Introduction to SystemVerilog HDL design

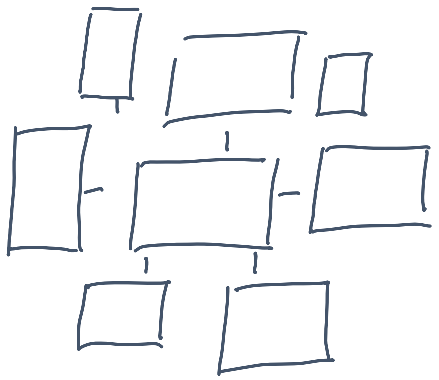
Today's Agenda

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

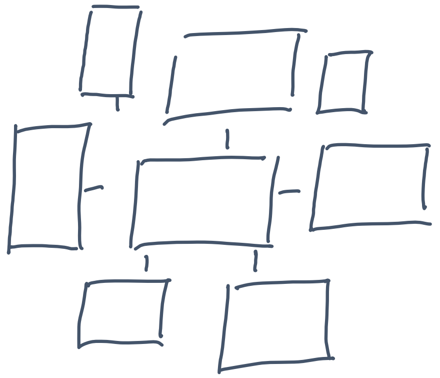
Prof. Dr. C. Jakob
University of Applied Sciences Darmstadt
h_da
Faculty of Electrical Engineering and Information Technology
fbeit

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

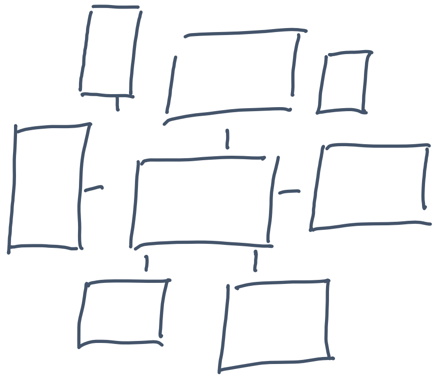
Prof. Dr. C. Jakob
University of Applied Sciences Darmstadt
h_da
Faculty of Electrical Engineering and Information Technology
fbeit

Introduction to SystemVerilog HDL design

Recommended Readings

Textbooks, Application Notes, White Papers …

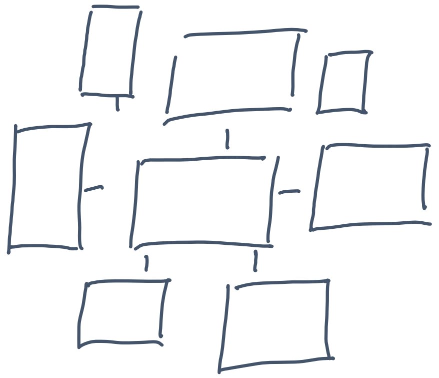
- Sutherland, S., "RTL Modeling with SystemVerilog for Simulation and Synthesis: Using SystemVerilog for ASIC and FPGA Design", CreateSpace Independent Publishing Platform, 2017
- Spear, C., "SystemVerilog for Verification: A Guide to Learning the Testbench Language Features", Springer, 3rd edition,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



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 – 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

What is an HDL?

- A Language to describe, simulate, and create hardware (popular examples are VHDL, Verilog or SystemVerilog).
- HDLs are an integral part of modern electronic design automation (EDA) systems.
- 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.

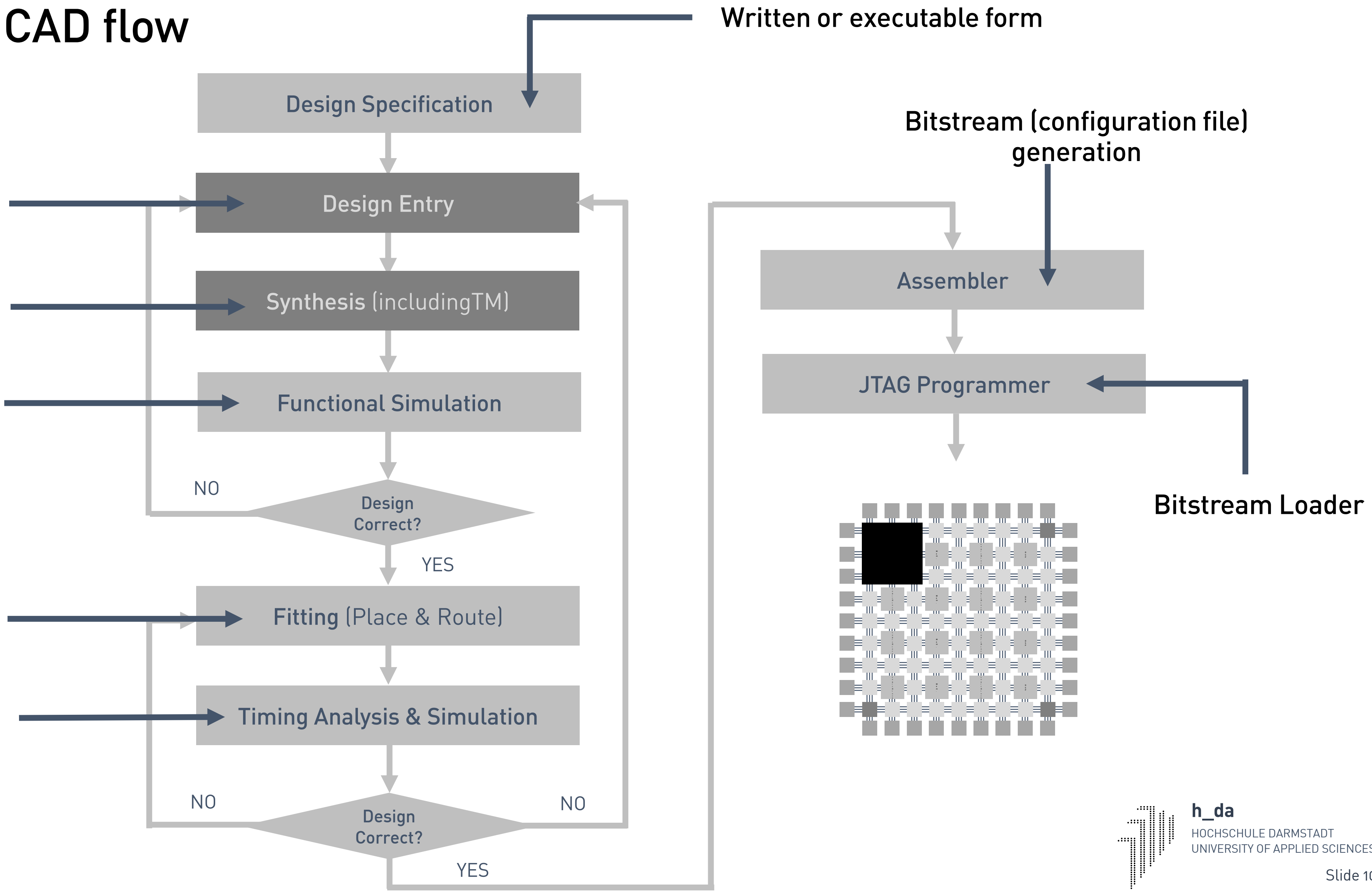
SystemVerilog at a glance …

- 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 …

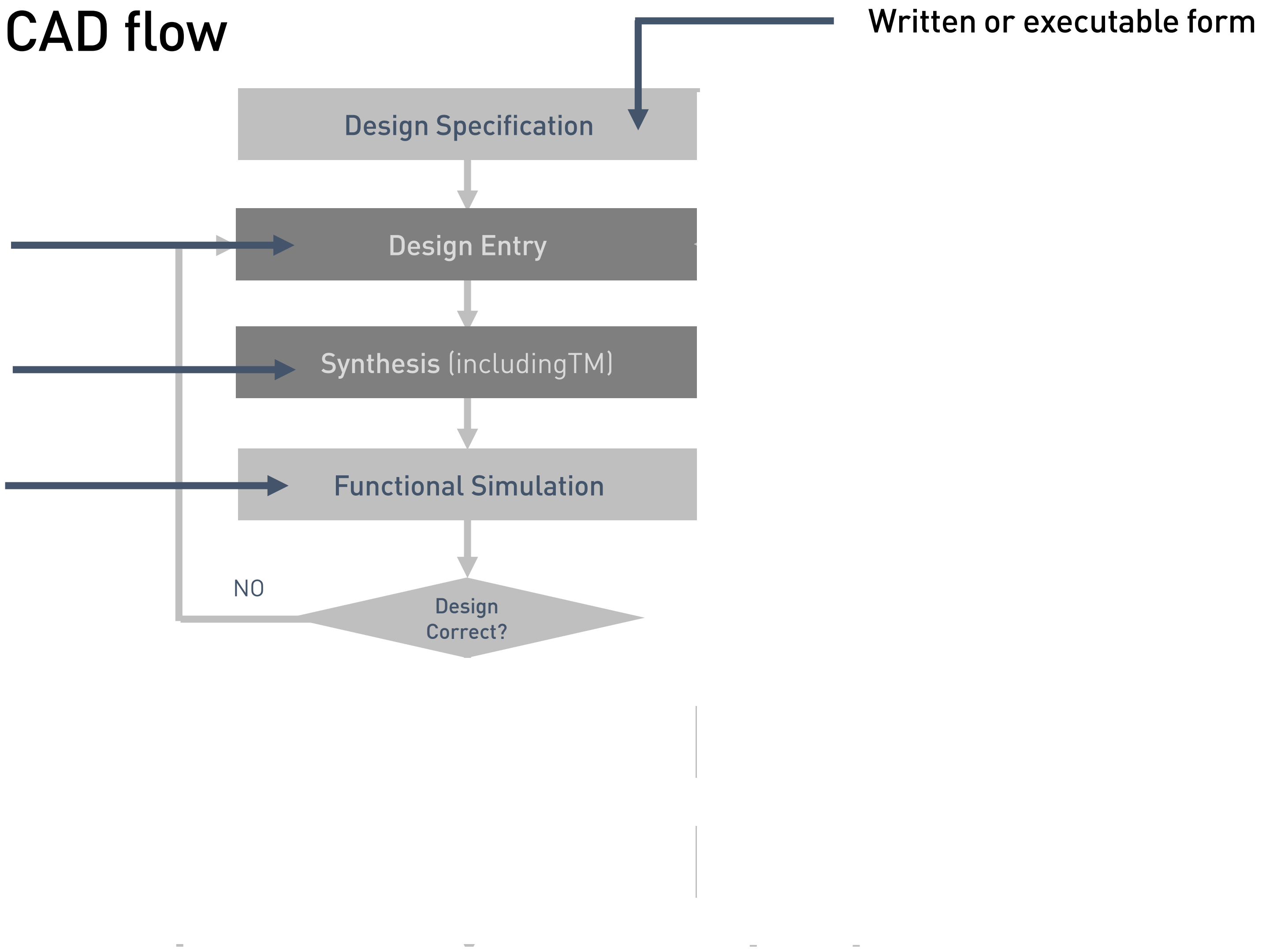
SystemVerilog at a glance …

- 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 a netlist of 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

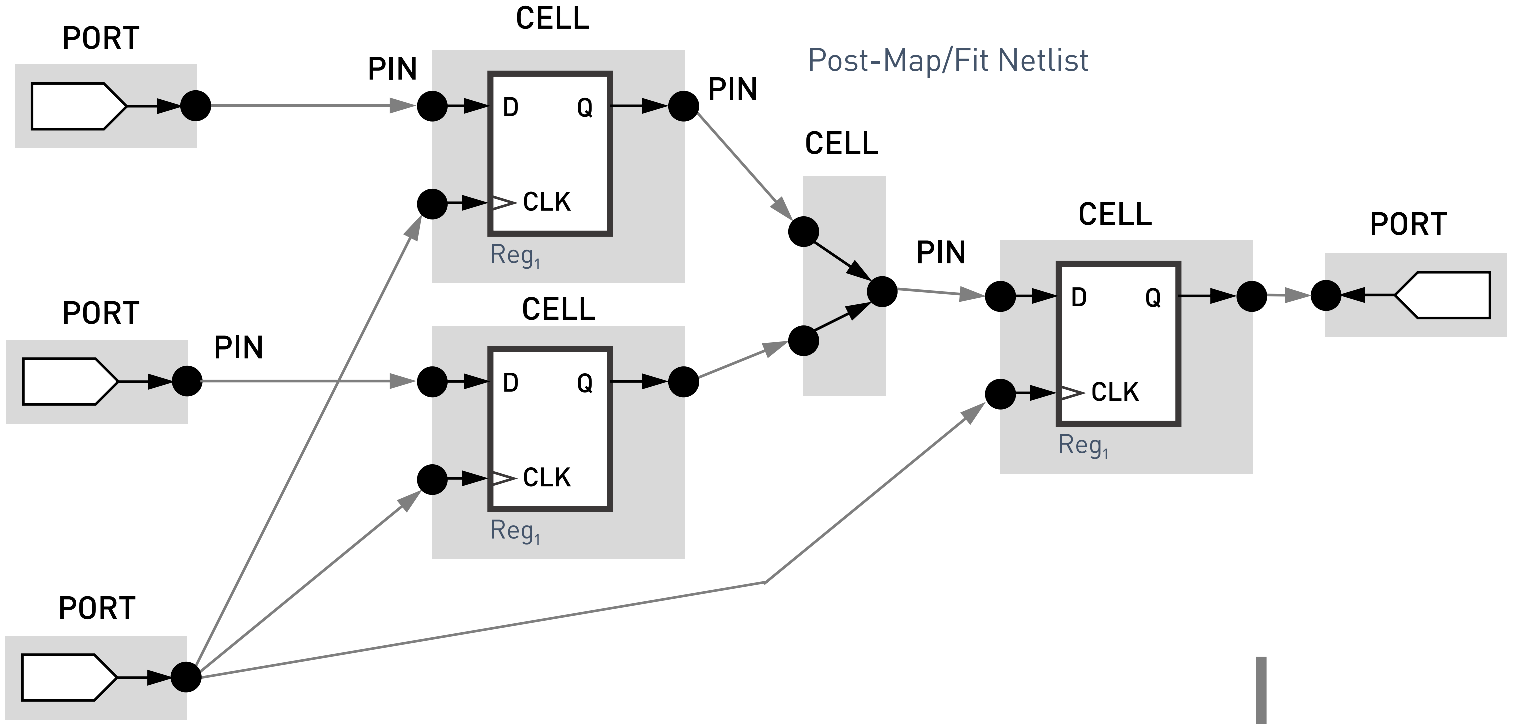
Intel Quartus Prime CAD flow

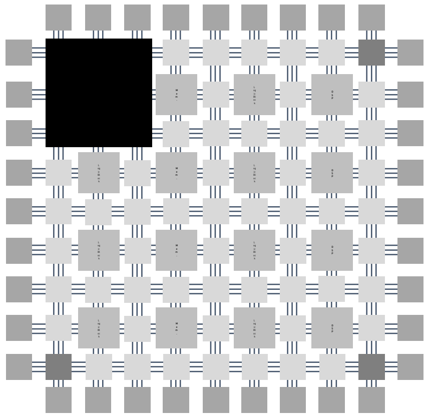

Written or executable form
Behavioral or structural HDL design description
Translate design into a netlist of device specific primitives
Verify logic model and data flow, no timing delays
Design Specification
Design Entry
Synthesis (includingTM)
Functional Simulation
NO
Design Correct?

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

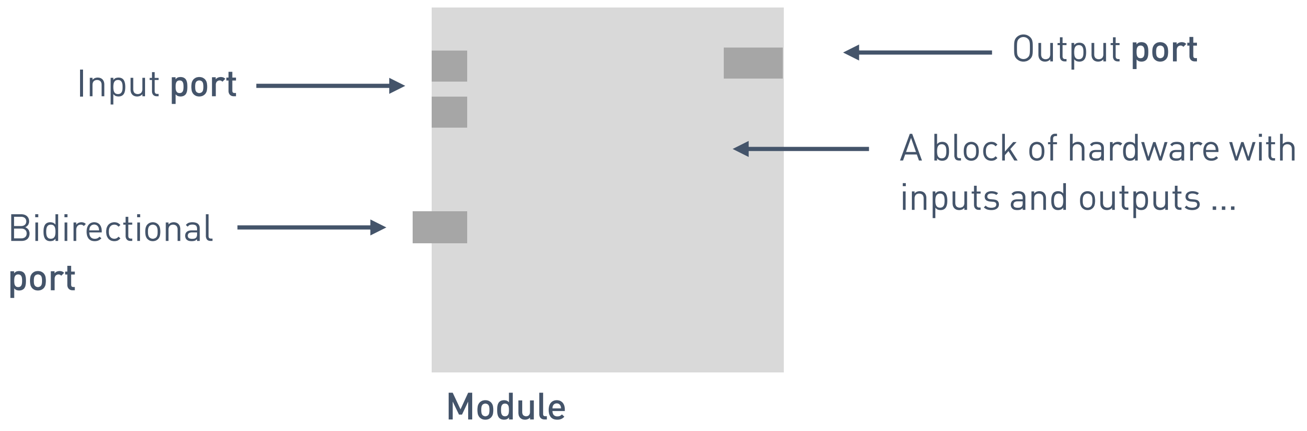


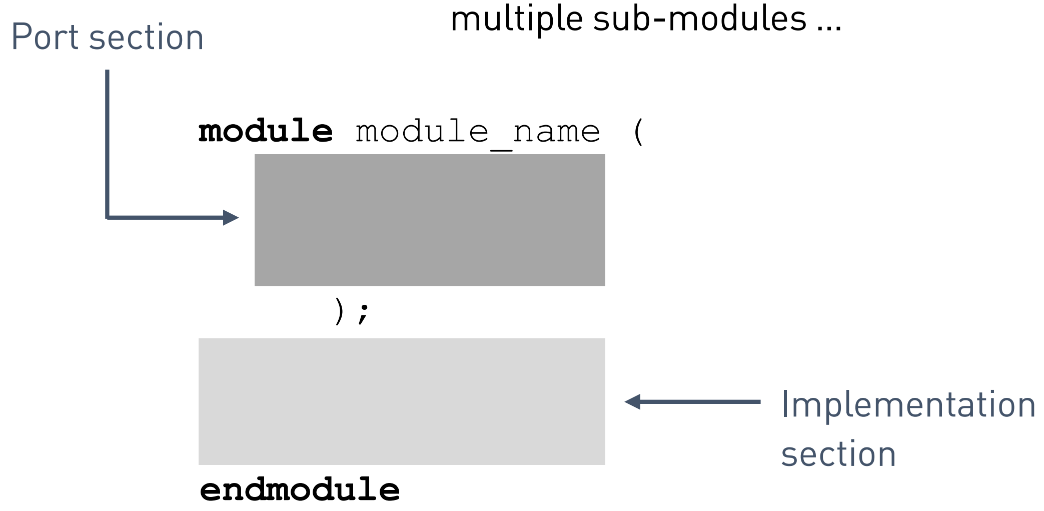


The Module – The basic building block in SystemVerilog

- 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 …



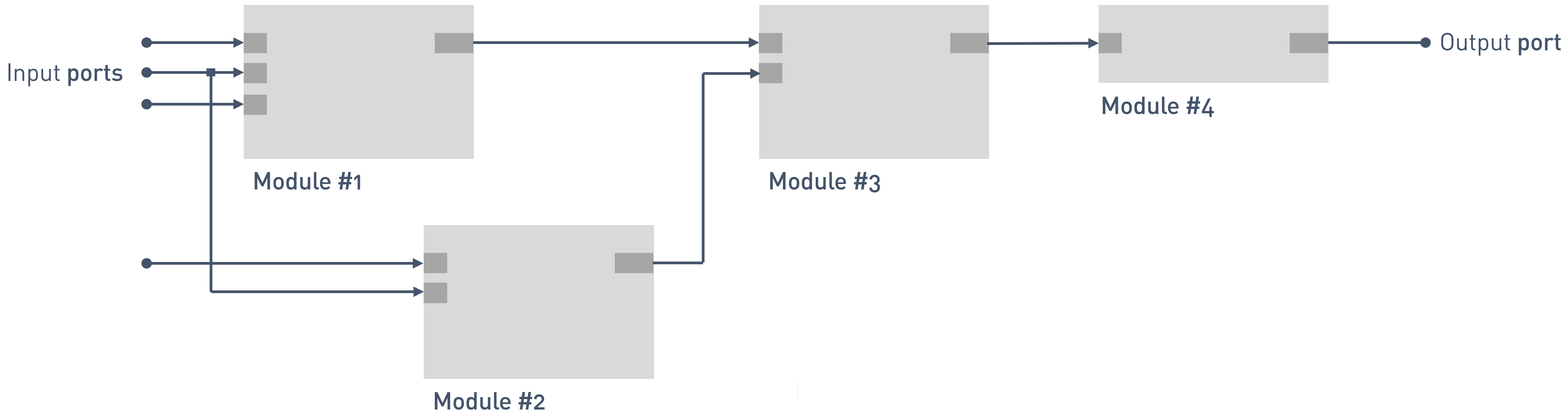


```
module module_name (

    );

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

- Verilog designs consist of multiple interconnected modules.



The Module – The basic building block in SystemVerilog

Module name, must be identical with the file name

1/1

```
module [module_name](
   input  [net_type] [range] port_identifier,
   inout  [net_type] [range] port_identifier,
   output [net_type] [range] port_identifier
   );
              logic          [3:0]        q


endmodule
```

Port or IO section

Implementation section

[SystemVerilog] Source Code Snippet

Notes

- Port type **input**: The module receives data from the outside using input ports.
- Port type **output**: The module sends data to the outside using output ports.
- Port type **inout**: The module can either sends or receive data through inout ports.

The Module – The basic building block in SystemVerilog

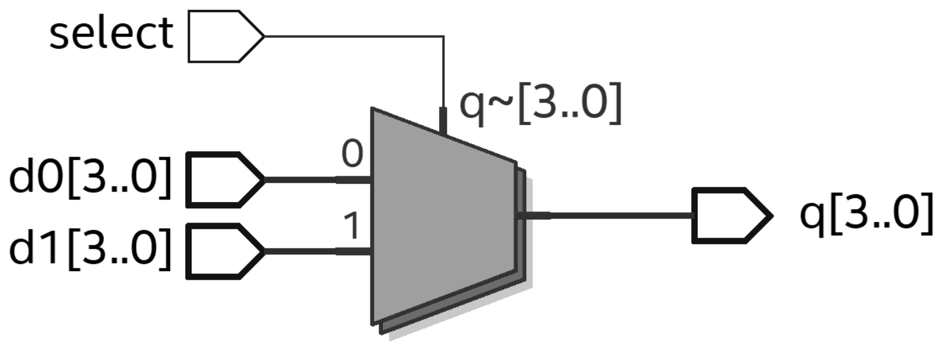
Port or IO section

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

Module name, must be identical with the file name

net typ: single bit

1/1

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

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

endmodule
```

port name

continuous assignment. another way for describing combinational logic

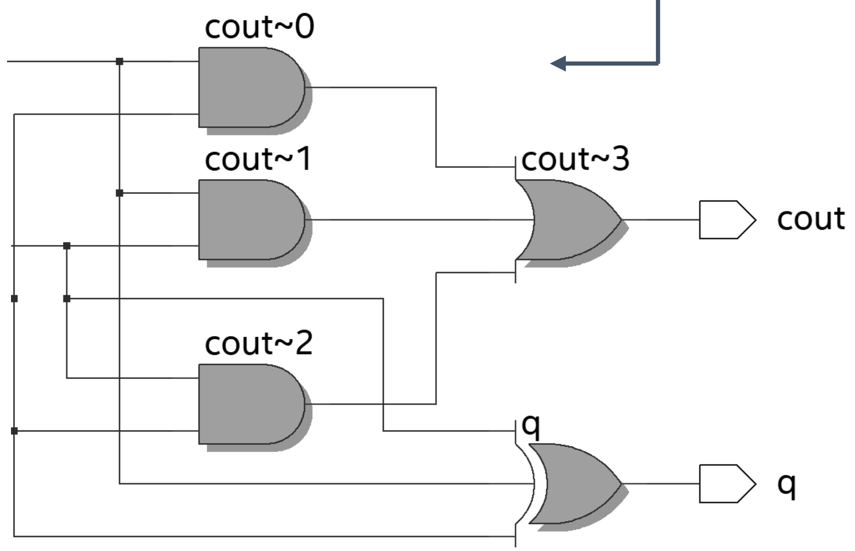
(do & d1) | (do ^ d1) & cin

logical expression

[SystemVerilog] Source Code: full_adder.sv

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

!



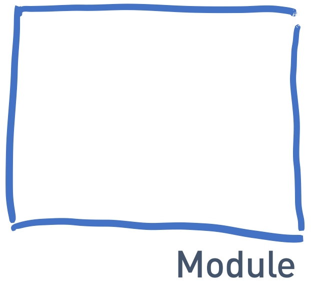
RTL netlist

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
```

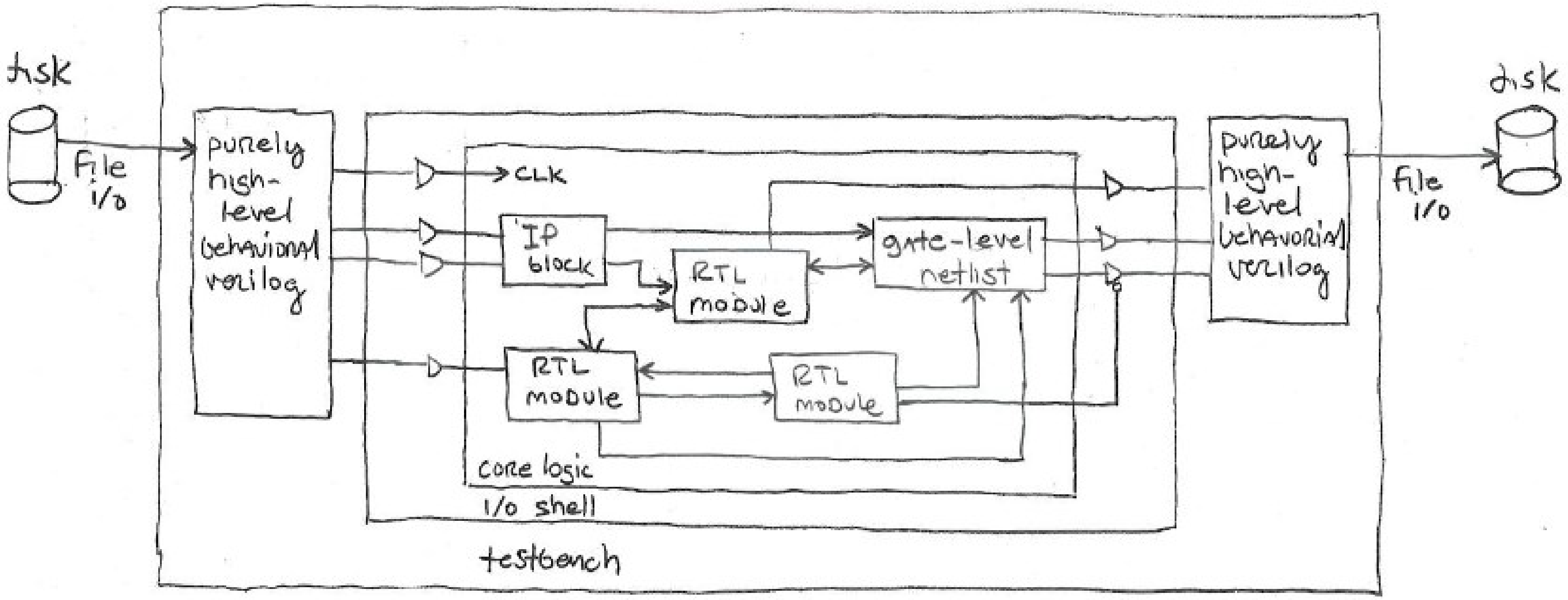


```
module add_sub_4bit(
   input  logic [3:0] d0, d1, input logic cin,
   output logic [3:0] q, output logic cout
   );
   function logic [4:0] add (input logic [3:0] a, b,
   input logic cin );
      logic [illegible]:0] s; logic c; c = cin;
      fo[illegible] = 0; i < 4; i++) begin
         [illegible]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

   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

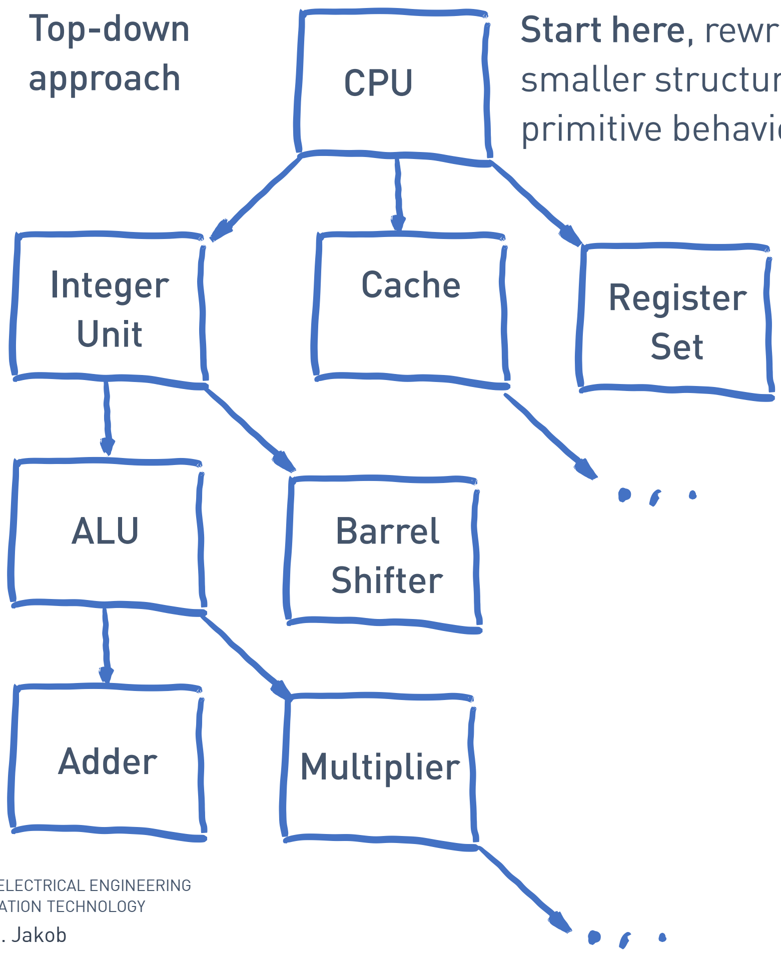
Structural description, describes the interconnect and usage of lower-level components

The Module – The basic building block in SystemVerilog

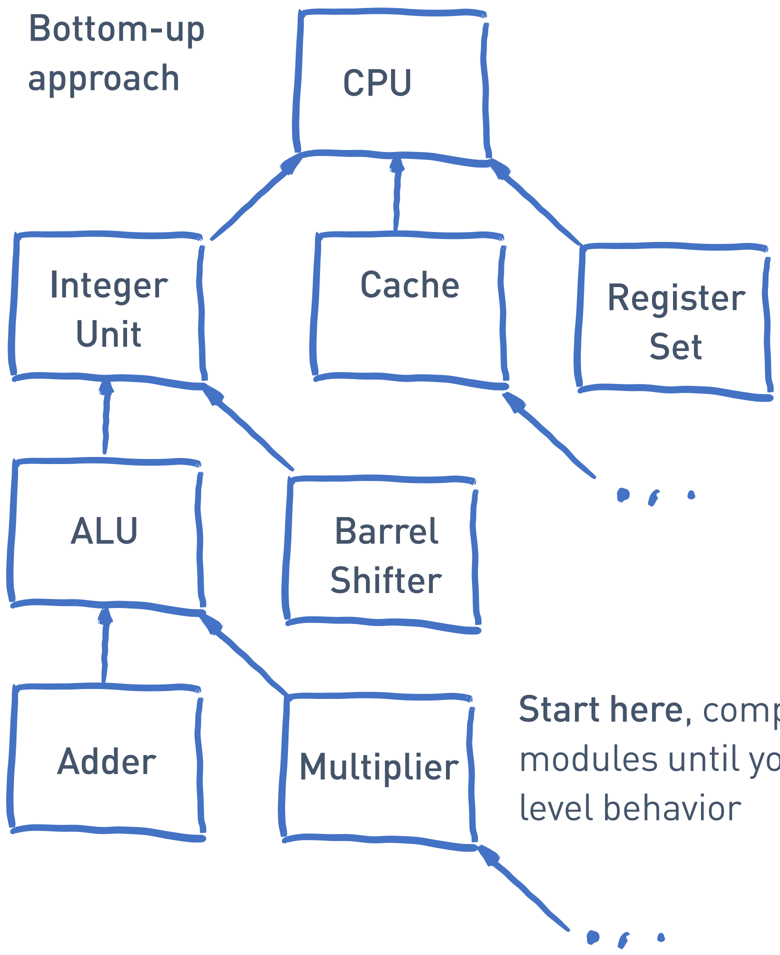


- A hierarchical design has a **top level module** and lower level ones. Lower level modules are instantiated within the higher level module. Lower level modules are connected together with wires.

Top-down and bottom-up design HDL descriptions



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



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

Module Instantiation

As stated before, more complex designs are built by integrating multiple modules in a hierarchical manner: Modules can be **instantiated** within other modules. The ports of theses **instances** are then connected with other signals in the parent module.

Name of the top-level/parent module

1/1

```
// ...
module my_design(
   input logic clk, input logic reset_n,
   input logic data,
   ...
   );

   module_name inst_name (port_connections);


endmodule
```

name of the **instance**

name of the **module to instantiate**

port connections describe how the instantiated module is interconnected with signals in the top-level module

[SystemVerilog] Source Code: my_design.sv

Module Instantiation

As stated before, more complex designs are built by integrating multiple modules in a hierarchical manner: Modules can be **instantiated** within other modules. The ports of theses **instances** are then connected with other signals in the parent module.

Name of the top-level/parent module

1/1

```
// ...
module my_design(
   input logic clk, input logic reset_n,
   input logic data,
   ...
   );

   module_name inst_name (port_connections);


endmodule
```

name of the **instance**

name of the **module to instantiate**

port connections describe how the instantiated module is interconnected with signals in the top-level module

[SystemVerilog] Source Code: my_design.sv

Notes

- We can instantiate previously designed modules or pre-defined gates (and, or, xor, nand, nor, xnor)

Module Instantiation – Using pre-defined Gates

As stated before, more complex designs are built by integrating multiple modules in a hierarchical manner: Modules can be **instantiated** within other modules. The ports of theses **instances** are then connected with other signals in the parent module.

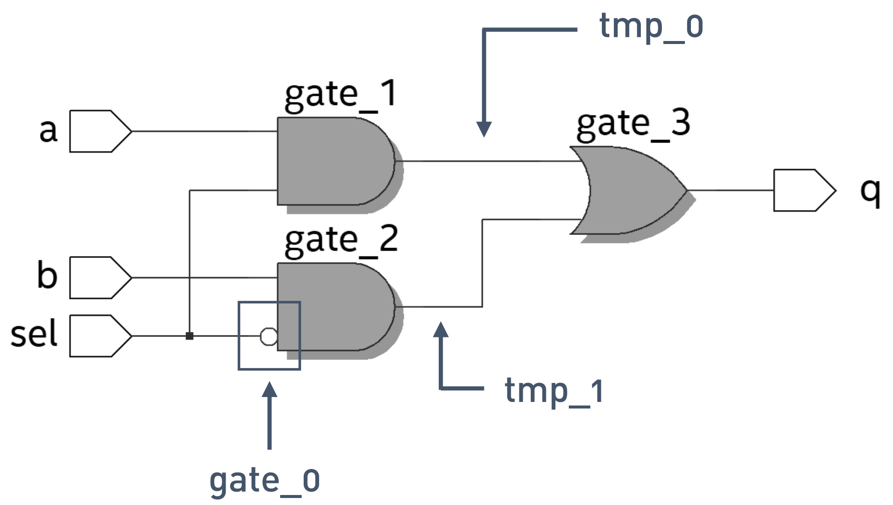
1/1

```
// ...
module mux_2_1_1bit(
   input logic a, b, sel,
   output logic q
   );

   logic tmp_0, tmp_1, not_sel;

   not gate_0(not_sel, sel);
   and gate_1(tmp_0, a, sel);
   and gate_2(tmp_1, b, not_sel);
   or  gate_3(q, tmp_0, tmp_1);

endmodule
```

intermediate wires used to interconnect the following logic gates.

port order: output, input(s)

[SystemVerilog] Source Code: mux_2_1_1bit.sv



Notes

- All pre-defined gates (and, or, xor, nand, nor, xnor) have fixed port order: output, input(s).
- For self-developed modules, you can define the port order. More on that on a later point in time.

Module Instantiation – Using pre-designed Modules

1/1

```
module my_and_gate(
   input logic a, b, output logic q
   );
   assign q = a & b;

endmodule
```

[SystemVerilog] Source Code: my_and_gate.sv

1/1

```
module my_top_level_design_0(
   input logic d_0, d_1, output logic q_0
   );
   my_and_gate inst_0(d_0, d_1, q_0);

endmodule
```

connection via **port order**

[SystemVerilog] Source Code: my_top_level_design_0.sv

Module Instantiation

Notes

- We can access the ports of the instantiated module **by order** or by name.

Module Instantiation – Using pre-designed Modules

1/1

```systemverilog
module my_and_gate(
   input logic a, b, output logic q
   );
   assign q = a & b;

endmodule
```

[SystemVerilog] Source Code: my_and_gate.sv

1/1

```systemverilog
module my_top_level_design_1(
   input logic d_0, d_1, output logic q_0
   );
   my_and_gate inst_0(.q(q_0), .a(d_0), .b(d_1));

endmodule
```

connection via **port name**

[SystemVerilog] Source Code: my_top_level_design_1.sv

Module Instantiation

Notes

- We can access the ports of the instantiated module by order or by **name**.

Module Instantiation

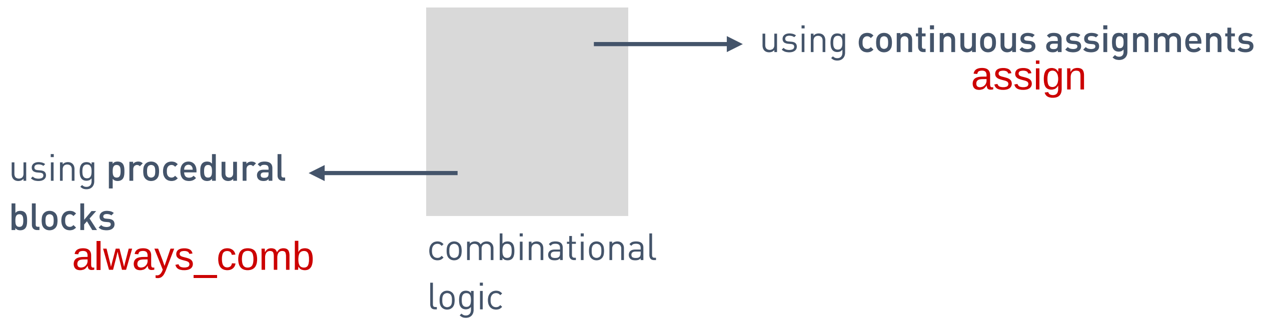
- More about interconnecting and parametrizing instantiated modules within the next lecture sessions.

Describing Combinational Logic using SystemVerilog

Introduction to SystemVerilog

Describing Combinational Logic using SystemVerilog

- There are two different ways to model combinational logic in SystemVerilog.



- Both are concurrent statements that may co-exist within a single module.
- In case you have a VHDL background:
  - Procedural blocks are the counterpart to processes in VHDL.
  - Continuous assignments are the SystemVerilog equivalents to signals in VHDL.

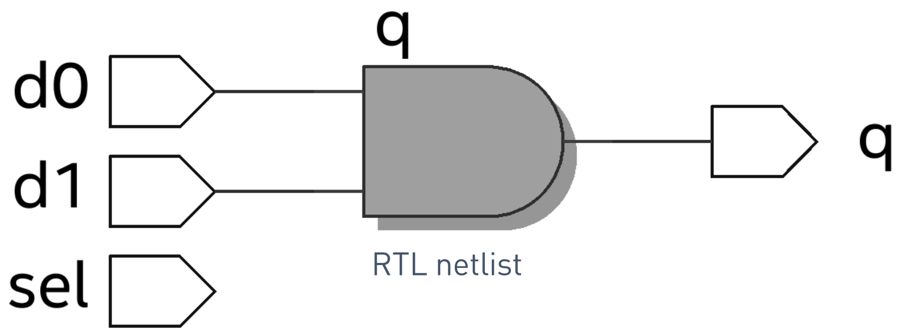
Describing Combinational Logic using SystemVerilog

1/1

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

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

endmodule
```

* Continuous assignment

[SystemVerilog] Source Code: and_2_1bit.sv



RTL netlist

Notes

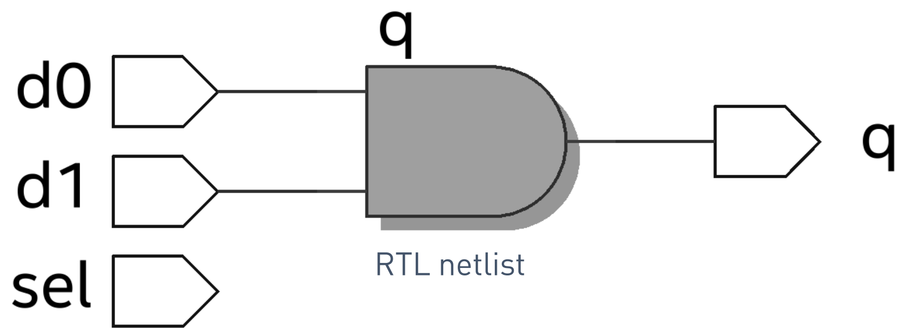
- The above example shows a so called continuous assignments (assign). These constructs are intended for modelling combinational logic.
- A continuous assignment drives a net similar to how a gate drives a net. The expression on the right hand side can be thought of as a combinatorial circuit that drives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

Describing Combinational Logic using SystemVerilog

1/1

```systemverilog
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



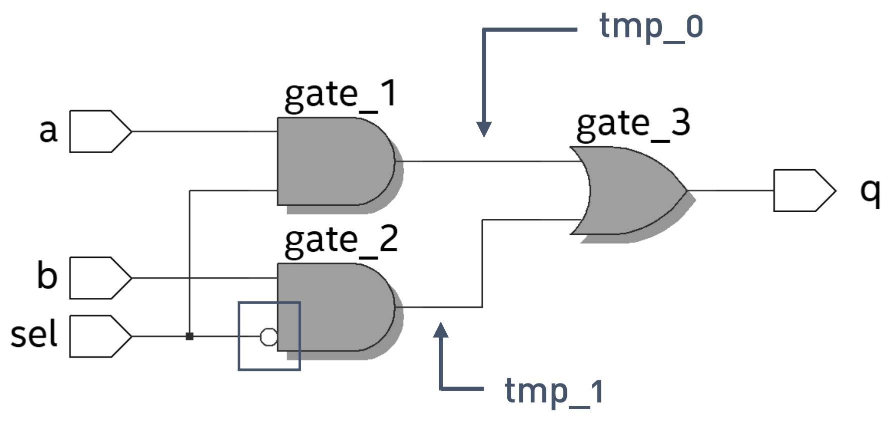
RTL netlist

Continuous assignment statements execute when a variable on the RHS changes. When the change occurs, the LHS is updated immediately. This behaviour pretty much reflects the behaviour of a real physical logic gate

Describing Combinational Logic using SystemVerilog

```
module mux_2_1_1bit(
   input logic a, b, sel,
   output logic q
   );

   logic tmp_0, tmp_1, not_sel;

   assign tmp_0 = a &   sel;
   assign tmp_1 = b & (~sel);
   assign q     = tmp_0 | tmp_1;

endmodule
```

[SystemVerilog] Source Code: mux_2_1_1bit.sv



Notes

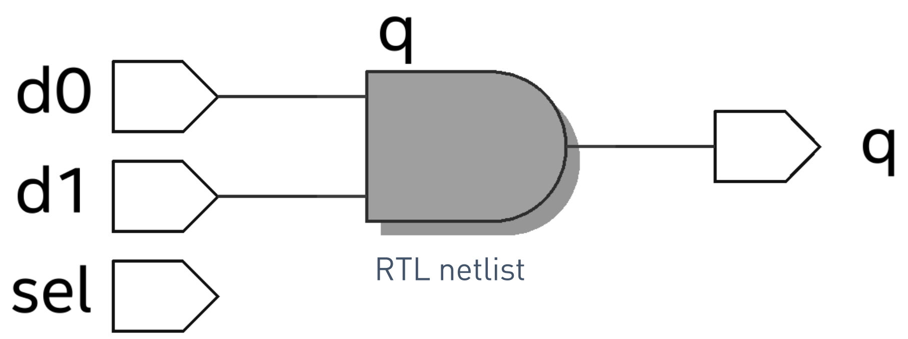
- Multiple continuous assignments within a single module.
- All continuous assignment statements are evaluated **concurrently**, the order is of no interest.

Describing Combinational Logic using SystemVerilog

1/1

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

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

endmodule
```

[SystemVerilog] Source Code: and_2_1bit.sv



RTL netlist

Notes

- An alternative way to model combinational logic is to use the **procedural always_comb** construct.

Describing Combinational Logic using SystemVerilog

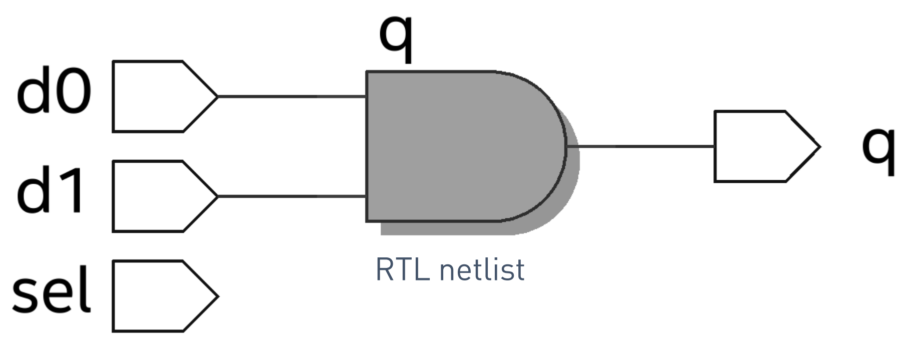
1/1

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

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

endmodule
```

blocking assignment operator

procedural block

[SystemVerilog] Source Code: and_2_1bit.sv



RTL netlist

Notes

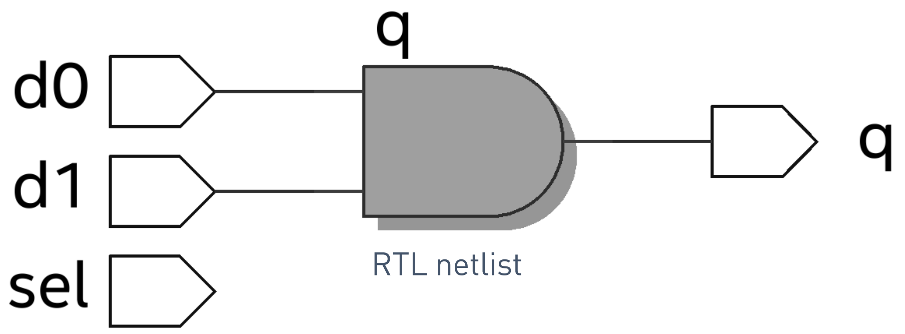
- **Procedural block**: An alternative way to model combinational logic is to use the procedural always_comb construct.
- The meaning of the blocking assignment operator will be discussed on a later stage of this lecture. It comes into play when multiple statements are grouped within a single always_comb block.

Describing Combinational Logic using SystemVerilog

1/1

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

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

endmodule
```

procedural block in SystemVerilog with implicit sensitivity list.



[SystemVerilog] Source Code: and_2_1bit.sv

Notes

- 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).

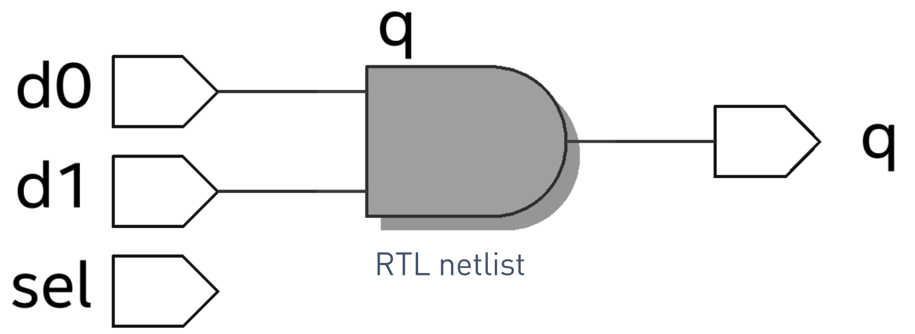
Describing Combinational Logic using SystemVerilog

1/1

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

   // 2-input AND gate
   always@(d0 or d1)
      q = d0 & d1;

endmodule
```

Verilog !

procedural block in **Verilog** with **explicit** sensitivity list.



[SystemVerilog] Source Code: and_2_1bit.v

Notes

- **Sensitivity list**: If an input signal that is specified in the sensitivity list changes its state, all related outputs are re-evaluated.
- The Verilog-2001 standard introduced the use of commas "," to separate items in the sensitivity list.

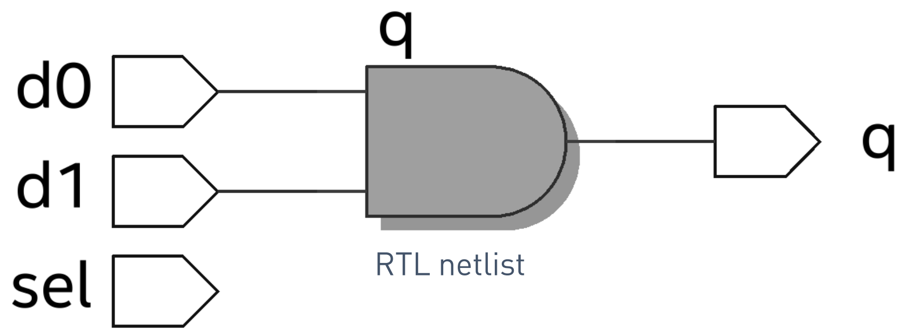
Describing Combinational Logic using SystemVerilog

1/1

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

   // 2-input AND gate
   always@*
      q = d0 & d1;

endmodule
```

Verilog !

procedural block in Verilog with implicit sensitivity list.



[SystemVerilog] Source Code: and_2_1bit.v

Notes

- Missing items in the sensitivity list has been a critical issue in Verilog and often led to undesirable and erroneous behavior.
- Therefore, the Verilog 2001 standard a wildcard construct (* - asterisk) that is pretty much the same as the always_comb statement in SystemVerilog.

IMP

Describing Combinational Logic using SystemVerilog

we use the **begin/end** keywords to group multiple statements

1/1

```
// ...
always_comb begin
   q_0 =  d_0;
   q_1 =  d_0 & d_1;
   q_2 = ~d_1;
   ...

   end
```

sequentially executed

blocking assignment operator

Inside the **always_comb** block, we describe the behaviour of combinational logic in a sequential, algorithmic way with **if**, **else**, **while** and **case** statements.

[SystemVerilog] Source Code Snippet

Notes

- We use blocking assignments within an procedural always_comb block to model the behavior of combinational logic gates.
- Blocking assignments evaluate the RHS of an expression and update the LHS immediately before any other instruction is executed.
- The order of statements within a procedural always_comb block matters!

IMP

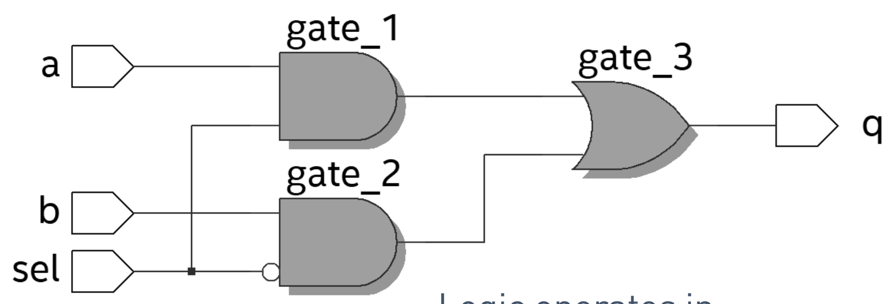
Describing Combinational Logic using SystemVerilog

1/1

```
    // ...
    always_comb begin
       ...
       end

    always_comb begin
       ...
       end

```

Parallel, concurrent execution

[SystemVerilog] Source Code Snippet



Logic operates in **parallel**

Notes

- **Logic gates operate in parallel**: The number of always blocks in a single module is not limited and all blocks are **executed concurrently**, without any predefined order ...
- All statements inside the begin/end section of a procedural always_comb statement **executed sequentially**.

Describing Combinational Logic using SystemVerilog

1/1

```
// ...
always_comb begin
   a = ...
   end

always_comb begin
   a = ...
   end

```

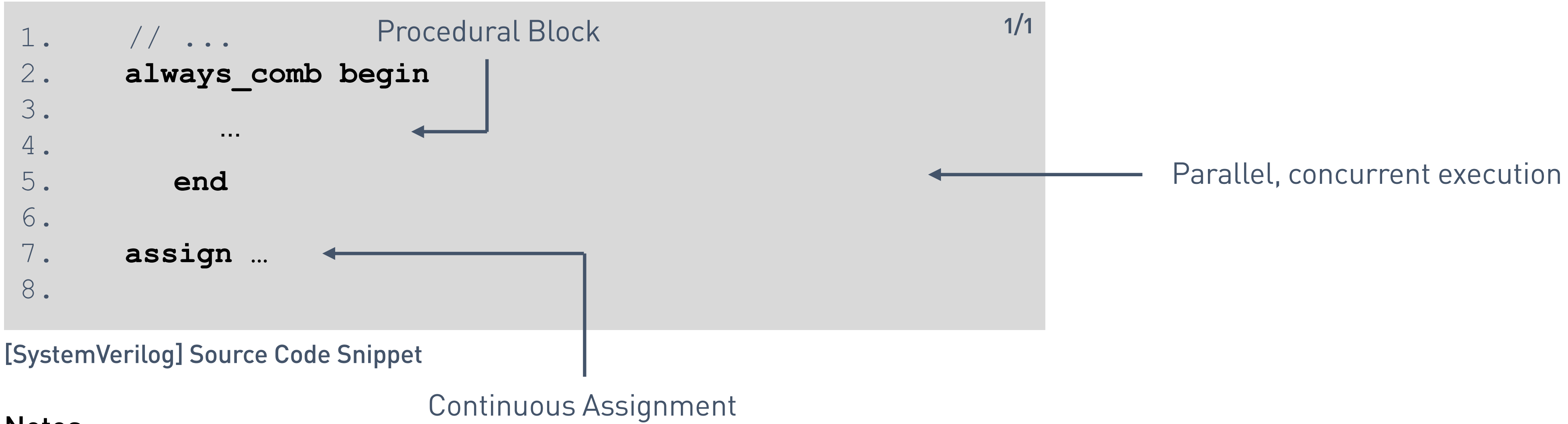
[SystemVerilog] Source Code Snippet

Multiple-driver error: Both blocks are driving the exact same signal ...

Notes

- Logic signals can only have a single driving source.
- Therefore, a variables on the LHS of a procedural block **cannot be assigned** within another procedural blocks.

Describing Combinational Logic using SystemVerilog



```
// ...
always_comb begin
      ...
  end

assign …
```

[SystemVerilog] Source Code Snippet

Notes

- SystemVerilog models concurrency with two basic constructs: Continuous assignments and procedural blocks.
- Both constructs can co-exist within a single module.

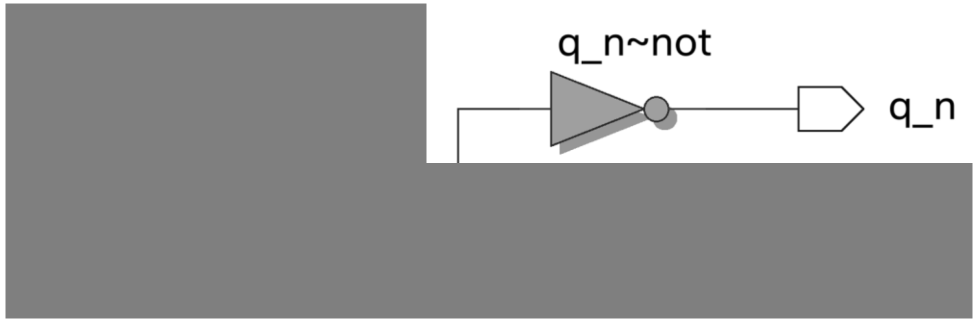
Describing Combinational Logic using SystemVerilog

```
module mux_2_1_1bit_n(
   input logic a, b, sel,
   output logic q, q_n
   );    …
[redacted]

   assign q_n = ~q;

endmodule
```

Procedural Block

Continuous Assignment

[SystemVerilog] Source Code: mux_2_1_1bit.sv



[redacted]

Conclusion so far ...

Introduction to SystemVerilog

Conclusion so far ...

- There are two different ways to model combinational logic in SystemVerilog:
  - Procedural blocks: always_comb statements
  - Continuous assignments: assign statements
- Both constructs can co-exists in a single module.
- They are so called concurrent statements which means that they are executed in parallel and might operate independently of each other.
- In case of continuous assignments, the RHS expression is continuously evaluated as a function of arbitrarily-changing inputs. The target of a continuous assignment is always a net driven by combinational logic.
- A procedural always_comb block using blocking assignments allows to model the behaviour of combinational logic in a sequential, algorithmic way using constructs such as if, else, while and case statements.

More about Describing Combinational Logic using SystemVerilog ...

Introduction to SystemVerilog

Describing Combinational Logic using SystemVerilog

- All of the following examples can be either implemented using **procedural blocks** or **continuous assignments.**

always_comb

assign

Bitwise Operators in SystemVerilog

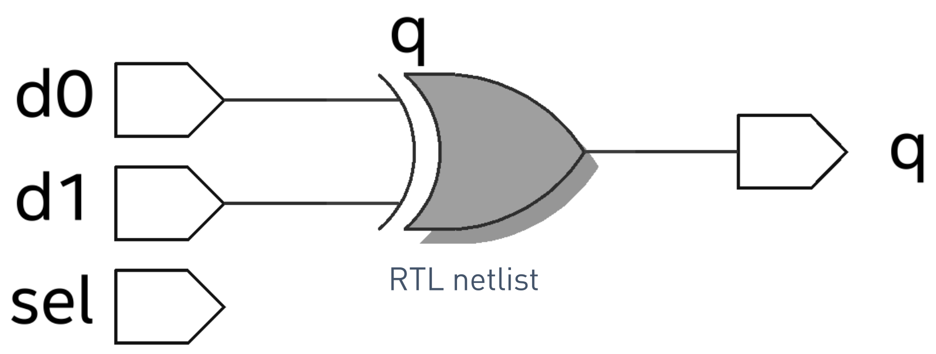
Introduction to SystemVerilog

Bitwise Operators in SystemVerilog

1/1

```systemverilog
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:	~
▪ Bit-wise NAND:	~&
▪ Bit-wise NOR:	~|
▪ Bit-wise XNOR:	~^

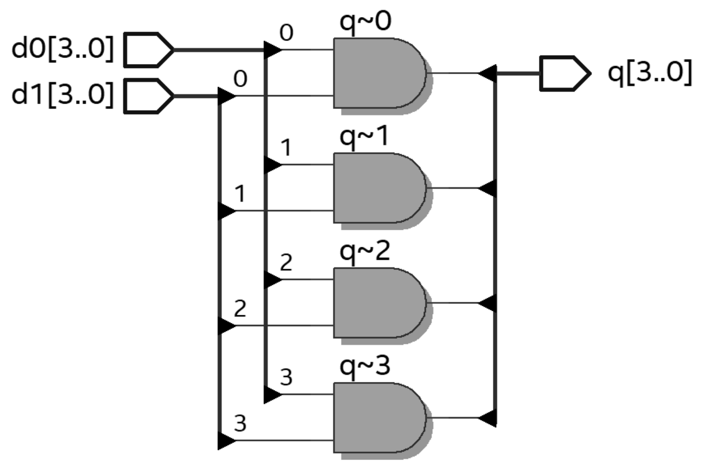
Bitwise Operators in SystemVerilog

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
```

Four bit wide vector, Little-Endian convention

[SystemVerilog] Source Code: and_2_4bit.sv



RTL netlist

Notes

- Bitwise operators operate on either **scalars** (single bit inputs) or **vectors** (multiple bit inputs).
- Little-Endian: The value of bit '0' is stored at the vector location with index '0' location

?

0001
0101 result in high signal

1010
0101 $\boxed{\text{result in low signal}}$

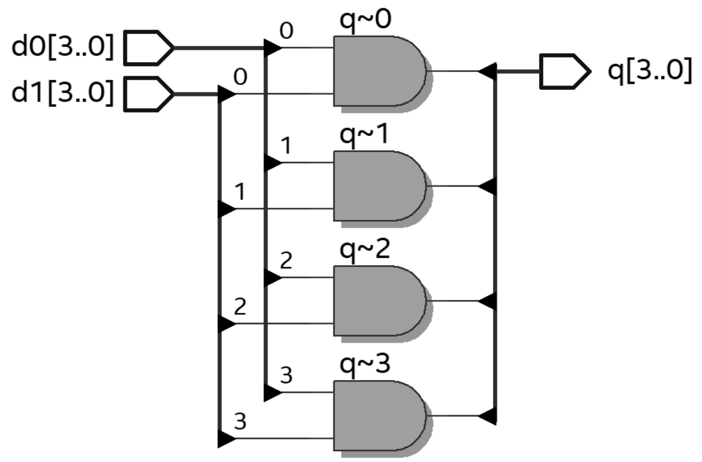
Bitwise Operators in SystemVerilog

1/1

```
module and_2_4bit(
   input  logic [3:0] d0, d1,
   output logic [3:0] q
   );
   // four 2-input AND gates
   always_comb
      q = d0 & d1;

endmodule
```

bit-by-bit operation on two inputs

[SystemVerilog] Source Code: and_2_4bit.sv



RTL netlist

Notes

- Bitwise operators operate on either **scalars** (single bit inputs) or **vectors** (multiple bit inputs).

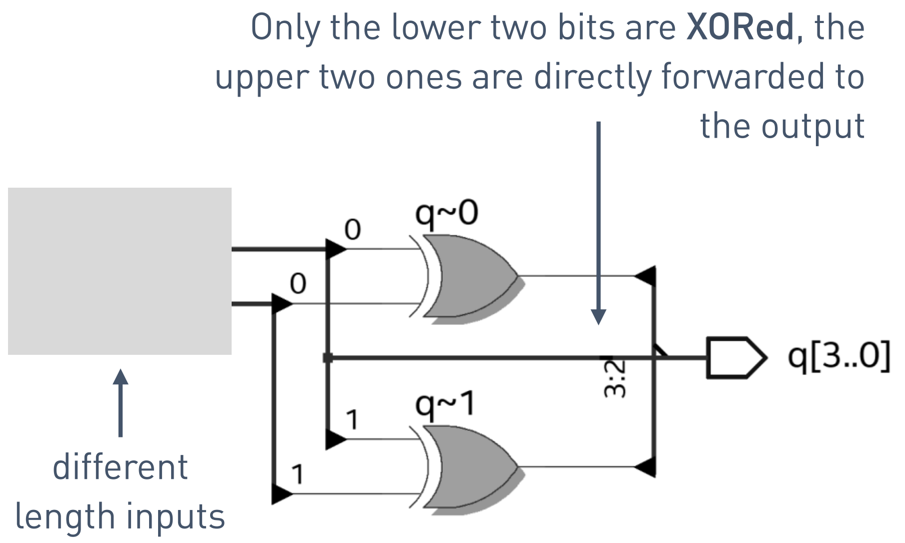
IMP

Bitwise Operators in SystemVerilog

1/1

```
module or_2_4bit(
   input  logic [3:0] d0, input logic [1:0] d1,
   output logic [3:0] q
   );

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

endmodule
```

[SystemVerilog] Source Code: or_2_4bit.sv


Only the lower two bits are **XORed**, the upper two ones are directly forwarded to the output
q~0
0
0
q~1
1
1
3:2
q[3..0]
different length inputs

Notes

- **It is not strictly necessary that both inputs have the same length**: If one input is not as long as the other, it will automatically **be left-extended with zeros** to match the length of the other input.

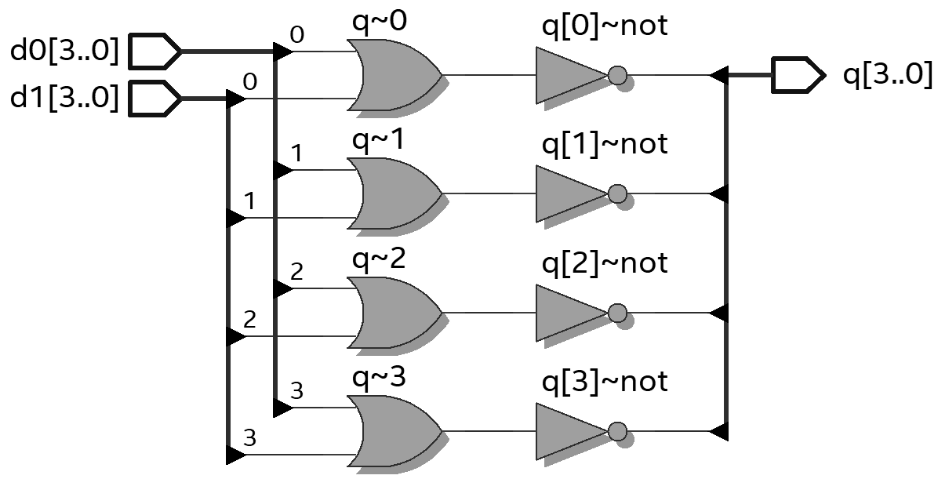
Bitwise Operators in SystemVerilog

NOR

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

Relational Operators in SystemVerilog

Introduction to SystemVerilog

Relational Operators in SystemVerilog

1/1

```
// greater than ...
q = d0 > d1;

// greather than or equal to ...
q = d0 >= d1;

// less than ...
q = d0 < d1;

// less than or equal to ...
q = d0 <= d1;

// equal to ...
q = d0 == d1;

// not equal to
q = d0 != d1;
```

Relational operators evaluate to a 1-bit value, representing true and false respectively.

[SystemVerilog] Source Code Snippet

Relational operators are used to compare the value of two different variables in SystemVerilog.

Logical Operators in SystemVerilog

Introduction to SystemVerilog

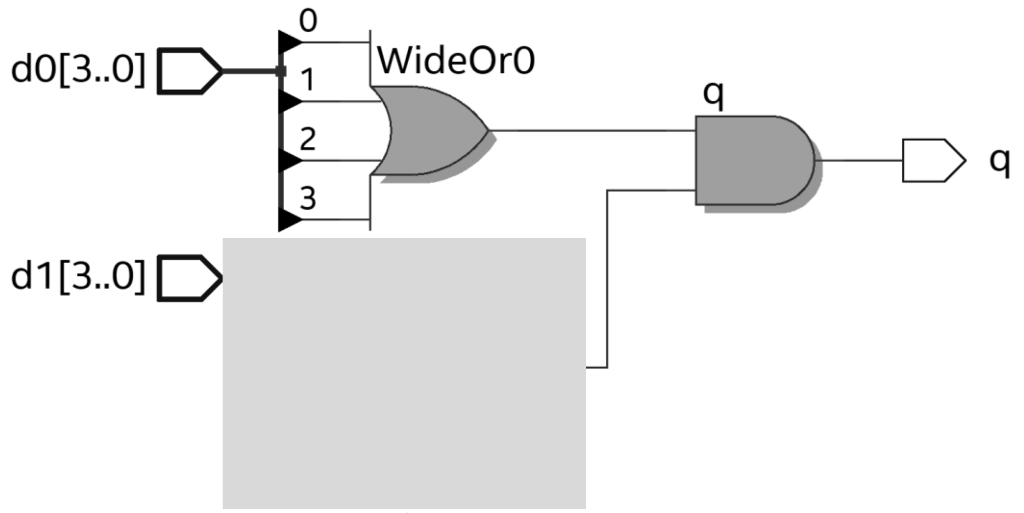
IMP

Logical Operators in SystemVerilog

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



the output is high if at least one input is high

Notes

- Logical operators evaluate to a 1-bit value.
- All operands not equal to zero are equivalent to one.
- SystemVerilog Logical Operators:

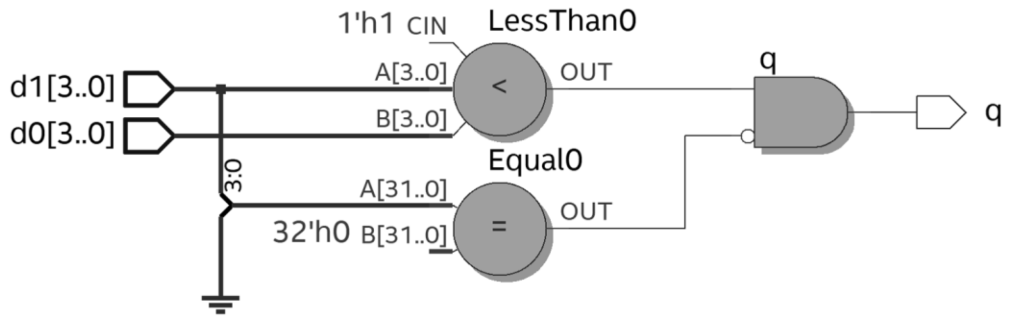
Logical AND:	&&
Logical OR:	||
Logical NOT:	!

Logical Operators in SystemVerilog

1/1

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

  assign q = (d0 >= d1) && (d1 != 0);

endmodule
```

[SystemVerilog] Source Code: sys_proc.sv



Notes

- Rather than using these operators to model gates we use them to **combine relational operators.**

Multiplexers in SystemVerilog

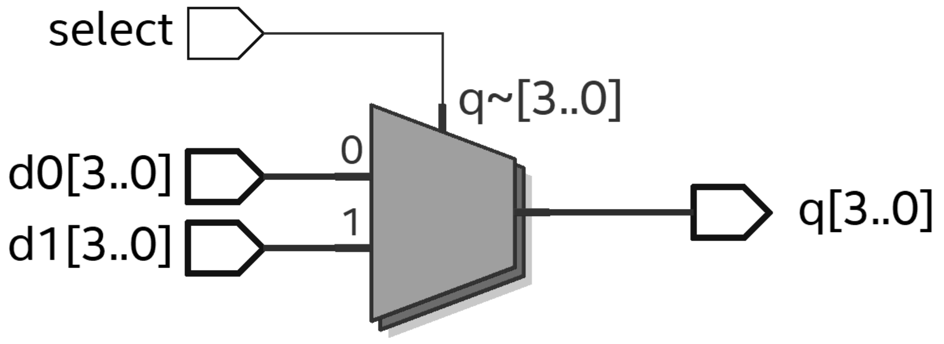
Introduction to SystemVerilog

Multiplexers - Conditional assignment

1/1

```
module mux_2_4bit(
   input  logic [3:0] d0, d1, input logic select,
   output logic [3:0] q
   );

   assign q = (select == 1) ? d1 : d0;

endmodule
```

[SystemVerilog] Source Code: mux_2_4bit.sv



RTL netlist

Notes

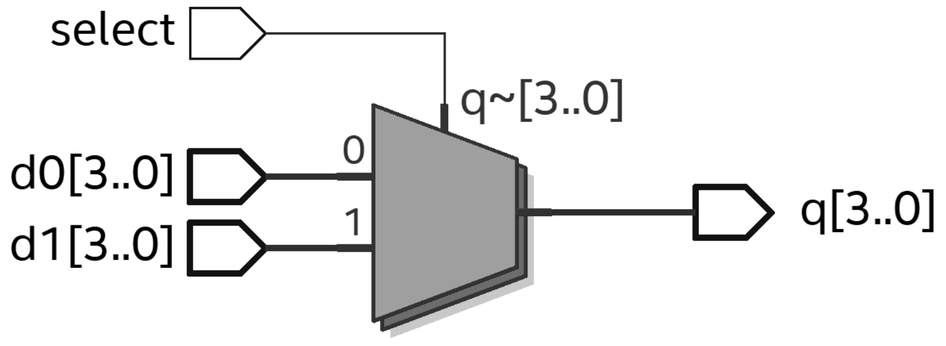
- The ? expression is also called a **ternary operator** because it operates on three inputs: select, d0 and d1.
- For more complex structures including several sequentially placed multiplexers, use the always_comb construct in conjunction with if-else or case statements ...

Multiplexers - Conditional assignment

1/1

```systemverilog
module mux_2_4bit(
   input  logic [3:0] d0, d1, input logic select,
   output logic [3:0] q
   );

   assign q = select ? d1 : d0;

endmodule
```

[SystemVerilog] Source Code: mux_2_4bit.sv



RTL netlist

A slightly more compact way of specifying the select signal. This is possible if select is a single bit signal ...

Notes

- The same functionality as before ...

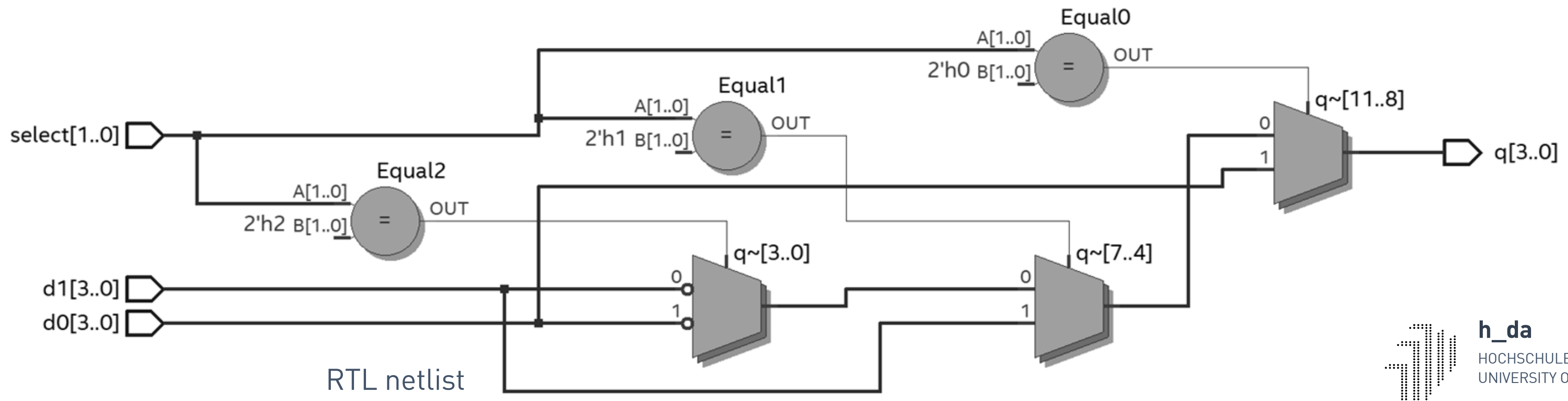
Multiplexers - Conditional assignment

Two input mux with 2-bit select signal ...

1/1

```
module multistage_mux_2_4bit(
   input  logic [3:0] d0, d1, input logic [1:0] select,
   output logic [3:0] q
   );

   assign q = (select == 2'b00) ? d0 : (select == 2'b01) ? d1 : (select == 2'b10) ? ~d0 : ~d1;

endmodule
```

[SystemVerilog] Source Code: multistage_mux_2_4bit.sv



RTL netlist

Multiplexers - Conditional assignment

```systemverilog
module multistage_mux_2_4bit(
   input  logic [3:0] d0, d1, input logic [1:0] select,
   output logic [3:0] q
   );

   assign q = (select == 2'b00) ?  d0 :
              (select == 2'b01) ?  d1 :
              (select == 2'b10) ? ~d0 : ~d1;
endmodule
```

1/1

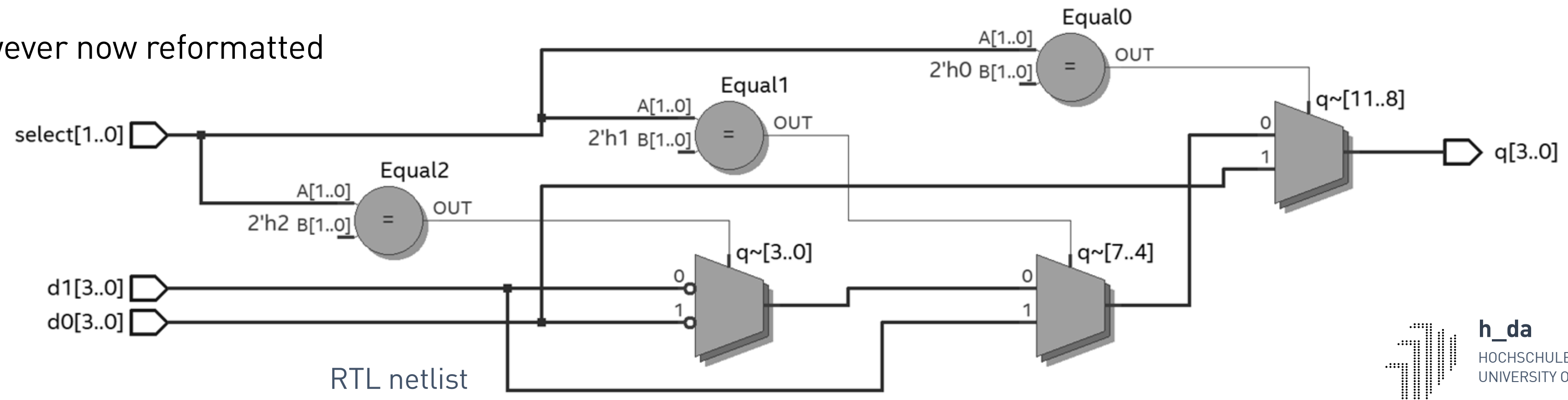
Conditions

Respective output selections

[SystemVerilog] Source Code: multistage_mux_2_4bit.sv

Notes

- The same module as before, however now reformatted to improve the readability.



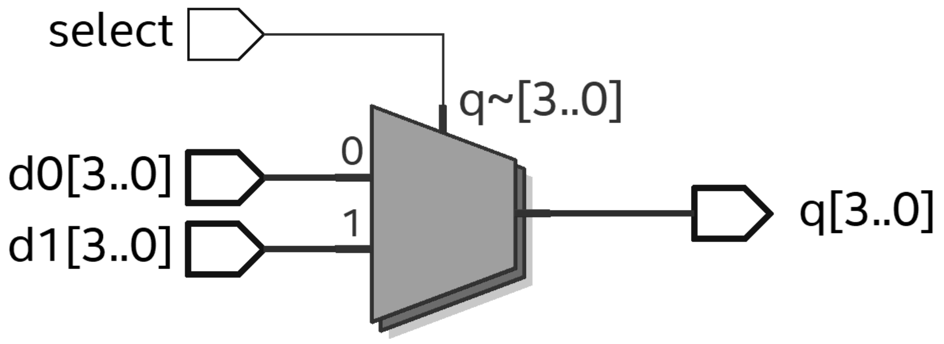
RTL netlist

Multiplexers – always_comb

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'b1)
         q = d1;
      else
         q = d0;
endmodule
```

[SystemVerilog] Source Code: mux_2_4bit_comb.sv



RTL netlist

Notes

- The same as before, but now using a procedural always_comb block in conjunction with an if-else statement.

Multiplexers – always_comb

1/1

```
module multistage_mux_2_4bit_comb(
   input  logic [3:0] d0, d1, input logic [1:0] select,
   output logic [3:0] q
   );

   always_comb
      if(select == 2'b00)
         q = d0;
      else if(select == 2'b01)
         q = d1;
      else if(select == 2'b10)
         q = ~d0;
      else
         q = ~d1;
endmodule
```

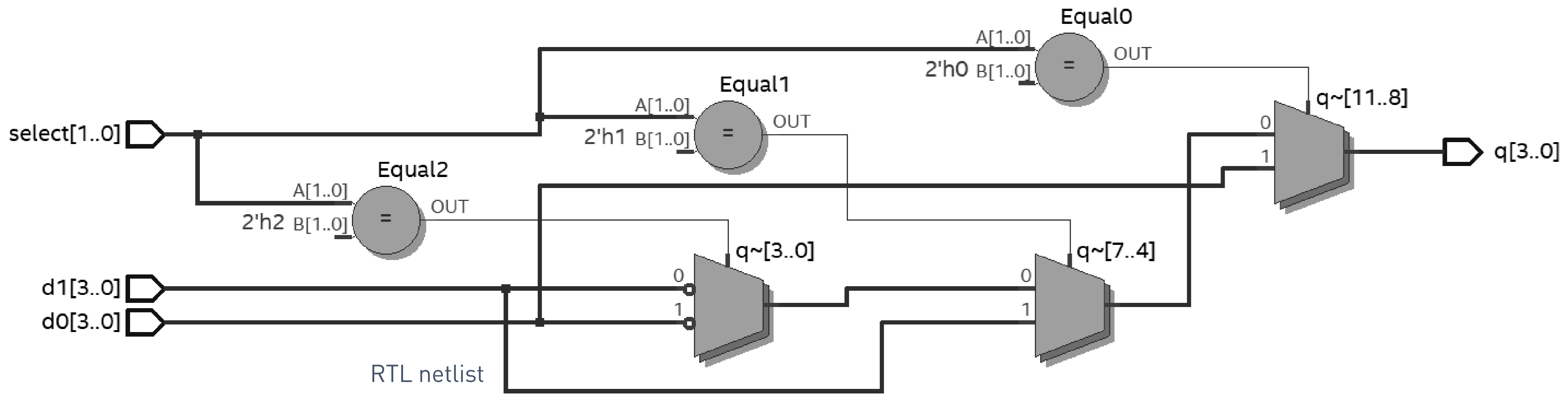
2-bit select signal

[SystemVerilog] Source Code: multistage_mux_2_4bit_comb.sv

Notes

- The same as before (`multistage_mux_2_4bit.sv`), but now with a if/else-if statement within the always_comb block.

Multiplexers – always_comb



Notes

- The synthesized netlist is the same as before …

Comparators in SystemVerilog

Introduction to SystemVerilog

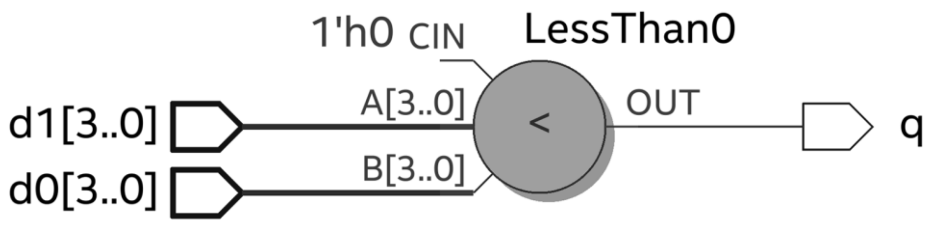
Comparators - Conditional assignment

1/1

```
module comp_gt_4bit(
   input  logic [3:0] d0, d1,
   output logic q
   );

   assign q = (d0 > d1) ? 1’b1 : 1’b0;

endmodule
```

[SystemVerilog] Source Code: comp_gt_4bit.sv

Here, we use a conditional assignment to implement a greater-than comparator



RTL netlist

Notes

- Output q is set to one if the input vector d0 is larger than d1
- All other relational operators are possible as well:

a greater than b	a > b
a greater than or equal to	a >= b
a less than b	a < b
a less than or equal to b	a <= b

Nested if-else statements might lead to priority logic ...

Introduction to SystemVerilog

Encoder – nested if statements

1/1

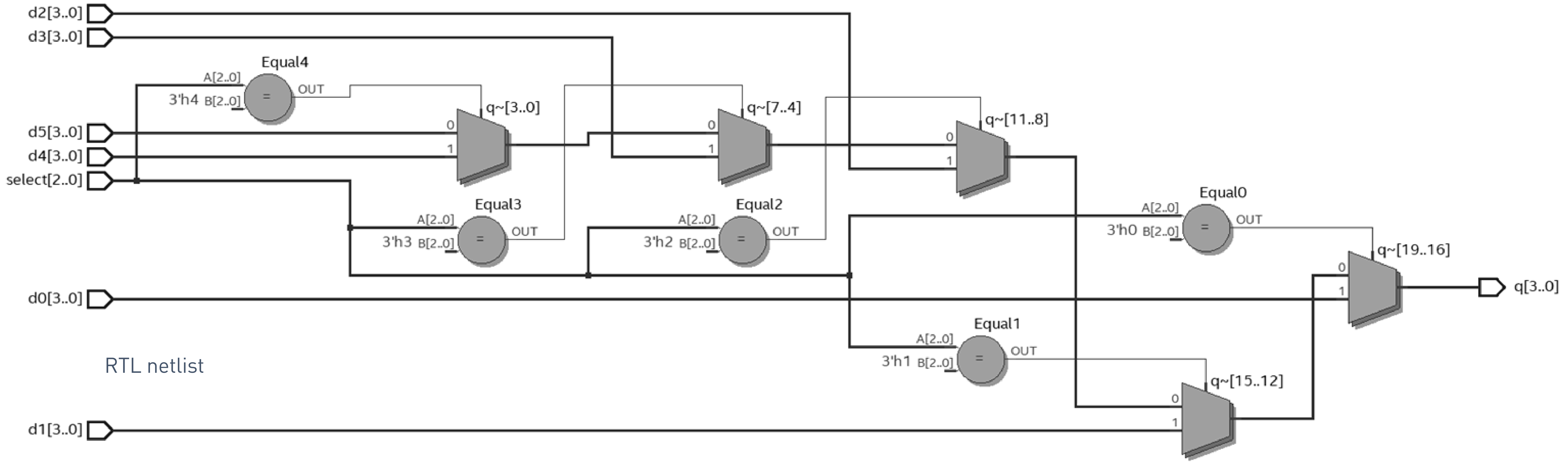
```
module encoder_6_4_bit(
   input logic [3:0] d0, d1, d2, d3, d4, d5,
   input logic [2:0] select, output logic [3:0] q
   );
   always_comb
      if(select == 3'd0)
         q = d0;
      else if(select == 3'd1)
         q = d1;
      else if(select == 3'd2)
         q = d2;
      else if(select == 3'd3)
         q = d3;
      else if(select == 3'd4)
         q = d4;
      else
         q = d5;
endmodule
```

[SystemVerilog] Source Code: encoder_6_4_bit.sv

Notes

- Consider the example of an encoder implemented using a set of nested if-statements.
- The task of the encoder is to compress multiple binary inputs into a smaller number of outputs.
- In the present example, we've got a 6-to-1 selection.
- The drawback of using nested if-statements becomes visible by observing the synthesis netlist generated. ?

Encoder – nested if statements



Notes

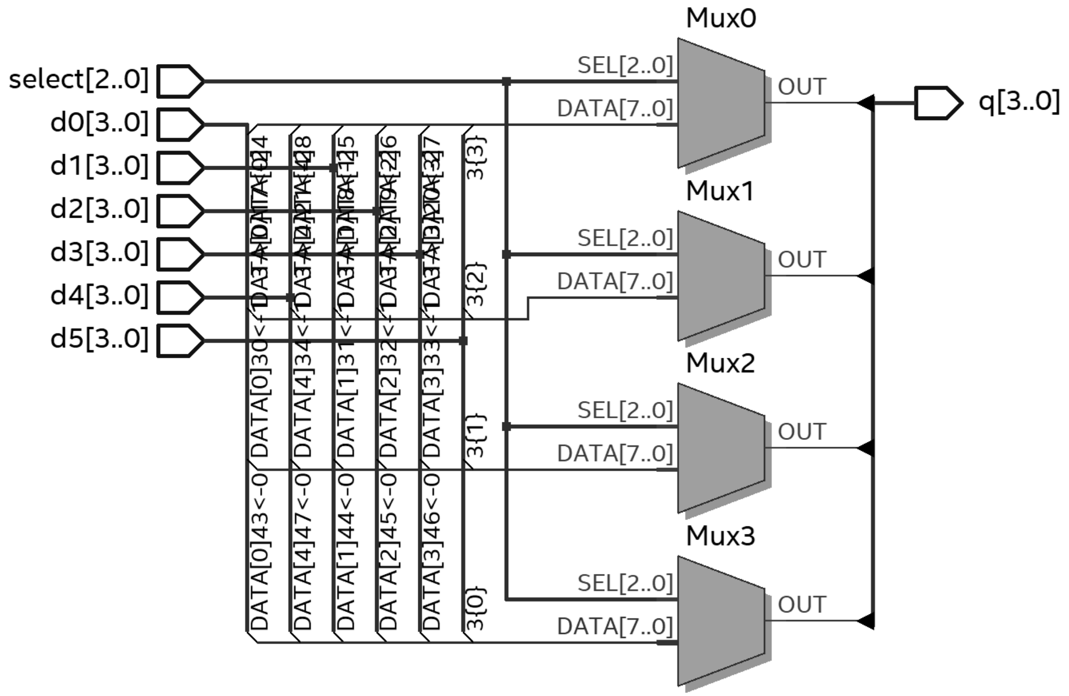
- Nested if-statements might lead to priority logic, which adds both extra logic gates and longer timing paths to the logic.
- The cascaded logic affects the performance of the circuit due to an increase of area and delay. Observe the path of input d5 to the output q: It passes five logic stages (muxes) while input d0 only traverses a single one …

Encoder – case statements

1/1

```
module encoder_6_4_bit_case(
   input logic [3:0] d0, d1, d2, d3, d4, d5,
   input logic [2:0] select, output logic [3:0] q
   );
   always_comb
      case(select)
         0: q = d0;
         1: q = d1;
         2: q = d2;
         3: q = d3;
         4: q = d4;
         default:
            q = d5;
      endcase
endmodule
```

[SystemVerilog] Source Code: encoder_5_4_bit_case.sv



RTL netlist

Notes

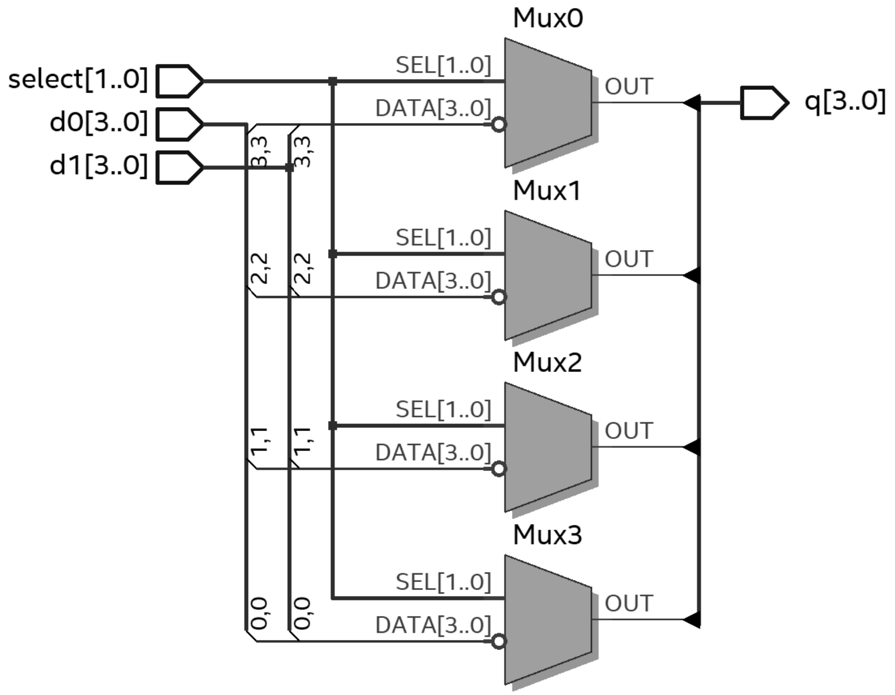
- If the use-case or application doesn’t require an explicit priority scheme, it is possible to replace the nested if-statements by a case statement based implementation. All cases are now matched in parallel.

Multiplexers – always_comb

1/1

```
module multistage_mux_2_4bit_comb(
   input  logic [3:0] d0, d1, input logic [1:0] select,
   output logic [3:0] q
   );

   always_comb
      case(select)
         0: q =  d0;
         1: q =  d1;
         2: q = ~d0;
         3: q = ~d1;
      endcase

endmodule
```

[SystemVerilog] Source Code: multistage_mux_2_4bit_comb.sv



RTL netlist

Notes

- The same as before, but now using an case statement instead of nested if-statements. The synthesized netlist is different right now, the functionality is however still the same as before …

Selecting arithmetic operations

1/1

```
module multistage_mux_4_4bit_comb(
   input  logic [3:0] d0, d1, input logic [2:0] select,
   output logic [3:0] q_0, q_1
   );
   always_comb
      case(select)
          0: begin q_0 = d0; q_1 = d1; end
          1: begin
              q_0 = d0 + d1;
              q_1 = d0 - d1;
          end
          2,3,4 : begin q_0 = ~d1; q_1 = ~d0; end
          default: begin
              q_0 = 4'b0000;
              q_1 = 4'b0000;
          end
      endcase
endmodule
```

simple branch

group multiple statements with begin/end

match multiple select values

default outputs for all other select values

[SystemVerilog] Source Code: multistage_mux_4_4bit_comb.sv

Select arithmetic operations

1/1

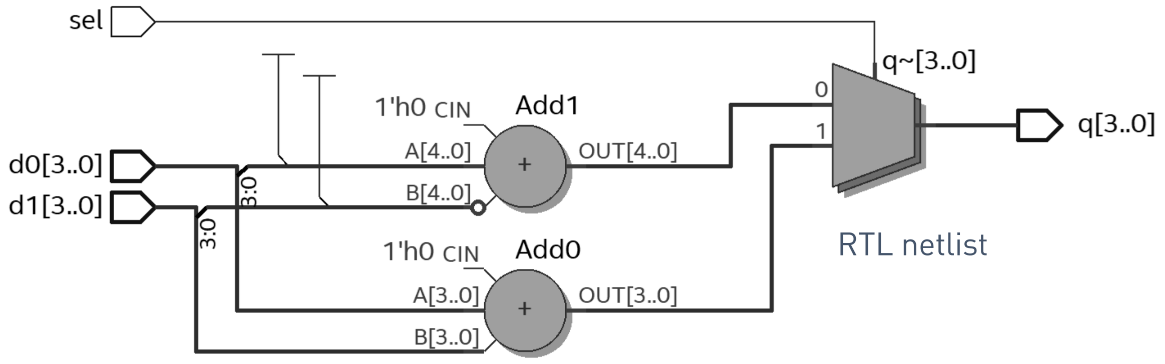
```
module op2_sel(
   input  logic [3:0] d0, d1, input logic sel,
   output logic [3:0] q
   );

   always_comb
      if(sel)
         q = d0 + d1;
      else
         q = d0 - d1;

endmodule
```

[SystemVerilog] Source Code: op2_sel.sv

Notes

?

- Both operations are computed in parallel.
- A multiplexer is used to select the sum or the difference computed.

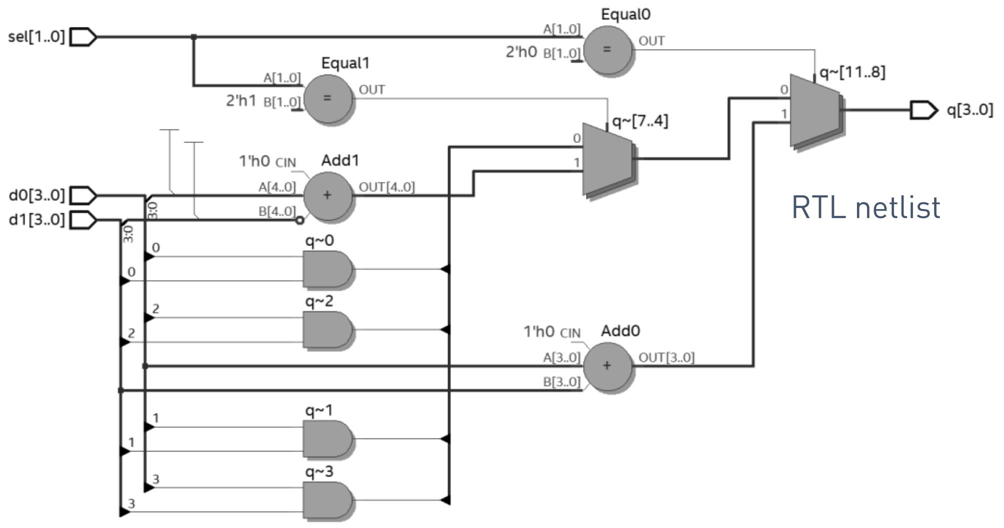


RTL netlist

Select arithmetic operations

1/1

```systemverilog
module op3_sel(
   input  logic [3:0] d0, d1, input logic [1:0] sel,
   output logic [3:0] q
   );

   always_comb
      if(sel == 2’b00)
         q = d0 + d1;
      else if(sel == 2’b01)
         q = d0 – d1;
      else
         q = d0 & d1;

endmodule
```



RTL netlist

[SystemVerilog] Source Code: op3_sel.sv

Notes

- Again, all three operations are computed in parallel. Two (cascaded) multiplexers implement priority, i.e. the all zero state for sel is prioritized over non-zero states.

Select arithmetic operations

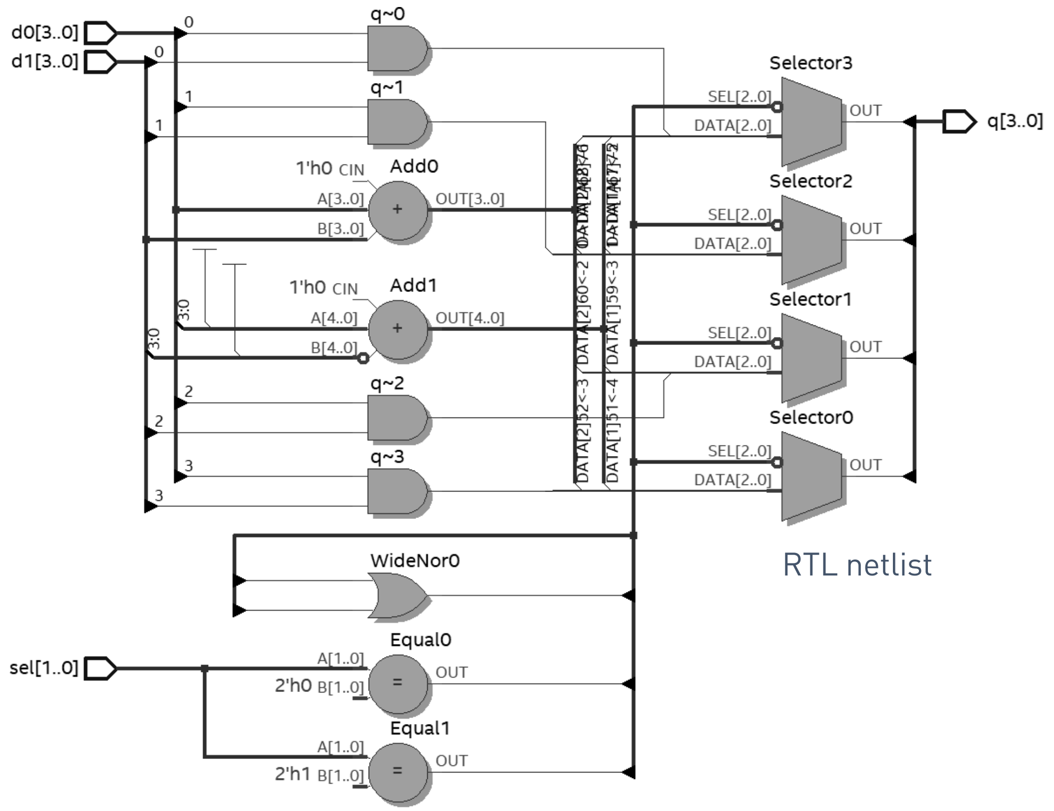
1/1

```
module op3_sel_case(
   input  logic [3:0] d0, d1, input logic [1:0] sel,
   output logic [3:0] q
   );

   always_comb
      case(sel)
         0: q = d0 + d1;
         1: q = d0 - d1;
         default:
            q = d0 & d1;
      endcase
 endmodule
```

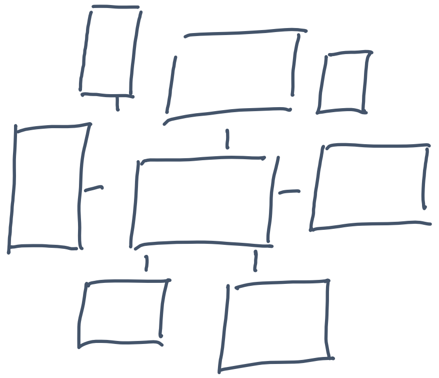
[SystemVerilog] Source Code: op3_sel_case.sv

Notes

- All operations are computed in parallel without any priority logic implemented.



RTL netlist



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.