

# Digitaltechnik

## Wintersemester 2017/2018

### 10. Vorlesung



TECHNISCHE  
UNIVERSITÄT  
DARMSTADT





1. Einleitung
2. Historie von Hardwarebeschreibungssprachen
3. SystemVerilog für kombinatorische Logik
4. SystemVerilog Modulhierarchie
5. Zusammenfassung

# Einleitung



TECHNISCHE  
UNIVERSITÄT  
DARMSTADT

0101111011100001111100111100101110101110  
0011011110100110100000111101010001010000  
0011011100011100010001001100100011111001  
0000010110101001111011110111011010100111  
0000110001000110001100100100100011101101  
1111101110110011011111111011111000010100  
1001000000001110011001011110111001101011  
0110100001100010000001100111110011111100  
0001110011111111100011011111111010111111  
0010110111110100101100010010100111111010  
0101000011101100000001111111101001110111  
0011010111010011010111000100111101000000  
0000101000100100011100110110101100101100  
0011010101011010100000100001101111110110  
1010110010010101110000100110011101100000  
1101001100011110110100111101000100110011



- ▶ Zusammenlegung von Übungsgruppen ab KW 51
  - ▶ G01 → G07
    - ▶ Mo 08:00-09:40 S202/C205
    - ▶ Thomas Kampa
    - ▶ Roland Schurig
  - ▶ G22 → G02
    - ▶ Mo 15:20-17:00 S311/006
    - ▶ Timo Henz
    - ▶ Moritz Nottebaum
- ▶ Hausaufgaben fürs neue Jahr: SystemVerilog Tools ausprobieren
  - ▶ Ende KW 51 im Moodle

# Rückblick auf die letzten Vorlesungen



TECHNISCHE  
UNIVERSITÄT  
DARMSTADT

- ▶ Sequentielle Logik
  - ▶ Sequentielle Schaltungen
  - ▶ Speicherelemente
  - ▶ Synchrone sequentielle Logik
- ▶ Endliche Zustandsautomaten
  - ▶ Konzept, Notationen und Anwendungsbeispiele
  - ▶ Moore vs. Mealy
  - ▶ Zerlegen von Zustandsautomaten
- ▶ Zeitverhalten synchroner sequentieller Schaltungen
- ▶ Parallelität



Harris 2013  
Kap. 3.1 - 3.6

# Wiederholung: Parallelität

## Nochmal Plätzchen backen



TECHNISCHE  
UNIVERSITÄT  
DARMSTADT

### ► Annahmen:

- genug Teig ist fertig
- 5 Minuten Teig ausrollen (R)
- 5 Minuten Blech belegen (B)
- 15 Minuten backen (O)
- 5 Minuten verzieren (V)

### ⇒ Durchsatz steigern mit

- zeitlicher Parallelität
- räumlicher Parallelität



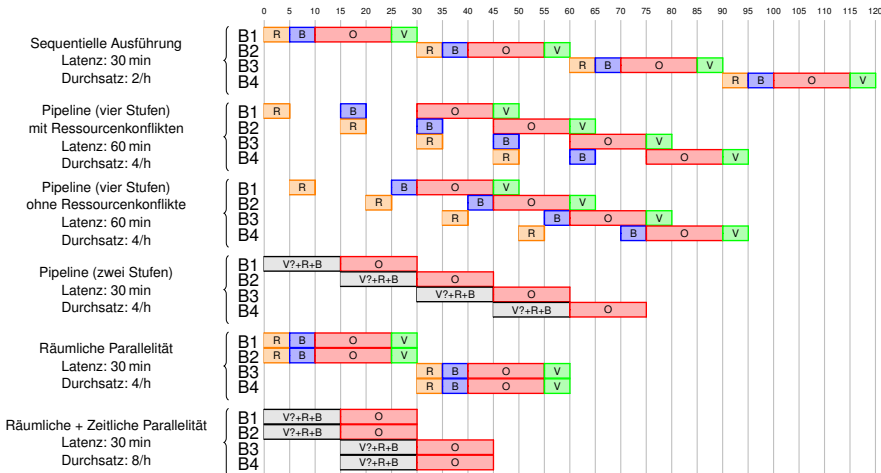
# Wiederholung: Parallelität

## Nochmal Plätzchen backen



TECHNISCHE  
UNIVERSITÄT  
DARMSTADT

Zeit (Minuten)



# Wiederholung: Parallelität

## Nochmal Plätzchen backen



TECHNISCHE  
UNIVERSITÄT  
DARMSTADT

- ▶ einfacher Pipeline-Ansatz:
  - ▶ alle (vier) Teilaufgaben jeweils in eigener Pipeline-Stufe
  - ▶ längste Teilaufgabe (O) bestimmt Taktrate (15 min)
  - ⇒ Ressourcenkonflikt wegen Überlappung der manuellen Teilaufgaben (V,B,R) aufeinanderfolgender Bleche (nur ein Bäcker)
- ▶ scheinbare Lösung:
  - ▶ Teilaufgaben R und B innerhalb ihres Slots nach hinten verschieben
  - ▶ ist in richtiger Pipeline aber nicht möglich (alle Stufen starten gleichzeitig)
- ▶ bessere Lösung:
  - ▶ kleinere Teilaufgaben in einer Pipeline-Stufe ( $V+R+B$ ) zusammenfassen
  - ▶ Verzieren (V) nur, wenn Blech bereits gebacken (O)
  - ⇒ B1 wird erst verziert, wenn B2 bereits im Ofen ist bzw. B3 vorbereitet wird
  - ⇒ am Ende bleibt ein unverziertes Blech übrig
- ▶ Analog: oft Prolog / Epilog bei Schleifen-Pipelining notwendig



# Wiederholung: Schichtenmodell



TECHNISCHE  
UNIVERSITÄT  
DARMSTADT

Anwendungs- software	Programme
Betriebs- systeme	Gerätetreiber
Architektur	Befehle Register
Mikro- architektur	Datenpfade Steuerung
Logik	Addierer Speicher
Digital- schaltungen	UND Gatter Inverter
Analog- schaltungen	Verstärker Filter
Bauteile	Transistoren Dioden
Physik	Elektronen

# Überblick der heutigen Vorlesung



TECHNISCHE  
UNIVERSITÄT  
DARMSTADT

- ▶ Historie von Hardwarebeschreibungssprachen
- ▶ SystemVerilog für kombinatorische Logik
- ▶ SystemVerilog Modulhierarchie



Harris 2013  
Kap. 4.1-4.3  
Seite 167 - 190

# Historie von Hardwarebeschreibungssprachen



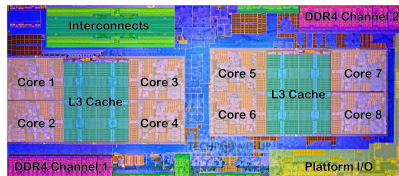
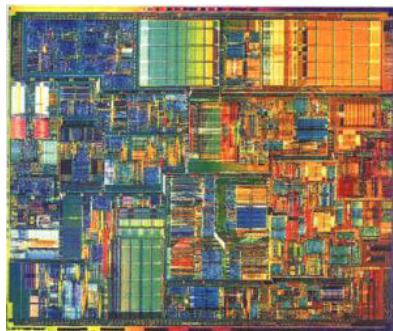
TECHNISCHE  
UNIVERSITÄT  
DARMSTADT

01000001001011011110000011010100101111001  
1110010110000101010110101010101101100100  
0000110110101110101100101011010101100101  
0001101100111010001110100010111110110000  
1111101100011001011101100111001110000001  
1001110001010110011011000111100011011010  
0011001111010110101100100000011010101101  
01100000000001111000101010111110000011  
1010111100101010110111000001010110100011  
0111000110101111101011001011010100001110  
0110011110100010100111101000010101100011  
0110100110001011001000010110110110110000  
1010010101011000100001101000001011101000  
1101001001100000011110110100010100000011  
1111101010100010001100000011100000000111  
1001011010110011110011000001000100111111

# Notwendigkeit von HDLs

## *Hardware Description Language*

- ▶ Komplexität technischer Systeme steigt ständig (vgl. Moores Gesetz)
  - ▶ 2000: Intel Pentium 4:  
 $42 \cdot 10^6$  Transistoren auf  $217 \text{ mm}^2$
  - ▶ 2017: AMD Ryzen:  
 $4,8 \cdot 10^9$  Transistoren auf  $192 \text{ mm}^2$
- ⇒ ohne rechnergestützte Hilfsmittel nicht zu beherrschen
- ⇒ Hardware-Beschreibungssprachen zum Beherrschen von Komplexität
  - ▶ Hierarchie
  - ▶ Modularität
  - ▶ Regularität





- ▶ seit Beginn der Rechnerentwicklung:
  - ▶ Suche nach verständlichen und einheitlichen Beschreibungssprachen für
    - ▶ Designspezifikation
    - ▶ Simulation
    - ▶ Verifikation
    - ▶ Dokumentation
  - ⇒ nutzt auch der Kommunikation zwischen Entwicklern
- ▶ zunächst Hochsprachen (bspw. Pascal, LISP, Petri-Netze) zur Hardware-Beschreibung eingesetzt
- ▶ 1960/70: Register-Transfersprachen
  - ▶ *Datentransfer* zwischen *Registern* durch kombinatorische Operatoren
  - ⇒ synchrone sequentielle Schaltungen als Abstraktionslevel

# Robert Piloty, 1924 - 2013



TECHNISCHE  
UNIVERSITÄT  
DARMSTADT

- ▶ Maßgeblich an Einführung/Entwicklung des Informatikstudiums in Deutschland beteiligt
  - ▶ Forschung an
    - ▶ programmgesteuerten Rechenanlagen (PERM)
    - ▶ rechnergestützter Schaltungsentwurf
  - ▶ RTS 1a (Register Transfer System Language)
    - ▶ an TH Darmstadt entwickelt
    - ▶ entstand aus praktischer Erfahrung
    - ▶ sollte Fehler früherer Ansätze vermeiden (bspw. zu hohes Abstraktionsniveau)
    - ▶ sollte leicht zu lernen und zu lehren sein
    - ▶ sollte verschiedene Entwurfsmethoden und Entwurfsebenen abdecken
    - ▶ einfache syntaktische und semantische Regeln
- ⇒ jede gültige Hardwarebeschreibung in RTS 1a soll auch realisierbar sein



# Beispiel für RTS 1a Beschreibung

Quelle: Computer Aids for VLSI Circuits, 1981



TECHNISCHE  
UNIVERSITÄT  
DARMSTADT

example.rts

```
1  -INPUTTERMINAL- START, MRE[1:4], MDE[1:4];
2  -REGISTER- AK[1:4], MR[1:4], MD[1:4], CC[1:4], ST[1:2];
3  -CASE- ST
4  :0: -IF- START -THEN-
5      AK <= #0B4, MR <= MRE, MD <= MDE, CC <= #12D4, ST <= #1B2 -FI-
6  :1: -IF- MR[4] -THEN-
7      AK <= ADD(AK, MD), ST <= #2D2
8      -ELSE-
9      (AK,MR) <= RSH(#0, (AK,MR)), CC <= INC(CC),
10     -IF- EQ(CC, #15D4) -THEN- ST <= #0B2 -ELSE- ST <= #1B2 -FI-
11     -FI-
12 :2: (AK,MR) <= RSH(#0, (AK,MR)), CC <= INC(CC),
13     -IF- EQ(CC, #15D4) -THEN- ST <= #0B2 -ELSE- ST <= #1B2 -FI-
14 -ESAC-
15 -FINIS-
```

# 1983 - Geburtsstunde wichtiger HDL Standards



TECHNISCHE  
UNIVERSITÄT  
DARMSTADT

- ▶ Consensus Language (CONLAN)
  - ▶ allgemeine, erweiterbare Sprache
  - ▶ sollte den akademischen „Wildwuchs“ in geordnete Bahnen lenken
  - ⇒ Akzeptanz von HDLs in Industrie fördern
- ▶ Very High-Speed Integrated Circuits Hardware Description Language (VHDL)
  - ▶ vom US Department of Defense maßgeblich gefördert
  - ▶ IEEE Standard 1076 (1987, 1993, 2002, 2008)
  - ▶ Erweiterung:
    - ▶ 1998: VHDL-AMS (Analog and Mixed-Signal)
- ▶ Verilog HDL
  - ▶ von Gateway Design Automation (Cadence) zur Simulation entwickelt
  - ▶ IEEE Standard 1364 (1995, 2001)
  - ▶ Erweiterung:
    - ▶ 1998: Verilog-AMS (Analog and Mixed-Signal)
    - ▶ 2002: SystemVerilog (Verifikation)



# Aktueller Tendenz: Anstieg des Abstraktionslevels



TECHNISCHE  
UNIVERSITÄT  
DARMSTADT

- ▶ SystemC
    - ▶ C++ Klassenbibliothek
    - ▶ erlaubt besonders schnelle Simulation
  - ▶ Constructing Hardware in a Scala Embedded Language (Chisel)
    - ▶ von UC Berkeley
    - ▶ durch Einbettung in Scala (funktionales Java) sehr flexibel
  - ▶ BlueSpec-Verilog (BSV)
    - ▶ vom MIT, aber inzwischen kommerzialisiert
    - ▶ erbt Abstraktionsniveau von funktionalem Haskell
  - ▶ High-Level-Synthese: low-level Verilog/VHDL aus abstrakten Anwendungsbeschreibungen (bspw. in C, Java, Matlab) erzeugen
- ⇒ Schritt von Beschreibung zur Ausführung (Semantic Gap) wird immer größer



- ▶ Simulation des funktionalen/zeitlichen Verhaltens der beschriebenen Schaltung
  - ▶ berechnete Ausgaben zu vorgegebenen Eingaben werden auf Korrektheit geprüft
  - ⇒ Fehlersuche einfacher (billiger) als in realer Hardware
- ▶ Synthese übersetzt Hardware-Beschreibungen in Netzlisten
  - ▶ Schaltungselemente (Logikgatter) + Verbindungsknoten
  - ▶ entspricht Registertransferebene
  - ▶ kann auf Gatter-Bibliothek einer konkreten Zielarchitektur abgebildet werden (Technology-Mapping)
    - ▶ wenige CMOS-Basisgatter für Application-Specific Integrated Circuits (ASICs)
    - ▶ kleine Lookup-Tabellen für Field-Programmable Gate Arrays (FPGAs)
- ▶ WICHTIG: für effiziente Hardware-Beschreibung muss HDL-Programmierer *immer* die Zielarchitektur im Auge behalten

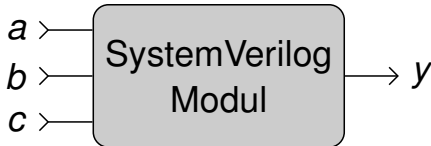
# SystemVerilog für kombinatorische Logik



TECHNISCHE  
UNIVERSITÄT  
DARMSTADT

0101101100010010100011110000001100111001  
0010001101100000001101011000100011001110  
1011000000001001110110100011010110101111  
1011110011010100100101010000111000011111  
000010001000000011111011011111011111010  
0001000000101111011000111110010001011011  
1000010000100100010011010001011000110010  
000000111111001010001100000001110110010  
1000101001110101111001101010111000101011  
0011100110101110010100110111010010110110  
0000010010000110010000000010110100000101  
0111010010001010110010111110111010111010  
0100111010010011011001100111100100011101  
0111110000110100100100100001011010010100  
0011000100001010101001110111010000101010  
1100100010111010001110100011101110010101

- ▶ Schnittstellenbeschreibung:
    - ▶ Eingänge
    - ▶ Ausgänge
    - ▶ (Parameter)
  - ▶ zwei Arten von Modul-Beschreibungen:
    - ▶ Struktur: Wie ist die Schaltung aus (Sub-)Modulen aufgebaut?
    - ▶ Verhalten: Was tut die Schaltung?
- ⇒ strukturelle Modul-Hierarchie mit Verhaltensbeschreibung auf unterster Ebene



# Beispiel für Verhaltensbeschreibung



example.sv

```
1 module example(input logic a, b, c, output logic y);  
2  
3     assign y = ~a & ~b & ~c | a & ~b & ~c | a & ~b & c;  
4  
5 endmodule
```

- ▶ **module** Beginn der Schnittstellenbeschreibung
- ▶ **example** Modulname
- ▶ **input, output** Port-Richtung
- ▶ **logic** Port-Datentyp
- ▶ **a,b,c,y** Port-Namen
- ▶ **assign** (kombinatorische) Signalzuweisung
- ▶ **~ & |** (kombinatorische) Operatoren (NOT, AND, OR)
- ▶ **endmodule** Ende der Schnittstellenbeschreibung



- ▶ Unterscheidet Groß- und Kleinschreibung
  - ▶ bspw. `reset`  $\neq$  `Reset`
- ▶ Bezeichner für Modul- und Signalnamen dürfen nicht mit Ziffern anfangen
  - ▶ bspw. `2mux` ungültig
- ▶ Anzahl von Leerzeichen, Leerzeilen und Tabulatoren irrelevant
- ▶ Kommentare:
  - ▶ *// bis zum Ende der Zeile*
  - ▶ */\* über mehrere Zeilen \*/*

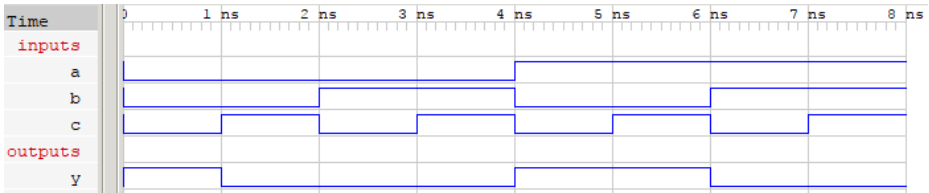
# Simulation von Verhaltensbeschreibungen



TECHNISCHE  
UNIVERSITÄT  
DARMSTADT

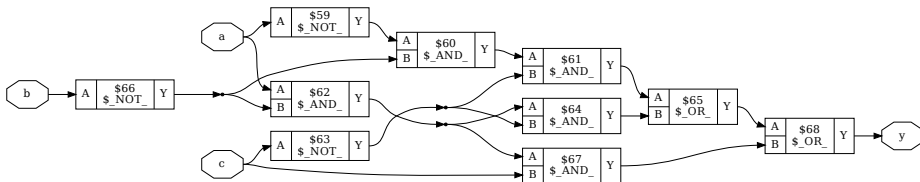
example.sv

```
1 module example(input logic a, b, c, output logic y);  
2  
3     assign y = ~a & ~b & ~c | a & ~b & ~c | a & ~b & c;  
4  
5 endmodule
```



example.sv

```
1 module example(input logic a, b, c, output logic y);  
2  
3   assign y = ~a & ~b & ~c | a & ~b & ~c | a & ~b & c;  
4  
5 endmodule
```





# SystemVerilog Modulhierarchie

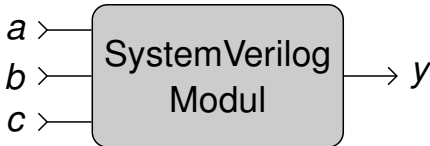


TECHNISCHE  
UNIVERSITÄT  
DARMSTADT

1111110100111101010111000100001011010100  
0100101001001000010110001100011010011101  
1111011001001101001001011001111000110011  
1011100011111101100110100010111011110110  
1011010110001011001100111001101000000101  
0110010110101000110010111001010000111010  
0000000000011111101110010010111011001110  
0111111001000101111010010000100011100001  
01111111010011111110011010001000000010100  
01111110101001111011001110111110101001011  
01001100001110110111111011000100100001010  
0001110100001100100101010010001100110001  
0000011100101011100010000100100111101111  
00010111101101001100111110111110000111110  
0000000101001001100101001111110100011111  
100011100111111010010011111110101111001011

- ▶ Schnittstellenbeschreibung:
  - ▶ Eingänge
  - ▶ Ausgänge
  - ▶ (Parameter)
- ▶ zwei Arten von Modul-Beschreibungen:
  - ▶ Struktur: Wie ist die Schaltung aus (Sub-)Modulen aufgebaut?
  - ▶ Verhalten: Was tut die Schaltung?

⇒ strukturelle Modul-Hierarchie mit Verhaltensbeschreibung auf unterster Ebene



# Strukturelle Beschreibung: Modulinstantiierung



TECHNISCHE  
UNIVERSITÄT  
DARMSTADT

and3.sv

```
1 module and3(input logic a, b, c, output logic y);  
2     assign y = a & b & c;  
3 endmodule
```

inv.sv

```
1 module inv(input logic a, output logic y);  
2     assign y = ~a;  
3 endmodule
```

nand3.sv

```
1 module nand3 (input logic d, e, f, output logic w);  
2     logic s;                //internes Signal für Modulverbindung  
3     and3 andgate(d, e, f, s); //Instanz von and3 namens andgate  
4     inv inverter(s, w);      //Instanz von inv namens inverter  
5 endmodule
```

# Strukturelle Beschreibung: Portzuweisung nach Position oder Namen



TECHNISCHE  
UNIVERSITÄT  
DARMSTADT

nand3.sv

```
1 module nand3 (input logic d, e, f, output logic w);
2     logic s;                //internes Signal für Modulverbindung
3     and3 andgate(d, e, f, s); //Instanz von and3 namens andgate
4     inv inverter(s, w);      //Instanz von inv namens inverter
5 endmodule
```

nand3\_named.sv

```
1 module nand3_named(input logic d, e, f, output logic w);
2     logic s;
3     and3 andgate(.a(d), .b(e), .c(f), .y(s));
4     inv inverter(.a(s), .y(w));
5 endmodule
```

► 10 bis 100 ports pro Modul nicht unüblich

⇒ absolute Portzuweisung per Namen übersichtlicher (selbstdokumentierend)

# Bitweise Verknüpfungsoperatoren



gates.sv

```
1  module gates (input  logic [3:0] a, b,  
2                  output logic [3:0] y1,y2,y3,y4,y5);  
3  
4      /* Fünf unterschiedliche Logikgatter  
5         mit zwei Eingängen, jeweils 4b Busse */  
6      assign y1 =    a & b;    // AND  
7      assign y2 =    a | b;    // OR  
8      assign y3 =    a ^ b;    // XOR  
9      assign y4 = ~(a & b);    // NAND  
10     assign y5 = ~(a | b);    // NOR  
11  
12 endmodule
```

# Reduktionsoperatoren (unär)



and8.sv

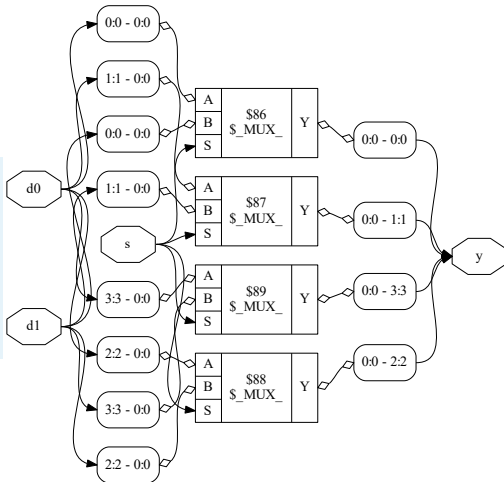
```
1 module and8 (input logic [7:0] a, output logic y);
2
3     assign y = &a;
4     // Abkürzung für
5     // assign y = a[7] & a[6] & a[5] & a[4] &
6     //             a[3] & a[2] & a[1] & a[0];
7
8 endmodule
```

## ► analog:

- | OR
- ^ XOR
- ~| NOR
- ~& NAND
- ~^ XNOR

# Bedingte Zuweisung (ternär) und deren Syntheseergebnis

```
1  module mux2
2  (input logic [3:0] d0,d1,
3   input logic s,
4   output logic [3:0] y);
5
6   assign y = s ? d1 : d0;
7
8  endmodule
```



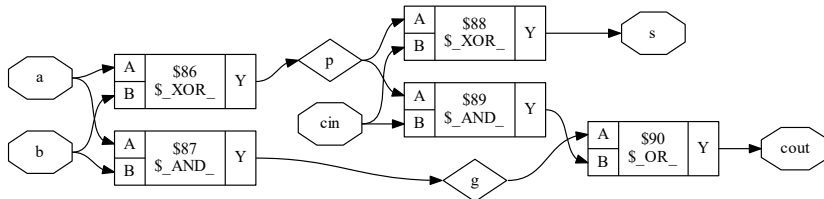
# Interne Verbindungsknoten (Signale)



TECHNISCHE  
UNIVERSITÄT  
DARMSTADT

fulladder.sv

```
1 module fulladder(input logic a, b, cin,  
2                   output logic s, cout);  
3  
4   logic p, g;    // interne Verbindungsknoten  
5   assign p = a ^ b;  
6   assign g = a & b;  
7   assign s = p ^ cin;  
8   assign cout = g | (p & cin);  
9  
10  endmodule
```





# Bindung von Operatoren (Präzedenz)



TECHNISCHE  
UNIVERSITÄT  
DARMSTADT

- ▶ `[]` Zugriff auf Vektorelement (höchste Präzedenz)
- ▶ `~, !, -, &, ^` unäre Operatoren: NOT, Negation, Reduktion
- ▶ `*, /, %` Multiplikation, Division, Modulo
- ▶ `+, -` Addition, Subtraktion
- ▶ `<<, >>, <<<, >>>` logischer und arithmetischer Shift
- ▶ `<, <=, >, >=` Vergleich
- ▶ `==, !=` gleich, ungleich
- ▶ `&, ~&` bitweise AND, NAND
- ▶ `^, ~^` bitweise XOR, XNOR
- ▶ `|, ~|` bitweise OR, NOR
- ▶ `&&` logisches AND (Vektoren sind genau dann wahr,
- ▶ `||` logisches OR wenn wenigstens ein Bit 1 ist)
- ▶ `?:` ternärer Operator
- ▶ `{}` Konkatination (niedrigste Präzedenz)

# Syntax für numerische Literale



TECHNISCHE  
UNIVERSITÄT  
DARMSTADT

- ▶ Syntax: `<N>'<B><wert>`
  - ▶ `<N>` = Bitbreite
  - ▶ `<B>` = Basis (d,b,o,h)
  - ▶ beide Angaben optional (default: 32'd)
  - ▶ Unterstriche als optische Trenner möglich (werden ignoriert)

Literal	Bitbreite	Basis	Dezimal	Binär
3'b101	3	binär	5	101
'b11	32	binär	3	0000...0000011
8'b11	8	binär	3	00000011
8'b1010_1011	8	binär	171	10101011
3'd6	3	dezimal	6	110
6'o42	6	oktal	34	100010
8'hAB	8	hexadezimal	171	10101011
42	32	dezimal	42	0000...0101010



concat.sv

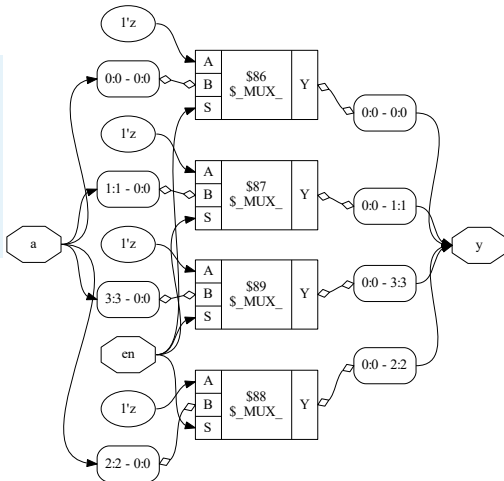
```
1 module concat(input logic [2:0] a, b, output logic [11:0] y);  
2  
3     assign y = {a[2:1], {3{b[0]}}, a[0], 6'b100010};  
4 // entspricht  
5 // y = a[2] a[1] b[0] b[0] b[0] a[0] 1 0 0 0 1 0  
6  
7 endmodule
```

# Hochohmiger Ausgang (Z) und dessen *falsche* Synthese



tristate.sv

```
1 module tristate
2   (input  logic [3:0] a,
3    input  logic      en,
4    output logic [3:0] y);
5
6   assign y = en ? a : 4'bz;
7
8 endmodule
```

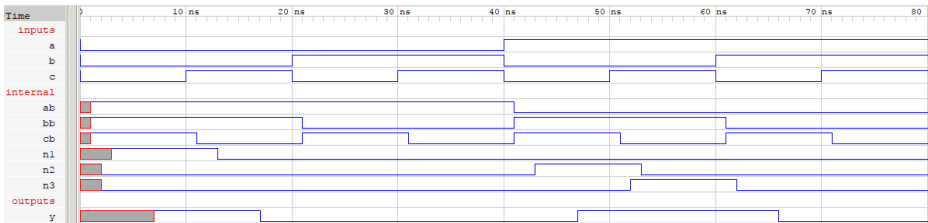


- ▶ Z darf *nur an Ausgängen* verwendet werden
- ▶ für interne Signale aber trotzdem simulierbar

# Verzögerungen: # Zeiteinheiten

example\_delay.sv

```
1 'timescale 1ns / 10ps
2 module example_delay(input logic a, b, c, output logic y);
3     logic ab, bb, cb, n1, n2, n3;
4     assign #1 {ab, bb, cb} = ~{a, b, c};
5     assign #2 n1 = ab & bb & cb;
6     assign #2 n2 = a & bb & cb;
7     assign #2 n3 = a & bb & c;
8     assign #4 y = n1 | n2 | n3;
9 endmodule
```



# Zusammenfassung



TECHNISCHE  
UNIVERSITÄT  
DARMSTADT

01111100010001001010010010010011011000001  
1011101011100000100111000111011110100100  
0001010111111000010010010111011111101111  
0101101100101110111101100110111010100011  
101110111011111001000110101011111111000  
01111110101100001001011111000011111101110  
1101110110101110001110111010001010111111  
01000100100000100001010010110000111111  
1010000010011001010100011111010100010010  
1100110011110000100010111011011110000011  
00011110111111001111110100011101000111010  
1100010100000110110000100111110010001000  
1001001100101111101101010010001011000001  
0111110110101101100101000010001001111010  
0000111000111000111111100010110010010110  
1000011011110001011010000101101110001000



- ▶ Historie von Hardwarebeschreibungssprachen
- ▶ SystemVerilog für kombinatorische Logik
- ▶ SystemVerilog Modulkonstruktion
  
- ▶ Nächste Vorlesung behandelt
  - ▶ SystemVerilog für sequentielle Logik