig kein Nachteil sein – es gibt bereits Anwendungsfälle, wo FPGAs im laufenden Betrieb neu konfiguriert werden um dedizierte Aufgaben mit höherer Geschwindigkeit und gesteigerter Effizienz wahrnehmen zu können.

Während des Konfigurationsvorganges werden alle User-I/O-Leitungen in den hochohmigen Zustand geschaltet. Je nach Größe des Konfigurationsfiles und der Lesegeschwindigkeit beträgt die Konfigurationszeit eines Cyclone-III FPGAs zwischen 30ms und 150ms.

- **Unterschied CPLD \leftrightarrow FPGA**

Die Grenzen zwischen CPLD und FPGA verschwinden zunehmend. Vorteile eines PLD-Typs werden in der Regel bald in einem neueren FPGA Typ kompensiert (und umgekehrt). So besitzen CPLDs in der Regel einen geringeren Stromverbrauch als FPGAs – als Antwort auf diese Entwicklung haben viele FPGA-Bausteine die Möglichkeit erhalten, gezielt ganze Bereiche stromlos zu schalten, sobald diese nicht benötigt werden. Ebenfalls am Markt erhältlich sind SRAM-basierte CPLDs – d.h. auch diese CPLDs müssen bei Inbetriebnahme erst konfiguriert werden und benötigen somit eine definierte Power-Up Zeit bevor sie vollständig funktionsfähig sind.

Prinzipiell kann jedoch gesagt werden, dass CPLDs einen einfacheren Aufbau aufweisen als FPGAs und nur einen Bruchteil an Makrozellen/Logik-Elemente besitzen wie FPGAs.

Beispiel: CPLDs der MAX Serie von Intel gibt es mit 32 LE bis hin zu 2 210 LE, FPGAs der Cyclone III Serie beginnen bei 5 136 LE und sind bis 119 088 LE erhältlich.

- **FPGA – Hersteller**

Die größten Marktsegmente teilen sich Xilinx² und Intel.

Obwohl Lattice Semiconductor³ lange Zeit Marktführer bei GALs war, konnten sie bei FPGAs kaum Marktanteile gewinnen. Microsemi⁴ (früher Actel, jetzt Microchip) bietet FPGAs in Flash-Technologie an (d.h. nicht SRAM-basiert; diese FPGAs benötigen daher keine Ladephase).

Der Trend geht zu sog. SoC FPGAs, d.h. FPGA-Bausteinen mit einem oder mehreren (fest verdrahteten) Prozessorkernen.

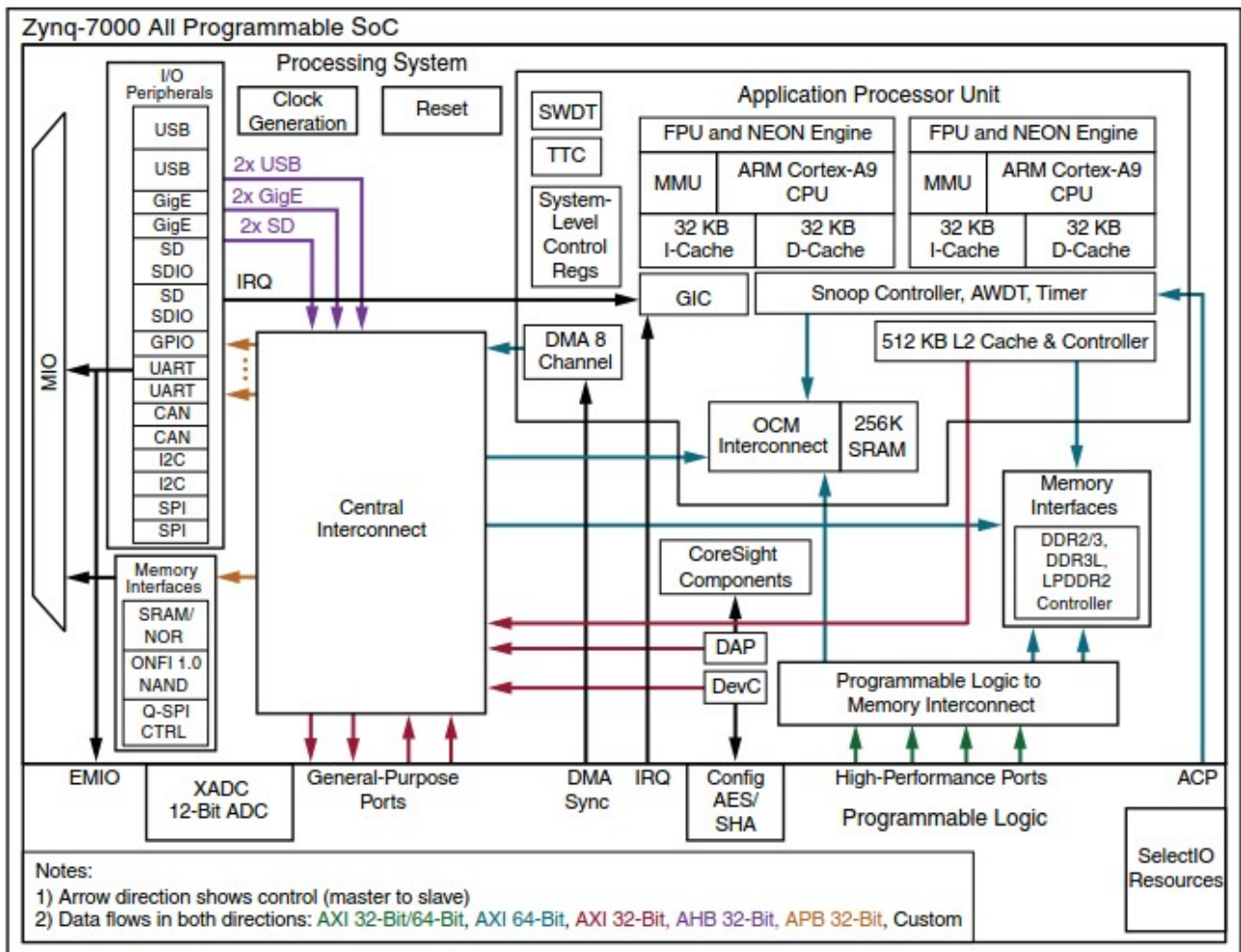
² <http://www.xilinx.com>

³ <http://www.latticesemi.com/>

⁴ <http://www.microsemi.com/>

Beispiele:

Xilinx: Zynq-7000 SoC/FPGA



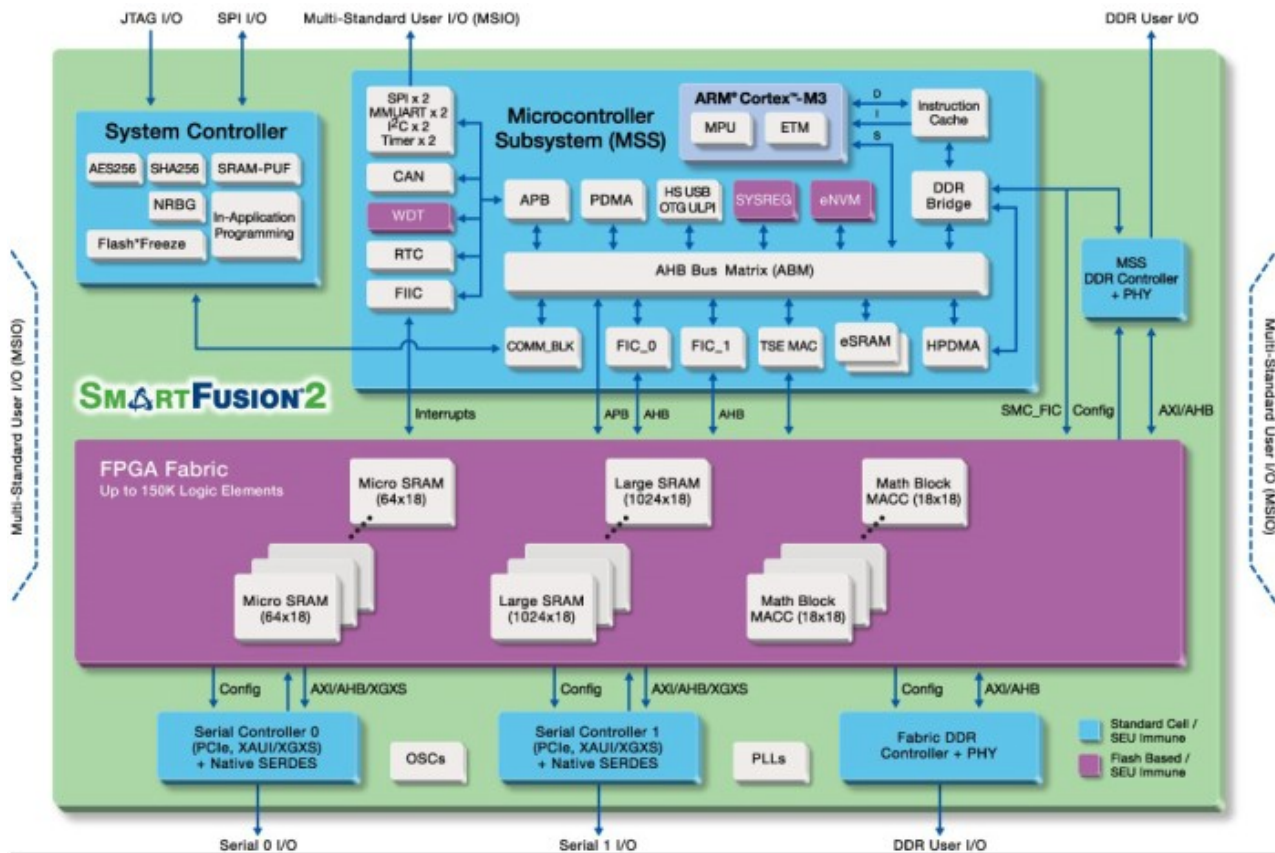
- bis zu 444K LE, 277 LUTs, 555 Flip Flops, 26,5 MB RAM, 2020 DSP Slices
- dual-core ARM Cortex A9 Prozessor; max. 1 GHz, DMA
- High-Speed Schnittstellen (2xUSB2.0 OTG, 2x Gigabit Ethernet, ...); CAN, LIN, ...
- Memory Unterstützung für DDR3, DDR3L, DDR2, LPDDR2, QSPI, NAND, NOR, ...

In dem open source Mess-System RedPitaya (<http://redpitaya.com/>) ist z.B. die kleinste ZYNQ Ausbaustufe (ZYNQ7010) zu finden.

Intel (früher: Altera): Cyclone V SoC

... gibt es in unterschiedlichen Ausbaustufen – nur mit Logik-Elementen als FPGA (bis zu 301K LE, 12,200 KB RAM, 684 18bitx18bit Multiplizierer, 8 PLLs, 342 DSP Blöcke), als FPGA mit bis zu 6,144 Gbps Transceivern, als FPGA mit single/dual ARM Cortex A9 Prozessor(en), als FPGA mit single/dual ARM Cortex A9 Prozessor & bis zu 6,144 Gbps Transceivern.

Microsemi (jetzt: Microchip): SmartFusion 2



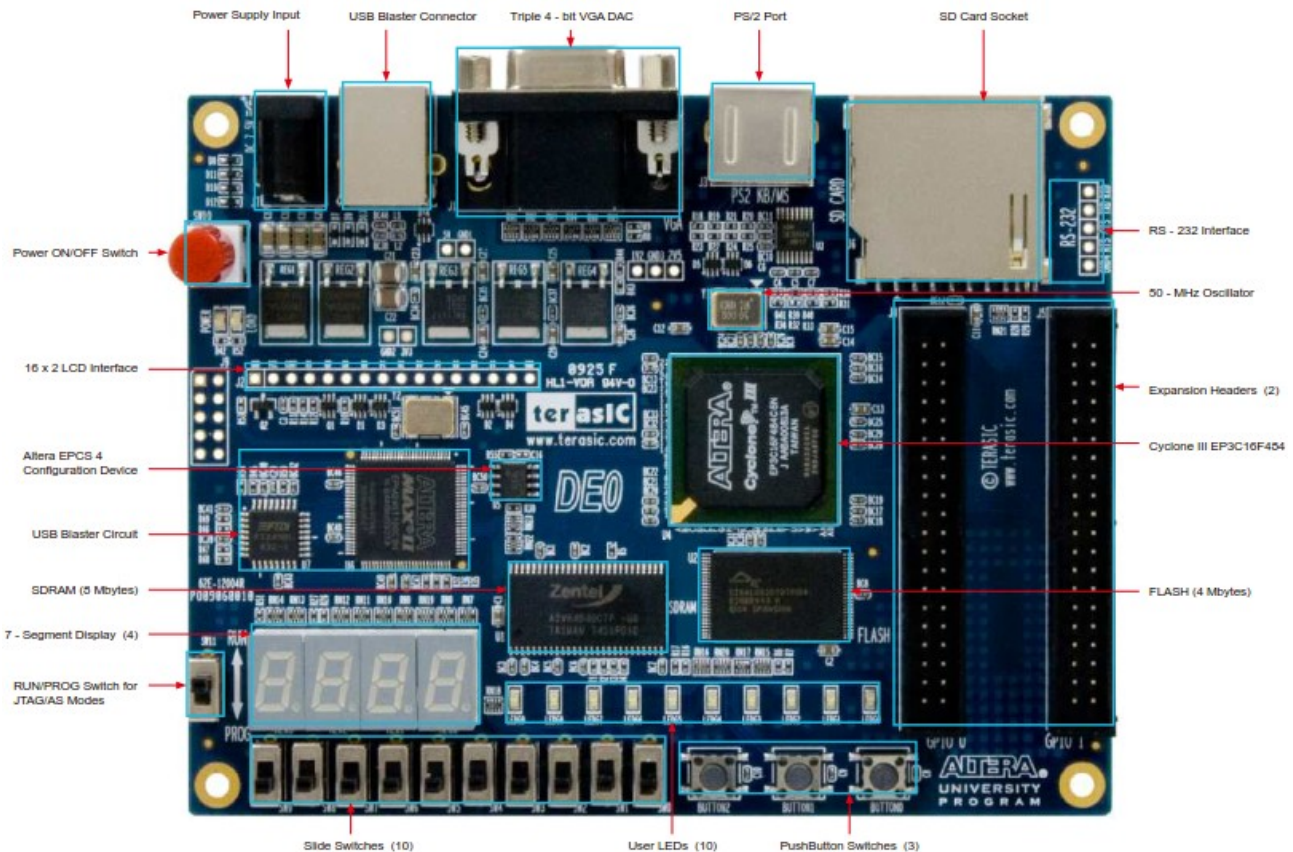
Bildquelle: SF2-Brochure_SinglePage.pdf

- bis zu 150K LE (Logic Elements)
- bis zu 240 schnelle Mathematik-Blöcke mit 18x18bit Multiplizierer und 44bit Akkumulator
- 166MHz, 32bit Cortex M3 mit 64kB embedded SRAM, bis zu 512kB embedded non-volatile memory (eNVM)
- High-Speed Schnittstellen (10Gbps Ethernet, PCI Express, USB2.0 OTG, ...)
- High-Speed Memory Schnittstellen (DDR2/DDR3/LPDDR Unterstützung, ...)

Der SmartFusion2 FPGA ist immun gegen SEU (Single Event Upset) Fehler; aufgrund interner Prüfbits kann er 2-Bit-Fehler intern erkennen, 1-Bit Fehler intern erkennen und beheben. Der SmartFusion2 Baustein erfüllt die höchsten Design Assurance Level (DAL) Anforderungen und kann daher z.B. in der Luftfahrt oder in der Medizintechnik eingesetzt werden. Der SmartFusion2 Baustein ist in einer speziellen S-Serie (mit zusätzlichen Security Features ausgestattet) erhältlich, und daher überall dort interessant, wo ein Reverse-Engineering / Hacking extrem erschwert werden soll (Militär-Anwendungen, ..., Schutz geistiges Eigentum in eigenen Produkten).

DE0-Board von Terasic

Wir arbeiten mit dem DE0 – Board von Terasic (www.terasic.com). Dieses Board wurde speziell für den Unterrichtseinsatz entwickelt und ist deshalb auch im University-Program von Intel gelistet.

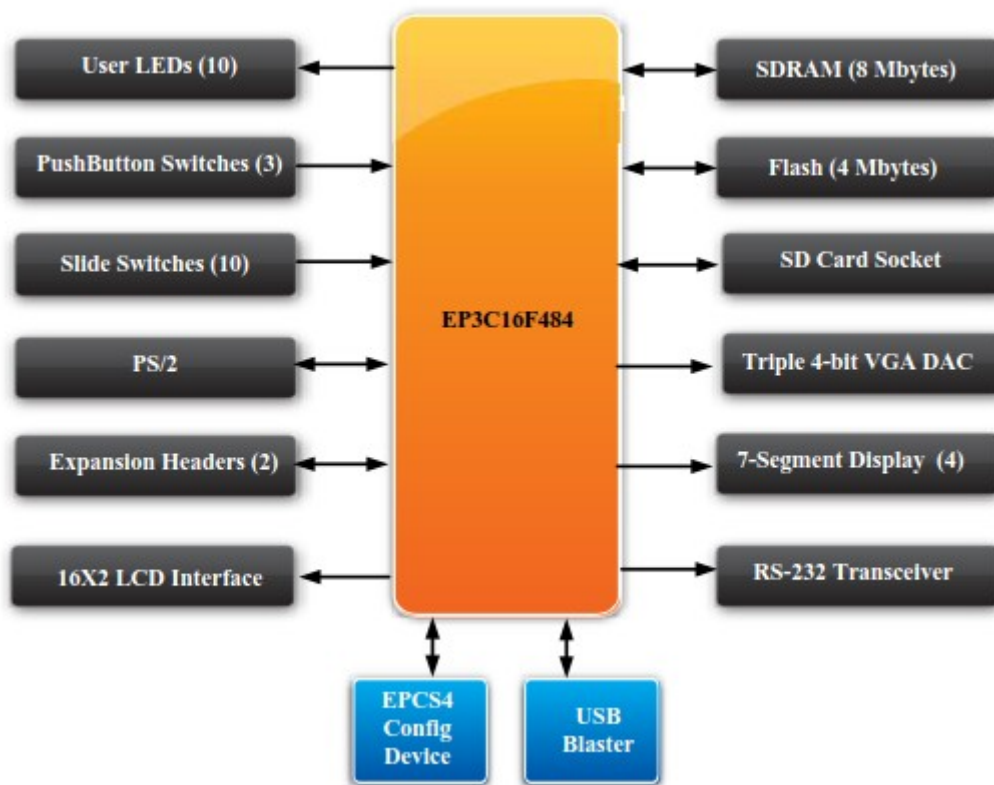


Bildquelle: Terasic

Neben dem Cyclone III FPGA (EP3C16F484) befindet sich 10 LEDs, 10 Schalter, 3 Taster, 4 7-Segment-Displays sowie eine USB-Blaster-Programmiereinheit auf dem Board, um problemlos die ersten FPGA-Versuche durchführen zu können. Für umfangreichere Projekte befindet sich auf diesem Board auch ein VGA-Anschluss, ein PS/2 Board (für den Anschluss einer PS/2 Tastatur und Maus), ein SD-Kartenschacht, eine RS-232 Schnittstelle, die Anschlussmöglichkeit eines LCD-Displays, 8MByte SDRAM und 4MByte Flash (jeweils 16bit Daten). Zwei 40pol. - Erweiterungsstecker erlauben den Anschluss eigener Hardware. Als primäre Taktquelle dient ein 50MHz Oszillator.

Auf dem DE0-Board ist ein serielles Konfigurations-EEPROM vorhanden (EPCS4). Der FPGA Baustein liest während des Power-UP Zyklus seine Konfiguration aus diesem Speicher. Mit dem Schalter RUN/PROG wird ausgewählt, ob der FPGA mit der Konfiguration direkt beschrieben werden (Schalterstellung: RUN) sollte oder das serielle EEPROM (Schalterstellung: PROG). **Für alle unsere Versuche muss der Schalter auf RUN stehen.**

D.h. bei jedem Aus- und wieder Einschalten startet die Standard-Applikation (Ausgabe von Mustern an den LEDs, den 7-Segment-Anzeigen, ..., VGA-BildschirmAusgabe). Damit ist für uns die korrekte Funktion ersichtlich.



Bildquelle: Terasic

Die Spannungsversorgung erfolgt entweder über USB oder durch das mitgelieferte 7.5V Netzteil. Vor allem beim Anschluss externer Peripherie ist es ratsam, das externe Steckernetzteil zu verwenden.

Aufgrund des leistungsfähigen FPGAs mit 15 408 LE und der Zusatzperipherie können Softcore-Projekte umgesetzt werden.

2 Die Hardwarebeschreibungssprache VHDL

Neben der graphischen Entwurfsmöglichkeit kann die Verhaltensbeschreibung eines (digitalen) Systems auch mittels HDL (Hardware Description Language) vorgenommen werden. Unter einer Verhaltensbeschreibung versteht man die funktionale Modellierung des Ein-/Ausgabeverhaltens. Die aktuell populärsten Beschreibungssprachen sind VHDL (Europa) und Verilog (USA, Asien). Herstellerspezifische Sprachen wie Altera-HDL (AHDL), ABEL, Log/iC oder UDL/I finden kaum mehr noch Einsatz.

Relativ neu sind Ansätze, FPGAs und CPLDs in der Programmiersprache C zu beschreiben. Dabei gilt es, einige „Herausforderungen“ zu meistern – vor allem die Umsetzung der in der Regel sequentiellen Programmierweise in die parallele Struktur einer Hardware. Dabei wird versucht, auf den weit verbreiteten Standardsprachen ANSI C und C++ aufzusetzen und diese mit Sprachkonstrukten zu erweitern (HardwareC, HandelC, BachC). Als Oberbegriff hat sich C2HW (sprich: C-to-Hardware) etabliert.

Aus den (Hardwarebeschreibungs-) Sprachen erzeugen Synthese-Werkzeuge bausteinspezifische Netzlisten, aus denen dann die FPGA-Konfigurationsdatei generiert wird. Viele Entwurfswerkzeuge erlauben auch die graphische Eingabe in Form eines Schaltplans. Dabei erfolgt in der Regel eine interne Umsetzung der graphischen Eingabe in eine Hardwarebeschreibungssprache wie Verilog oder VHDL (→ siehe Altium Designer).

Die Entwurfswerkzeuge erlauben meist auch eine Simulation / Verifikation. Dabei ist zwischen der funktionellen Simulation (ohne Berücksichtigung von (Gatter-)Laufzeiten) und der sog. Timing-Simulation zu unterscheiden. Um eine Timing-Simulation durchführen zu können, muss der Ziel - CPLD/FPGA Typ bekannt und ausgewählt sein und meist ist diese Simulation nur innerhalb der jeweiligen herstellerspezifischen Design-Software möglich.

VHDL wurde zu Beginn der 80er Jahre im Auftrag der US-Regierung von IBM, Texas Instruments und Intermetrics entwickelt und 1987 standardisiert (IEEE⁵ 1076; landläufige Bezeichnung VHDL87). VHDL ist die Abkürzung von VHSIC Hardware Description Language, wobei VHSIC wiederum für Very High Speed Integrated Circuit steht. 1993 erfolgte eine Überarbeitung und Erweiterung des IEEE Standards (VHDL 93). Obwohl es später weitere Überarbeitungen gab, ist VHDL93 derzeit weit verbreitet⁶. Gemäß IEEE-Richtlinien muss ein Standard mindestens alle 5 Jahre überarbeitet werden, um nicht zu verfallen. So wurde -zum Beispiel- in einer solchen Überarbeitung- der XNOR-Operator⁷ eingeführt. Mittlerweile ist VHDL auch als ANSI⁸-Standard definiert.

HDLs unterstützen den sog. Top-Down-Entwurf. Dabei kann eine Aufgabenstellung auf der Verhaltensebene modelliert und simuliert werden. Durch schrittweises Ersetzen der „Verhaltens-Blöcke“ durch eine Struktur kann die Aufgabenstellung in eine synthetisierbare Form gebracht werden. Jeder dieser Schritte kann wiederum durch Simulation überprüft werden. Eine Entwurfsaufgabe kann sehr einfach auf mehrere Personen verteilt werden. Auch die Wiederverwendung von bestehenden Entwürfen (oder Teilen davon) ist möglich.

Nicht alle VHDL Sprachanweisungen sind in allen Stadien des Entwurfs anwendbar. So kann zum Beispiel die Anweisung „*after 100ms*“ nicht synthetisiert werden.

5 IEEE: Institute of Electrical and Electronics Engineering

6 In Quartus kann zwischen VHDL 1987, VHDL 1993 und VHDL 2008 gewählt werden; default ist VHDL 1993

7 Es werden nicht nur neue Operatoren eingeführt, sondern auch die Sprachkonstrukte weiterentwickelt. Zum Beispiel vor 2008: *if (A='1' and B='1') then ...* mit 2008: *if (A and B) then ...*

8 ANSI: American National Standards Institute

2.1 Grundlegende VHDL-Konzepte anhand einfacher Beispiele

Der Aufbau einer VHDL Beschreibung soll anhand von Beispielen aus dem Themengebiet der kombinatorischen Logik schrittweise erklärt werden (*zur Erinnerung: Eine kombinatorische Logik besteht aus reinen Schaltnetzen, die wiederum nur aus Zusammenschaltungen von Gattern bestehen. Speicherelemente und Automaten sind nicht Teil einer kombinatorischen Logik*).

→ **Beispiel:** Wechselschalter

Eine LED soll über 2 unabhängige Schalter (SW0, SW1) ein- bzw. ausgeschaltet werden können. Die LED ist dabei aktiv-High, d.h. bei einem log. 1 am Ausgang leuchtet die LED, bei log. 0 hingegen nicht.

Wahrheitstabelle:

SW1	SW0	LED
0	0	
0	1	
1	0	
1	1	

Bool'sche Gleichung: LED =

→ es handelt sich um eine Antivalenz-Verknüpfung der beiden Eingänge, auch als Exklusiv-ODER oder XOR bezeichnet.

Schaltung in der Digitaltechnik:

Umsetzung in VHDL:

```
entity Wechselschalter is
  port ( SW1, SW0: in bit;
         LED:      out bit
       );
end Wechselschalter;

architecture Architektur of Wechselschalter is
begin
  LED <= SW1 xor SW0;
end Architektur;
```

Mit dem VHDL Schlüsselwort **ENTITY** werden die Schnittstellen des Entwurfsblockes nach außen beschrieben. Die Deklaration der Anschlüsse wird mit der **PORT**-Anweisung durchgeführt. Ein

PORT-Signal beschreibt somit die Kommunikation einer **ENTITY** nach außen.

Unser Wechselschalter-Beispiel hat zwei Eingangssignale (SW1, SW0) und ein Ausgangssignal (LED).

Die **ARCHITECTURE** beschreibt das Innenleben, d.h. die eigentliche Funktionalität der Entwurfseinheit. Jede **ENTITY** muss mindestens eine **ARCHITECTURE** aufweisen.

VHDL Anweisungen müssen mit einem **;** abgeschlossen werden. Groß-/Kleinschreibung wird ignoriert, ebenfalls die Anzahl an Leerzeichen- und Zeilenumbrüchen.

Bezeichnungen (engl. identifier) müssen mit einem Buchstaben⁹ beginnen; danach können auch Ziffern oder ein Unterstrich folgen (natürlich auch weitere Buchstaben).

Jedes Signal muss von einem eindeutig zu identifizierenden Typ sein. Ursprünglich wurde nur der Datentyp **BIT** in der VHDL Sprache spezifiziert. Mehrere **BIT**-Signale können zu einem Bus zusammengefasst werden, dann ist der Datentyp **BIT_VECTOR**¹⁰ zu verwenden. Der Wertevorrat dieses Datentyps besteht nur aus den log. Werten '0' und '1'.

Einem **BIT_VECTOR** muss die Busbreite mitgegeben werden. Dies kann entweder

aufsteigend	BIT_VECTOR (0 TO 7)	
oder absteigend	BIT_VECTOR (7 DOWNTO 0)	erfolgen.

Vor allem die absteigende Angabe ist sehr üblich, da man in der Regel das niederwertigste Bit an der ganz rechten Stelle hat (d₇ d₆ d₅ d₄ d₃ d₂ d₁ d₀).

Signale werden in VHDL durch den Zuweisungsoperator **<=** zugewiesen.

Zuweisung und Abfrage von Signalwerten vom Typ **BIT** müssen zwischen Apostrophen gesetzt werden – zum Beispiel:

```
Y <= '0';
```

... bei **BIT_VECTOR** in Anführungszeichen

```
Y_BUS <= "10110010";
```

... oder alternativ

```
Y_BUS <= ( '1' , '0' , '1' , '1' , '0' , '0' , '1' , '0');
```

Wichtig: Wenn mehrere Zuweisungsoperatoren untereinander stehen (aufeinander folgen), so kann man daraus nicht immer auf eine sequentielle Ausführung schließen. Stattdessen erfolgt eine nebenläufige (= parallele) Ausführung!

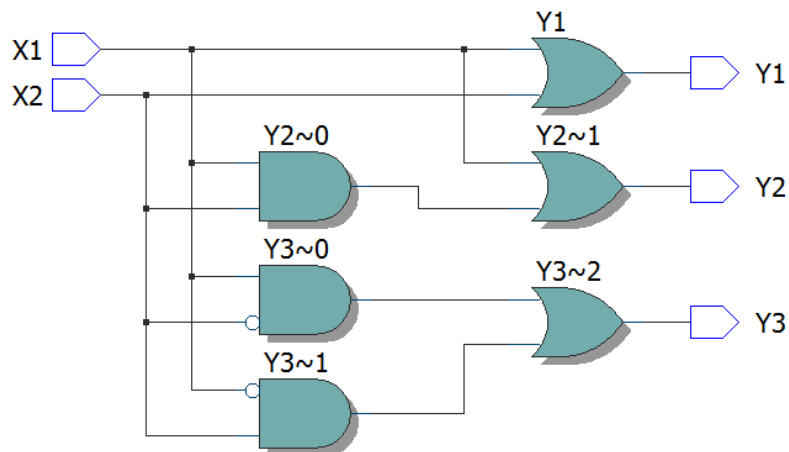
```
entity BspParalleleUmsetzung is
  port ( X1, X2:      in bit;
         Y1, Y2, Y3: out bit
  );
end BspParalleleUmsetzung;
```

```
architecture Architektur of BspParalleleUmsetzung is
begin
  Y1 <= X1 or X2;
  Y2 <= (X1 and X2) or X1;           -- Klammern werden benoetigt!
  Y3 <= (X1 and not X2) or (not X1 and X2); -- entspricht xor !
end Architektur;
```

⁹ Deshalb gibt es auch Umsetzungsfehler, wenn eine Bezeichnung wie z.B. 7SegmentDecoder verwendet wird!

¹⁰ bit / bit_vector sollte nicht mehr verwendet werden, da man hier von einer „heilen“ Digitaltechnik-Welt aus nur 0 und 1 ausgeht → wir machen dies nur in den ersten Anfangsbeispielen...

Umsetzung:



Signale haben eine Richtung, diese muss bei der Deklaration festgelegt werden.

IN	Eingangssignal. Dieses kann nicht modifiziert werden, darf also nur auf der „rechten“ Seite stehen oder bei einer Signalabfrage verwendet werden.
OUT	Ausgangssignal. Darf nur auf der linken Seite einer Signalzuweisung stehen ¹¹ .
BUFFER	Ausgangssignal, das auch als Eingang oder zur Signalabfrage verwendet werden kann
INOUT	bidirektionales Signal beim Datentyp <code>std_logic</code>

Kommentare können an beliebiger Stelle stehen und beginnen mit `--`.

Boolsche Operatoren

VHDL unterstützt die folgenden Booleschen Operatoren:

NOT, AND, NAND, OR, NOR, XOR, XNOR

NOT ist einstellig (d.h. ein Eingang, ein Ausgang), alle anderen Operatoren sind zweistellig (d.h. zwei Eingänge, ein Ausgang). Wenn daher ein Gatter mit 3 oder mehreren Eingängen benötigt wird, so ist dies durch zweistellige Operatoren zu realisieren.

Dabei ist aber unbedingt auf die Klammersetzung aufzupassen !

Assoziative Operatoren (**AND, OR, XOR, XNOR**) bedürfen keiner Klammerung, wenn mehr als zwei Operatoren selben Typs verknüpft werden – bei **NAND, NOR** ist dies aber unbedingt notwendig !

Beispiele:

```
Y <= A and B and C;           -- o.k., 3 assoziative Operatoren selben Typs
Y <= not (A and B and C);     -- NAND-Gatter mit 3 Eingängen
Y <= A nand B nand C;         -- falsch!
Y <= (A nand B) nand C;       -- syntaktisch richtig; aber KEIN 3fach NAND!
```

Boolsche Operatoren können auch auf Busse/Vektoren (**BIT_VECTOR**) angewendet werden; dabei

¹¹ Wurde in VHDL 2008 geändert

wirken sie auf alle Bitstellen parallel. Natürlich müssen die Längen der Operanden und des Ergebnis-Vektors übereinstimmen.

Beispiel:

```
...
begin
  A <= "10101010";
  B <= "11110000";
  Y_1 <= A and B;
  Y_2 <= not A or B;
end Architektur;
```

Der Datentyp std_ulogic

Der Datentyp `std_ulogic` (und der Sub-Typ `std_logic`) ist eine Erweiterung von **BIT**. Dies ist notwendig, um die realen Anforderungen eines HW-Entwurfs zu genügen. Oft sind die Zustände '0' oder '1' nicht ausreichend, Signale können weitere Zustände annehmen – zum Beispiel 'Z' (hochohmig). Nach dem Einschalten kann erstmals ein Signal undefiniert sein, bis ein bestimmtes Ereignis eintritt (zum Beispiel ein Reset). Auch ein Logiksimulator, der einen digitalen Schaltungsentwurf simulieren soll, muss mit mehreren Zuständen arbeiten.

Deshalb wurde der Datentyp `std_ulogic` spezifiziert, der neunwertig ist - Auszug aus dem `std_logic_1164` Package:

```
-----
-- logic state system (unresolved)
-----
TYPE std_ulogic IS ( 'U', -- Uninitialized
                    'X', -- Forcing Unknown
                    '0', -- Forcing 0
                    '1', -- Forcing 1
                    'Z', -- High Impedance
                    'W', -- Weak Unknown
                    'L', -- Weak 0
                    'H', -- Weak 1
                    '-' -- Don't care
                  );
```

...

```
-----
-- *** industry standard logic type ***
-----
```

```
SUBTYPE std_logic IS resolved std_ulogic;
```

Wert	Bedeutung	Verwendung
U	Unknown	Nicht initialisiertes Signal
X	Konflikt	Wenn mehr als ein Signaltreiber aktiv ist
0	Starke log. 0	Entspricht einem '0' eines bit-Signals
1	Starke log. 1	Entspricht einem '1' eines bit-Signals
Z	Hochohmig	Tri-State-Ausgang aktiv

L	Schwache log. 0	Ausgang mit aktivem Pull-Down-Widerstand
H	Schwache log. 1	Ausgang mit aktivem Pull-Up-Widerstand
W	Schwach unbekannt	Konflikt zwischen 'L' und 'H'
-	Don't care	Zustand bedeutungslos, kann für Minimierung verwendet werden

Da `std_ulogic` nicht Teil der VHDL Sprache ist, sondern in Form einer Bibliothek vorliegt, muss diese Bibliothek vor Verwendung eingebunden werden.

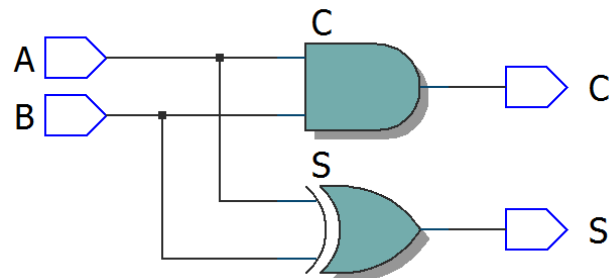
```
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;
```

→ **Beispiel:** Halbaddierer unter Verwendung des Datentyp `std_ulogic`

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

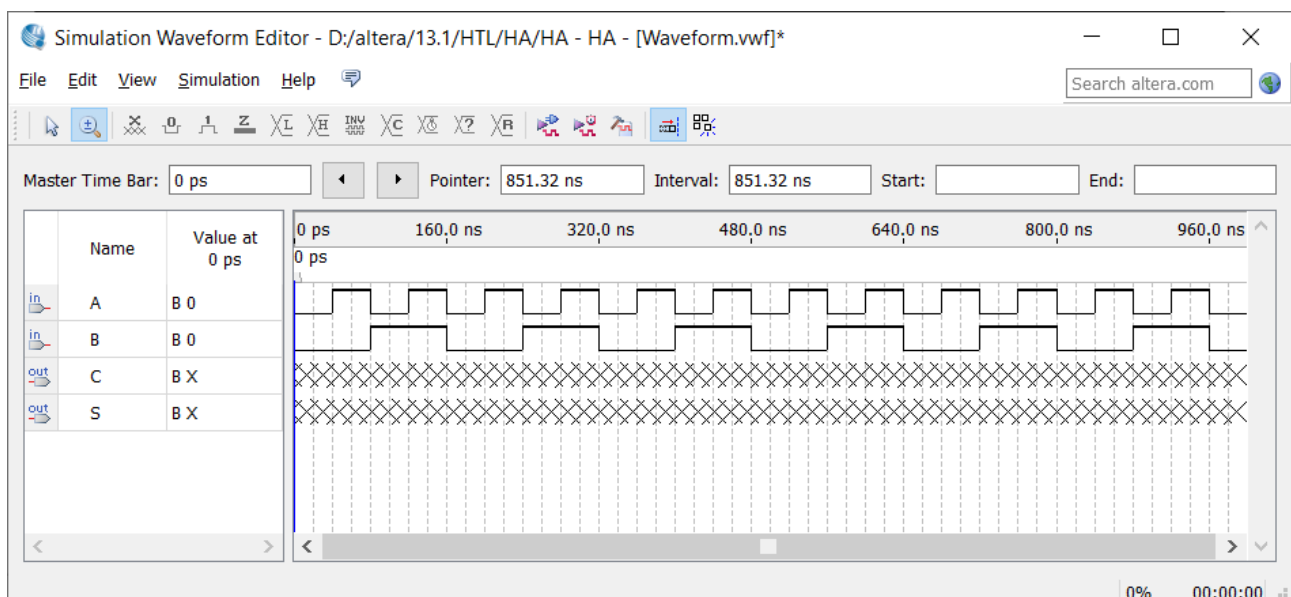
entity HA is
    port ( A, B:    in std_ulogic;
          S, C:    out std_ulogic
    );
end HA;

architecture Architektur of HA is
begin
    S <= A xor B;           -- Summe
    C <= A and B;           -- Uebertrag
end Architektur;
```




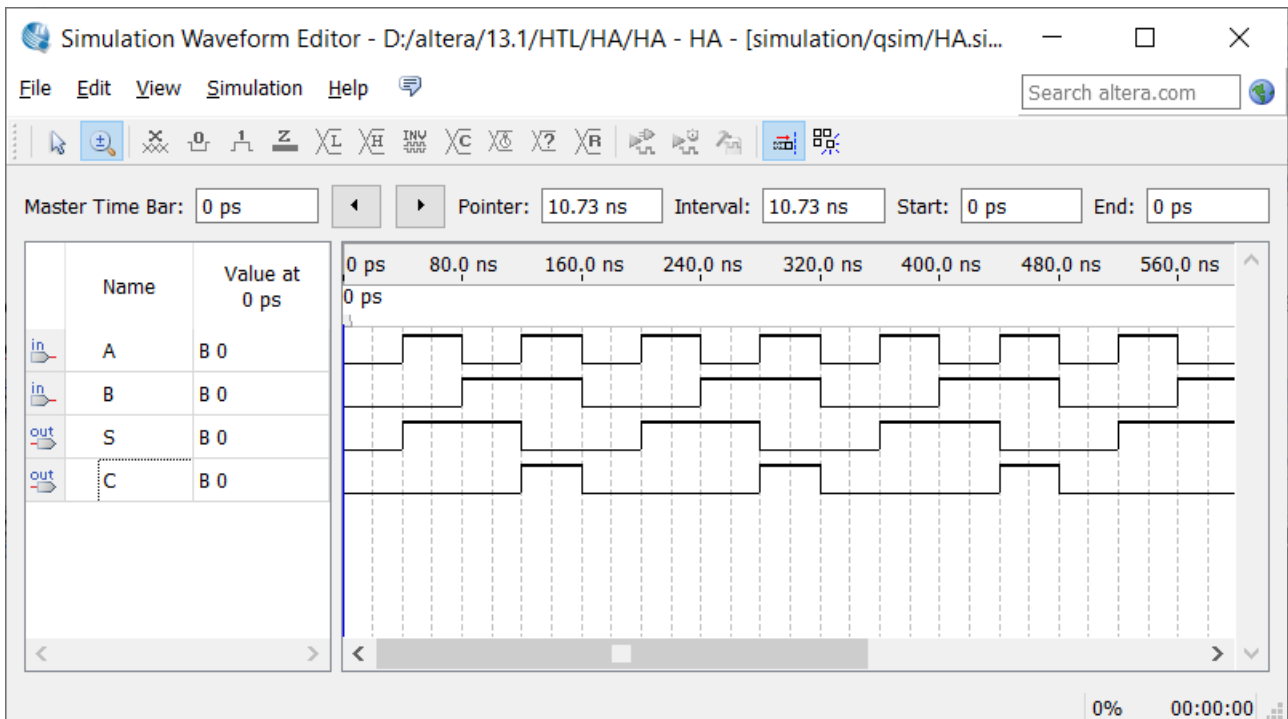
Das Ergebnis wird mittels Simulation überprüft. Zur Erstellung gibt es zwei Wege... (halb-)automatisch (ähnlich wie bei Altium) oder graphisch.

Wir werden zu Beginn das Testfile graphisch erstellen: File → New → Verification/Debugging Files → University Program VWF. Ein Grafik-Editor wird geöffnet; im ersten Schritt fügen wir die Ein- und Ausgangssignale hinzu und legen dann im zweiten Schritt das Eingangsbitmuster fest:



Mit File → Save wird der Entwurf gespeichert, wobei der **vorgeschlagene Name** (→ hier Waveform.vwf) **nicht verändert** wird.

Mit Klick auf  wird die RTL Simulation (= funktionale Simulation) gestartet...



→ S ist log. 1 wenn A oder B (aber nicht gleichzeitig!) log. 1 ist, C ist log. 1 wenn A und B log. 1 sind... somit wurde die korrekte Funktion des Halbaddierers per Simulation überprüft!

→ **Übung:** Beschreibe eine Schaltung, die zwei Eingangsbit a und b auf Gleichheit prüft und verifiziere das Ergebnis mittels Simulation.

→ **Übung:** Beschreibe einen 1-aus-4 Codeumsetzer mit Enable-Eingang











(verkürzte) Funktionstabelle:

Eingänge			Ausgänge
en	a(1)	a(0)	X(3..0)
0	-	-	0 0 0 0
1	0	0	0 0 0 1
1	0	1	0 0 1 0
1	1	0	0 1 0 0
1	1	1	1 0 0 0

Um das Design auf dem DE0-Board testen zu können, muss die Zuweisung zu den externen Anschlüssen („Pinbelegung“) vorgenommen werden. Die Ausgänge X(3..0) werden mit den grünen

LEDs (3..0) verbunden, die Eingänge a(1..0) mit den Schiebeschaltern SW(1..0) und Enable mit dem Schiebeschalter SW9:

Assignments → Pin Planner:

X Named: *		Edit:   		Filter: Pins: all		
Node Name	Direction	Location	I/O Bank	VREF Group	I/O Standard	Reserved
 a[1]	Input	PIN H5	1	B1 N0	3.3-V LVTTTL	
 a[0]	Input	PIN J6	1	B1 N0	3.3-V LVTTTL	
 en	Input	PIN D2	1	B1 N0	3.3-V LVTTTL	
 x[3]	Output	PIN H1	1	B1 N1	3.3-V LVTTTL	
 x[2]	Output	PIN J3	1	B1 N1	3.3-V LVTTTL	
 x[1]	Output	PIN J2	1	B1 N1	3.3-V LVTTTL	
 x[0]	Output	PIN J1	1	B1 N1	3.3-V LVTTTL	
<<new node>>						

BEVOR wir jedoch die Kompilierung durchführen, ändern wir noch das Default-Verhalten nicht verwendeter IO-Pins (da in Summe der EP3C16F484C6 immerhin über 347 User-I/Os verfügt):

Assignments → Device → Device and Pin Options → Unused Pins:

Category:

- General
- Configuration
- Programming Files
- Unused Pins
- Dual-Purpose Pins
- Capacitive Loading
- Board Trace Model
- I/O Timing
- Voltage

Unused Pins

Specify device-wide options for reserving all unused pins on the device. To reserve individual dual-purpose configuration pins, go to the Dual-Purpose Pins tab. To reserve other pins individually, use the Assignment Editor.

Reserve all unused pins: As input tri-stated

Signale:

Mit der **PORT**-Deklaration werden die Anschlüsse einer **ENTITY** festgelegt. Oft ist es aber notwendig, mit internen Signalen, die nur innerhalb des **ARCHITECTURE** – Blockes benötigt werden, zu arbeiten. Dazu dient das VHDL Schlüsselwort **SIGNAL**.

→ *Beispiel:*

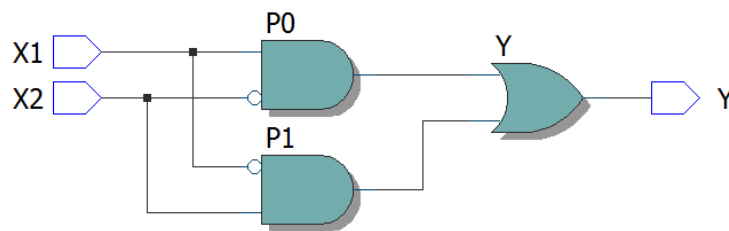
```

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity Wechselschalter is
    port ( X1, X2: in std_ulogic;      -- hier wird nun X und Y verwendet...
           Y:      out std_ulogic); --
end Wechselschalter;

architecture Architektur of Wechselschalter is
    signal P0, P1: std_ulogic;
begin
    -- aufgrund der Parallelitaet spielt die Reihenfolge keine Rolle!
    -- daher koennen wir erstmals die Summe der Produktterme Y zuweisen
    Y <= P0 or P1;
    -- Produktterme:
    P0 <= X1 and not X2;
    P1 <= not X1 and X2;
end Architektur;

```



Wenn man sich diese Darstellung ansieht, dann stellt man fest, dass P0 und P1 gleichzeitig oder zumindest zeitlich unabhängig voneinander erzeugt werden. Genau dies ist unter „nebenläufiger Zuweisung“ zu verstehen und hier unterscheiden sich HDLs von den meisten anderen Programmiersprachen (mit sequentieller Abarbeitung!).

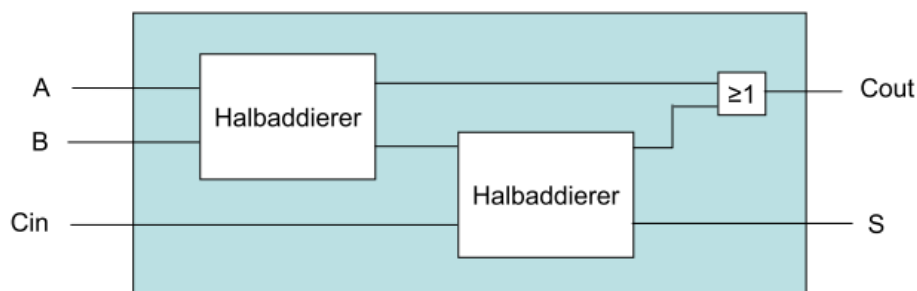
2.2 Hierarchie

Es ist immer ratsam, eine Aufgabenstellung in kleiner Untereinheiten zu zerlegen, die jeweils für sich ausgearbeitet und getestet werden können. Somit erreicht man auch, dass man einige Blöcke auch in anderen Projekten wiederverwenden kann.

Bislang bestanden unsere VHDL Projekte nur aus einer textuell beschriebenen „Top Level“ – Anweisung. VHDL ist eine textbasierte Beschreibungssprache und daher gibt es auch Schlüsselwörter, die es ermöglichen, hierarchische Strukturen in Textform zu beschreiben.

VHDL hat sich über die Jahre hinweg weiterentwickelt – da viele (automatisierte) Tools noch Code gemäß VHDL 1987 generieren (wie z.B. der Altium Designer; und diese Syntax z.B. in höheren VHDL Versionen nach wie vor gültig ist), schauen wir uns die Versionen gem. VHDL 87 und 93 anhand eines Volladdierers an:

→ **Beispiel:** Aus zwei Halbaddierern (HA) und einem ODER-Gatter ist ein Volladdierer (VA) aufzubauen. Die hierarchische Struktur eines VA sieht folgendermaßen aus:



Der **ENTITY**-Block unseres VA ist wie gewohnt mit den Ein-/Ausgangsdefinitionen aufgebaut:

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

entity VA is
    port ( A, B, Cin: in std_ulogic;
           S, Cout: out std_ulogic);
end VA;
```

Die Hierarchiebeschreibung erfolgt in **ARCHITECTURE**... und nur dort. Das macht auch Sinn. VHDL

erlaubt mehrere unterschiedliche Architektur-Beschreibungen, die inhaltlich unterschiedlich codiert sein können, nach außen sich aber (meist) ident verhalten sollen. Nicht jede Architekturbeschreibung muss hierarchisch aufgebaut sein, daher wäre eine Komponentendeklaration der Logikblöcke in der **ENTITY** am falschen Platz.

Folgend nun die **ARCHITECTURE**-Beschreibung nach VHDL 87:

```
architecture Struktur_nach_VHDL87 of VA is
```

```
component HA
```

```
  port ( A, B:      in std_logic;  
         S, C:      out std_logic  
       );
```

```
end component;
```

```
signal S_HA1, C_HA1:  std_logic;
```

```
signal S_HA2, C_HA2:  std_logic;
```

```
begin
```

```
  -- Instanziere die beiden HA
```

```
  HA1: HA port map (  
    A => A,  
    B => B,  
    S => S_HA1,  
    C => C_HA1  
  );
```

```
  HA2: HA port map (  
    A => S_HA1,  
    B => Cin,  
    S => S_HA2,  
    C => C_HA2  
  );
```

```
  -- Verknuepfung
```

```
  Cout <= C_HA1 or C_HA2;  
  S <= S_HA2;
```

```
end Struktur_nach_VHDL87;
```

Das Signal S_HA2 hätte man nicht benötigt, man hätte den S-Ausgang des 2. HA-Blockes direkt mit S verknüpfen können. Da der Compiler dies in der Umsetzung durchführen wird, hat dies keinerlei Nachteil.

Die Komponentendeklaration erfolgt mit dem Schlüsselwort **COMPONENT**. Damit wird ein Verweis auf eine Komponente hergestellt, die der IDE bereits bekannt sein muss. Danach folgt die Schnittstellenbeschreibung der Komponente, wobei die Signale in derselben Form und in derselben Reihenfolge auftreten müssen, wie dies bei der **ENTITY**-Deklaration der Komponente erfolgte.

Bei der nachfolgenden Instanzierung der Komponente wird dann durch **PORT MAP** eine Zuordnung der Signale mit den Komponenten-Anschlüssen vorgenommen. Dabei gilt immer die Reihenfolge:

```
Name_Komponenten_Anschluss <= neuer_Signal_Name ;
```

Um den VHDL Code so übersichtlich und leicht nachvollziehbar wie möglich zu halten, werden sowohl bei der Komponenten-Beschreibung als auch bei der Instanzierung die selben Komponenten-Anschluss-Namen verwendet (die Reihenfolge muss die selbe bleiben).

Und hier die **ARCHITECTURE**-Beschreibung nach VHDL 93:

```
architecture Struktur_nach_VHDL93 of VA is
```

```
signal S_HA1, C_HA1: std_ulogic;
```

```
signal S_HA2, C_HA2: std_ulogic;
```

```
begin
```

```
-- Instanziere die beiden HA
```

```
HA1: entity work.HA port map (
```

```
    A => A,
```

```
    B => B,
```

```
    S => S_HA1,
```

```
    C => C_HA1
```

```
);
```

```
HA2: entity work.HA port map (
```

```
    A => S_HA1,
```

```
    B => Cin,
```

```
    S => S_HA2,
```

```
    C => C_HA2
```

```
);
```

```
-- Verknuepfung
```

```
Cout <= C_HA1 or C_HA2;
```

```
S <= S_HA2;
```

```
end Struktur_nach_VHDL93;
```

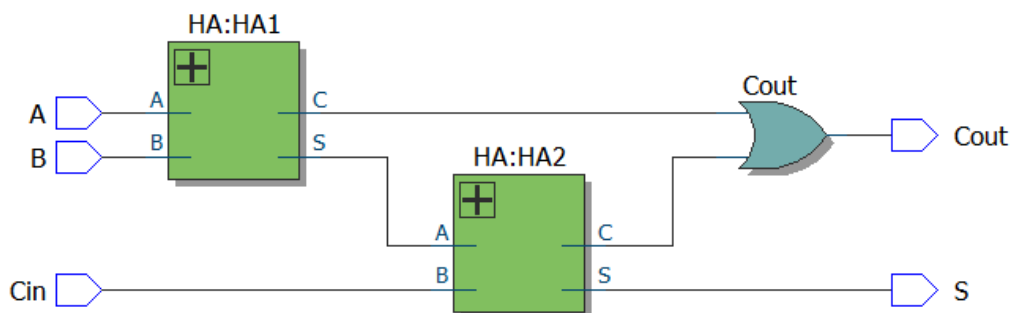
Hier wird jeder Sub-Hierarchieblock als Bibliothek aufgefasst. Befindet sich das entsprechende VHDL File im selben Arbeitsverzeichnis, so ist der Bibliotheksname **work** zu verwenden.

Aufbau:

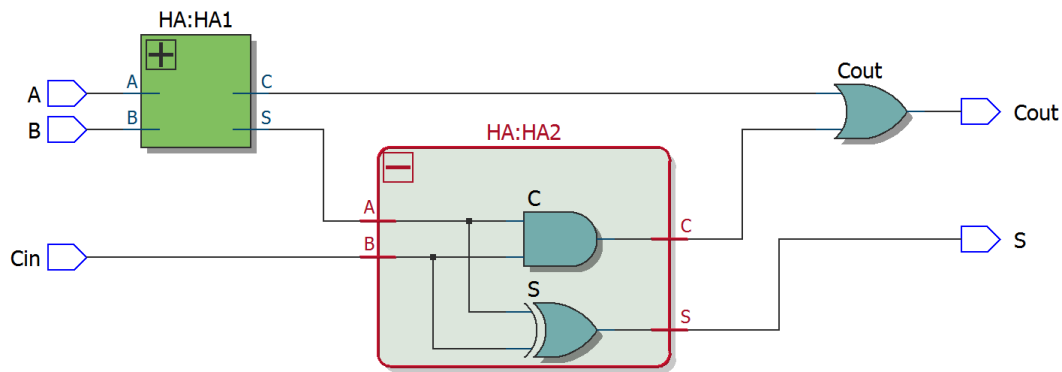
```
Unit_label: entity lib.name.entity_name(architecture_name) port map (...);
```

Die Angabe des Architecture-Namen ist optional. Wenn dieser weggelassen wird, wird die zuletzt kompilierte Architektur verwendet.

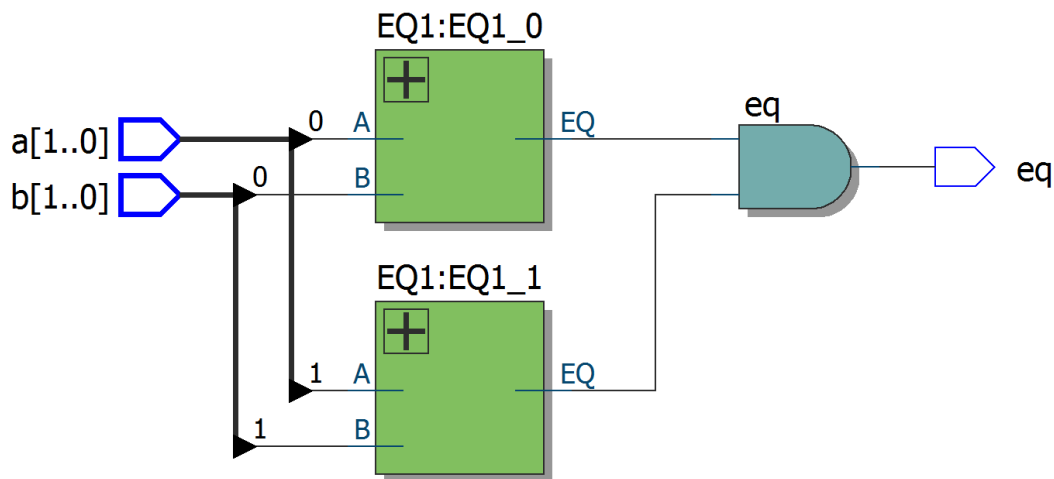
Darstellung im RTL Viewer:



... sowie mit „aufgeklapptem“ HA2-Block:



→ **Übung:** Durch Wiederverwendung der Komponente, die zwei einzelne Eingangssignale auf Gleichheit prüft, ist eine Schaltung zu entwerfen, die zwei 2-bit Eingangsvektoren (a(1), a(0) sowie b(1), b(0)) auf Gleichheit prüft.



→ **Übung:** Entwerfe einen 4-bit Volladdierer.

Hinweis: Bei dieser Übung dürfen NUR VHDL-Anweisungen und Bibliotheken verwendet werden, die wir bis zu diesem Zeitpunkt kennengelernt und besprochen haben (!)

Signalflussrichtung

Ein paar Anmerkungen zur Signalflussrichtung. Wie einleitend erklärt, müssen Signale eine Richtung haben, die bei der Deklaration festgelegt wird.

Zur Wiederholung:

- IN** Eingangssignal. Dieses kann nicht modifiziert werden, darf also nur auf der „rechten“ Seite stehen oder bei einer Signalabfrage verwendet werden.
- OUT** Ausgangssignal. Darf nur auf der linken Seite einer Signalzuweisung stehen¹².

¹² Wurde in VHDL2008 geändert...

BUFFER	Ausgangssignal, das auch als Eingang oder zur Signalabfrage verwendet werden kann
INOUT	bidirektionales Signal beim Datentyp <code>std_logic</code>

Angenommen, ein RS-Flipflop wird benötigt und dieses RS-FlipFlop soll mit 2 NOR Gattern realisiert werden.

Vermutlich würde der Code so (oder so ähnlich) erstellt werden:

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

entity Nor_RS_FF is
  port ( R, S:    in std_ulogic;
        Q, Q_N:  out std_ulogic
  );
end Nor_RS_FF;

architecture Entwurf1 of Nor_RS_FF is
begin
  Q <= R nor Q_N;
  Q_N <= S nor Q;
end Entwurf1;
```

Die Beschreibung ist nicht VHDL93 konform¹³... warum – und wie sieht der richtige Code aus?

Verknüpfungsoperator &

Mit dem Verknüpfungsoperator & (concatenation operator) können Segmente von Elementen zusammengefasst werden – und somit kann ein breiterer Bus (Vektor) erzeugt werden.

Beispiel:

```
...
signal E1: std_ulogic;
signal E3: std_ulogic_vector(2 downto 0);
signal E4: std_ulogic_vector(3 downto 0);
signal E7: std_ulogic_vector(6 downto 0);
signal E8: std_ulogic_vector(7 downto 0);
begin
  E3 <= E1 & "00";
  E8 <= E4 & E7(3 downto 0);
...

```

Durch geschickte Verwendung des Verknüpfungsoperators können auch Schiebeoperationen sehr einfach durchgeführt werden.

Beispiel:

```
...
signal X: std_ulogic_vector(7 downto 0);
signal SHL, ROT: std_ulogic_vector(7 downto 0);
begin
  -- schiebe X um 3 bits nach rechts und fülle mit 0 (logic shift)
  SHL <= "000" & X(7 downto 3);

  -- rotiere X um 3 bits nach rechts

```

¹³ Assignments → Settings → Analysis & Synth... → VHDL Input → VHDL 2008: Umsetzung fehlerfrei...

```

    ROT <= X(2 downto 0) & X(7 downto 3);
    ...

```

when-else Bedingung

VDHL gehört zur Gruppe der Hardware – Beschreibungssprachen. Bislang wurde eine Aufgabenstellung mit einer Beschreibung auf der Logikebene gelöst. D.h. eine gegebene Aufgabenstellung musste dahingehend analysiert werden, dass sie in einer Bool'schen Algebra beschrieben werden konnte... also unter Verwendung von UND, ODER, NICHT, ...

Hier befinden wir uns auf einer sehr niedrigen Ebene – und der Weg, um die Bool'sche Beschreibungsform herzuleiten, kann mitunter sehr aufwendig und fehlerträchtig sein (Aufstellen einer Wahrheitstabelle, Herleiten einer disjunktiven oder konjunktiven Normalform, ...). VHDL bietet jedoch auch „mächtigere“ Sprachkonstrukte an. Im Zuge der sequentiellen Logik werden wir einige davon kennen lernen, als „Vorgeschmack“ sollen hier zwei vorgestellt werden - die **WHEN-ELSE** Bedingungsabfrage und die **WITH-SELECT** Anweisung.

Syntax der *when-else* Anweisung:

```

Signal_Name <= Anweisung_1 when boolean_Bedingung_1_erfüllt else
                Anweisung_2 when boolean_Bedingung_2_erfüllt else
                ...
                Anweisung_3;

```

→ **Beispiel:** Prioritätsdekoders:

Eingang E = {E3, E2, E1, E0}	Ausgangscode
1 - - -	100
0 1 - -	011
0 0 1 -	010
0 0 0 1	001
0 0 0 0	000

```

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity Prio_Dekoder is
    port ( E: in std_ulogic_vector(3 downto 0);
          Code: out std_ulogic_vector(2 downto 0)
    );
end Prio_Dekoder;

architecture Verhalten of Prio_Dekoder is
begin
    Code <= "100" when (E(3)='1') else
            "011" when (E(2)='1') else
            "010" when (E(1)='1') else
            "001" when (E(0)='1') else
            "000";
end Verhalten;

```

Anmerkung: als Architektur-Name wurde "Verhalten" (engl. behaviour) gewählt, da hier keine

Beschreibung mehr rein auf Gatterebene (\rightarrow RTL) erfolgt.

\rightarrow **Übung:** Entwerfe einen 1-aus-4 Codeumsetzer mit Enable-Eingang unter Verwendung von *when-else*.

with – select

\rightarrow **Beispiel:** Es ist ein 4-zu-1 Multiplexer zu entwerfen. Über einen 2bit breiten Select-Bus S muss jedes einzelne Signal eines 4-bit breiten Eingangsbusses auf einen Ausgang geschaltet werden können.

Vermutlich würden wir –mit unserem bisherigen Kenntnisstand– die Dekoder-Ansteuerungsgleichungen aufstellen, über vier 2-fach UND-Gatter die „Schalter“ realisieren und die „Schalter“-Ausgänge mit einem 4-fach-ODER Gatter zu einem Ausgangssignal zusammenfassen.

Hier nun das Code-Beispiel unter Anwendung von *with-select*:

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

entity Mux4x1 is
    port ( S: in std_ulogic_vector(1 downto 0);
          E: in std_ulogic_vector(3 downto 0);
          Y: out std_ulogic
    );
end Mux4x1;

architecture behaviour of Mux4x1 is
begin
    with S select
        Y <= E(0) when "00",
              E(1) when "01",
              E(2) when "10",
              E(3) when others;    -- when "11";
end behaviour;
```

OTHERS ist ein weiteres, reserviertes Schlüsselwort und steht für alle, noch nicht berücksichtigten Kombinationen.

Somit haben wir eine Abstraktionsebene des HW-Schaltungsentwurfes erreicht, bei dem wir uns nicht mehr über Logikgatter Gedanken machen müssen. Der HDL-Compiler wird dies für uns korrekt umsetzen.

With-select ist ein Beispiel, bei dem VHDL-Anweisungen mit einem Beistrich (,) getrennt werden. Der gesamte **WITH-SELECT** Block endet mit einem Semikolon (;). Da man sehr oft englischsprachigen VHDL Code findet, wurde hier der gängige Architekturname „behaviour“ verwendet.

→ **Beispiel:** Prioritätsdekoder

```

architecture Realisierung_OK of Prio_Dekoder is
begin
  with E select
    Code <= "100" when "1000" | "1001" | "1010" | "1011" | "1100" | "1101" |
              "1110" | "1111",
            "011" when "0100" | "0101" | "0110" | "0111",
            "010" when "0010" | "0011",
            "001" when "0001",
            "000";
end Realisierung_OK ;

```

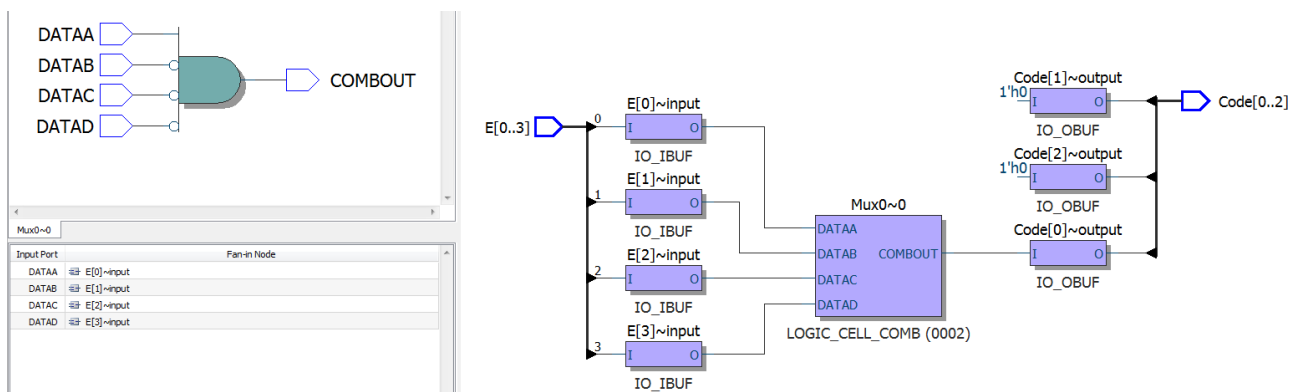
Da man in der Regel Schreibarbeit sparen will, würde man vermutlich intuitiv zu ‘-’ (don’t care) greifen...

```

architecture Realisierung_NOK of Prio_Dekoder is
begin
  Code <= "100" when "1---" else
            "011" when "01--" else
            "010" when "001-" else
            "001" when "0001" else
            "000";
end Realisierung_NOK ;

```

... und da dies syntaktisch korrekt ist, lässt sich dieser VHDL Code auch umsetzen. Allerdings wird er nicht so funktionieren, wie vermutlich erwartet... das Ausgangscodewort ist nur bei der Eingangskombination 0001 = 001, bei allen anderen Eingangskombinationen ist Code = 000:



Begründung: “-” ist ein eigener, gültiger Wert und wurde nicht von “0” oder “1” abgeleitet!

Hinweis-Meldungen während der Kompilierung:

```

Warning (10325): VHDL Choice warning at Prio_Dekoder.vhd(14): ignored choice containing meta-value ""1---""
Warning (10325): VHDL Choice warning at Prio_Dekoder.vhd(15): ignored choice containing meta-value ""01--""
Warning (10325): VHDL Choice warning at Prio_Dekoder.vhd(16): ignored choice containing meta-value ""001-""
Warning (13024): Output pins are stuck at VCC or GND
Warning (13410): Pin "Code[1]" is stuck at GND
Warning (13410): Pin "Code[2]" is stuck at GND

```

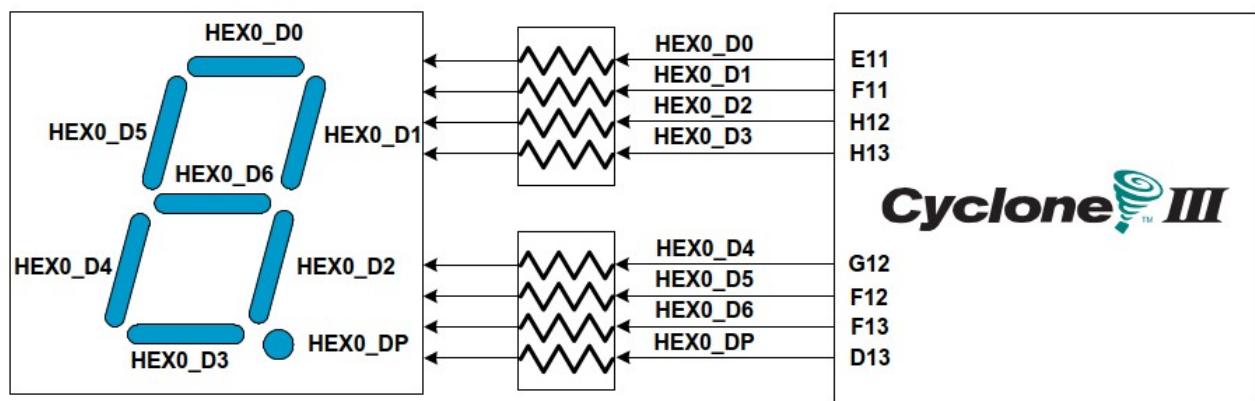

→ **Übung:** Entwerfe einen 7-Segment-Decoder, der ein 4 bit Eingangssignal (bcd(3:0)) in die Zeichen 0-9, A-F codiert und diese Daten in Form eines 7-bit LED-Ansteuerungsbusses ausgibt. Zusätzlich sollte über einen Eingang dp der Dezimalpunkt ein-/ausschaltbar sein.

Beachte: die 7-Segment-Anzeigen des DE0-Boards sind Low-Aktiv!

Pinbelegung: $bcd(3:0) \rightarrow SW(3:0)$ $dp \rightarrow SW9$ $HEX0: DP, (gfedcba)$

Node Name	Direction	Location	I/O Bank	VREF Group	Fitter Location	I/O Standard
in bcd[3]	Input	PIN_G4	1	B1_N0	PIN_G4	3.3-V LVTTTL
in bcd[2]	Input	PIN_H6	1	B1_N0	PIN_H6	3.3-V LVTTTL
in bcd[1]	Input	PIN_H5	1	B1_N0	PIN_H5	3.3-V LVTTTL
in bcd[0]	Input	PIN_J6	1	B1_N0	PIN_J6	3.3-V LVTTTL
in dp	Input	PIN_D2	1	B1_N0	PIN_D2	3.3-V LVTTTL
out sseg[7]	Output	PIN_D13	7	B7_N1	PIN_D13	3.3-V LVTTTL
out sseg[6]	Output	PIN_F13	7	B7_N1	PIN_F13	3.3-V LVTTTL
out sseg[5]	Output	PIN_F12	7	B7_N1	PIN_F12	3.3-V LVTTTL
out sseg[4]	Output	PIN_G12	7	B7_N1	PIN_G12	3.3-V LVTTTL
out sseg[3]	Output	PIN_H13	7	B7_N1	PIN_H13	3.3-V LVTTTL
out sseg[2]	Output	PIN_H12	7	B7_N1	PIN_H12	3.3-V LVTTTL
out sseg[1]	Output	PIN_F11	7	B7_N1	PIN_F11	3.3-V LVTTTL
out sseg[0]	Output	PIN_E11	7	B7_N1	PIN_E11	3.3-V LVTTTL

HEX0



→ **Übung:** zwei 4-bit Zahlen sind zu addieren und das Ergebnis (in Hexadezimaler Form) ist darzustellen. Die Zahlen sind per Schieberegister vorzugeben (Zahl1: SW(9:6), Zahl2: SW(3:0)) – die Umsetzung ist unter Verwendung bereits erstellter VHDL-Blöcke vorzunehmen!

2.3 Abschlussprojekt

Der TTL-Baustein 74LS85 ist ein kaskadierbarer 4-bit Komparator. Studiere das Datenblatt und baue diesen Baustein (funktionell) in VHDL nach!