Verilog crash course

Steffen Reith, Thorsten Knoll

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

1. Introduction
2. Verilog elements
3. Simple circuits: Combinational
4. Simple circuits: Sequential
5. Selected feature: Parameterized counter
6. Selected feature: Preprocessor
7. Selected feature: Yosys and Systemverilog

Section 1

Introduction

Introduction

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)

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] https://theopenroadproject.org/

Literature

- Donald E. Thomas, Philip R. Moorby, The Verilog Hardware Description Language, Kluwer Academic Publishers, 2002, ISBN 978-1475775891
- Blaine Readler, Verilog by example, Full Arc Press, 2011, ISBN 978-0983497301

Contributions, mentions and license

- This course is a translated, modified and ‘markdownized’ version of a Verilog crash course from Steffen Reith, original in german language.

  https://github.com/SteffenReith

- The initial rework (translate, modify and markdownize) was done by:

  https://github.com/ThorKn

- The build of the PDF slides is done with pandoc:

  https://pandoc.org/

- Pandoc is wrapped within this project:

  https://github.com/alexeygumirov/pandoc-beamer-how-to

- License:

  GPLv3

Synthesis tool: Yosys

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

> Yosys is a framework for VerilogRTLsynthesis. It currently has extensive Verilog-2005 support and provides a ba- sic set of synthesis algorithms for various application do mains. Selected features and typical applications:

- Process almost any synthesizable Verilog-2005 design
- Converting Verilog to BLIF / EDIF/ BTOR / SMT-LIB /simple RTL Verilog / etc.
- …

Section 2

Verilog elements

Structure of a verilog module

```
1  module module_name (port_list);
2  // Definition der Schnittstelle
3  Port-Deklaration
4  Parameter-Deklaration
5
6  // Beschreibung des Schaltkreises
7  Variablen-Deklaration
8  Zuweisungen
9  Modul-Instanzierungen
10
11 always-Bloecke
12
13 endmodule
```

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

Example: A linear shiftregister

```
1  module LSFR (
2
3  input  wire   load,
4  input  wire   loadIt,
5  input  wire   enable,
6  output wire   newBit,
7  input  wire   clk,
8  input  wire   reset);
9
10 wire       [17:0]   fsRegN;
11 reg        [17:0]   fsReg;
12 wire                taps_0, taps_1;
13 reg                 genBit;
14
15 assign taps_0 = fsReg[0];
16 assign taps_1 = fsReg[11];
17
18 always @(*) begin
19   genBit = (taps_0 ^ taps_1);
20   if(loadIt) begin
21     genBit = load;
22   end
23 end
```

```
1  assign newBit = fsReg[0];
2  assign fsRegN = {genBit,fsReg[17 : 1]};
3
4  always @(posedge clk) begin
5    if(reset) begin
6      fsReg <= 18'h0;
7    end else begin
8      if(enable) begin
9        fsReg <= fsRegN;
10     end
11   end
12 end
13
14 endmodule
```

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

Constants and Operators

Es gibt vier Werte für eine Konstante / Signal:

- 0 oder 1
- X bzw. x (unbekannt)
- 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)
- d (dezimal) zulässig.

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

Passend zu den Konstanten existiert eine Array-Schreibweise:

- wire [7:0] serDat;
- reg [0:32] shiftReg;
- Einzelne Bits können gesliced werden:
  - serDat[3 : 0] (low-nibble)
  - 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)

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:

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

```
1 if (4'b110z == 4'b110z)
2 // not taken
3 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.

Parameters (old style)

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

```
1  module mux (
2    in1, in2,
3    sel,
4    out);
5
6    parameter WIDTH = 8;  // Anzahl der Bits
7
8    input  [WIDTH - 1 : 0] in1, in2;
9    input sel;
10   output [WIDTH - 1 : 0] out;
11
12   assign out = sel ? in1 : in2;
13
14 endmodule
```

Instances and structural descriptions

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

```
1 module xor2 (
2   input  wire a,
3   input  wire b,
4   output wire e);
5   assign e = a ^ b;
6 endmodule
```

```
1  module xor3 (
2    input  wire a,
3    input  wire b,
4    input  wire c,
5    output wire e);
6
7    wire tmp;
8    xor2 xor2_1 // Instanz 1
9      (
10       .a(a),
11       .b(b),
12       .e(tmp)
13     );
14   xor2 xor2_2 // Instanz 2
15     (
16       .a(c),
17       .b(tmp),
18       .e(e)
19     );
20
21 endmodule
```

Code for sequential circuits

Wie besprochen übernimmt ein Flipflop die Eingabe an den steigenden oder fallenden Flanken des Taktes. Dafür wird die Ereignissteuerung mit dem @-Symbol und always-Blöcken verwendet:

```
1  module FF (input  clk,
2             input  rst,
3             input  d,
4             output q);
5    reg q;
6
7    always @ ( posedge clk or
8               posedge reset)
9      begin
10       if ( rst )
11         q <= 1'b0;
12       else
13         q <= d;
14     end
15 endmodule
```

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

Section 3

Simple circuits: Combinational

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

```
1  module mux4to1 (in1, in2, in3, in4, sel, out);
2
3    parameter WIDTH = 8;
4
5    input [WIDTH - 1 : 0] in1, in2, in3, in4;
6    input [1:0] sel;
7    output [WIDTH - 1 : 0] out;
8
9    assign out = (sel == 2'b00) ? in1 :
10                (sel == 2'b01) ? in2 :
11                (sel == 2'b10) ? in3 :
12                in4;
13 endmodule
```

Priority encoder

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

```
1  module prienc (input  wire [4 : 1] req,
2                 output wire [2 : 0] idx);
3
4     assign idx = (req[4] == 1'b1) ? 3'b100 :
5                  (req[3] == 1'b1) ? 3'b011 :
6                  (req[2] == 1'b1) ? 3'b010 :
7                  (req[1] == 1'b1) ? 3'b001 :
8                  3'b000;
9
10 endmodule
```

Priority encoder (alternative version)

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

```
1  module prienc (input  [4:1] req,
2                 output reg [2:0] idx);
3
4    always @(*) begin
5      casez (req) // casez erlaubt don't-care
6          4'b1???: idx = 3'b100; // Auch: idx = 4;
7          4'b01??: idx = 3'b011;
8          4'b001?: idx = 3'b010;
9          4'b0001: idx = 3'b001;
10         default: idx = 3'b000;
11     endcase
12   end
13
14 endmodule
```

Section 4

Simple circuits: Sequential

Synchronous design

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!

Synchronous circuits

Die Struktur von synchronen Schaltkreisen ist idealisiert wie folgt aufgebaut:

TODO: Picture here

A binary counter

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

```
1  module freecnt (value, clk, reset);
2
3    parameter WIDTH = 8;
4
5    input  wire clk;
6    input  wire reset;
7    output wire [WIDTH - 1 : 0] value;
8
9    wire [WIDTH - 1 : 0] valN;
10   reg  [WIDTH - 1 : 0] val;
11
12   always @(posedge clk) begin
13
14    if (reset) begin // Synchron reset
15      val <= {WIDTH{1'b0}};
16    end else begin
17      val <= valN;
18    end
19
20   end
21
22   assign valN = val + 1; // Nextstate logic
23   assign value = val; // Output logic
24 endmodule
```

Synthesis result of the binary counter

TODO: Picture here

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.

Some remarks

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

A modulo counter

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

```
1  module modcnt (value, clk, reset, sync);
2
3    parameter WIDTH  = 10,
4              MODULO = 800,
5              hsMin  = 656,
6              hsMax  = 751;
7
8    input  wire clk;
9    input  wire reset;
10   output wire [WIDTH - 1 : 0] value;
11   output wire sync;
12
13   wire [WIDTH - 1 : 0] valN;
14   reg  [WIDTH - 1 : 0] val;
```

```
1    always @(posedge clk) begin
2
3      if (reset) begin // Synchron reset
4        val <= {WIDTH{1'b0}};
5      end else begin
6        val <= valN;
7      end
8
9    end
10
11   // Nextstate logic
12   assign valN = (val < MODULO) ? val + 1 : 0;
13
14   // Output logic
15   assign value = val;
16   assign sync = ((val >= hsMin) && (val <= hsMax)) ? 1 : 0;
17
18 endmodule
```

Synthesis result of the modulo counter

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

TODO: Picture here

A register file

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

```
1  module regfile (input clk,
2                  input [4:0] writeAdr, input [31 : 0] dataIn,
3                  input wrEn,
4                  input [4:0] readAdrA, output reg [31:0] dataOutA,
5                  input [4:0] readAdrB, output reg [31:0] dataOutB);
6
7    reg [31 : 0] memory [1 : 31];
8
9    always @(posedge clk) begin
10
11     if ((wrEn) && (writeAdr != 0)) begin
12
13       memory[writeAdr] <= dataIn;
14
15     end
16
17     dataOutA <= (readAdrA == 0) ? 0 : memory[readAdrA];
18     dataOutB <= (readAdrB == 0) ? 0 : memory[readAdrB];
19
20   end
21
22 endmodule
```

Synthesis result of the register file

Das Syntheseergebnis ist dann schon etwas unübersichtlicher:

TODO: Picture here

Section 5

Selected feature: Parameterized counter

Selected feature: Parameterized counter

Die neueren Varianten von Verilog bieten eine verbesserte Version des Parameter-Features:

```
1  module cnt
2    #(parameter N = 8,
3      parameter DOWN = 0)
4
5    (input clk,
6     input resetN,
7     input enable,
8     output reg [N-1:0] out);
9
10    always @ (posedge clk) begin
11
12     if (!resetN) begin // Synchron
13       out <= 0;
14     end else begin
15     if (enable)
16       if (DOWN)
17         out <= out - 1;
18       else
19         out <= out + 1;
20     else
21       out <= out;
22     end
23
24    end
25
26 endmodule
```

```
1  module doubleSum
2    #(parameter N = 8)
3     (input clk,
4      input resetN,
5      input enable,
6      output [N  :  0] sum);
7
8    wire [N - 1  :  0] val0;
9    wire [N - 1  :  0] val1;
10
11   // Counter 0
12   cnt #(.N(N), .DOWN(0)) c0 (.clk(clk),
13                              .resetN(resetN),
14                              .enable(enable),
15                              .out(val0));
16
17   // Counter 1
18   cnt #(.N(N), .DOWN(1)) c1 (.clk(clk),
19                              .resetN(resetN),
20                              .enable(enable),
21                              .out(val1));
22
23   assign sum = val0 + val1;
24
25 endmodule
```

Synthesis result of the parameterized counter

TODO: Picture here

An alternative version

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

```
1  module double
2    #(parameter N = 8)
3    (input clk,
4     input resetN,
5     input enable,
6     output [N : 0] sum);
7
8    wire [N - 1 : 0] val0;
9    wire [N - 1 : 0] val1;
10
11   // Counter 0
12   defparam c0.N = N;
13   defparam c0.DOWN = 0;
14   cnt c0 (.clk(clk),
15           .resetN(resetN),
16           .enable(enable),
17           .out(val0));
```

```
1    // Counter 1
2    defparam c1.N = N;
3    defparam c1.DOWN = 1;
4    cnt c1 (.clk(clk),
5            .resetN(resetN),
6            .enable(enable),
7            .out(val1));
8
9    assign sum = val0 + val1;
10
11 endmodule
```

Diese Variante führt zum gleichen Syntheseergebnis.

Section 6

Selected feature: Preprocessor

Selected feature: Preprocessor

Verilog kennt einen Präprozessor (vgl. C/C++) mit ’define, ’include und ’ifdef. Dabei definiert ein *parameter* eine Konstante und ‘define eine Textsubstitution.

```
1  `define SHIFT_RIGHT
2  module defineDemo (input clk, s_in,
3                     output s_out);
4
5    reg [3:0] regs;
6
7    always @(posedge clk) begin // Next State Logic im always - Block
8      `ifdef SHIFT_RIGHT
9        regs <= {s_in, regs[3:1]};
10     `else
11       regs <= {regs[2:0], s_in};
12     `endif
13   end
14
15   `ifdef SHIFT_RIGHT
16     assign s_out = regs[0];
17   `else
18     assign s_out = regs[3];
19   `endif
20
21 endmodule
```

Two results of the synthesis

Durch die bedingte Synthese erhält man zwei unterschiedliche Schieberegister:

TODO: Picture here

Modularisation

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 unbeabsichtigte Ersetzungen passieren.

Section 7

Selected feature: Yosys and Systemverilog

Selected feature: Yosys and Systemverilog

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.

Another counter

Nun soll der free-running counter neu implementiert werden:

```
1  module freecnt2
2
3    #(parameter WIDTH = 8)
4    (input  logic clk,
5     input  logic reset,
6     output logic [WIDTH - 1 : 0] value);
7
8    logic [WIDTH - 1 : 0] valN;
9    logic [WIDTH - 1 : 0] val;
10
11   always_ff @(posedge clk) begin
12
13     if (reset) begin // Synchron reset
14       val <= {WIDTH{1'b0}};
15     end else begin
16       val <= valN;
17     end
18
19   end
20
21   always_comb begin
22
23     valN  = val + 1; // Nextstate logic
24     value = val; // Output logic
25
26   end
27 endmodule
```

Blocking and Non-blocking assignments in always_ff

Vorsicht mit falschen Zuweisungen in always_ff:

```
1  module demoOk (input clk,
2                 input d,
3                 output q1,
4                 output q2,
5                 output q3);
6
7    always_ff @(posedge clk) begin
8      q1 <= d;
9      q2 <= q1;
10     q3 <= q2;
11   end
12
13 endmodule
```

TODO: picture

```
1  module demoWrong (input clk,
2                    input d,
3                    output q1,
4                    output q2,
5                    output q3);
6
7    always_ff @(posedge clk) begin
8      q1 = d;
9      q2 = q1;
10     q3 = q2;
11   end
12
13 endmodule
```

TODO: picture