Die Hardwarebeschreibungssprache Verilog
Eine kurze Einführung

25. 1. 2010

Diese Kurzbeschreibung betrifft Schaltungsbeschreibungen mit dem Ziel der Synthese. Sie beschränkt sich auf das Nötigste (Grundgedanke: in möglichst kurzer Zeit zu ersten Erfolgserlebnissen). Die Simulation wird nicht betrachtet.

Module

Die Grundform der Schaltungsbeschreibung ist die Modulbeschreibung. Das Modul (module) ist die Entsprechung zur Funktion in einer höheren Programmiersprache. In einer Modulbeschreibung können weitere Module aufgerufen werden. Das jeweilige Entwurfsvorhaben wird insgesamt als ein Modul beschrieben. Dies ist die oberste Beschreibungsebene (Top-Level Module), vergleichbar zum Hauptprogramm (Main) in einer höheren Programmiersprache.

Der grundsätzliche Aufbau einer Modulbeschreibung:

module Bezeichner (Schnittstellensignale);
Definition der Signale;

Beschreibung;

endmodule

Beschreibungsarten:

- Strukturbeschreibung,
- RTL-Beschreibung,
- Verhaltensbeschreibung.

Typische Nutzungsweisen:

- Die obersten Beschreibungsebenen: Strukturbeschreibung. Komplexe Entwürfe bestehen aus Funktionseinheiten, die ihrerseits als Module beschrieben werden. Hier wird angegeben, wie die Funktionseinheiten untereinander verbunden sind.
- Beschreibung von Modulen mit einem allgemein üblichen, überschaubaren Funktionsumfang: Verhaltensbeschreibung.
- Beschreibung von Modulen, wenn es auf Spitzfindigkeiten ankommt: RTL-Beschreibung.
- Beschreibung zu Simulationszwecken: Funktionsbeschreibung.

Die Verhaltensbeschreibung ist nur dann anwendbar, wenn es überschaubare Funktionen sind oder wenn es auf den Ressourcenverbrauch oder auf die Verzögerungszeiten nicht ankommt. Auf die für die jeweilige Schaltkreisbaureihe jeweils empfohlenen Codierrichtlinien (Coding Styles) achten. Manche Funktionen können mit den Mitteln der Verhaltensbeschreibung nicht dargestellt werden (z. B. Taktsteuerung oder DDR-Flipflops). Solche Funktionen müssen als fertige Funktionsblöcke in eine Strukturbeschreibung eingebaut werden (Instantiation).

Mit der Verhaltensbeschreibung kann z. B. ausgedrückt werde, daß zwei Zahlenwerte zueinander addiert werden sollen, nicht aber, daß hierfür beispielsweise ein Carry-Select-Addierer verwendet werden soll. Die Auswege: (1) RTL-Beschreibung, (2) Einbauen entsprechender fertiger Module (Instantiation). Geht es um programmierbare Schaltkreise, so ist die RTL-Beschreibung üblicherweise günstiger als eine Strukturbeschreibung auf Gatterebene, da der Synthesizer die Gatteranordnung ohnehin in eine Belegung der Logikzellen umwandeln muß. Dann sollte er eigentlich von der Booleschen Gleichung aus zum gleichen Ergebnis kommen (ggf. anhand einiger Probeentwürfe nachsehen, was das jeweilige System leistet).

Reservierte Schlüsselwörter:

always	and	assign	begin
buf	bufif0	bufif1	case
casex	casez	cmos	deassign
default	defparam	disable	edge
else	end	endcase	endfunction
endmodule	endprimitive	endspecify	endtable
endtask	event	for	force
forever	fork	function	highz1
highz0	if	initial	inout
input	integer	join	large
macromodule	medium	module	nand
negedge	nmos	nor	not
notif0	notif1	or	output
parameter	pmos	posedge	primitive
pull1	pull0	pullup	pulldown
rcmos	real	reg	release
repeat	rnmos	rpmos	rtran
rtranif1	rtranif0	scalared	small
specify	specparam	strong1	strong0
supply1	supply0	table	task
time	tran	tranif1	tranif0
tri	tri1	tri0	triand
trior	trireg	vectored	wait
wand	weak1	weak0	while
wire	wor	xnor	xor

Auf die Groß- und Kleinschreibung achten.

Kommentare:

- In einer Zeile: //.....
- Über mehrere Zeilen hinweg: /*.... */

(Vgl. Programmiersprache C)

Signaldefinitionen

Ein- und Ausgänge:

input
output
inout

Speicherelemente (= Flipflops oder Register; Verhaltensbeschreibung):

reg

Signalleitungen; Verbindungen:

wire	binäre Verbindung
wand	Wired-And-Verbindung
wor	Wired-OR-Verbindung
tri	Tri-State-Verbindung
tri0	Tri-State-Verbindung. Führt Nullpegel, wenn inaktiv.
tri1	Tri-State-Verbindung. Führt Einspegel, wenn inaktiv.
trireg	Tri-State-Verbindung. Führt den vorhergehenden aktiven Pegel, wenn inaktiv.
trior	Tri-State-Verbindung mit Wired-Or-Verknüpfung.
triand	Tri-State-Verbindung mit Wired-AND-Verknüpfung.
supply0	Festwert Null
supply1	Festwert Eins

Syntax:

Einzelsignale:
Definitionsbezeichner Liste der Signalbezeichner;

Bussignale:
Definitionsbezeichner [Bereichsangabe] Liste der Signalbezeichner;

Bereichsangabe = MSB_index : LSB_Index

Beispiele:

input A, B, C;

wire w1;

reg [7:0] count;

Wenn die Anzahl der Signale veränderlich sein soll:

parameter w = 11;
reg [w:0] count; entspricht **reg** [11:0] count;

Strukturbeschreibung

Beschreibung der Verbindungen zwischen Modulen. Die einfachsten Module sind die Gatter.

AND
OR
NOT
NAND
NOR
XOR
XNOR

Gatterbeschreibung:

Gattertyp Gatterliste;

Ein Element der Gatterliste:

(Ausgang, 1. Eingang, 2. Eingang, ...)

Einfügen anderer Module (Instantiation):

Modultyp Modulliste;

Ein Element der Modulliste:

Einzelname (Signale gemäß Strukturbeschreibung des Modultyps)

Signalangaben
Es gibt zwei Varianten:

1. Positionsangabe (Positional Notation)
Aufzählung der Signale gemäß der Beschreibung des Modultyps. Es kommt auf die Reihenfolge an (wie Funktionsaufruf in höheren Programmiersprachen).

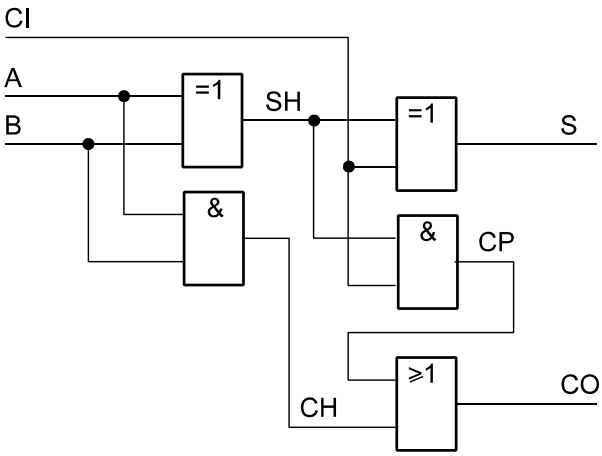
2. Namensangabe (Named Notation)
Direkte Zuweisung der jeweiligen Signale zu den Signalen in der Beschreibung des Modultyps:

.a(b) heißt:

Das Signal a gemäß Modulbeschreibung ist mit dem Signal b des aktuellen Moduls beschaltet. Reihenfolge gleichgültig. Nicht belegte Signale des Moduls bleiben offen.

Wird ein Modul eingefügt, so können dessen Ausgangs-Ports entweder nur vollständig oder gar nicht genutzt werden. Ist beispielsweise ein Ausgang mit acht Bitpositionen definiert, so muß man alle acht Bits anschließen, oder man nutzt den Ausgang gar nicht. Es ist nicht möglich, beliebige Einzelverbindungen herzustellen, beispielsweise zu den Bits 3 und 5.

Beispiel: 1-Bit-Addierer



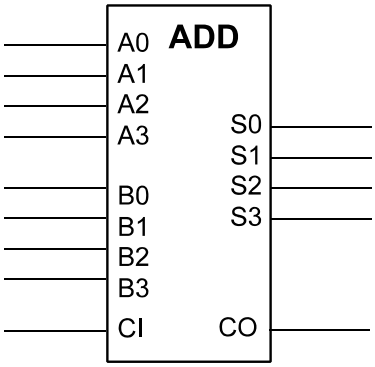

```
module ADDBIT (S, CO, A, B, CI);

output    S, CO;
input     A, B, CI;
wire      SH, CH, CP;

xor       (SH, A, B), (S, SH, CI);
and       (CH, A, B), (CP,CI, SH);
or        (CO,CP, CH);

endmodule
```

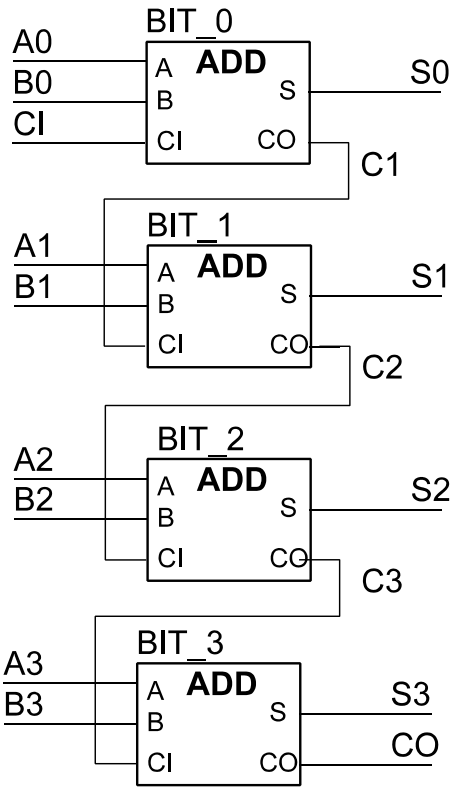
Es ist zulässig, in der Strukturbeschreibung Ausdrucksmittel der RTL-Beschreibung zu verwenden.

```
module addbit(S, CO, A, B, CI);     // Mit Zuweisungen in den Gatterangaben
output S, CO;
input A, B, CI;

xor (S, A^B, CI);
or  (CO, CI & (A |B), A&B);

endmodule
```

Beispiel: 4-Bit-Addierer, der aus 1-Bit-Addierern aufgebaut wird.

Schaltsymbol:





module VOLLAD (S, CO, A, B, CI); // Aufruf mit Positionsangabe

output [3:0] S;
output CO;
input [3:0] A, B;
input CI;
wire C1, C2, C3;

// Es kommen vier Exemplare von 1-Bit-Addieren zum Einsatz (A0 bis A3).

ADDBIT BIT_0 (S[0], C1, A[0], B[0], CI), BIT_1 (S[1], C2, A[1], B[1], C1),
BIT_2 (S[2], C3, A[2], B[2], C2), BIT_3 (S[3], CO, A[3], B[3], C3);

endmodule

Beispiel eines Moduls mit Namensangabe:

BIT_2 (.A(A[2]), .B(B[2]), .CI(C2), .S(S[2]), .CO(C3))

RTL-Beschreibung

Die Schaltung wird nicht durch Gatter oder Module und Verbindungen beschrieben, sondern durch Boolesche Gleichungen und Operandenverknüpfungen. Die Symbolik entspricht weitgehend der Programmiersprache C

Bitweise logische Verknüpfungen:

~	NICHT
|	ODER
&	UND
^	XOR

NAND, NOR, XNOR ergeben sich durch Negieren der jeweils entsprechenden Verknüpfung.

Beispiele: ~(A & B), ~(A | B) , ~(A ^ B).

Reduktionsoperatoren
Hat eine logische Verknüpfung UND, ODER, NAND, NOR, XOR, XNOR nur einen Operanden, so wird die Verknüpfung quer über alle Operandenbits ausgeführt.

|	ODER
&	UND
~&	NAND
~|	NOR
^	XOR
^~	XNOR

Logische Operatoren (wie in C)
Wert Null entspricht logisch Null (FALSE), Wert ungleich Null entspricht logisch Eins (TRUE):

!	NICHT
&&	UND
||	ODER
= =	GLEICH
!=	UNGLEICH

Verschiebeoperatoren:

<<	Linksverschiebung
>>	Rechtsverschiebung

(Operand << Anzahl der Bitpositionen, über die zu verschieben ist).

Arithmetische Operatoren:

+, –, *, /, % (modulo(Rest))

Relationale Operatoren:

>, >=, <, <=

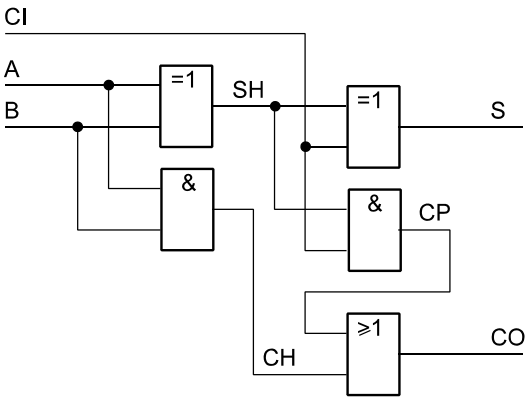
IF-Operator (?)

(Ausdruck) ? Anweisung bei TRUE; Anweisung bei FALSE.

Verkettung von Bitpositionen:

{ } Einschließen der zu verkettenden Ausdrücke in geschweifte Klammern

Beispiel: 1-Bit-Addierer




```
module ADDBIT_RTL_1 (S, CO, A, B, CI);

output    S, CO;
input     A, B, CI;
wire      SH, CH, CP;

assign    SH =   A ^ B;
assign    CH =   A & B;
assign    S =    SH ^ CI;
assign    CP =   SH & CI;
assign    CO =   CP | CH;

endmodule
```

Alternative: direkte Zuweisung gemäß den Booleschen Gleichungen der binären Addition:

```
module ADDBIT_RTL_2 (S, CO, A, B, CI);

output    S, CO;
input     A, B, CI;

assign    S =     A ^ B ^ CI;
assign    CO =    (A & B) | (CI & (A | B));

endmodule
```

Alternative: Summenbildung mit Additionsoperator

```
module ADDBIT_RTL_3 (S, CO, A, B, CI);

output    S, CO;
input     A, B, CI;

assign    S =     A + B + CI;
assign    CO =    (A & B) | (CI & (A | B));

endmodule
```

Beispiel: 4-Bit-Addierer mit ausgangsseitiger Nullerkennung (1. Versuch)

```
module VOLLAD_RTL (S, CO, A, B, CI);

output [3:0]      S;
output            CO, ZERO;
input [3:0]       A, B;
input             CI;

assign            {CO, S} = A + B + CI;
assign            ZERO = ~|S;

endmodule
```

Die Verkettung der Ergebnisbits klappt aber nicht. Deshalb ein zweiter Versuch. Um Ausgangsübertrag und Summe aus dem Additionsergebnis entnehmen zu können, wird eine Hilfsstruktur AUX von 5 Bits Länge eingeführt.

```
module VOLLAD_RTL (S, CO, A, B, CI);

output [3:0]      S;
output            CO, ZERO;
input [3:0]       A, B;
input             CI;
wire [4:0]        AUX;

assign            AUX = A + B + CI;
assign            S = AUX[3:0];
assign            CO = AUX[4];
assign            ZERO = ~| AUX[3:0];

endmodule
```

Verhaltensbeschreibung

Es werden die Informationswandlungen zwischen Eingängen, Speichergliedern und Ausgängen beschrieben. Die Algorithmen werden ausgeführt, wenn bestimmte Ereignisse eintreten. Die Ereignisse werden mit **initial**- und **always**-Anweisungen ausgewertet. Die jeweils auszuführenden Algorithmen sind in die entsprechenden **initial**- oder **always**-Blöcke einzufügen. **always**-Blöcke werden immer wieder (zyklisch) durchlaufen, **initial**-Blöcke nur einmal.

Hinweis:
initial-Blöcke dienen nur dazu, die Anfangswerte für die Simulation einzustellen. Sind Rücksetzvorgänge durch Signale auszulösen (in zu synthetisierenden Schaltungen), so müssen sie in entsprechenden **always**-Blöcken beschrieben werden.

Struktur eines **always**-Blockes:

always @ (*Ereignis*)
    Anweisung;

oder

always @ (*Ereignis*)

begin
    Anweisungen;
end

Struktur einer Ereignisangabe:

- Signaländerung allgemein: Signalbezeichner
- Signalflanke von Low nach High: **posedge** Signalbezeichner
- Signalflanke von High nach Low: **negedge** Signalbezeichner

Ereignisangaben können disjunktiv verknüpft werden (**or**-Anweisung).

Zuweisungen (Assignments)
Zuweisungen dienen dazu, Speicher- oder Ausgangssignale mit den Ergebnissen von Informationswandlungen zu belegen. Es gibt zwei Arten der Zuweisung: die blockierende und die nichtblockierende Zuweisung (Blocking / Non-Blocking Assignment).

Grundschema:

Zu belegendes Signal Zuweisungssymbol Algorithmus, der die Signalbelegung bestimmt;

Blockierende Zuweisung (=)
Jede einzelne dieser Zuweisungen wird unmittelbar ausgeführt.

```
Zu belegendes Signal  =  Algorithmus, der die Signalbelegung bestimmt;
```

Nichtblockierende Zuweisung (<=)
Alle zuzuweisenden Werte in einem **always**-Block werden auf Grundlage der jeweils vorliegenden Signalbelegung gebildet. Tritt das Ereignis auf, werden sie alle auf einmal zugewiesen.

```
Zu belegendes Signal  <=  Algorithmus, der die Signalbelegung bestimmt;
```

Hinweise:

1. Die nichtblockierende Zuweisung bildet das typische Verhalten vollsynchroner, flankengesteuerter Schaltwerke oder Zustandsautomaten nach. (Mit den aktuellen Inhalten der Flipflops werden die Eingangsbelegungen gebildet, die mit der jeweils nächsten Taktflanke in die Flipflops übernommen werden.)

2. In einem **always**-Block sollte jeweils nur eine Art von Zuweisungen vorkommen. Bei der Synthese werden typischerweise blockierende Zuweisungen in sequentiellen **always**-Blöcken als nichtblockierende Zuweisungen implementiert. Bei der Simulation kann es aber Probleme geben. Deshalb in sequentiellen **always**-Blöcken stets <= verwenden.
3. Zuweisungsziele müssen stes als **reg** definiert sein, auch dann, wenn nur eine kombinatorische Zuweisung beabsichtigt ist.

Einfachbeispiele mit blockierenden Zuweisungen:

1. 1-Bit-Addierer

```
module ADDBIT_BHV (S, CO, A, B, CI);

output      S, CO;
input       A, B, CI;
reg [1:0]   AUX;
reg         S, CO;

always @ (A or B or CI)
    begin
            AUX = A + B + CI;
            S = AUX[0];
            CO = AUX[1];
    end

endmodule
```

2. 4-Bit-Volladierer mit Anzeige der Nullbedingung

```
module VOLLAD_BHV (S, CO, ZERO, A, B, CI);

output [3:0]        S;
output              CO, ZERO;
input [3:0]         A, B;
input               CI;
reg [4:0]           AUX;
reg [3:0]           S;
reg                 CO, ZERO;

always @ (A or B or CI)
    begin
            AUX = A + B + CI;
            S = AUX[3:0];
            CO = AUX[4];
            ZERO = ~| AUX[3:0];
    end

endmodule
```

Einfachbeispiele mit nichtblockierenden Zuweisungen (synchrone Schaltwerke)

Grundsatz: Es kann nicht kombinatorisch ausgegeben werden, sondern nur auf Register.

1. Ein ganz einfaches D-Flipflop:

```
module DFF (Q, D, C);

output      Q;
input       D, C;
reg         Q;

always @ (posedge C)
            Q <= D;

endmodule
```

2. D-Flipflop mit asynchronem Rücksetzen (1. Versuch):

```
module DFF_CLR (Q, D, C, CLR);

output      Q;
input       D, C, CLR;
reg         Q;

always @ (CLR)

            Q = 0;

always @ (posedge C)

            Q <= D;

endmodule
```

So geht es aber nicht ... Kombinatorische und flankengesteuerte Zuweisung lassen sich nicht mischen.

Flipflops müssen nach einem bestimmten Schema beschrieben werden, damit das Syntheseprogramm die Absicht erkennen kann:

1. Asynchrone Setz- oder Rücksetzsignale sind als flankengetriggert einzuführen und demgemäß in die Ereignisliste der always-Anweisung aufzunehmen.
2. Die erste auf always folgende Anweisung muß ein if sein.
3. Die asynchronen Setz- und Rücksetzfunktionen sind zuerst zu erledigen.

Setzen oder Rücksetzen aktiv High:

```
always @ (... posedge SET ...)

        begin
                if (SET) ...
        ...
```

Setzen oder Rücksetzen aktiv Low:

```
always @ (... negedge SET ...)

        begin
                if (~SET) ...
        ...
```

Beispiel:

```
module DFF_CLR (Q, D, C, CLR#);

output          Q;
input           D, C, CLR#;
reg             Q;

        always @ (posedge C or negedge CLR#)

        begin

                if (~CLR#) Q <= 0;
                else Q <= D;
        end

endmodule
```

Das Entwicklungssystem erkennt die Entwurfsabsicht und erzeugt ein D-Flipflop mit asynchronem Rücksetzen. Latches werden nach einem ähnlichen Schema beschrieben.

Beispiele:

```
module SIMPLE_D_LATCH (Q, D, G);

output          Q;
input           D, G;
reg             Q;

        always @ (G)

                Q = D;

endmodule

module D_LATC_W_CLEAR(Q, D, G,CLR#);

output          Q;
input           D, G, CLR#;
reg             Q;

always @ (G or CLR#)

        begin
                if (~CLR#)    Q = 0;
                else if (G)   Q = D;
        end

endmodule
```

Ein einfacher 24-Bit-Zähler, dessen höchstwertige acht Bits ausgegeben werden (LED-Anzeige):

```
module SIMPLE_CTR (C, LED);

output [7:0]    LED;
input           C;
reg [23:0]      CNT;
reg [7:0]       LED;

always @ (posedge C)

                begin

                        CNT <= CNT+1;
                        LED <= CNT[23:16];
                end

endmodule
```

Ausgänge können nur Register sein. Zum Ausgeben der höchstwertigen acht Bits wurde hier ein eigenes Ausgaberegister vorgesehen. Die Synthese ergibt deshalb 32 Zellen statt 24.

Ein Versuch, um mit 24 Zellen auszukommen:
Ist ein Ausgangsport einmal definiert, so muß das betreffende Register insgesamt angeschlossen werden. Deshalb wird der Zähler in zwei Teile zerlegt:

```
module SIMPLE_CTR_24_8 (C, LED);

output [7:0]        LED;
input               C;
reg [15:0]          CNT;
reg [7:0]           LED;

always @ (posedge C)

                    begin
                        if (CNT == 'hffff)
                                        LED <= LED+1;
                        CNT <= CNT+1;
                    end

endmodule
```

Jetzt soll dieser Zähler als Funktionseinheit in ein weiteres Modul eingebaut werden (Instantiation). Es ist nicht möglich, einen Ausgang von 24 Bits zu definieren und dort nur acht Bits anzuschließen:

```
module CTR_24_8 (C, CD);

output [23:16]      CD;
input               C;
reg [23:0]          CNT;

    SIMPLE_CTR_24_8 SCTR (.C(C), .LED ({CD[23], CD[22], CD[21], CD[20], CD[19],
    CD[18], CD[17], CD[16]});

endmodule
```

– Beim Definieren von Modulen, die als Bausteine verwendet werden sollen, daran denken, was ausgangsseitig angeschlossen werden sollen. Ggf. einen Port mit vielen Bits in mehrere kleinere Ports auflösen oder ihn mit Parameter definieren. –

```
module PARAMETRIC_CTR_16_LED (C, LED);

parameter               LED_WIDTH = 4;
output [LED_WIDTH:0]    LED;
input                   C;
reg [15:0]              CNT;
reg [LED_WIDTH:0]       LED;

always @ (posedge C)

                begin
                        if (CNT == 'hffff)
                                        LED <= LED+1;
                        CNT <= CNT+1;
                end

endmodule
```

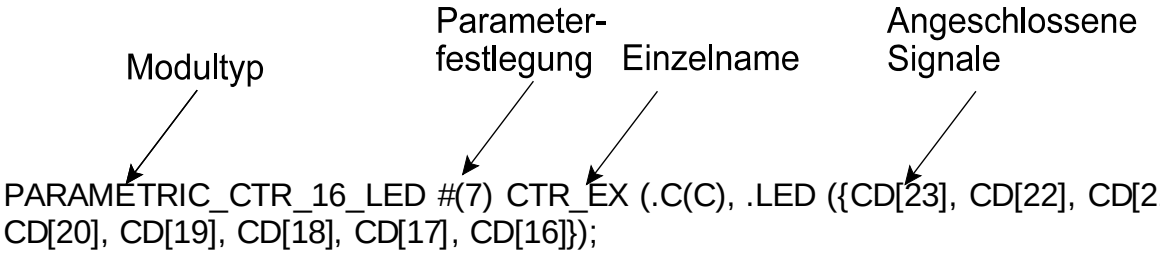
Jetzt wird ein solcher Zähler in ein anderes Modul eingebaut.

Aufruf:

Modulbezeichner # (Parameterwertliste) Einzelname (Signalangaben);

```
module CTR_24_PARM (C, CD);

output [23:16]      CD;
input               C;
reg [23:0]          CNT;

    PARAMETRIC_CTR_16_LED #(7) CTR_EX (.C(C), .LED ({CD[23], CD[22], CD[21],
    CD[20], CD[19], CD[18], CD[17], CD[16]});

endmodule
```

Einzelheit Modulaufruf:




```
PARAMETRIC_CTR_16_LED #(7) CTR_EX (.C(C), .LED ({CD[23], CD[22], CD[21],
CD[20], CD[19], CD[18], CD[17], CD[16]});
```

Bedingte Anweisungen

if (Bedingung)
Anweisung;

else
Anweisung;

Fallunterscheidungen

case (Signal)

1. Signalwert: Anweisung;

2. Signalwert: Anweisung;

usw.

default: Anweisung, wenn keiner der angegebenen Signalwerte vorliegt;

endcase

Einbeziehen von Don’t Cares: **casex** statt **case**.

Einbeziehen von hochomigen Zuständen: **casez** statt **case**.

Handelt es sich um mehrere Anweisungen, diese jeweils in **begin**-**end**-Blöcke einsetzen.

Ein vollsynchrones Mehrfunktionsregister mit acht Bits und den Funktionen

- CLR = Löschen,
- LD = Laden,
- CTF = Vorwärtszählen,
- CTR = Rückwärtszählen.

Die Priorität nimmt in der Reihenfolge der Aufzählung ab.

```
module REG_8_1 (D_OUT, D_IN, CLR, LD, CTF, CTB, C);

output [7:0]        D_OUT;
input [7:0]         D_IN;
input               CLR, LD, CTF, CTB, C
reg [7:0]           D_OUT;

        always @ (posedge C)

                begin

                if (CLR)
                        D_OUT <= 0;
                else if (LD)
                        D_OUT <= D_IN;
                else if (CTF)
                        D_OUT <= D_OUT+1;
                else if (CTB)
                        D_OUT <= D_OUT -1;
                end

endmodule
```

Das gleiche Register mit **casex***-Anweisung:*

```
module REG_8_2 (D_OUT, D_IN, CLR, LD, CTF, CTB, C);

output [7:0]        D_OUT;
input [7:0]         D_IN;
input               CLR, LD, CTF, CTB, C
reg [7:0]           D_OUT;

        always @ (posedge C)

                casex  ({CLR, LD, CTF, CTB})

                        4'b1xxx:        D_OUT <= 0;

                        4'b01xx:        D_OUT <= D_IN;


                        4'b001x:        D_OUT <= D_OUT+1;

                        4'b0001:        D_OUT <= D_OUT -1;

                endcase

endmodule
```

Verhaltensbeschreibung von Zustandsautomaten (Prinzip):

```
always @ (posedge CLOCK or posedge RESET)  // Beispiel für den Maschinentakt
    begin
        if (RESET) STATE <= 0;
        else
        case (STATE)
        0:          begin
                        STATE <= ...;
                        OUTPUT <= ...;
                    end

        1:          begin
                        STATE <= ...;
                        OUTPUT <= ...;
                    end
        2:          begin
                        STATE <= ...;
                        OUTPUT <= ...;
                    end
        default:                    // Reaktion auf einen undefinierten Zustand
                    begin
                        STATE <= ...;  // Beispielsweise zurück zum Anfang
                        OUTPUT <= ...;
                    end
        endcase
    end
```