



DH BW

Duale Hochschule
Baden-Württemberg

Entwurf Digitaler Systeme

Torben Mehner, 30. September 2024

Organisatorischer Teil

Ablauf der Vorlesung

- ▶ 5. Semester
 - ▶ Veranstaltung 3 SWS
 - ▶ Klausur
- ▶ 6. Semester
 - ▶ Nur NT: Labor 2 SWS (Dorwarth?)

Veranstaltungsaufbau

- ▶ Grundlagen
- ▶ Aufbau Digitalschaltungen
- ▶ Kombinatorische Logik
- ▶ Sequentielle Logik
- ▶ Zustandsautomaten
- ▶ Übungen

Prüfung

- ▶ 90 Minuten
- ▶ Hilfsmittel: VHDL-Spickzettel
- ▶ Note zählt 100% (mit Labor)
- ▶ Aufbau (ohne Gewähr)
 - ▶ Verständnisfragen
 - ▶ Programmieraufgaben (Stift und Papier)
 - ▶ Fehlersuche im Quellcode

Lernziele

- ▶ Grundkenntnisse CMOS-Technologie
- ▶ Implementierungsvarianten, Vor- und Nachteile
- ▶ Übersicht über Hardwarebeschreibungssprachen
- ▶ Anwendung von VHDL (Modellierung, Simulation, Synthese)
- ▶ Selbstständig Hardwareentwürfe erstellen und testen

Literatur (1/2)

- ▶ Reichardt, J. / Schwarz B.; VHDL-Synthese; Oldenbourg Verlag
- ▶ VHDL-Buch aus dem Netz (Kurslaufwerk)
- ▶ Pong, P. Chu: FPGA Prototyping by VHDL examples
- ▶ Kessel, Bartholomä: Entwurf von digitalen Schaltungen und Systemen
- ▶ Lehmann, Gunther; Schaltungsdesign mit VHDL
- ▶ Künzli, Martin; Vom Gatter zu VHDL; vdf Hochschulverlag Zürich

Literatur (2/2)

- ▶ ARM - Rechnerarchitekturen für System-on-Chip-Design, Mitp-Verlag
- ▶ Siemers, Christian; Prozessorbau; Hanser Verlag

Wiederholung

Wahrheitstabellen

- ▶ Beliebige Anzahl Eingänge und Ausgänge
- ▶ 2^n Zustände um jede Möglichkeit abzubilden
- ▶ Bilden logische Funktionen ab

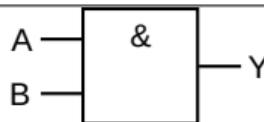
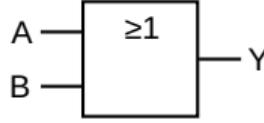
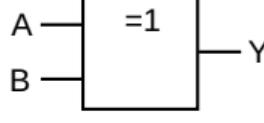
Beispiel

a	b	q
0	0	0
0	1	0
1	0	0
1	1	1

UND-Funktion mit 2 Eingängen

Wiederholung

Bool'sche Funktionen

Name	Rechenzeichen	Symbol	Beschreibung
AND	\wedge		Alle 1
OR	\vee		Mindestens eine 1
XOR	\oplus		Genau eine 1
NOT	\neg		Logikwert umkehren

Einstieg

Ziel der Vorlesung

- ▶ Kenntnisse über digitale Schaltungen
- ▶ Abbildung von Funktionen in digitalen Systemen
- ▶ Verwendung der Hardwarebeschreibungssprache VHDL

Hardwarebeschreibungssprachen

Anwendungsgebiete

- ▶ Förmliche Beschreibung von Logikfunktionen
- ▶ KEINE Programmiersprache
- ▶ Parallel Ausführung
- ▶ Konfigurieren von FPGAs

Hardwarebeschreibungssprachen

VHDL

- ▶ Very High Speed Integrated Circuit Hardware Description Language
- ▶ Modulbasierte Sprache
- ▶ Entwickelt von DoD, IBM, TI und Intermetrics
- ▶ In Europa weit verbreitet

```
1 entity XOR_ent is
2     port(    A: in std_logic;
3             B: in std_logic;
4             Q: out std_logic
5     );
6 end XOR_ent;s
7
8 architecture behavioral of
9 XOR_ent is
10 begin
11     Q <= A xor B;
12
13 end behavioral;
```

Verilog

- ▶ C-ähnliche Syntax
- ▶ Entwickelt von Gateway Design Automation und Cadence
- ▶ In den USA weit verbreitet

```
1 // Deklaration der Variablen
   als einfache Leitung
2 wire result, a, b;
3
4 // Es gibt 3 Varianten, um ein
   (bitweises) UND-Gatter zu
   beschreiben
5 assign result = a ^ b;
```

Erweiterungen

- ▶ AMS-Erweiterung für VHDL und Verilog verfügbar
- ▶ AMS: Analog and Mixed-Signal
- ▶ Ermöglicht Simulation von Designs in analogen Umgebungen

SystemC

- ▶ C++-Library um Softwareentwicklern Hardwareentwicklung zu ermöglichen
- ▶ Fügt Nebeläufigkeit, Timing und HW-Datentypen zu C++ hinzu
- ▶ Hohe Abstraktionsebene

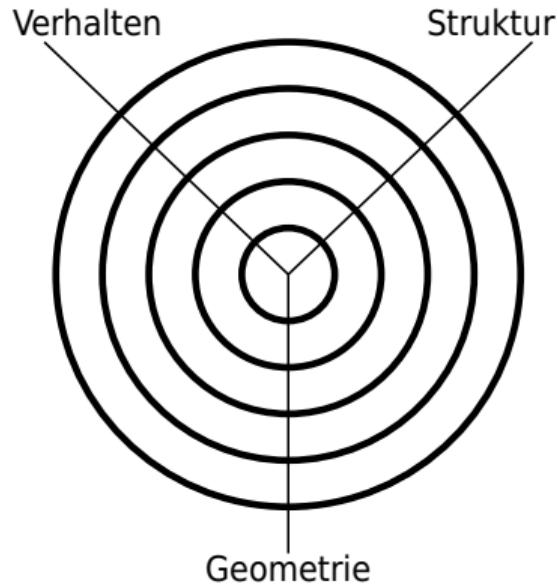
```
1 #include "systemc.h"
2 int sc_main (int argc, char*
3   argv[]) {
4     int a = 1;
5     int b = 1;
6     int c;
7
8     //bitwise AND operator ,
9     c = a ^ b;
10
11    cout <<"Value of c = " << c
12    << endl;
13    return 0;// Terminate
14    simulation
15 }
```

Grundlagen

Y-Diagramm

Y-Diagramm

- ▶ auch: Gajski-Diagramm
- ▶ Darstellung
 - ▶ Verhalten
 - ▶ Struktur
 - ▶ Geometrie
- ▶ weit außen, hohe Abstraktion



Verhalten

- ▶ Algorithmus (Behavioral)
- ▶ Register-Transfer
- ▶ Boolesche Gleichungen
- ▶ Differentialgleichungen

$$f = (\bar{a} \wedge b \wedge \bar{c}) \vee (a \wedge \bar{b} \wedge d)$$

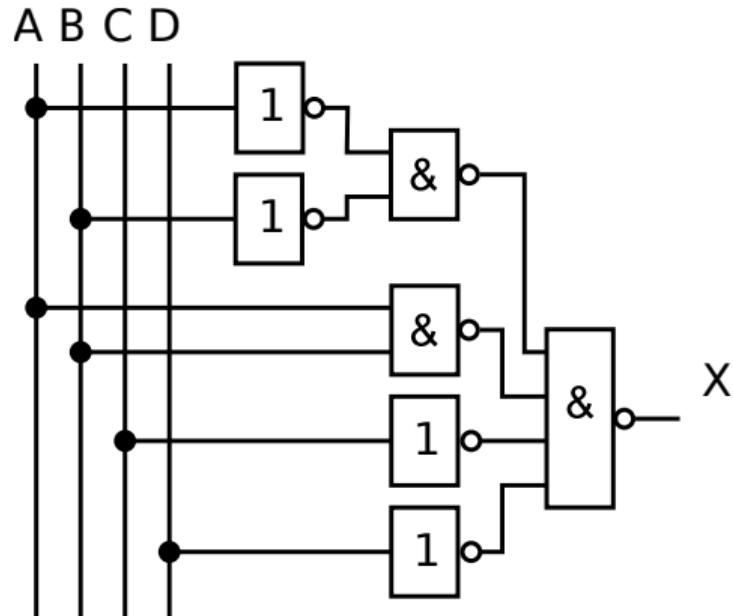
„Wenn das, dann das.“

Y-Diagramm

Struktur

- ▶ CPU, Speicher, Multiplexer
- ▶ Zustandsautomat
- ▶ Gatter, Flip-Flops
- ▶ Transistoren

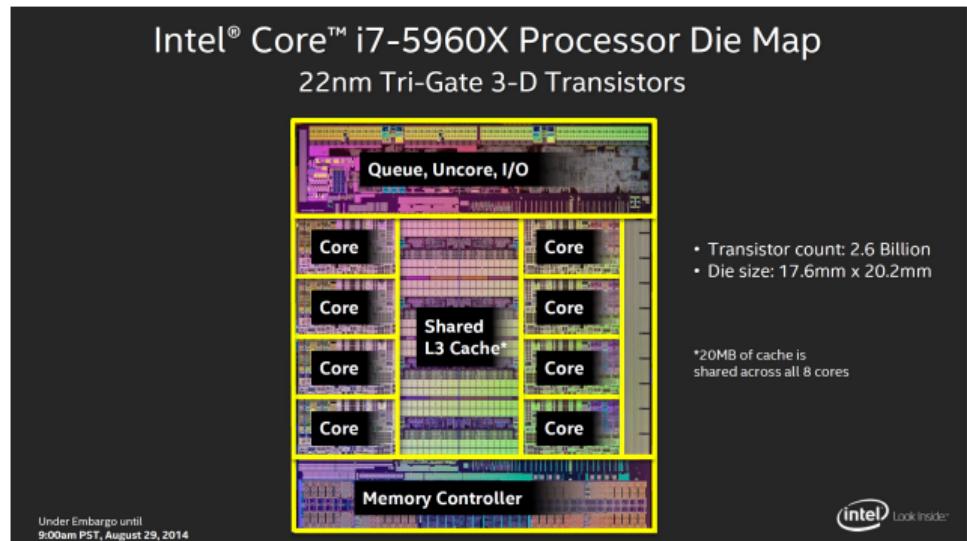
„Das ist mit dem verbunden.“



Y-Diagramm

Geometrie

- ▶ Cluster
- ▶ Floor Plan
- ▶ Zellen
- ▶ Masken



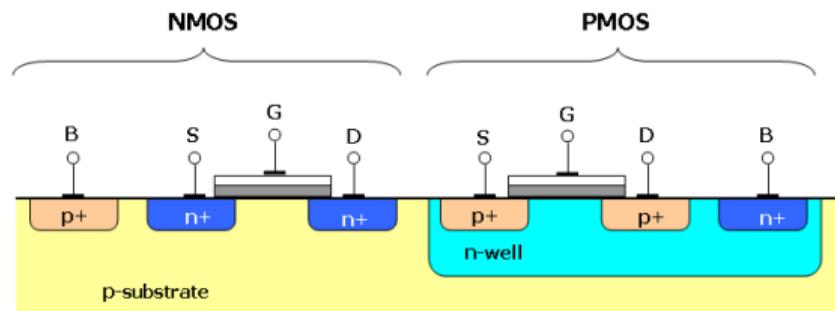
Grundlagen

Technologien

- ▶ TTL
Transistor-Transistor-Logic
(veraltet)
- ▶ CMOS
- ▶ FinFET (22nm - 3nm)
- ▶ GAAFET (kleiner 5nm)

Zukunftstechnologien:

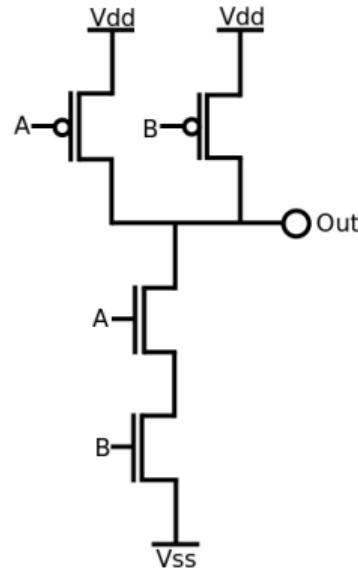
- ▶ Kohlenstoff-Nanoröhrchen
(CNT)
- ▶ Plastik (Geringe



By Reza Mirhosseini, Public Domain

CMOS

- ▶ Complementary Metal-Oxide-Semiconductor
- ▶ Grundlage: MOSFETs
- ▶ Planarer Aufbau des Transistors
- ▶ Kaum Verlustleistung (statisch)
- ▶ Hohe Leckströme bei zu kleinen Strukturen

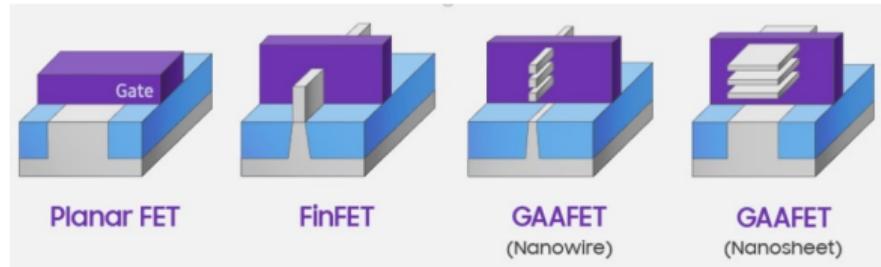


By JustinForce, CC BY-SA 3.0

Grundlagen

FinFET, GAAFET

- ▶ FinFET: Finnenförmige Kanäle
- ▶ GAAFET: gate-all-around FET
- ▶ Dreidimensionaler Aufbau
- ▶ Gate umschließt Kanal

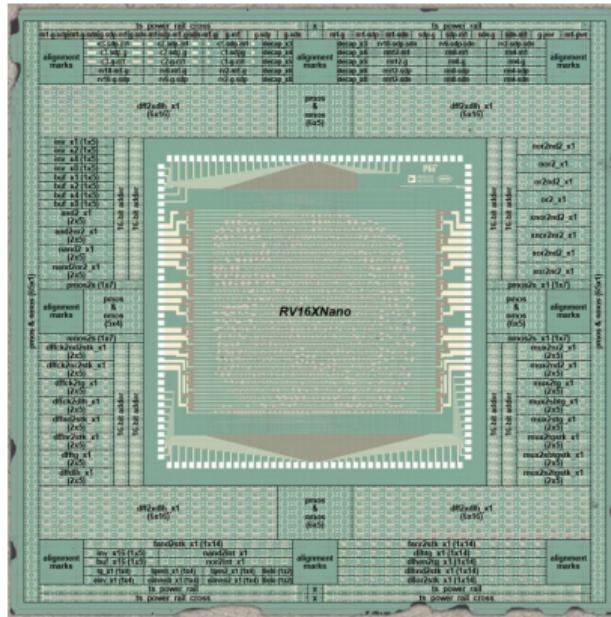


By Cadence

Grundlagen

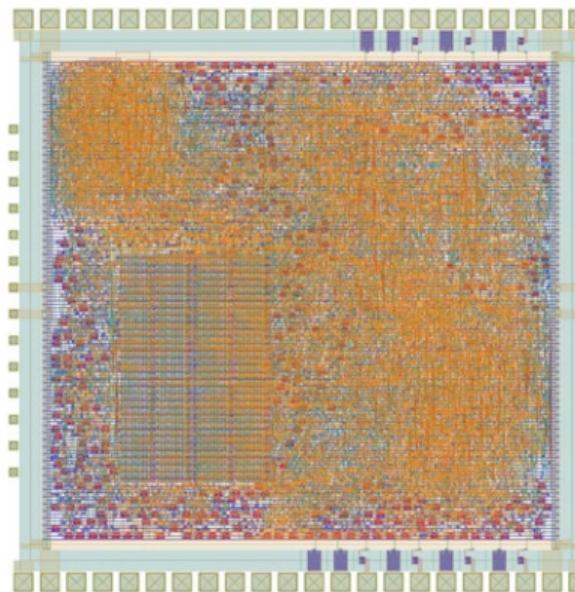
CNT - Carbon-Nanotubes

- ▶ Aktuell ca. 1µm Nanoröhrchen
- ▶ Geringe Verlustleistung
- ▶ August 2019: Erster 16-Bit Prozessor (MIT)
- ▶ Herstellung ist fehleranfällig



PlasticARM

- ▶ 0,8µm Metallocid TFTs
- ▶ Geringe Verlustleistung
- ▶ Januar 2021: Erster 32-Bit Prozessor (Cambridge)
- ▶ Herstellung mit Thin-Film-Transistors (TFT)

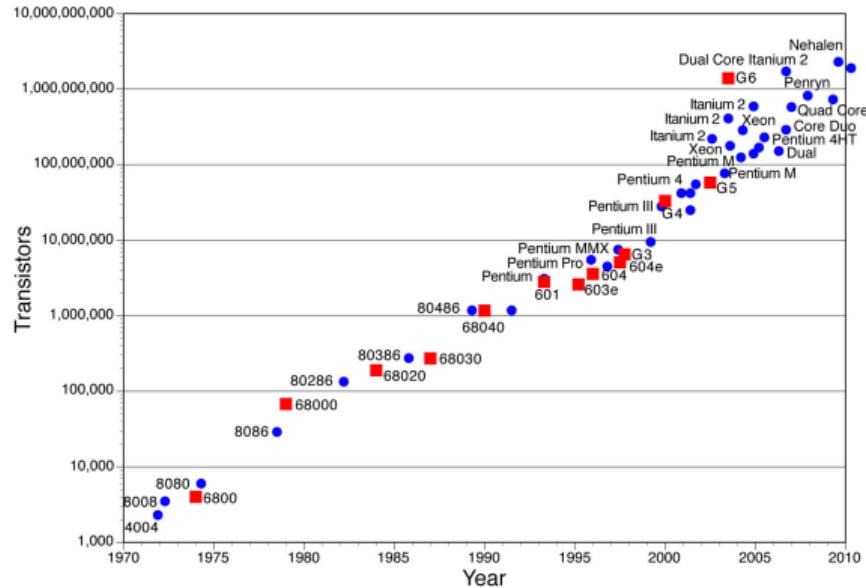


Nature - PlasticARM

Grundlagen

Mooresches Gesetz

Die Anzahl der Transistoren auf einem IC verdoppelt sich alle 12-24 Monate

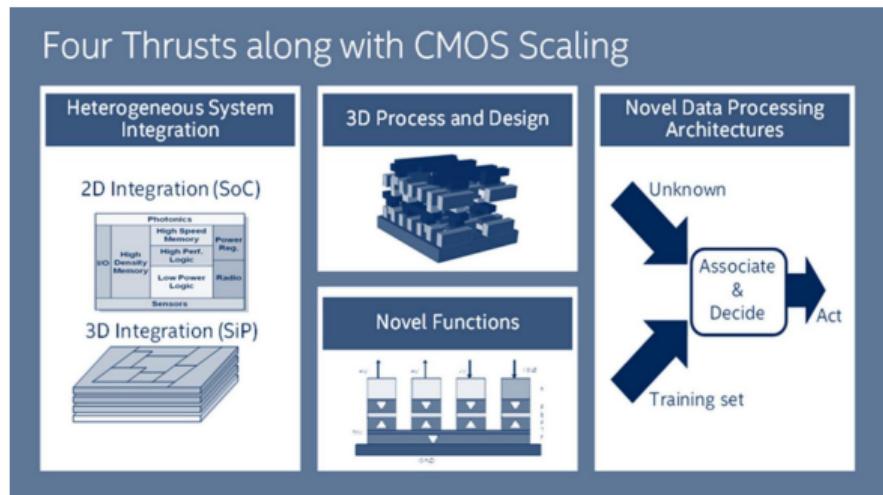


Grundlagen

Mooresches Gesetz

„Das Mooresche Gesetz ist tot –
lang lebe das Mooresche Gesetz!“

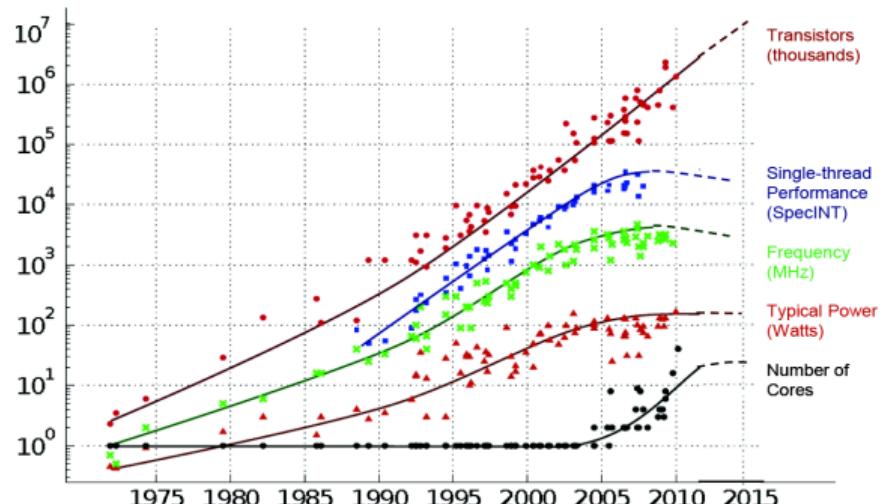
- ▶ Neue Technologien zur Beibehaltung des Wachstums nötig
- ▶ z.B. Chiplet-Design, neue Fertigungstechnologien



Grundlagen

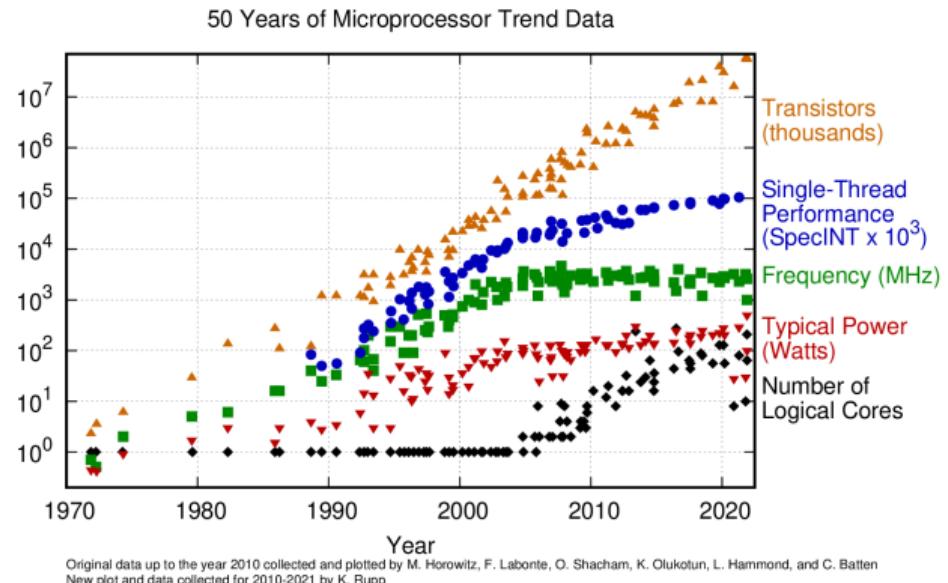
Mooresches Gesetz

35 YEARS OF MICROPROCESSOR TREND DATA



Grundlagen

Mooresches Gesetz



Implementierung digitaler Logik

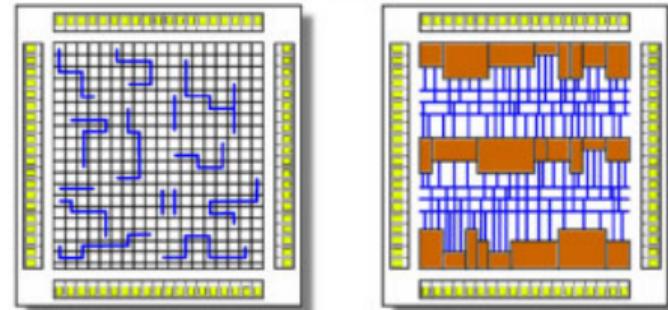
Übersicht

- ▶ General Purpose
 - ▶ Mikroprozessoren (MCU, CPU)
 - ▶ Signalprozessor (DSP)
 - ▶ Speicher
- ▶ Anwender-konfigurierbare ICs
 - ▶ FPGAs
 - ▶ PLDs
- ▶ Application Specific Integrated Circuit (ASIC)
 - ▶ Semi-Custom ICs
 - ▶ Sea of Gate-Arrays
 - ▶ Standardzellen
 - ▶ Full-Custom ICs

Standardzellen-Design

- ▶ vorgegebene Standardzellen
 - ▶ Anordnung und Verdrahtung flexibel
 - ▶ Vor- und Nachteile wie Sea-of-Gate-Array
 - ▶ schneller Entwurf durch EDA-Tools
- Mittlere Stückzahlen rentabel (<1 Mio.)

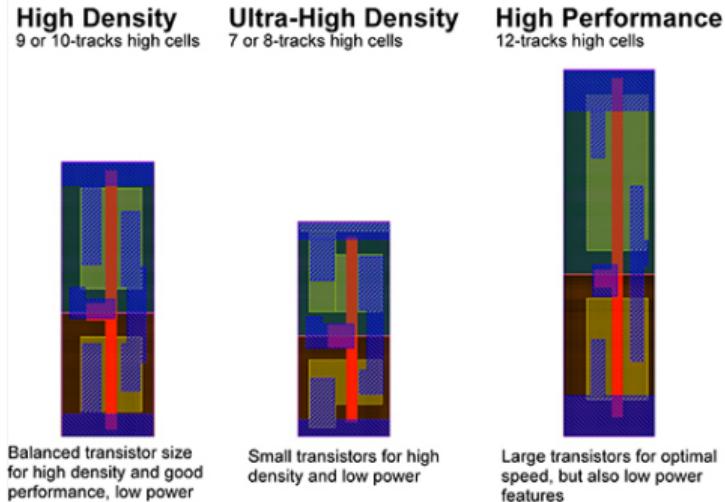
Gate Arrays Standard Cells



Standardzellen vs Sea-of-Gate-Array

Standardzellen-Design

- ▶ Unterschiedliche Standardzellen gleichen Typs
- ▶ Optimierung nach verschiedenen Faktoren

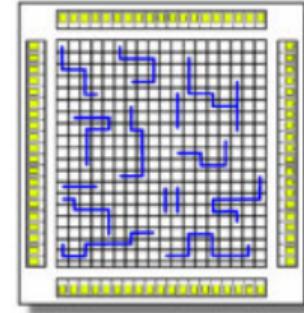


AND-Gate in drei Ausführungen

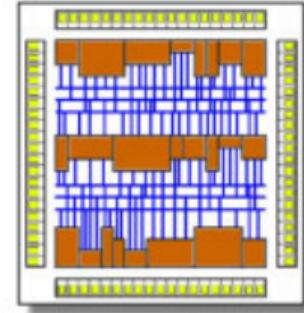
Sea of Gate-Arrays

- ▶ Logikschaltungen vorgegeben
 - ▶ Flexible Verdrahtung
 - ▶ Performant bei kleinen Entwürfen
 - ▶ Verdrahtungsprobleme bei großen Entwürfen
 - ▶ schneller Entwurf durch EDA-Tools
- Mittlere Stückzahlen rentabel (<1 Mio.)

Gate Arrays



Standard Cells

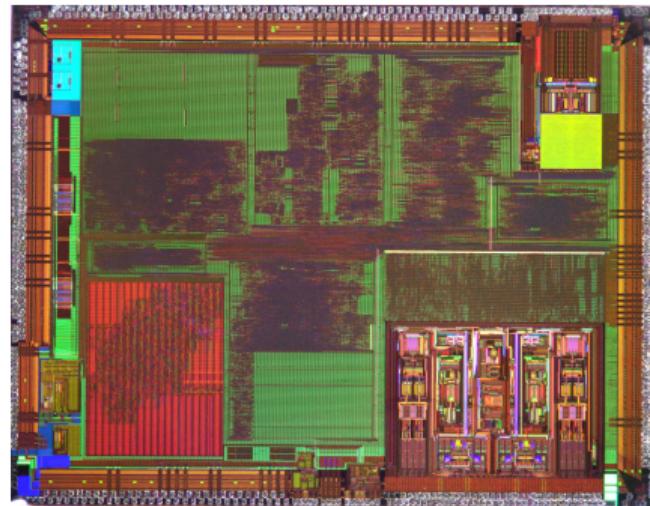


Standardzellen vs Sea-of-Gate-Array

Full-Custom-ICs

- ▶ voll flexibel (analog und digital)
- ▶ beste Performance
- ▶ geringe Verlustleistung
- ▶ Entwurf aufwendig/teuer

→ Lohnt sich bei sehr großen Stückzahlen



Broadcom BCM1103
Gigabit Ethernet

General Purpose Processors

Instruction-Set-Architectures

- ▶ x86 (Desktop PCs)
- ▶ ARM (Eingebettete Systeme, Mobile Geräte)
- ▶ AVR, MSP430 (Mikrocontroller)
- ▶ RISC-V (Mikrocontroller, Mobile Geräte)
- ▶ POWER (Mikrocontroller)
- ▶ und weitere weniger bekannte

Instruction-Sets legen die verfügbaren Assembler-Befehle einer Architektur fest.

General Purpose Processors

x86

- ▶ Zuerst in Intels 8086 eingesetzt (1978)
- ▶ CISC (Complex Instruction Set)
- ▶ High-Power Anwendungen
 - ▶ Desktop
 - ▶ Laptop
 - ▶ Server
 - ▶ Rechenzentren



General Purpose Processors

ARM

- ▶ Acorn (heute ARM Holdings) entwickelt ARM (1985)
- ▶ Acorn RISC Machine, später Advanced RISC Machine
- ▶ RISC (Reduced Instruction Set)
- ▶ Heute Teil von Nvidia (2020)
- ▶ ARM verkauft nur Lizenzen an Hersteller, baut keine Chips



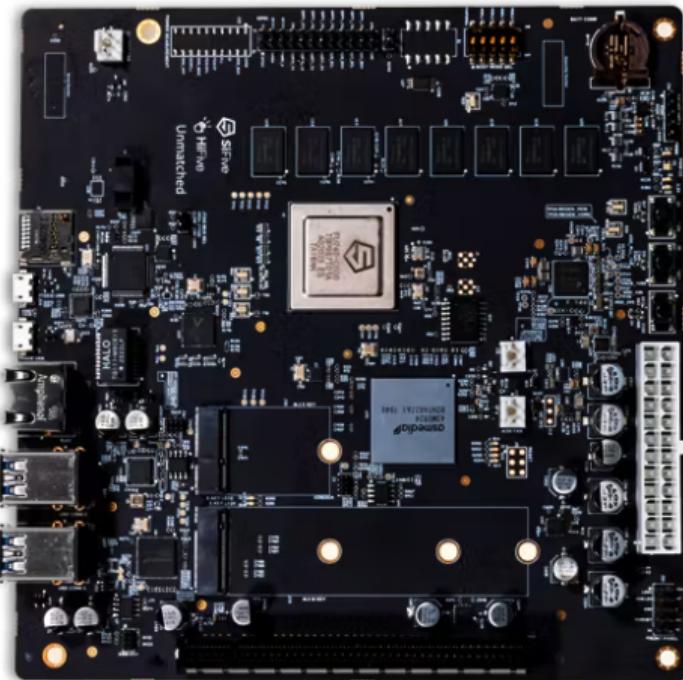
Seit ARM v6 in drei Anwendungsbereiche gegliedert:

	Verwendung	Besonderheit
Cortex-M	Mikrocontroller	Energiesparend, Kostengünstig, Zahlreiche, einfache Schnittstellen, Vorhersehbare Programmausführung
Cortex-A	Anwendungen	Hohe Leistung, Floating-Point-Einheiten, Speicherzugriffskontrolle, schnelle Schnittstellen
Cortex-R	Echtzeit	Wie M-Serie, Hohe Leistung
Cortex-X	Anwendungen	Anpassbare Performance-Version der Cortex-A

General Purpose Processors

RISC-V

- ▶ Projekt an der UC Berkely (2010-2019)
- ▶ Open-Source RISC-Architektur
- ▶ Gleicher Markt wie ARM (Mobile, Embedded)
- ▶ Verwendung in Full-Custom-Designs (IoT-Chips, Speichercontroller), CPUs und Mikrocontrollern
- ▶ 2 RISCV-Kerne im neuem RP2340



Einteilung GPPs

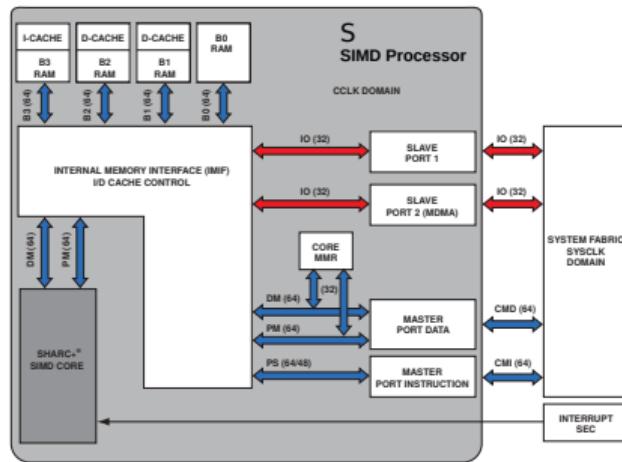
GPPs können in vier Gruppen eingeteilt werden:

	Bedeutung	Verwendung
SISD	Single Instruction Single Data	Mikroprozessor
SIMD	Single Instruction Multiple Data	DSP (u. a. Grafikkarte)
MISD	Multiple Instruction Single Data	Neuronale Netze (KI)
MIMD	Multiple Instruction Multiple Data	Parallel Computing (Multicore-CPU)

General Purpose Processors

Einteilung GPPs

Beispiel: Analog Devices Sharc



FPGAs

Übersicht

- ▶ Field Programmable Gate Array
- ▶ Abbildung beliebiger Logikfunktionen
- ▶ Test von Prozessorbaugruppen oder ganzen Prozessorarchitekturen
- ▶ Parallel Datenverarbeitung
- ▶ Im Betrieb rekonfigurierbare Schaltung

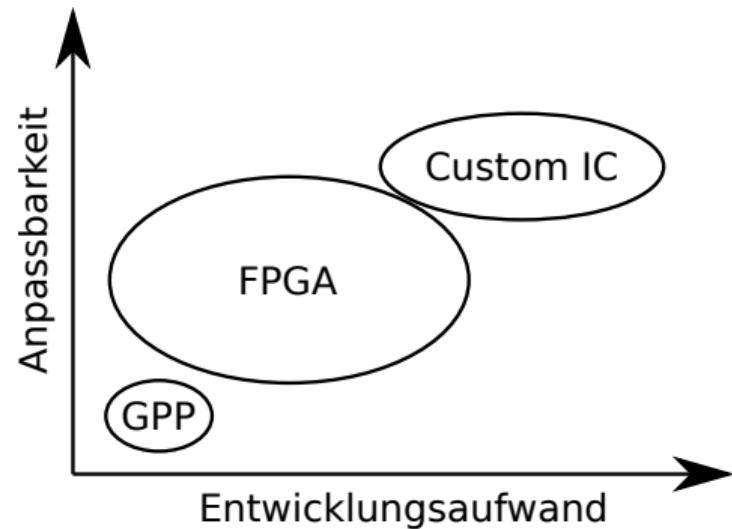
Verbreitete Marken:



FPGAs

Warum FPGAs?

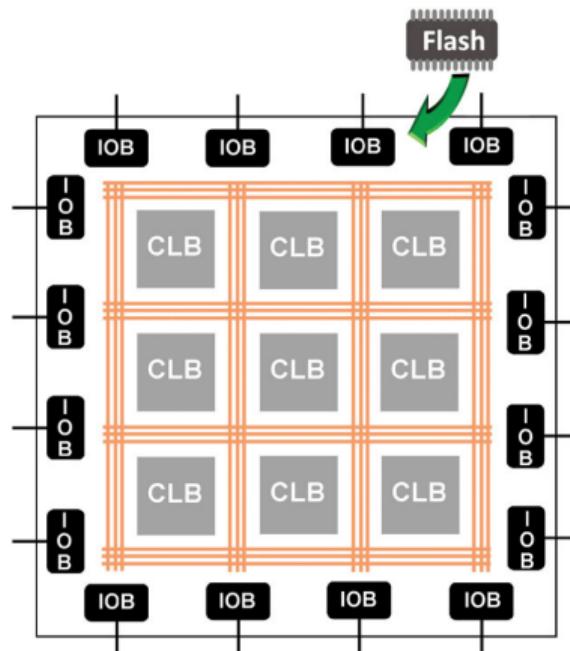
- ▶ Füllt Lücke zwischen ASIC und General Purpose Prozessoren
- ▶ Geringer Entwicklungsaufwand für hohe Anpassbarkeit
- ▶ Überwindung der Einschränkungen von GPPs durch spezialisierte Logik
- ▶ Hohe IO-Bandbreite



FPGAs

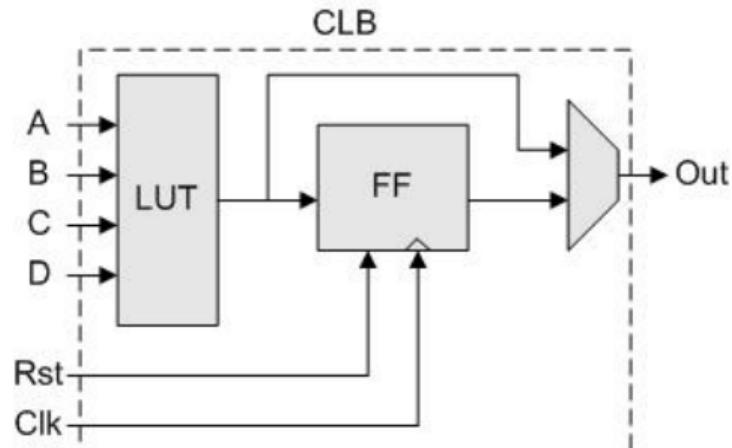
Aufbau

- ▶ CLB (auch: Slice)
- ▶ IOB
- ▶ Interconnections
- ▶ dedizierte Baugruppen



Configurable Logic Block

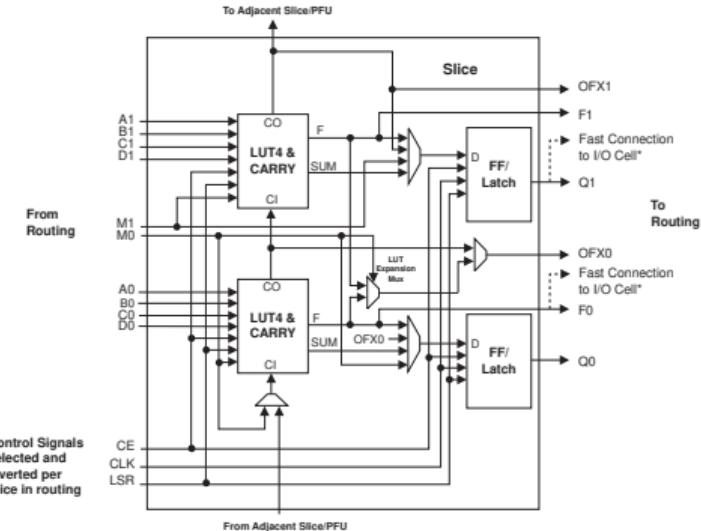
- ▶ LUT bildet Logikfunktion ab
- ▶ FF für synchrone Vorgänge
- ▶ MUX wählt synchrones/asynchrones Signal



Configurable Logic Block

Lattice MachXO Family

- ▶ 2x LUT4 oder LUT5
- ▶ Carry-Logic für Addition
- ▶ MUX kann benachbarte Slices zu LUT6-8 verbinden



Configurable Logic Block

Lattice MachXO Family

Function	Type	Signal Names	Description
Input	Data signal	A0, B0, C0, D0	Inputs to LUT4
Input	Data signal	A1, B1, C1, D1	Inputs to LUT4
Input	Multi-purpose	M0/M1	Multipurpose Input
Input	Control signal	CE	Clock Enable
Input	Control signal	LSR	Local Set/Reset
Input	Control signal	CLK	System Clock
Input	Inter-PFU signal	FCIN	Fast Carry In ¹
Output	Data signals	F0, F1	LUT4 output register bypass signals
Output	Data signals	Q0, Q1	Register Outputs
Output	Data signals	OFX0	Output of a LUT5 MUX
Output	Data signals	OFX1	Output of a LUT6, LUT7, LUT8 ² MUX depending on the Slice
Output	Inter-PFU signal	FCO	Fast Carry Out ¹

1. See Figure 2-4 for connection details.

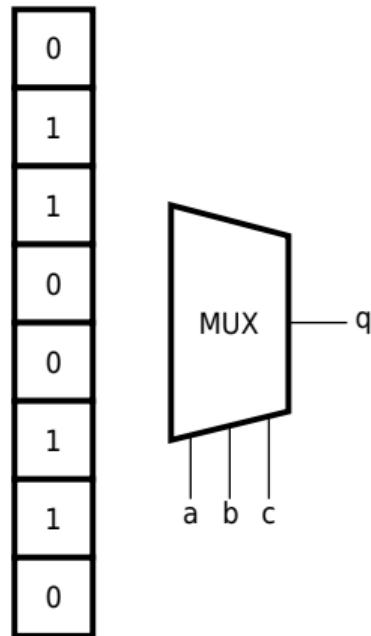
2. Requires two PFUs.

Look-Up-Tables

- ▶ SRAM-Zellen speichern Antworten
- ▶ Multiplexer wählt richtige Antwort aus

Übung

$$q = (a \oplus b) \oplus c$$



Look-Up-Tables

- ▶ SRAM-Zellen speichern Antworten
- ▶ Multiplexer wählt richtige Antwort aus

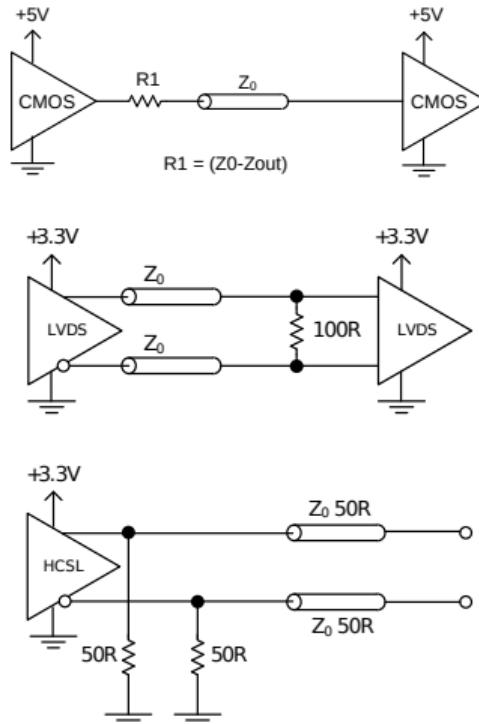
Übung

$$q = (a \oplus b) \oplus c$$

c	b	a	sig1	q
0	0	0	0	0
0	0	1	1	1
0	1	0	1	1
0	1	1	0	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	0	1

Input/Output-Blocks

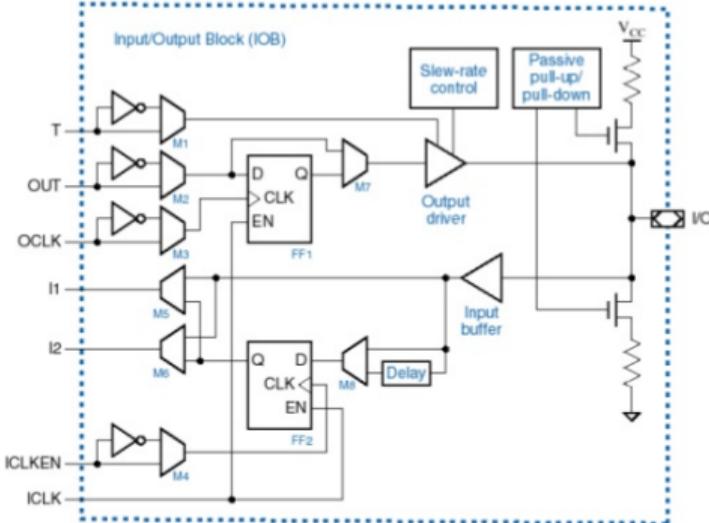
- ▶ Treibt Ausgang
 - ▶ LVCMOS
Low Voltage CMOS
 - ▶ LVDS
Low Voltage Differential Signaling
 - ▶ HCSL/CML
High Speed Current Steering Logic /
Current Mode Logic
- ▶ Liest Eingang



Input/Output-Blocks

- ▶ LVCMOS-Ausgangssignale
 - ▶ Pull-Up (H)
 - ▶ Pull-Down (L)
 - ▶ High-Impedance (Z)
 - ▶ High (1)
 - ▶ Low (0)

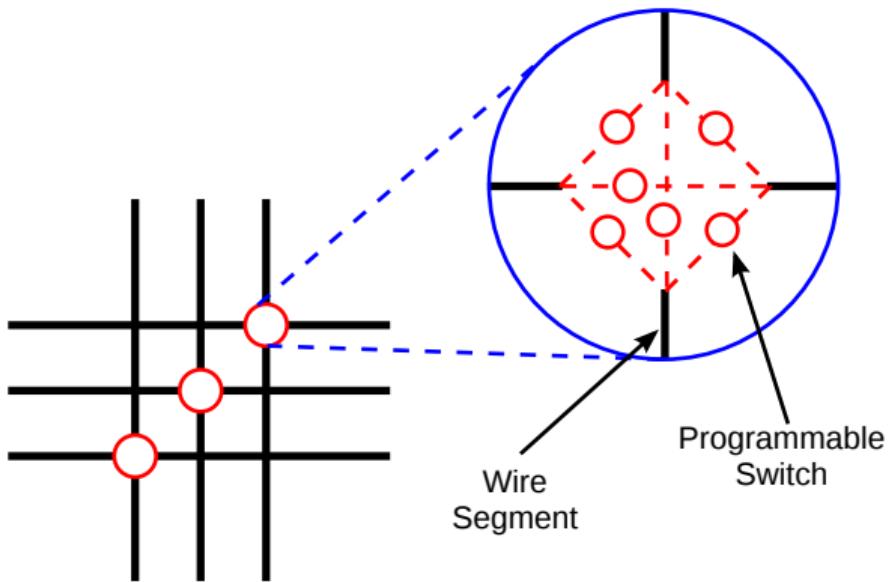
I/O Blocks



44

Interconnections

- ▶ Stellt Verbindungen her
- ▶ Aufbau programmierbare Schalter
 - ▶ SRAM-Zelle
 - ▶ MOSFET



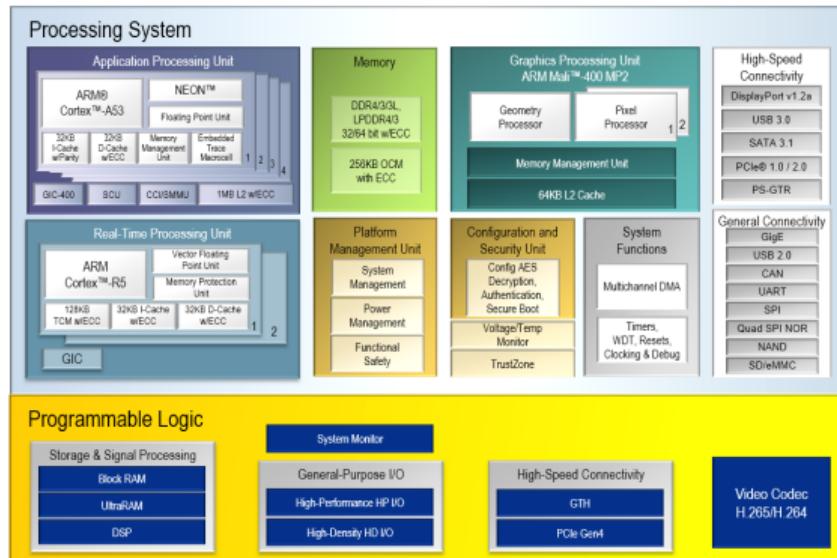
Zweckgebundene Blocks

Fest vorgegebene Strukturen erfüllen eine einzige, spezielle Aufgabe sehr gut.
Beispiele hierfür sind:

- ▶ Block-RAM
- ▶ Multipliziereinheiten (DSP-Slice)
- ▶ Timer, PLLs
- ▶ Hard-Core CPUs (Risc-V, Microblaze)
- ▶ Serializer/Deserializer (Gigabit Transceiver)

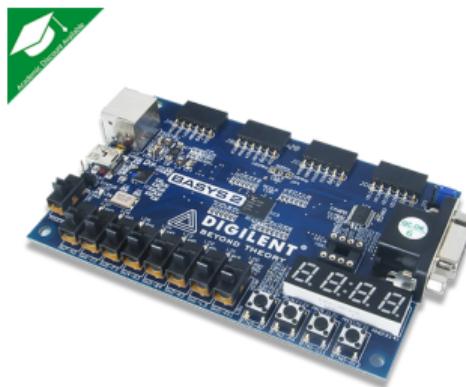
FPGAs

Aufbau Zynq Ultrascale+ EV SoC



FPGAs

Beispiele



Basys 2 Board, Spartan-3E FPGA

FPGAs

Beispiele



ZedBoard, Zynq-7000 SoC, 2x ARM A9 Core

FPGAs

Beispiele



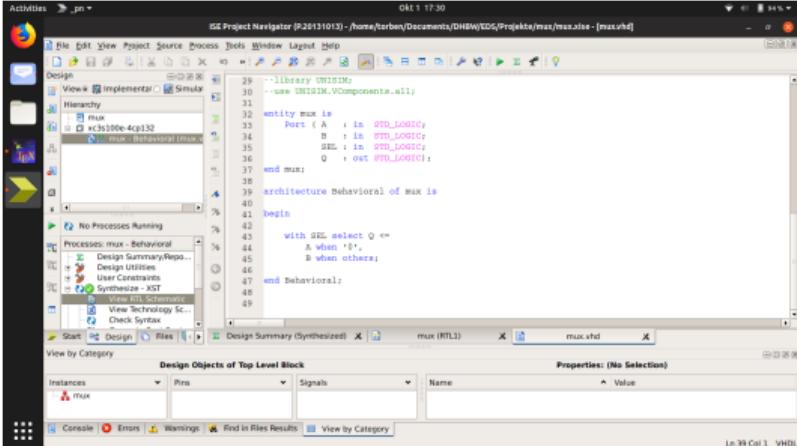
ZCU102, Zynq Ultrascale+ MPSoC, 4x ARM A53, 2x ARM R5 Cores

Entwicklungsumgebung

Entwicklungsumgebung

Xilinx ISE

- ▶ Von Xilinx entwickelt
- ▶ Erzeugt FPGA-Konfiguration aus VHDL-Datei
 - ▶ Synthese
 - ▶ Implementierung
 - ▶ Bitstream-Erzeugung



```

library UNISIM;
use UNISIM.VComponents.all;

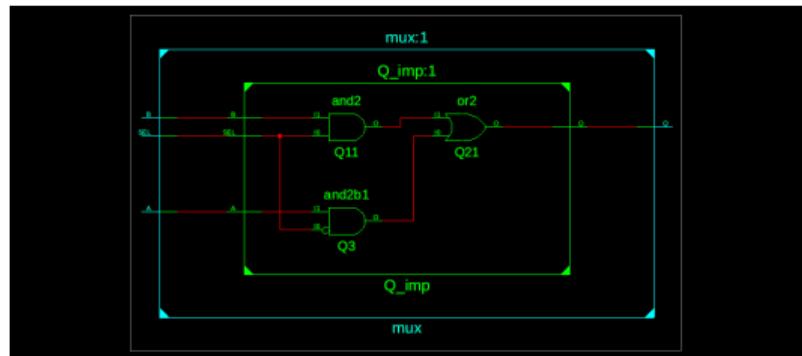
entity mux is
  Port ( A : in STD_LOGIC;
         B : in STD_LOGIC;
         SEL : in STD_LOGIC;
         Q : out STD_LOGIC);
end mux;

architecture behavioral of mux is
begin
  with SEL select Q <=
    A when '0',
    B when others;
end Behavioral;
  
```

Design Objects of Top Level Block			
Instances	Pins	Signals	Name
mux			

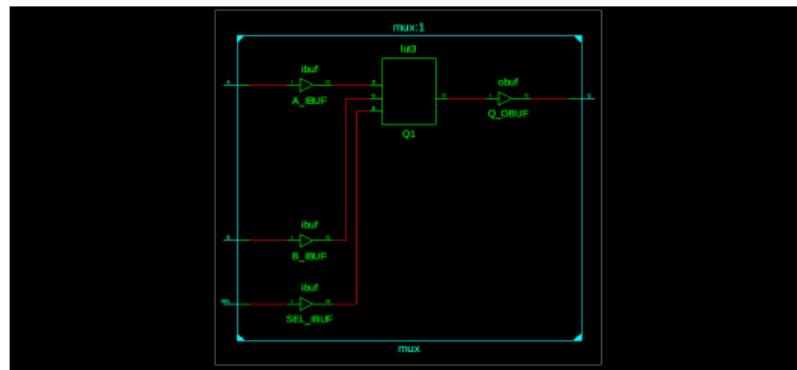
Synthese

- ▶ Übersetzt VHDL in Netzliste
 - ▶ Transformation zum RTL (Y-Diagramm)
 - ▶ Netzliste vom FPGA-Modell abhängig



Implementierung

- ▶ Zuweisung physikalischer Logikeinheiten
- ▶ Optimierung nach:
 - ▶ Platzbedarf
 - ▶ Geschwindigkeit
 - ▶ Verlustleistung
- ▶ Benötigt constraints-Datei

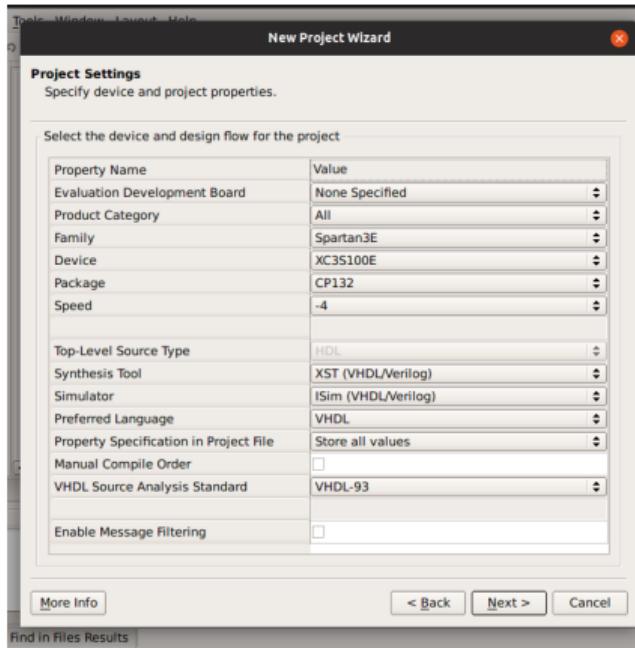


Constraints-Datei

- ▶ Enthält „Beschränkungen“, z.B.:
 - ▶ IO Pin Platzierung
 - ▶ IO Logik-Standard
 - ▶ Taktfrequenz
 - ▶ u.v.m.
- ▶ Bei Xilinx als ucf, tcl oder xdc-Datei

Entwicklungsumgebung

Einstellung für FPGA



Digilent Adept

- ▶ Lädt Konfiguration auf FPGA
- ▶ Bereitgestellt von Digilent

Kombinatorische Logik

Definition

Kombinatorische Logik

- ▶ Hat keinen Zustand (keinen Speicher)
- ▶ Ausgang reagiert sofort auf Eingang
- ▶ Vergleichbar mit LTI-Systemen

Identifier

Identifier sind Namen/Bezeichnungen

- ▶ Buchstaben, Zahlen, Unterstriche
- ▶ Beginnen mit Buchstabe
- ▶ Verboten:
 - ▶ Unterstrich am Ende
 - ▶ Zwei Unterstriche nacheinander
 - ▶ VHDL-Schlüsselwörter
 - ▶ Umlaute, Sonderzeichen

Entity

- ▶ Beschreibt Ein- und Ausgänge
- ▶ Port-Directions
 - ▶ IN: Eingang
 - ▶ OUT: Ausgang
 - ▶ BUFFER:
Lesbarer Ausgang (Nicht benutzen)
 - ▶ INOUT:
Ein- und Ausgang (Schnittstellen)

```
1 entity mux is
 2   Port ( A    : in  STD_LOGIC;
 3          B    : in  STD_LOGIC;
 4          SEL  : in  STD_LOGIC;
 5          Q    : out STD_LOGIC);
 6 end mux;
```

Entity

Allgemeine Darstellung:

```
1 entity <entity_name> is
2
3     -- Konstanten fuer Entity
4     generic(<generic_name> : <type> := <value>;
5     <other_generics> );
6
7     -- Ein- und Ausgaenge
8     port(   <port_name> : <mode> <type>;
9     <other_ports> );
10
11 end <entity_name>;
```

Generics

- ▶ Konstanten für Entity
- ▶ Erhöhen Flexibilität

```
1 entity mux is
2     Generic( N : integer := 4 );
3     Port(
4         A    : in  STD_LOGIC_VECTOR( N-1 downto 0);
5         B    : in  STD_LOGIC_VECTOR( N-1 downto 0);
6         SEL  : in  STD_LOGIC;
7         Q    : out STD_LOGIC_VECTOR( N-1 downto 0)
8     );
9 end mux;
```

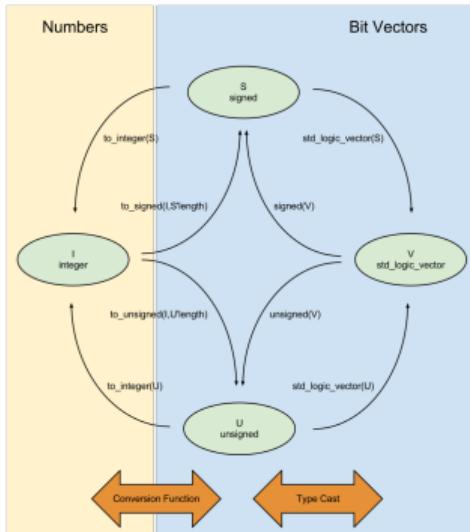
Datentypen

STD_LOGIC	Darstellung einzelner Bits
STD_LOGIC_VECTOR	Zusammenfassung mehrerer Bits, Anzahl ist zu bestimmen
SIGNED	Wie STD_LOGIC_VECTOR, signalisiert der Entwicklungsumgebung Behandlung als vorzeichenbehaftete Zahl
UNSIGNED	Wie STD_LOGIC_VECTOR, signalisiert der Entwicklungsumgebung Behandlung als nicht-vorzeichenbehaftete Zahl
INTEGER	SIGNED, mindestens mit 32 Bit synthetisiert

Datentypen

```
1 -- Bit als Low initialisiert
2 A : in STD_LOGIC := '0';
3
4 -- 8-Bit breiter Vektor, Little-Endian
5 B : in STD_LOGIC_VECTOR( 7 downto 0 ) := "00000000";
6 C : in STD_LOGIC_VECTOR( 7 downto 0 ) := (others=>'0');
7
8 -- 4-Bit breite, nicht-vorzeichenbehaftete Zahl
9 D : in UNSIGNED( 3 downto 0 ) := to_unsigned( 0, 4 );
10 E : in UNSIGNED( 3 downto 0 ) := "0000";
11
12 -- Integer Zahl
13 F : in INTEGER := 0;
```

Datentypen



<http://www.bitweenie.com/listings/vhdl-type-conversion/>

STD_LOGIC

- ▶ 'U': Uninitialisiert
- ▶ 'X': Unbekannt wegen Konflikt
- ▶ '0': Logisch 0
- ▶ '1': Logisch 1
- ▶ 'Z': Hohe Impedanz
- ▶ 'W': Schwaches unbekanntes Signal wegen Konflikt
- ▶ 'L': Pull-Down (Schwaches Low)
- ▶ 'H': Pull-Up (Schwaches High)
- ▶ '-': Don't care.

STD_LOGIC

	U	X	0	1	Z	W	L	H	-
U	'U'								
X	'U'	'X'							
0	'U'	'X'	'0'	'X'	'0'	'0'	'0'	'0'	'X'
1	'U'	'X'	'X'	'1'	'1'	'1'	'1'	'1'	'X'
Z	'U'	'X'	'0'	'1'	'Z'	'W'	'L'	'H'	'X'
W	'U'	'X'	'0'	'1'	'W'	'W'	'W'	'W'	'X'
L	'U'	'X'	'0'	'1'	'L'	'W'	'L'	'W'	'X'
H	'U'	'X'	'0'	'1'	'H'	'W'	'W'	'H'	'X'
-	'U'	'X'							

STD_LOGIC_VECTOR

```
1 -- little endian
2 S1 : out STD_LOGIC_VECTOR( 3 downto 0 );
3
4 -- big endian
5 S2 : out STD_LOGIC_VECTOR( 0 to 3 );
```

Architecture

Beschreibt Aufbau/Verhalten

```
1 architecture <arch_name> of <entity_name> is
2   -- component declaration
3   -- signal declaration
4   -- constant declaration
5   -- variable declaration
6 begin
7   -- architecture body
8 end <arch_name>;
```

Signale, Konstanten, Variablen

	Verwendung
Signal	„Übliche“ Art wird am Ende eines Prozesses auf den zuletzt geschriebenen Wert gesetzt Eignet sich nicht für Schleifen
Constant	Konstante, die aus Generics berechnet wird
Variable	Für Zwischenergebnisse innerhalb von Prozessen Wird sofort auf neuen Wert gesetzt Nur in Prozessen

Signale, Konstanten, Variablen

Beispiel mit Initialisierung im Architecture-Header:

```
1 architecture Behavioral of mux is
2     signal sig_a : STD_LOGIC := '0';
3     constant const_b : UNSIGNED( 3 downto 0) := "1111";
4 begin
5 -- architecture body
6 end Behavioral;
```

Unbedingte Zuweisungen

Zuweisung

```
1 signal sig_1 : STD_LOGIC := '0';
2 variable var_1 : STD_LOGIC := '1';
3
4 sig_1 <= '1';
5 var_1 := '1';
```

- ▶ Eine Zuweisung pro Signal! (Multi-Source-Error)
- ▶ Immer durch Pfeil (\leq) außer:
 - ▶ Initialisierung
 - ▶ Variablen

Unbedingte Zuweisungen

others-Zuweisung (schreibt alles auf '0'):

```
1 signal sig_2 : STD_LOGIC_VECTOR( N-1 downto 0 );
2 sig_2 <= (others => '0');
```

Kombinierte Zuweisung:

```
1 signal sig_1 : STD_LOGIC_VECTOR( 3 downto 0 );
2 signal sig_2 : STD_LOGIC_VECTOR( 1 downto 0 );
3 signal sig_3 : STD_LOGIC_VECTOR( 1 downto 0 );
4
5 sig_1 <= sig_2 & sig_3;
```

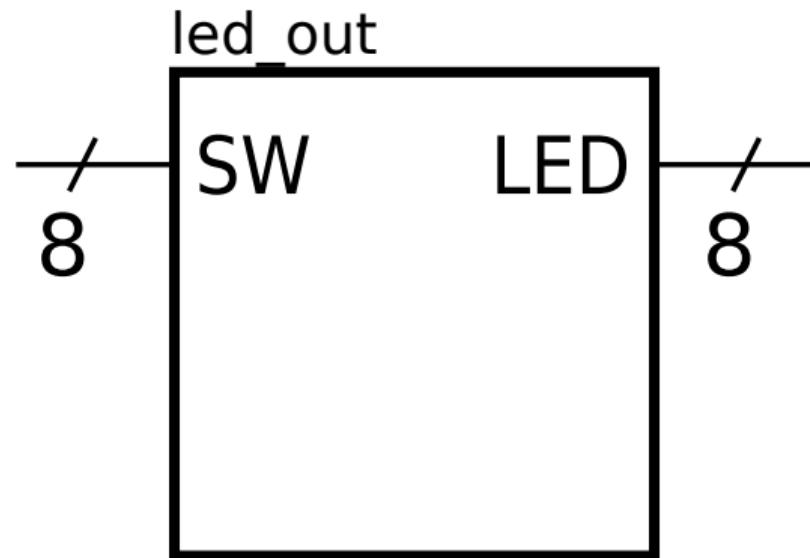
ucf-Datei

- ▶ „User Constraints File“
- ▶ Zuweisung von Port-Namen an Chip-Pins
- ▶ Kopie der Basys2-Master-ucf zu Projekt hinzufügen
- ▶ Per User Constraints →Edit Constraints öffnen

```
1 # + Port - Name
2 # | + IC - Port (BGA - Name)
3 # | | + Kommentar
4 # | | |
5 NET "Led<1>" LOC = "M11" ; # Bank = 2, Signal name = LD1
6 NET "Led<0>" LOC = "M5" ; # Bank = 2, Signal name = LD0
```

Übung 01

- ▶ Eingänge lesen
- ▶ Direkt auf LEDs schreiben



Übung 01

```
1 entity led_out is
2     Port (  SW : in  STD_LOGIC_VECTOR (7 downto 0);
3             LED : out STD_LOGIC_VECTOR (7 downto 0) );
4 end led_out;
5
6 architecture Behavioral of led_out is
7
8 begin
9
10    LED <= SW;
11
12 end Behavioral;
```

Übung 01

```
1 # Pin assignment for LEDs
2 NET "Led<7>" LOC = "G1" ; # Bank = 3, Signal name = LD7
3 NET "Led<6>" LOC = "P4" ; # Bank = 2, Signal name = LD6
4 NET "Led<5>" LOC = "N4" ; # Bank = 2, Signal name = LD5
5 NET "Led<4>" LOC = "N5" ; # Bank = 2, Signal name = LD4
6 NET "Led<3>" LOC = "P6" ; # Bank = 2, Signal name = LD3
7 NET "Led<2>" LOC = "P7" ; # Bank = 3, Signal name = LD2
8 NET "Led<1>" LOC = "M11" ; # Bank = 2, Signal name = LD1
9 NET "Led<0>" LOC = "M5" ; # Bank = 2, Signal name = LD0
```

Übung 01

```
1 # Pin assignment for SWs
2 NET "sw<7>" LOC = "N3";    # Bank = 2, Signal name = SW7
3 NET "sw<6>" LOC = "E2";    # Bank = 3, Signal name = SW6
4 NET "sw<5>" LOC = "F3";    # Bank = 3, Signal name = SW5
5 NET "sw<4>" LOC = "G3";    # Bank = 3, Signal name = SW4
6 NET "sw<3>" LOC = "B4";    # Bank = 3, Signal name = SW3
7 NET "sw<2>" LOC = "K3";    # Bank = 3, Signal name = SW2
8 NET "sw<1>" LOC = "L3";    # Bank = 3, Signal name = SW1
9 NET "sw<0>" LOC = "P11";   # Bank = 2, Signal name = SW0
```

Bedingte Signalzuweisungen

With-Select:

```
1 with a select b <=
2   "1000" when "00",
3   "0100" when "01",
4   "0010" when "10",
5   "0001" when "11",
6   "0000" when others;
```

- ▶ others-Fall nötig, da STD_LOGIC 9 Werte hat

Bedingte Signalzuweisungen

When-Else:

```
1 b <= "1000" when a = "00" else
2   "0100" when a = "01" else
3   "0010" when a = "10" else
4   "0001" when a = "11" else
5   "0000";
```

Logische Operatoren

```
1 a <= NOT b;  
2 a <= b AND c;  
3 a <= b NAND c;  
4 a <= b OR c;  
5 a <= b NOR c;  
6 a <= b XOR c;  
7 a <= b XNOR c;
```

- ▶ AND, OR, XOR bei 2 und mehr Operatoren
- ▶ NAND, NOR, XNOR bei genau 2 Operatoren

Arithmetische Operatoren

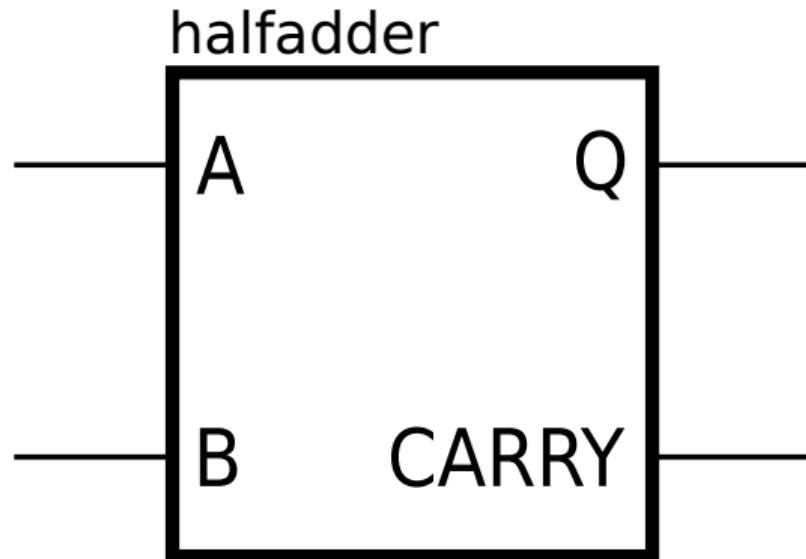
```
1 signal a, b, c unsigned ()  
2  
3 a <= b > c; -- a of type STD_LOGIC  
4 a <= b + c; -- a of type STD_LOGIC_VECTOR  
5 a <= b - c;  
6 a <= b * c;
```

- ▶ Bei +, -:
Länge von a entspricht dem längeren Vektor aus b oder c
- ▶ Bei *:
Länge von a entspricht Summe der Länge aus b oder c

Übung 02

Halbaddierer:

- ▶ $q = a \oplus b$
- ▶ $c = a \wedge b$
- ▶ Eingänge: Schalter
- ▶ Ausgänge: LED



Übung 02

```
1 entity halfadder is
2 Port ( A, B : in STD_LOGIC;
3         Q, C : out STD_LOGIC );
4 end halfadder;
5
6 architecture Structural of halfadder is
7 begin
8     Q <= A xor B;
9     C <= A and B;
10 end Structural;
```

Strukturierter Entwurf

Überblick

- ▶ Innerhalb einer vhd-Datei
 - ▶ Prozesse
 - ▶ Funktionen
- ▶ Mehrere vhd-Dateien übergreifend
 - ▶ Komponenten

Prozesse

- ▶ Verpacken einer Aufgabe in einen Prozess
- ▶ Quasi-sequentielle Ausführung
- ▶ *Erinnerung:* Signal wird am Ende des Prozesses auf den neusten Wert geschrieben
- ▶ Neue Funktionen:
 - ▶ If-Statement
 - ▶ Case-Statement
 - ▶ For-Schleife

```
1 and_proc: process (a, b)
2     -- Deklarationen
3     -- (hier nicht noetig)
4 begin
5     q <= a and b;
6 end process and_proc;
```

If-Statement

```
1 if (a = "00") and (Y = 9) then
2     b <= "1000";
3 elsif a <= "10" then
4     b <= "0100";
5 elsif a > "10" then
6     b <= "0010";
7 else
8     b <= "0000";
9 end if;
```

Case-Statement

```
1 case a is
2     when "00" => b <= "1000";
3     when "01" => b <= "0100";
4     when "10" => b <= "0010";
5     when "11" => b <= "0001";
6     when others => b <= "0000";
7 end case;
```

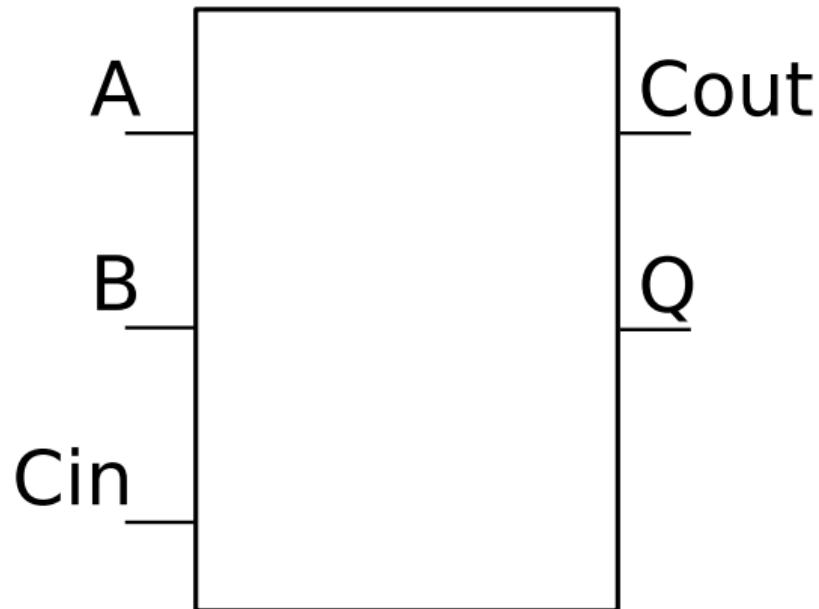
For-Schleife

```
1 Z <= "0000";
2 for I in 0 to 3 loop
3     if (A = I) then
4         Z(I) <= '1';
5     end if;
6 end loop;
```

Übung 03

Volladdierer:

- ▶ $q = (a \oplus b) \oplus c_{in}$
- ▶ c_{out} ist 1, wenn mindestens 2 Eingänge 1 sind
- ▶ Jeweils eigenen Prozess für q und c_{out}



Übung 03

```
1 entity fulladder is
2     Port ( A, B, CIN : in STD_LOGIC;
3             Q, COUT      : out STD_LOGIC);
4 end fulladder;
5
6 architecture Behavioral of fulladder is
7 begin
8
9     Q_proc : process( A, B, CIN )
10    begin
11        Q <= (A xor B) xor CIN;
12    end process Q_proc;
```

Übung 03

```
1 C_proc : process( A, B, CIN )
2 begin
3     if( A = '1' and B = '1' ) then
4         COUT <= '1';
5     elsif( A = '1' and CIN = '1' ) then
6         COUT <= '1';
7     elsif( B = '1' and CIN = '1' ) then
8         COUT <= '1';
9     else
10        COUT <= '0';
11    end if;
12 end process C_proc;
13 end Behavioral;
```

Komponenten

- ▶ Einbinden anderer Entities als Komponente
- ▶ Komponente muss an Signale, Ein- und Ausgänge angeschlossen werden
- ▶ *Black Box*-Prinzip
- ▶ Deklaration in Architecture-Header
- ▶ Instanziierung in Architecture-Body

Komponenten

Initialisierung (im Architecture Header):

```
1 component and_comp
2 -- Ein- und Ausgaenge
3 port(
4     a : in STD_LOGIC;
5     b : in STD_LOGIC;
6     q : out STD_LOGIC );
7 end component;
```

Komponenten

Instantiierung (im Architecture Body):

```
1 and_0: and_comp
2     port map ( Eingang_1 , Eingang_2 , Ausgang );
```

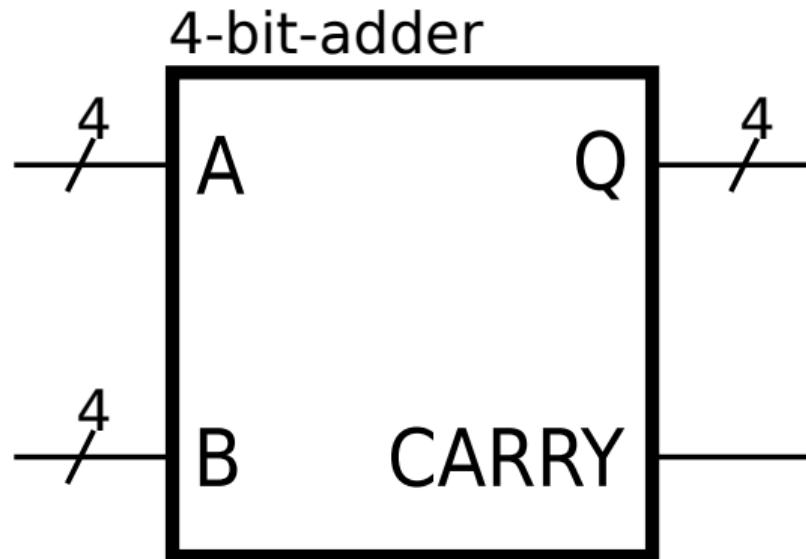
oder:

```
1 and_0: and_comp
2 port map (
3     a => Eingang_1 ,
4     b => Eingang_2 ,
5     q => Ausgang );
```

Übung 04

4-Bit-Addierer:

- ▶ 4 Volladdierer hintereinander
- ▶ Aus Komponenten
(Volladdierer)
- ▶ Eingänge: Schalter
- ▶ Ausgänge: LEDs



Übung 04

```
1 entity four_bit_adder is
2     Port ( A : in  STD_LOGIC_VECTOR (3 downto 0);
3             B : in  STD_LOGIC_VECTOR (3 downto 0);
4             Q : out STD_LOGIC_VECTOR (3 downto 0);
5             C : out STD_LOGIC);
6 end four_bit_adder;
```

Übung 04

```
1 architecture Behavioral of four_bit_adder is
2     component fulladder
3         Port ( A      : in  STD_LOGIC;
4                  B      : in  STD_LOGIC;
5                  CIN   : in  STD_LOGIC;
6                  Q      : out STD_LOGIC;
7                  COUT  : out STD_LOGIC);
8     end component;
9
10    signal sig_c0 : STD_LOGIC := '0';
11    signal sig_c1 : STD_LOGIC := '0';
12    signal sig_c2 : STD_LOGIC := '0';
13 begin
```

Übung 04

```
1 fulladder_0 : fulladder
2     port map(    A => A(0),
3                  B => B(0),
4                  CIN => '0',
5                  Q => Q(0),
6                  COUT => sig_c0 );
7 fulladder_1 : fulladder
8     port map(    A => A(1),
9                  B => B(1),
10                 CIN => sig_c0,
11                 Q => Q(1),
12                 COUT => sig_c1 );
```

Strukturierter Entwurf

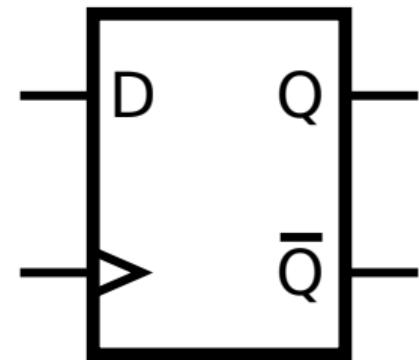
Übung 04

```
1 fulladder_2 : fulladder
2     port map(    A => A(2),
3                  B => B(2),
4                  CIN => sig_c1,
5                  Q => Q(2),
6                  COUT => sig_c2 );
7 fulladder_3 : fulladder
8     port map(    A => A(3),
9                  B => B(3),
10                 CIN => sig_c2,
11                 Q => Q(3),
12                 COUT => C );
13 end Behavioral;
```

Sequentielle Logik

Definition

- ▶ Speichern von Zuständen
- ▶ Realisierung durch Flip-Flops
- ▶ Nötige zusätzliche Eingänge:
 - ▶ CLK
 - ▶ Reset



Sequentielle Logik

Flankensteuerung

Realisierung mit asynchronem Reset:

```
1 if( RST = '1' ) then
2     -- alles zuruecksetzen
3 elsif( rising_edge(CLK) )
4     then
5         -- Logik ausfuehren
6 else
7     -- Nichts tun
8 end if;
```

Nur eine Flanke pro Prozess!

Empfohlen:

```
1 if( rising_edge(CLK) )
2 if( falling_edge(CLK) )
```

Veraltet:

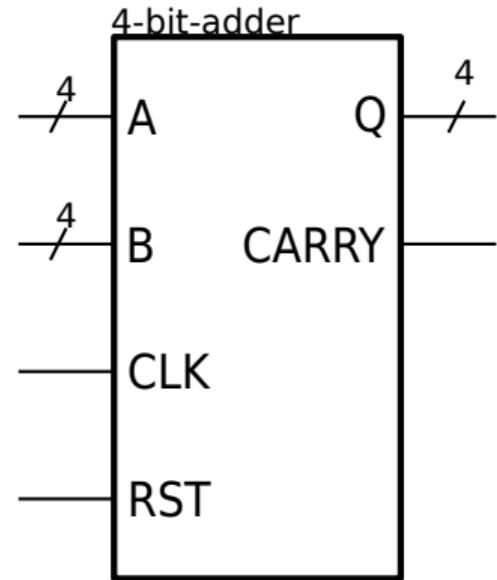
```
1 if( CLK'Event and CLK='1' )
2 if( CLK'Event and CLK='0' )
```

Rising_Edge beachtet 9-wertige Logik

Übung 05

Synchroner 4-Bit-Addierer:

- ▶ 4-Bit-Addierer (Übung 04) erweitern
- ▶ Ausgang soll taktsynchron sein
- ▶ Positiver, asynchroner Reset
- ▶ Takt durch Button



Sequentielle Logik

Übung 05

```
1 -- Architecture Header
2 signal sig_Q : STD_LOGIC_VECTOR( 3 downto 0 ) := (others=>'0');
3 signal sig_C : STD_LOGIC := '0';
4
5 -- Beispiel fuer Instanziierung
6 fulladder_3 : fulladder
7 port map(
8     A => A(3),
9     B => B(3),
10    CIN => sig_c2,
11    Q => sig_Q(3),
12    COUT => sig_C );
```

Sequentielle Logik

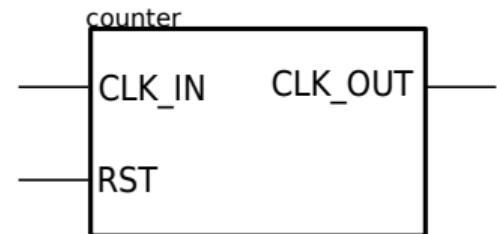
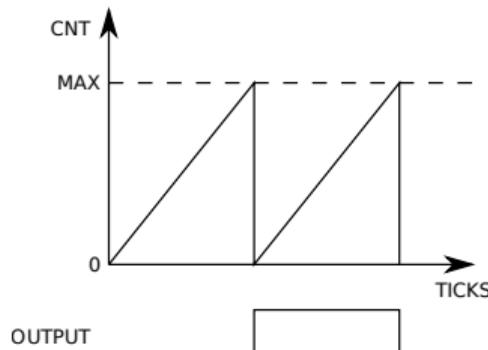
Übung 05

```
1 -- Hinter Instanziierungen
2 takt : process( CLK, RST, sig_Q, sig_C )
3 begin
4     if( RST = '1' ) then
5         Q <= (others=>'0');
6         C <= '0';
7     elsif( rising_edge(CLK) ) then
8         Q <= sig_Q;
9         C <= sig_C;
10    end if;
11 end process takt;
```

Übung 06a

Taktteiler:

- ▶ Generics
 - ▶ FREQ_IN als Integer
 - ▶ FREQ_OUT als Integer



Sequentielle Logik

Übung 060

```
1 entity counter is
2     Generic (
3         FREQ_IN : INTEGER := 50000000;
4         FREQ_OUT : INTEGER := 5 );
5     Port (
6         CLK_IN : in STD_LOGIC;
7         RST : in STD_LOGIC;
8         CLK_OUT : out STD_LOGIC);
9 end counter;
```

Sequentielle Logik

Übung 06

```
1 architecture Behavioral of counter is
2     constant MAX : UNSIGNED(31 downto 0) :=
3         to_unsigned((FREQ_IN/FREQ_OUT/2)-1, 32);
4     signal sig_CNT : UNSIGNED(31 downto 0) := (others=>'0');
5     signal sig_CLK : STD_LOGIC := '0';
6 begin
7
8     out_proc: process( sig_CLK )
9     begin
10        CLK_OUT <= sig_CLK;
11    end process out_proc;
```

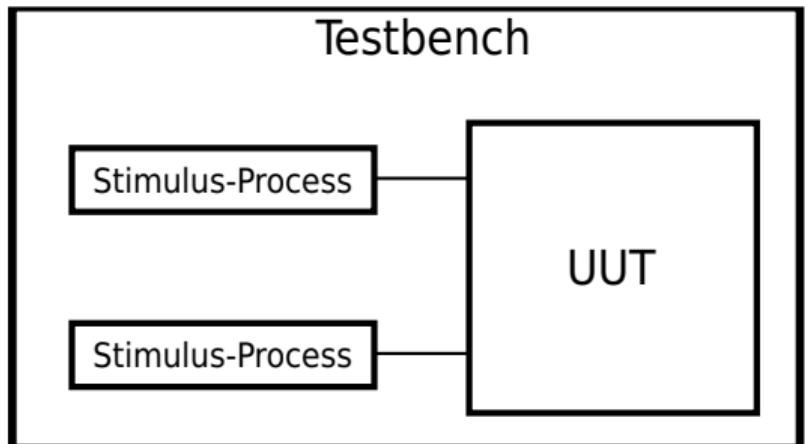
Sequentielle Logik

```
1  cnt_proc: process( CLK_IN, RST )  
2  begin  
3      if( RST = '1' ) then  
4          sig_CNT <= (others => '0');  
5          sig_CLK <= '0';  
6      elsif( rising_edge( CLK_IN ) ) then  
7          if( sig_CNT = MAX ) then  
8              sig_CNT <= (others => '0');  
9              sig_CLK <= not sig_CLK;  
10         else  
11             sig_CNT <= sig_CNT + 1;  
12         end if;  
13     end if;  
14 end process cnt_proc;
```

Simulation

Überblick

- ▶ Einzelne Module testen (Unit Under Test)
 - ▶ Fehlersuche
 - ▶ Kollaboration
- ▶ UUT wird durch Testbench stimuliert
- ▶ Ausgänge können kontrolliert werden



Testbench erzeugen

- ▶ Von Xilinx ISE automatisch erzeugt
- ▶ Entity ohne Ports
- ▶ Einbinden der UUT als Komponente
- ▶ Prozesse zur Ansteuerung der Eingänge

```
1 ENTITY any_tb IS
2 END any_tb;
3
4 ARCHITECTURE behavior OF any_tb IS
5 -- Komponenten Deklaration der UUT
6 -- Signale
7 BEGIN
8
9 -- Instanziierung der UUT
10 -- Takt-Stimulus-Prozess
11 -- Andere Stimuli-Prozesse
12
13 END;
```

Stimuli-Prozesse

Prozess wiederholt sich außer es steht ein `wait` am Ende.

```
1 wait for 10 ns;
```

Wartet die angegebene Zeit

```
1 wait;
```

Wartet unendlich lange

```
1 clk_process : process
2 begin
3     CLK_IN <= '0';
4     wait for CLK_IN_period/2;
5     CLK_IN <= '1';
6     wait for CLK_IN_period/2;
7     CLK_IN <= '0';
8     wait;
9 end process clk_process;
```

Stimuli-Prozesse

```
1 G0 <= '1' after 5 ns,  
2           '0' after 10 ns;
```

Schaltet nach 5 ns auf 1 und nach 5 weiteren (insgesamt 10) auf 0.

Weitere Wait-Funktionen existieren

```
1 AB: process  
2 begin  
3     A <= '1' after 5 ns,  
4             '0' after 10 ns,  
5             '1' after 15 ns,  
6             '0' after 20 ns;  
7     B <= '1' after 5 ns,  
8             '0' after 15 ns;  
9     wait;  
10 end process AB;
```

Assertion-Prozesse

```
1 assert condition
  2   report string
  3   severity severity_level;
```

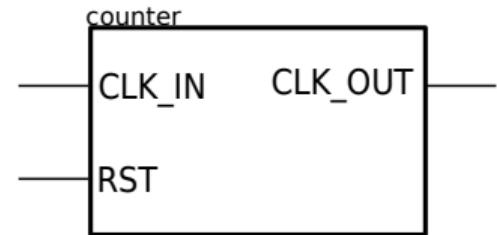
- ▶ Wenn Kondition nicht zutrifft kann Fehler ausgegeben werden
- ▶ 4 severity levels

Severity	Erklärung
NOTE	Nachrichten
WARNING	Leichte Fehler
ERROR	Fehler, führt zu Abbruch der Simulation
FAILURE	Schwerer Fehler, führt zu Abbruch

Übung 06b

Taktteiler simulieren:

- ▶ Ein Prozess für Clock ($T=10\text{ns}$)
- ▶ Ein Prozess für Reset
 - ▶ 5 CLK-Cycles High
 - ▶ 50 CLK-Cycles Low
 - ▶ High



Zustandsautomaten

Übersicht

- ▶ Medwedew-Automat
 - Ausgabe ist Zustand
 - $Out = State$
- ▶ Moore-Automat
 - Ausgabe abhängig von Zustand
 - $Out = f(State)$
- ▶ Mealy-Automat
 - Ausgabe abhängig von Zustand und Eingabe
 - $Out = f(State, In)$

Zustandsvariable

- ▶ Eigenen Typ erstellen
- ▶ Zustände sinnvoll benennen

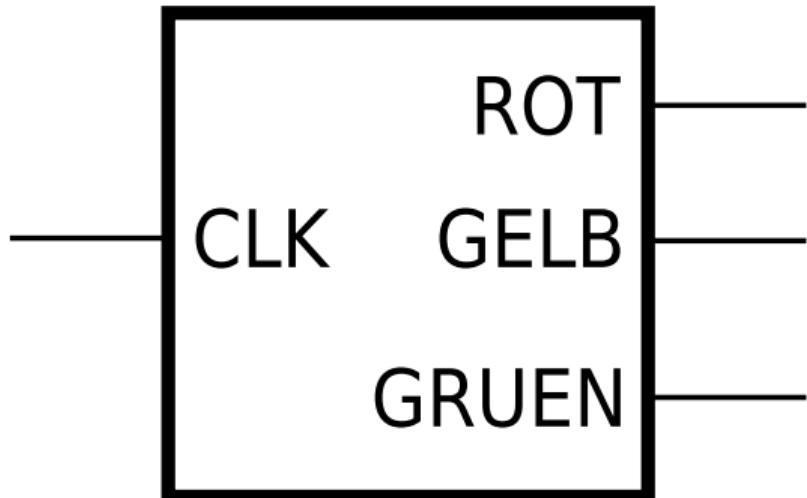
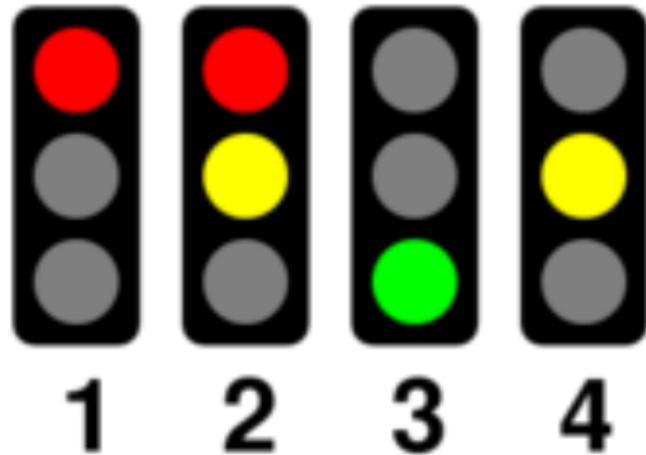
```
1 TYPE <typ_name> IS <liste>;
```

Im Architecture Header

```
1 TYPE state_type IS (rot, rotgelb,  
                      gruen, gelb);  
2  
3 signal state : state_type := rot;
```

Übung 07

Ampelschaltung:

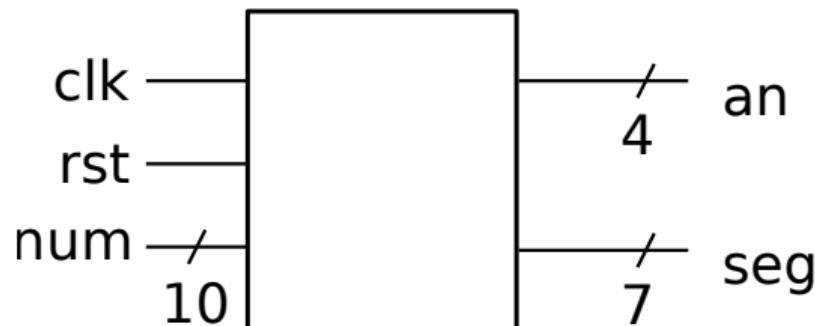


Anwendungen

7-Segment-Anzeige

7-Segment-Anzeige:

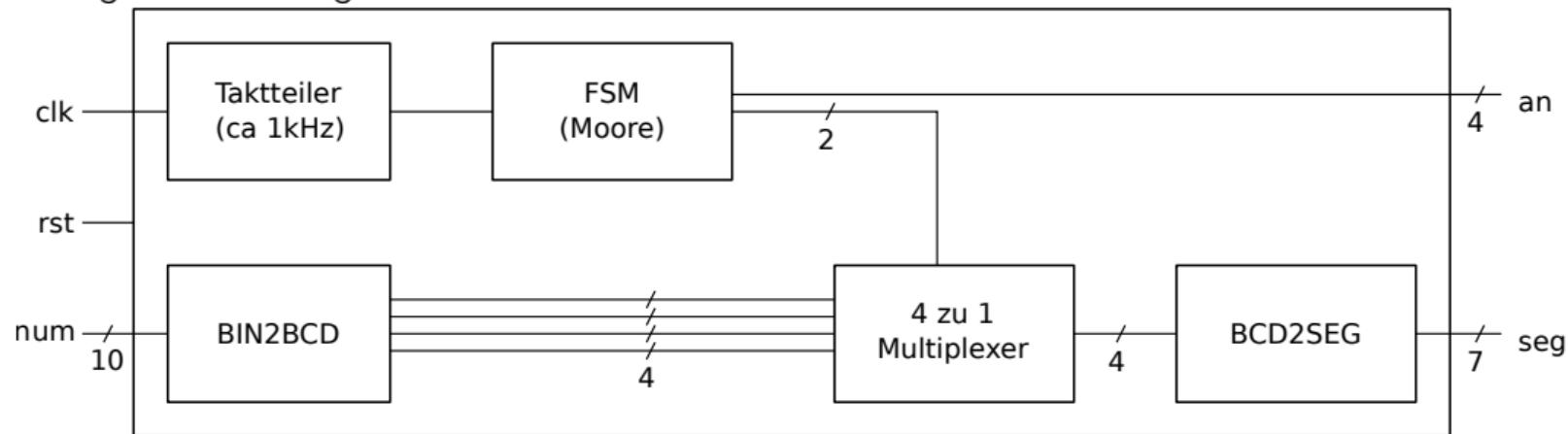
- ▶ 10-Bit breiter Eingang
- ▶ Ausgabe auf 7-Segment-Anzeige
 - ▶ Multiplexing (Einzelne Zahlen sehr schnell nacheinander)
- ▶ Hierarchischer Entwurf
- ▶ Testbenches stehen bereit



Anwendungen

7-Segment-Anzeige

7-Segment-Anzeige:



Hardware-Multiplizierer

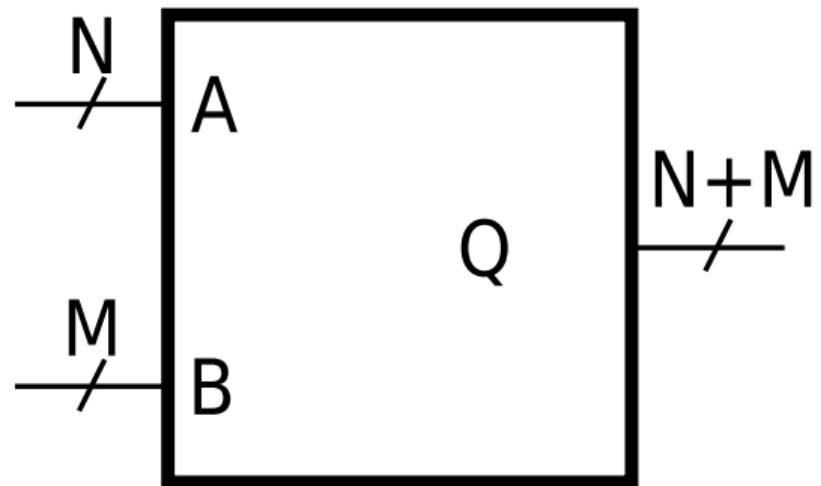
- ▶ Angelehnt an Schriftliche Multiplikation
- ▶ Faktoren A und B
- ▶ Wenn Bit an n-ter Stelle von B gesetzt ist, addiere A um n nach links verschoben zum Ergebnis

1	1	0	*	1	0	1	6 * 5
				1	1	0	n = 0
				0	0	0	n = 1
				1	1	0	n = 2
0	0	1	1	1	1	0	Summe: 30

Anwendungen

Hardware-Multiplizierer

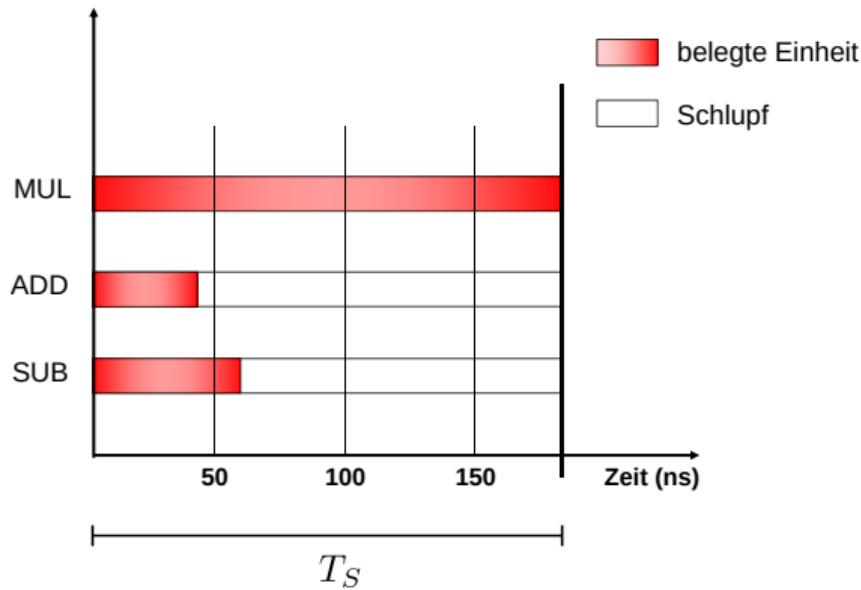
- ▶ Generics für Breite von A, B
- ▶ $N = M = 4$



Anwendungen

Pipelining

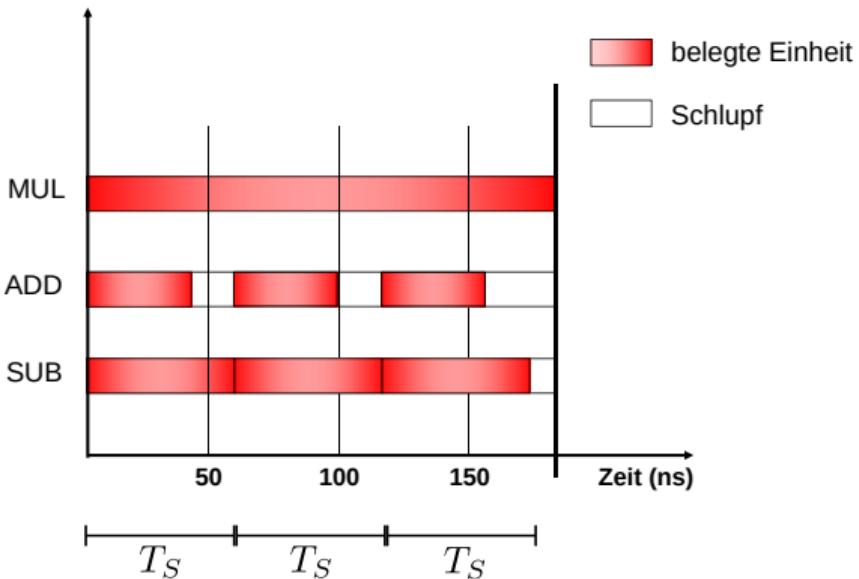
- ▶ Einfache Berechnungen sind schneller als komplexe
- ▶ Schnellster Takt wird durch langsamste Berechnung bestimmt



Anwendungen

Pipelining

- ▶ Idee:
 - ▶ Berechnung auf mehrere Takte aufteilen
 - ▶ Taktschlupf minimieren



Pipelining

VHDL Array

```
1 type <name> is array (0 to n) of <typ> (m downto 0);
```

Beispiel Array aus 8 STD_LOGIC_VECTOR der Länge 16

```
1 type mul_array is array (n to 7) of std_logic_vector (15 downto 0);
```

Pipelined HW-Multiplizierer

- ▶ Benötigte Werte
 - ▶ Zwischenwert mit 0 initialisiert
 - ▶ Faktoren A und B
 - ▶ Valid-Bit um gewollte Multiplikation zu Symbolisieren

ZW	A	B	VALID
0	0	0	0
0	0	0	0
0	0	0	0

Pipelined HW-Multiplizierer

- ▶ Faktoren einsetzen
- ▶ Valid-Bit setzen
- ▶ Zwischenergebnis für 0-ten Schritt berechnen

ZW	F1	F2	VAL
110	110	101	1
0	0	0	0
0	0	0	0

Pipelined HW-Multiplizierer

- ▶ Faktoren in nächste Stufe schieben, neue Faktoren einsetzen
- ▶ Valid-Bit schieben und neues setzen
- ▶ Zwischenergebnis für 0-ten und 1-ten Schritt berechnen

ZW	F1	F2	VAL
0	011	110	1
110	110	101	1
0	0	0	0

Pipelined HW-Multiplizierer

- ▶ Faktoren in nächste Stufe schieben
- ▶ Valid-Bits schieben
- ▶ Zwischenergebnis für 1-ten Schritt berechnen
- ▶ Endergebnis für 2-ten Schritt berechnen

ZW	F1	F2	VAL
0	0	0	0
0110	011	110	1
11110	110	101	1

Pipelined HW-Multiplizierer

- ▶ Faktoren in letzte Stufe schieben
- ▶ Valid-Bit schieben
- ▶ Endergebnis für 2-ten Schritt berechnen

ZW	F1	F2	VAL
0	0	0	0
0	0	0	0
10010	011	110	1

Pipelined HW-Multiplizierer

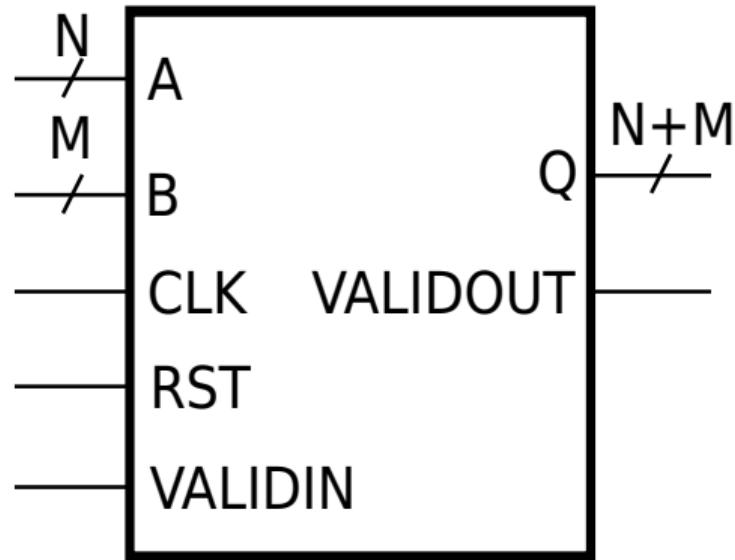
Fazit:

- ▶ Eingänge in Schritt 0 speichern
- ▶ Immer von Schritt n nach n+1 arbeiten
(Schritt entspricht dem Speicherbereich)
- ▶ Zwischenergebnis im letzten Schritt entspricht dem Endergebnis
- ▶ Berechnung immer durchführen
(egal ob valid oder nicht)

Anwendungen

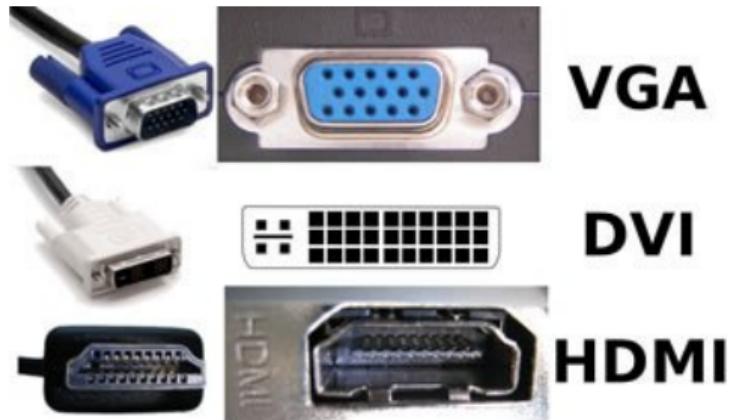
Pipelined HW-Multiplizierer

- ▶ Hardware Multiplizierer erweitern
- ▶ Testbench liegt vor
- ▶ Eingänge in 0
- ▶ Von n nach n+1
- ▶ Höchstes n in Ausgänge



VGA-Ausgabe

- ▶ Analoge Schnittstelle für Bildübertragung
- ▶ VGA: Video Graphics Array
- ▶ 15-poliger Mini-Sub-D Stecker
- ▶ Veralteter Standard (heute: DVI, HDMI, DisplayPort)

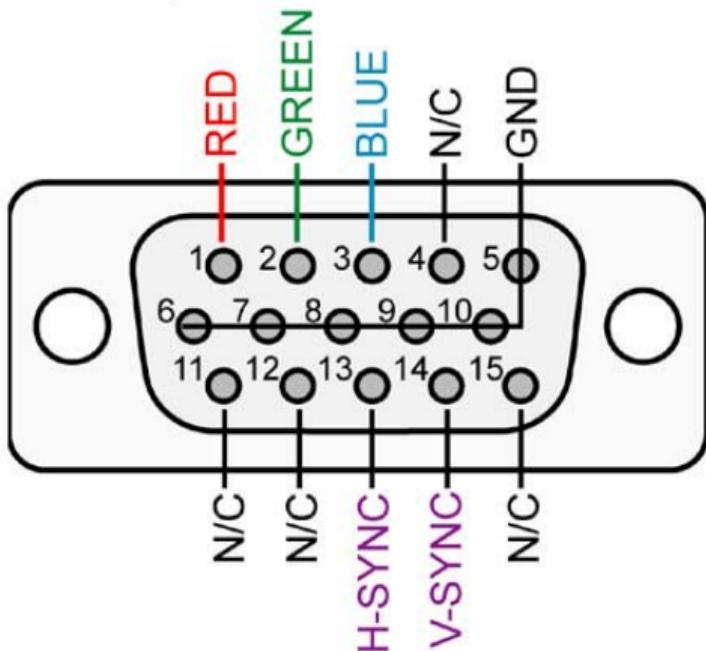


Anwendungen

VGA-Ausgabe

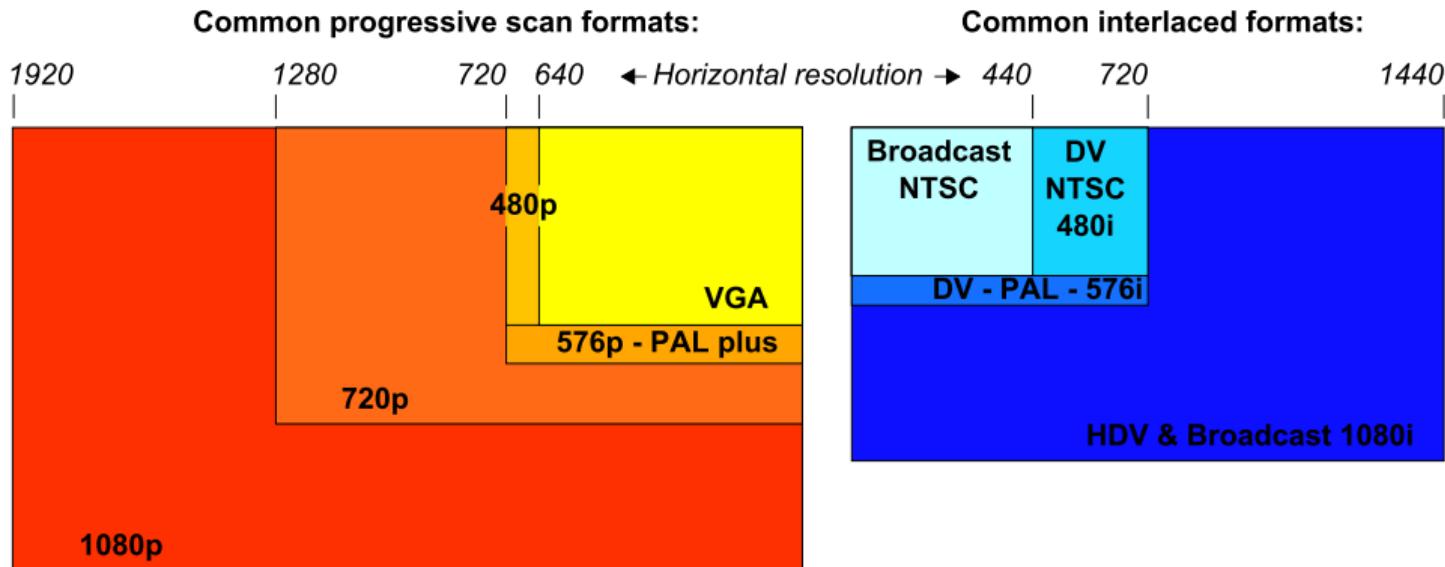
- ▶ Analoge Signale für Rot, Grün und Blau (0V-0,7V)
- ▶ Digitale Signale für HSYNC und VSYNC

VGA port, view from Wire Side



Anwendungen

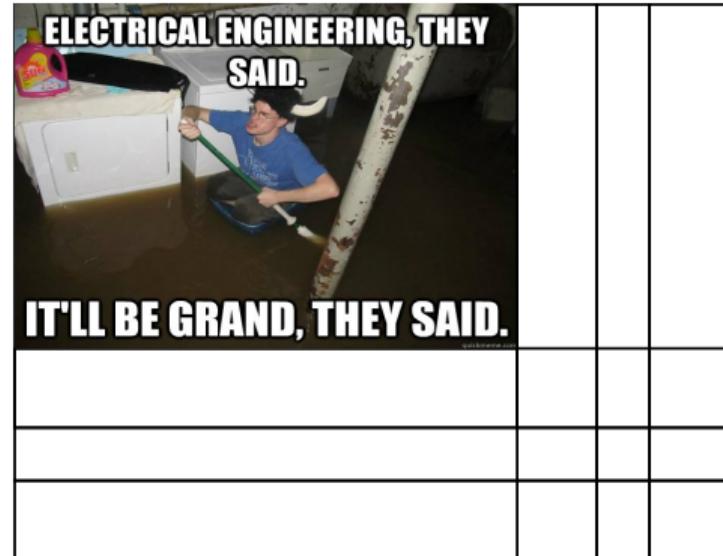
VGA-Ausgabe



VGA-Ausgabe

Steuersignale

- ▶ Bereiche außerhalb des Bildbereichs
- ▶ Vertikale und Horizontale Begrenzung

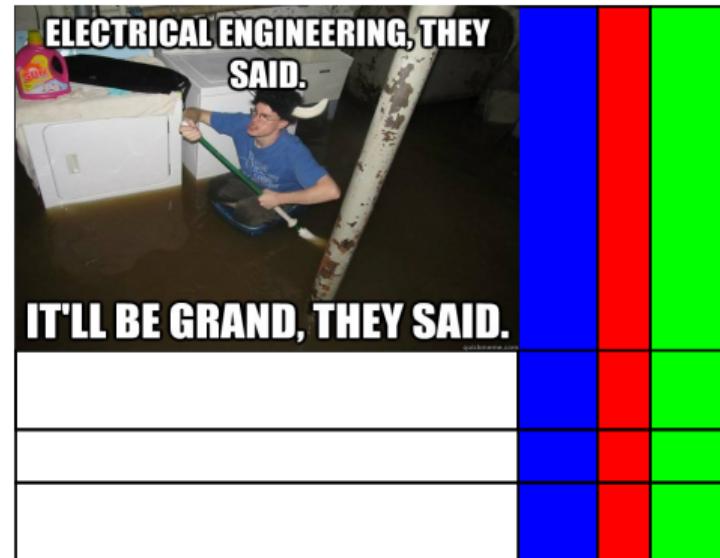


Anwendungen

VGA-Ausgabe

Steuersignale

- ▶ Horizontal Front Porch
- ▶ Horizontal Retrace (Sync)
- ▶ Horizontal Back Porch

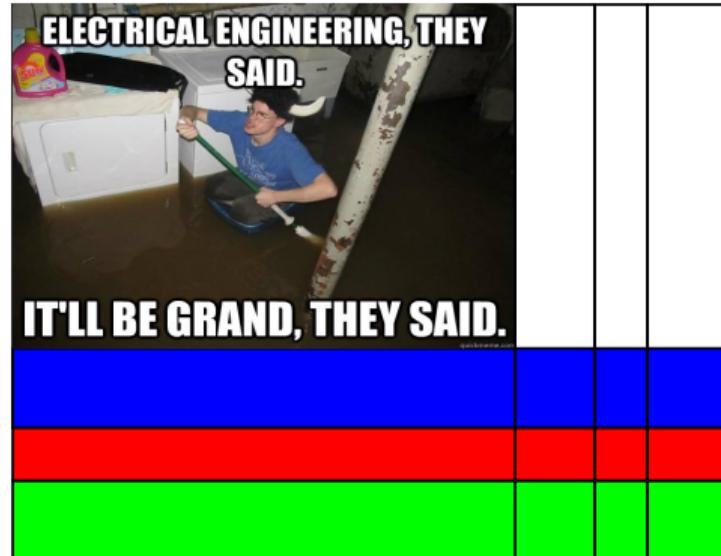


Anwendungen

VGA-Ausgabe

Steuersignale

- ▶ Vertical Front Porch
- ▶ Vertical Retrace (Sync)
- ▶ Vertical Back Porch

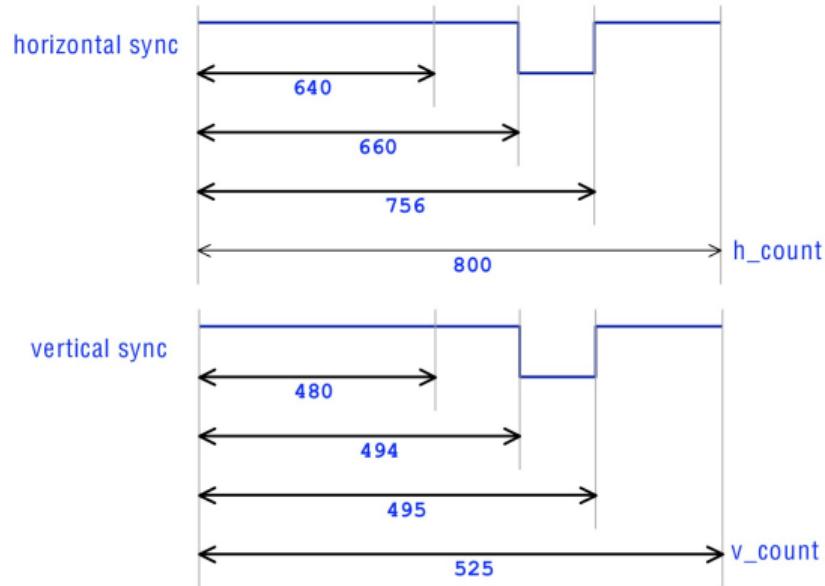


Anwendungen

VGA-Ausgabe

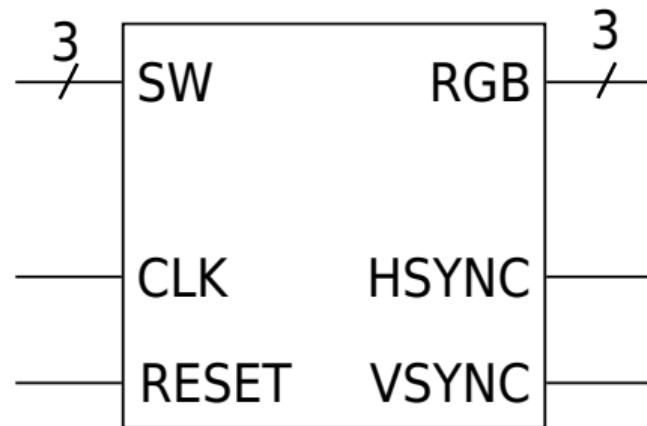
Steuersignale

Horizontal Display Area	640
Horizontal Front Porch	16
Horizontal Retrace	96
Horizontal Back Porch	48
Vertical Display Area	480
Vertical Front Porch	10
Vertical Retrace	2
Vertical Back Porch	33



VGA-Ausgabe

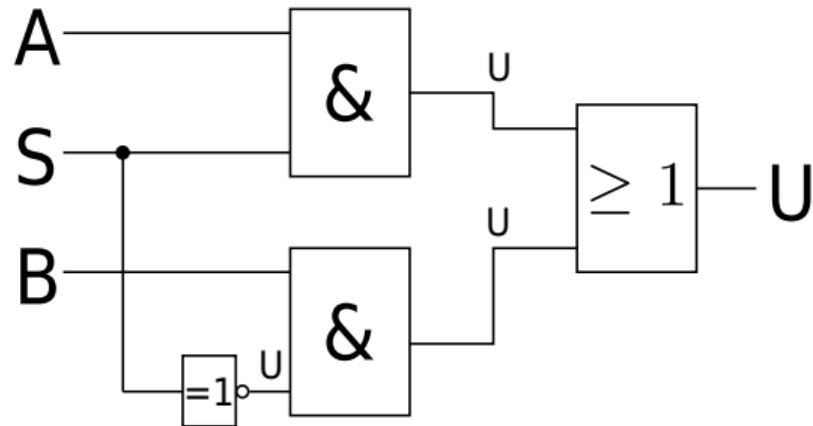
Übung: VGA mit Schalter



Anwendungen

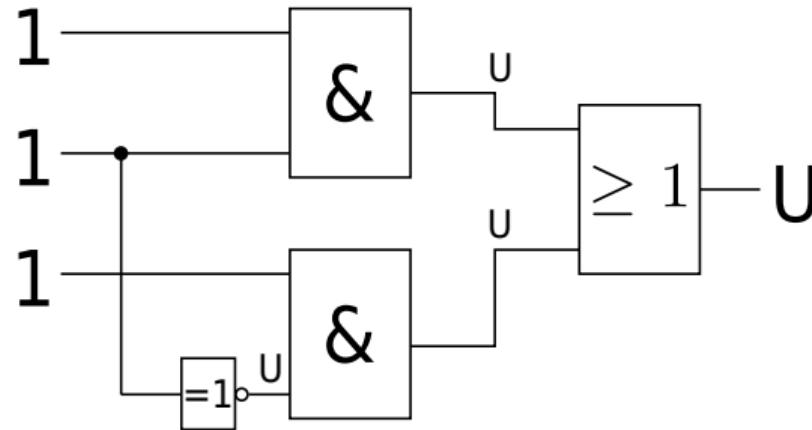
Hazards

- ▶ Probleme kombinatorischer Logik
- ▶ Unerwartete Zustände



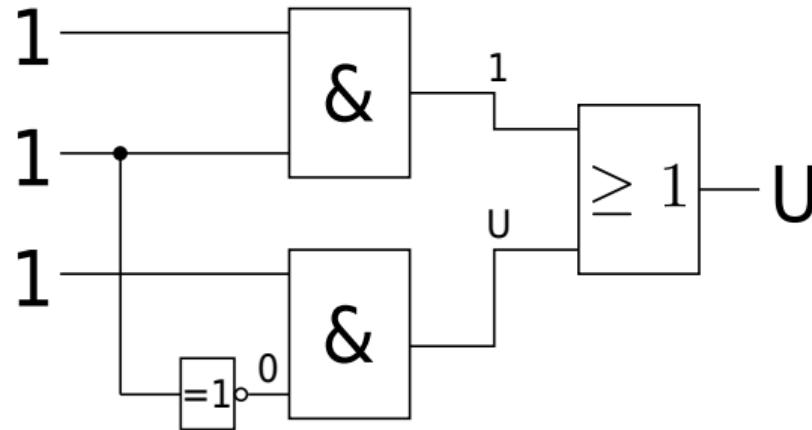
Hazards

- ▶ Probleme kombinatorischer Logik
- ▶ Unerwartete Zustände



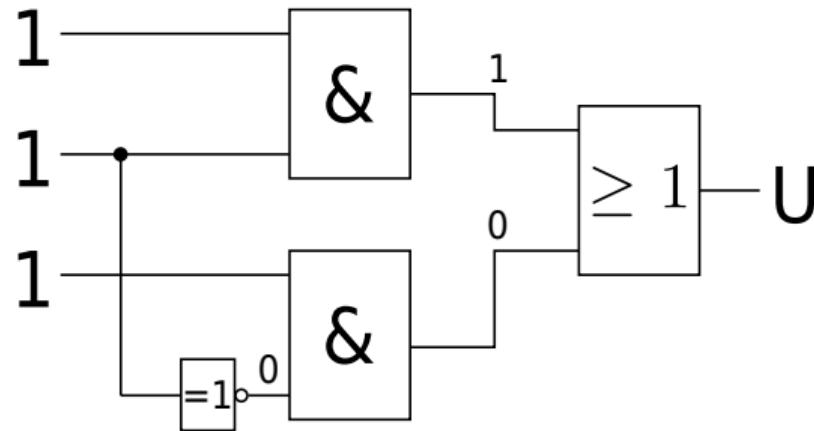
Hazards

- ▶ Probleme kombinatorischer Logik
- ▶ Unerwartete Zustände



Hazards

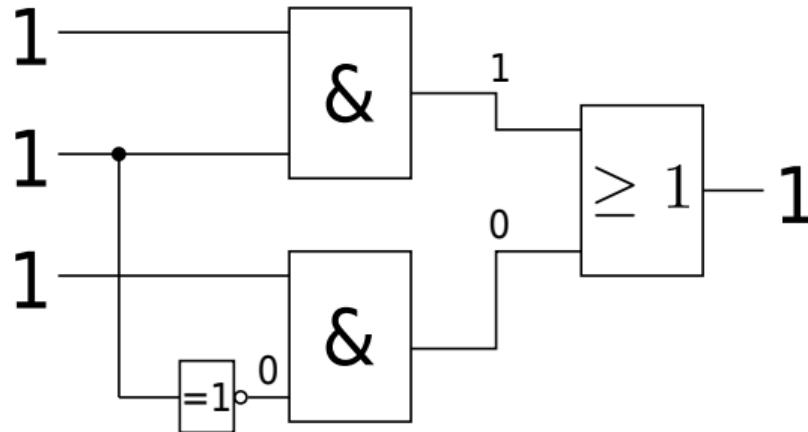
- ▶ Probleme kombinatorischer Logik
- ▶ Unerwartete Zustände



Anwendungen

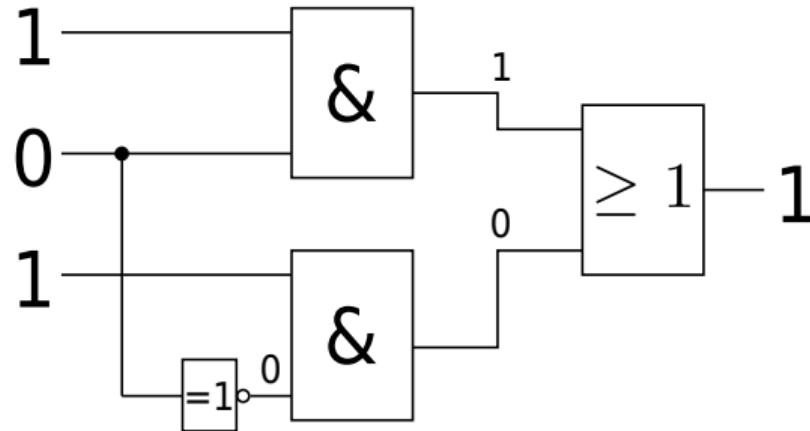
Hazards

- ▶ Probleme kombinatorischer Logik
- ▶ Unerwartete Zustände



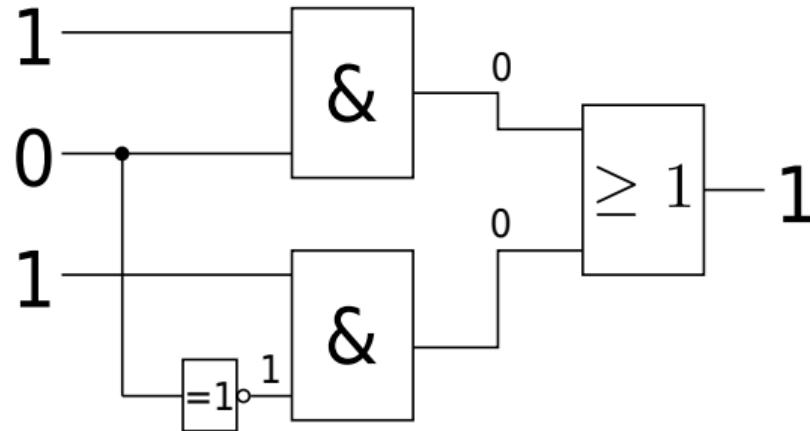
Hazards

- ▶ Probleme kombinatorischer Logik
- ▶ Unerwartete Zustände



Hazards

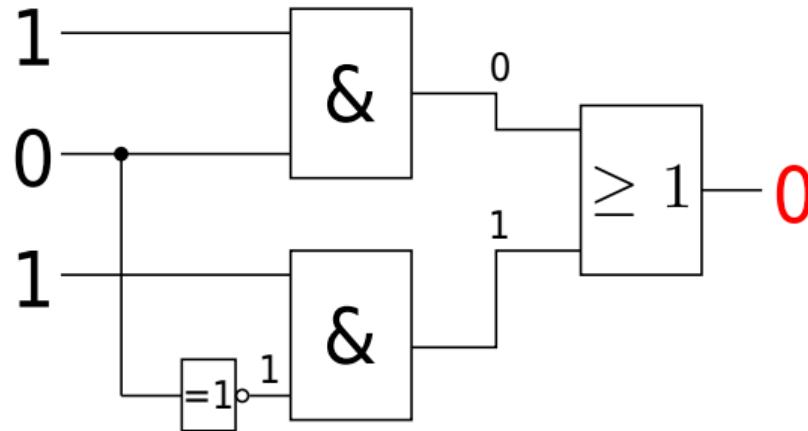
- ▶ Probleme kombinatorischer Logik
- ▶ Unerwartete Zustände



Anwendungen

Hazards

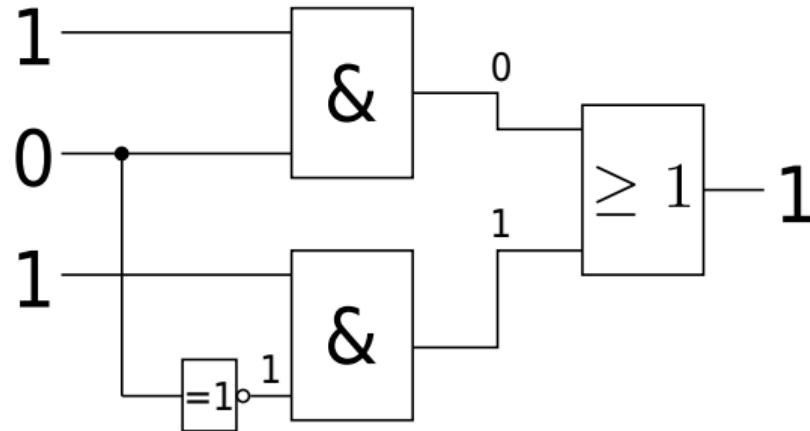
- ▶ Probleme kombinatorischer Logik
- ▶ Unerwartete Zustände



Anwendungen

Hazards

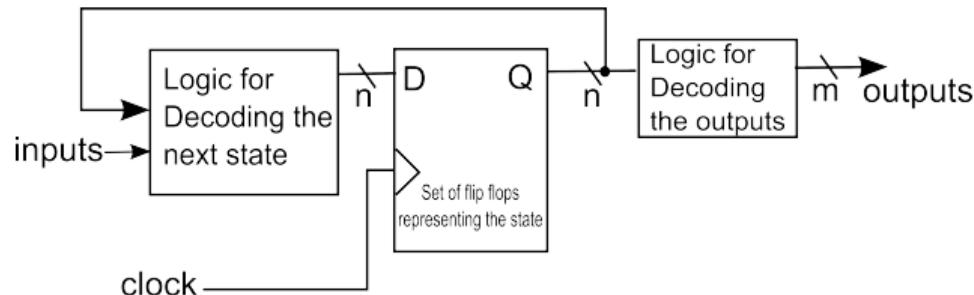
- ▶ Probleme kombinatorischer Logik
- ▶ Unerwartete Zustände



Hazards

- ▶ Lösung: Einführung von NEXT und REG Signalen
 - ▶ REG:
 - Enthält aktuellen Zustand
 - wird bei steigender Flanke durch NEXT-Wert ersetzt
 - ▶ NEXT:
 - Enthält Berechnungsergebnis des nächsten Zustands

Hazards



State-Machine die Hazards am Ausgang erzeugen kann

- ▶ Problem: Hazards auch am Ausgang möglich
- ▶ Lösung: NEXT- und REG-Signal auch für Ausgänge

Übung: VGA_SCHALTER ändern



Übung: VGA_SCHALTER ändern

