

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 was initially developed as a simulation language in 1983/1984, bought up by Cadence and freely released in 1990.

The first standardization took place in 1995 by the IEEE (Verilog 95). A newer version is IEEE Standard 1364-2001 (Verilog 2001).

- Syntax comparable to C (VHDL was started on ADA / Pascal) with compact code
- Spread in North America and Japan (less in Europe)
- Can also be used as the language for netlists
- Support from open source tools
- The majority of the ASICs are developed in Verilog.
- Less expressive than VHDL (curse and blessing)

The proximity to C and Java may lead to confusion. In Verilog, too, lines that describe a combinatorial circuit can also be replaced.

**** Verilog is a hardware description language (HDL)****

This crash course is limited to a subset of synthesizable language constructs in Verilog.

The aim of this selection is not commercial tools, but open-source development tools such as OpenRoad¹ or Toolchains for FPGAs, i.e. we also use some language constructs from Systemverilog, which are supported by the Yosys synthesis tool.

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

One should also deal with the peculiarities of the synthesis tool. The well-known open source synthesis tool Yosys writes about this

Yosys is a framework for VerilogRTLsynthesis. 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.
- ...

Section 2

Verilog elements

Structure of a verilog module

```
1  module module_name (port_list);  
2  // Definition of the interface  
3  Port declaration  
4  Parameter declaration  
5  
6  // Description of the circuit  
7  Variables declaration  
8  Assignmente  
9  Module instantiations  
10  
11 always-blocks  
12  
13 endmodule
```

Port list and port declaration can be brought together in modern verilog.

// introduces a comment.

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

```

input and **output** define the directions of the ports.

Constants and Operators

There are four values for a constant / signal:

- 0 or 1
- X or x (unknown)
- Z or Z (high impedance)

One can specify the width of constants:

- Hexadecimal constant with 32 bit: `32'hDEADBEEF`
- Binary constant with 4 bit: `4'b1011`
- For better readability you can also use underscores: `12'B1010_1111_0001`

To specify the number base use

- b (binary)
- h (hexadecimal),
- o (octal)
- d (decimal)

The default is decimal (d) and the bit width is optional, i.e. `4711` is a permissible (decimal) constant.

There is an array notation:

- `wire [7:0] serDat;`
- `reg [0:32] shiftReg;`
- Arrays can be sliced to Bits:
 - `serDat[3 : 0]` (low-nibble)
 - `serDat[7]` (MSB).
- `{serDat[7:6], serDat[1:0]}` notes the concatenation.
- Bits can be replicated and converted into an array, i.e `{8{serData[7 : 4]}}` contains eight copies of the high-nibble from `serDat` and has a width of 32.

Arithmetic operations, relations, equivalences and 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)`

**** Attention: **** If x or z do occur, the simulator determines false in a comparison. If you want to avoid this, the operators `===` and `!==` exist. So the following applies:

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

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

Boolean operations exist as usual:

bitwise operators: `&` (AND), `|` (OR), `~` (NOT), `^` (XOR) und auch `~^` (XNOR)

logic operators: `&&` (AND), `||` (OR) und `!` (NOT)

Shiftoperations: `a « b` (shift a for b positions to the left) und `a » b` (shift a for b positions to the right). A negative number b is not permitted, empty spots are filled with 0.

Parameters (old style)

In order to be able to adapt designs easier, Verilog offers the use of parameters.

```
1  module mux (  
2      in1, in2,  
3      sel,  
4      out);  
5  
6      parameter WIDTH = 8;  // Number of 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

If you describe a circuit through its (internal) structure or if a partial circuit is to be reused, an instance is generated and wired.

```
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 // Instance 1  
9     (  
10        .a(a),  
11        .b(b),  
12        .e(tmp)  
13    );  
14     xor2 xor2_2 // Instance 2  
15     (  
16        .a(c),  
17        .b(tmp),  
18        .e(e)  
19    );  
20  
21 endmodule
```


Code for sequential circuits

A flip-flop takes over the input of the rising or falling edges of the clock. For this, the block entry is used with the *@-symbol* and *always* blocks:

```
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
```

The list of signals after the *@-symbol* means sensitivity list. The reset is synchronized when you remove or *posedge reset*.

Section 3

Simple circuits: Combinational

Combinational circuits correspond to pure boolean functions and therefore do not contain the key word *reg*. No memory (flip-flops) gets generated and assignments are done with *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 erinnern!

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     end
24 endmodule
25
26
```

```

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     end
19
20     always_comb begin
21
22         valN = val + 1; // Nextstate logic
23         value = val; // Output logic
24
25     end
26 endmodule
27

```

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