TECHNISCHE UNIVERSITÄT DRESDEN

Elektrotechnik und Informationstechnik, Stiftungsprofessur hochparallele VLSI Systeme und Neuromikroelektronik

Schaltkreis- und Systementwurf

Hardwarebeschreibungssprache Verilog

FAKULTÄT **ELEKTROTECHNIK UND INFORMATIONSTECHNIK**

DRESDEN concept
Exzellenz aus Wissenschaft und Kultur

Inhalt

- Verilog HDL
  - Übersicht
  - Grundlagen
  - Prozedurale Zuweisungen
  - Hierarchische Elemente (Module, Funktionen, Tasks)
  - Gate-Level Modellierung
  - Verhaltensmodellierung
  - Systemfunktionen
- Schaltungssimulation und Verifikation

Ziele dieses Vorlesungsteils

- Dieser Vorlesungsteil wird:
  - Grundkonzepte der Verilog HDL vorstellen
  - Methoden zur Beschreibung und Modellierung digitaler Schaltungsblöcke vermitteln
  - Verifikationsstrategien vermitteln
  - → Grundlage für den Einsatz von Verilog im Praktikum
  - Anregung zum Selbststudium geben

- Dieser Vorlesungsteil wird **NICHT:**
  - eine komplette Sprachreferenz der Verilog HDL bereitstellen
  - die Übungen im Praktikum ersetzen

Verilog - Übersicht

- Hardwarebeschreibungssprache (engl. Hardware Description Language HDL)
- Beschreibung von digitalen Schaltungen in maschinenlesbarer Textform
- Anwendung zur
  - Modellierung
  - Simulation
  - Verifikation
  - Synthese
- Möglichkeit zur Beschreibung von Hardware-Eigenschaften, z.B.
  - zeitliche Abläufe
  - Gleichzeitigkeit

Die Hardwarebeschreibungssprache Verilog

- 1983/84 entwickelt von Phil Moorby (Gateway Design Automation) als **Simulationssprache** „Verilog"
- 1985 erweiterter Sprachumfang und Simulator „Verilog-XL"
- 1988 Synthesewerkzeuge basierend auf Verilog verfügbar (Synopsys Design Compiler)
- 1990 Übernahme durch Cadence Design Systems
- 1993 85% aller ASIC Chipentwürfe in Verilog
- seit 1995 freier Standard → IEEE Standard 1364-1995 (Verilog-95)
- 2001 Erweiterung IEEE Standard 1364-2001, „Verilog2001"

- Standard
  - IEEE Standard Verilog® Hardware Description Language; IEEE Std 1364-2005 (Revision of IEEE Std 1364-2001)
    - *verfügbar über IEEE Explore*
- Bücher
  - HDL Chip Design; Douglas J. Smith; Doone Publications; 1996
  - The Verilog hardware description language; Thomas, Donald E. ; Moorby, Philip R.; Springer; 2002

- Online Tutorials
  - http://www.asic-world.com/verilog/veritut.html
  - http://www.ece.umd.edu/courses/enee359a/verilog_tutorial.pdf

Abstraktionsebenen der Verilog Beschreibungen

Abstraktion

- Algorithmisch/Verhalten
  - High-level Design Konstrukte

```
…
real a,b,c;
always c = #1 a*b;
…
```

- Register-Transfer-Level (RTL)
  - Datenfluss zwischen Registern
  - Grundlage für die Synthese

```
…
reg [7:0] a, b;
always @(posedge clk) begin
    a<=b+1;
end
…
```

- Gatter-Level
  - Beschreibung von Logikgattern und deren Verbindungen

```
…
wire out, in1, in2;
nor(out,in1,in2);
…
```

- Schalter-Level
  - Beschreibung von Schaltern (Transistoren) sowie Speicherknoten

```
…
wire g, s, d;
nmos(d,s,g);
…
```

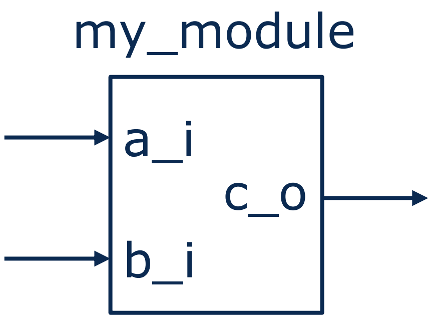
Gemeinsame Simulation aller Abstraktionsebenen möglich!

Verilog Erweiterungen

- VerilogA
  - Sprache zur Modellierung von analogen (zeit- und wertkontinuierlich) Komponenten
  - keine reinen digitalen Verilog Konstrukte möglich
  - Simulation erfordert reinen Analogsimulator
  - Anwendungen:
    - Bauelementemodellierung
    - Modellierung von analogen Komponenten (z.B. OPV)
- VerilogAMS
  - Erweiterung des Verilog Sprachumfangs zur Modellierung von analogen Signalen
  - Erlaubt Mixed-Signal Simulation unter Nutzung von digitalem und analogem Solver
  - Anwendungen:
    - Modellierung von Mixed-Signal Systemen
- System Verilog
  - Hardware-Beschreibungs- und Verifikationssprache (HDVL)
  - Enthält komplexere Datentypen (ähnlich C/C++), Möglichkeit der objektorientierten Programmierung sowie Konstrukte zur Verifikation
  - Anwendungen:
    - Modellierung und Verifikation komplexer Systeme
    - Testbenches und Verifikations IP

Verilog - Grundlagen

Design Unit Module

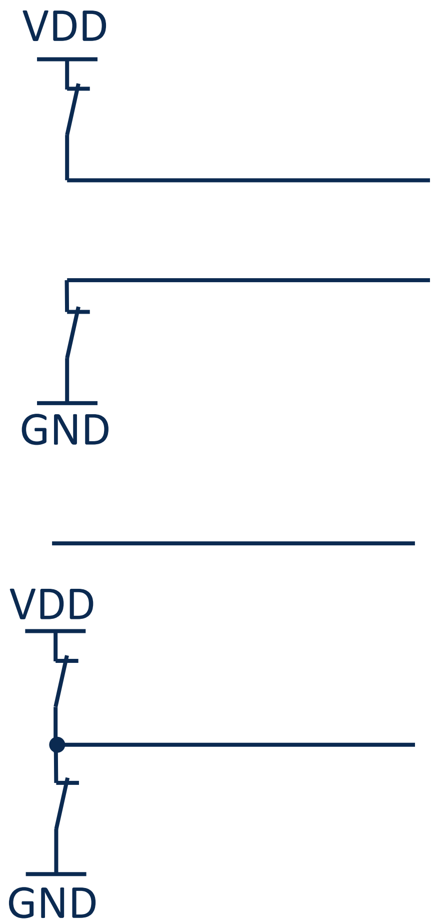



```
// Company   :    tud
// Author    :    hoeppner
// E-Mail    :    <email>
// Filename  :    my_module.v
// Project Name          :    p_cool
// Subproject Name       :    s_cool28soc
// Description           :    <short description>
// Create Date           :    Mon Jan 30 14:10:45 2012
// Last Change           :    $Date: 2012-10-24 12:07:59 +0200$
// by                    :    $Author: scholze $
//------------------------------------------------------------
module my_module (a_i, b_i, c_o);
 input a_i;
 input b_i;
 output c_o;
…


…
endmodule
```

- Coding Guideline:
  - für RTL 1 Module pro Verilog .v File
  - für Bibliotheken (z.B. Standardzellen) auch mehrere Module pro Datei

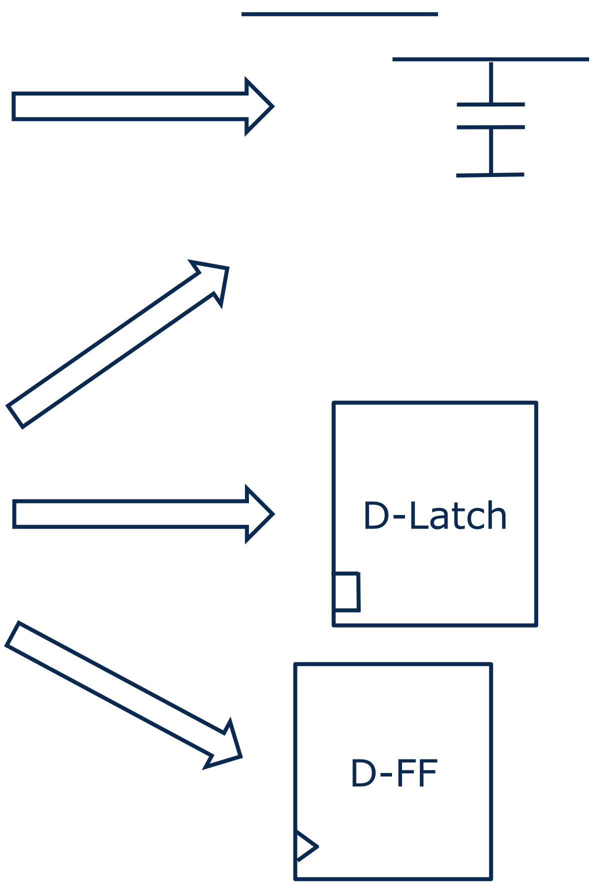
Logikpegel

- Verilog beschreibt 4 Logikpegel
  - Logische 1 : 1‘b1
  - Logische 0 : 1‘b0
  - hochohmig/tri-state : 1‘bz
  - unbekannt : 1‘bx



Datentypen

- Es gibt 2 verschiedene Datentypen zur Beschreibung elektrischer Signale:
- „Netz“ Datentyp
  - Speichert keinen Wert
  - muss kontinuierlich getrieben werden
  - Beispiel:
    - **wire** → elektrische Leitung
- „Register“ Datentyp
  - Abstraktion eines Speicherknotens
  - beschreibt „Register“ im Simulator
  - Speichert Wert zwischen einzelnen Zuweisungen
  - Beispiele:
    - **reg** → logisches Signal
    - integer → 32 Bit „Register“
    - real → reelle Zahl
    - time → Zeitpunkt
  - **ACHTUNG:** der Register Datentyp kann verwendet werden zur Beschreibung von
    - kombinatorischen Signalen
    - Speichern (Latch, FlipFlop, SRAM)

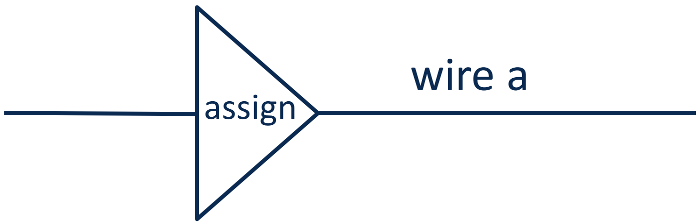


wire Datentyp

```
wire a,b,c,d;
assign a=b;
nor(b,c,d);
```

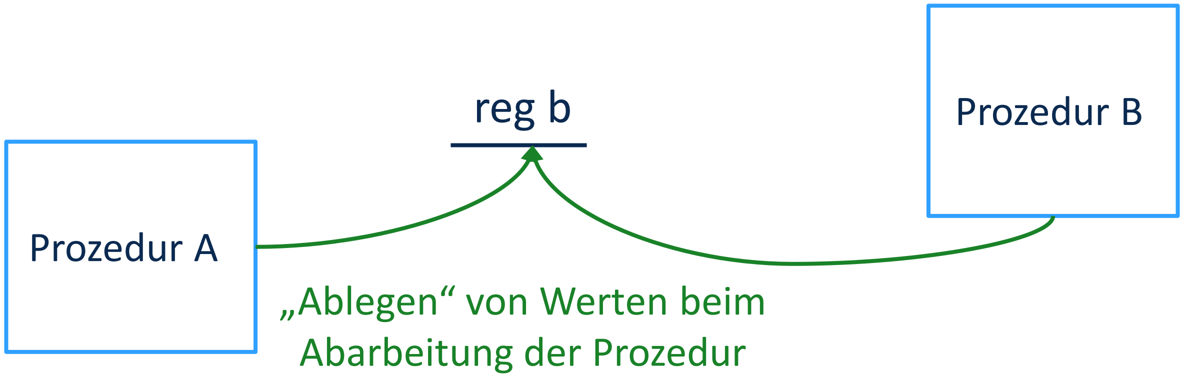
- Beschreibt elektrische Signale
- Kontinuierliche Wertzuweisung durch
  - **assign** statement
  - Gatter- oder Schalter
  - Modulinstanzen
- Verhalten bei Mehrfachzuweisungen:

wire	0	1	x	z
0				
1				
x				
z				



reg Datentyp

- Register Datentyp reg als Abstraktion eines **Speicherknotens**
- beschreibt „Register“ im Simulator
- Zuweisungen durch Prozeduren
- Speichert Wert zwischen einzelnen Zuweisungen



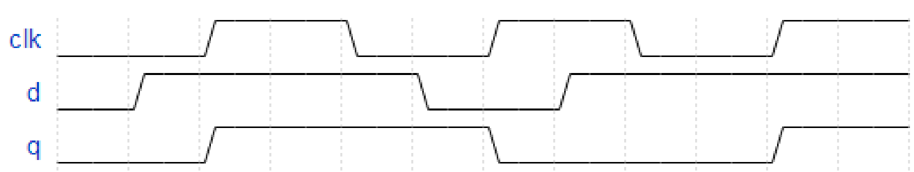
reg Datentyp Beispiele

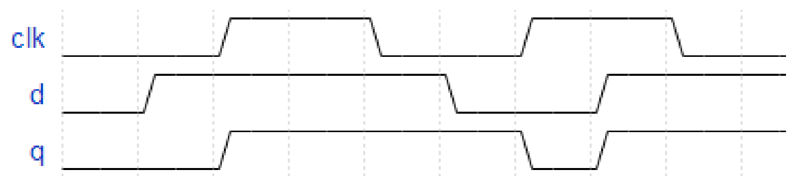
- Register Datentyp reg zur Beschreibung von:
  - kombinatorischen Signalen
  - Latches und RAM Zellen (zustandsgesteuert)
  - Flip-Flops (flankengesteuert)

```
//Kombinatorik
wire a;
reg b;
always @(a) begin
   b=a;
end
```

```
// D-FlipFlop
wire d,clk;
reg q;
always @(posedge clk) begin
   q<=d;
end
```

```
// D-Latch
wire d,clk;
reg q;
always @(clk or d) begin
   if (clk==1'b1) q<=d;
end
```





Der reg Datentyp beschreibt nicht zwingend ein physisches Register!

Zuweisung von konstanten Werten

- Möglichkeiten der übersichtlichen Zuweisung von Konstanten:

```
//Beispiele zur Zuweisung von Konstanten

//sized
reg [7:0] a =8‘d23;          //dezimaler Wert
reg [7:0] b =8‘b00010111;    //binärer Wert
reg [7:0] c =8‘h17;          //hexadezimaler Wert

//unsized
reg [7:0] d =23;             //führt zu 8‘b00010111
reg [7:0] e =‘h3;            //führt zu 8‘b00000011
reg [7:0] f =‘hf3;           //führt zu 8‘b11110011

// ‘_‘ zur besseren Lesbarkeit
reg [7:0] f =8‘b0001_0111; //binärer Wert
```

Parameter

- Die Nutzung von Parametern kann Source Code übersichtlicher gestalten sowie dessen **Wiederverwendbarkeit** erleichtern
- Parameter verhalten sich als **konstante Werte**
- Parameter lassen sich bei der Modul-Instantiierung überschreiben

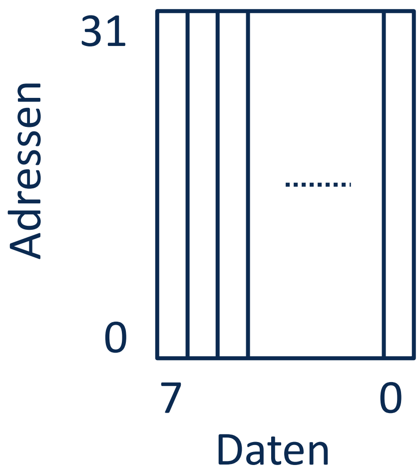
```
//Beispiel Modul mit Parametern
module incrementer (in_i,out_o);
   parameter C_DWIDTH=4;
   parameter C_STEP=2;

   input  [C_DWIDTH-1:0] in_i;
   output [C_DWIDTH-1:0] out_o;

   assign out_o=in_i+C_STEP;
endmodule
```

- Signale und Modul Pins können zu Bussen zusammengefasst werden
- Beispiele:

```
//8-Bit Register ohne Reset
module reg_8 (clk, d_i, q_o)
   input clk;
   input [7:0] d_i;
   output [7:0] q_o;
   reg [7:0] q;
   always @(posedge clk) begin
      q<=d_i;
   end
   assign q_o=q;
endmodule
```

```
//Signalzuweisungen mit Bussen
wire a;
wire [7:0] b,c,d,e; //8-Bit Bus
reg  [3:0] r;    //4-Bit Register

assign a=b[7];          //1-Bit Zuweisung
assign c=8'b0011_11xz; //8-Bit Konstante

//Zusammengesetztes Signal (Concatenation)
assign d={b[6:4],1'b0,c[3:0]};

//Zuweisung von 8'b00000000
assign e={8{1'b0}};

always @(b) begin
   r[3:0]=b[3:0];    //Teil-Wort Zuweisung
end
```

- Speicherblöcke lassen sich als Array definieren




```
wire we;
wire [4:0] addr;
wire [7:0] data_write;
wire [7:0] data_read;

// memory array
reg [7:0]   memory_32x8 [0:31];

//write logic
always @(data_write or we or addr) begin
  if (we==1'b1) begin
      memory_32x8[addr]=data_write;
  end

//continuous read
assign data_read=memory_32x8[addr];
```

Treiberstärken

- Signaltreiber können verschiedene Treiberstärken besitzen

	Name	Level	Abbk.	
1	supply1	7	Su1	
	strong1	6	St1	← Default
	pull1	5	Pu1	
	large1	4	La1	
	weak1	3	We1	
	medium1	2	Me1	
	small1	1	Sm1	
	highz1	0	HiZ1	
0	highz0	0	HiZ0	
	small0	1	Sm0	
	medium0	2	Me0	
	weak0	3	We0	
	large0	4	La0	
	pull0	5	Pu0	
	strong0	6	St0	← Default
	supply0	7	Su0	

Zuweisung von Treiberstärken

- Kontinuierliche Zuweisungen

```
//Beispiele für Treiberstärken für kontinuierliche Zuweisungen
assign (strong1, pull0)  a = b; //Unsymmetrischer Treiber
assign (highz1, strong0) c = d; //Open-Drain Treiber
```

- Gatter Schaltungen

```
//Beispiele für Treiberstärken von Gattern
and (strong1, pull0) (out,in1,in2);
nor (strong1, highz0)(out,in1,in2);
```

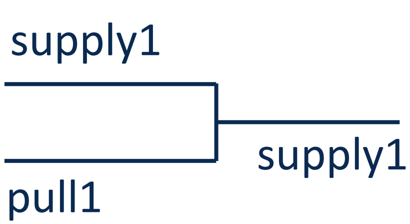
Kombination verschiedener Treiberstärken

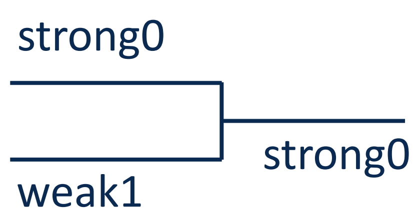
- Ein Netz kann durch mehrere Zuweisungen getrieben werden

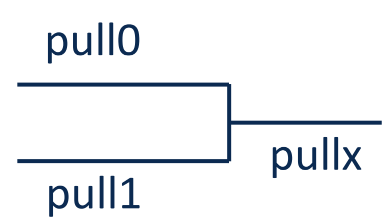
```
//Beispiel Treiberstärken
wire a, b, c;
assign (strong1, pull0) a = b;
assign (pull1, strong0) a = c;
```

- Es können Konflikte beim Zusammenschalten aufgelöst werden
- Wichtigste Fälle:
  - unterschiedliche Treiberstärken; unterschiedliche Logikpegel
    - → Logischer Pegel des stärkeren Signal
    - → Treiberstärke des stärkeren Signals
  - gleiche Treiberstärken; unterschiedliche Logikpegel
    - → Logischer Pegel ergibt sich zu 1‘bx bei gleicher Treiberstärke

- Beispiele:







Logische Operatoren in Verilog

- Operatoren verknüpfen und/oder modifizieren Signale

<table>
<tr><td>{}</td><td>concatenation</td></tr>
<tr><td>+ - * /</td><td>arithmetic</td></tr>
<tr><td>%</td><td>modulus</td></tr>
<tr><td>> >= < <=</td><td>relational</td></tr>
<tr><td>!</td><td>logical negation</td></tr>
<tr><td>&&</td><td>logical and</td></tr>
<tr><td>||</td><td>logical or</td></tr>
<tr><td>==</td><td>logical equality</td></tr>
<tr><td>!=</td><td>logical inequality</td></tr>
</table>

<table>
<tr><td>~</td><td>bit-wise negation</td></tr>
<tr><td>&</td><td>bit-wise and</td></tr>
<tr><td>|</td><td>bit-wise or</td></tr>
<tr><td>^</td><td>bit-wise xor</td></tr>
<tr><td>^~ or ~^</td><td>bit-wise equivalence</td></tr>
<tr><td>&&</td><td>logical and</td></tr>
<tr><td>&</td><td>reduction and</td></tr>
<tr><td>|</td><td>reduction or</td></tr>
<tr><td>^</td><td>reduction xor</td></tr>
<tr><td>^~ or ~^</td><td>reduction xnor</td></tr>
<tr><td><<</td><td>left shift</td></tr>
<tr><td>>></td><td>right shift</td></tr>
<tr><td>?:</td><td>conditional</td></tr>
</table>

Priorität von Operatoren

! ~	höchste Priorität
* / %	
+ -	
<< >>	
< <= > >=	
== != === !===	
&	
^ ^~	
|	
&&	
||	
?:	niedrigste Priorität

```
//Beispiel Priorität von Operatoren
wire a,b,c,d;
assign a= d==c&b ? d|b&c : c;
```

d	c	b	a
0	0	0	
0	0	1	
0	1	0	
0	1	1	
1	0	0	
1	0	1	
1	1	0	
1	1	1	

- Beispiele zu arithmetischen Operatoren

```
wire [4:0] a,b,c,d,e,f;

//Arithmetic Operators
assign a= 27+2‘b01; // Result 28
assign b= 27+5;     // Result 0
assign c= 27-2‘b01; // Result 26
assign d= 3*2;      // Result 6
assign e= 5/2;      // Result 2
assign f= 10%3;     // Result 1
```

Logische Operatoren und Vergleichsoperatoren

- Beispiele zu logischen Operatoren und Vergleichsoperatoren

```
wire a,b,c,d,e,f,g,h,i,j;

//Relational and Logic operators
assign a=(2‘b01==2‘b10); //Result 1‘b0
assign b=(2‘b01!=2‘b10); //Result 1‘b1
assign c=(1‘bx===1‘bx);  //Result 1‘b1
assign d=(1‘b1&&(2‘b10>=2‘b01)); //Result 1‘b1
```

Gleichheits Operatoren	
`a === b`	a gleich b, einschließlich x und z, **nicht synthesefähig!**
`a !== b`	a ungleich b, einschließlich x und z **nicht synthesefähig!**
`a == b`	a gleich b, Ergebnis kann x sein
`a != b`	a ungleich b, Ergebnis kann x sein

Bit-wise und Reduction Operatoren

- Beispiele zu Bit-wise und Reduction Operatoren

```
wire [3:0] a,b,c,d;

//Bitwise operators
assign a=4'b0010&4'b1110;  //Result 4'b0010
assign b=4'b0010|4'b1110;  //Result 4'b1110
assign c=4'b0010^4'b1110;  //Result 4'b1100
assign d=~4'b0010;         //Result 4'b1101
```

```
wire a,b,c,d;

//Reduction operators
assign a=&4'b0010;  //Result 1'b0
assign b=|4'b0010;  //Result 1'b1
assign c=^4'b0010;  //Result 1'b1
assign d=^~4'b0010; //Result 1'b0
```

~	
0	1
1	0
x	x

&	0	1	x
0	0	0	0
1	0	1	x
x	0	x	x

|	0	1	x
0	0	1	x
1	1	1	1
x	x	1	x

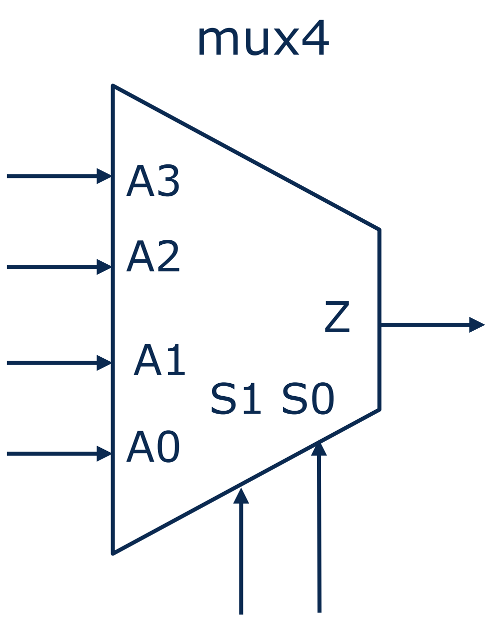
^	0	1	x
0	0	1	x
1	1	0	x
x	x	x	x

^~	0	1	x
0	1	0	x
1	0	1	x
x	x	x	x

- Beispiele zu Shift Operatoren

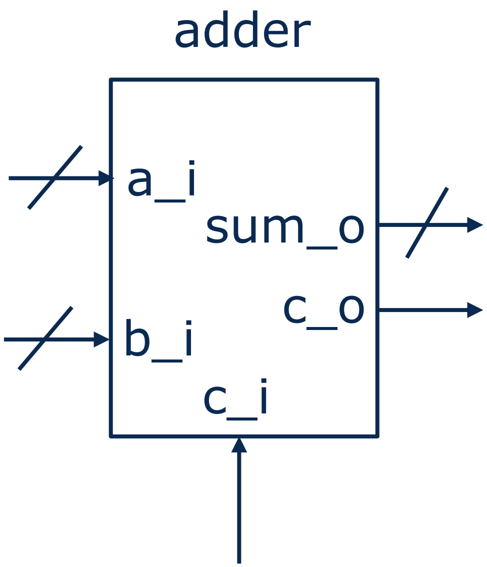
```
wire [3:0] a,b,c,d;

//Shift operators
assign a=4'b0010 << 1; //Result 4'b0100
assign b=4'b0010 << 2; //Result 4'b1000
assign c=4'b0010 >> 1; //Result 4'b0001
assign d=4'b0010 >> 2; //Result 4'b0000
```

Beispiel: Multiplexer




```
//4-zu-1 Multiplexer
module mux4 (A3,A2,A1,A0,Z,S1,S0);
   input A3,A2,A1,A0;
   input S1,S0;
   output Z;

   wire s32,s10,s3210;

   //MUX logic
   assign s10   =(S0) ? A1  : A0 ;
   assign s32   =(S0) ? A3  : A2 ;
   assign s3210 =(S1) ? s32 : s10 ;

   //output assignment
   assign Z=s3210;

endmodule
```

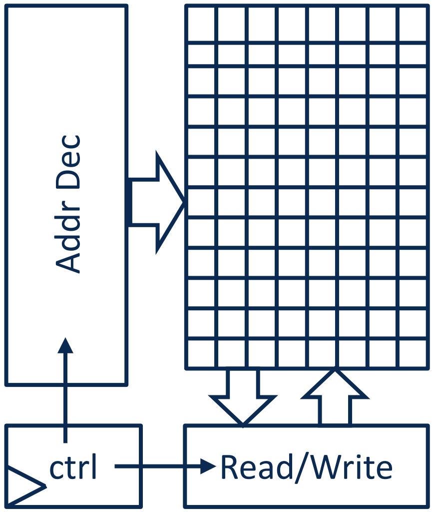
Beispiel: Addierer




```
//Addierer
module adder (sum_o, c_o, c_i, a_i, b_i) ;
   parameter C_DWIDTH=4;

   input [C_DWIDTH-1:0]  a_i, b_i;
   input c_i;
   output [C_DWIDTH-1:0] sum_o;
   output c_o;

   assign {c_o, sum_o} = a_i + b_i + c_i;
endmodule
```

Beispiel: Single-Port SRAM Makro Block

```
module HM_1P_core_1cr (CLK_I,ADDR_I,DW_I,WE_I,RE_I,
CS_I,DR_O);

   parameter P_ADDR_WIDTH=8;
   parameter P_DATA_WIDTH=128;

   input CLK_I, WE_I, RE_I, CS_I;
   input [P_ADDR_WIDTH-1:0] ADDR_I;
   input [P_DATA_WIDTH-1:0] DW_I;
   output [P_DATA_WIDTH-1:0] DR_O;

   reg [P_DATA_WIDTH-1:0]    mem [0:2**P_ADDR_WIDTH-1];
   reg [P_DATA_WIDTH-1:0]    dr_r;

   always @(posedge CLK_I) begin
      if(CS_I==1'b1 && WE_I==1'b1) begin
         mem[ADDR_I] <= DW_I;   //write
      end
      if(CS_I==1'b1 && RE_I==1'b1) begin
        if (WE_I==1‘b1) dr_r<=DW_I;      //write through
        else            dr_r<=mem[ADDR_I];      //read
        end
   end
   assign DR_O=dr_r;
endmodule
```



Beispiel: Single-Port SRAM Makro Block – Timing Diagramm