Digitaltechnik

Wintersemester 2025/2026

Vorlesung 7

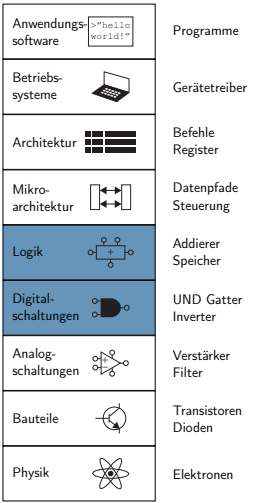
TECHNISCHE UNIVERSITÄT DARMSTADT

ENCRYPTO
CRYPTOGRAPHY AND PRIVACY ENGINEERING

Inhalt

1. Arithmetische Grundschaltungen
2. SystemVerilog Datentypen
3. SystemVerilog für kombinatorische Logik (Forts.)

Harris 2016
4.2, 4.5, 4.7, 5.2



ENCRYPTO
CRYPTOGRAPHY AND
PRIVACY ENGINEERING

Abgabefrist für Hausaufgabe C zu Vorlesungen 05 und 06 nächste Woche Freitag 23:59!

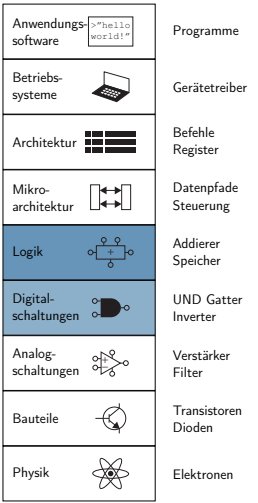
Wöchentliches Moodle-Quiz nicht vergessen!

Agenda

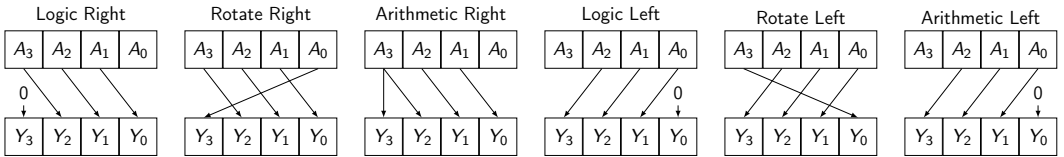
1 Arithmetische Grundschaltungen

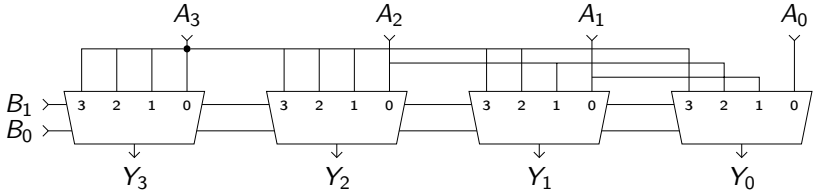
2 SystemVerilog Datentypen

3 SystemVerilog für kombinatorische Logik (Forts.)

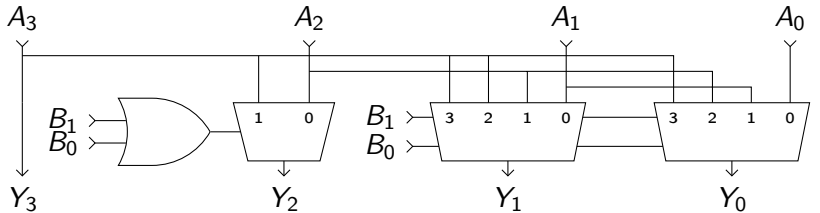


- A um B Stellen nach links/rechts verschieben
- Strategien zum Auffüllen der freien Stellen (Beispiele unten sind für $B = 1$):
  - *logischer* Rechts- oder Linksshift: Auffüllen mit Nullen
  - *umlaufender* Rechts- oder Linksshift: Auffüllen mit den aus der anderen Seite herausfallenden Bits (Rotation)
  - *arithmetischer* Rechtsshift: Auffüllen mit Vorzeichen des als Zweierkomplement interpretierten Dateneingangs (entspricht Division durch 2^B)
  - *arithmetischer* Linksshift: Auffüllen mit Nullen (entspricht Multiplikation mit 2^B)





Minimierte Schaltung



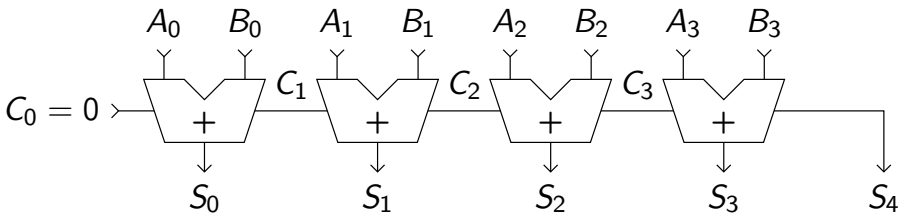
- Arithmetischer Linksshift um n Stellen multipliziert den Zahlenwert mit 2^n
  - $00001_2 <<< 3 = 01000_2 = 1 \cdot 2^3 = 8$
  - $11101_2 <<< 2 = 10100_2 = -3 \cdot 2^2 = -12$

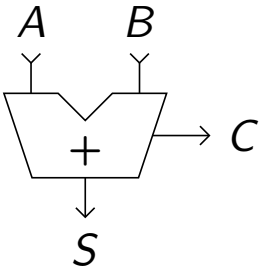
$\Rightarrow$ Multiplikation mit Konstanten kann aus Arithmetischen Linksshifts und Additionen zusammengesetzt werden

  - $a \cdot 6 = a \cdot 110_2 = (a <<< 2) + (a <<< 1)$
- Arithmetischer Rechtsshift um n Stellen dividiert den Zahlenwert durch 2^n
  - $010000_2 >>> 4 = 000001_2 = 16/2^4 = 1$
  - $100000_2 >>> 2 = 111000_2 = -32/2^2 = -8$

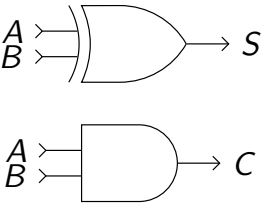
Ripple-Carry-Adder (RCA)

	1	0	1	1	0	Übertrag	C_4	C_3	C_2	C_1	C_0
		1	0	1	1	Summand		A_3	A_2	A_1	A_0
+		1	0	1	1	Summand		B_3	B_2	B_1	B_0
=	1	0	1	1	0	Summe	S_4	S_3	S_2	S_1	S_0

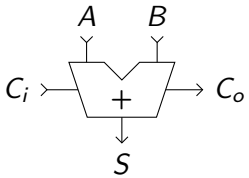




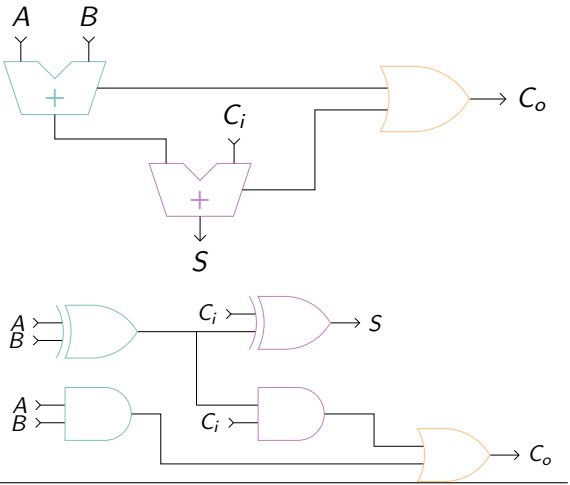
A	B	C	S
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0

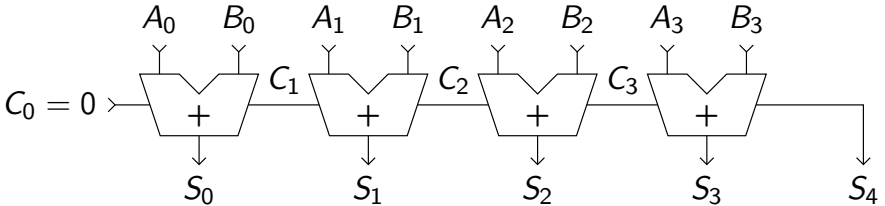


Volladdierer



A	B	C_i	C_o	S
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1





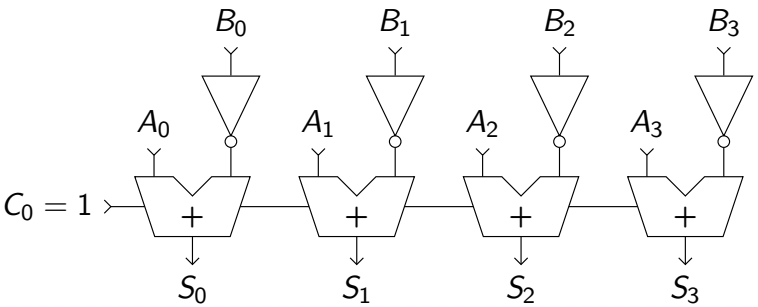
- Überträge werden über Kette von 1 bit Volladdierern vom LSB zum MSB weitergegeben

⇒ **Problem:** Langer kritischer Pfad (steigt linear mit Bitbreite)

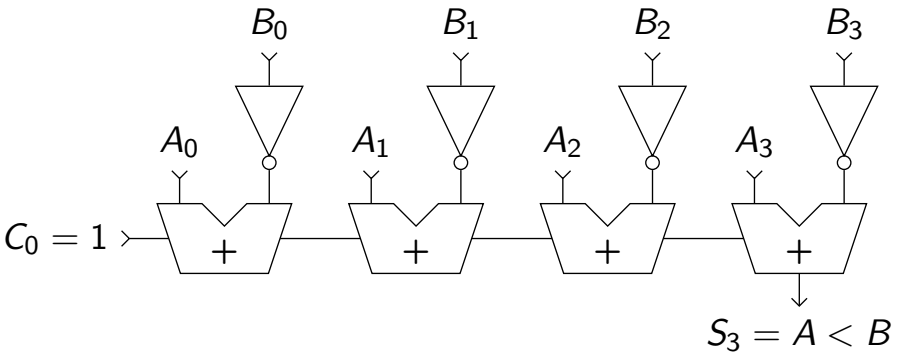
⇒ Mögliche Lösungen sehen wir in der nächsten Vorlesung!

- kann mit Addition und Negation realisiert werden: $A - B = A + (-B)$
- Negation im Zweierkomplement: Komplement und Inkrement

$\Rightarrow$ Addierer mit NOT-Gatter an B-Eingängen und $C_0 = 1$

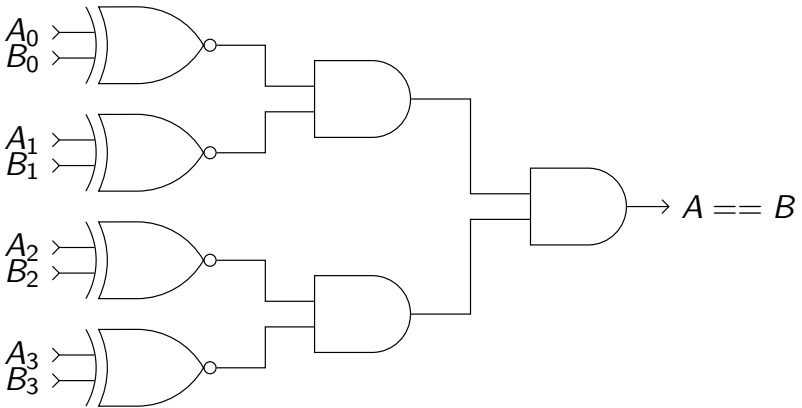


- kann mit Subtraktion realisiert werden: $A < B \Leftrightarrow A - B < 0$



Vergleich: Gleichheit

- Bitweise XNOR und AND-Baum



Multiplizierer

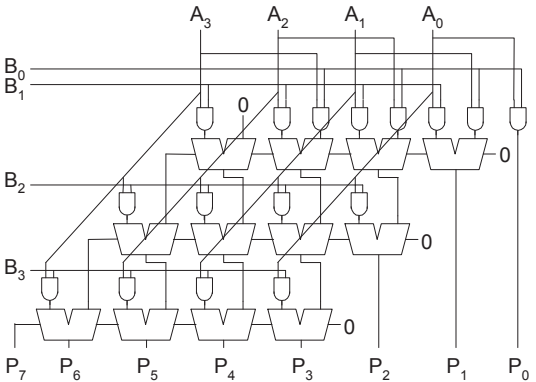
- Produkt von n und m bit breiten Faktoren ist $n + m$ bit breit
- Teilprodukte aus einzelnen Ziffern des Multiplikators mit dem Multiplikanden
- verschobene Teilprodukte danach addieren

Decimal		**Binary**
230	Multiplikand	0101
x 42	Multiplikator	x 0111
460		0101
+ 920	Teilprodukte	0101
9660		0101
		+ 0000
	Ergebnis	0100011

230 x 42 = 9660 5 x 7 = 35

Kombinatorische 4 × 4 Multiplikation

$$
\begin{array}{cccccccc}
 & & & A_3 & A_2 & A_1 & A_0 & \\
 & & \times & B_3 & B_2 & B_1 & B_0 & \\
\hline
 & & & A_3B_0 & A_2B_0 & A_1B_0 & A_0B_0 & \\
 & & A_3B_1 & A_2B_1 & A_1B_1 & A_0B_1 & & \\
 & A_3B_2 & A_2B_2 & A_1B_2 & A_0B_2 & & & \\
+ & A_3B_3 & A_2B_3 & A_1B_3 & A_0B_3 & & & \\
\hline
P_7 & P_6 & P_5 & P_4 & P_3 & P_2 & P_1 & P_0
\end{array}
$$



ENCRYPTO
CRYPTOGRAPHY AND
PRIVACY ENGINEERING

Quiz 3 - The Return of the King

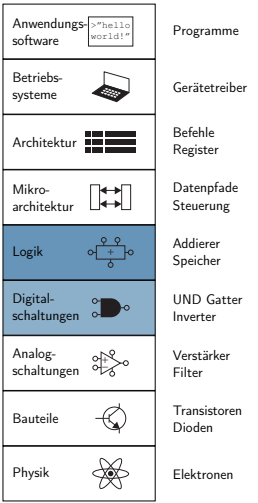
(Hat nichts mit dem Moodle-Quiz zu tun)

Agenda

1 Arithmetische Grundschaltungen

2 SystemVerilog Datentypen

3 SystemVerilog für kombinatorische Logik (Forts.)



Auswahl wichtiger Datentypen
In dieser Veranstaltung wichtige in rot

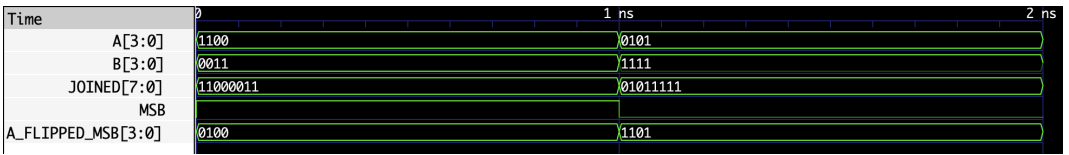
- `bit` = `{1'b0, 1'b1}` (zweiwertige Logik)
- `logic` = `{1'b0, 1'b1, 1'bx, 1'bz}` (vierwertige Logik)
- `int` = `{-2**31, ..., 2**31-1} = bit signed [31:0]`
- `integer` = `{-2**31, ..., 2**31-1} = logic signed [31:0]`
- `enum` = Aufzählung symbolischer Werte (bspw. für endliche Automaten)
- `time, real, typedef, struct, ...`
- Vektoren und Arrays

Vektoren

ENCRYPTO
CRYPTOGRAPHY AND
PRIVACY ENGINEERING

vectors.sv

```
1  module vectors(input  logic [3:0] A, B,   // 4 bit Vektoren [MSB:LSB]
2                 output logic [7:0] JOINED, // 8 bit Vektor [MSB:LSB]
3                 output logic MSB,
4                 output logic [3:0] A_FLIPPED_MSB);
5
6    assign JOINED[7:4] = A;            // Vektorbereich überschreiben
7    assign JOINED[3:0] = B;            // Vektorbereich überschreiben
8
9    assign MSB = A[3];                 // Einzelnes Vektorbit lesen
10
11   assign A_FLIPPED_MSB[3] = ~A[3];  // Einzelnes Vektorbit lesen/schreiben
12   assign A_FLIPPED_MSB[2:0] = A[2:0];
13
14 endmodule
```

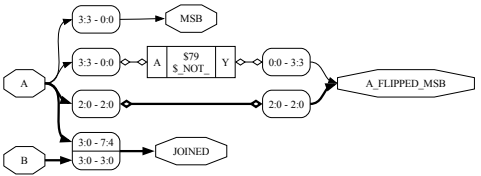


Synthese von Vektoren

ENCRYPTO
CRYPTOGRAPHY AND
PRIVACY ENGINEERING

vectors.sv

```
1  module vectors(input  logic [3:0] A, B,   // 4 bit Vektoren [MSB:LSB]
2                 output logic [7:0] JOINED, // 8 bit Vektor [MSB:LSB]
3                 output logic MSB,
4                 output logic [3:0] A_FLIPPED_MSB);
5
6    assign JOINED[7:4] = A;            // Vektorbereich überschreiben
7    assign JOINED[3:0] = B;            // Vektorbereich überschreiben
8
9    assign MSB = A[3];                 // Einzelnes Vektorbit lesen
10
11   assign A_FLIPPED_MSB[3] = ~A[3];  // Einzelnes Vektorbit lesen/schreiben
12   assign A_FLIPPED_MSB[2:0] = A[2:0];
13
14 endmodule
```

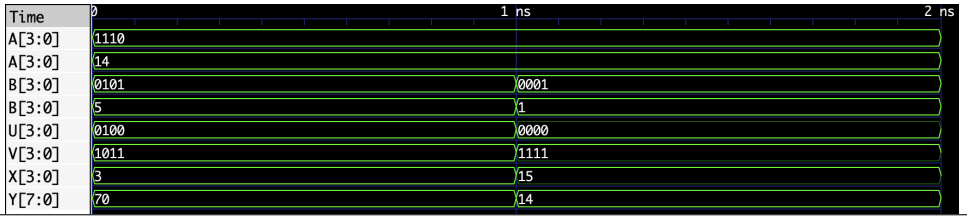


Operationen auf Vektoren

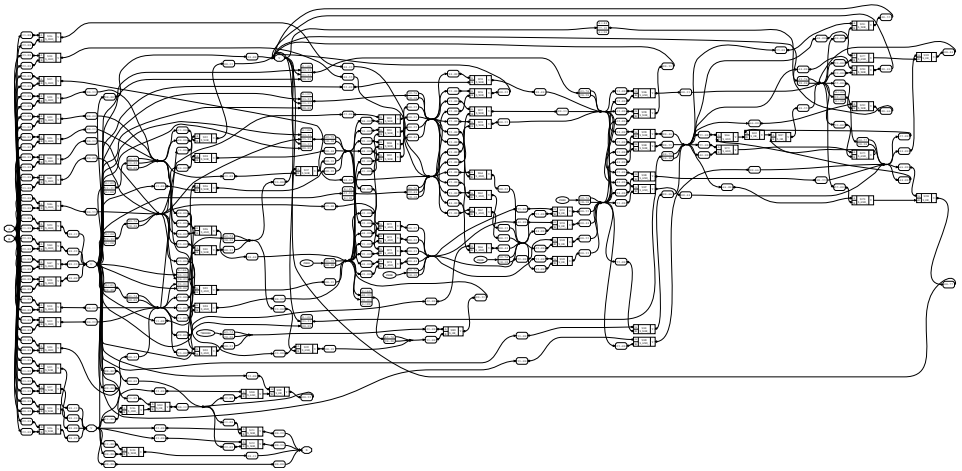
ENCRYPTO CRYPTOGRAPHY AND PRIVACY ENGINEERING

vecop.sv

```
1  module vecop(input  logic [3:0] A, B,
2               output logic [3:0] U, V, X,
3               output logic [7:0] Y);
4
5    // bitweise Verknüpfung
6    assign U = A & B;   // U[0] = (A[0] & B[0]), U[1] = (A[1] & B[1]), ...
7    assign V = A ^ B;   // V[0] = (A[0] ^ B[0]), V[1] = (A[1] ^ B[1]), ...
8
9    // (unsigned) Arithmetik
10   assign X = A + B;
11   assign Y = A * B;
12
13 endmodule
```



Synthese von Operationen auf Vektoren



Vektoren $\neq$ Arrays

vectors_vs_arrays.sv

```
1 logic [3:0] A;          // 4 bit Vektor
2 logic A [1:0];          // Array mit 2 bits
3 logic [3:0] A [1:0];    // Array mit 2 4 bit Vektoren
```

- SystemVerilog unterstützt Vektoren und Arrays
- Arrays unterstützen jedoch weniger Operationen, beispielsweise keine bitweisen oder arithmetischen Operationen

$\Rightarrow$ Jetzt: Nur Vektoren

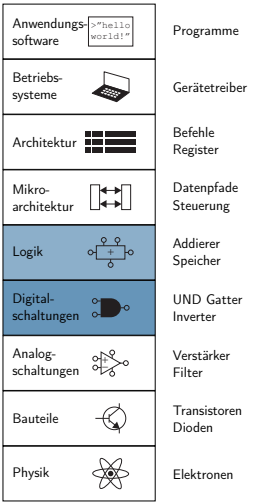
$\Rightarrow$ Später: Arrays für Speicher

Agenda

1 Arithmetische Grundschaltungen

2 SystemVerilog Datentypen

3 SystemVerilog für kombinatorische Logik (Forts.)



gates.sv

```
1  module gates (input  logic [3:0] a, b, // 4 bit Vektoren
2                output logic [3:0] y1, y2, y3, y4, y5);
3
4     /* Fünf unterschiedliche Logikgatter
5        mit zwei Eingängen, jeweils 4 bit Vektoren */
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
```

gates_logic.sv

```
1  module gates_logic (input  logic [3:0] a, b, // 4 bit Vektoren
2                      output logic [3:0] y,
3                      output logic       z);
4
5    assign y = a & b;  // y[0] = (a[0] & b[0]), y[1] = (a[1] & b[1]),
6                       // y[2] = (a[2] & b[2]), y[3] = (a[3] & b[3])
7    assign z = a && b; // z = (a[0] | a[1] | a[2] | a[3])
8                       //   & (b[0] | b[1] | b[2] | b[3])
9
10 endmodule
```

Reduktionsoperatoren (unär)

and8.sv

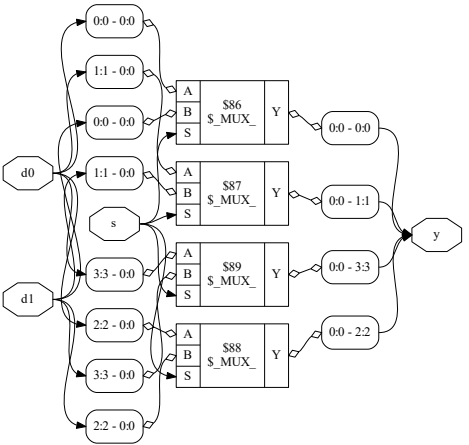
```
1  module and8 (input logic [7:0] a, output logic y);
2
3      // assign y = a[7] & a[6] & a[5] & a[4] &
4      //            a[3] & a[2] & a[1] & a[0];
5      assign y = &a;
6
7  endmodule
```

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

MUX auf Vektoren

mux2x4.sv

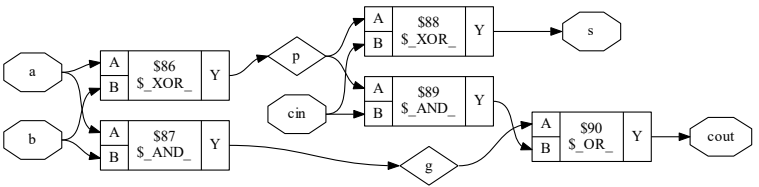
```
1  module mux2x4
2   (input  logic [3:0] d0,d1,
3    input  logic       s,
4    output logic [3:0] y);
5
6    assign y = s ? d1 : d0;
7
8  endmodule
```



Interne Verbindungsknoten (Signale)

fulladder.sv

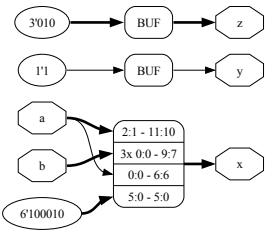
```
1  module fulladder(input  logic a, b, cin,
2                   output logic s, cout);
3
4    logic  p, g;   // interne Verbindungsknoten
5    assign p = a ^ b;
6    assign g = a & b;
7    assign s = p ^ cin;
8    assign cout = g | (p & cin);
9
10 endmodule
```



Konkatenation

concat.sv

```
1 module concat(input  logic [2:0]  a, b,
2          output logic [11:0] x, output logic y,  output logic [2:0]  z);
3
4    assign x = {a[2:1], {3{b[0]}}, a[0], 6'b100010};
5    // x = a[2] a[1] b[0] b[0] b[0] a[0] 1 0 0 0 1 0
6
7    assign {y, z} = 4'b1010;
8    // y = 1, z = 010
9 endmodule
```



Bindung von Operatoren (Präzedenz)

ENCRYPTO
CRYPTOGRAPHY AND
PRIVACY ENGINEERING

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

Wiederholung: Assign Statement

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
```

- auch *continuous assignment* genannt
- Linke Seite (LHS, "left hand side"): Variable oder Port
- Rechte Seite (RHS, "right hand side"): logischer Ausdruck
- Zuweisung, wenn der Wert von RHS sich ändert

Alternative für kombinatorische Logik: Der always_comb Block

ex_always_comb.sv

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

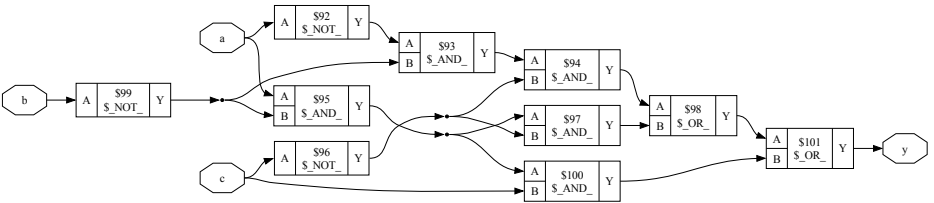
- always_comb <instruction>
  - Zuweisung ebenfalls, wenn sich der Wert von RHS ändert
  - LHS Variablen dürfen nicht von anderen Blöcken geschrieben werden

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
```

ex_always_comb.sv

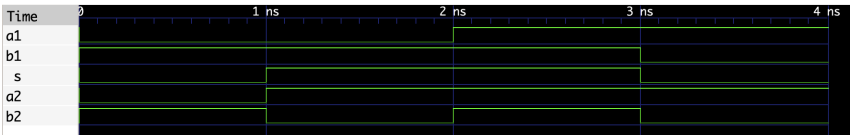
```
1  module ex_always_comb(input logic a, b, c, output logic y);
2
3    always_comb y = ~a & ~b & ~c | a & ~b & ~c | a & ~b & c;
4
5  endmodule
```



if-else-Konstrukt
Anwendung des always_comb Block

swap.sv

```
1  module swap(input  logic a1, b1, s,
2              output logic a2, b2);
3
4    always_comb begin
5      if (s) begin
6        b2 = a1;
7        a2 = b1;
8      end else begin
9        b2 = b1;
10       a2 = a1;
11     end
12   end
13
14 endmodule
```



- if-else darf nur in always_comb und always/always_ff/... Blöcken (später in der Vorlesung) verwendet werden

Synthese von `if-else`
Mehr Abstraktion, gleiches Ergebnis

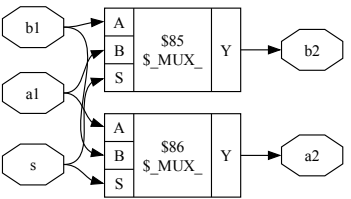
ENCRYPTO
CRYPTOGRAPHY AND PRIVACY ENGINEERING

swap.sv

```
1  module swap(input  logic a1, b1, s,
2              output logic a2, b2);
3
4    always_comb begin
5      if (s) begin
6        b2 = a1;
7        a2 = b1;
8      end else begin
9        b2 = b1;
10       a2 = a1;
11     end
12   end
13
14 endmodule
```

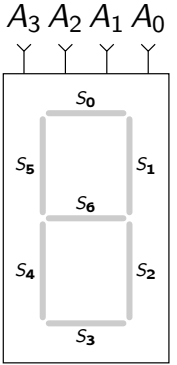
swap_assign.sv

```
1  module swap_assign(input  logic a1, b1, s,
2                     output logic a2, b2);
3
4    assign b2 = s ? a1 : b1;
5    assign a2 = s ? b1 : a1;
6
7  endmodule
```



sevenseg.sv

```
1  module sevenseg (input  logic [3:0] A,
2                   output logic [6:0] S);
3    always_comb case (A)
4            0: S = 7'b011_1111;
5            1: S = 7'b000_0110;
6            2: S = 7'b101_1011;
7            3: S = 7'b100_1111;
8            4: S = 7'b110_0110;
9            5: S = 7'b110_1101;
10           6: S = 7'b111_1101;
11           7: S = 7'b000_0111;
12           8: S = 7'b111_1111;
13           9: S = 7'b110_1111;
14     default: S = 7'b000_0000;
15   endcase
16 endmodule
```



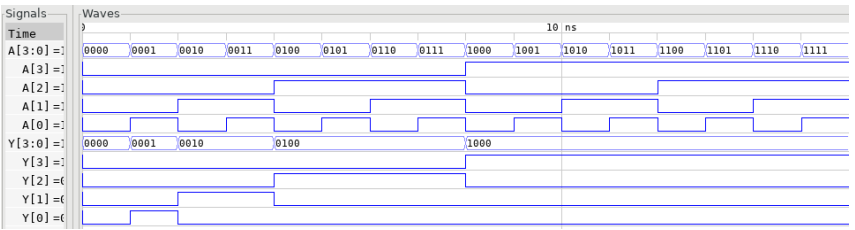
- case darf nur in `always_comb` und `always` Blöcken (später in der Vorlesung) verwendet werden
- für kombinatorische Logik müssen alle Eingabe-Optionen abgedeckt werden
- explizit oder per `default` („alle anderen“)

Fallunterscheidungen (casez)
Beispiel: Prioritätsencoder

ENCRYPTO
CRYPTOGRAPHY AND PRIVACY ENGINEERING

priority_encoder.sv

```
1  module priority_encoder(input  logic [3:0] A,
2                          output logic [3:0] Y);
3    always_comb casez(A)     // casez erlaubt don't cares
4      4'b1???: Y = 4'b1000; // ? = don't care
5      4'b01??: Y = 4'b0100;
6      4'b001?: Y = 4'b0010;
7      4'b0001: Y = 4'b0001;
8      default: Y = 4'b0000;
9    endcase
10 endmodule
```

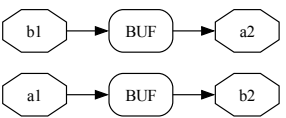


Eigenschaften von `assign` und `always_comb`

- werden bei Simulation immer ausgeführt, wenn sich ein Signal auf der rechten Seite ändert
- erlauben nur kombinatorische Logik
- Reihenfolge im Quellcode *oft* nicht relevant
  - nebenläufige Signalzuweisungen („concurrent signal assignments“)
  - **Aber:** *Blockierende Signalzuweisungen* (`a = b`) **innerhalb** von Blöcken (`begin`/`end`) werden nacheinander ausgeführt.

always_swap.sv

```
1  module always_swap(input  logic a1, b1,
2                     output logic a2, b2);
3    always_comb begin
4      a2 = a1;
5      b2 = b1;
6      // swap a2 and b2 step by step
7      logic tmp;
8      tmp = a2;
9      a2 = b2;
10     b2 = tmp;
11   end
12 endmodule
```



Zusammenfassung und Ausblick

1. Arithmetische Grundschaltungen
2. SystemVerilog Datentypen
3. SystemVerilog für kombinatorische Logik (Forts.)

nächste Vorlesung beinhaltet

- Sequentielle Schaltungen
- Speicherelemente: Latches und Flip-Flops
- Synchrone sequentielle Logik

Hausaufgabe C zu Vorlesungen 05 und 06 muss bis nächste Woche Freitag 23:59 abgegeben werden. Wöchentliches Moodle-Quiz nicht vergessen!