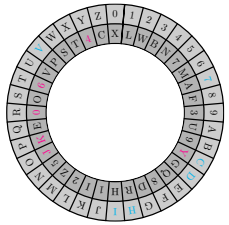
Hochschule **RheinMain**
University of Applied Sciences
Wiesbaden Rüsselsheim



HARDWARE-BESCHREIBUNGSSPRACHEN

Hardwareentwurf von digitalen Schaltungen

12. November 2024
Revision: a01c695 (2024-11-12 16:52:57 +0100)

Steffen Reith

Theoretische Informatik
Studienbereich Angewandte Informatik
Hochschule **RheinMain**

EIN VERILOG-CRASHKURS

EINLEITUNG

Verilog wurde 1983/1984 zunächst als Simulationssprache entwickelt, von Cadence aufgekauft und 1990 frei gegeben.

Erste Standardisierung 1995 durch die IEEE (Verilog 95). **Neuere Version** IEEE Standard 1364–2001 (**Verilog 2001**).

- → **Syntax** vergleichbar mit **C** (VHDL ist an ADA / Pascal angeleht) mit kompakten Code
- → **Verbreitet** in **Nordamerika** und **Japan** (**weniger** in **Europa**)
- → Kann auch als Sprache von **Netzlisten** verwendet werden
- → Unterstützung durch **Open-Source-Tools**
- → Die **Mehrheit** der **ASICs** wird in **Verilog** entwickelt.
- → **Weniger ausdruckstark** als **VHDL** (Fluch und Segen)

EINLEITUNG (TEIL II)

Die Nähe zu C und Java führt evtl. zu Verwechselungen! Auch in Verilog können z.B. Zeilen, die eine kombinatorische Schaltung beschreiben vertauscht werden.

Verilog ist eine **Hardwarebeschreibungssprache**!

In diesem Abschnitt legen wir auf auf einen Subset von **synthetisierbaren** Sprachkonstrukten fest.

Ziel unserer Auswahl sind **nicht** kommerzielle Tools, sondern offene Entwicklungswerkzeuge wie **OpenRoad**[1] oder Toolchains für bekannte FPGAs, d.h. wir verwenden auch **einige** Sprachkonstrukte von SystemVerilog, die durch das **Synthesewerkzeug** `yosys` **unterstützt** werden.

[1] siehe **https://theopenroadproject.org/**

LITERATUR & VERILOG

- → Donald E. Thomas, Philip R. Moorby, Hardware Description Language, Kluwer Academic Publishers, 2002
- → Blaine Readler, Verilog by example, Full Arc Press, 2011

Man sollte sich auch mit den Eigenheiten des Synthesetools beschäftigen! Das bekannte Open-Source-Synthesetool yosys schreibt dazu

> *Yosys is a framework for **Verilog RTL synthesis**. It currently has extensive Verilog-2005 support and provides a basic set of synthesis algorithms for various application domains. Selected features and typical applications:*
>
> - → *Process **almost any synthesizable Verilog-2005 design***
> - → *Converting Verilog to BLIF / EDIF/ BTOR / SMT-LIB / simple RTL Verilog / etc.*
> - → *...*

PRINZIPIELLER AUFBAU EINES VERILOG-MODULES

```
module module_name (port_list);

// Definition der Schnittstelle
Port-Deklaration
Parameter-Deklaration

// Beschreibung des Schaltkreises
Variablen-Deklaration
Zuweisungen
Modul-Instanzierungen
always-Blöcke

endmodule
```

In modernem Verilog können **Portliste und Portdeklaration zusammengezogen** werden. // leitet einen Kommentar ein.

BEISPIEL - EIN LINEARES SCHIEBEREGISTER

```
module LSFR (

input  wire   load,
input  wire   loadIt,
input  wire   enable,
output wire   newBit,
input  wire   clk,
input  wire   reset);

wire       [17:0]   fsRegN;
reg        [17:0]   fsReg;
wire                taps_0, taps_1;
reg                 genBit;

assign taps_0 = fsReg[0];
assign taps_1 = fsReg[11];

always @(*) begin

  genBit = (taps_0 ^ taps_1);

  if(loadIt) begin

    genBit = load;

  end

end

assign newBit = fsReg[0];
assign fsRegN = {genBit,fsReg[17 : 1]};

always @(posedge clk) begin

  if(reset) begin

    fsReg <= 18'h0;

  end else begin

    if(enable) begin

      fsReg <= fsRegN;

    end

  end

end

endmodule
```

Dabei geben **input** und **output** die **Richtung** des **Ports** an.

KONSTANTEN & OPERATOREN

Es gibt vier Werte für eine Konstante / Signal: 0, 1, X bzw. x (unbekannt) und Z bzw. z (hochohmig).

Man kann die Breite von Konstanten angeben:

- → **Hexadezimalkonstante** mit 32 Bit: `32'hDEADBEEF`
- → **Binärkonstante** mit 4 Bit: `4'b1011`
- → Zur **besseren Lesbarkeit** kann man auch den **Underscore** verwenden: `12'b1010_1111_0001`

Zur Spezifikation der **Zahlenbasis** sind `b` (binär), `h` (hexadezimal), `o` (oktal) und `d` (dezimal) zulässig.

Der **Default ist dezimal (d)** und die Bitbreite ist optional, d.h. `4711` ist eine zulässige (dezimal) Konstante.

KONSTANTEN & OPERATOREN (II)

Passend zu den Konstanten existiert eine **Array-Schreibweise**:

- → `wire [7:0] serDat;` oder `reg [0:32] shiftReg;`
- → Einzelne Bits können mit **gesliced** werden: `serDat[3 : 0]` (low-nibble) oder `serDat[7]` (MSB).
- → `{serDat[7:6], serDat[1:0]}` notiert die **Konkatenation**.
- → Bits können **repliziert** und in ein Array umgewandelt werden, d.h. `{8{serData[7 : 4]}}` enthält acht Kopien des high-nibble von `serDat` und hat eine Breite von 32.

Weiterhin existieren die üblichen **arithmetischen Operationen**, **Ordnungsrelationen**, **Äquivalenzen** und Negation:

- → `a + b`, `a - b`, `a * b`, `a / b` und `a % b`
- → `a > b`, `a <= b`, und `a >= b`
- → `a == b` und `a != b`,
- → `!(a = b)`

KONSTANTEN & OPERATOREN (III)

Achtung: Kommen `x` oder `z` vor, so ermittelt der Simulator bei einem Vergleich `false`. Will man dies vermeiden, so existieren die Operatoren `===` und `!==`. Also gilt:

```
if (4'b110z === 4'b110z)
  // taken
  then_statement;
```

```
if (4'b110z == 4'b110z)
  // not taken
  then_statement;
```

Es existieren auch **boolesche Operationen** wie gewohnt:

bitweise Operatoren: `&` (UND), `|` (ODER, `~` (NICHT), `^` (XOR) und auch `~^` (XNOR)

logische Operatoren: `&&` (UND), `||` (ODER) und `!` (NICHT)

Schiebeoperation: `a << b` (schiebe `a` um b Stellen nach links) und `a >> b` (verschiebe `a` um b Positionen nach rechts). Eine negative Anzahl `b` ist nicht zulässig, leere Stellen werden mit 0 aufgefüllt.

PARAMETER (OLD STYLE)

Um **Designs leichter anpassen** zu können, bietet Verilog die Verwendung von **Parametern** an.

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

  parameter WIDTH = 8;  // Anzahl der Bits

  input  [WIDTH-1:0] in1, in2;
  input sel;
  output [WIDTH-1:0] out;

  assign out = sel ? in1 : in2;

endmodule
```

INSTANZEN UND STRUKTURELLE BESCHREIBUNGEN

Beschreibt man einen Schaltkreis **durch seine (interne) Struktur** oder soll ein Teilschaltkreis **wiederverwendet** werden, dann wird eine Instanz erzeugt und verdrahtet.

```
module xor2 (
  input  wire a,
  input  wire b,
  output wire e);
  assign e = a ^ b;
endmodule

module xor3 (
  input  wire a,
  input  wire b,
  input  wire c,
  output wire e);

  wire tmp;

  xor2 xor2_1 // Instanz 1
    (
      .a(a),
      .b(b),
      .e(tmp)
    );
  xor2 xor2_2 // Instanz 2
    (
      .a(c),
      .b(tmp),
      .e(e)
    );

endmodule
```

SEQUENTIELLER CODE

Wie besprochen **übernimmt** ein Flipflop die Eingabe an den **steigenden oder fallenden Flanken** des Taktes.

Dafür wird die **Ereignissteuerung** mit dem **`@`-Symbol** und `always`-Blöcke verwendet:

```
                                    always @ ( posedge clk or
                                               posedge reset)
module FF (input  clk,          begin
           input  rst,            if ( rst )
           input  d,                q <= 1'b0;
           output q);             else
                                    q <= d;
  reg q;                        end

endmodule
```

Die Signalliste hinter dem `@` heißt **Sensitivity-List**. Der `reset` wird **synchron**, wenn man `or posedge reset` **entfernt**.

EINIGE BASISSCHALTKREISE IN VERILOG

KOMBINATORISCHE SCHALTKREISE

KOMBINATORISCHE SCHALTKREISE

Kombinatorische Schaltkreise entsprechen reinen booleschen Funktionen und enthalten demzufolge **nicht** das Schlüsselwort `reg`. Es wird kein Speicher (Flipflops) erzeugt und Zuweisungen geschehen mit `assign`.

```
module mux4to1 (in1, in2, in3, in4, sel, out);

  parameter WIDTH = 8;

  input [WIDTH - 1 : 0] in1, in2, in3, in4;
  input [1:0] sel;
  output [WIDTH - 1 : 0] out;

  assign out = (sel == 2'b00) ? in1 :
               (sel == 2'b01) ? in2 :
               (sel == 2'b10) ? in3 :
               in4;
endmodule
```

PRIORITÄTSENDCODER

Analog zur VHDL-Version formulieren beschreiben wir den Prioritätsencoder wie folgt:

```
module prienc (input  wire [4 : 1] req,
               output wire [2 : 0] idx);

    assign idx = (req[4] == 1'b1) ? 3'b100 :
                 (req[3] == 1'b1) ? 3'b011 :
                 (req[2] == 1'b1) ? 3'b010 :
                 (req[1] == 1'b1) ? 3'b001 :
                 3'b000;

endmodule
```

PRIORITÄTSENDCODER (ALTERNATIVE)

Für einen Prioritätsencoder kann man das `don't care`-**Feature von Verilog** verwenden.

```
module prienc (input  [4:1] req,
               output reg [2:0] idx);

  always @(*) begin
    casez (req) // casez erlaubt don't-care
      4'b1???: idx = 3'b100; // Auch: idx = 4;
      4'b01??: idx = 3'b011;
      4'b001?: idx = 3'b010;
      4'b0001: idx = 3'b001;
      default: idx = 3'b000;
    endcase
  end

endmodule
```

Analog ist der **VHDL-Decoder in Verilog** übertragbar.

SEQUENTIELLE SCHALTKREISE (WIEDERHOLUNG)

SYNCHRONES DESIGN (WIEDERHOLUNG)

Im **Gegenteil zu kombinatorischen Schaltkreisen** verwenden sequentielle Schaltkreise **internen Speicher**, d.h. die Ausgabe hängt **nicht nur** von der Eingabe ab.

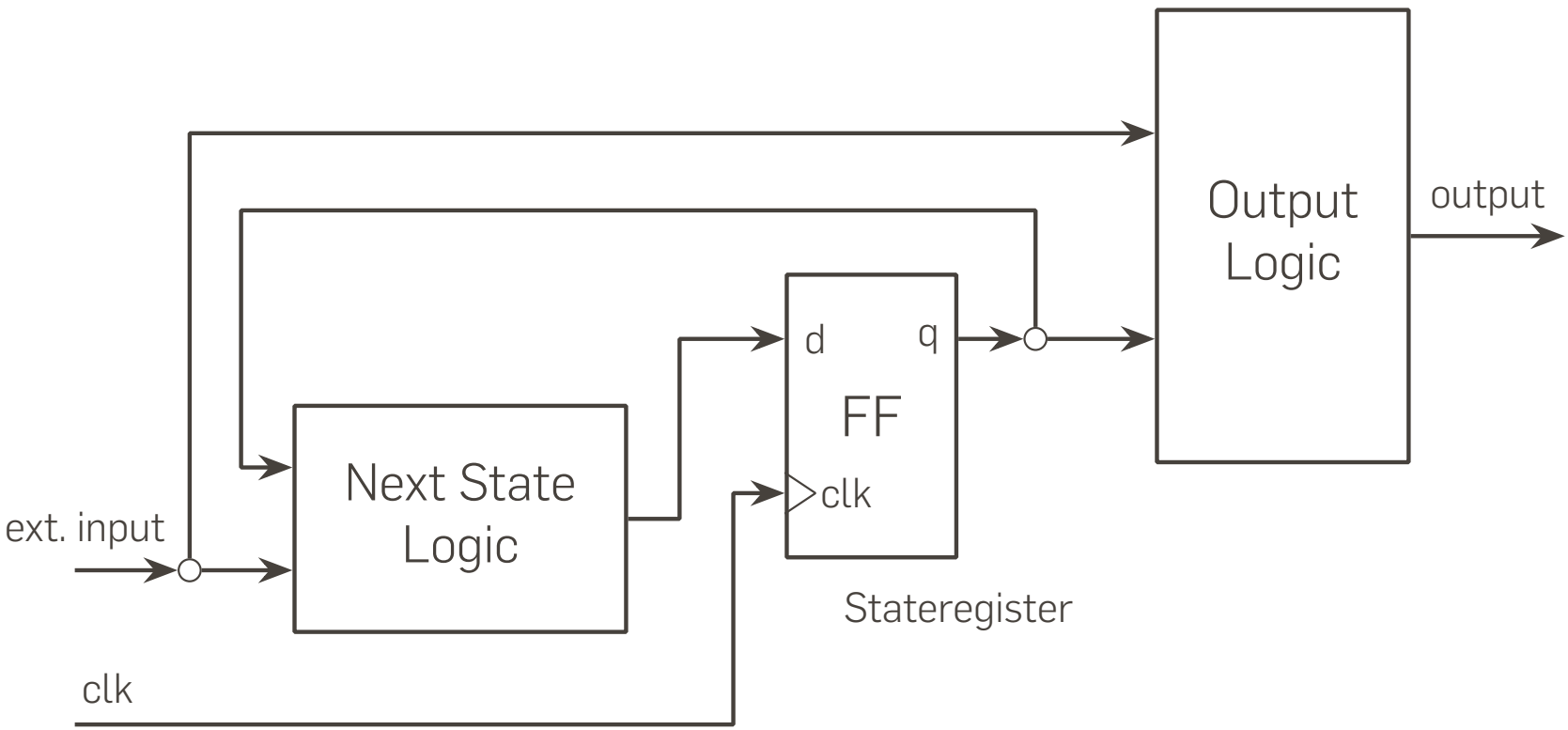
Bei der synchronen Methode werden **alle Speicherelemente** durch einen **globalen Takt kontrolliert** / **synchronisiert**. Alle **Berechnungen** werden an der steigenden (und/oder) fallenden **Flanke des Taktes** vorgenommen.

Das synchrone Design ermöglicht den Entwurf, Test und die Synthese von **großen** Schaltkreisen mit marktüblichen Tools. Aus diesem Grund ist es empfehlenswert dieses Designprinzip zu erinnerlichen!

Weiterhin sollte **keine (kombinatorische) Logik im Taktpfad** sein, da dies zu Problemen mit der Laufzeit der Clocksignale führen kann!

SYNCHRONE SCHALTKREISE (WIEDERHOLUNG)

Die Struktur von synchronen Schaltkreisen ist **idealisiert** wie folgt aufgebaut:

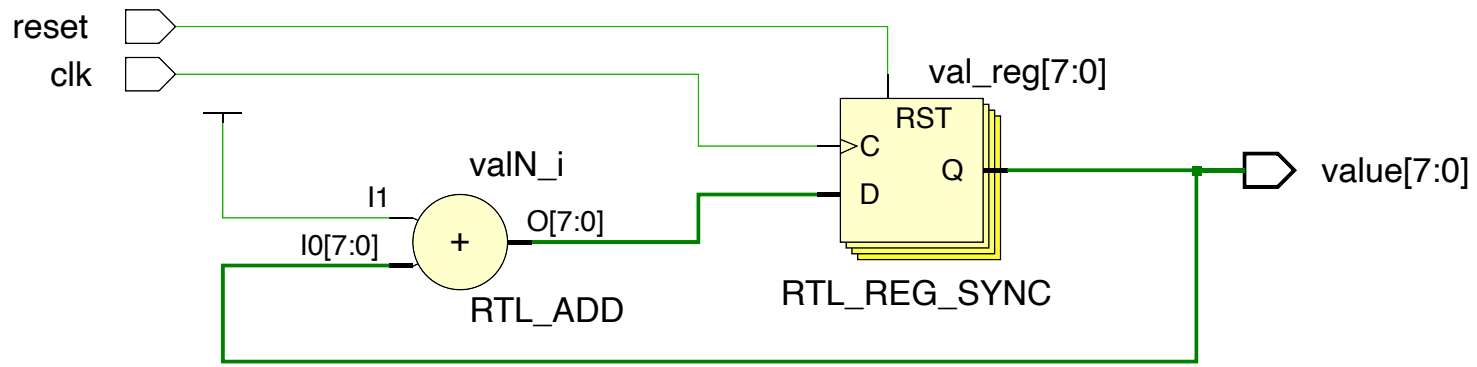


EIN BINÄRZÄHLER

Entsprechend dem synchronen Design kann ein **frei laufender Binärzähler** (free running binary counter) realisiert werden:

```
module freecnt (value, clk, reset);

  parameter WIDTH = 8;

  input  wire clk;
  input  wire reset;
  output wire [WIDTH - 1 : 0] value;

  wire [WIDTH - 1 : 0] valN;
  reg  [WIDTH - 1 : 0] val;

  always @(posedge clk) begin

   if (reset) begin // Synchron reset
     val <= {WIDTH{1'b0}};
   end else begin
     val <= valN;
   end

  end

  assign valN = val + 1; // Nextstate logic
  assign value = val; // Output logic

endmodule
```

DAS RESULTAT EINER SYNTHESE



An dieser Stelle kann man genau sehen, dass das Ergebnis dem Schaubild des synchronen Designs folgt.

`RTL_REG_SYNC` entspricht dem **Stateregister** und `RTL_ADD` entspricht der **Next State Logic**.

EINIGE BEMERKUNGEN

Bisher verwenden wir **drei Zuweisungsoperatoren**:

- → `assign signal0 = value`,
- → `signal2 <= value` und
- → `signal1 = value`

Die `assign`-Anweisung ist als **continuous assignment** bekannt und entspricht (grob) einer **immer aktiven Drahtverbindung**. Sie wird für Signale vom Typ `wire` verwendet und ist für `reg` **(Register) nicht zulässig**.

Der Operator `<=` heißt **non-blocking assignment**. Diese Zuweisung wird für **synthetisierte Register** verwendet, d.h. in `always`-Blöcken mit `posedge clk` in der Sensitivity-Liste.

Die Variante `=` heißt **blocking assignment** und wird für kombinatorische `always`-Blöcke verwendet. Achtung: Für Signale vom Typ `wire` **nicht zulässig**! Also **Typ** `reg` **verwenden**.

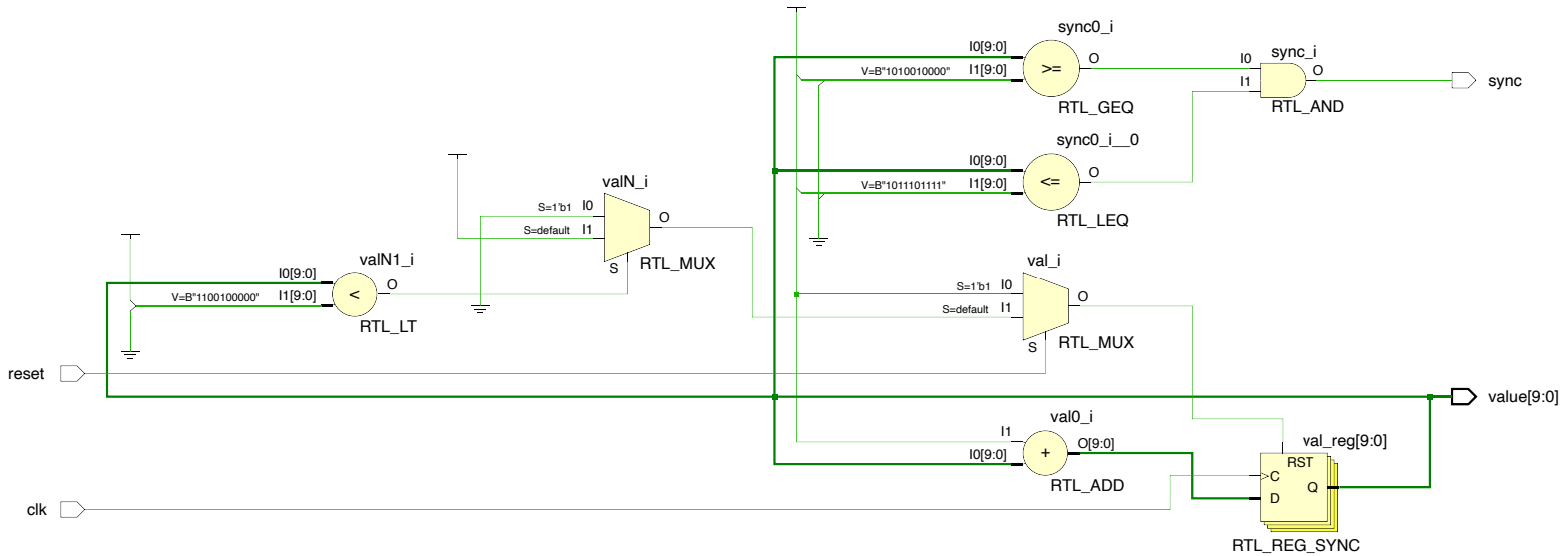
EIN MODULO-ZÄHLER

Entsprechend dem synchronen Design kann ein **frei laufender Binärzähler** (free running binary counter) realisiert werden:

```
module modcnt (value, clk, reset, sync);

  parameter WIDTH  = 10,
            MODULO = 800,
            hsMin  = 656,
            hsMax  = 751;

  input  wire clk;
  input  wire reset;
  output wire [WIDTH - 1 : 0] value;
  output wire sync;

  wire [WIDTH - 1 : 0] valN;
  reg  [WIDTH - 1 : 0] val;

  always @(posedge clk) begin

    if (reset) begin // Synchron reset
      val <= {WIDTH{1'b0}};
    end else begin
      val <= valN;
    end

  end

  // Nextstate logic
  assign valN = (val < MODULO) ? val + 1 : 0;

  // Output logic
  assign value = val;
  assign sync = ((val >= hsMin) && (val <= hsMax))

endmodule
```

DAS RESULTAT EINER SYNTHESE

In diesem Fall sind **Next State Logic** und **Output Logic** natürlich deutlich komplizierter:



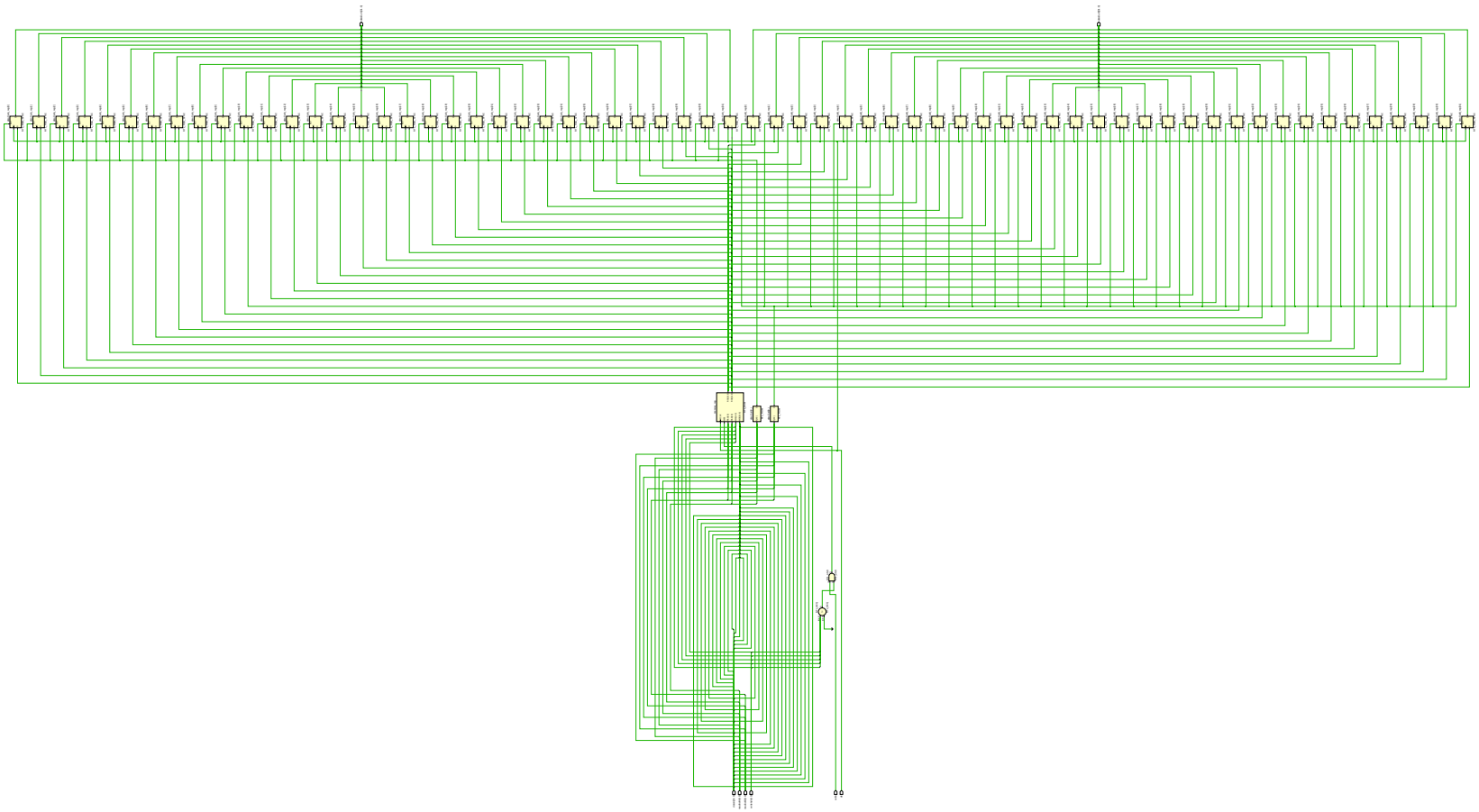
EIN REGISTERFILE

RISC-V Prozessoren besitzen ein **Registerfile** mit einem besonderen **Zero-Register**. Lesen **liefert immer eine 0** und Schreiboperationen werden ignoriert.

```
module regfile (input clk,
                input [4:0] writeAdr, input [31 : 0] dataIn,
                input wrEn,
                input [4:0] readAdrA, output reg [31:0] dataOutA,
                input [4:0] readAdrB, output reg [31:0] dataOutB);

  reg [31 : 0] memory [1 : 31];

  always @(posedge clk) begin

    if ((wrEn) && (writeAdr != 0)) begin

      memory[writeAdr] <= dataIn;

    end

    dataOutA <= (readAdrA == 0) ? 0 : memory[readAdrA];
    dataOutB <= (readAdrB == 0) ? 0 : memory[readAdrB];

  end

endmodule
```

SYNTHESE DES REGISTERFILES

Das Syntheseergebnis ist dann schon etwas unübersichtlicher:



AUSGEWÄHLTE FEATURES VON VERILOG

PARAMETRISIERTE HARDWARE

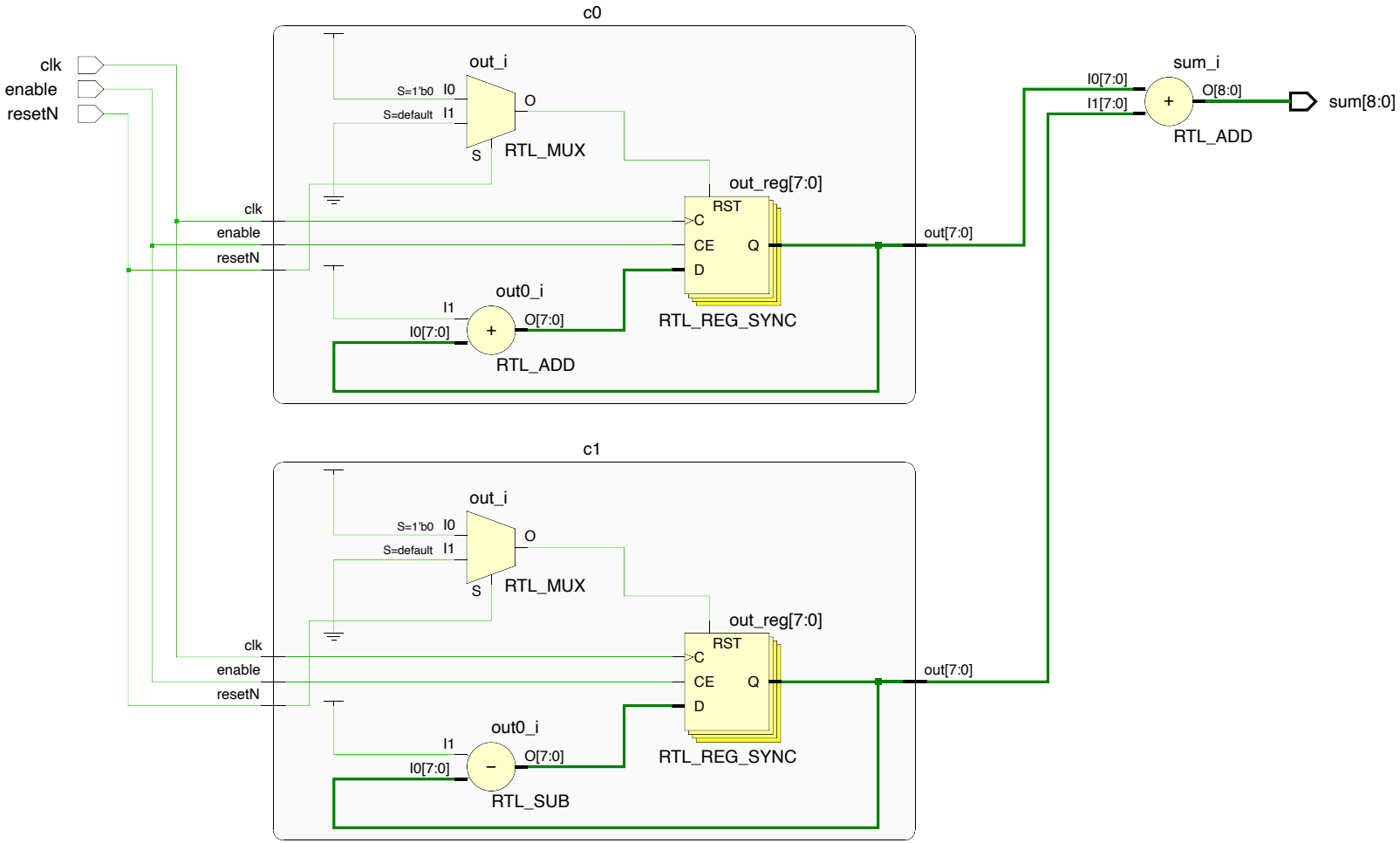
EIN PARAMETRISIERTER COUNTER

Die **neueren Varianten** von Verilog bieten eine verbesserte Version des schon bekannten `Parameter`**-Features**:

```
module cnt
  #(parameter N = 8,
    parameter DOWN = 0)

   (input clk,
    input resetN,
    input enable,
    output reg [N-1:0] out);

    always @ (posedge clk) begin

     if (!resetN) begin // Synchron
       out <= 0;
     end else begin
     if (enable)
       if (DOWN)
        out <= out - 1;
      else
        out <= out + 1;
     else
       out <= out;
     end

    end

endmodule
```

```
module doubleSum
  #(parameter N = 8)
   (input clk,
    input resetN,
    input enable,
    output [N   :   0] sum);

  wire [N - 1   :   0] val0;
  wire [N - 1   :   0] val1;

  // Counter 0
  cnt #(.N(N), .DOWN(0)) c0 (.clk(clk),
                             .resetN(resetN),
                             .enable(enable),
                             .out(val0));

  // Counter 1
  cnt #(.N(N), .DOWN(1)) c1 (.clk(clk),
                             .resetN(resetN),
                             .enable(enable),
                             .out(val1));

  assign sum = val0 + val1;

endmodule
```

SYNTHESE EINES PARAMETRISIERTEN COUNTERS



EINE ALTERNATIVE VORGEHENSWEISE

Verilog bietet weiterhin eine **(ältere) Möglichkeit** für die Parametrisierung eines Designs:

```
module double
  #(parameter N = 8)
  (input clk,
   input resetN,
   input enable,
   output [N : 0] sum);

  wire [N - 1 : 0] val0;
  wire [N - 1 : 0] val1;

  // Counter 0
  defparam c0.N = N;
  defparam c0.DOWN = 0;
  cnt c0 (.clk(clk),
          .resetN(resetN),
          .enable(enable),
          .out(val0));

  // Counter 1
  defparam c1.N = N;
  defparam c1.DOWN = 1;
  cnt c1 (.clk(clk),
          .resetN(resetN),
          .enable(enable),
          .out(val1));

  assign sum = val0 + val1;

endmodule
```

Diese Variante führt zum **gleichen Syntheseergebnis**.

DER PRÄPROZESSOR

BEDINGTE ÜBERSETZUNG

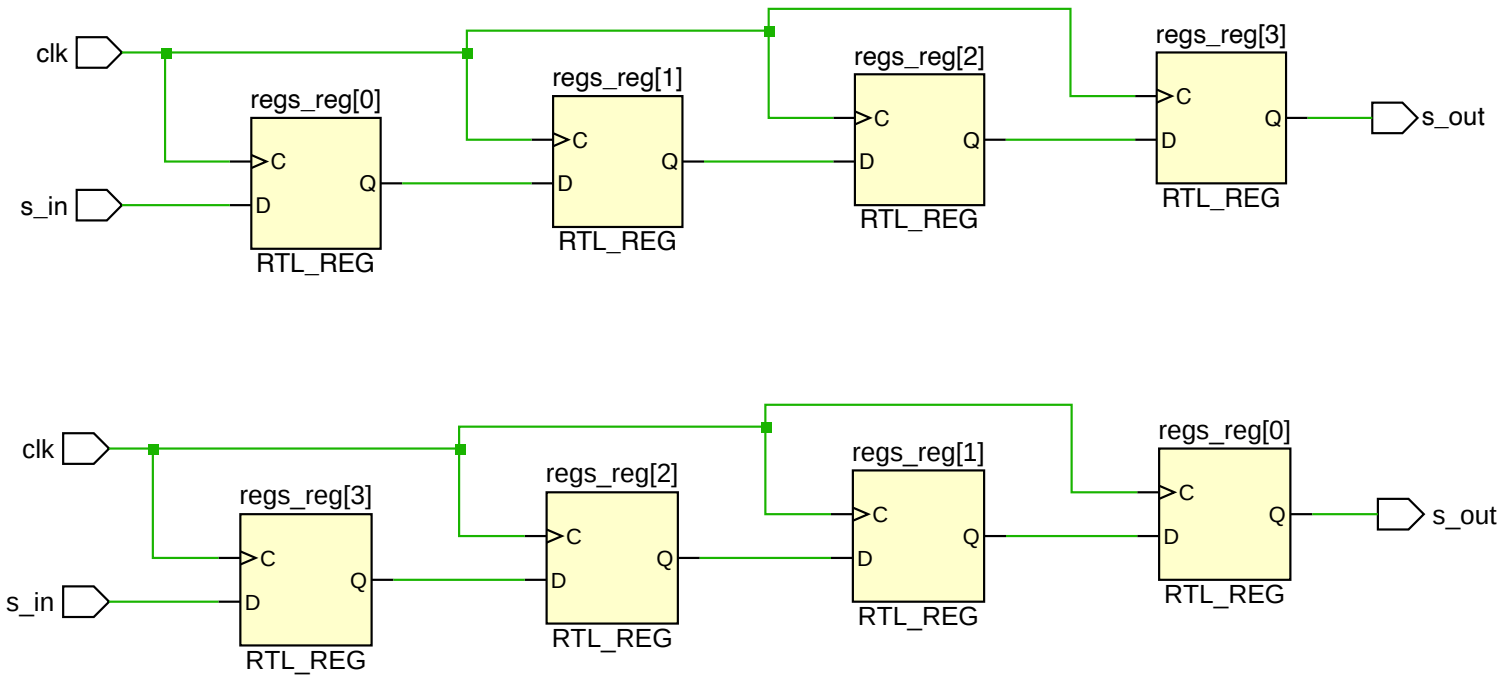
Verilog kennt einen Präprozessor (vgl. C/C++) mit `` `define, `include `` **und** `` `ifdef ``.

Dabei definiert ein `parameter` eine Konstante und `` `define `` eine **Textsubstitution**.

```
‘define SHIFT_RIGHT
module defineDemo (input clk, s_in,
                   output s_out);

  reg [3:0] regs;

  always @(posedge clk) begin // Next State Logic im always - Block
    ‘ifdef SHIFT_RIGHT
      regs <= {s_in, regs[3:1]};
    ‘else
      regs <= {regs[2:0], s_in};
    ‘endif
  end

  ‘ifdef SHIFT_RIGHT
    assign s_out = regs[0];
  ‘else
    assign s_out = regs[3];
  ‘endif

endmodule
```

ZWEI SYNTHESEERGEBNISSE

Durch die bedingte Synthese erhält man **zwei unterschiedliche Schieberegister**:



MODULARISIERUNG

Vergleichbar mit dem Include-Mechanismus von C/C++ bietet Verilog die Möglichkeit einer **primitiven Modularisierung** mit `` `include ``.

Dabei ist das **Tick-Symbol** `` ` `` wieder der Marker für einen Präprozessor-Befehl, vergleichbar mit # bei C/C++.

Mit `` `include headers_def.h `` können z.B. Konfigurationseinstellungen aus der Datei `headers_def.h` **inkludiert** werden. Da ein **reiner Textersatz** durchgeführt wird, ist die **Dateiendung** im Prinzip **beliebig**. Sinnvollerweise verwendet man `.h` analog zu C.

Sollte ein `` `define `` vor einem `` `include `` angeordnet sein, so wird der Textersatz auch im inkludierten Headerfile durchgeführt, d.h. ein `` `define `` **gilt global** ab der Definition. Damit können aber vergleichbar mit C **unbeabsichtige Ersetzungen** passieren.

YOSYS UND SYSTEMVERILOG

EINIGE ERWEITERUNGEN

Das Open-Source Synthesetool **yosys** stellt einige **ausgewählte Erweiterungen** aus **SystemVerilog** zur Verfügung.

- → Besonders interessant ist der `logic`**-Datentyp**, der **Zuweisungen** und `reg` und `wire` **deutlich vereinfacht**. Mit `logic signed` deklariert man **vorzeichenbehaftete Zahlen**.
- → Für **sequentielle Logik** wurde der spezielle Block `always_ff` eingeführt. Für Zuweisungen werden ausschließlich **non-blocking Assignments (<=)** verwendet.
- → Für **kombinatorische Logik** ersetzt `always_comb` das Konstrukt `always @(*)`. In `always_comb`-Blöcken werden nur **blocking Assignments (=)** verwendet.

ERWEITERUNGEN - EIN BEISPIEL

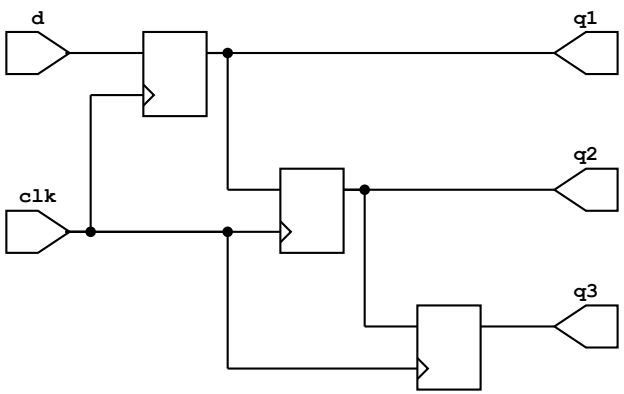
Nun soll der free-running counter neu implementiert werden:

```
module freecnt2

  #(parameter WIDTH = 8)

  (input  logic clk,
   input  logic reset,
   output logic [WIDTH - 1 : 0] value);

  logic [WIDTH - 1 : 0] valN;
  logic [WIDTH - 1 : 0] val;

  always_ff @(posedge clk) begin

    if (reset) begin // Synchron reset
      val <= {WIDTH{1'b0}};
    end else begin
      val <= valN;
    end

  end

  always_comb begin

    valN  = val + 1; // Nextstate logic
    value = val; // Output logic

  end

endmodule
```

BLOCKING UND NON-BLOCKING ASSIGNMENTS IN ALWAYS_FF

Vorsicht mit **falschen Zuweisungen** in `always_ff`:

```
module demoOk (input clk,
               input d,
               output q1,
               output q2,
               output q3);

  always_ff @(posedge clk) begin
    q1 <= d;
    q2 <= q1;
    q3 <= q2;
  end

endmodule
```




```
module demoWrong (input clk,
                  input d,
                  output q1,
                  output q2,
                  output q3);

  always_ff @(posedge clk) begin
    q1 = d;
    q2 = q1;
    q3 = q2;
  end

endmodule
```