Introduction to SystemVerilog HDL design

Today's Agenda

Intended topics for today's session

- What is an HDL and how to learn it?
- Learning by example – A first taste: A series of basic combinational and sequential SystemVerilog designs

Appendix

- History of SystemVerilog

FPGA-based SoC Design

International Master of Science in Electrical Engineering

Prof. Dr. C. Jakob
University of Applied Sciences Darmstadt
h_da
Faculty of Electrical Engineering and Information Technology
fbeit

Introduction to SystemVerilog HDL design

Objectives

By the end of this lecture you will be able to ...

- understand the basic structure of a SystemVerilog module
- design simple combinational circuits in SystemVerilog

FPGA-based SoC Design

International Master of Science in Electrical Engineering

Prof. Dr. C. Jakob

University of Applied Sciences Darmstadt
h_da
Faculty of Electrical Engineering and Information Technology
fbeit

Introduction to SystemVerilog HDL design

Recommended Readings

Textbooks, Application Notes, White Papers …

- Sutherland, S., "RTL Modeling with SystemVerilog for Simulation and Synthesis: Using SystemVerilog for ASIC and FPGA Design", CreateSpace Independent Publishing Platform, 2017
- Spear, C., "SystemVerilog for Verification: A Guide to Learning the Testbench Language Features", Springer, 3rd edition, 2012.

Prof. Dr. C. Jakob

University of Applied Sciences Darmstadt

h_da

Faculty of Electrical Engineering and Information Technology

fbeit

Introduction to SystemVerilog HDL – Part #1

International Master of Science in Electrical Engineering

Prof. Dr. C. Jakob

University of Applied Sciences Darmstadt

h_da

Faculty of Electrical Engineering and Information Technology

fbeit

HDL - Hardware Description Language
VHDL - Verilog Hardware Description Language

What is an HDL?

It is a language used for describing a digital system like a network switch or a microprocessor or a memory or a flipflop

- A Language to describe, simulate, and create hardware (popular examples are VHDL, Verilog or SystemVerilog).
- Despite similar syntax, an HDL cannot be used like typical programming languages
- Express the dimensions of timing and concurrency.
- At Register Transfer Level (RTL), an HDL design describes a hardware structure, not an algorithm.
- At behavioral level, HDL models describe only the behavior of the design with no implied structure.
- Something that you should keep in mind: “If you can’t draw it, don’t try to code it!”

How to learn an HDL …

- Learning by doing.
- Learning from mistakes.
- Try to understand what and why something went wrong … Otherwise nothing has been learned.
- Start designing simple designs, slowly add complexity.
- Start your design on paper …

SystemVerilog at a glance ...

- SystemVerilog represents a unified hardware design, specification and verification language.
- It provides support for all Verilog constructs (Verilog-2005). In addition, it combines synthesizable constructs from Accelera's language Superlog and Verification constructs from Synopsys OpenVera.
- In general, the feature-set of SystemVerilog (IEEE standard 1800-2012) can be divided into two distinct sections:

1. SystemVerilog for RTL design is an extension of Verilog-2005; all features of that language are available in SystemVerilog.
2. SystemVerilog for verification uses extensive object-oriented programming techniques and is more closely related to Java than Verilog.

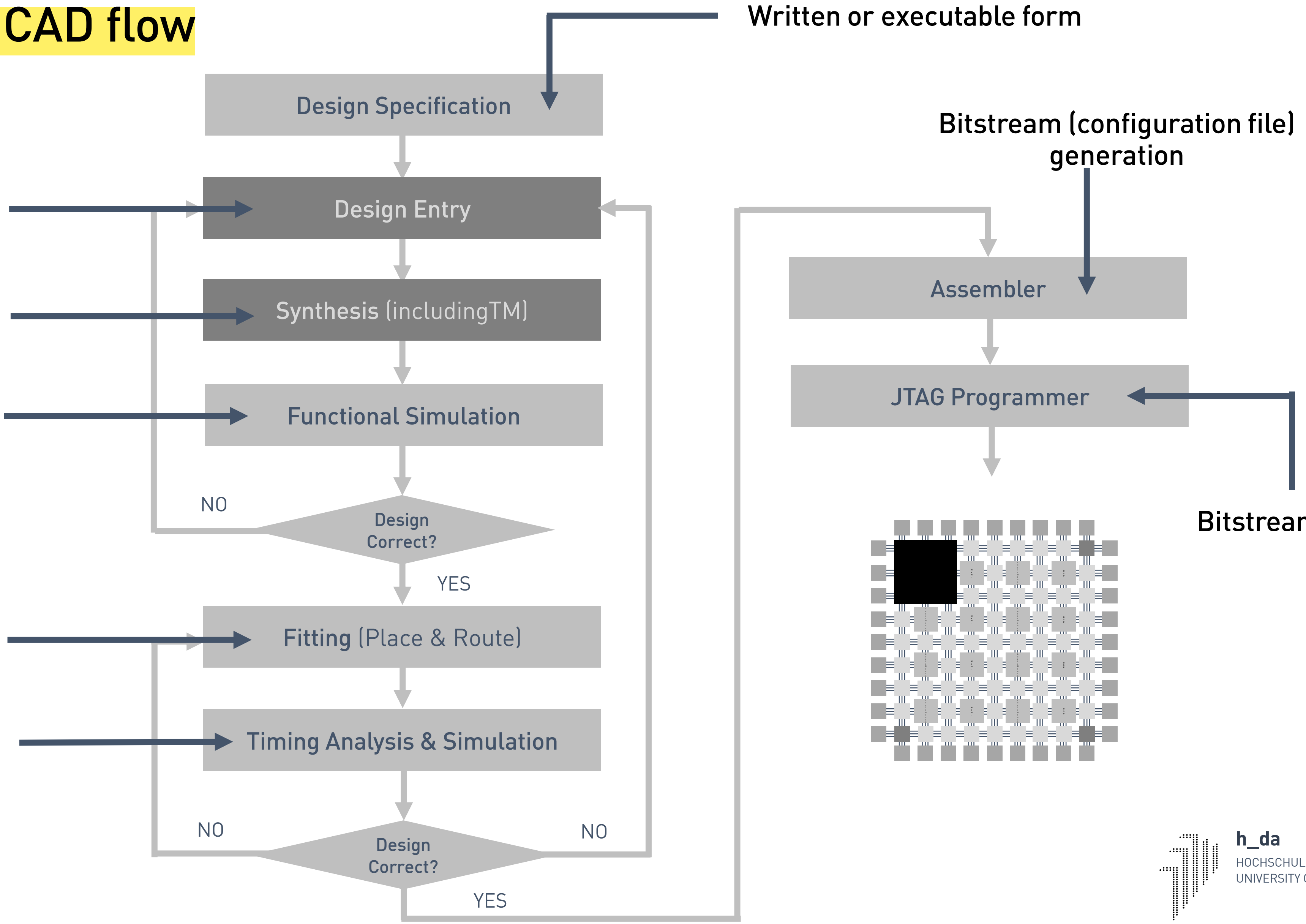
SystemVerilog at a glance …

- It is important to understand that SystemVerilog is both a ‘synthesis’ and ‘simulation/verification’ language.
- A certain subset of the language is used for synthesizing a hardware description into dedicated set of logic gates and flip-flops.
- Another subset of the language provides features for simulation and verification purposes. These constructs can not be translated into equivalent hardware structures …

SystemVerilog at a glance ...

- Despite the fact that SystemVerilog syntactically looks like ‘C’, it is no software programming language.
- This leads to the point that certain construct can be easily misinterpreted.
- It is a good strategy to think of the hardware synthesized that each line of SystemVerilog code will produce.

Intel Quartus Prime CAD flow



Written or executable form

Design Specification

Behavioral or structural HDL design description

Design Entry

Translate design into device specific primitives

Synthesis (includingTM)

Verify logic model and data flow, no timing delays

Functional Simulation

NO

Design Correct?

YES

Map primitives to specific locations on the target FPGA architecture and connect them

Fitting (Place & Route)

Mainly static timing analysis

Timing Analysis & Simulation

NO

Design Correct?

NO

YES

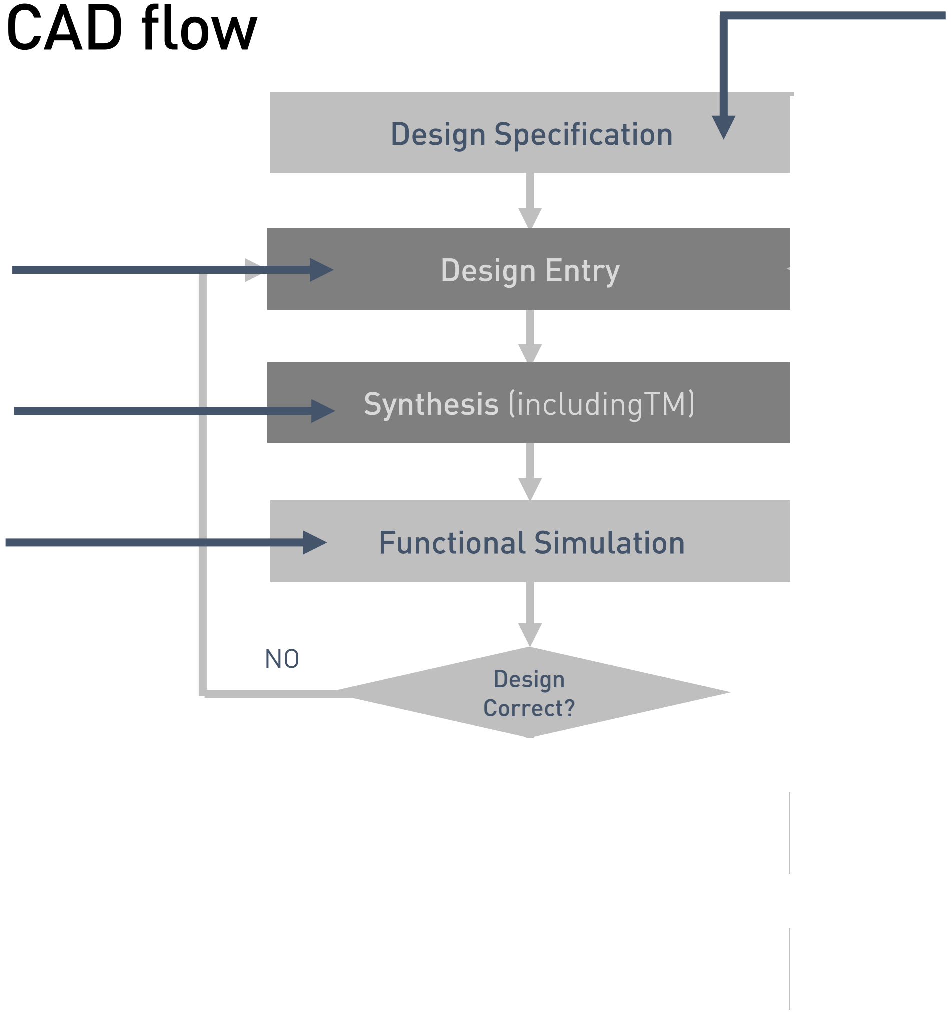
Bitstream (configuration file) generation

Assembler

JTAG Programmer

Bitstream Loader

Intel Quartus Prime CAD flow



Written or executable form

Behavioral or structural HDL design description

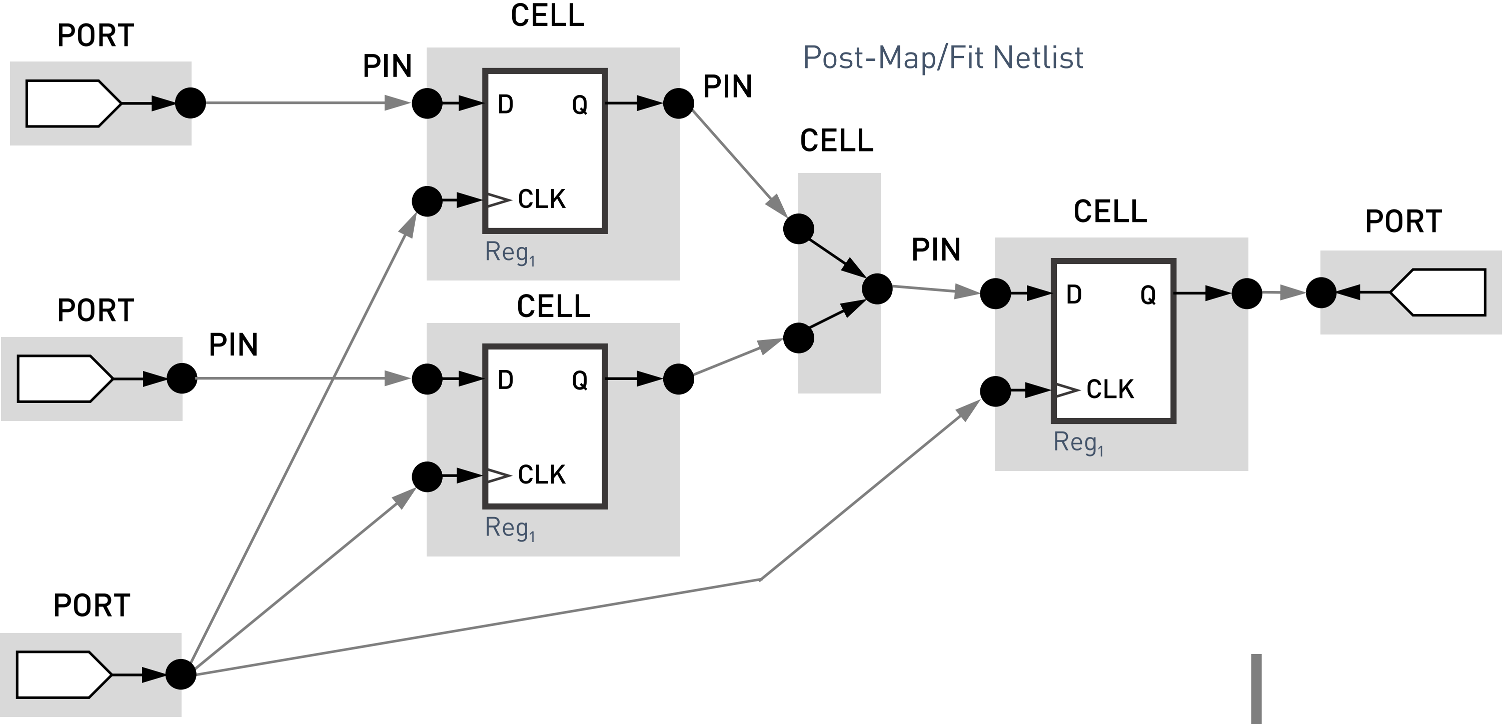
Translate design into device specific primitives

Verify logic model and data flow, no timing delays

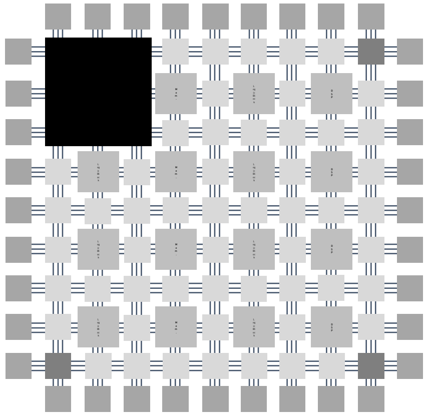
What is an HDL?

```
module sys_proc(
   input  logic [3:0] d0, d1,
   ...
   );
   always_ff@(posedge clk)
      if(reset_n == 1’b1)
         q = 1’b0;
      ...

endmodule
```

[SystemVerilog] Source Code Excerpt: sys_proc.sv



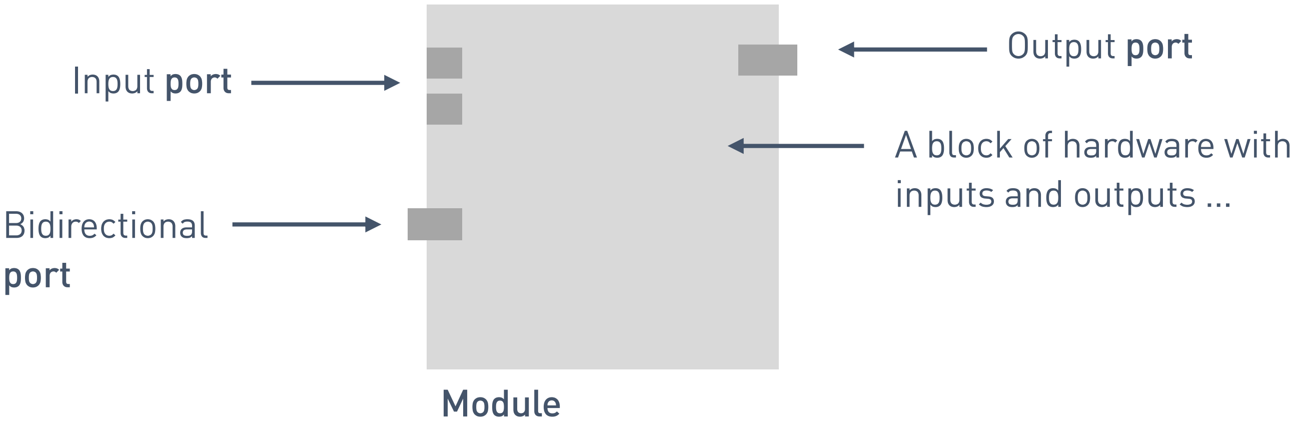
Post-Map/Fit Netlist

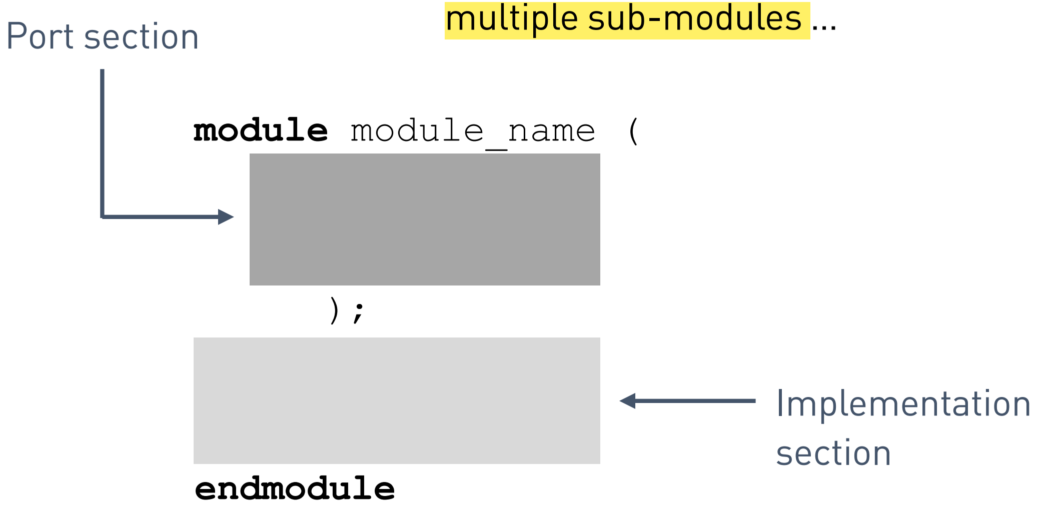


The Module – The basic building block in SystemVerilog

- The module is the basic unit of hierarchy in SystemVerilog.
- Modules are used to provide the **coarse-grained** structure of a design.

Module name should match the file name. A file can however contain multiple sub-modules ...





```
module module_name (
    );
endmodule
```

- The module could implement a simple AND gate, a multiplexer or even something complex such as CPU. In addition, a module can be a single element or collection of lower level modules.
- Ports are used to interface the module with the outside. They are either inputs, outputs or bidirectional (tri-state logic).
- In conclusion, modules describe the boundaries of a design unit [module, endmodule], its inputs and outputs [ports] as well how it works [behavioral or RTL code].
- SystemVerilog is able to model a design at different levels of abstraction. A high level express little detail, low levels express much. Boundaries between levels are often not well defined.

The Module – The basic building block in SystemVerilog

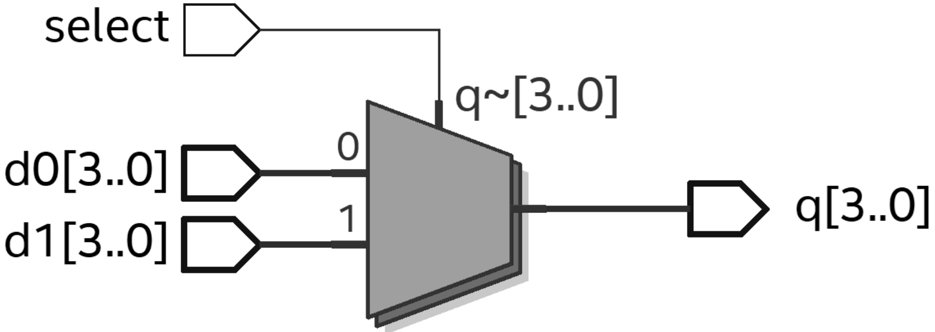
Port or IO section

1/1

```
module mux_2_4bit_comb(
   input logic [3:0] d0, d1,
   input logic select,
   output logic [3:0] q
   );
   always_comb
      if(select == 1)
         q = d1;
      else
         q = d0;
endmodule
```

Implementation section

[SystemVerilog] Source Code: mux_2_4bit_comb.sv



RTL netlist

The Module – The basic building block in SystemVerilog

This is a **graphical representation of** the design netlist after Analysis & Elaboration and netlist extraction, but before Quartus Prime synthesis and fitting optimizations. **This RTL netlist is not the final structure of the design**, because not all optimizations are included; instead it is the closest possible view to the original RTL design

Module name, must be identical with the file name

data typ: single bit

1/1

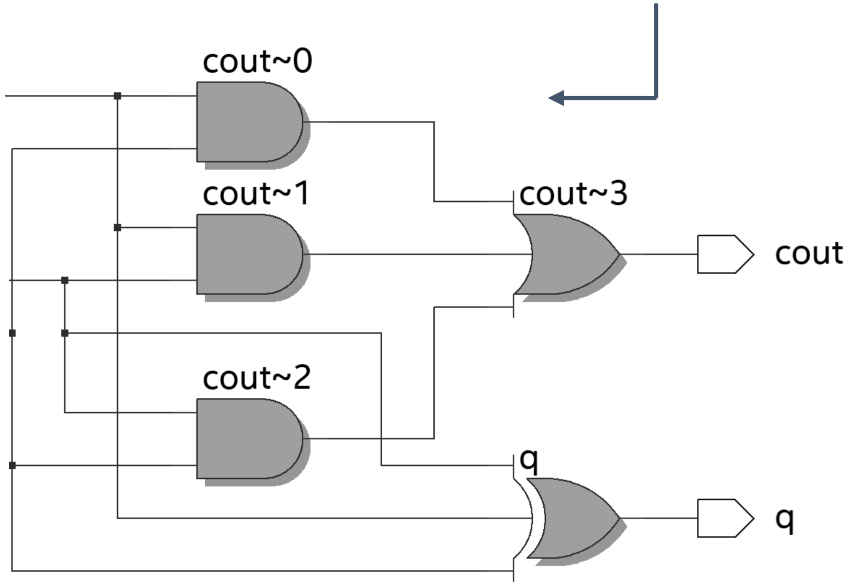
```systemverilog
// ...
module full_adder(
   input logic d0, d1, cin, output logic q,
   output logic cout
   );

   assign q    = d0 ^ d1 ^ cin;
   assign cout = (d0 & d1) | (d0 & cin) | (d1 & cin);

endmodule
```

port name

continuous assignment. another way for describing combinational logic

logical expression

[SystemVerilog] Source Code: full_adder.sv



RTL netlist

Notes

- SystemVerilog is case sensitive: cin and Cin are not the same
- No names start with numbers

Structural and behavioral HDL descriptions

```
module full_adder(
   input  logic d0, d1, cin,
   output logic q, cout
   );

   logic tmp_sum, tmp_cout_0, tmp_cout_1;

   half_adder inst_0( .d0(d0),d1(d1),.q(tmp_sum),
                      .cout(tmp_cout_0));
   half_adder inst_1( .d0(cin),d1(tmp_sum),.q(q),
                      .cout(tmp_cout_1));

   assign cout = tmp_cout_0 | tmp_cout_1;

endmodule
```

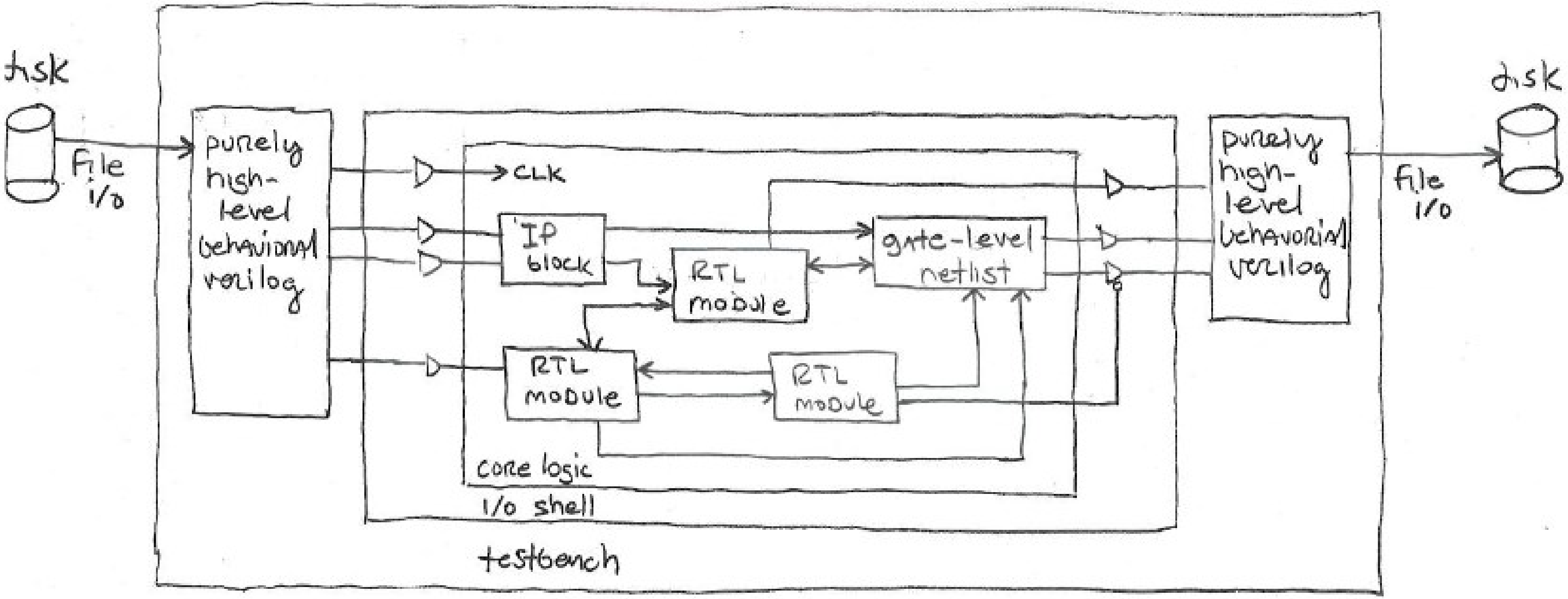
Structural description, describes the interconnect and usage of lower-level components

Module

```
module add_sub_4bit(
   input  logic [3:0] d0, d1, input logic cin,
   output logic [3:0] q, output logic cout
   );
   function logic [4:0] add (input logic [3:0] a, b,
   input logic cin );
      logic    ·0] s; logic c; c = cin;
      fo        = 0; i < 4; i++) begin
                  a[i] ^ b[i] ^ c; c = (a[i] & b[i])|(a[i] &
                                        |(c & c[i]);
      en
      adc          s};
   endfunc

   always_comb
      if(operation)
         {cout, q} = adder(d0, ~d1, 1);
      else
         {cout, q} = adder(d0, d1,  0);
endmodule
```

Behavioral description, describes the algorithmic behavior of the module rather than its structure

The Module – The basic building block in SystemVerilog



- A hierarchical design has a top level module and lower level ones. Lower level modules are instantiated within the higher level module. Lower level modules are connected together with wires.

Top-down and bottom-up design HDL descriptions

Top-down approach

CPU

Start here, rewrite behavior into smaller structures until you reach primitive behavior

Integer Unit

Cache

Register Set

ALU

Barrel Shifter

Adder

Multiplier

Bottom-up approach

CPU

Integer Unit

Cache

Register Set

ALU

Barrel Shifter

Adder

Multiplier

Start here, compose structure into larger modules until you reach the intended top-level behavior

Describing Combinational Logic using SystemVerilog

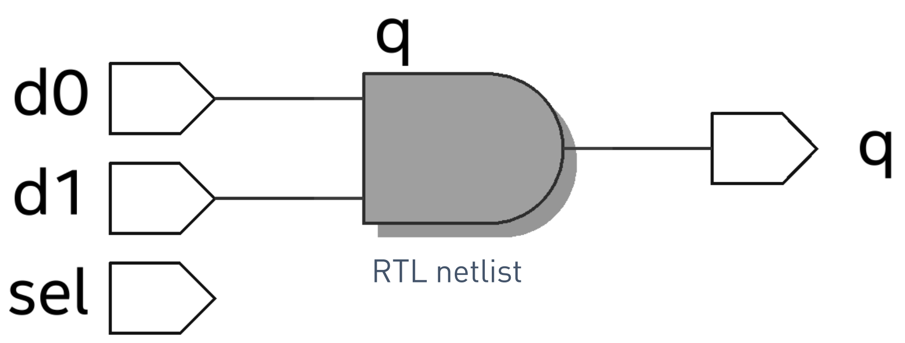
A series of basic SystemVerilog designs

Simple AND gate - Logical bit-wise operators

1/1

```
module and_2_1bit(
   input  logic d0, d1,
   output logic q
   );

   // 2-input AND gate
   assign q = d0 & d1;

endmodule
```

[SystemVerilog] Source Code: and_2_1bit.sv



RTL netlist

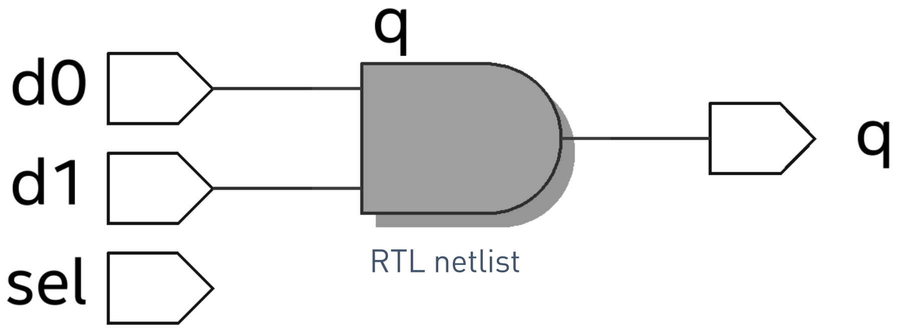
Notes

- The above example shows a so called continuous assignments (**assign**). These constructs are intended for modelling combinational logic.
- A continuous assignment drives a net similar to how a gate drives a net. The expression on the right hand side can be thought of as a combinatorial circuit that drives the net continuously.

Simple AND gate - Logical bit-wise operators

1/1

```
module and_2_1bit(
   input  logic d0, d1,
   output logic q
   );

   // 2-input AND gate
   always_comb
      q = d0 & d1;

endmodule
```



RTL netlist

[SystemVerilog] Source Code: and_2_1bit.sv

Notes

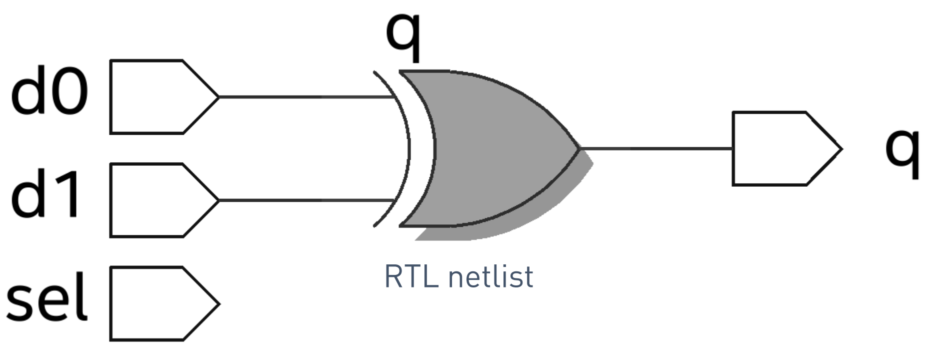
- An alternative way to model combinational logic is to use the always_comb construct (in conjunction with **blocking** assignment operators).
- The always_comb implicitly creates a complete sensitivity list including all variables and nets that are read in the process (pretty much the same always as @* construct in Verilog-2001).
- Please note that variables on the LHS of assignments cannot be assigned by other procedural blocks.

Simple XOR gate - Logical bit-wise operators

1/1

```
module xor_2_1bit(
   input  logic d0, d1,
   output logic q
   );

   // 2-input XOR gate
   assign q = d0 ^ d1;

endmodule
```

[SystemVerilog] Source Code: xor_2_1bit.sv



RTL netlist

Notes

- Bit-wise AND: &
- Bit-wise OR: |
- Bit-wise XOR: ^
- Bit-wise NOT: ~

Simple AND gate - Logical bit-wise operators

1/1

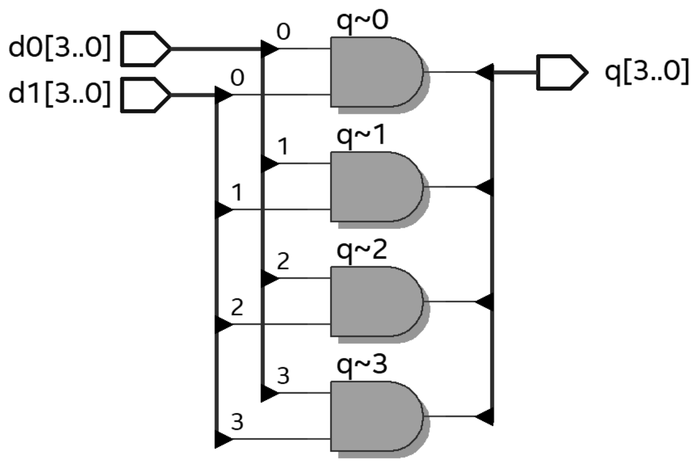
```
module and_2_4bit(
   input  logic [3:0] d0, d1,
   output logic [3:0] q
   );

   // four 2-input AND gates
   assign q = d0 & d1;

endmodule
```

[SystemVerilog] Source Code: and_2_4bit.sv

- Four bit wide vector, little-endian convention

Notes

- Bitwise operators perform bit-oriented operations on vectors



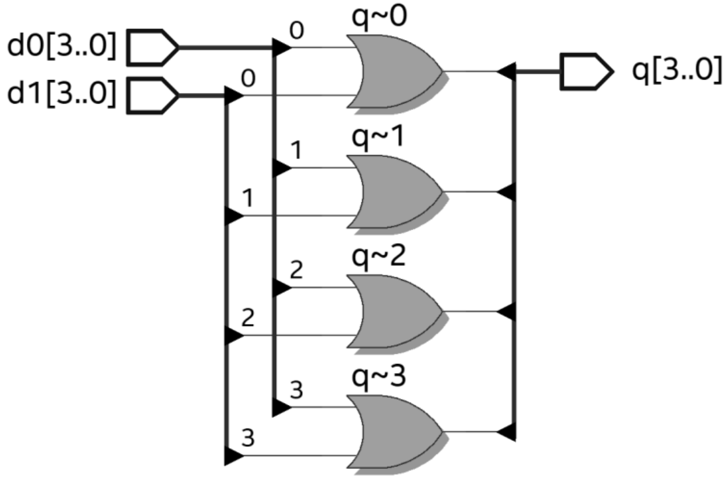
RTL netlist

Multi-bit Boolean gates - Logical bit-wise operators

1/1

```
module or_2_4bit(
  input  logic [3:0] d0, d1,
  output logic [3:0] q
  );

  // four 2-input OR gates
  assign q = d0 | d1;

endmodule
```

[SystemVerilog] Source Code: or_2_4bit.sv



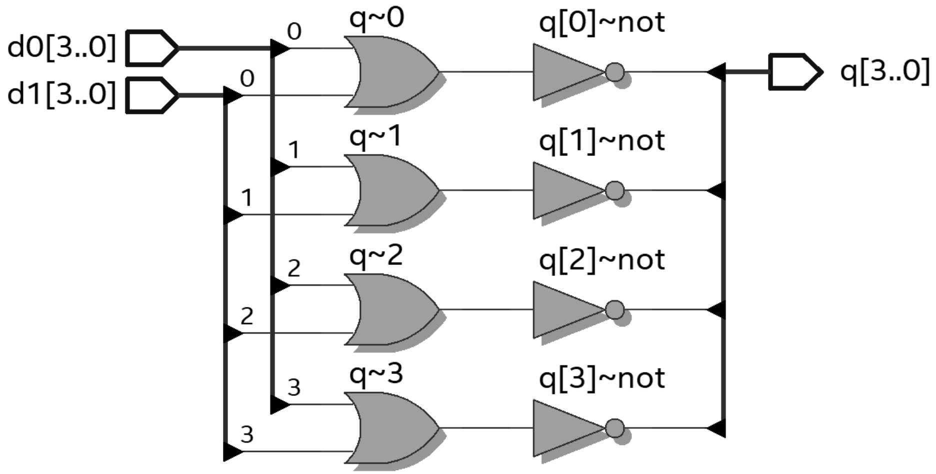
RTL netlist

Multi-bit Boolean gates - Logical bit-wise operators

1/1

```systemverilog
module nor_2_4bit(
  input  logic [3:0] d0, d1,
  output logic [3:0] q
  );

  // four 2-input NOR gates
  assign q = ~(d0 | d1);

endmodule
```

[SystemVerilog] Source Code: nor_2_4bit.sv



RTL netlist

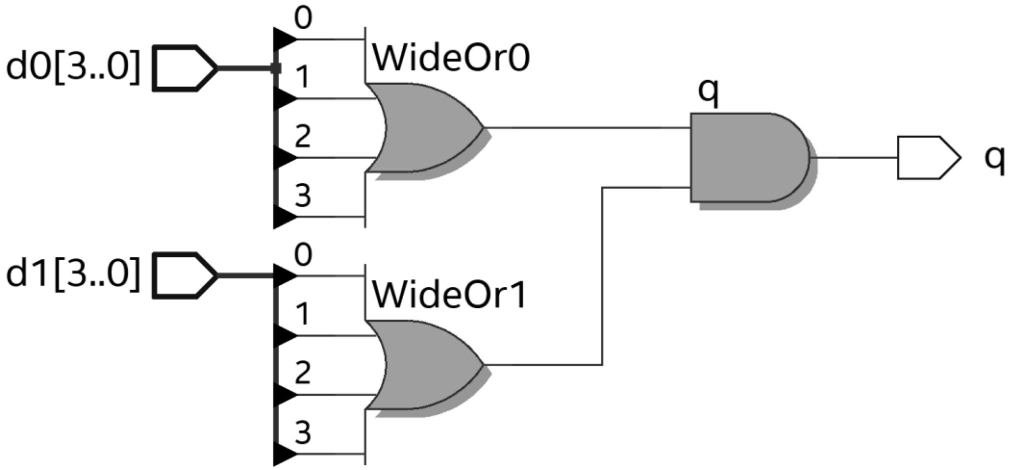
Logical operators

IMP

1/1

```
module logical_and_2_4bit(
   input  logic [3:0] d0, d1,
   output logic q
   );

   assign q = (d0 && d1);

endmodule
```

[SystemVerilog] Source Code: logical_and_2_4bit.sv



RTL netlist

Notes

- Logical operators evaluate to a 1-bit value.
- Operands not equal to zero are equivalent to one.
- Verilog Logical Operators:

Logical AND:	&&
Logical OR:	||
Logical NOT:	!

Appendix

International Master of Science in Electrical Engineering

Prof. Dr. C. Jakob

University of Applied Sciences Darmstadt

h_da

Faculty of Electrical Engineering and Information Technology

fbeit

History of SystemVerilog

- Verilog was developed by Gateway Design Automation as a proprietary language for logic simulation (Verilog-XL simulator) in 1984. The term Verilog is made-up of the two words "verification" and "logic“.
- Gateway Design Automation was acquired by Cadence Design Systems in 1989.
- Verilog became an open standard in 1990 when Cadence Design Systems released Verilog to the public domain. The administration of the language was assigned to the Open Verilog International (OVI) organization which was founded in the same year. OVI had the task of taking the language through the IEEE standardization process.
- In 1995, Verilog was adopted as an IEEE standard (IEEE standard 1364-1995, also denoted as Verilog-95).
- A significantly revised version of Verilog-95 became an IEEE standard in 2001 (IEEE standard 1364-2001, also known as Verilog-2001).

History of SystemVerilog contd.

- The Accellera Systems Initiative was founded in 2000 through the unification of Open Verilog International and VHDL International. Accellera was established as an independent, non-profit organization dedicated to create, support, promote, and advance system-level design, modeling, and verification standards.
- In the early 2000s, Accellera started to develop SystemVerilog as an extension of the Verilog IEEE 1364-2001 standard.
- In 2005, SystemVerilog was adopted as an IEEE standard (1800-2005).
- In the same year, a (slightly) modified and extended Version of Verilog-2001 became an IEEE Standard (IEEE 1364-2005).
- In 2009, the SystemVerilog standard (IEEE 1800-2005) was merged with the base Verilog (IEEE 1364-2005) standard, forming IEEE Standard 1800-2009.
- The current version is IEEE standard 1800-2012.