

# Einführung in Computer Microsystems

## Sommersemester 2015

Teil 0: Einleitung

Teil 2: Ausgewählte Themen zum Rechnerentwurf



TECHNISCHE  
UNIVERSITÄT  
DARMSTADT





# Organisatorisches

## Dozent

- ▶ Andreas Koch
- ▶ Sprechstunde: In der Regel Mi 14-15 Uhr, E101
- ▶ [koch@esa.cs.tu-darmstadt.de](mailto:koch@esa.cs.tu-darmstadt.de)

## Wissenschaftlicher Mitarbeiter

- ▶ Jaco Hofmann
- ▶ Sprechstunde: In der Regel Mi 13-14 Uhr, E106
- ▶ [jah@esa.cs.tu-darmstadt.de](mailto:jah@esa.cs.tu-darmstadt.de)

- ▶ Diesmal **Übergangsvorlesung** zum neu reakkreditierten Studiengang
- ▶ Besteht im wesentlichen aus drei Teilen
  - ▶ 1. Teil: Einführung in Hardware-Beschreibungssprache Bluespec
  - ▶ 2. Teil: Reststoff aus TGDI (demnächst in VL Rechnerorganisation)
  - ▶ 3. Teil: Architektur und Entwurf von Rechnersystemen
- ▶ Kaum noch Wiederholung aus TGDI
  - ▶ TGDI (inkl. Verilog!) ist **Voraussetzung** für Kanonik CMS
- ▶ Falls Kenntnisse fehlen bzw. vertieft werden sollen
  - ▶ Grundlagen Hardware-Design: TGDI-Aufzeichnungen und Buch
  - ▶ Verilog: TGDI-Aufzeichnungen und Buch, **alte** CMS 2012-Aufzeichnungen



## Grundlagen TGDI + Reststoff

D.M. Harris und S.L. Harris: *Digital Design and Computer Architecture*, 2. Auflage, MKP, 2012

## Erster Teil der Vorlesung

- ▶ Bluespec Online-Dokumentation der RBG unter `/usr/local/bluespec/doc/BSV`
- ▶ Insbesondere Einführungsbuch *Bluespec by Example*

## Dritter Teil der Vorlesung

Michael J. Flynn und Wayne Luk: *Computer System Design – System-on-Chip*, Wiley, 2011

## Form: Integrierte Veranstaltung

- ▶ Mittwoch, 8:55-11:20 Uhr, C 205
- ▶ Parallele Übertragung (solange nötig) nach C110 und C120
- ▶ In der Regel wird die Vorlesung aufgezeichnet werden

## Aufteilung

- ▶ 3x 45min Blöcke: 135 Minuten, aber mit Pause
- ▶ Anfangs nur Vorlesungen: 8:55-10:00, 10:10-11:20
- ▶ Regelbetrieb ist Tafelübung (8:55-9:40) und Vorlesung (9:50-11:20)
- ▶ Gelegentlich reine Tafelübungen zur Vorstellung größerer Beispiele

## Übungsaufgaben

- ▶ Freiwillige Aufgaben (auch keine Bonuspunkte!)
- ▶ Werden in Tafelübungen besprochen



- ▶ Klausur
- ▶ Derzeit geplant am 21.07.2015, 12-14 Uhr
- ▶ 90 Minuten reine Schreibzeit

## Verilog-Simulator und Signalverlaufs-Anzeige

- ▶ Open Source: Icarus Verilog `iverilog` und `gtkwave`
  - ▶ Auch in RBG Pool installiert
- ▶ Kostenlos, z.B.
  - ▶ Xilinx WebPack <http://www.xilinx.com>
  - ▶ Altera Quartus Web Edition <http://www.altera.com>

## Bluespec-Compiler und Simulator

- ▶ Kommerzielles Werkzeug, **nicht** frei verfügbar
  - ▶ Auch keine “Studentenversion”
- ▶ 200 Lizenzen im RBG Pool installiert
  - ▶ Über SSH auch von außen zugänglich
  - ▶ Auch (bedingt) graphische Oberfläche
    - ▶ Konsolenbedienung wird aber häufig ausreichend sein





## Vorlesung benutzt das aus TGD I bekannte interaktive Quizsystem

- ▶ Antworten via Browser
- ▶ Smartphone, Laptop, Tablet verbunden mit [eduroam](#) (WLAN der TU)



# Einführung in Bluespec

## **Bluespec hat einen eigenen Foliensatz!**

Danach hier weiter ...



# Entwicklung der Mikroelektronik

Am Anfang war das Relais und die Röhre...

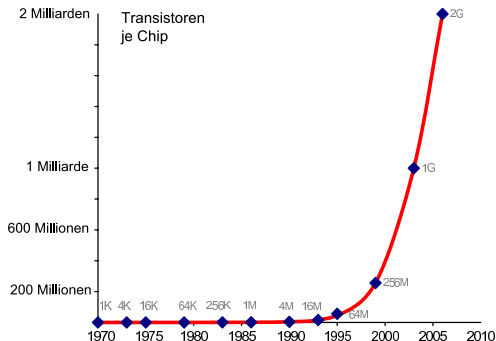
- ▶ 1947/48 Erfindung des Bipolartransistors durch Shockley, Bardeen, Braitain
- ▶ ab 1950 Grundlagenforschung auf dem Gebiet der Bipolarschaltungen
- ▶ ca. 1960 Entwicklung der Silizium-Planartechnik und erste *integrierte Schaltungen* mit ca. 10 Bauelementen
- ▶ ca. 1960 Konzeption des MOS-Transistors durch Khang und Atalla
- ▶ ca. 1970 erster Mikroprozessor
- ▶ und was waren bzw. sind die *Vorteile der integrierten Schaltungen*
  - ▶ Transistoren, Dioden und Widerstände sowie die Verbindung der Bausteine untereinander werden in einem gemeinsamen Herstellungsprozess in einem Silizium-Einkristall integriert.
  - ▶ Höhere Zuverlässigkeit, höhere Schaltgeschwindigkeiten, höhere Packungsdichte.

- ▶ Mikroelektronik umfaßt viele Gebiete:
  - ▶ Elektrotechnik: Transistorschaltungen/Differentialgleichungen
  - ▶ Physik, Verfahrenstechnik/Chemie
- ▶ Warum Mikroelektronik in der Informatik?
  - ▶ Grundverständnis kann nicht schaden
  - ▶ Es geht um Computer Microsystems
  - ▶ Zur Entwicklung von Mikroelektronik wird Software verwendet.
  - ▶ Die Anforderungen an den Entwurf (z. B. Logikminimierung, Platzierung von Bauelementen) werden mit Algorithmen gelöst.

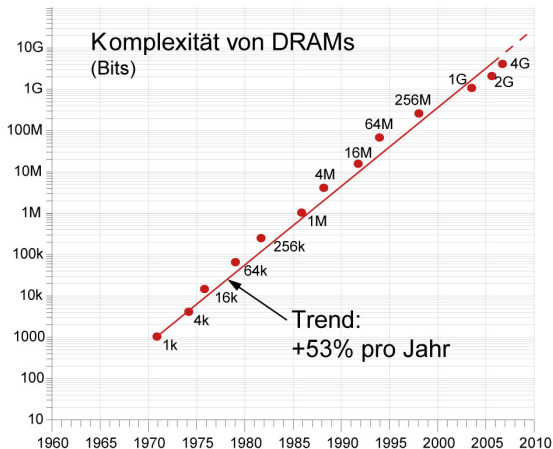
- ▶ Integrierte Schaltungen (ICs), oft auch nur als **Chips** bezeichnet.
- ▶ Chipmarkt: in 2004 Umsatz von 213 Milliarden USD weltweit  
für 2012 Umsatz von 323 Milliarden USD
- ▶ Nahezu überall verbaut
  - ▶ Offensichtlich: Rechner
    - ▶ PC, Server, Supercomputer, ...
  - ▶ Versteckt: eingebettete Systeme
    - ▶ Autos, Fernseher, Herzschrittmacher, ...
- ▶ Alleine in Deutschland: 9 Milliarden EUR Chip-Umsatz
  - ▶ Bildet aber Basis für 50x größeren Markt
  - ▶ 3 Millionen Arbeitsplätze
- ▶ Sollte man sich auch als Informatiker genauer anschauen!
- ▶ Ursprung dieser Bedeutung?

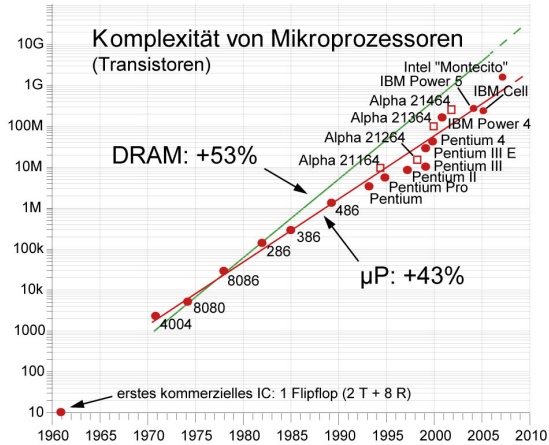
# Moore's Gesetz - Exponentielles Wachstum

Alle 18-24 Monate verdoppelt sich die Anzahl der wirtschaftlichsten Transistoren auf einem Chip. Rekordhalter 2015 NVIDIA GM200 (Titan X): 8 Milliarden Transistoren



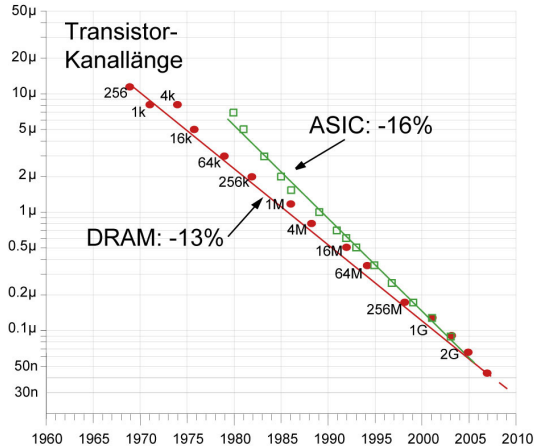






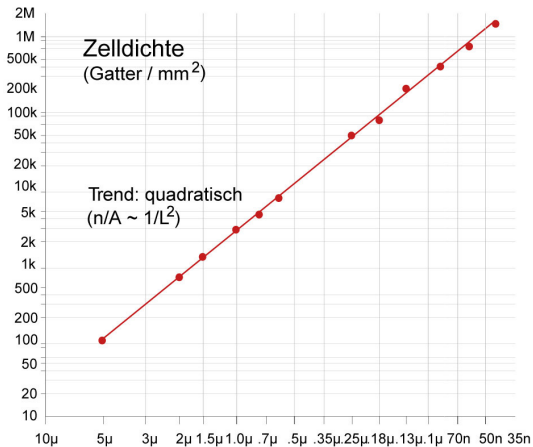
# Verbesserung der Fertigungsprozesse

## Transistor-Kanallängen



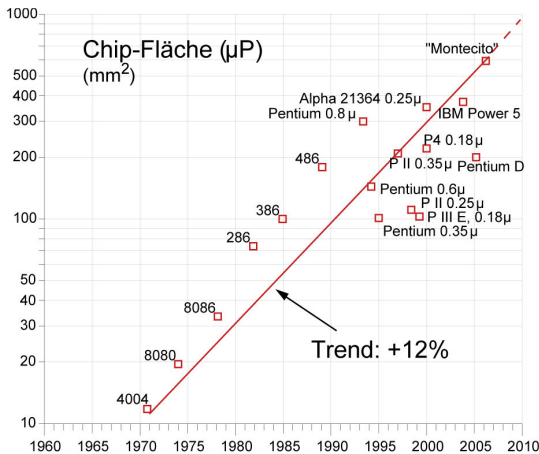
# Verbesserung der Fertigungsprozesse

## Auswirkungen - Transistoren schrumpfen um 13% jährlich



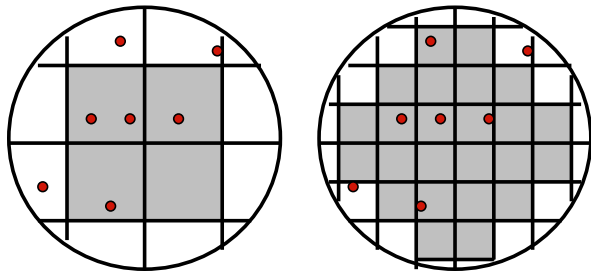
► pro Flächeneinheit  $1/0,87^2 = 33\%$  mehr Elemente

Nicht nur kleinere Strukturen, auch größere Chip-Flächen



# Ausbeute

## Effekte der Chip-Größe



Heute zuverlässig erreichbar: Nur ca. 1 Fehler pro  $\text{cm}^2$ .

# Beispiel Cell-Prozessor - Layout

Verwendet unter anderem in PlayStation 3



TECHNISCHE  
UNIVERSITÄT  
DARMSTADT

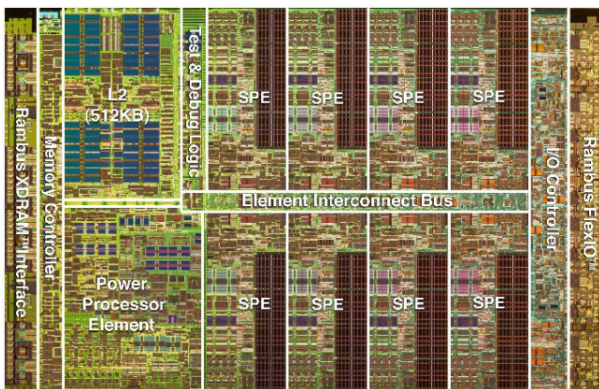
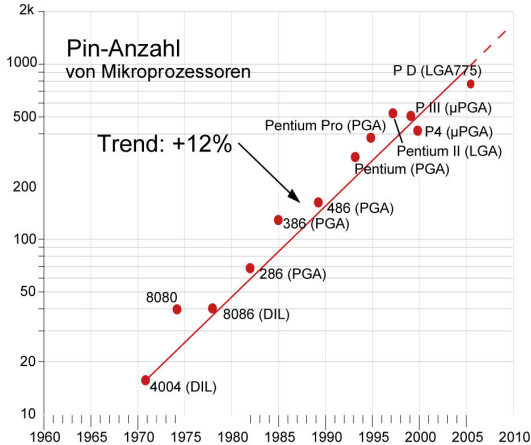


Figure: Layout (Die) des Cell-Prozessors, Quelle: IBM

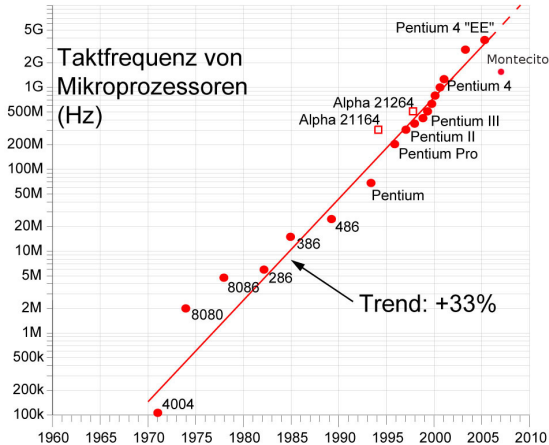
In der PlayStation 3 werden nur sechs SPEs verwendet.



Problem: Komplexität (+53% p. a.) wächst stärker als Kommunikationsmöglichkeit

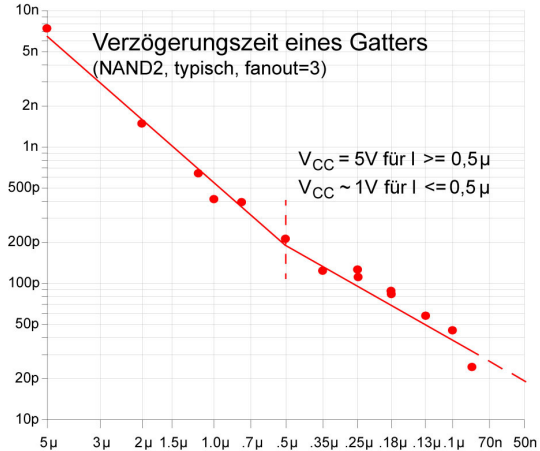


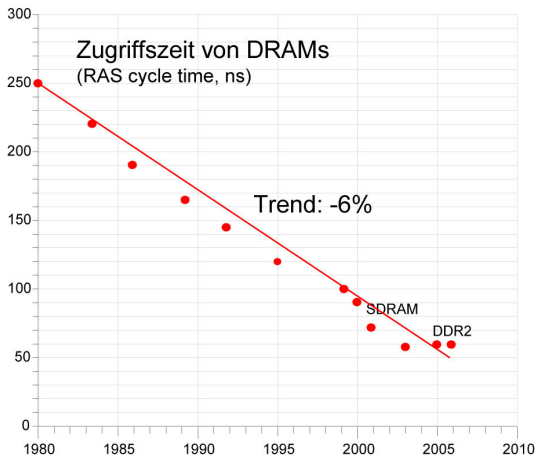
# Taktfrequenz - Entwicklung





- ▶ Leistungssteigerung wurde lange Zeit, durch erhöhen der Taktfrequenz erreicht. Aktuell liegt der Prozessortakt vieler Mikroprozessoren bei 3.x GHz.
  - ▶ Intel Pentium 4 EE, 3,8 GHz Takt, 11,5 SPECint2006
  - ▶ Intel Montecito 9050, 1,6 GHz Takt, 14,5 SPECint2006
- ▶ Bedingt durch die Technologie (CMOS-Technologie) steigt der Leistungsumsatz der Prozessoren mit dem Takt ( $P \approx U^2 \cdot f \cdot C_L$ ).
- ▶ Die entstehende Wärme ist nur mit großem Aufwand abzutransportieren.
- ▶ Parallelrechner:
  - ▶ Integration mehrerer CPUs auf einem Chip (aktuell: 2-12 CPU-Kerne von Intel und AMD verfügbar), Cell B. E. Prozessor von IBM
  - ▶ Massiv parallele Systeme mit mehreren tausend Prozessoren, Spezialarchitekturen z. B. Vektorrechner

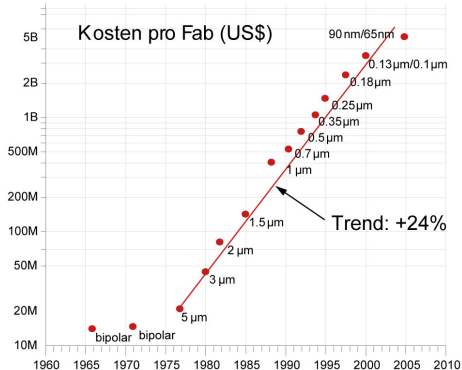




Gatterlaufzeiten liegen im ps-Bereich. Hauptspeicher ist vergleichsweise langsam.

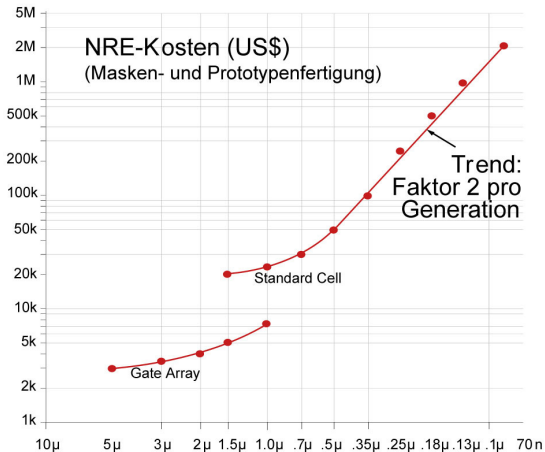


- ▶ +53% p. a. Chip-Komplexität
  - ▶ Zahl der Transistoren, Speichergröße
- ▶ +33% p. a. Packungsdichte
  - ▶ Elemente/Flächeneinheit
- ▶ +33% p. a. Taktfrequenz
- ▶ +12% p. a. Chip-Fläche
- ▶ +12% p. a. mehr Pins (Flaschenhals!)
- ▶ -6% p. a. Speicherzugriffszeit (Flaschenhals!)



- ▶ In zehn Jahren haben sich die Kosten für eine Fabrik fast verzehnfacht.
- ▶ Aktueller Stand 22nm Technologie: Ca. USD 10 Milliarden pro Fabrik

# Kostenentwicklung II

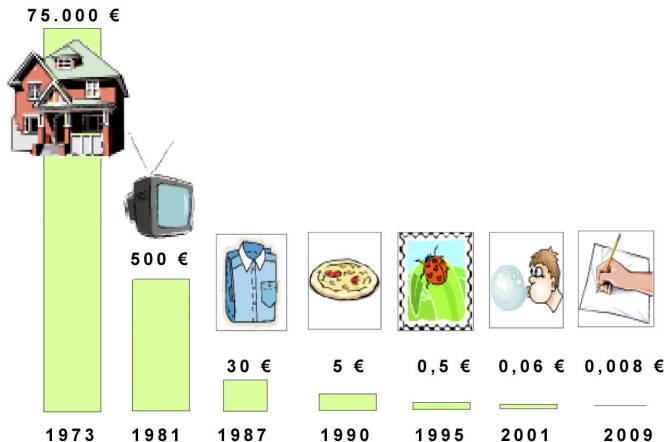


- ▶ Fertigung auf älterer Technologie
  - ▶ 180nm ... 110nm *sehr* weit verbreitet
- ▶ Multi-Projekt-Chips
  - ▶ Sonderangebote für EU Forschung und Lehre, z.B. von ST Micro
  - ▶ 130nm: EUR 2.200 / mm<sup>2</sup>
  - ▶ 65nm: EUR 7.500 / mm<sup>2</sup>
  - ▶ 28nm: EUR 12.000 / mm<sup>2</sup>
  - ▶ Für 15 Chips, Gehäuse kosten extra (EUR 30-200 je Stück)
- ▶ Programmierbare/konfigurierbare Schaltungen
  - ▶ Keine photochemische Chip-Fertigung mehr erforderlich
  - ▶ Beispiele sind: PALs, PLAs, FPGAs
  - ▶ FPGAs werden gerne zum Rapid-Prototyping eingesetzt.



# Kostensenkung durch Massenfertigung

## Kosten für 1 Mb DRAM





# Hardware-Entwurfstechniken

- ▶ Vergleichbar einem Puzzle mit einer Milliarde Teile
- ▶ Zusammensetzen unter hohem Zeitdruck
  - ▶ Time-to-Market (TTM)
- ▶ Ein einziger Fehler kann Millionen USD kosten
  - ▶ Erneute Chip-Fabrikation (“re-spin”)
  - ▶ Intel Pentium Bug (FDIV): deutlicher Gewinneinbruch

Wie bekommt man die Komplexität eines solchen Entwurfs mit 100 Millionen Transistoren (und mehr) in den Griff?

- ▶ Abstraktere Vorgehensweisen
- ▶ Beschreibe
  - ▶ ... nicht mehr einzelne Transistoren
  - ▶ ... sondern komplette Systeme
- ▶ Vergleichbar Software-Entwicklung
  - ▶ ... statt Assembler
  - ▶ ... Beschreibung von Systemen als interagierende Komponenten (service-oriented architectures)
- ▶ Mittel der Wahl: (Hardware)-Beschreibungssprachen
  - ▶ Sehr abstrakt: MATLAB/Simulink (Signalverarbeitung)
  - ▶ Abstrakt: SystemC
  - ▶ Noch recht hardware-nah: Verilog HDL - VHDL → Bluespec SystemVerilog
- ▶ Entkoppeln von Entwurf und technischer Realisierung
- ▶ Umsetzung idealerweise automatisch (Synthese)
  - ▶ Klappt aber noch nicht immer!

# Entwurfsebenen 1

## Unterschiedlicher Abstraktionsgrad

### ▶ **Verhaltensebene**

Was soll passieren? Realisierung bleibt offen.

$$y = f(x)$$

### ▶ **Systemebene**

Grobe Aufteilung von Struktur, Zeit, Daten und Kommunikation

CPU, FPGA, DRAM,

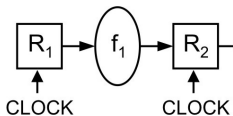
4 Busse, 32b Integer

### ▶ **Register-Transfer-Ebene**

Synchron, getaktet

always @(posedge CLOCK)

```
R2 <= f1(R1);
```

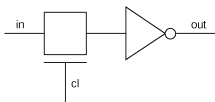


# Entwurfsebenen 2

## Unterschiedlicher Abstraktionsgrad

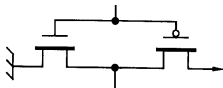
### ▶ Logik- oder Gatterebene

Netze aus Gattern, Flip-Flops, etc.



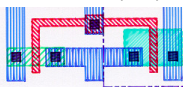
### ▶ Transistorebene

Elektrischer Schaltplan



### ▶ Layoutebene

Maßstabsgetreue geometrische Anordnung des Chips mit verschiedenen Schichten (3D)



- ▶ Verilog HDL unterstützt auch die Beschreibung des Verhaltens.

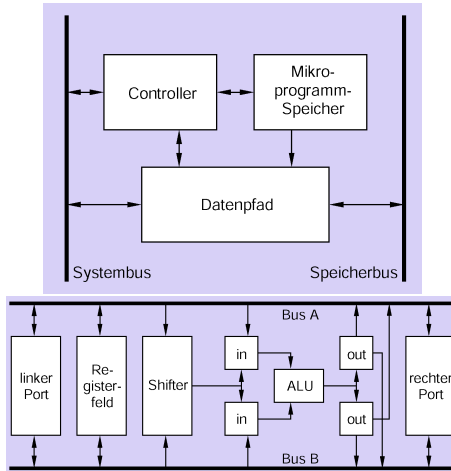
```
module MULT_SCALED2(  
  input wire [15:0] a, b,  
  output wire [32:0] prod  
);
```

```
  assign prod = a * b * 2;
```

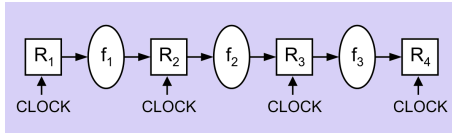
```
endmodule
```

- ▶ Keine Angaben über
  - ▶ Art des Multiplizierers (seriell, parallel, seriell/parallel)
  - ▶ Zeitverhalten

# Systemebene Komponenten eines Mikrokontrollers

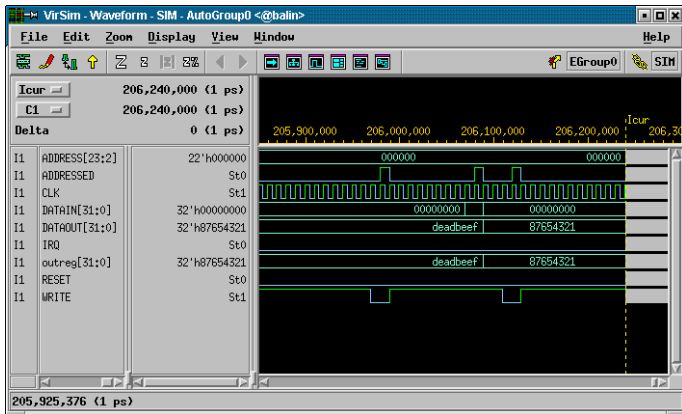




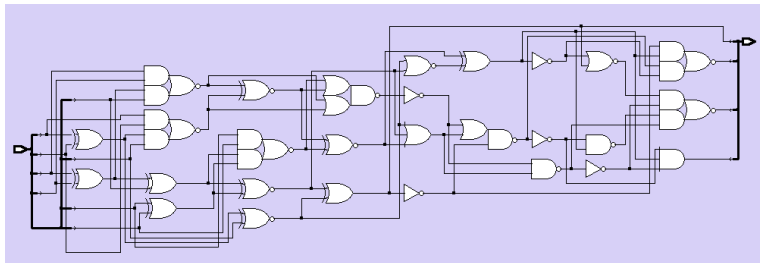


```
always @(posedge CLOCK) // mit jeder steigenden
begin // Taktflanke
    R2 <= f1(R1); // Register-Transfer von
    R3 <= f2(R2); // Ri durch fi nach Ri+1
    R4 <= f3(R3);
end
```

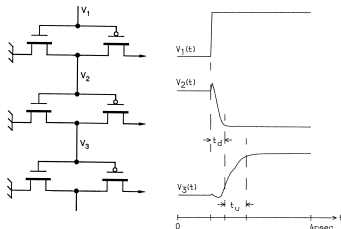
- ▶ Sehr wichtige Entwurfsebene
- ▶ Fließbandverarbeitung (Pipelines)/Automatennetze
- ▶ Effiziente Umsetzung in Hardware automatisch möglich



- ▶ Digitalsimulation noch ohne reale Verzögerungszeiten
- ▶ Alternativ auch Textausgabe möglich



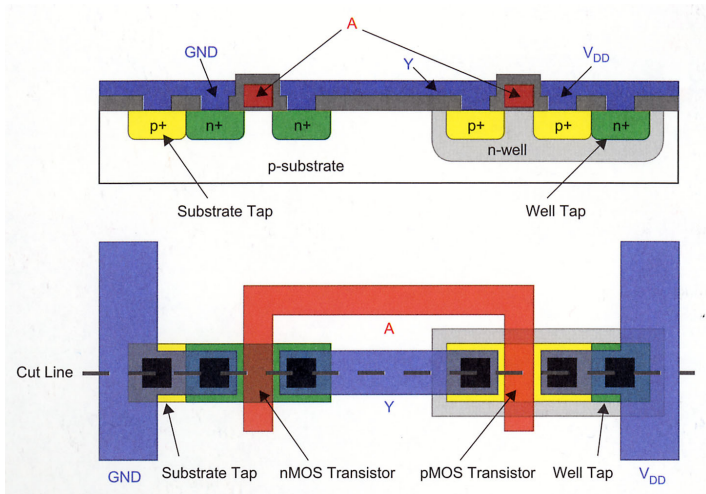
- ▶ Ergebnis der Logiksynthese
- ▶ Generische AND, NOR, Flip-Flops, etc.
- ▶ Zeitverhalten abschätzbar (aber noch sehr ungenau)
- ▶ Noch keine endgültige Hardware-Beschreibung
  - ▶ Hängt von konkreter Zieltechnologie ab
  - ▶ ASIC, FPGA, Gate-Array, ...



- ▶ Schaltpläne aus Transistoren, Widerständen, etc.
- ▶ Beim Digitalschaltungsentwurf verborgen
- ▶ Ausnahmen
  - ▶ Analoge Teilschaltungen
  - ▶ Full-Custom-Entwurf
- ▶ Analogsimulation mit Schaltzeiten

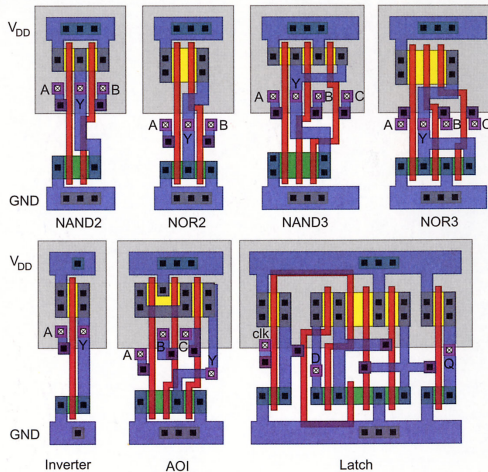
# Layout-Ebene

## Geometrien der Transistoren und Leitungen



# Layout-Ebene

## Einige Basisgatter (=Zellen)





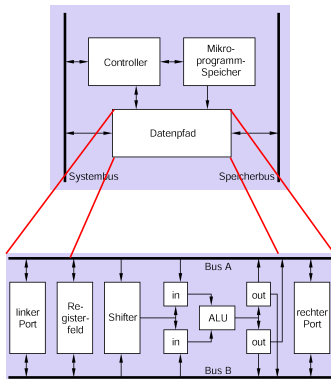
- ▶ Maßstabsgetreue Darstellung des endgültigen Chips
- ▶ Ergebnis von Platzieren und Verdrahten
  - ▶ Transistoren und Leitungen als Polygone
  - ▶ Abmessungen haben Einfluß auf elektrische Eigenschaften
  - ▶ Nun genaues Zeitverhalten bekannt
  - ▶ Weitergabe an Halbleiterhersteller (Tape-Out)

- ▶ Weglassen für die *aktuelle* Beschreibung *unwichtiger* Details
- ▶ Arbeiten auf unterschiedlichen Ebenen
  - ▶ Von ungenau bis sehr genau
  - ▶ Verhaltensebene, . . . , Layout-Ebene
- ▶ Beispiel: Entwurf eines MP3-Encoder-Chips
  - ▶ Manuell von funktionaler bis RTL-Ebene
  - ▶ Andere Schritte i. d. R. automatisch
- ▶ Beispiel: Entwurf von Empfangsteil für UMTS-Handy
  - ▶ Manuell von funktionaler bis Layout-Ebene
  - ▶ Sehr komplizierte Chips
    - ▶ Entwurf erfordert hochspezialisierte Kenntnisse.
    - ▶ Hochfrequenztechnik-Schaltungen sind schwierig zu modellieren.



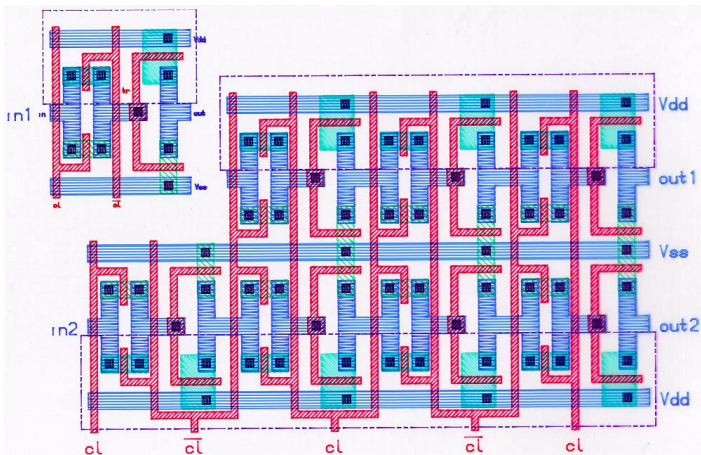
# Hierarchische Zerlegung - Aufteilen eines Problems in kleinere Unterprobleme

- ▶ Alte Idee: divide et impera (Philip II, -381 ... -335 )
- ▶ Auch rekursiv anwendbar
- ▶ Damit entsteht eine Hierarchie von Zerlegungen



# Reguläre Zerlegung

## Gezielte Vervielfältigung von Komponenten





- ▶ Top-down-Entwurf (Detaillierungsprozeß)
- ▶ Bottom-up-Entwurf (Kompositionsprozeß)
- ▶ Meet-in-the-Middle-Entwurf
- ▶ Diese Methoden existieren auch im Softwareentwurf.

- ▶ Der Schaltungsentwickler beginnt mit seinem Wissen über die zu realisierende Gesamtfunktion
- ▶ Danach Zerlegung in kleinere Teilfunktionen (Teileinheiten, Komponenten) und ein Verbindungsnetz (Verbindung der Schnittstellen durch Signale)
- ▶ Kriterien: z.B. Kosten, Geschwindigkeit oder Chipfläche
- ▶ Der Prozeß endet, wenn Basis-Funktionen verwendet werden können.



- ▶ Der Top-down-Entwurf setzt voraus, dass sich Teilsysteme zunächst abstrakt, nämlich als „black boxes“ beschreiben lassen.
- ▶ Die nach außen hin sichtbare Funktionalität der Teileinheiten muß abstrakt definierbar sein, ohne dass auf die Details ihrer internen Realisierung näher eingegangen werden muß.
- ▶ Damit ist es möglich, das Zusammenwirken von Systemkomponenten zu evaluieren bevor diese vollständig auf Logik-Ebene entworfen wurden.

- ▶ Hier wird von den verfügbaren Primitiven (z. B. TTL-Gatter) ausgegangen, welche in einer Bibliothek abgelegt sind.
- ▶ Aus den Bibliothekselementen werden komplexere Komponenten gebildet, die ihrerseits auf der nächst höheren Ebene als (elementare) Bausteine eingesetzt werden können.



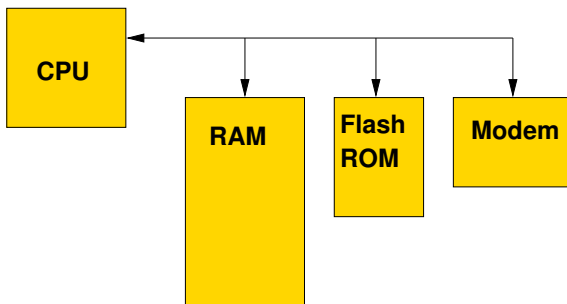
- ▶ Der Meet-in-the-middle-Entwurf vereinigt die Top-down- und die Bottom-up-Vorgehensweise
- ▶ Man beachtet beim Vorgehen von der einen Seite des Entwurfsprozesses (z. B. Top) die Auswirkungen an der anderen



# Kommunikation und Adressierung



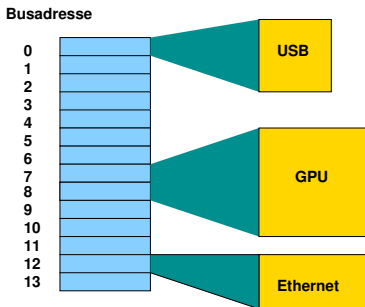
- ▶ Verschiedene Untereinheiten
- ▶ Kommunikation untereinander
- ▶ Auf verschiedene technische Weisen realisierbar



- ▶ Gemeinsame Verbindungen zwischen Komponenten
- ▶ Maximal ein **Initiator** / **Master** gleichzeitig
  - ▶ **Veranlasst** Aktivitäten auf Bus
- ▶ Ein oder mehrere **Targets** / **Slaves**
  - ▶ **Reagieren** auf Aktivitäten auf dem Bus
- ▶ Grundlegende **Transaktionen** auf dem Bus
  - ▶ Initiator fordert Daten von Target an: **Lesezugriff**
  - ▶ Initiator überträgt Daten zum Target: **Schreibzugriff**
- ▶ Busorganisation in **einfachen** Systemen
  - ▶ Nur **ein** Master, häufig die **CPU**
- ▶ In **leistungsstärkeren** Systemen
  - ▶ **Mehrere** Master (*multi-master* Ansatz, DMA)
  - ▶ Beispiele: Gb/s-Ethernet, Festplatten-Controller, Graphikprozessor, USB 2.0...
  - ▶ Hier nicht weiter behandelt

- ▶ **Gemeinsam** genutztes Medium
- ▶ Vorteile
  - ▶ **Einfache** Realisierung
  - ▶ Wenig **Chip-Fläche**
- ▶ Nachteile
  - ▶ Nur **eine** Verbindung gleichzeitig
    - ▶ Vereinfacht, kann **etwas** verbessert werden
    - ▶ → Disconnect/Reconnect, Split Transaction, ...
  - ▶ Probleme bei Bussen
    - ▶ Wie Bus vergeben, wenn **mehrere** Initiator/Master **gleichzeitig** Zugriff benötigen? → nicht weiter behandelt
    - ▶ Wie Target **gezielt** ansprechen? → kommt jetzt

- ▶ Vergabe von **Adressen** für Teilnehmer an Bus
- ▶ Können auch **sehr** große Bereiche sein
  - ▶ Beispiel: Moderne GPU braucht ca. 12 GB an **Adressraum**





- ▶ CPU **kennt schon** Adressen
- ▶ Nämlich für den **Speicher**
- ▶ Wie damit **Buszugriffe** realisieren?
- ▶ Zwei Alternativen

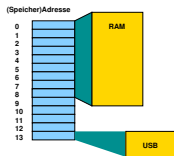
# 1. Getrennter Adressbereich für Bus



- ▶ Häufig **Ein-/Ausgabe-Adressbereich** genannt
- ▶ Verwendet z.B. bei älteren X86-Architekturen
- ▶ Spezielle **Instruktionen** für Ein-/Ausgabe
- ▶ Interpretieren angegebene Adresse immer als **Busadresse**
- ▶ Beispiel: `outb %a1, 0x60`
  - ▶ **Schreibt** Inhalt des AL-Registers (1 Byte) an Busadresse 0x60
- ▶ Beispiel: `inb 0x71, %a1`
  - ▶ **Liest** 1 Byte von Busadresse 0x71 in AL Register
- ▶ Beispiele für die klassische **PC-Architektur**
  - ▶ **Tastatur** liegt auf 0x60 und 0x64
  - ▶ **1. Serielle Schnittstelle** liegt auf 0x3f8-3ff
  - ▶ **1. Festplatten-Controller** liegt auf 0x1f0-1f7 und 0x3f6-3f7

## 2. Gemeinsamer Adressbereich

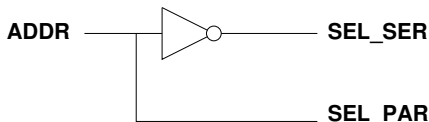
- ▶ **Nachteil** des getrennten Adressbereichs
  - ▶ **Zusätzliche** Instruktionen
  - ▶ Ungünstig, Instruktionsbits sind **rares Gut**
- ▶ Anderer Ansatz: **Blende** Busadressen in **normalen** Adressraum ein: **memory mapped I/O**
  - ▶ So bei den meisten anderen Prozessoren
  - ▶ Auch bei X86 möglich, heute überwiegend verwendet
- ▶ Zugriff nun mit ganz normalen Lade/Speicherbefehlen
  - ▶ `lw $t0, 4`: Lade 32b Wort aus **RAM-Speicher**
  - ▶ `sb $t1, 13`: Gebe Byte auf **USB-Schnittstelle** aus



- ▶ Problem: Wie erkennen Busteilnehmer, ob sie **angesprochen** werden?
  - ▶ Also Target / Slave eines Zugriffs sind
- ▶ Gängige Lösung
  - ▶ **Select**-Signal pro Teilnehmer
  - ▶ Wird **aktiviert**, wenn Teilnehmer vom Initiator/Master angesprochen wird
- ▶ **Adressdekodierung** übersetzt Busadressen in Select-Signale
- ▶ Triviales Beispiel: Zwei Teilnehmer auf Bus
  - ▶ Auf Adressen 0 und 1

## Adresse

0	
1	





# Komplizierteres Szenario

Annahme: 64KB Adressraum



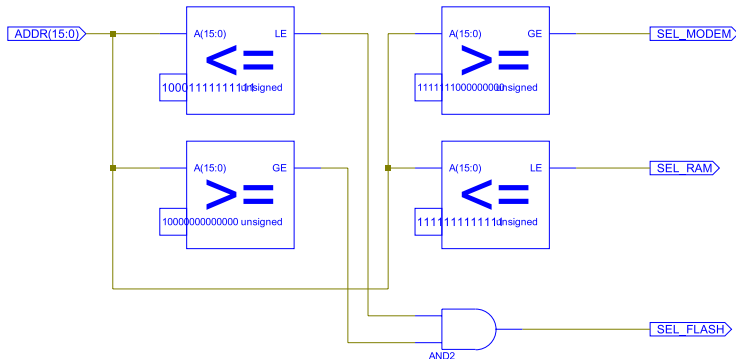
TECHNISCHE  
UNIVERSITÄT  
DARMSTADT

Startadresse	Endadresse	Teilnehmer
0x0000	0x1FFF	RAM 8KB
0x2000	0x23FF	Flash-Speicher 1KB
0xFE00	0xFFFF	Modem 512B

Wie realisierbar? Erster Versuch:

```
module decoder1 (  
  input [15:0] ADDR,  
  output      SEL_RAM, SEL_FLASH, SEL_MODEM  
);  
  
  assign SEL_RAM    = (ADDR >= 16'h0000 && ADDR <= 16'h1FFF);  
  assign SEL_FLASH  = (ADDR >= 16'h2000 && ADDR <= 16'h23FF);  
  assign SEL_MODEM  = (ADDR >= 16'hFE00 && ADDR <= 16'hFFFF);  
  
endmodule
```

# Diskussion: Syntheseresultat



- ▶ Optimiert: Immerhin nur **vier statt sechs** Vergleicher
- ▶ Jeder einzelne Vergleicher aber **groß und langsam!**



Müssen wir immer **alle** Bits vergleichen?

Startadresse	Endadresse	Teilnehmer
16'b0000_0000_0000_0000	16'b0001_1111_1111_1111	RAM 8KB
16'b0010_0000_0000_0000	16'b0010_0011_1111_1111	Flash-Speicher 1KB
16'b1111_1110_0000_0000	16'b1111_1111_1111_1111	Modem 512B

- ▶ **Nein**, versuche nur **Startadressen** eindeutig zu unterscheiden
- ▶ Mit möglichst **wenigen** Bits!
- ▶ Beginne mit **höchstwertigen** (=linken) Bits
  - ▶ Vermeide so falsche Erkennung von Adressbereichen
  - ▶ Gegenbeispiel: Verwende Bit 9, um Modem (=1) und Flash (=0) auseinanderzuhalten
  - ▶ Klappt nicht: Bei Zugriff auf RAM kann Bit 9 **auch** 1 werden!
  - ▶ Bereiche dürfen sich **nicht** überlappen
  - ▶ Niemals **gleichzeitig** mehr als ein Select-Signal aktiv

# Nächster Versuch ...

Konstruiere **Entscheidungsbaum** von links nach rechts



TECHNISCHE  
UNIVERSITÄT  
DARMSTADT

Startadresse	Endadresse	Teilnehmer
16'b0000_0000_0000_0000	16'b0001_1111_1111_1111	RAM 8KB
16'b0010_0000_0000_0000	16'b0010_0011_1111_1111	Flash-Speicher 1KB
16'b1111_1110_0000_0000	16'b1111_1111_1111_1111	Modem 512B

- ▶  $ADDR[15]==1 \rightarrow$  Zugriff auf **Modem**
- ▶  $ADDR[15]==0 \ \&\& \ ADDR[13]==0 \rightarrow$  Zugriff auf **RAM**
- ▶  $ADDR[15]==0 \ \&\& \ ADDR[13]==1 \rightarrow$  Zugriff auf **Flash**

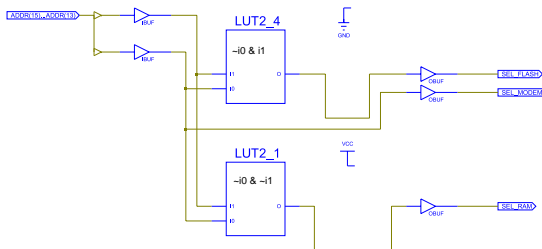
## Diskussion

- ▶ **Überlappungsfrei**
- ▶ Select-Signal wird korrekt 1 bei Zugriff auf **Startadresse** und folgende Adressen
- ▶ Aber was ist mit der **Endadresse**?  $\rightarrow$  später!

```
module decoder2 (  
  input [15:0] ADDR,  
  output      SEL_RAM, SEL_FLASH, SEL_MODEM  
);
```

```
assign SEL_RAM    = ~ADDR[15] & ~ADDR[13];  
assign SEL_FLASH  = ~ADDR[15] & ADDR[13];  
assign SEL_MODEM  = ADDR[15];
```

```
endmodule
```





```
interface Decoder2;
  method Bool isEnabledRAM    (Bit#(16) addr);
  method Bool isEnabledFLASH  (Bit#(16) addr);
  method Bool isEnabledMODEM  (Bit#(16) addr);
endinterface
```

```
module mkDecoder (Decoder2);
```

```
  method Bool isEnabledRAM (Bit#(16) addr);
    return (addr[15] == 0 && addr[13] == 0);
  endmethod
```

```
  method Bool isEnabledFLASH (Bit#(16) addr);
    return (addr[15] == 0 && addr[13] == 1);
  endmethod
```

```
  method Bool isEnabledMODEM (Bit#(16) addr);
    return (addr[15] == 1);
  endmethod
```

```
endmodule
```

```
module top (Empty);
```

```
  Decoder2 d    <- mkDecoder;
  Reg#(Bit#(16)) a <- mkReg(0);
```

```
  rule isRAM if (d.isEnabledRAM(a));
    $display("RAM_a=%4x",a);
  endrule
```

```
  rule isFlash if (d.isEnabledFLASH(a));
    $display("Flash_a=%4x",a);
  endrule
```

```
  rule isModem if (d.isEnabledMODEM(a));
    $display("Modem_a=%4x",a);
  endrule
```

```
  rule genaddress;
    a <= a + 8192;
    if (a[15:14] == 1)
      $finish;
  endrule
endmodule
```

# Dekodierung **innerhalb** eines Teilnehmers

## Einfaches Beispiel 1KB Flash-ROM, organisiert als 1Kx8b



```
module rom1kx8 (  
  input      SELECT,  
  input [9:0] ADDR,  
  output [7:0] DATA  
);  
  
  reg [7:0] MEM [0:1023]  
  
  assign DATA = (SELECT) ? MEM[ADDR] : 8'bz;  
  
  initial begin          // einige Beispieldaten eintragen  
    MEM[0]   = 8'h42;  
    MEM[1]   = 8'h23;  
    ...  
    MEM[1022] = 8'h20;  
    MEM[1023] = 8'h07;  
  end  
  
endmodule
```

# Dekodierung **innerhalb** eines Teilnehmers

## Komplexeres Beispiel: Modem-Steuerregister



```
module modem (  
  input    CLOCK,  
  input    SELECT,  
  input [8:0] ADDR,  
  input    WRITE,  
  inout [7:0] DATA  
);  
  
reg [7:0] baudrate;  
reg [1:0] parity;  
reg [7:0] inchar, outchar;  
  
assign DATA = (~SELECT | WRITE) ? 8'bz :  
              ((ADDR==0) ? baudrate :  
               (ADDR==1) ? {6'b0,parity} :  
               (ADDR==2) ? inchar // <-- ADDR=2 liest Zeichen  
               : 8'h42);          // <-- Default-Wert für Debugging  
  
always @(posedge CLOCK) begin  
  if (SELECT & WRITE)  
    case (ADDR)  
      0 : baudrate <= DATA;  
      1 : parity   <= DATA[1:0];  
      2 : outchar  <= DATA;          // <-- ADDR=2 schreibt Zeichen  
    endcase  
end  
  
endmodule
```



# Anschluss an Bus

## Am Beispiel des 1KB Flash-ROMs



```
module mysystem;
...

wire [15:0] ADDR;
wire [7:0] DATA;
wire      SEL_RAM, SEL_FLASH, SEL_MODEM;

// Adressdecoder
decoder2 DECODER (ADDR, SEL_RAM, SEL_FLASH, SEL_MODEM);

// Flash-ROM
rom1kx8 FLASH (SEL_FLASH, ADDR[9:0], DATA);

...
endmodule
```

# Verhalten des Flash-ROMs



Startadresse	Endadresse	Teilnehmer
16'b0000_0000_0000_0000	16'b0001_1111_1111_1111	RAM 8KB
16'b0010_0000_0000_0000	16'b0010_0011_1111_1111	Flash-Speicher 1KB
16'b1111_1110_0000_0000	16'b1111_1111_1111_1111	Modem 512B

Zugriff auf Busadresse	Flash Select	Zugriff auf ROM-Adresse	Ausgabedatum	
16'h0000	0	10'h000	8'bz	Zugriff auf RAM-Bereich
16'hFE00	0	10'h200	8'bz	Zugriff auf Modem-Bereich
16'h2000	1	10'h000	8'h42	Zugriffe auf Flash-Bereich
16'h2001	1	10'h001	8'h23	
16'h23FE	1	10'h3FE	8'h20	
16'h23FF	1	10'h3FF	8'h07	
16'h2400	1	10'h000	8'h42	Hinter Flash-Ende!
16'h2401	1	10'h001	8'h23	
16'h27FE	1	10'h3FE	8'h20	
16'h27FF	1	10'h3FF	8'h07	
16'h2800	1	10'h000	8'h42	
16'h2801	1	10'h001	8'h23	

...

➡ Der Flash-Bereich **wiederholt** sich!



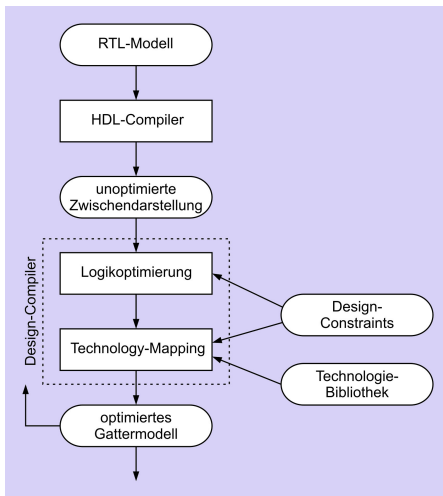
- ▶ Speicherbereich **wiederholt** sich
- ▶ Es sind aber immer die **gleichen** Daten
- ▶ Sichtbarkeit der gleichen *lokalen* Adressen an unterschiedlichen *Busadressen*: **Aliasing**
- ▶ Schadet in vielen Fällen **nicht**
  - ▶ Manchmal schon: **Hack** der Microsoft XBOX
  - ▶ Schreibzugriffe auf 80008008 werden **abgefangen**
  - ▶ Aber Fehler im **Adressdecoder** der Southbridge
  - ▶ Adressen sind auch **aliased** als 80008x08
  - ▶ Zugriffe **erlaubt** für  $x \neq 0$
  - ▶ So Zugriff auf "geheimen" **Boot-Code** möglich
  - ▶ → "MIST Premature Unmap Attack "
  - ▶ → [http://bunniefoo.com/nostarch/HackingTheXbox\\_Free.pdf](http://bunniefoo.com/nostarch/HackingTheXbox_Free.pdf)
- ▶ Erlaubt aber sehr **einfache und schnelle** Adressdekodierung

- ▶ Zeige Busteilnehmern durch **Select-Signal** an, wenn sie angesprochen werden
- ▶ Select-Signale werden aus **Busadressen** erzeugt
- ▶ Adressbereiche müssen **überlappungsfrei** sein
- ▶ Gleichzeitig darf **maximal ein** Select-Signal aktiv sein
- ▶ Erstelle Adressdekodierlogik durch Aufbau eines **Entscheidungsbaumes**
  - ▶ Von höherwertigen zu niederwertigen Adressbits
  - ▶ Erfülle Anforderungen dabei durch Auswertung von möglichst **wenigen** Adressbits
- ▶ Dabei **darf** derselbe Adressbereich i.d.R. mehrfach auftauchen (aliasing)
- ▶ Er **muß** aber mindestens an den spezifizierten Adressen erreichbar sein



# Verfeinerter Ablauf der Synthese

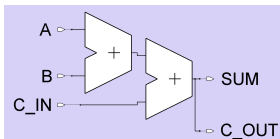
# Verfeinerter Entwurfsablauf der Synthese



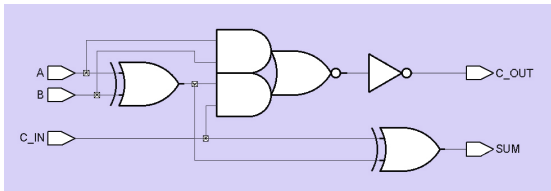
# Syntheseablauf: Technologieabbildung

## *technology mapping*

Unoptimierte Zwischendarstellung eines 1b-Addierers



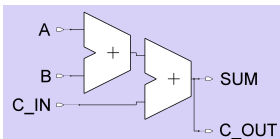
Ergebnis für ASIC-Zielbibliothek (LSI 10K)



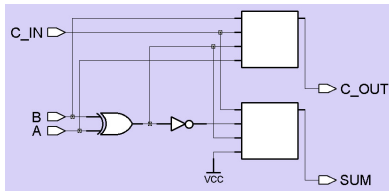
# Syntheseablauf: Technologieabbildung

## *technology mapping*

Unoptimierte Zwischendarstellung eines 1b-Addierers

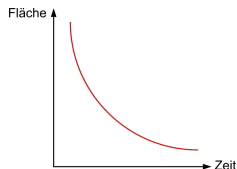


Ergebnis für FPGA (Xilinx XC4000)





- ▶ Zeit
  - ▶ Timing-Analyse
  - ▶ Geschätzt nach Synthese (ohne Verdrahtungsverzögerung)
  - ▶ Exakt nach Platzieren und Verdrahten
  - ▶ ➡ Layout-Ebene
- ▶ Fläche
  - ▶ Geschätzt nach Synthese (ohne Verdrahtungsfläche!)
  - ▶ Exakt nach Platzieren und Verdrahten
- ▶ Elektrische Leistungsaufnahme
  - ▶ Simulation auf Layout-Ebene
  - ▶ Bestimmung von Umschaltfrequenzen (*toggle frequencies*) von Signalen



# Beispiel: 4b-Vergleicher



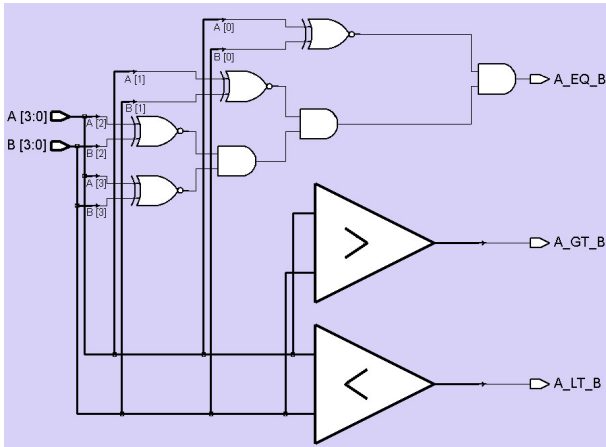
- ▶ Größenvergleich von zwei 4b breiten Eingabewerten A und B
- ▶ Bestimmt Flags für  $<$ ,  $>$ , und  $=$
- ▶ Erstes Ziel: Möglichst schnelle Schaltung, Fläche egal

```
// Vergleich
module mag_comp (
  input wire [3:0] A, B,
  output wire      A_GT_B, A_LT_B, A_EQ_B);

assign A_GT_B = (A > B);      // A groesser B
assign A_LT_B = (A < B);      // A kleiner B
assign A_EQ_B = (A == B);     // A gleich B

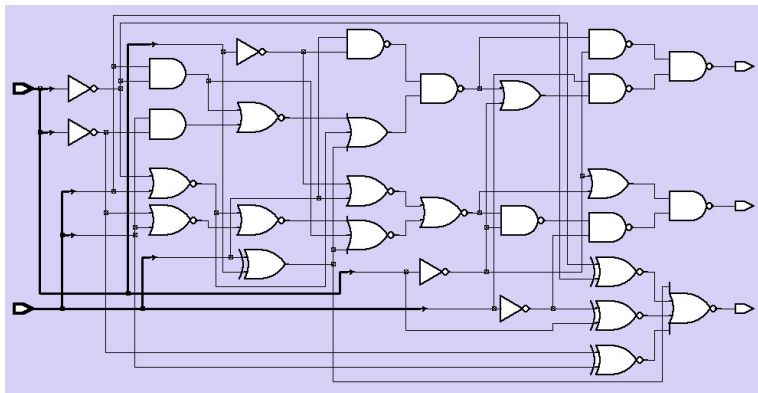
endmodule
```

# Zwischendarstellung des 4b-Vergleichers



# 4b-Vergleicher in LSI Logic 10K ASIC- Technologie

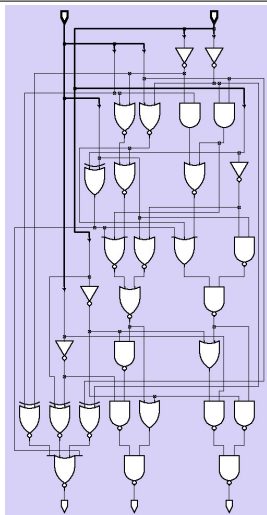
Benutzt um 1993



# Gatternetzliste des 4b-Vergleichers

## Als **strukturelles** Verilog

```
module mag_comp_gate (  
  input wire [3:0] A,  
         B,  
  output wire A_GT_B, A_LT_B, A_EQ_B);  
  
  wire  
  N107, N108, N109, N110, N111, N112, N113, N114, N115,  
  N116, N117, N118, N119, N120, N121, N122, N123, N124,  
  N125, N126, N127, N128, N129, N130, N131, N132, N133;  
  
  nr4 U89 (.A(N107), .B(N108), .C(N109), .D(N110), .Z(A_EQ_B));  
  nd2 U90 (.A(N111), .B(N112), .Z(A_GT_B));  
  nd2 U91 (.A(N113), .B(N114), .Z(A_LT_B));  
  iv U92 (.A(B[0]), .Z(N115));  
  iv U93 (.A(B[1]), .Z(N116));  
  iv U94 (.A(B[2]), .Z(N117));  
  nr2 U95 (.A(N119), .B(N120), .Z(N118));  
  iv U96 (.A(B[3]), .Z(N121));  
  nd2 U97 (.A(N123), .B(N124), .Z(N122));  
  iv U98 (.A(A[3]), .Z(N125));  
  en U99 (.A(N116), .B(A[1]), .Z(N108));  
  en U100 (.A(N125), .B(B[3]), .Z(N109));  
  en U101 (.A(N115), .B(A[0]), .Z(N110));  
  an2 U102 (.A(A[1]), .B(N116), .Z(N126));  
  nr2 U103 (.A(N116), .B(A[1]), .Z(N127));  
  nr2 U104 (.A(N115), .B(A[0]), .Z(N128));  
  nr2 U105 (.A(N127), .B(N128), .Z(N129));  
  nr3 U106 (.A(N129), .B(N126), .C(N107), .Z(N120));  
  nr2 U107 (.A(N117), .B(A[2]), .Z(N119));  
  nd2 U108 (.A(N118), .B(N121), .Z(N130));  
  nd2 U109 (.A(N130), .B(N125), .Z(N114));  
  or2 U110 (.A(N121), .B(N118), .Z(N113));  
  an2 U111 (.A(A[0]), .B(N115), .Z(N131));  
  ...  
endmodule
```





```
// 2-Input-AND-Gatter
```

```
// physikalische Zeiteinheit / Simulationsschrittweite
```

```
'timescale 100 ps / 10 ps
```

```
'celldefine
```

```
module an2 (Z, A, B);
```

```
output Z;
```

```
input A, B;
```

```
// Instanz einer VERILOG-Primitive (in KCMS nicht weiter behandelt!)
```

```
and And1 (Z, A, B);
```

```
// Verzögerungszeiten fuer steigende und fallende Flanken
```

```
specify
```

```
(A -> Z) = (1, 1);
```

```
(B -> Z) = (1, 1);
```

```
endspecify
```

```
endmodule
```

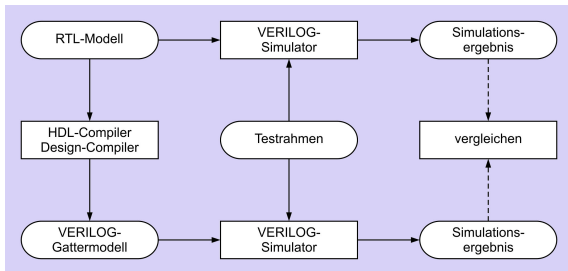
```
'endcelldefine
```



# Verifikation

# Verifikation

Verhält sich synthetisierte Schaltung noch so wie Simulationsmodell?



- ▶ Prä-Synthese ./ Post-Synthese-Simulation
- ▶ Gleiche Testdaten
  - ▶ Bei Post-Synthese aber genauere Tests möglich
  - ▶ Hier nun **genauer**es Zeitverhalten
- ▶ **Differenzen** in Ergebnissen durch anderes Zeitverhalten
- ▶ Vergleich etwas aufwendiger





## Testrahmen

```
module stimulus;
```

```
reg [3:0] A, B;
```

```
wire A_GT_B, A_LT_B, A_EQ_B;
```

```
// Instanz des Vergleichers
```

```
mag_comp Mag_comp (A, B, A_GT_B, A_LT_B, A_EQ_B);
```

```
initial
```

```
  $monitor ($time,
```

```
    "_A=%d,_B=%d,_A_GT_B=%b,_A_LT_B=%b,_A_EQ_B=%b",  
    A, B, A_GT_B, A_LT_B, A_EQ_B);
```

```
// Testmuster
```

```
initial begin
```

```
  A = 10; B = 9;  
  #10 A = 14; B = 15;  
  #10 A = 0; B = 0;  
  #10 A = 8; B = 12;  
  #10 A = 6; B = 14;  
  #10 A = 14; B = 14;
```

```
end
```

```
endmodule
```

```
0 A=10, B= 9, A_GT_B=1, A_LT_B=0, A_EQ_B=0  
10 A=14, B=15, A_GT_B=0, A_LT_B=1, A_EQ_B=0  
20 A= 0, B= 0, A_GT_B=0, A_LT_B=0, A_EQ_B=1  
30 A= 8, B=12, A_GT_B=0, A_LT_B=1, A_EQ_B=0  
40 A= 6, B=14, A_GT_B=0, A_LT_B=1, A_EQ_B=0  
50 A=14, B=14, A_GT_B=0, A_LT_B=0, A_EQ_B=1
```

Prä-Synthese-Ergebnis

# Vergleich: Prä-Synthese mit Post-Synthese

Gleiche Stimuli wie oben



TECHNISCHE  
UNIVERSITÄT  
DARMSTADT

## Prä-Synthese

0 A=10, B= 9, A\_GT\_B=1, A\_LT\_B=0, A\_EQ\_B=0  
10 A=14, B=15, A\_GT\_B=0, A\_LT\_B=1, A\_EQ\_B=0  
20 A= 0, B= 0, A\_GT\_B=0, A\_LT\_B=0, A\_EQ\_B=1  
30 A= 8, B=12, A\_GT\_B=0, A\_LT\_B=1, A\_EQ\_B=0  
40 A= 6, B=14, A\_GT\_B=0, A\_LT\_B=1, A\_EQ\_B=0  
50 A=14, B=14, A\_GT\_B=0, A\_LT\_B=0, A\_EQ\_B=1

## Post-Synthese

0 A=10, B= 9, A\_GT\_B=x, A\_LT\_B=x, A\_EQ\_B=x  
3 A=10, B= 9, A\_GT\_B=x, A\_LT\_B=x, A\_EQ\_B=0  
6 A=10, B= 9, A\_GT\_B=x, A\_LT\_B=0, A\_EQ\_B=0  
8 A=10, B= 9, A\_GT\_B=1, A\_LT\_B=0, A\_EQ\_B=0  
10 A=14, B=15, A\_GT\_B=1, A\_LT\_B=0, A\_EQ\_B=0  
16 A=14, B=15, A\_GT\_B=1, A\_LT\_B=1, A\_EQ\_B=0  
18 A=14, B=15, A\_GT\_B=0, A\_LT\_B=1, A\_EQ\_B=0  
20 A= 0, B= 0, A\_GT\_B=0, A\_LT\_B=1, A\_EQ\_B=0  
23 A= 0, B= 0, A\_GT\_B=0, A\_LT\_B=1, A\_EQ\_B=1  
28 A= 0, B= 0, A\_GT\_B=0, A\_LT\_B=0, A\_EQ\_B=1  
30 A= 8, B=12, A\_GT\_B=0, A\_LT\_B=0, A\_EQ\_B=1  
32 A= 8, B=12, A\_GT\_B=1, A\_LT\_B=0, A\_EQ\_B=0  
34 A= 8, B=12, A\_GT\_B=0, A\_LT\_B=0, A\_EQ\_B=0  
35 A= 8, B=12, A\_GT\_B=0, A\_LT\_B=1, A\_EQ\_B=0  
40 A= 6, B=14, A\_GT\_B=0, A\_LT\_B=1, A\_EQ\_B=0  
50 A=14, B=14, A\_GT\_B=0, A\_LT\_B=1, A\_EQ\_B=0  
53 A=14, B=14, A\_GT\_B=0, A\_LT\_B=0, A\_EQ\_B=1

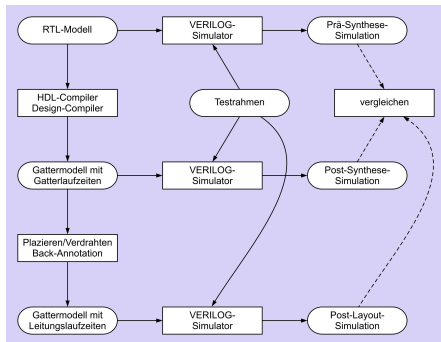
Annahme: Jedes Gatter hat 1  
Zeiteinheit **Verzögerung**



- ▶ Unterschiedlich **viele** Ergebnisse
- ▶ Verschiedene **Werte**
- ▶ Unterschiedliche **Zeiten**
  - ▶ Vorher **gar keine** ausser den im Testrahmen
- ▶ Manche Ergebnisse schlicht **falsch** (z.B. bei  $t=32$ )
- ▶ Interpretation nötig  
“Wenn man lange genug wartet, ist das Ergebnis richtig”!
- ▶ Was ist “**lange genug**”?
- ▶ Antwort: **Kritischer Pfad** (TGDI)
- ▶ Damit passender Takt für RTL wählbar **zwischen**
  - ▶ Eingangsregistern
  - ▶ Ausgangsregistern

# Weitere Verfeinerung der Verifikation

## Nun bis auf Layout-Ebene



- ▶ Post-Layout-Simulation schliesst ein
  - ▶ Gatterverzögerungen
  - ▶ Leitungsverzögerung
  - ▶ Kann umfassen: Widerstände, Kapazitäten, Induktivitäten



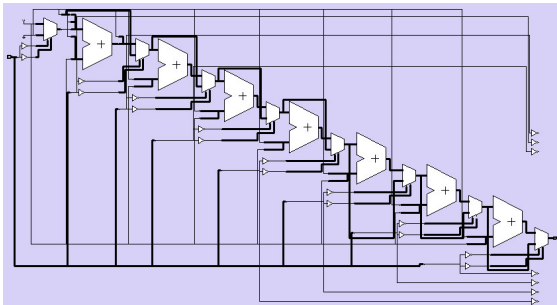
# Synthesebeispiel: Zero-Counter

- ▶ Spezifikation
  - ▶ **Eingabe** ist ein 8b Datenwort
  - ▶ **Ausgabe** soll sein die Anzahl der Null-Bits in der Eingabe
- ▶ Genauer betrachtet
  - ▶ RTL-Modell kann Synthese-Ergebnis **direkt** beeinflussen
  - ▶ Wirkung von **Design-Constraints**
- ▶ Nicht mehr so relevant
  - ▶ **Konkrete** Umsetzung in Gatter-Modell
  - ▶ Bei größeren Schaltungen oft schlicht zu **unübersichtlich**

# Zero-Counter: Intuitive Lösung

```
module count(  
  input wire [7:0] IN,  
  output reg [3:0] OUT);  
  
  integer I;  
  
  always @(IN) begin  
    OUT = 0;  
    for (I=0; I<=7; I=I+1)  
      if (IN[I]==0) OUT = OUT + 1;  
  end  
endmodule
```

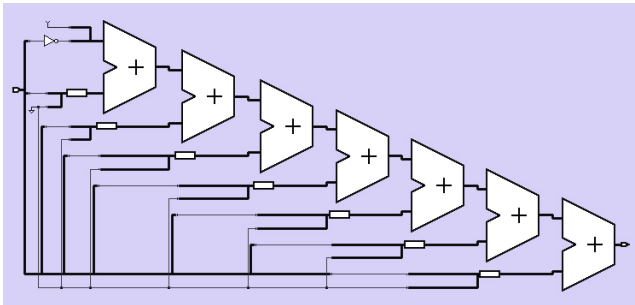
- ▶ for-Schleife wird räumlich abgerollt
- ▶ Addierer-Kaskade
- ▶ Multiplexer wählen bei jedem Bit, ob addiert wird



# Zero-Counter: Vereinfachte Lösung

```
module count(  
  input wire [7:0] IN,  
  output reg [3:0] OUT);  
  
  integer i;  
  
  always @(IN) begin  
    OUT = ~IN[0];  
    for (i=1; i<8; i=i+1)  
      OUT = OUT + ~IN[i];  
  end  
endmodule
```

- ▶ for-Schleife wird räumlich abgerollt
- ▶ Addierer-Kaskade
- ▶ Multiplexer entfallen, Bits werden direkt aufaddiert





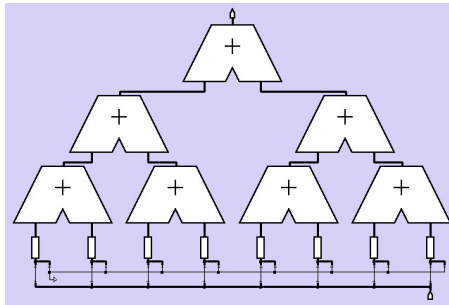
# Zero-Counter: Schlaue Lösung

```
module count(  
  input wire [7:0] IN,  
  output reg [3:0] OUT);
```

```
  always @(IN)  
    OUT = ((~IN[0]+~IN[1]) + (~IN[2]+~IN[3]))  
          + ((~IN[4]+~IN[5]) + (~IN[6]+~IN[7]));
```

```
endmodule
```

- ▶ Bits werden direkt aufaddiert
- ▶ Jetzt aber **hierarchische** Klammerung
- ▶ Damit **parallele** Berechnung
- ▶ Addierer-**Baum**

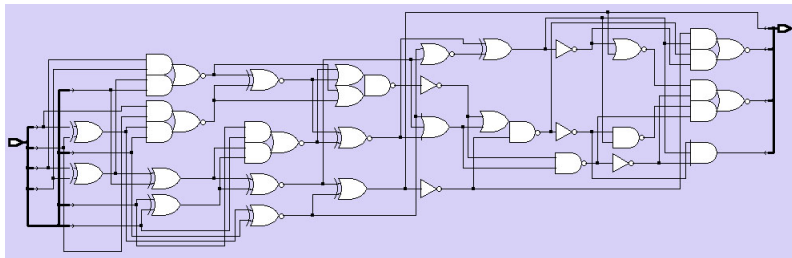




- ▶ Festlegen unterschiedlicher **Optimierungsziele**
- ▶ Üblich
  - ▶ Flächenbedarf
  - ▶ Geschwindigkeit (niedrige Verzögerung)
- ▶ Noch seltener
  - ▶ Energieverbrauch
  - ▶ Ausfallsicherheit

# Optimierung auf Fläche

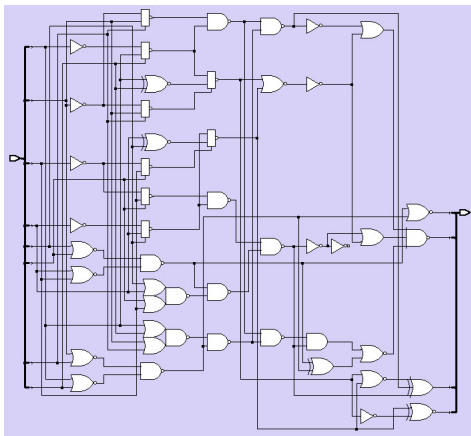
## 8b-Zero-Counter



52 Gatter, 17.8ns Verzögerung

# Optimierung auf Geschwindigkeit

## 8b-Zero-Counter

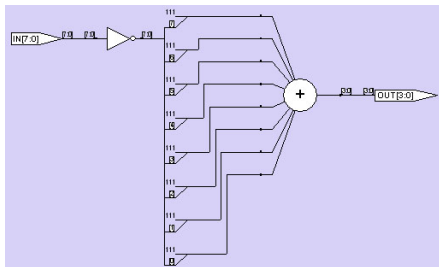


99 Gatter, 8.2ns Verzögerung

# Zero-Counter: Noch bessere Lösung

```
module count(  
  input wire [7:0] IN,  
  output reg [3:0] OUT);  
  
  always @(IN)  
    OUT = ((~IN[0]+~IN[1]) + (~IN[2]+~IN[3]))  
          + ((~IN[4]+~IN[5]) + (~IN[6]+~IN[7]));  
  
endmodule
```

- ▶ Schläueres Synthesewerkzeug
- ▶ Erkennt **Natur** der Berechnung
- ▶ Addiert alle Bits **gleichzeitig** mit  
einem 8b-Addierer





# Architektur und Entwurf von Rechnersystemen



**Wird auf eigenem Foliensatz behandelt ...**