Chapter 3 - Verilog crash course

Course authors (Git file)

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 synthesible language constructs in Verilog.

The aim of this selection is not commercial tools, but open-source development tools such as OpenRoad [1] 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 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 of the interface
3  Port declaration
4  Parameter declaration
5
6  // Description of the circuit
7  Variables declaration
8  Assignmente
9  Module instanciations
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 feedback shiftregister

```
1  module LFSR (
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
```

```
24 assign newBit = fsReg[0];
25 assign fsRegN = {genBit,fsReg[17 : 1]};
26
27 always @(posedge clk) begin
28   if(reset) begin
29     fsReg <= 18'h0;
30   end else begin
31     if(enable) begin
32       fsReg <= fsRegN;
33     end
34   end
35 end
36
37 endmodule
```

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

This part of the code

```
18 always @(*) begin
19   genBit = (taps_0 ^ taps_1);
20   if(loadIt) begin
21     genBit = load;
22   end
23 end
```

becomes this combinational circuit:

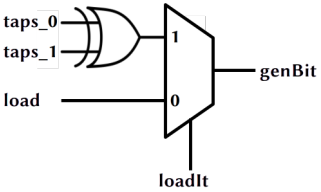


Figure 1: combinational part

And this part of the code

```
18 always @(posedge clk) begin
19   if(reset) begin
20     fsReg <= 18'h0;
21   end else begin
22     if(enable) begin
23       fsReg <= fsRegN;
24     end
25   end
26 end
```

becomes a sequential with memory (flip flops) in it. The flip flops will look something like this:

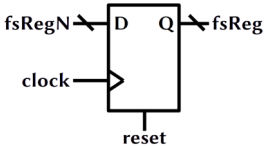


Figure 2: sequential part

Constants and Operators

There are four values available for constants and signals:

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

One can specify the width of constants:

- Hexadepimal 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 valid (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) and ~^ (XNOR)

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

Shiftoperations:

- `<<` Shift Left, Logical (fill with zero)
- `>>` Shift Right, Logical (fill with zero)
- `<<<` Shift Left, Arithmetic (keep sign)
- `>>>` Shift Right, Arithmetic (keep sign)

Syntax: `a<<b`, `a>>b`, `a<<<b`, `a>>>b`.

A negative number b is not permitted.

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

Similarly to the VHDL version, we describe the priority encoder as follows:

```
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)

For a priority encoder you can use the *don't care* feature from Verilog.

```
1  module prienc (input  [4:1] req,
2                 output reg [2:0] idx);
3
4    always @(*) begin
5      casez (req) // casez allows don't-care
6          4'b1???: idx = 3'b100; // Also: 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

Contrary to combinational circuits, sequential circuits use internal memory, i.e. the output not only depends on the input.

In the synchronous method, all memory elements are checked / synchronized by a global clock. All calculations are carried out on the rising (and/or) falling edge of the clock.

The synchronous design enables the draft, test and the synthesis of large circuits with market tools. For this reason, it is advisable to remember this design principle.

Furthermore, there should be no (combinational) logic in the clock path, as this can lead to problems with the distribution times of the clock signals.

Synchronous circuits

The structure of synchronous circuits is idealized as follows:

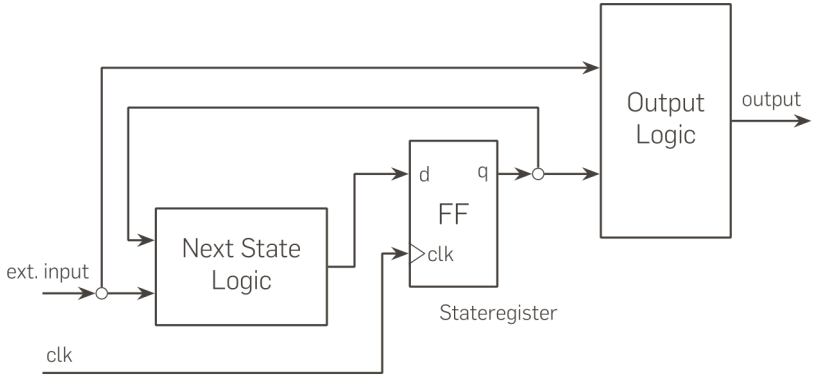


Figure 3: Idealized diagram of a synchronous circuit

A binary counter

According to the synchronous design, a free running binary counter can be realized:

```
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

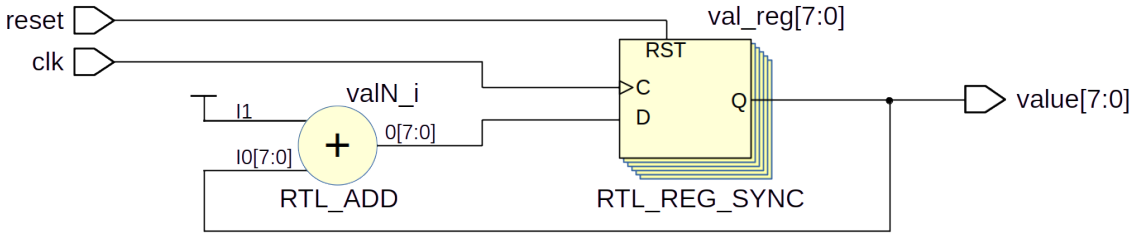


Figure 4: Synthesis diagram of the binary counter

At this point you can see that the result follows the diagram of the synchronous design.

RTL_REG_SYNC corresponds to the stateregister and *RTL_ADD* corresponds to the next state logic.

Some remarks

So far we use three assignment operators:

- assign signal0 = value
- signal2 <= value
- signal1 = value

The *assign* instructions is known as the continuous assignment and corresponds (roughly) to an ever active wire connection. It is used for signals of the type *wire* and is not permitted for *reg* (register).

The operator <= means non-blocking assignment. This assignment is used for synthesized registers, i.e. in *always*-blocks with *posedge clk* in the sensitivity list.

The variant = is called blocking assignment and is used for combinational *always*-blocks. Attention: Not allowed for signals of the type *wire*. So use the type *reg*.

A modulo counter

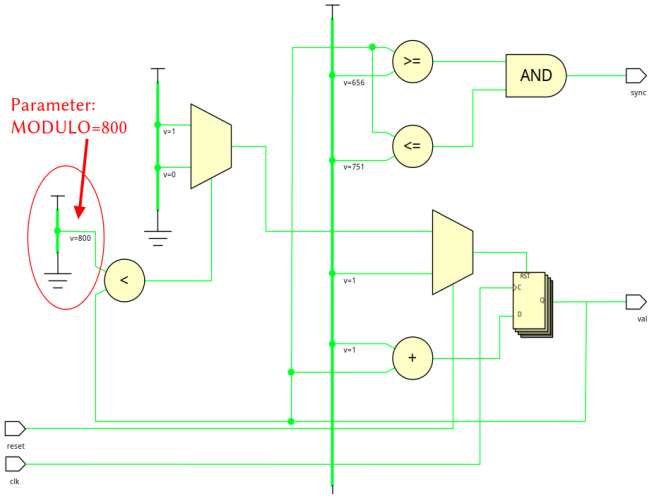
According to the synchronous design, a freely running modulo counter can be realized:

```
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;
```

```
15   always @(posedge clk) begin
16
17     if (reset) begin // Synchron reset
18       val <= {WIDTH{1'b0}};
19     end else begin
20       val <= valN;
21     end
22
23   end
24
25   // Nextstate logic
26   assign valN = (val < MODULO) ? val + 1 : 0;
27
28   // Output logic
29   assign value = val;
30   assign sync = ((val >= hsMin) && (val <= hsMax)) ? 1 : 0;
31
32 endmodule
```

Synthesis result of the modulo counter

In this case, next state logic and output logic are of course more complex:



A register file

RISC-V processors have a register file with a special zero register. Reading always provides 0 and writing operations are ignored.

```
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
```

Section 5

Selected feature: Parameterized counter

Selected feature: Parameterized counter

The newer variants of Verilog offer an improved version of the parameter feature:

```
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
```

Synthesis result of the parameterized counter

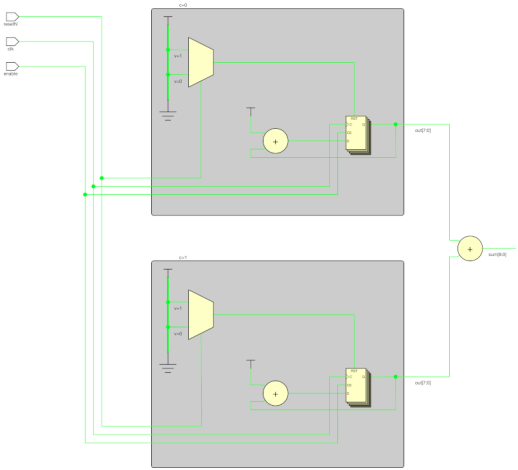


Figure 6: Parameterized counter

An alternative version

Verilog still offers a (older) possibility for the parameterization of a design:

```
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));
```

```
18   // Counter 1
19   defparam c1.N = N;
20   defparam c1.DOWN = 1;
21   cnt c1 (.clk(clk),
22           .resetN(resetN),
23           .enable(enable),
24           .out(val1));
25
26   assign sum = val0 + val1;
27
28 endmodule
```

This variant leads to the same synthesis result.

Section 6

Selected feature: Preprocessor

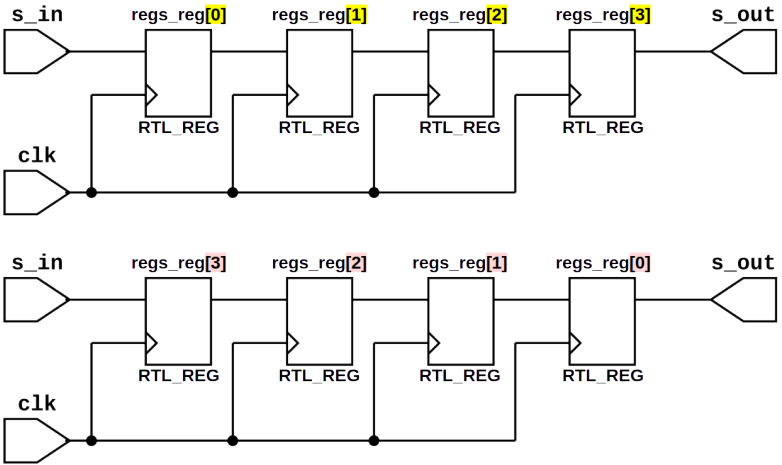
Selected feature: Preprocessor

Verilog knows a preprocessor (cf. C/C ++) with ‘define, ‘include and ‘ifdef. A *parameter* defines a constant and ‘define a text substitution.

```
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 in 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

The conditional synthesis gives you two different shift registers:



Modularisation

Comparable to the include mechanism of C/C ++, Verilog offers the possibility of primitive modularization with ‘include.

The tick symbol ‘ is again the marker for a preprocessor command, comparable to # at C/C++.

With ‘include headers_def.h, for example, configuration settings from the file headers_def.h can be included. Since a pure text replacement is carried out, the file extension is basically arbitrary. It is meaningfully to use .h analogous to C.

If a ‘define is arranged in front of a ‘include, the text replacement is also carried out in the included header file, i.e. a ‘define applies globally from the definition on. However, this can happen comparable to C unintentionally.

Section 7

Selected feature: Yosys and Systemverilog

Selected feature: Yosys and Systemverilog

The open source synthesetool Yosys provides some selected extensions from SystemVerilog.

- The logic datatype is particularly interesting, which simplifies allocations with *reg* and *wire*. With *logic signed* you declare signed numbers.
- The special block *always_ff* was introduced for sequential logic. Only non-blocking assignments (<=) are used for assignments.
- For combinatorial logic, *always_comb* replaces the construct *always @()*. Only blocking assignments (=) are used in *always_comb* blocks.

Another counter

Now the free running counter is to be re-implemented:

```
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
```

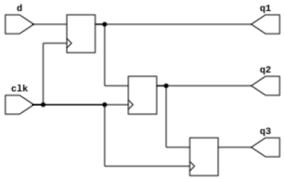
Blocking and Non-blocking assignments in always_ff

Caution with false assignments in *always_ff*:

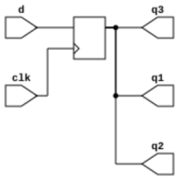
```
1  module demoOk (input clk,
2                 input d,
3                 output q1,
4                 output q2,
5                 output q3);
6    always_ff @(posedge clk) begin
7      q1 <= d;
8      q2 <= q1;
9      q3 <= q2;
10   end
11 endmodule
```

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

Okay:



Wrong:



Section 8

Simulation and Verification

Introduction

What can be done and what not (in open-source EDA):

- A complete simulation and verification of arbitrary complex digital designs at GDS level is not available at the moment.
- Spice and Xyce can do electrical simulations of small digital circuits, like for single standard cells. This topic resides way more in the analog design domain and is therefore not part of this course. A good starting point are the Siliwiz lessons:
  - https://app.siliwiz.com/
- Verification at RTL level is an ongoing (WiP) research effort. SymbiYosys offers formal verification. Other tools might be available soon. For verification with SymbiYosys a good starting point is this tutorial:
  - https://github.com/SymbioticEDA/getting-started-FV
- Simulation at RTL level (Verilog) is available with some tools.
- The following slides provide an overview and a small example.

Simulating Verilog

Why simulate a microchip design?

- Find errors in the design before the production of microchips.
- Production of microchips takes long and is expensive.
- Microchips can't be updated, like software.
- Many more reasons… (Name some yourself, round-robin in the room).

What is simulating a design?

- The design gets different inputs over time.
- The outputs can be observed and analyzed.
- it can be checked, if the output shows the wanted behaviour.

Verilog Testbenches

1. The Verilog that shall get simulated is now handled as the **Device Under Test (DUT)**. It still is the same Verilog module(s).
2. The DUT gets a **stimulus input over time**. This is done with a **testbench**, also written in Verilog. In this simple example, a clock signal starts running into the DUT.
3. Simulated **output gets generated**. The output can be **asserted** (in tb.v) or **viewed** (i.e. gtkwave) as a diagram over time.

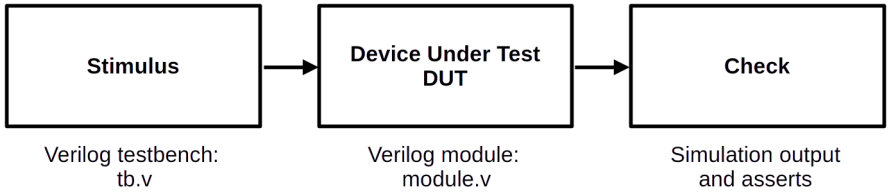


Figure 8: Device under test overview

iVerilog example

- The following slides contain an example with iVerilog.
- It is copied from the iVerilog documentation as it is.
- https://steveicarus.github.io/iverilog/usage/getting_started.html

DUT module: counter.v

counter.v:

```
1  module counter(out, clk, reset);
2
3    parameter WIDTH = 8;
4
5    output [WIDTH-1 : 0] out;
6    input clk, reset;
7
8    reg [WIDTH-1 : 0] out;
9    wire clk, reset;
10
11   always @(posedge clk)
12     out <= out + 1;
13
14   always @reset
15     if (reset)
16       assign out = 0;
17     else
18       deassign out;
19
20 endmodule // counter
```

Testbench verilog module: tb.v

tb.v:

```
module test;

   /* Make a reset that pulses once. */
   reg reset = 0;
   initial begin
      $dumpfile("test.vcd");
      $dumpvars(0,test);

      # 17 reset = 1;
      # 11 reset = 0;
      # 29 reset = 1;
      # 5  reset =0;
      # 513 $finish;
   end

   /* Make a regular pulsing clock. */
   reg clk = 0;
   always #1 clk = !clk;

   wire [7:0] value;
   counter c1 (value, clk, reset);

   initial
      $monitor("At time %t, value = %h (%0d)",
               $time, value, value);
endmodule // test
```

Simulation output with gtkwave

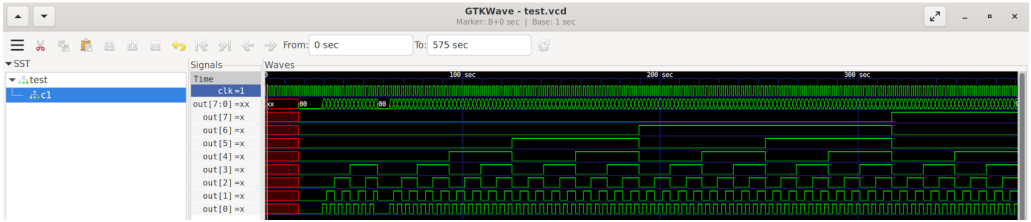


Figure 9: Simulation output of the counter with gtkwave

What is to see?

- The counter (8 bits) increments binary +1 with each clock input.

More Tools

SymbiYosys

Frontend for Yosys based formal verification flows.
https://github.com/YosysHQ/sby

Verilator

Converts Verilog to cycle-accurate C++/SystemC excetuable code.
https://github.com/verilator

CocoTB

Test and verify chip designs in Python. Supports VHDL and (System)Verilog.
https://github.com/cocotb/cocotb