TU Darmstadt
Fachbereich Informatik
Mathias Halbach

Einführung in Verilog

Technische Grundlagen der Informatik II

(Rechnertechnologie II)

Beschreibung von Hardware

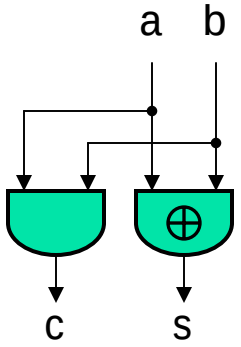
- umgangssprachlich
- Logiktabelle
- als Schaltplan
- Beschreibungssprache Verilog
  - simulierbar
  - auf FPGA (Hardware) nach Synthese übertragbar

entnommen aus Werbebroschüren für Xilinx Spartan™-3E FPGAs, 2005 bzw. Altera Cyclone Series, August 2004

Halbaddierer

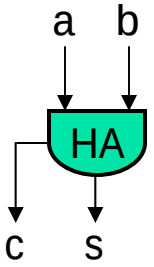
- Zwei Bit Eingang (a, b)
- Ausgang
  - 1 Bit Summe s (niederwertiges Bit)
  - 1 Bit Übertrag c (carry, höherwertiges Bit)

a	b	c	s
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0



Zusammenfassung als „Black Box“

- module Name(Port1, Port2, ...);
  - Ein-/Ausgänge (Ports)
  - Deklarationen
  - Beschreibungen
- endmodule

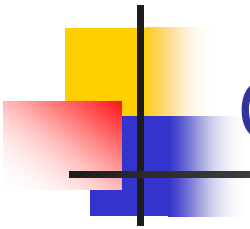


Hochsprachliche Operatoren

```
module     hadd1(a, b, c, s);
   input     a, b;
   output    c, s;
   assign {c, s} = a + b;
        /* { , }   Konkatenation */
endmodule
```

Logische Operatoren

```
module    hadd2(a, b, c, s);
  input   a, b;
  output  c, s;
  assign s = a ^ b;    // exklusiv-oder
  assign c = a & b;    // und
endmodule
```



Gatter-Bausteine (Primitive)

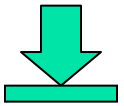
```
module     hadd3(a, b, c, s);
   input     a, b;
   output    c, s;
   xor xor1(s, a, b);
   and and1(c, a, b);
   // Baustein  aktueller Name  Verbindung
endmodule
```

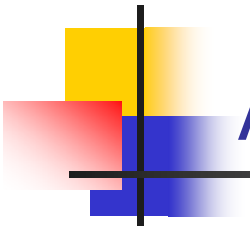
Von Logik-Schaltung zu Schaltnetz – Beschreibungen innerhalb von module

- assign a = b;
  - immerwährende Zuweisung

- always @(c) a <= b;
  - Ausführung, wenn sich c ändert
  - Achtung! Ohne Bedingung: in Simulation Endlosschleife!

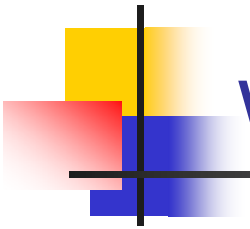
- initial a = b;
  - einmalige Ausführung
  - nur für Simulation (nicht Synthese)





Alternative Bedingungen nach always

- @(posedge clock), @(negedge clock)
  - positive/negative Taktflanke von clock
- @(a or b or posedge c)
  - Änderung von a oder b (jeweils positive oder auch negative Taktflanke)
    oder positive Taktflanke c

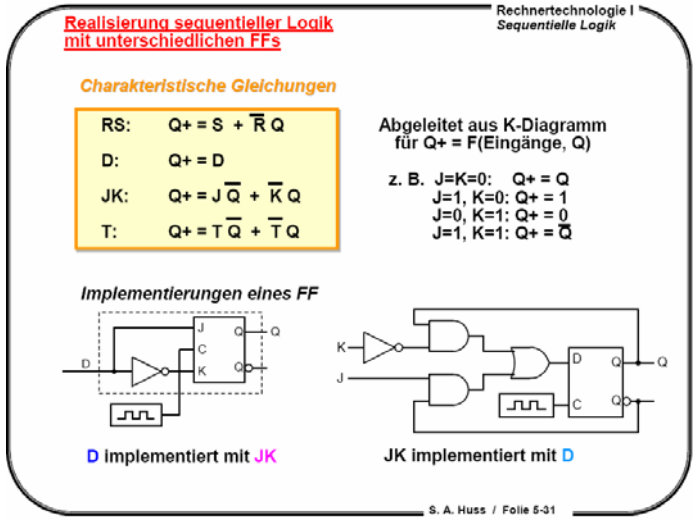


Wohin Daten puffern?

- reg h;
  - Speicher für ein Bit (Flip-Flop, Daten-Register)
- reg [5:3] g;
  - Ein 3-Bit-Register, Zugriff z. B. mit g[4]
- reg [7:0] a [4:0];
  - Speicher mit 5 mal 8 Bit
- Wertzuweisung mit
  - <= wenn synchron, parallel (erst alle Anweisungen berechnen, dann zuweisen)
  - = wenn asynchron, sequentiell (blockierend, d. h. nächste Anweisung wird verzögert!)

Beispiel JK-Flipflop

```
module jkff(clk, j, k, q, _q);
  input clk, j, k;
  output q, _q;
  reg q;
  initial q = 1'b0;
  always @(posedge clk)
    q <= (j & !q) | (!k & q);
  assign _q = !q;
endmodule
```

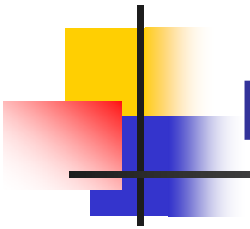


entnommen aus Vorlesungsfolien TGdI1 von Prof. Huss (TU Darmstadt), Wintersemester 2005/2006, Kapitel 5

Mehrere Befehle

- nach always und initial nur eine Anweisung (sequential statement)!
- Lösung: begin/end
- always @(c) begin: Schleifenname

```
    integer i;
    for(i = 0; i < 5; i = i + 1) begin
        a[i] <= b – i;
        d[i] <= e * i;
    end
end
```



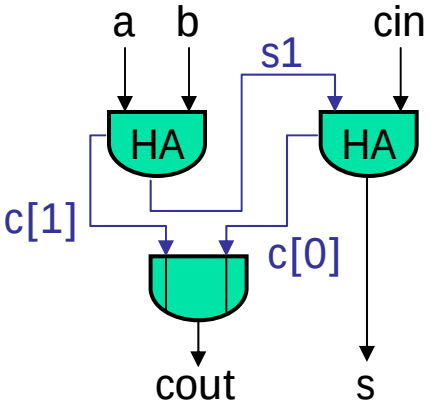
Datenweiterleitung

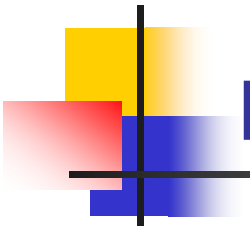
- Für eine Verbindung: wire w;
- Für Leitungsbündel: wire [3:0] b;

- Zur Verbindung von
  - Gattern
  - Instanzen von Modulen

Wiederverwendung von Modulen

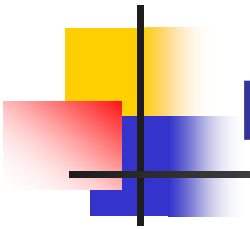
```
module volladdierer(a, b, cin, cout, s);
   input a, b, cin;
   output cout, s;
   wire s1; wire [1:0] c;
   hadd2 a1(a, b, c[1], s1),
         a2(s1, cin, c[0], s);
   assign cout = c[0] | c[1];
endmodule
```





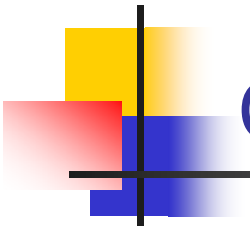
Parametrisierung

- module addierer(a, b, s);
  parameter n = 8;
  input [n – 1 : 0] a, b;
  output [n – 1 : 0] s;
  assign s = a + b;
  endmodule
- module multiadd(a, b, c, s);
  input [3 : 0] a, b, c;
  output [3 : 0] s;
  wire [3 : 0] s1;
  addierer #(4) x(a, b, s1), y(c, s1, s);
  endmodule



Konstante Zahlenwerte

- Angabe der Bitbreite (optional)
- Angabe der Basis (hex., oktal, dez., binär)
- eigentlicher Wert, bestehend aus Ziffern
  - 0 bis 9 und A bis F (sofern gültig)
  - x für unbekannt bzw. undefiniert
  - z für hochohmig (bei Tristate-Gattern)
  - Groß-/Kleinschreibung irrelevant
- z. B. 4'he, 'o347, 5'b01xz1, 8, 9'd2, -2'B1



Operatoren

- Reduktion:
  statt c[0] | c[1] einfacher |c[1:0] oder |c
- Replikation: statt {b, b, b} einfacher {3{b}}
- Falls irgendein Bit eines Operanden den Wert unbekannt („x“) hat, ist der Wert des Gesamtergebnisses ebenfalls unbekannt!
  - Ausnahme: Bit-Vergleichsoperatoren
    === und !==
- weitere auf dem Übersichtsblatt

Vergleichsoperatoren
Ergebnisse von "a == b" und "a === b"

a	b	==	===
0	0	1	1
0	1	0	0
0	'bx	'bx	0
0	'bz	'bx	0
1	0	0	0
1	1	1	1
1	'bx	'bx	0
1	'bz	'bx	0

a	b	==	===
'bx	0	'bx	0
'bx	1	'bx	0
'bx	'bx	'bx	1
'bx	'bz	'bx	0
'bz	0	'bx	0
'bz	1	'bx	0
'bz	'bx	'bx	0
'bz	'bz	'bx	1

Bedingte Ausführung

- if (formel) statement
- if (formel) statement else statement
- case (ausdruck)
  wert1, wert2: statement
  wert3: statement
  default: statement //Doppelpunkt optional
  endcase
- Es wird kein "break" benötigt, da nur ein statement oder begin/end!

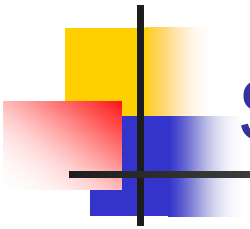
Beispiel case, 4-Kanal Multiplexer

```
module mux4_case(a, b, c, d, select, ena, y);
   input a, b, c, d, ena; input [1:0] select;
   output y; reg y;
   // y ist Hilfsvariable, kein Hardware-Register
   always @(a or b or c or d or select or ena)
      if (ena)
         case (select)
           0: y = a;  1: y = b;  2: y = c;  3: y = d;
           default: y = 1'bx;
         endcase
      else y = 1'bz;
endmodule
```

Statt case gibt es auch casex und casez

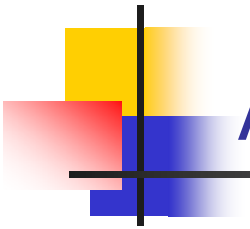
- die aufgeführten Werte werden als Übereinstimmung gewertet
- angegeben ist nur der niedrigste Standard,
  z. B. bei Icarus ist bei case auch x und z möglich
- wenn keine Bedingung zutrifft, wird der default-Teil verwendet

bei expr	case	casez	casex
0	0	0, z	0, x, z
1	1	1, z	1, x, z
x			0, 1, x, z
z		0, 1, x, z	0, 1, x, z
?			0, 1, x, z



Simulation

- Testen einer Schaltung
- Zeitverzögerung angeben
  - `timescale 1ns/1ns
  - #3 vor einer Anweisung: 3 ns Verzögerung
  - Taktgenerator: reg clk; initial clk = 1'b1; always #1 clk = !clk;
- Beenden der Simulation
  - bis sich nichts mehr ändert oder
  - explizit durch Befehl $finish;
  - z. B. initial #500 $finish;



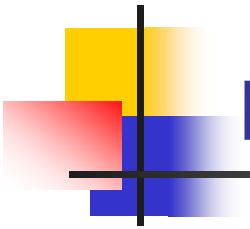
Ausgabe auf die Konsole

- $write (formatstring { , varID })
- $display (formatstring { , varID })
  - wie $write, mit zusätzlichem Zeilenende
- Beispiel:
  $write(" AC hat den Wert %h", ac);

Beispiel
Textausgabe mit Icarus

- Beobachtung des Signals data
- always @(data)
  $display("Wert von data: %b (@%0t) ",
  data, $time);

- Aufruf:
  - iverilog meine_datei.v
  - vvp a.out

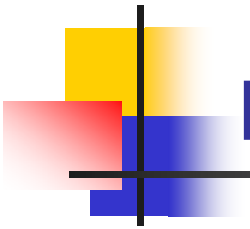


Datenänderung aufzeichnen

- $dumpfile("dateiname.vcd");
  $dumpvars(level);
  $dumpon; $dumpoff;
- level = 1
  - nur Variablen des aktuellen Moduls protokollieren
- level = 0
  - alle Variablen (eigene und auch untergeordnete Module) protokollieren

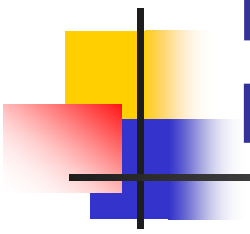
Beispiel
Ausgabe mit Icarus und GTKWave

- initial begin
  $dumpfile("ausgabe.vcd");
  $dumpvars(0);
  end
- Aufruf:
  - iverilog meine_datei.v
  - vvp a.out
  - winwave ausgabe.vcd (mit Windows)
  - gtkwave ausgabe.vcd (mit X, z. B. bei RBG)



Externe Daten einlesen

- $readmemb("Dateiname_binär", memory [, startadresse [, endadresse]])
- $readmemh("Dateiname_hexadezimal", memory [, startadresse [, endadresse]])
- Dateiinhalt:
  - je Adresse ein Zahlenwert der angegebenen Basis
  - mit Leerzeichen oder Zeilenumbruch voneinander getrennt
  - Anzahl Einträge muss stimmen!

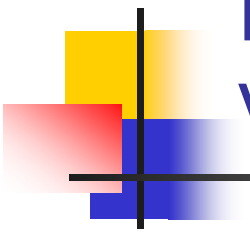


Beispiel
Externe Daten einlesen

- Verilog-Fragment
  - reg [7:0] mem [1:5];
    initial $readmemh("init.txt", mem, 1, 3);
- Dateiinhalt init.txt
  - 9 8 4
- Ergebnis
  - mem[1] = 9, mem[2] = 8, mem[3] = 4
  - mem[4] und mem[5] sind nicht initialisiert

Beispiel
synchroner Vorwärts-/Rückwärts-Zähler

- Eingabe
  - Clock (Takt)
  - Reset (auf Null setzen)
  - Incr (erhöhe um eins)
  - Decr (verringere um eins)
- Ausgabe
  - 2 Bit Wert (Z)



Modul Vorwärts/Rückwärts-Zähler

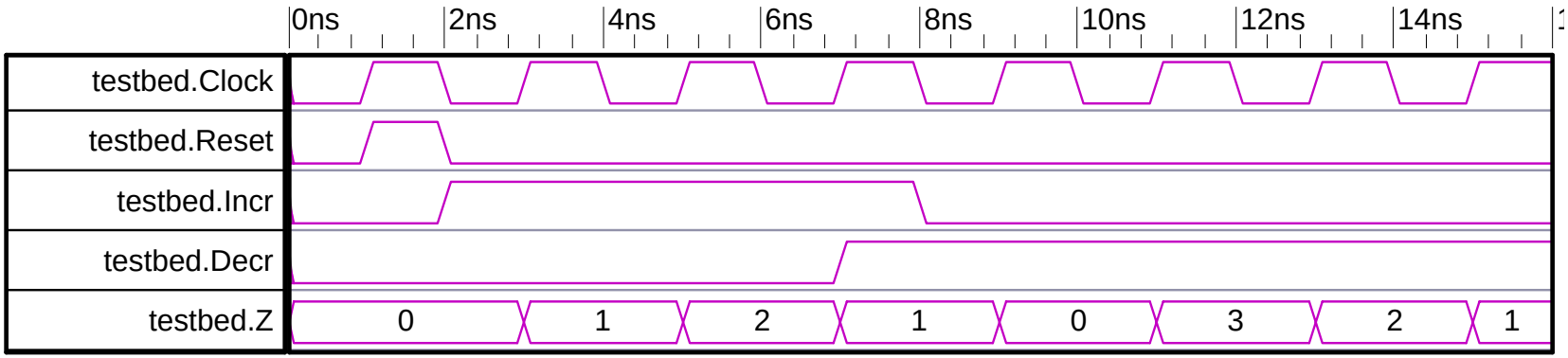
```
module Counter(Clock, Reset, Incr, Decr, Z);
    input  Clock, Reset, Incr, Decr;
    output [1:0] Z;
    reg    [1:0] Z;
    initial Z = 0;
    always @(posedge Clock) begin
        if (Reset)   Z <= 0;
        if (Incr)    Z <= Z + 1;
        if (Decr)    Z <= Z – 1;
    end
endmodule
```

Testumgebung Vorwärts/Rückwärts-Zähler

```
module testbed();
    reg Clock, Reset, Incr, Decr;
    wire [1:0] Z;
    Counter test(Clock, Reset, Incr, Decr, Z);
    initial Clock=0;
    always #1 Clock=~Clock;
    initial  #16 $finish;
    initial ... // Reset, Incr, Decr: nächste Folie
endmodule
```

Zeitdiagramm Vorwärts/Rückwärts-Zähler

```
initial begin
#0  Reset=0; Incr=0;  Decr=0;
#1  Reset=1;          // Zähler auf 0 setzen
#1  Reset=0; Incr=1;// zählen
#5  Decr=1;           // Achtung auf die Konfliktmöglichkeit!
#1  Incr=0; Decr=1;  // ... unterschiedliche Interpretationen!
end
```

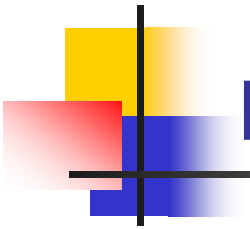


verbessertes Modul Vorwärts/Rückwärts-Zähler

```
module Counter(Clock, Reset, Incr, Decr, Z);
    input  Clock, Reset, Incr, Decr;
    output [1:0] Z;
    reg    [1:0] Z;
    initial Z = 0;
    always @(posedge Clock) begin
        if (Reset)   Z <= 0;       else
        if (Incr)    Z <= Z + 1;   else
        if (Decr)    Z <= Z – 1;
    end
endmodule
```

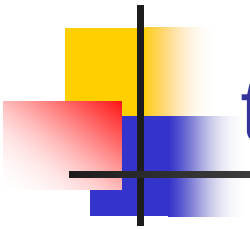
Zustandsmaschine Ampel

```
module ampel(clk, schalte, rot, gelb, gruen);
    input clk, schalte;   output rot, gelb, gruen;
    reg [2:0] state; initial state = 3'd0;
    always @(posedge clk) if (schalte)
            case (state)
            3'b000: state <= 3'b010;
            3'b010: state <= 3'b101;
            3'b101: state <= 3'b011;
            3'b011: state <= 3'b000;
            default  state <= 3'b000;
            endcase
    assign {gruen, gelb} = state[2:1];
    assign rot = !state[0];
endmodule
```



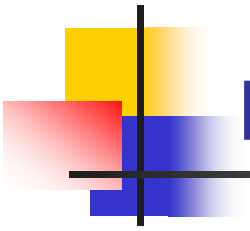
Function

- Wertberechnung
  - Rückgabe nur über Funktionsnamen
  - Parameterübergabe: by value
- function [3:0] max;
  input [3:0] op_a, op_b;
  begin
    max = (op_a >= op_b)? op_a : op_b;
  end
  endfunction
- Aufruf: assign m = max(a, b);



task

- Sammlung von sequential statements
- Ein-/Ausgabe-Parameter
  - bei Aufruf in gleicher Reihenfolge
- Rekursiver Aufruf möglich, jedoch keine automatische Speicherverwaltung (d. h. nur statische Variablen)
- task inc; input [2:0] a, b; begin x <= x + a; y <= y – x + b; end endtask
- Aufruf: always @(posedge clk) inc(5, 8);

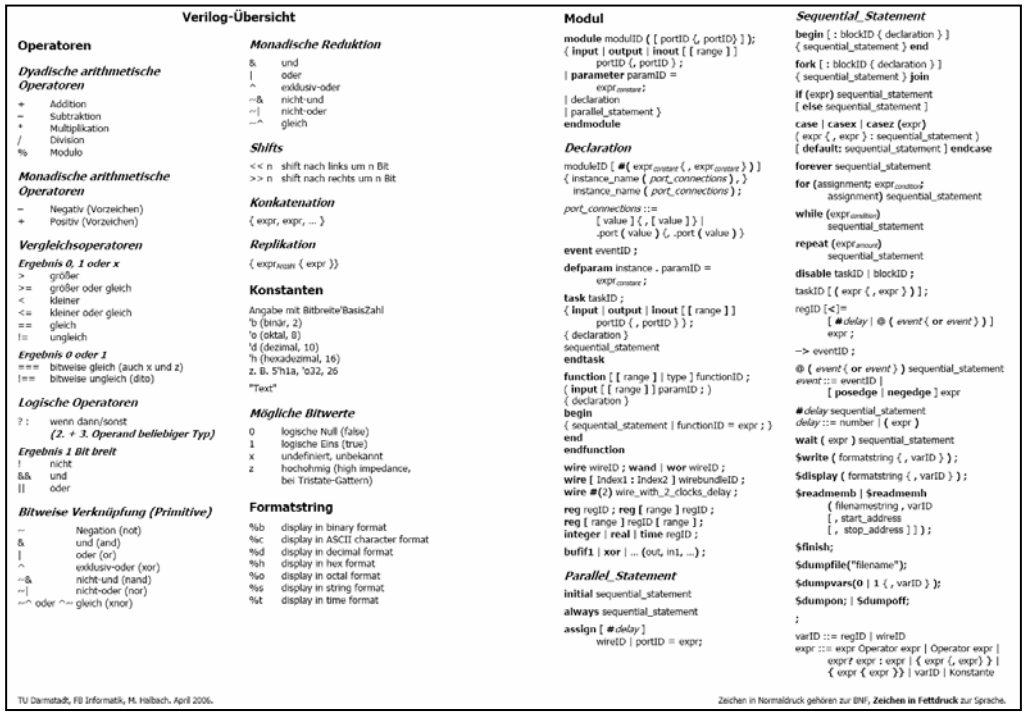


Beispiel 8-Bit-Tri-State-Bus

```
module bustreiber(bus, sende, in, out);
   inout [7 : 0] bus;    input sende;
   output [7 : 0] in;    input [7 : 0] out;
   assign bus = sende? out : 8'bzzzzzzzz;
   assign in = bus;
endmodule
```

Weitere Informationen

- IEEE Standard Verilog® Hardware Description Language, IEEE Std 1364
- Übersichtsblatt (teilweise BNF-Notation)



Verilog-Übersicht

Operatoren

Dyadische arithmetische Operatoren

+	Addition
–	Subtraktion
*	Multiplikation
/	Division
%	Modulo

Monadische arithmetische Operatoren

–	Negativ (Vorzeichen)
+	Positiv (Vorzeichen)

Vergleichsoperatoren

Ergebnis 0, 1 oder x

>	größer
>=	größer oder gleich
<	kleiner
<=	kleiner oder gleich
==	gleich
!=	ungleich

Ergebnis 0 oder 1

===	bitweise gleich (auch x und z)
!==	bitweise ungleich (dito)

Logische Operatoren

? :	wenn dann/sonst
	(2. + 3. Operand beliebiger Typ)

Ergebnis 1 Bit breit

!	nicht
&&	und
||	oder

Bitweise Verknüpfung (Primitive)

~	Negation (not)
&	und (and)
|	oder (or)
^	exklusiv-oder (xor)
~&	nicht-und (nand)
~|	nicht-oder (nor)
~^ oder ^~	gleich (xnor)

Monadische Reduktion

&	und
|	oder
^	exklusiv-oder
~&	nicht-und
~|	nicht-oder
~^	gleich

Shifts

<< n	shift nach links um n Bit
>> n	shift nach rechts um n Bit

Konkatenation

{ expr, expr, ... }

Replikation

{ expr$_{Anzahl}$ { expr }}

Konstanten

Angabe mit Bitbreite'BasisZahl
'b (binär, 2)
'o (oktal, 8)
'd (dezimal, 10)
'h (hexadezimal, 16)
z. B. 5'h1a, 'o32, 26

"Text"

Mögliche Bitwerte

0	logische Null (false)
1	logische Eins (true)
x	undefiniert, unbekannt
z	hochohmig (high impedance, bei Tristate-Gattern)

Formatstring

%b	display in binary format
%c	display in ASCII character format
%d	display in decimal format
%h	display in hex format
%o	display in octal format
%s	display in string format
%t	display in time format

Modul

module modulID ([portID {, portID}]);
{ **input** | **output** | **inout** [[range]]
portID {, portID } ;
| **parameter** paramID =
expr$_{constant}$;
| declaration
| parallel_statement }
endmodule

Declaration

moduleID [**#(** expr$_{constant}$ { , expr$_{constant}$ } **)**]
{ instance_name **(** *port_connections* **)** , }
instance_name **(** *port_connections* **)** ;

port_connections ::=
[value] { , [value] } |
.port **(** value **)** {, .port **(** value **)** }

event eventID ;

defparam instance . paramID =
expr$_{constant}$;

task taskID ;
{ **input** | **output** | **inout** [[range]]
portID { , portID }) ;
{ declaration }
sequential_statement
endtask

function [[range] | type] functionID ;
{ **input** [[range]] paramID ; }
{ declaration }
begin
{ sequential_statement | functionID = expr ; }
end
endfunction

wire wireID ; **wand** | **wor** wireID ;
wire [Index1 : Index2] wirebundleID ;
wire #(2**)** wire_with_2_clocks_delay ;

reg regID ; **reg [** range **]** regID ;
reg [range **]** regID [range] ;
integer | **real** | **time** regID ;

bufif1 | **xor** | ... (out, in1, ...) ;

Parallel_Statement

initial sequential_statement

always sequential_statement

assign [*#delay*]
wireID | portID = expr;

Sequential_Statement

begin [: blockID { declaration }]
{ sequential_statement } **end**

fork [: blockID { declaration }]
{ sequential_statement } **join**

if (expr**)** sequential_statement
[**else** sequential_statement]

case | **casex** | **casez (**expr**)**
{ expr { , expr } : sequential_statement }
[**default:** sequential_statement] **endcase**

forever sequential_statement

for (assignment; expr$_{condition}$;
assignment**)** sequential_statement

while (expr$_{condition}$**)**
sequential_statement

repeat (expr$_{amount}$**)**
sequential_statement

disable taskID | blockID ;

taskID [**(** expr { , expr } **)**] ;

regID [**<**]=
[*#delay* | **@** { *event* { **or** *event* } }]
expr ;

–> eventID ;

@ { *event* { **or** *event* } } sequential_statement
event ::= eventID |
[**posedge** | **negedge**] expr

#delay sequential_statement
delay ::= number | **(** expr **)**

wait (expr **)** sequential_statement

$write (formatstring { , varID } **)** ;

$display (formatstring { , varID } **)** ;

$readmemb | $readmemh
(filenamestring , varID
[, start_address
[, stop_address]]) ;

$finish;

$dumpfile("filename**");**

$dumpvars(0 | 1 { , varID } **);**

$dumpon; | $dumpoff;

;

varID ::= regID | wireID
expr ::= expr Operator expr | Operator expr |
expr ? expr : expr | { **{** expr {, expr} **}** |
{ expr { expr **}}** | varID | Konstante

TU Darmstadt, FB Informatik, M. Halbach, April 2006.

Zeichen in Normaldruck gehören zur BNF, **Zeichen in Fettdruck** zur Sprache.