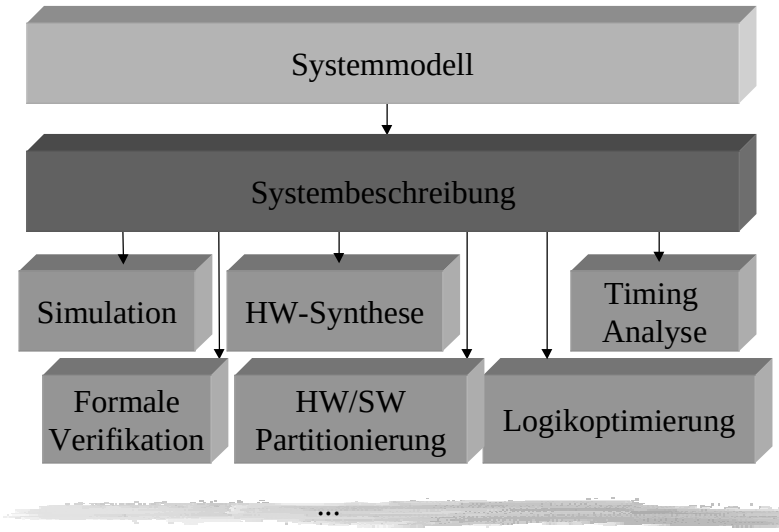
1 Hardwareentwurf

- **1.1** Überblick, Hardwareentwurfsschritte
- **1.2** Hardwarebeschreibungssprachen
- **1.3** Hardwaresimulation-/Verifikation
- **1.4** Hardwaresynthese
- **1.5** Plazierung und Verdrahtung

Literatur

- James M. Lee, Verilog Quickstart! Kluwer Academic Publishers, ISBN 0-7923-8515-2

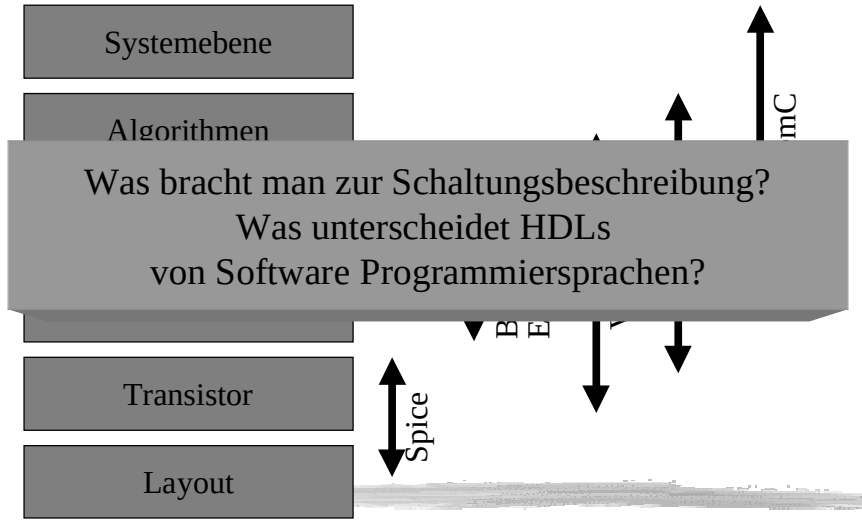
Zusammenhang



Hardware-Beschreibungssprachen

- präzise Spezifikation
- Möglichkeit der Simulation
- Automatisierung
- Dokumentation
- Beschleunigung
- ...

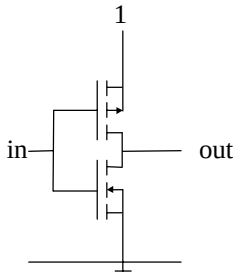
Einordnung verschiedener HDLs



Schaltungsbeschreibungen

Schaltungen bestehen aus

- Komponenten
- und Verbindungen dieser Komponenten

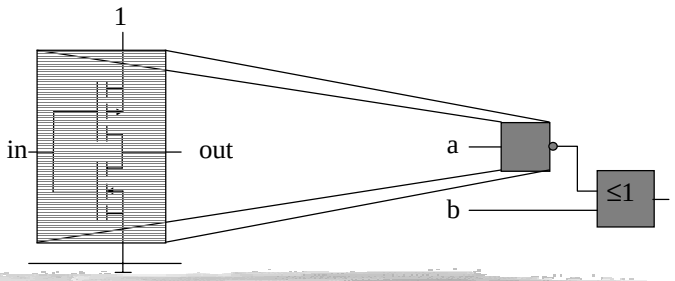


Zu komplex für große Schaltungen

Hierarchiebildung

Zusammenfassung von Teilen zu neuen Modulen

- Gatter, FlipFlops
- ALU, Register, Speicher,
- Prozessor



Algorithmische Beschreibung

Strukturelle Beschreibung ist oft zu komplex für große Entwürfe (mit 20 Millionen Gattern)

⇒ algorithmische Beschreibungen notwendig

Das Verhalten der Module wird durch eine (imperative) Programmiersprache definiert. Diese ist Teil der Hardwarebeschreibungs-sprache

Algorithmische Beschreibung II

Besonderheiten von Hardware:

- Funktionen verbrauchen Zeit
  - ➜ Zeitbegriff
- Funktionen können parallel arbeiten
  - ➜ parallele Tasks
- Kommunikation zwischen Modulen
  - ➜ Signale und Ereignisse (events)
- zweiwertige Logik nicht ausreichend
  - ➜ mehrwertige Logik (0,1,x,z,...)

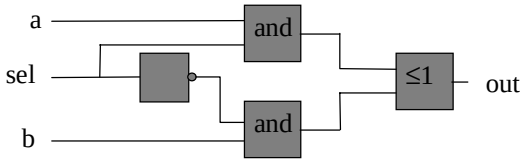
Verilog

- entwickelt von Philip Moorby 1983/1984 bei Gateway Design Automation
- wurde anfangs gemeinsam mit dem Simulator entwickelt
- 1987 Verilog-basiertes Synthesewerkzeug von Synopsys
- 1989 Gateway wurde von Cadence aufgekauft
- Verilog wird public domain um mit VHDL zu konkurrieren

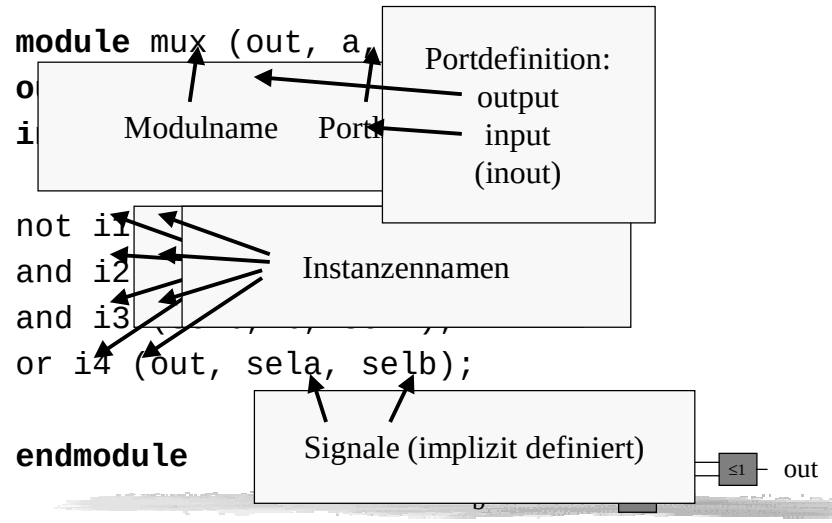
Verilog

- Standard
  - einheitliche Schnittstelle zwischen Werkzeugen und Firmen
- Portabilität
  - Entwürfe können durch verschiedene Synthesewerkzeuge optimiert und durch verschiedene Analysewerkzeuge simuliert werden
  - Technologie- und Firmenunabhängigkeit
    - Wechsel des Technologiepartners möglich

Strukturelle Beschreibung: Multiplexer



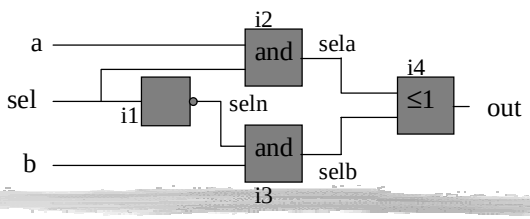
Strukturelle Beschreibung: Multiplexer




```
module mux (out, a,
o
i
not i1
and i2
and i3
or i4 (out, sela, selb);

endmodule
```

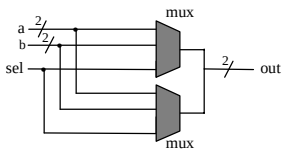

Strukturelle Beschreibung: Multiplexer

```
module mux (out, a, b, sel);
output out;
input a, b, sel;

not i1 (seln, sel);
and i2 (sela, a, sel);
and i3 (selb, b, seln);
or i4 (out, sela, selb);

endmodule
```



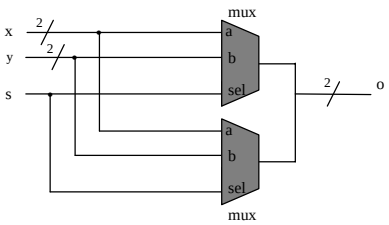
Hierarchie: Multiplexer 2

```
module mux2 (out, a, b, sel);
output [1:0] out;
input [1:0] a, b;
input sel;

mux hi (out[1], a[1], b[1], sel);
mux lo (out[0], a[0], b[0], sel);

endmodule
```



Modulverbindung durch Portnamen

```
module mux2 (o, x, y, s);
...
mux hi (  .out(o[1]),
          .a(x[1]),
          .b(y[1]),
          .sel(s));
mux lo (  .sel(s),
          .b(y[0]),
          .a(x[0]),
          .out(o[0]));
```



Grundlegende Sprachelemente I

Kommentare

// Zeilenkommentar bis zum Zeilenende

/* Bereichskommentar kann über
mehrere Zeilen bis zum schließenden
Kommentarzeichen gehen */

Grundlegende Sprachelemente II

Bezeichner

- Buchstaben (a-z,A-Z), Zahlen (0-9) oder _ $
- beginnt mit Buchstabe oder _
- case sensitive
- maximal 1024 Zeichen lang
- Escaped identifier
  `\hier/kann?jedes:Zeichen.kommen`

Grundlegende Sprachelemente IV

Macros

- Definition
  `define opcode_add 33
- Anwendung
  b = `opcode_add

Logikwerte

- 0 : logisch falsch, niedriger Signalpegel
- 1 : logisch wahr, hoher Signalpegel
- x : unbekannt (don't care)
- z : hochohmig (keine Verbindung)

Zahlen

Bits, Bitvektoren

Bitbreite	‘	Basis	Werte

```
8’b11001001
8’hff
16’d12
12’o777
```

Integers, Reals

- Bitbreite ist maschinenabhängig (z.B. 32 Bit)
- vorzeichenbehaftete Arithmetik

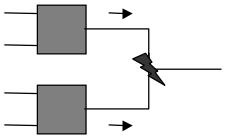
Netze (von Bitvektoren)

- Verbinden Module
- es gibt mehrere Netztypen:
  - wire (tri)
  - wand (triand)
  - wor (trior)
  - tri1, tri0
  - supply1, supply0
- Es können Bitvektoren gebildet werden, z.B.:

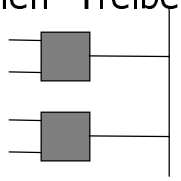
```
wire [63:32] high;
```

Resolution

In HW haben Logiksignale genau einen "Treiber"



Ausnahme:
Busse



Verhalten hängt vom Netztyp ab:

- wire: nur ein Treiber erlaubt
- wand: Konjunktion der Treibersignale
- wor: Disjunktion der Treibersignale
- tri0: wie wand mit pulldown (kein Treiber ⇒ Leitung=0)
- tri1:wie wand mit pullup (kein Treiber ⇒ Leitung=1)
- supply0, supply1: kein Treiber erlaubt

Grundlegende Sprachelemente VI

Operatoren:

- arithmetische: +, -, *, /, %
- logische: &, &&, |, ||, ^, ~, !, <<, >>, <<<, >>>
- Reduktion: &, |, ^, ~&, ~|, ~^
- relationale: < , <= , > , >= , == , === , != , !==
- bedingter Operator:
  `cond ? true_exp : false_exp`
- concatenation: { … }

```
x = { a, b, c };
{x,y} = 8'b10011101;
```

Beschreibung von Schaltnetzen

- Mit "built-in primitives" (siehe MUX)
- Mit "continous assignment"

```
module sn(out, in1, in2);
output out;
input in1, in2;

assign out = in1 & in2;

endmodule
```


Beschreibung von Schaltnetzen

- Mit "built-in primitives" (siehe MUX)
- Mit "continous assignment"

```
module sn(out, in1, in2);
output [32:0] out;
input [31:0] in1, in2;

assign out = in1 + in2;
// hunderte von Gattern
endmodule
```

Beschreibung von Schaltnetzen

- Mit "built-in primitives" (siehe MUX)
- Mit "continous assignment"

```
module sn(out, in1, in2);
output [63:0] out;
input [31:0] in1, in2;

assign out = in1 * in2;
// tausende von Gattern
endmodule
```


Continous Assignment Beispiel

```
module mux (out, in1, in2, sel);
output out;
input in1, in2, sel;

assign out = sel ? in2 : in1;

endmodule;
```

Verhaltensbeschreibung

initial

jeder initial-Block jedes Moduls wird zu Beginn der Simulation genau einmal bis zum Ende ausgeführt.

always

jeder always-Block jedes Moduls wird zu Beginn der Simulation ausgeführt und wird dann zyklisch immer wiederholt.

Alle Blöcke arbeiten parallel

Beispiele

```
module test;

initial $display("init block 1");
initial $display("init block 2");

endmodule
```

Systemtask für Bildschirmausgabe

Was erscheint auf dem Bildschirm?

Beispiele

```
module test;

initial $display("init block 1");
initial $display("init block 2");
always $display("always block");

endmodule
```

Was erscheint auf dem Bildschirm?

Der Zeitoperator

Alle bisher gezeigten Operationen sind (Simulations-) zeitfrei

Mit dem Operator # kann Zeit verbraucht werden

```
module test;
initial #2 $display("init block 1");
initial #5 $display("init block 2");
endmodule
```

Sequentielle Schachtelung

Mehrere Anweisungen können mit begin-end zusammengefaßt werden. Alle eingeschlossenen Anweisungen werden sequentiell abgearbeitet.

```
module test;
initial
  begin
     $display("init block 1");
     $display("init block 2");
  end
endmodule
```


Sequentielle Schachtelung II

Zeit vergeht "sequentiell", d.h. relativ zum letzten Zeitpunkt

```
module test;
initial
  begin
     #2 $display("init block 1");
     #2 $display("init block 2");
  end
endmodule
```

Parallele Schachtelung

Mehrere Anweisungen können mit fork-join zusammengefaßt werden. Alle einge-schlossenen Anweisungen werden parallel abgearbeitet.

```
module test;
initial
  fork
     $display("init block 1");
     $display("init block 2");
  join
endmodule
```


Parallele Schachtelung II

Da alle Anweisungen parallel abgearbeitet werden, wird Zeit immer absolut gemessen

```
module test;
initial
  fork
     #2 $display("i
     #4 $display("i
  join
endmodule
```

Begin-end Blöcke und fork-join Blöcke können beliebig geschachtelt werden

Register

```
reg val [7:0];
```

- Wird in algorithmischen Beschreibungen verwendet
- nur interne Signale und outputs können Register sein
- können auch zur Schaltnetzmodellierung verwendet werden
- Memories:

```
reg [7:0] mem [0:1023];
```


Ereignisse

- sind Signalwechsel

- warten auf Ereignisse: `@c`
- abfangen mehrere Ereignisse: `@(e1 or e2)`

Ereignisse Beispiel: DFlipFlop

```
module dff (out, clock, in);
output out;
input clock, in;
reg out;

always @(posedge clock)
  out = in;

endmodule
```


Ereignisse Beispiel: Multiplexer

```
module mux (out, in1, in2, sel);
output out;
input in1, in2, sel;
reg out;

always @(in1 or in2 or sel)
  if (sel) out = in2
  else     out = in1

endmodule
```

Ereignisse Beispiel: FehlerMux

```
module mux (out, in1, in2, sel);
output out;
input in1, in2, sel;
reg out;

always @sel
  if (sel) out = in2
  else     out = in1

endmodule
```


Levelabhängiges Warten

`wait (boolean-expression);`

Falls *boolean-expression* wahr ist, wird direkt mit der nachfolgenden Anweisung im Programmfluß fortgefahren

Falls *boolean-expression* falsch ist, dann wird der Programmfluß solange unterbrochen bis der Ausdruck wahr wird

Kontrollfluß

Bedingung

- **if** (cond) statement
- **if** (cond) statement1
  else statement2
- **case** (sel)
  0 : y = a
  1 : y = b
  default : y = 2'bxx
  endcase
- **casez, casex**

Kontrollfluß II

Schleifen

- **forever** statement

```
module ClockGen (clk);
output clk;
reg clk;

initial begin
          clk = 0;
          forever #50 clk = ~clk;
        end
endmodule
```

Kontrollfluß II

Schleifen

- **forever** statement
- **repeat** (num) statement

```
module ClockGen (clk);
initial repeat (5) $display("hallo");
endmodule
```


Kontrollfluß II

Schleifen

- **forever** statement
- **repeat** (num) statement
- **while** (cond) statement
- **for** (init; cond; incr) statement

Zuweisungen in algorithmischen Blöcken

var = expression;

Beispiel

```
initial begin
    x = 3;
    y = 4;
    fork
         x = y;
         y = x;
    join
end
```


Zuweisungen mit intra-assign delay

var = #num expression;

Beispiel

```
initial begin
    x = 3;
    y = 4;
    fork
         x = #1 y;
         y = #1 x;
    join
end
```

Nichtblockierende Zuweisungen

`var <= #num expression;`

Beispiel

```
initial begin
    x = 3;
    y = 4;
    begin
         x <= #1 y;
         y <= #1 x;
    end
end
```


Zuweisungen Zusammenfassung

Zuweisungstyp	LHS	Wann ausgeführt	Wo im Modul
Procedural assignment	reg	Bei Aufruf	In alg. Blöcken
Continous assignment	net	Bei RHS-Änderung	Im umgebenden Modul

Tasks

- dienen zum „verkapseln" von Verhalten
- werden innerhalb von Modulen definiert
- können auf umgebende Daten zugreifen
- können inputs und outputs haben
- können Verzögerungszeiten beinhalten
- Daten innerhalb eines Tasks gibt es nur einmal auch wenn mehrere identische Tasks laufen

Beispieltask

```
module counter(out,
  clk, reset);
output [7:0] out;
reg [7:0] out;
input clk, reset;

always @clk
  if (reset) out = 0;
  else incr(out);
```

```
task incr;
inout [7:0] x;
  x = x + 1;
endtask
endmodule
```

Funktionen

- dienen zum „verkapseln" von Verhalten
- werden innerhalb von Modulen definiert
- kann auf umgebende Daten zugreifen
- können inputs haben
- liefern immer ein Ergebnis zurück
- dürfen keine Verzögerungszeiten beinhalten
- Daten innerhalb einer Funktion gibt es nur einmal, d.h. es gibt keinen Laufzeitstack

Beispielfunktion

```
module mux (out, a, b,
  c, d, sel);
output out;
input [1:0] sel;
input [7:0] a, b, c,
  d;

assign out =
  muxfunct(sel,a,b,c,d
  );
```

```
function [7:0] muxfunct;
input [1:0] sel;
input [7:0] a,b,c,d;
  case (sel)
   2'b00 : muxfunct = a;
   2'b01 : muxfunct = b;
   2'b10 : muxfunct = c;
   2'b11 : muxfunct = d;
  endcase
endfunction
endmodule
```

Parametrisierte Module

- Dienen zur Beschreibung generischer Module
  - variable Bitbreite
  - variable Verzögerungszeiten
- Parameter müssen vor der Simulationszeit festgelegt werden
- Parameter werden mit dem Schlüsselwort `parameter` definiert
- Parameter sind defaultmäßig vom Typ integer

Beispiel: N-Bit Addierer

```
module nadder(cout, sum, a, b, cin);
parameter size = 32;
parameter delay = 1;
output [size-1:0] sum;
output cout;
input [size-1:0] a, b;
input cin;

assign #delay {cout,sum} = a + b + cin;

endmodule
```

Instanziierung

- Durch den Parameternamen

```
nadder a1 (z,a5,b5,c5,x);
defparam a1.size = 16;
defparam a1.delay = 4;
```

- Durch die Parameterreihenfolge

```
nadder #(16,4) a1 (z,a5,b5,c5,x);
```

- Durch Parameternamensliste

```
nadder #(.size(16), .delay(4))
   a1 (z,a5,b5,c5,x);
```


Systemtasks

- Simulationskontrolle
  - $finish, $stop
- Bildschirmausgabe
  - $display, $write, $strobe, $monitor
- Filezugriff
  - Filehandels sind integer (maximal 32)
  - $fopen("file1"), $fclose(i)
  - $fdisplay, $fwrite, $fstrobe, $fmonitor