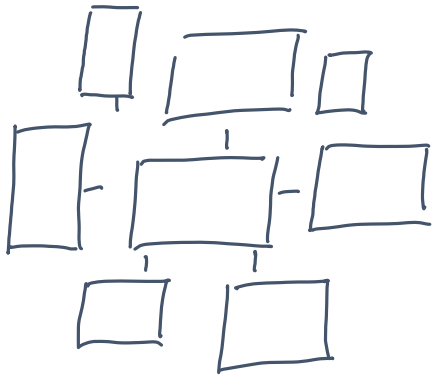


FPGA-based SoC Design

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

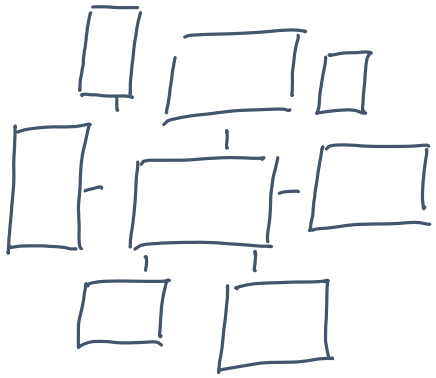
fbeit

Introduction to SystemVerilog HDL design

Today's Agenda

Intended topics for today's session

- Learning by example – A first taste: A series of basic combinational and sequential SystemVerilog designs
- Conclusion and coding guidelines



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

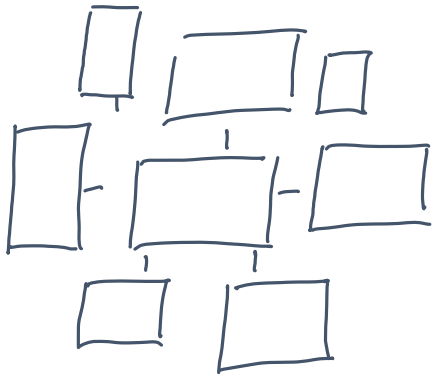
fbeit

Introduction to SystemVerilog HDL design

Objectives

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

- design simple combinational circuits in SystemVerilog
- design simple clock-synchronous logic 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

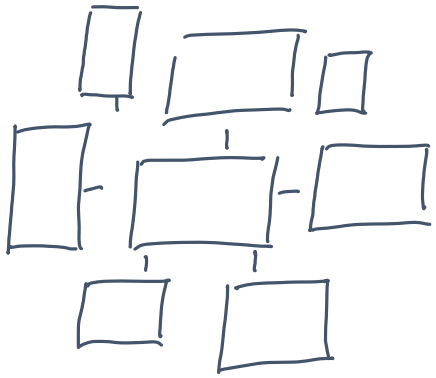
fbeit

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.



Introduction to SystemVerilog HDL – Part #2

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

Combinational logic using for-loops

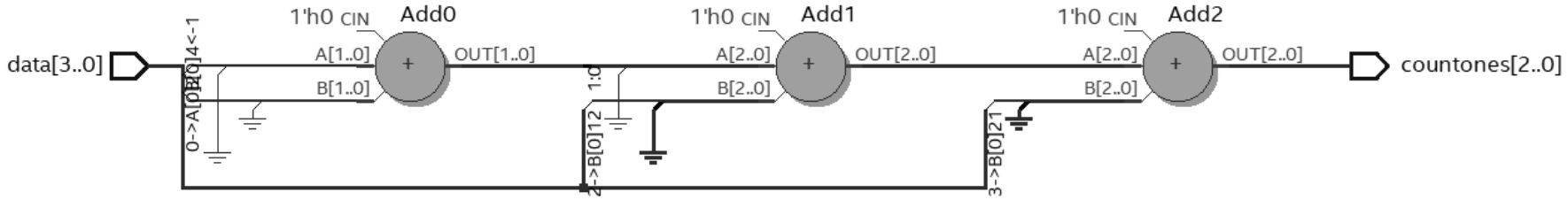
1/1

```
1.  module count_bits(
2.      input logic  [3:0] data,
3.      output logic [2:0] q
4.  );
5.      always_comb begin
6.          q = 0;
7.          for(int index = 0; index < 4; index++) begin
8.              q = q + data[index];
9.          end
10.     end
11.
12.  endmodule
```

[SystemVerilog] Source Code: count_bits.sv

Notes

- A for-loop statement is used to create multiple instances of the hardware structure described within the body of the for loop …
- The synthesis tool actually unrolls the loop.
- We will get to know even more powerful constructs to replicate hardware in future lectures



Combinational logic using for-loops

1/1

```
1.  module adder_4bit_behavioral(
2.     input  logic [3:0] d0, d1, input logic cin,
3.     output logic [3:0] q, output logic cout
4.     );
5.
6.     logic [4:0] carry;
7.     always_comb begin
8.        carry[0] = cin;
9.        for(int i = 0; i < 4; i++) begin
10.          q[i]      = d0[i] ^ d1[i] ^ carry[i];
11.          carry[i+1] = d0[i] & d1[i]    |
12.                       d0[i] & carry[i] |
13.                       d1[i] & carry[i];
14.       end
15.       cout = carry[4];
16.    end
17. endmodule
```

[SystemVerilog] Source Code: adder_4bit_behavioral.sv

Notes

- The "for statement" is executed at compile time (comparable to a macro expansion in 'C').
- Due to that, the for loop must have a specific range.

Combinational logic using for-loops

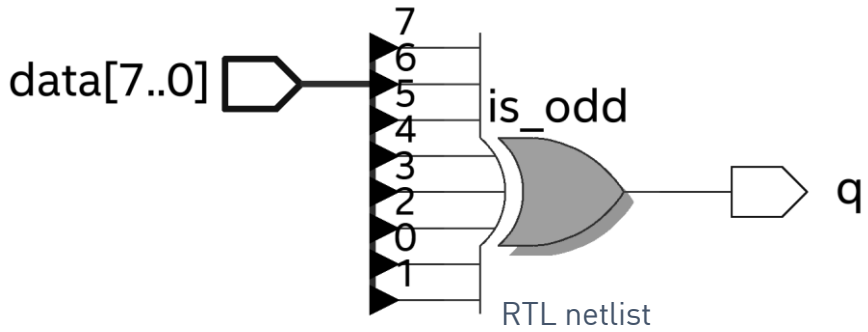
1/1

```
1.  module parity_check(
2.     input logic [7:0] data,
3.     output logic q
4.     );
5.
6.     logic is_odd;
7.     integer i;
8.     always_comb begin
9.        is_odd = 0;
10.       for(i=0; i <= 7; i++) begin
11.                is_odd = is_odd ^ data[i];
12.       end
13.    end
14.
15.    assign q = is_odd;
16.
17. endmodule
```

[SystemVerilog] Source Code: parity_check.sv

Notes

- This is a pretty complex description for something that can be implemented in single line of code using the XOR reduction operator.
- However, it again clearly illustrates the use of the for loop …



RTL netlist

Combining different procedural statements

```
1.
2.   ...
3.   for(...) begin
4.      if(...)
5.         case(...)
6.           ...
7.         endcase
8.      else begin
9.           ...
10.     end
11.  end
12.  ...
```

1/1

[SystemVerilog] Source Code Excerpt

Notes

- Different procedural statements can be nested.
- The synthesis tool will start to translate from the inner-most scope outwards. For the upper example, it will first create multiplexer logic for the case statement, then produce multiplexer for the if-else part. Finally, all that logic is replicated based on the number of iterations of the for loop …

More on SystemVerilog operators ...

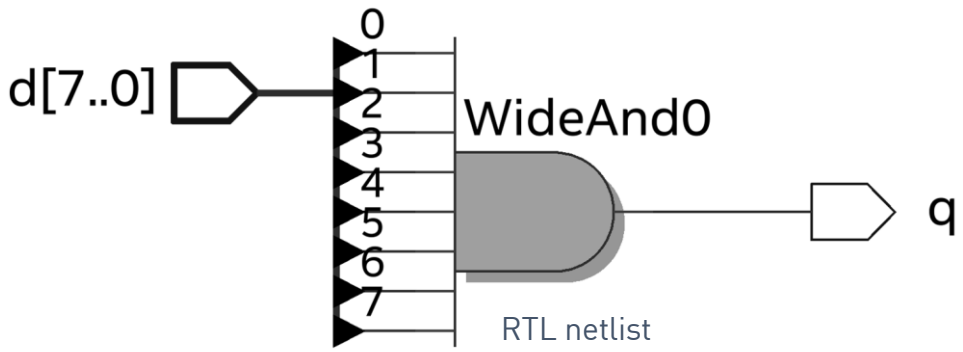
Introduction to SystemVerilog

fbeit
FACULTY OF ELECTRICAL ENGINEERING
AND INFORMATION TECHNOLOGY
Prof. Dr. C. Jakob

SystemVerilog Reduction Operators

```
1.  module and_8_reduction(
2.     input  logic [7:0] d,
3.     output logic q
4.     );
5.
6.     assign q = &d;
7.
8.  endmodule
```

1/1



[SystemVerilog] Source Code: and_8_reduction.sv

this is equivalent to writing:

```
assign q = d[0] & d[1] & d[2] & d[3] & d[4] & d[5] & d[6] & d[7]
```

Notes

- Overflow detection (AND)
- Verilog has six reduction operators, these operators accept a single vectored (multiple bit) operand, performs the appropriate bit-wise reduction on all bits of the operand, and returns a single bit result. Assume that the multi-bit signal is named d:

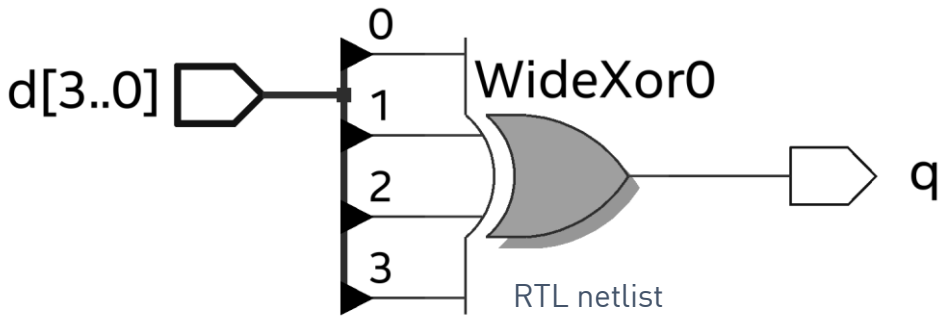
~| d	**NAND**
| d	**OR**
~| d	**NOR**
^ d	**XOR** (see next slide)
~^ d	**XNOR**
& d	**AND**

SystemVerilog Reduction Operators

1/1

```
1.  module xor_4_reduction(
2.     input  logic [3:0] d,
3.     output logic q
4.     );
5.
6.     assign q = ^d;
7.
8.  endmodule
```

[SystemVerilog] Source Code: xor_4_reduction.sv



this is equivalent to writing:

```
assign q = d[0] ^ d[1] ^ d[2] ^ d[3]
```

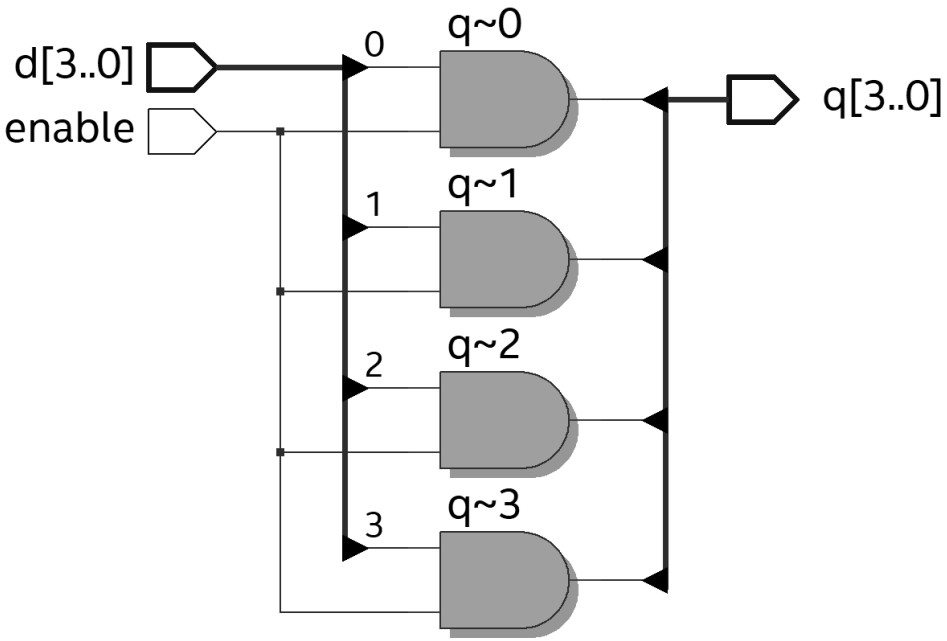
Notes

- Used for even parity bit generation.

SystemVerilog Replication Operator

```
1.  module bus_enable_4bit(                                    1/1
2.     input  logic [3:0] d,
3.     input  logic enable,
4.     output logic [3:0] q
5.     );
6.
7.     assign q = {4{enable}} & d;
8.
9.  endmodule
```

[SystemVerilog] Source Code: bus_enable_4bit.sv



RTL netlist

Notes

- Replication Operator { {} } - repetitive concatenation of the same number/signal.
- The operands are the number of repetitions, as well as the actual bus or the wire.
- In the above example, we realize a multi-bit switch: Forward the input signal or set the output low.

Bus Logic, Splitting and Reordering ...

1/1

```
1.  module bus_concatenation(
2.     input  logic [3:0] a,b,c,d,
3.     output logic [3:0] q
4.     );
5.
6.   assign q = {a[2],a[1],b[2],c[2]};
7.
8.  endmodule
```

[SystemVerilog] Source Code: bus_concatenation.sv

Notes

- Only four bits out of the three 4-bit wide input buses a, b and c are forwarded to the output q (see bit mapping).

Sequential Logic – Clock related logic

Introduction to SystemVerilog

Advantage of Clock-Synchronous Design – Guaranteeing Silicon Performance

- Excerpt from **Intel's Following Synchronous FPGA Design Practices** ... *The first step in good design methodology is to understand the implications of your design practices and techniques ...*

  Good synchronous design practices can help you meet your design goals consistently. Problems with other design techniques can include reliance on propagation delays in a device, which can lead to race conditions, incomplete timing analysis, and possible glitches. In a synchronous design, a clock signal triggers every event ...

  *... If you ensure that all the timing requirements of the registers are met, a synchronous design behaves in a predictable and reliable manner for all **process, voltage, and temperature** (PVT) **conditions**. You can easily migrate synchronous designs to different device families or speed grades ...*

Single-Bit Delay Flip-Flop (DFF)

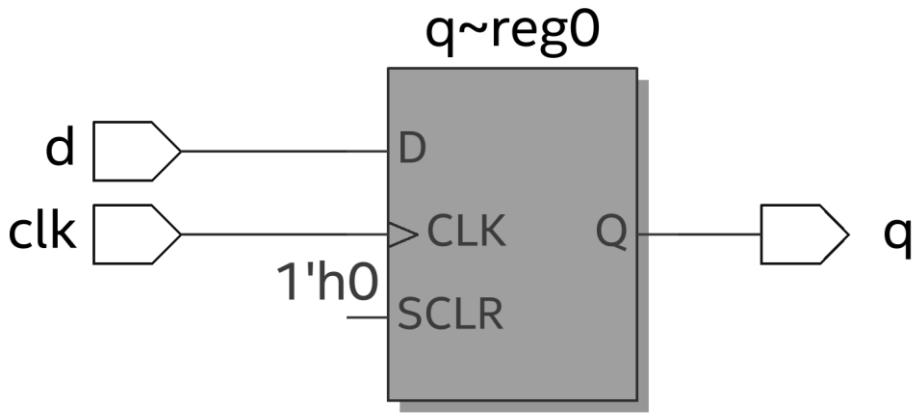
1/1

```
1. module dff(
2.    input  logic d, input logic clk,
3.    output logic q
4.    );
5.
6.    always_ff@(posedge clk)
7.       q <= d;
8.
9. endmodule
```

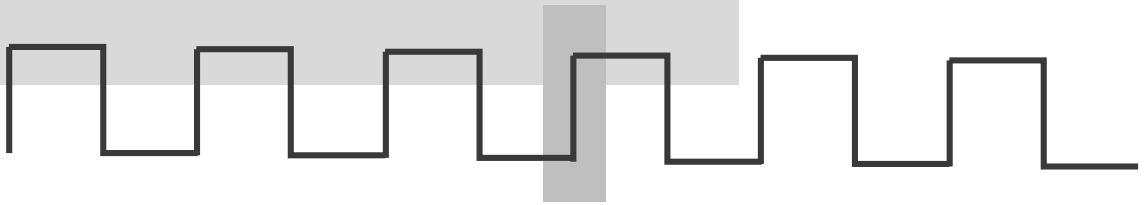
implements sequential logic

Process is triggered by the rising edge of clk

copy d to q



RTL netlist



[SystemVerilog] Source Code: dff.sv

Notes

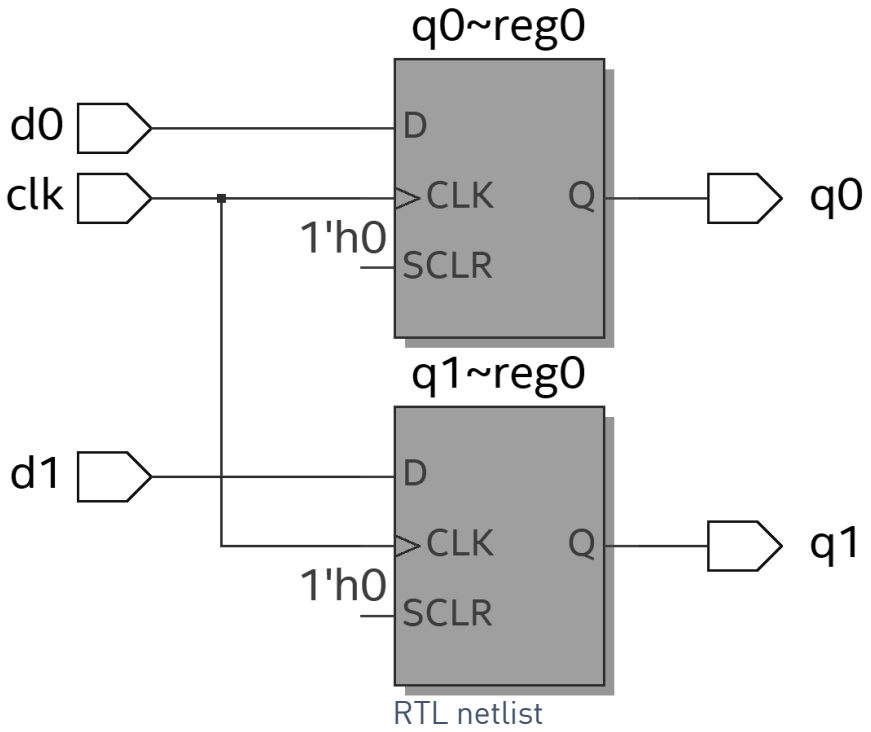
- The always_ff construct is used to describe synchronous logic (flip-flops or registers.
- In the above example, the output is updated with every rising edge of the clock signal. Please note that the always_ff construct must be always used in conjunction with a sensitivity list!
- Only use the always_ff construct with the **non-blocking** assignment operator (<=). Never use **blocking** assignments (=) in an always_ff block.
- Do not assign the same variable from more than one always_ff block (Race conditions in behavioral simulation, synthesizes incorrectly).

Two Single-Bit Delay Flip-Flops (DFF)

1/1

```
1. module dff_2bit(
2.     input  logic d0, d1, input logic clk,
3.     output logic q0, q1
4.     );
5.
6.     always_ff@(posedge clk) begin
7.        q0 <= d0;
8.        q1 <= d1;
9.     end
10.
11. endmodule
```

[SystemVerilog] Source Code: dff_2bit.sv



RTL netlist

Notes

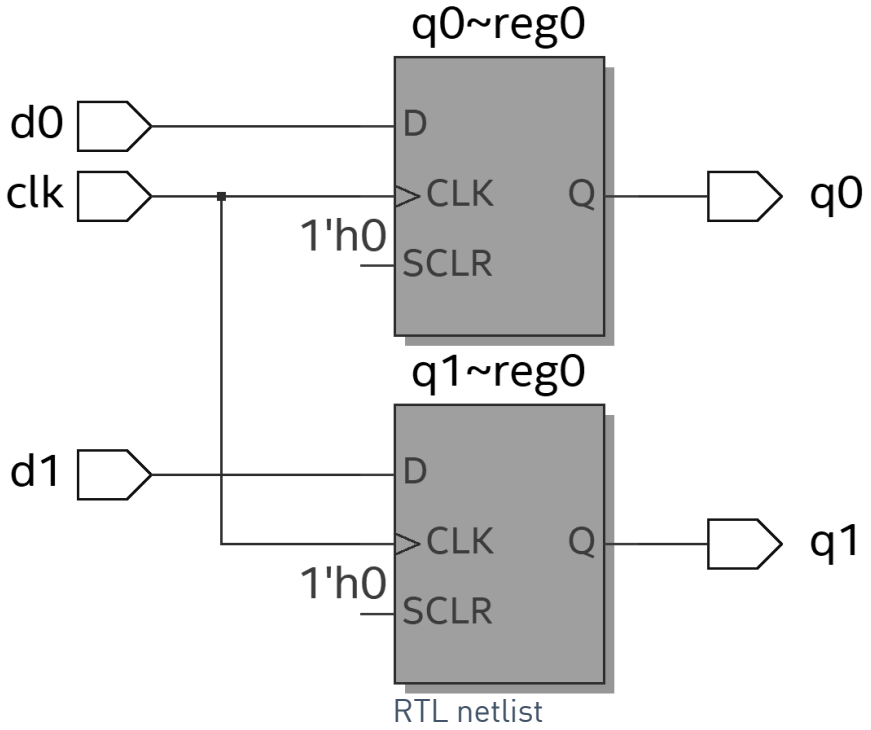
- Non-blocking assignments happen in parallel. In other words, if an always_ff block contains multiple non-blocking assignments, which are literally written sequentially, you should think of all of the assignments being set at exactly the same time (and thus in parallel).
- If there are multiple statements within always_ff block they must be bracketed with begin and end statements.

Two Single-Bit Delay Flip-Flops (DFF) contd.

1/1

```
1. module dff_2bit(
2.    input  logic [1:0] d, input logic clk,
3.    output logic [1:0] q
4.    );
5.
6.    always_ff@(posedge clk) begin
7.       q[0] <= d[0];
8.       q[1] <= d[1];
9.    end
10.
11. endmodule
```

[SystemVerilog] Source Code: dff_2bit.sv



RTL netlist

Notes

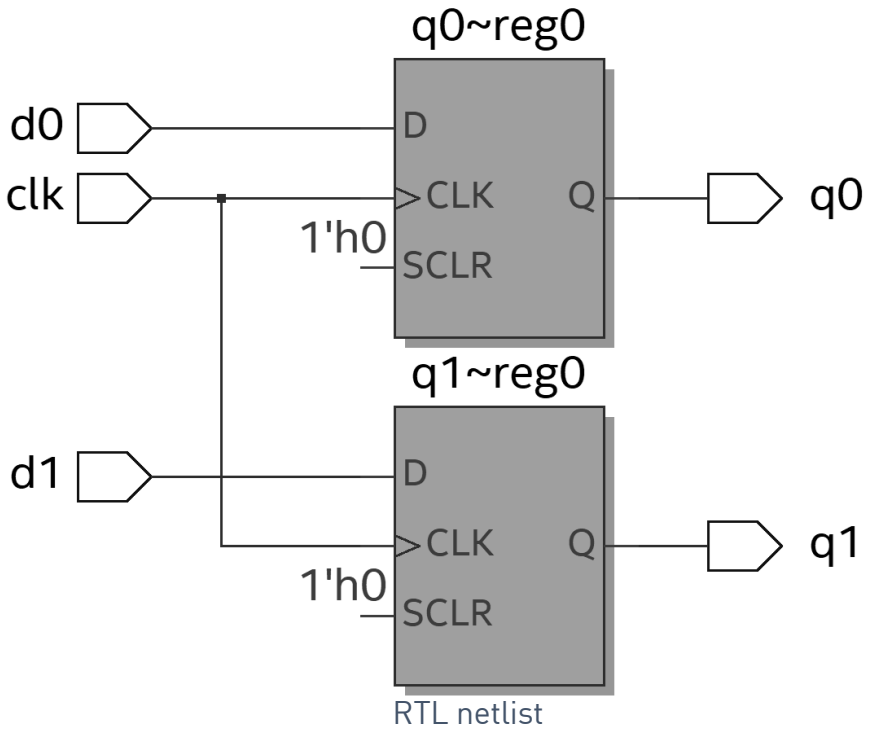
- The above example represents an alternative description of the previous one.

Two Single-Bit Delay Flip-Flops (DFF) contd.

1/1

```
1. module dff_2bit(
2.    input  logic [1:0] d, input logic clk,
3.    output logic [1:0] q
4.    );
5.
6.    always_ff@(posedge clk) begin
7.       q <= d;
8.    end
9.
10. endmodule
```

[SystemVerilog] Source Code: dff_2bit.sv



RTL netlist

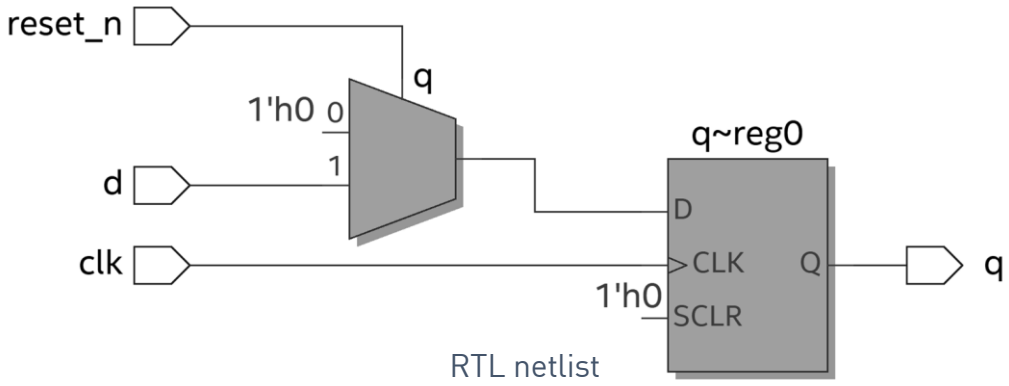
Notes

- The above example represents an alternative description of the previous two designs.

Synchronously resettable Delay Flip-Flop

1/1

```
1.  module dff_sreset(
2.     input  logic d, clk, reset_n,
3.     output logic q
4.     );
5.     always_ff@(posedge clk)
6.        if(reset_n == 0)
7.           q <= 0;
8.        else
9.           q <= d;
10. endmodule
```



[SystemVerilog] Source Code: dff_sreset.sv

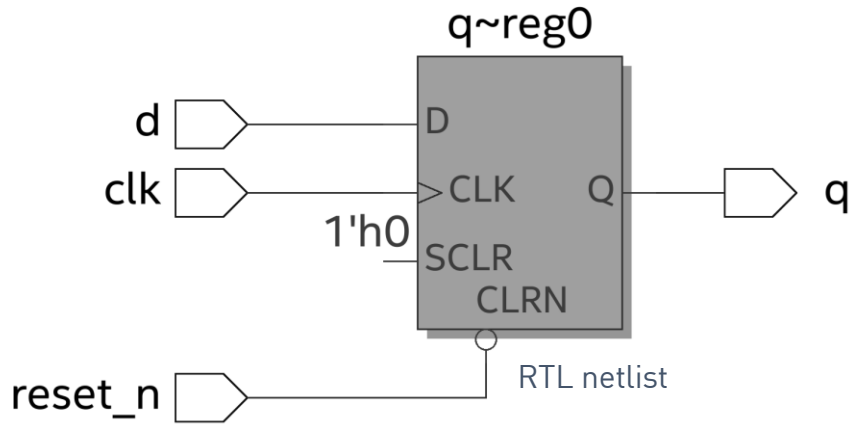
Notes

- In case of a synchronous reset, the state of the reset signal is evaluated with the occurrence of a rising clock edge.

Asynchronously resettable Delay Flip-Flop

1/1

```
1.  module dff_areset(
2.     input  logic d, clk, reset_n,
3.     output logic q
4.     );
5.     always_ff@(posedge clk, negedge reset_n)
6.        if(reset_n == 0)
7.           q <= 0;
8.        else
9.           q <= d;
10. endmodule
```



[SystemVerilog] Source Code: dff_areset.sv

Notes

- In case of a asynchronous reset, the reset signal is not sampled with the clock signal! It is fully asynchronous with respect to the clock.
- We are using the name extension "_n" to declare a low active signal.
- In the example above we are considering an **active low** asynchronous reset.

Synchronous or Asynchronous resets? Which one to use ?

Introduction to SystemVerilog

Note

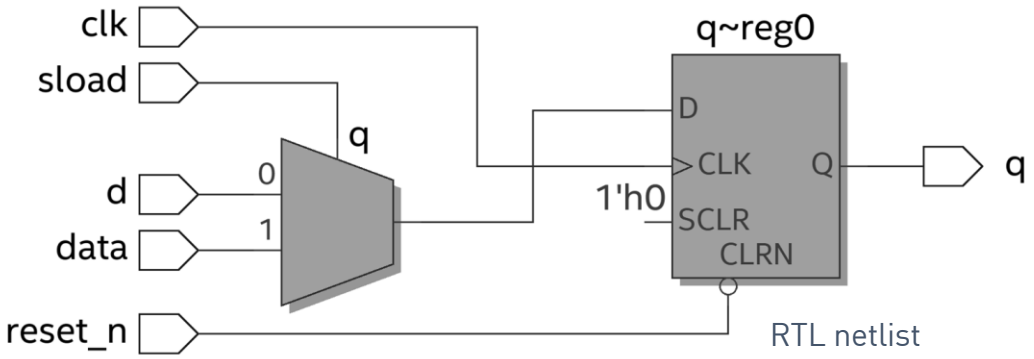
- In the FPGA context, this question is a bit more complex than it might look at the first glimpse. We will discuss this topic in greater detail in lectures about timing analysis.

Asynchronously resettable Delay Flip-Flop with synchronous load

1/1

```
1.  module dff_areset_sload(
2.     input  logic d, data, clk, reset_n, sload,
3.     output logic q
4.     );
5.     always_ff@(posedge clk, negedge reset_n)
6.        if(reset_n == 0)
7.           q <= 0;
8.        else if(sload)
9.           q <= data;
10.       else
11.          q <= d;
12. endmodule
```

short for sload == 1



[SystemVerilog] Source Code: dff_areset_sload.sv

Notes

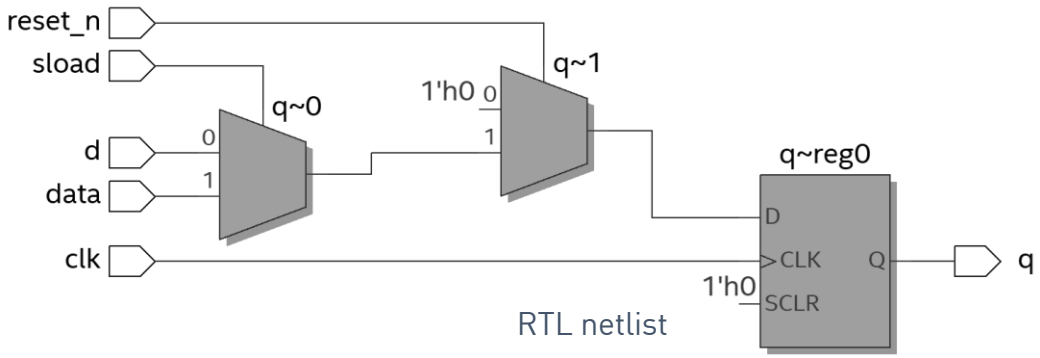
- The delay flip-flop is updated using sload with the occurrence of a rising clock edge.

Synchronously resettable Delay Flip-Flop with synchronous load

1/1

```
1.  module dff_areset_sload(
2.     input  logic d, data, clk, reset_n, sload,
3.     output logic q
4.     );
5.     always_ff@(posedge clk)
6.        if(reset_n == 0)
7.           q <= 0;
8.        else if(sload == 1)
9.           q <= data;
10.       else
11.          q <= d;
12. endmodule
```

[SystemVerilog] Source Code: dff_areset_sload.sv



RTL netlist

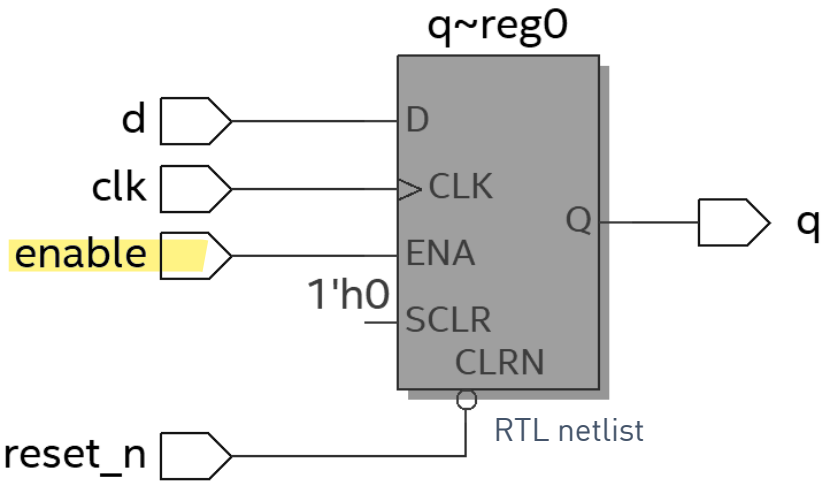
Notes

- The same circuit as before, however right now with a synchronous reset.
- As can be seen, the if-else construct infers a priority routing network.

Asynchronously resettable Delay Flip-Flop with dedicated enable input

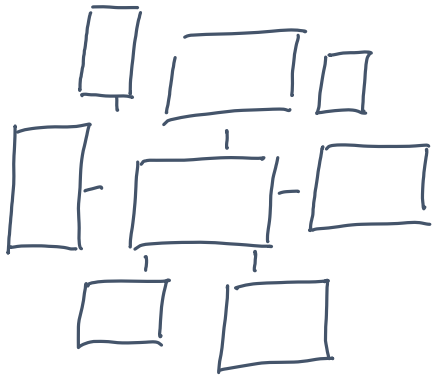

```
1.  module dff_areset_enable(
2.     input  logic d, clk, reset_n, enable,
3.     output logic q
4.     );
5.     always_ff@(posedge clk, negedge reset_n)
6.        if(reset_n == 0)
7.           q <= 0;
8.        else if(enable)
9.           q <= d;
10. endmodule
```

[SystemVerilog] Source Code: dff_areset_enable.sv



Notes

- The output is only updated if the enable signal is set high.
- The enable input configuration requires an asynchronous reset input.



SystemVerilog – A first set of coding guidelines

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

Coding Guidelines

- When modeling sequential (clock synchronous) logic, use non-blocking assignments!
- When modeling combinational logic with an always_comb block, use blocking assignments.
- When modeling both sequential and combinational logic within the same always_ff block, use non-blocking assignments.
- Do not mix blocking and non-blocking assignments within a single always block.
- Do not make assignments to the same variable from more than one always block.
- If using SystemVerilog for RTL design, we use the SystemVerilog logic data type to declare all point-to-point nets, for all variables (logic driven by always blocks), for all input ports as well as for all output ports.