

Digitaltechnik

Wintersemester 2017/2018

12. Vorlesung



TECHNISCHE
UNIVERSITÄT
DARMSTADT

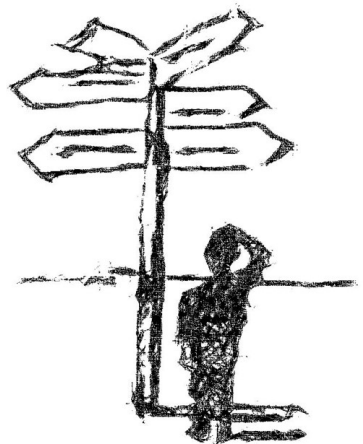




1. Einleitung
2. Mehr SystemVerilog für Testumgebungen
3. SystemVerilog Abschluss
4. Arithmetische Grundsaltungen
5. Zusammenfassung

Agenda

1. Einleitung
2. Mehr SystemVerilog für Testumgebungen
3. SystemVerilog Abschluss
4. Arithmetische Grundsaltungen
5. Zusammenfassung





- ▶ Lehrhospitation in V11
 - ▶ drei Zuhörer-Kategorien: aktiv, passiv, unbeteiligt
 - ▶ konkrete Verbesserungsvorschläge in V12 evaluieren
- ⇒ Feedback benötigt

Rückblick auf die letzte Vorlesung



TECHNISCHE
UNIVERSITÄT
DARMSTADT

- ▶ Mehr SystemVerilog für kombinatorische Logik
- ▶ SystemVerilog für sequentielle Logik
- ▶ SystemVerilog für Zustandsautomaten
- ▶ SystemVerilog für **parametrisierte Module**
- ▶ SystemVerilog für **Testumgebungen**



Harris 2013
Kap. 4.4-4.8

Wiederholungs-Bedarf laut Moodle Abfrage und Übungen



TECHNISCHE
UNIVERSITÄT
DARMSTADT

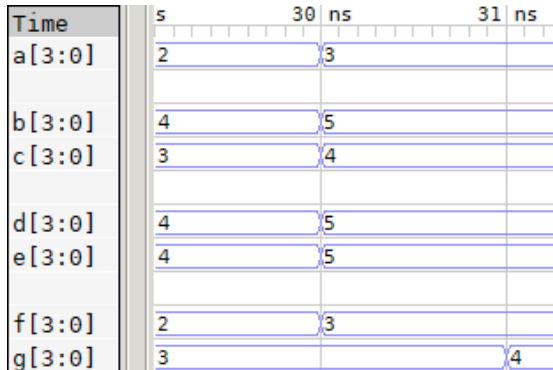
- ▶ (Nicht-)Blockierende Zuweisungen
- ▶ Ripple-Carry Adder

Wiederholung: (Nicht-)Blockierende Zuweisungen



TECHNISCHE
UNIVERSITÄT
DARMSTADT

```
1  logic [3:0] a = 0;  
2  always #10 a++;  
  
3  
4  logic [3:0] b,c,d,  
5             e,f,g;  
6  always @a begin a = 3  
7      b <= a+2;    b = 4 15  
8      c <= b;      c = 3 14  
9  
10     d = a+2;     d = 5  
11     e = d;       e = 5  
12  
13     f = c;  
14     #1;  
15     g = c;  
16 end
```



Wiederholung: Parametrisierte Module


Multiplexer mit Bitpacking



TECHNISCHE
UNIVERSITÄT
DARMSTADT

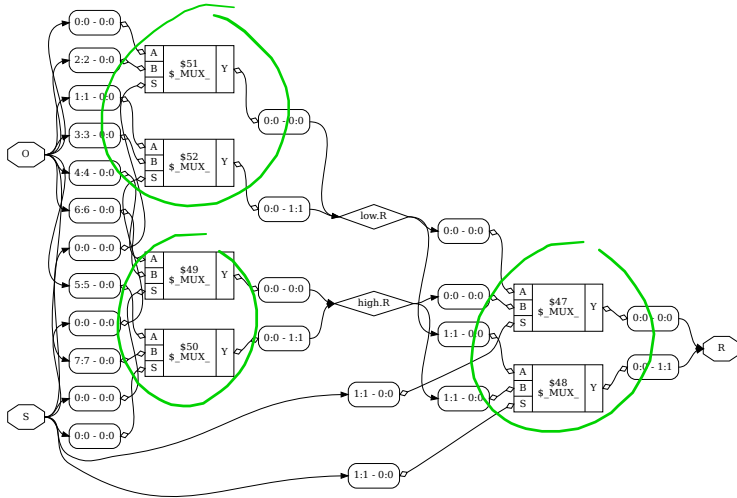
```
1 module lmux #(parameter WIDTH = 2, parameter DEPTH = 4)
2     (input logic [WIDTH*DEPTH-1:0] O,
3     input logic [$clog2(DEPTH)-1:0] S,
4     output logic [WIDTH-1:0] R);
5
6     localparam SW = $clog2(DEPTH);
7
8     generate
9         if (DEPTH > 2) begin
10             logic [WIDTH-1:0] rh, rl;
11             lmux #(WIDTH, DEPTH/2) high (O[ DEPTH *WIDTH-1 : (DEPTH/2)*WIDTH],
12             S[SW-2:0],
13             rh);
14             lmux #(WIDTH, DEPTH/2) low (O[(DEPTH/2)*WIDTH-1 : 0*WIDTH],
15             S[SW-2:0],
16             rl);
17             assign R = S[SW-1] ? rh : rl;
18         end else begin
19             assign R = S ? O[WIDTH+:WIDTH] : O[0+:WIDTH];
20         end
21     endgenerate
22 endmodule
```

2. W. 1 : 1. W
1. W. 1 : 0



Wiederholung: Parametrisierte Module

Multiplexer mit Bitpacking



Überblick der heutigen Vorlesung



TECHNISCHE
UNIVERSITÄT
DARMSTADT

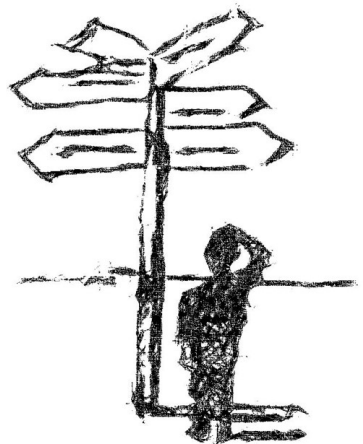
- ▶ Mehr SystemVerilog für Testumgebungen
- ▶ SystemVerilog Abschluss
- ▶ Arithmetische Grundsaltungen



Harris 2013
Kap. 4.4-5.2
Seite 214 - 248

Agenda

1. Einleitung
2. Mehr SystemVerilog für Testumgebungen
3. SystemVerilog Abschluss
4. Arithmetische Grundsaltungen
5. Zusammenfassung



Testumgebungen (testbenches)



TECHNISCHE
UNIVERSITÄT
DARMSTADT

- ▶ Modul ohne Ports
- ▶ Stimuli erzeugen (Takt, Reset, Eingabedaten)
- ▶ „unit under test“ instantiieren
- ▶ Ausgabedaten und Timing verifizieren
 - ▶ erschöpfend oder zufällig
 - ▶ Grenzfälle abdecken
- ▶ wird nicht synthetisiert
- ▶ speziell für Icarus-Verilog
 - ▶ VCD-Datei öffnen
 - ▶ beobachtete Signale konfigurieren
 - ▶ Simulation beenden

```
1 timescale 1 ns / 10 ps
2 module add_tb;
3     logic clk = 0, reset = 1;
4     always #(0.5/10) clk <= ~clk;
5     initial @(posedge clk) rst <= 0;
6
7     logic [4:0] a,b,y;
8     add_uut(clk,rst,a,b,y);
9
10    initial begin
11        $dumpfile("add_tb.vcd");
12        $dumpvars;
13
14        for (int i=0; i<256; i++) begin
15            {a,b} <= i;
16            @(posedge clk);
17            if (y != a+b) $display("error");
18        end
19
20        $display("FINISHED add_tb");
21        $finish;
22    end
23 endmodule
```



- ▶ `$display(<format>, <values>*)`;
- ▶ ähnlich `printf` in C und Java
- ▶ wichtige Platzhalter:
 - ▶ `%d` `%b` `%h` für dezimal, binär, hexadezimal
 - ▶ `%m` für Modulname (implizites Argument), bspw. `add_tb.uut`
 - ▶ `%t` für Zeit (mit Einheit)



- ▶ `$display(<format>, <values>*)`;
- ▶ ähnlich `printf` in C und Java
- ▶ wichtige Platzhalter:
 - ▶ `%d` `%b` `%h` für dezimal, binär, hexadezimal
 - ▶ `%m` für Modulname (implizites Argument), bspw. `add_tb.uut`
 - ▶ `%t` für Zeit (mit Einheit)
- ▶ `$timeformat(-9, 1, "ns", 8)`; zum Einstellen des Zeitformats
 - ▶ Skalierung auf 10^{-9}
 - ▶ eine Nachkommastellen
 - ▶ Einheiten-Suffix
 - ▶ Anzahl der anzuzeigenden Zeichen



- ▶ `$display(<format>, <values>*)`;
- ▶ ähnlich `printf` in C und Java
- ▶ wichtige Platzhalter:
 - ▶ `%d %b %h` für dezimal, binär, hexadezimal
 - ▶ `%m` für Modulname (implizites Argument), bspw. `add_tb.uut`
 - ▶ `%t` für Zeit (mit Einheit)
- ▶ `$timeformat(-9, 1, "ns", 8)`; zum Einstellen des Zeitformats
 - ▶ Skalierung auf 10^{-9}
 - ▶ eine Nachkommastellen
 - ▶ Einheiten-Suffix
 - ▶ Anzahl der anzuzeigenden Zeichen
- ▶ bspw.: `$display("%t@m: y = %0d", $time, y)`;
erzeugt: `3.0 ns@add_tb: y = 5`

Auslesen der Simulationszeit



TECHNISCHE
UNIVERSITÄT
DARMSTADT

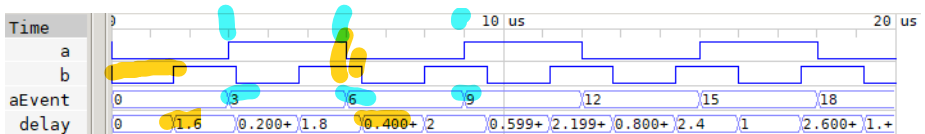
- ▶ `$time` - aktuelle Systemzeit als ganze Zahl (`int`)
- ▶ `$realtime` - aktuelle Systemzeit als rationale Zahl (`real`)

Auslesen der Simulationszeit

- ▶ \$time - aktuelle Systemzeit als ganze Zahl (int)
- ▶ \$realtime - aktuelle Systemzeit als rationale Zahl (real)
- ▶ Anwendungsbeispiel: Zeitspanne zwischen zwei Signalfanken bestimmen

deltat.sv

```
1 'timescale 1 us / 10 ns
2 module deltat;
3     logic a=0;    always #3    a <= ~a;
4     logic b=0;    always #1.6 b <= ~b;
5
6     real aEvent;  always @a    aEvent <= $realtime;
7     real delta;  always @b    delta <= $realtime - aEvent;
8 endmodule
```



- ▶ Erstellen effizienter Testpläne ist nicht trivial
 - ▶ Abdeckung maximieren (gezielt vs. zufällig)
 - ▶ Wiederverwendbarkeit maximieren
 - ▶ Überlappung minimieren



Chris Spear:
SystemVerilog
for Verification
(Springer)



- ▶ Erstellen effizienter Testpläne ist nicht trivial
 - ▶ Abdeckung maximieren (gezielt vs. zufällig)
 - ▶ Wiederverwendbarkeit maximieren
 - ▶ Überlappung minimieren
- ▶ Multi-Domänen Cosimulation von Hardware und
 - ▶ Software
 - ▶ Event-basierte Kommunikationsprotokolle
 - ▶ kontinuierliche physikalische Prozesse



Chris Spear:
SystemVerilog
for Verification
(Springer)



- ▶ Erstellen effizienter Testpläne ist nicht trivial
 - ▶ Abdeckung maximieren (gezielt vs. zufällig)
 - ▶ Wiederverwendbarkeit maximieren
 - ▶ Überlappung minimieren
- ▶ Multi-Domänen Cosimulation von Hardware und
 - ▶ Software
 - ▶ Event-basierte Kommunikationsprotokolle
 - ▶ kontinuierliche physikalische Prozesse
- ▶ Testgetriebene Entwicklung (TDD)



Chris Spear:
SystemVerilog
for Verification
(Springer)

- ▶ Erstellen effizienter Testpläne ist nicht trivial
 - ▶ Abdeckung maximieren (gezielt vs. zufällig)
 - ▶ Wiederverwendbarkeit maximieren
 - ▶ Überlappung minimieren
 - ▶ Multi-Domänen **Cosimulation** von Hardware und
 - ▶ Software
 - ▶ Event-basierte Kommunikationsprotokolle
 - ▶ kontinuierliche physikalische Prozesse
 - ▶ Testgetriebene Entwicklung (TDD)
- ⇒ SystemVerilog bringt hier viele Verbesserungen
- ▶ file IO
 - ▶ assertions, implications
 - ▶ (constrained) random
 - ▶ Klassen, Vererbung, Schnittstellen
 - ▶ Direct Programming Interface (DPI)



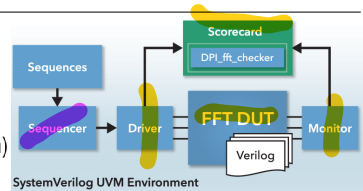
Chris Spear:
SystemVerilog
for Verification
(Springer)

Universal Verification Methodology (UVM)



TECHNISCHE
UNIVERSITÄT
DARMSTADT

- ▶ SystemVerilog **Klassenbibliothek**
 - ▶ Monitor (uut Schnittstelle lesen)
 - ▶ Driver (uut Schnittstelle beobachten)
 - ▶ Sequencer (**Transaktionssequenz** aufbereiten)
 - ▶ Agent (Monitor + Driver + Sequencer)
 - ▶ Scoreboard (sammelt Verifikationsergebnisse)



- ▶ Vereinheitlichtes Vorgehen durch vordefinierte **Phasen**
 - ▶ **build**
 - ▶ **connect**
 - ▶ end of elaboration
 - ▶ start of simulation
 - ▶ run
 - ▶ **extract/check/report**

⇒ wird inzwischen von vielen Design-Tools unterstützt

1 min Murmelphase



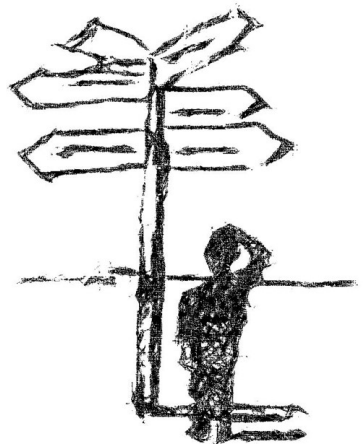
TECHNISCHE
UNIVERSITÄT
DARMSTADT

- ▶ Warum haben Testbenches keine Ports?
- ▶ Wieviele always Blöcke werden zum Überprüfen der Setup- und Hold-Bedingungen an Registern benötigt?



Agenda

1. Einleitung
2. Mehr SystemVerilog für Testumgebungen
3. SystemVerilog Abschluss
4. Arithmetische Grundsaltungen
5. Zusammenfassung





- ▶ alle SystemVerilog Konstrukte sind grundsätzlich simulierbar
- ▶ aber nicht alle Simulatoren unterstützen den kompletten Sprachstandard
- ▶ nicht synthetisierbar sind
 - ▶ Signalinitialisierung bei der Deklaration
 - ▶ initial Blöcke
 - ▶ explizite Verzögerungen (per #)
 - ▶ die meisten Funktionen wie \$display, \$time, \$clog2
 - ▶ real Signale

- ▶ SystemVerilog ist Weiterentwicklung von Verilog (für Verifikation)
- ▶ Verilog immer noch weiter verbreitet
- ▶ im Rahmen der Veranstaltung nur wenige Unterschiede:
 - ▶ separater Datentypen statt `logic`
 - ▶ `wire` für Zuweisung per `assign`
 - ▶ `reg` für Zuweisungen in `always` Blöcken
 - ▶ keine spezifischen `always` Blöcke für
 - ▶ Flip-Flops: `always @(posedge clk)`
 - ▶ Latches: `always @(clk, d)`
 - ▶ kombinatorische Logik: `always @*`

⇒ für Studierende idR. leichter verständlich

- ▶ Viele Sprachkonstrukte können in kurzer Einführung nicht behandelt werden

- ▶ Tasks, Funktionen und programm
- ▶ Klassen und Vererbung
- ▶ Verifikationsunterstützung
- ▶ fork und join
- ▶ Events
- ▶ Präprozessor
- ▶ ...

⇒ bei tieferem Interesse weitere Literatur verwenden



WWW

2 min Murmelphase



TECHNISCHE
UNIVERSITÄT
DARMSTADT

- ▶ Was ist der Unterschied zwischen `localparam` und `parameter`?
- ▶ Wozu werden Hardware-Beschreibungssprachen verwendet?
- ▶ Welche Konzepte bietet SystemVerilog zum Beherrschen von Design-Komplexität



Wiederholung: Schichtenmodell eines Computers

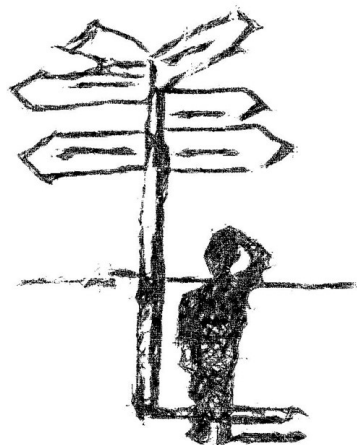


TECHNISCHE
UNIVERSITÄT
DARMSTADT

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

Agenda

1. Einleitung
2. Mehr SystemVerilog für Testumgebungen
3. SystemVerilog Abschluss
4. Arithmetische Grundsaltungen
5. Zusammenfassung



Ripple-Carry-Adder (RCA)



TECHNISCHE
UNIVERSITÄT
DARMSTADT

$$\begin{array}{rcccccc} & & 1 & 0 & 1 & 1 & \\ & & 1 & 0 & 1 & 1 & \text{Summand} \\ + & & 1 & 0 & 1 & 1 & \text{Summand} \\ \hline & 1 & 0 & 1 & 1 & 0 & \text{Summe} \end{array}$$

Ripple-Carry-Adder (RCA)



TECHNISCHE
UNIVERSITÄT
DARMSTADT

	1	0	1	1	0	Übertrag
		1	0	1	1	Summand
+		1	0	1	1	Summand
<hr/>						
=	1	0	1	1	0	Summe

Ripple-Carry-Adder (RCA)

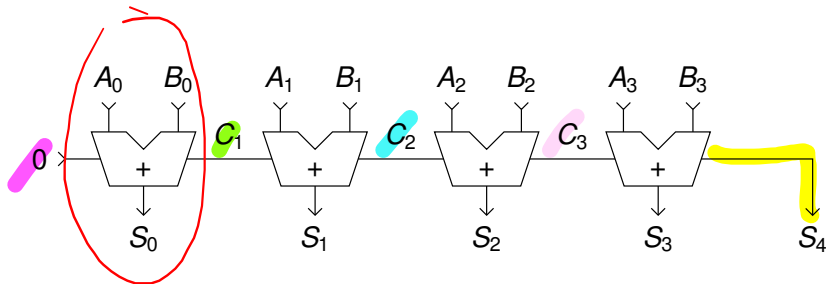


TECHNISCHE
UNIVERSITÄT
DARMSTADT

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

Ripple-Carry-Adder (RCA)

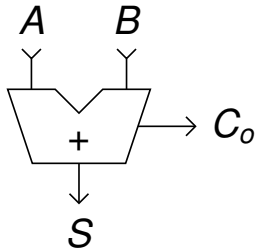
	1	0	1	1	0	Übertrag	C_4	C_3	C_2	C_1	C_0
		1	0	1	1	Summand		A_3	A_2	A_1	A_0
+		1	0	1	1	Summand		B_3	B_2	B_1	B_0
=	1	0	1	1	0	Summe	S_4	S_3	S_2	S_1	S_0



Halbaddierer



TECHNISCHE
UNIVERSITÄT
DARMSTADT

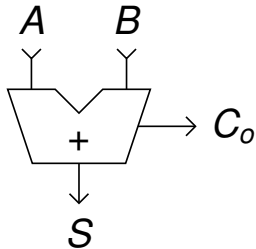


A	B	C_o	S
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0

Halbaddierer



TECHNISCHE
UNIVERSITÄT
DARMSTADT

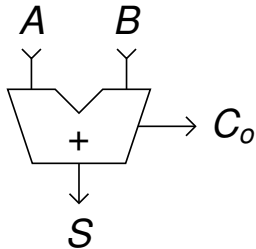


A	B	C_o	S
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0

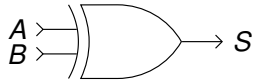
Halbaddierer



TECHNISCHE
UNIVERSITÄT
DARMSTADT



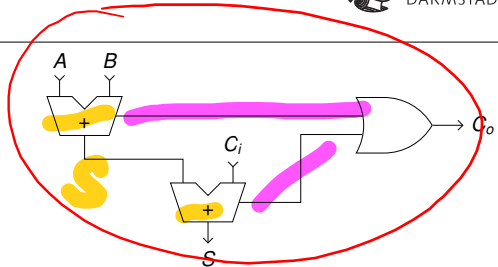
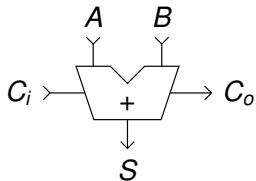
<i>A</i>	<i>B</i>	<i>C_o</i>	<i>S</i>
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0



Volladdierer



TECHNISCHE
UNIVERSITÄT
DARMSTADT

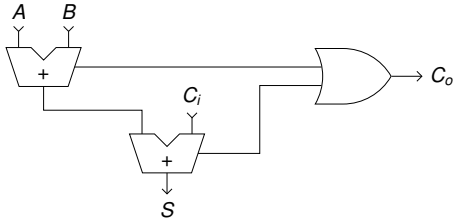
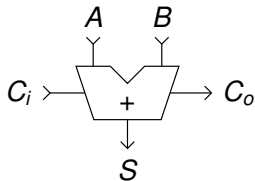


A	B	C_i	C_o	S
0	0	0		
0	0	1		
0	1	0		
0	1	1		
1	0	0		
1	0	1		
1	1	0		
1	1	1		

Volladdierer



TECHNISCHE
UNIVERSITÄT
DARMSTADT

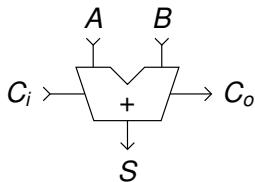


A	B	C _i	C _o	S
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

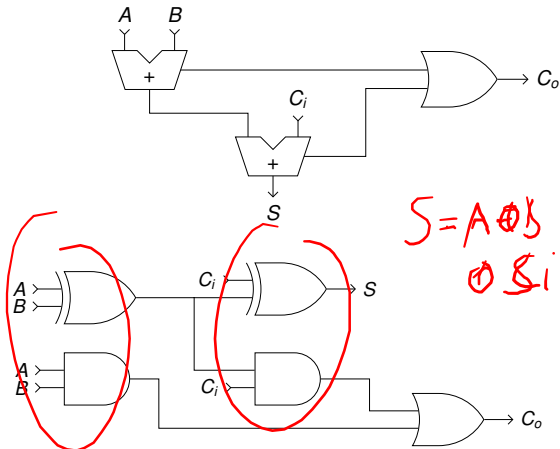
Volladdierer



TECHNISCHE
UNIVERSITÄT
DARMSTADT



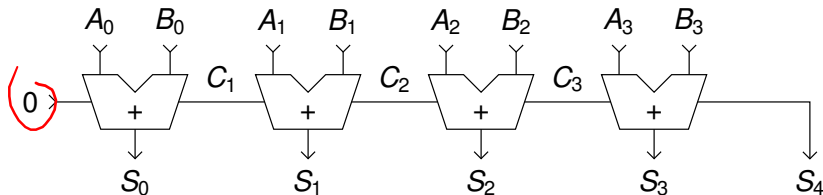
A	B	C _i	C _o	S
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1



Ripple-Carry-Adder



TECHNISCHE
UNIVERSITÄT
DARMSTADT



- ▶ Überträge werden über Kette von 1 bit Volladdierern vom LSB zum MSB weitergegeben
- ⇒ langer kritischer Pfad (steigt linear mit Bitbreite)
- ⇒ schnellere Addierer müssen Übertragskette aufbrechen (benötigen dafür mehr Hardware)
 - ▶ Conditional Sum Adder (CSA)
 - ▶ Carry-Lookahead Adder (CLA)
 - ▶ Prefix-Addierer



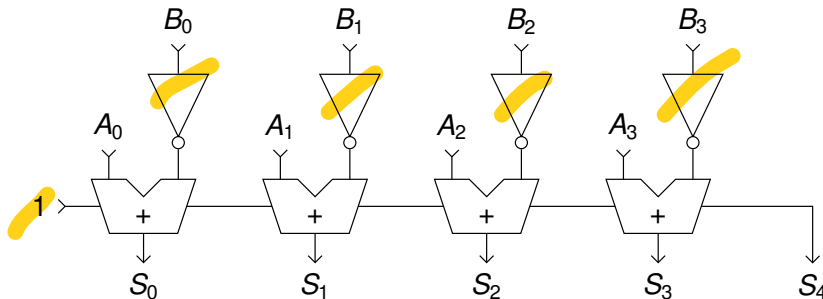
- ▶ kann mit Addition und Negation realisiert werden: $A - B = A + (-B)$
- ▶ Negation im Zweierkomplement: Komplement und Inkrement

Subtrahierer



TECHNISCHE
UNIVERSITÄT
DARMSTADT

- ▶ kann mit Addition und Negation realisiert werden: $A - B = A + (-B)$
 - ▶ Negation im Zweierkomplement: Komplement und Inkrement
- ⇒ RCA mit NOT-Gatter an B -Eingängen und 1 am ersten C_i



Vergleich: Kleiner als



TECHNISCHE
UNIVERSITÄT
DARMSTADT

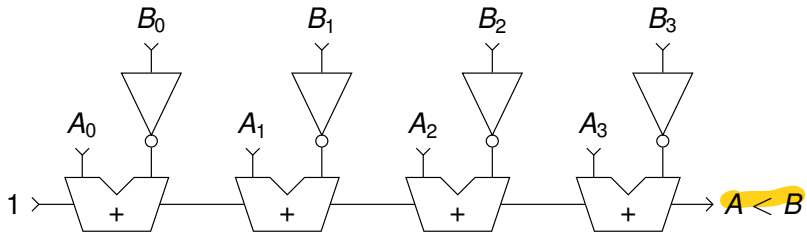
- ▶ kann mit Subtraktion realisiert werden: $A < B \Leftrightarrow A - B < 0$

Vergleich: Kleiner als



TECHNISCHE
UNIVERSITÄT
DARMSTADT

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

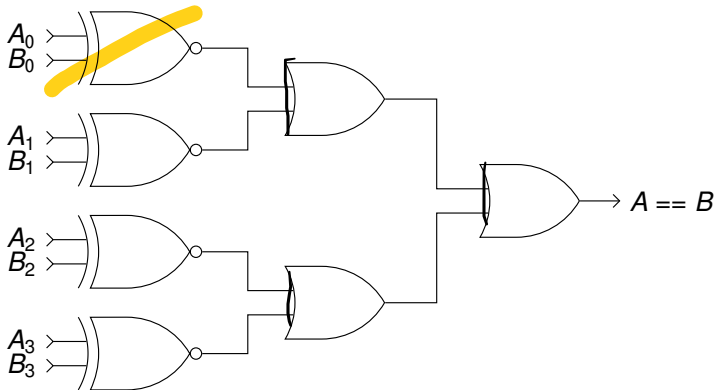


Vergleich: Gleichheit

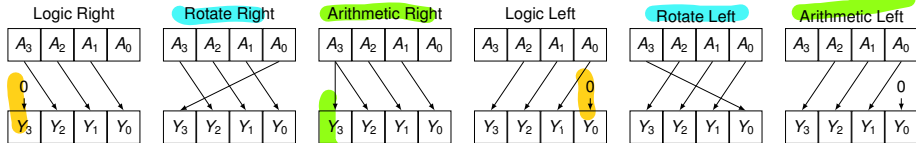


TECHNISCHE
UNIVERSITÄT
DARMSTADT

► Bitweise XNOR und OR-Baum



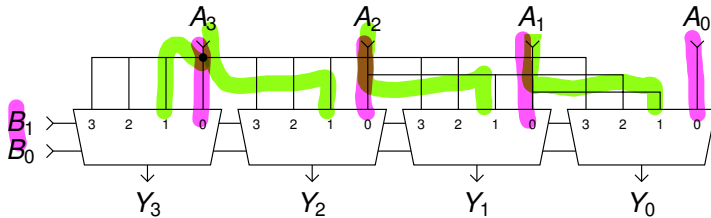
- ▶ A um B Stellen nach links/rechts verschieben
- ▶ Strategien zum Auffüllen der freien Stellen:
 - ▶ **logischer** Rechts- oder Linksshift: Auffüllen mit Nullen
 - ▶ **umlaufender** Rechts- oder Linksshift: Auffüllen mit den aus der anderen Seite herausfallenden Bits (**Rotation**)
 - ▶ **arithmetischer**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



Barrel-Shifter mit variabler Shift-Weite



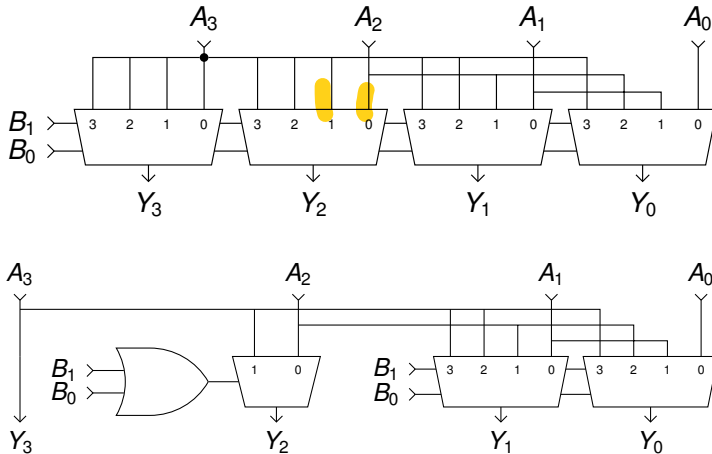
TECHNISCHE
UNIVERSITÄT
DARMSTADT



Barrel-Shifter mit variabler Shift-Weite



TECHNISCHE
UNIVERSITÄT
DARMSTADT



Arithmetische Shifter als Multiplizierer und Dividierer



TECHNISCHE
UNIVERSITÄT
DARMSTADT

- ▶ Schieben um n Stellen nach links multipliziert den Zahlenwert mit 2^n
 - ▶ $00001_2 \lll 3 = 01000_2 (1 \cdot 2^3 = 8)$
 - ▶ $11101_2 \lll 2 = 10100_2 (-3 \cdot 2^2 = -12)$

Arithmetische Shifter als Multiplizierer und Dividierer



TECHNISCHE
UNIVERSITÄT
DARMSTADT

- ▶ Schieben um n Stellen nach links multipliziert den Zahlenwert mit 2^n

- ▶ $00001_2 \lll 3 = 01000_2 (1 \cdot 2^3 = 8)$
- ▶ $11101_2 \lll 2 = 10100_2 (-3 \cdot 2^2 = -12)$

⇒ Multiplikation mit Konstanten kann aus Shift und Addition zusammengesetzt werden:

- ▶ $a \cdot 6 = a \cdot 110_2 = (a \lll 2) + (a \lll 1)$

Arithmetische Shifter als Multiplizierer und Dividierer



TECHNISCHE
UNIVERSITÄT
DARMSTADT

- ▶ Schieben um n Stellen nach links multipliziert den Zahlenwert mit 2^n

- ▶ $00001_2 \lll 3 = 01000_2 (1 \cdot 2^3 = 8)$
- ▶ $11101_2 \lll 2 = 10100_2 (-3 \cdot 2^2 = -12)$

⇒ Multiplikation mit Konstanten kann aus Shift und Addition zusammengesetzt werden:

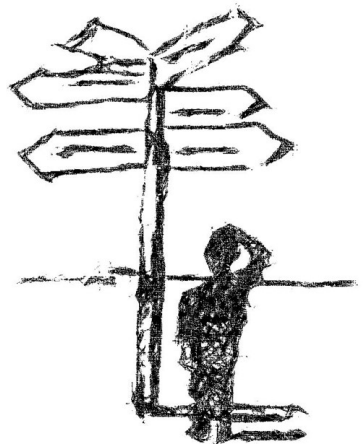
- ▶ $a \cdot 6 = a \cdot 110_2 = (a \lll 2) + (a \lll 1)$

- ▶ Schieben um n Stellen nach rechts dividiert den Zahlenwert durch 2^n

- ▶ $010000_2 \ggg 4 = 000001_2 (16/2^4 = 1)$
- ▶ $100000_2 \ggg 2 = 111000_2 (-32/2^2 = -8)$

Agenda

1. Einleitung
2. Mehr SystemVerilog für Testumgebungen
3. SystemVerilog Abschluss
4. Arithmetische Grundsaltungen
5. Zusammenfassung



Verbesserung durch Lehrevaluation?



TECHNISCHE
UNIVERSITÄT
DARMSTADT

- ▶ ausführlichere Einordnung der Vorlesung in Gesamtkontext
- ▶ Diskussionspausen am Ende von Themenkomplexen
- ▶ ausführlichere Überleitung zwischen Themenkomplexen



- ▶ Mehr SystemVerilog für Testumgebungen
- ▶ SystemVerilog Ausblick
- ▶ Arithmetische Grundsaltungen

- ▶ Nächste Vorlesung behandelt
 - ▶ Organisation eines einfachen Modellrechners