

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

Compiler-Direktive

`define ID text
`ID wird durch text ersetzt

Modul

```
module modulID ( [ portID {, portID} ] );  
{ input | output | inout [ [ range ] ]  
  portID {, portID } ;  
| parameter paramID = exprconstant ;  
| declaration  
| parallel_statement }  
endmodule
```

Declaration

```
moduleID [ #( exprconstant {, exprconstant } ) ]  
{ instance_name ( port_connections ), }  
  instance_name ( port_connections ) ;
```

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

```
event eventID ;
```

```
defparam instance . paramID =  
  exprconstant ;
```

```
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 {, wireID } ;  
wand | wor wireID ;  
wire [ Index1 : Index2 ] wireID ;  
wire #(2) wire_with_2_clocks_delay ;  
wire wireID = expr {, wireID = expr } ;
```

```
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 ] v = expr ;  
v ::= v[exprconstant [: exprconstant]] | {v {, v}} |  
  wireID | portID
```

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; exprcondition;  
  assignment) sequential_statement
```

```
while (exprcondition)  
  sequential_statement
```

```
repeat (expramount)  
  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 } );  
$monitor( 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
```