Introduction to SystemVerilog HDL design

Today's Agenda

Intended topics for today's session

- Learning by example – A first taste: A series of basic 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 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.

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 #3

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

(PIPO) Registers in SystemVerilog

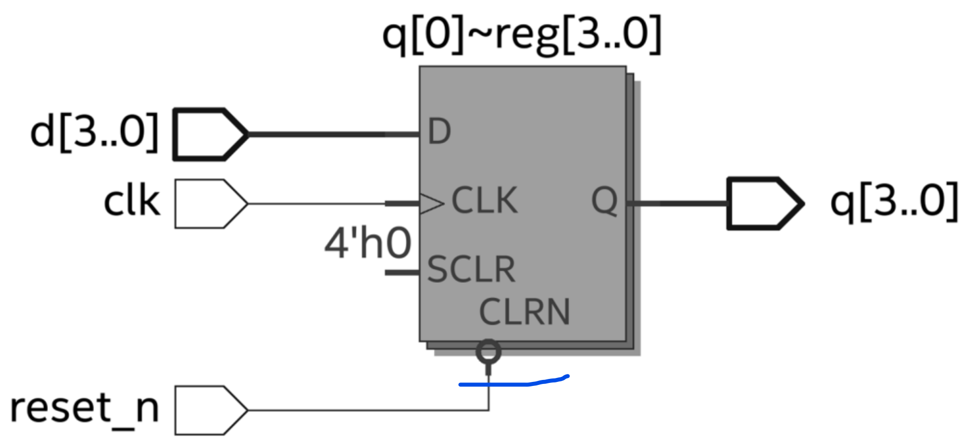
Introduction to SystemVerilog

Asynchronously resettable Register

1/1

```
module dff_4bit_load_areset(
   input  logic [3:0] d, clk, reset_n,
   output logic [3:0] q
   );
   always_ff@(posedge clk, negedge reset_n)
      if(reset_n == 0)
         q <= 0;
      else
         q <= d;
endmodule
```

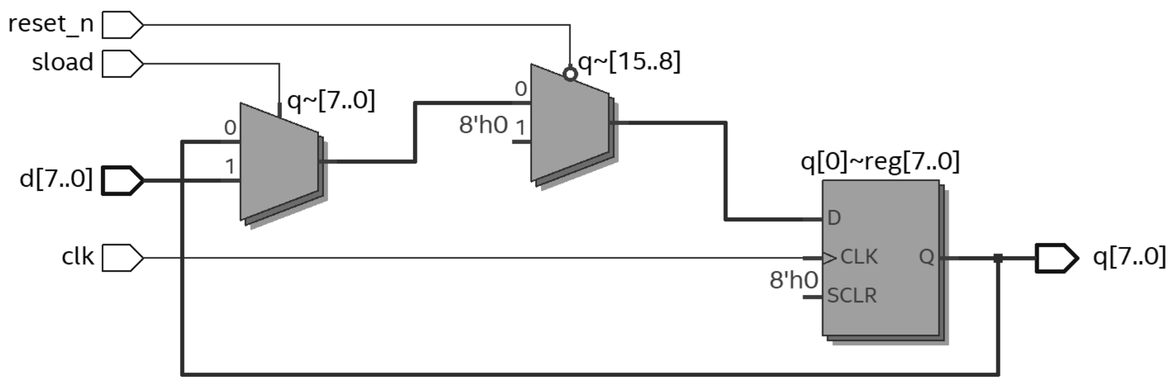
[SystemVerilog] Source Code: dff_4bit_load_areset.sv



Synchronously resettable Register with synchronous load

1/1

```systemverilog
module reg_pipo_8bit_sreset_sload(
   input logic [7:0] d, input logic clk, reset_n,
   sload, output logic [7:0] q
   );

   always_ff@(posedge clk)
      if(reset_n == 0)
         q <= 0;
      else if(sload == 1)
         q <= d;
      else
         q <= q;

endmodule
```



[SystemVerilog] Source Code: reg_pipo_8bit_sreset_sload.sv

Notes

- PIPO register stands for parallel-in parallel-out register, thus we've got 8 flip-flops which are loaded and read in parallel.

MCU Peripheral Configuration and Status Registers

1/1

```
logic [31:0] reg_0 = 0;
// timer0 peripheral configuration register placed
// address 0x40020014
always_ff@(posedge clk)
   if(reset_n == 0)
        reg_0 <= 0;
      else if(wren)
        reg_0 <= data;


assign TIMER_0_ENABLE     = reg_0[0];
assign TIMER_0_OVF_EN     = reg_0[1];
assign TIMER_0_OVF_IRQ_EN = reg_0[2];
```

[SystemVerilog] Source Code: timer0_config_reg.sv

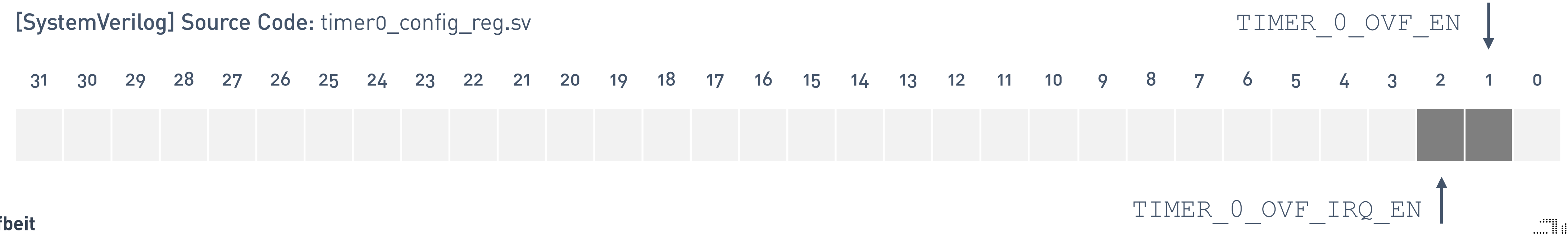
SW write access

Notes

- We will get back to this structure when we are designing status and control registers for microcontroller peripherals ...

```
*((unsigned int *) 0x40020014) |= 1 << 1;
*((unsigned int *) 0x40020014) |= 1 << 2;
...
```

You are defining the register layout as well as the corresponding meaning/functionality of the respective register bits.



(SISO) Registers in SystemVerilog

Introduction to SystemVerilog

Synchronously resettable Shift-Register, serial in, serial out

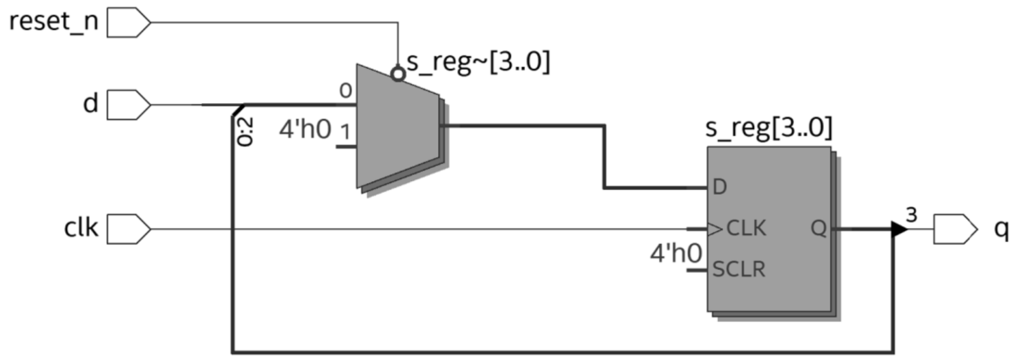
1/1

```
module shift_reg_4bit_areset(
   input  logic d, clk, reset_n,
   output logic q
   );
   logic [3:0] s_reg = 0;

   always_ff@(posedge clk)
      if(reset_n == 0)
              s_reg <= 0;
      else
              s_reg <= {s_reg[2:0],d};

   assign q = s_reg[3];

 endmodule
```

left-shift operation

new data is shifted in from the RHS:

[SystemVerilog] Source Code: shift_reg_4bit_areset.sv



	I_3	I_2	I_1	I_0	initial state after reset ...
t_0	I_2	I_1	I_0	D_0	
t_1	I_1	I_0	D_0	D_1	
...					

Shift direction
←

Notes

- Left-Shift operation, { , } - Concatenation Operator.

Synchronously resettable Shift-Register, serial in, serial out

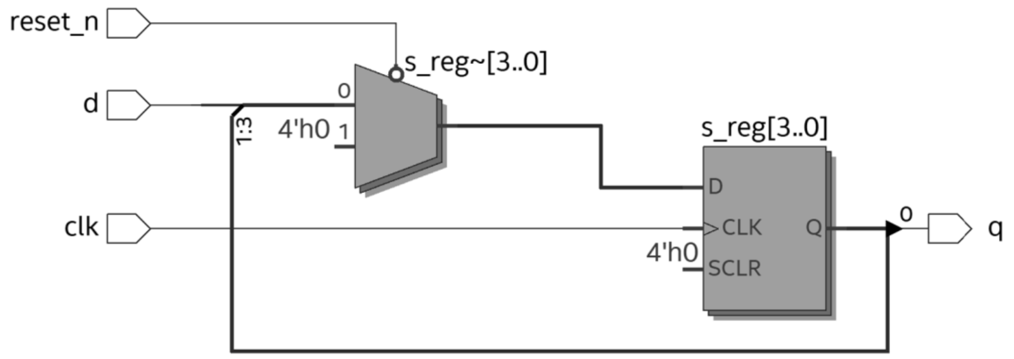
1/1

```
module shift_reg_4bit_areset(
   input  logic d, clk, reset_n,
   output logic q
   );
   logic [3:0] s_reg = 0;

   always_ff@(posedge clk)
      if(reset_n == 0)
            s_reg <= 0;
      else
            s_reg <= {d,s_reg[3:1]};

   assign q = s_reg[0];

endmodule
```

right-shift operation

new data is shifted in from the LHS:

[SystemVerilog] Source Code: shift_reg_4bit_areset.sv



	I_3	I_2	I_1	I_0	initial state after reset ...
t_0	D_0	I_3	I_2	I_1	
t_1	D_1	D_0	I_3	I_2	
...					

Shift direction

$\longrightarrow$

Notes

- Right-Shift operation, { , } - Concatenation Operator.

Synchronously resettable Shift-Register, serial in, serial out and shift-enable

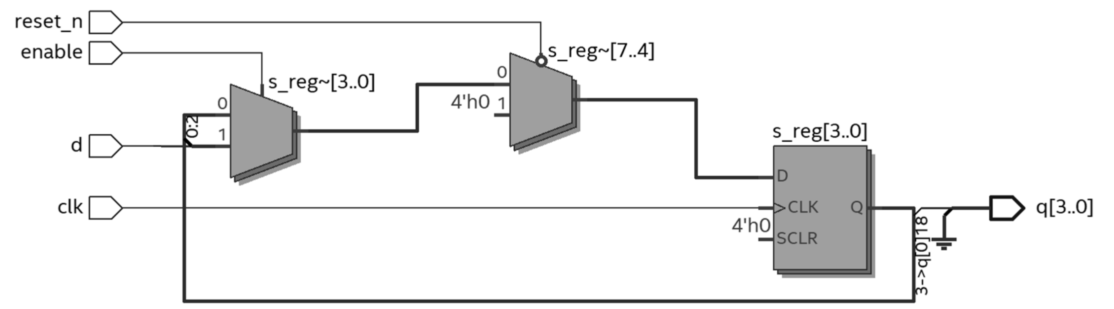
1/1

```
module shift_reg_4bit_areset(
   input  logic d, clk, reset_n, enable,
   output logic q
   );
   logic [3:0] s_reg = 0;

   always_ff@(posedge clk)
      if(reset_n == 0)
           s_reg <= 0;
      else if(enable)
           s_reg <= {s_reg[2:0],d};

   assign q = s_reg[3];

endmodule
```

0 or 1
shift-enable

Left shift

[SystemVerilog] Source Code: shift_reg_4bit_areset.sv



Notes

- Left-Shift operation, { , } - Concatenation Operator.

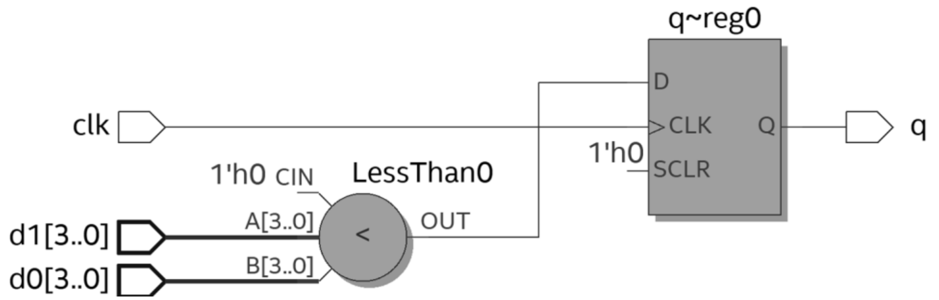
Combinational Logic with registered Outputs

Introduction to SystemVerilog

Comparator with reg. output - Conditional assignment

1/1

```
module comp_gt_4bit_reg(
   input  logic [3:0] d0, d1, input logic clk,
   output logic q
   );

   always_ff@(posedge clk)
      q <= (d0 > d1) ? 1 : 0; Conditional assignment

endmodule
```



[SystemVerilog] Source Code: comp_gt_4bit_reg.sv

Notes

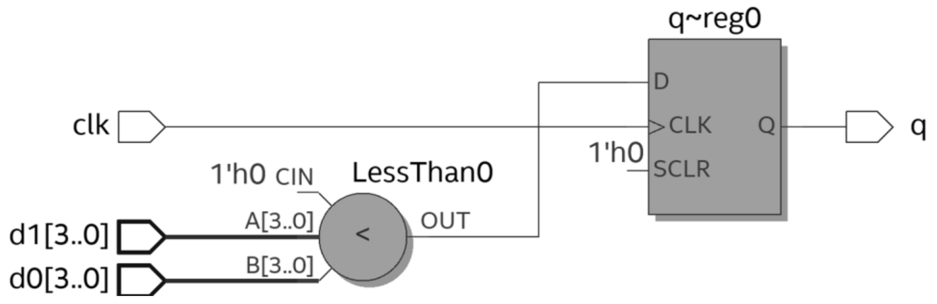
- The comparator stage is combined with another delay flip-flop …
- So by placing combinational logic, arithmetic operations or comparison logic into an always_ff block, we create Delay Flip-Flops along with the combinational logic. In all cases the logic is driving the inputs of the DFF's …

Comparator with reg. output - Conditional assignment

1/1

```
module comp_gt_4bit_reg(
  input  logic [3:0] d0, d1, input logic clk
  output logic q
  );

  always_ff@(posedge clk)
    if(d0 > d1)
       q <= 1;
    else
       q <= 0;

endmodule
```

[SystemVerilog] Source Code: comp_gt_4bit_reg.sv



Notes

- This is just an alternative description of the previous design …

When things go wrong – Using blocking statements in the wrong context ...

Introduction to SystemVerilog

When things go wrong – Using blocking statements in the wrong context ...

- The actual intention is to model two Delay-Flip-Flops connected in sequence ...

When things go wrong – Using blocking statements in the wrong context ...

=

1/1

```
module shift_reg_2bit_sreset(
  input  logic d, clk, reset_n,
  output logic q
  );
  logic [1:0] shift_reg = 0;

  always_ff@(posedge clk)
     if(reset_n == 0)
         shift_reg = 0;
     else begin
         shift_reg[0] = d;
         shift_reg[1] = shift_reg[0];
     end

  assign q = shift_reg[1];

endmodule
```

[SystemVerilog] Source Code: shift_reg_2bit_sync_reset.sv

Notes

- The actual intention is to model **two DFFs connected in sequence.**

When things go wrong – Using blocking statements in the wrong context ...

1/1

```
module shift_reg_2bit_sreset(
  input  logic d, clk, reset_n,
  output logic q
  );
  logic [1:0] shift_reg = 0;

  always_ff@(posedge clk)
     if(reset_n == 0)
         shift_reg = 0;
     else begin
         shift_reg[0] = d;
         shift_reg[1] = shift_reg[0];
     end

  assign q = shift_reg[1];

endmodule
```

clock-sync., sequential procedural block

blocking assignment operator

[SystemVerilog] Source Code: shift_reg_2bit_sync_reset.sv

Notes

- The actual intention is to model **two DFFs connected in sequence.**

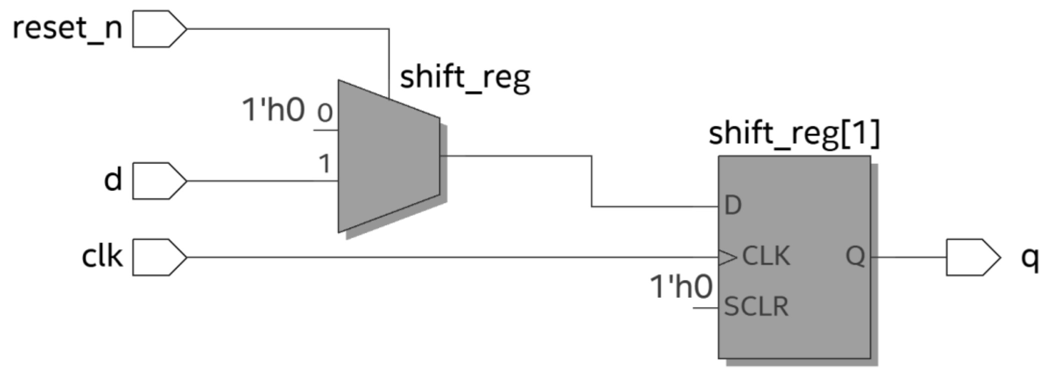
When things go wrong – Using blocking statements in the wrong context ...

1/1

```
module shift_reg_2bit_sreset(
  input  logic d, clk, reset_n,
  output logic q
  );
  logic [1:0] shift_reg = 0;

  always_ff@(posedge clk)
    if(reset_n == 0)
        shift_reg = 0;
    else begin
        shift_reg[0] = d;
        shift_reg[1] = shift_reg[0];
    end

  assign q = shift_reg[1];

endmodule
```

The **blocking assignment** operator causes the assignments to be performed as if in sequential order.

[SystemVerilog] Source Code: shift_reg_2bit_sync_reset.sv



Notes

- The actual intention is to model **two DFFs connected in sequence.**
- Please note the synthesis result ...

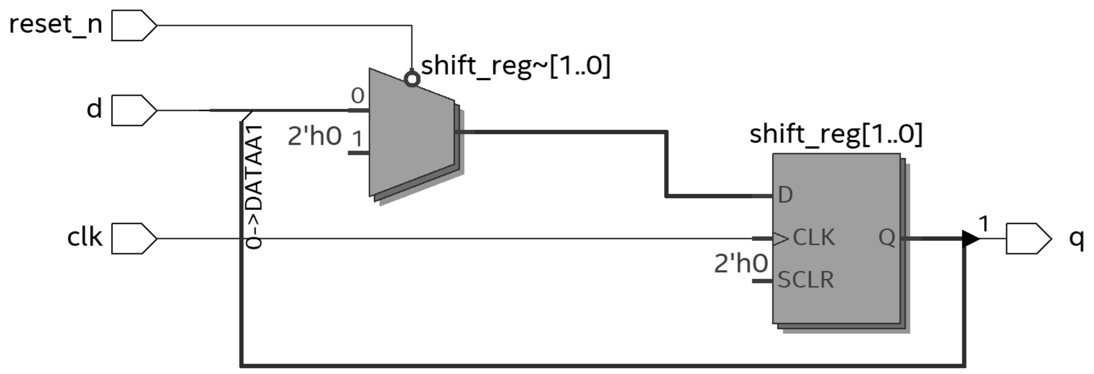
When things go wrong – Using blocking statements in the wrong context ...

1/1

```systemverilog
module shift_reg_2bit_sreset(
   input  logic d, clk, reset_n,
   output logic q
   );
   logic [1:0] shift_reg = 0;

   always_ff@(posedge clk)
      if(reset_n == 0)
          shift_reg <= 0;
      else begin
          shift_reg[0] <= d;
          shift_reg[1] <= shift_reg[0];
      end

   assign q = shift_reg[1];

endmodule
```

non-blocking assignment operator

[SystemVerilog] Source Code: shift_reg_2bit_sync_reset.sv



Note

- The code is almost the same as before, but now with non-blocking assignment operators what finally leads to the desired synthesis result.

Counters in SystemVerilog

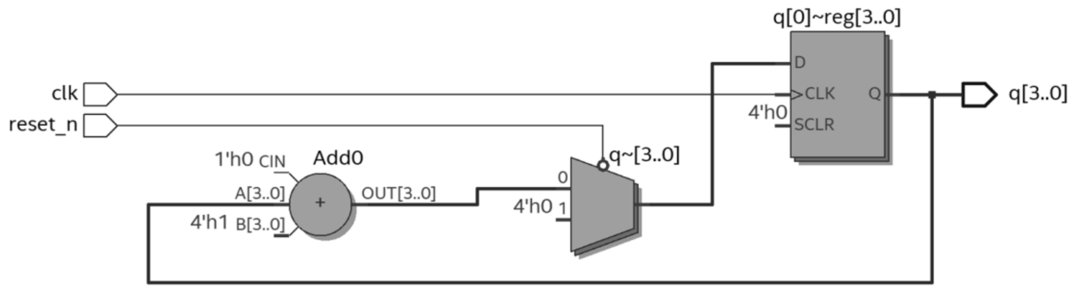
Introduction to SystemVerilog

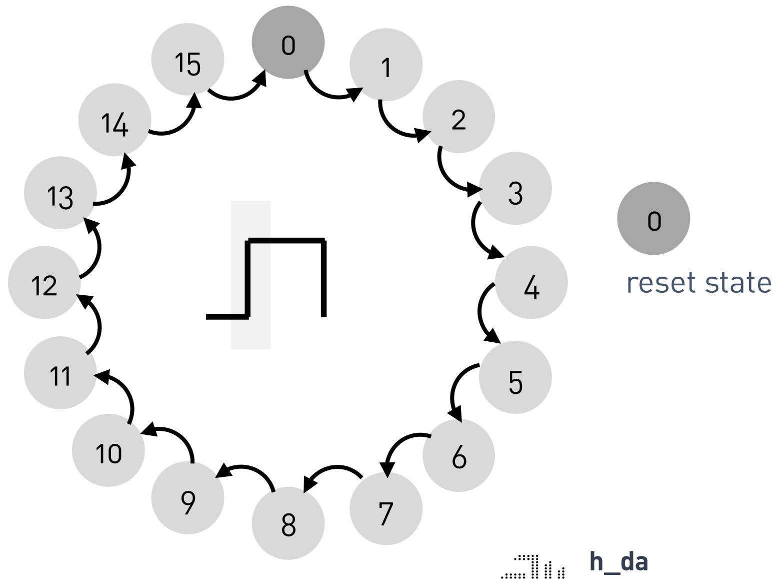
Synchronous Up-Counter Version ´1´

1/1

```
module sync_counter_v1(
   input logic clk, input logic reset_n,
   output logic [3:0] q
   );
   always_ff@(posedge clk)
      if(reset_n == 0)
         q <= 0;
      else
         q <= q + 1;
endmodule
```

[SystemVerilog] Source Code: sync_counter_v1.sv



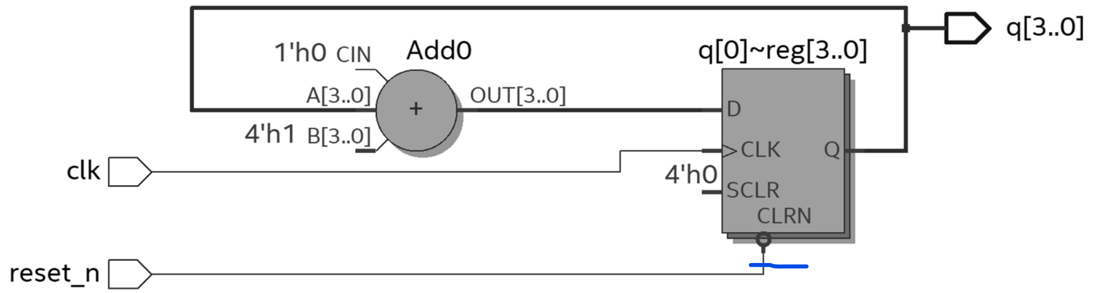


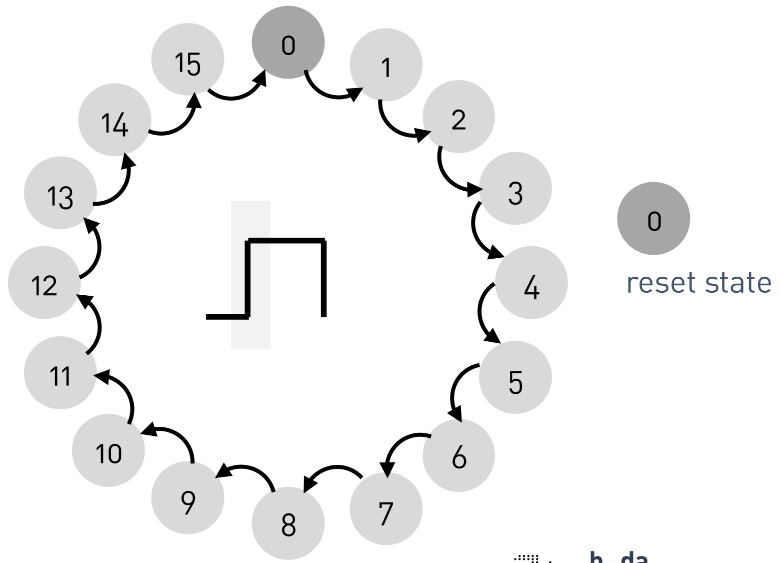
Synchronous Up-Counter Version ´2´

Asynchronous

```
module sync_counter_v2(
   input logic clk, input logic reset_n,
   output logic [3:0] q
   );
   always_ff@(posedge clk, negedge reset_n)
      if(reset_n == 0)
         q <= 0;
      else
         q <= q + 1;
endmodule
```
1/1

[SystemVerilog] Source Code: sync_counter_v2.sv





Synchronous Up-Counter Version ´3´

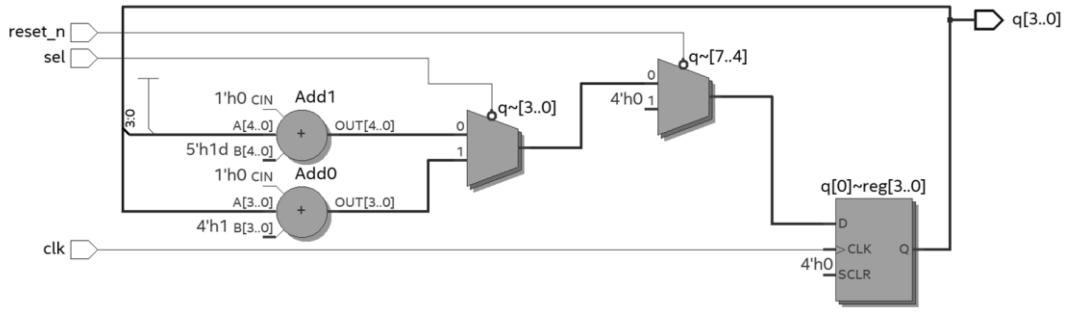
Up Down Counter

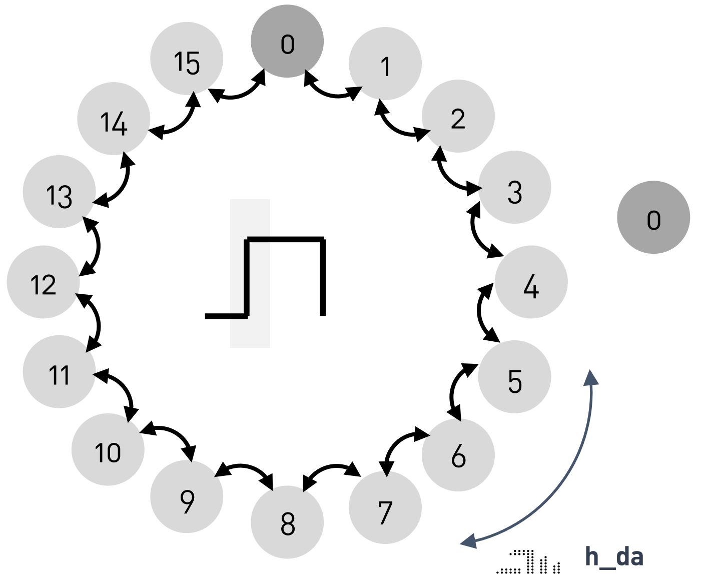
1/1

```
module sync_counter_v3(
  input logic clk, reset_n, sel,
  output logic [3:0] q
  );
  always_ff@(posedge clk)
     if(reset_n == 0)
        q <= 0;
     else if(sel == 0)
        q <= q + 1;
     else
        q <= q - 1;

endmodule
```

select determines the counting direction

[SystemVerilog] Source Code: sync_counter_v3.sv



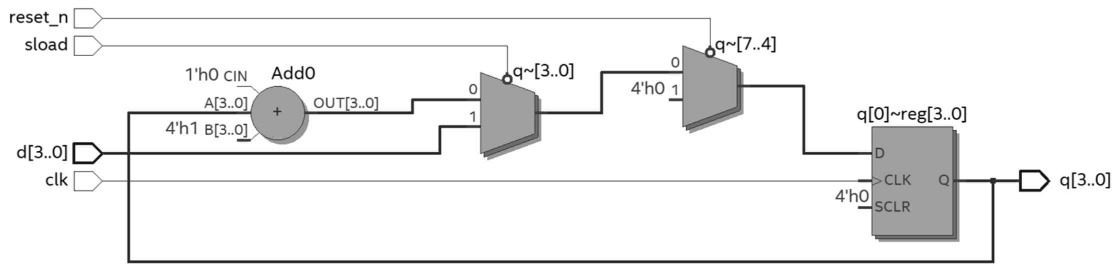


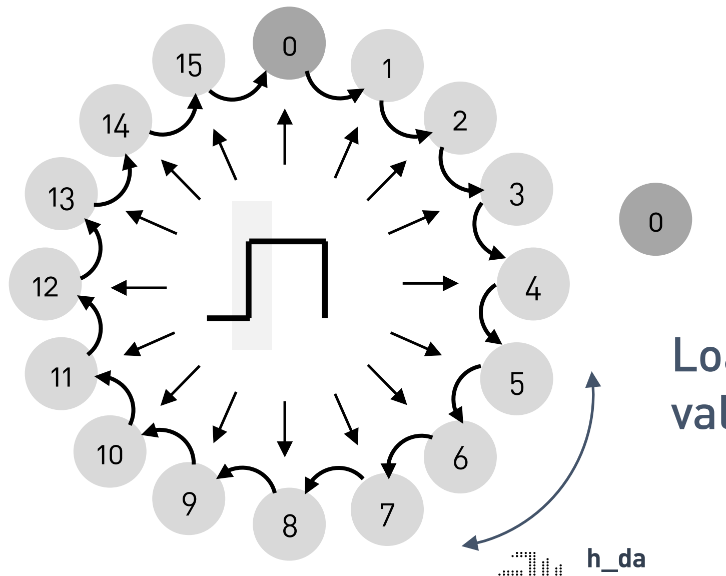
Synchronous Up-Counter Version ´4´

1/1

```
module sync_counter_v4(
   input logic clk, reset_n, sload,
   input logic [3:0] d, output logic [3:0] q
   );
   always_ff@(posedge clk)
      if(reset_n == 0)
         q <= 0;
      else if(sload == 0)      ← load new start value ...
         q <= d;
      else
         q <= q + 1;

endmodule
```

[SystemVerilog] Source Code: sync_counter_v4.sv





Load any value ...

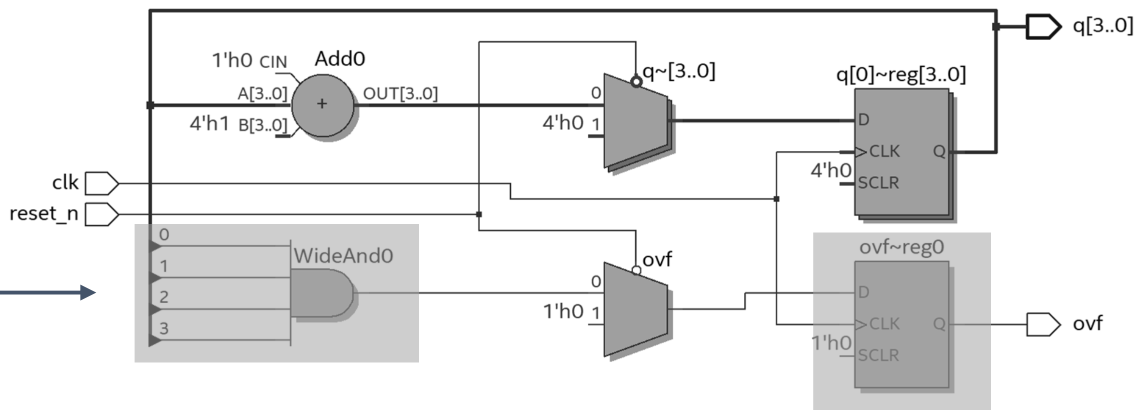
Synchronous Up-Counter Version ´5´

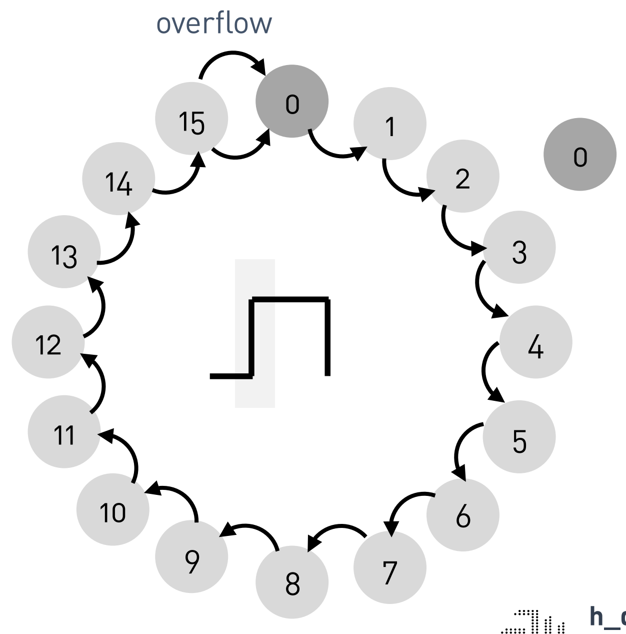
1/1

```
module sync_counter_v5(
   input logic clk, reset_n,
   output logic [3:0] q, output logic ovf
   );
   always_ff@(posedge clk)
      if(reset_n == 0) begin
         q <= 0; ovf <= 0;
      end
      else begin
         q <= q + 1; ovf <= &q;
      end

endmodule
```

overflow status signal

[SystemVerilog] Source Code: sync_counter_v5.sv





Notes

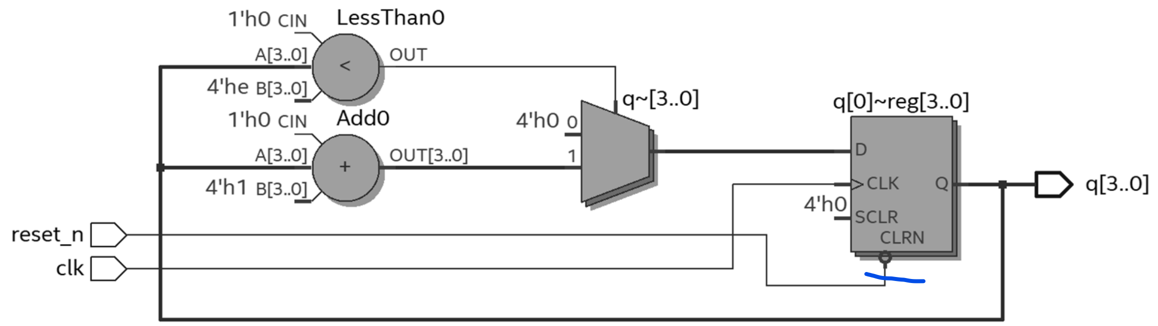
- In the above example, the SystemVerilog reduction ´&´ operator is used to generate the overflow signal ...
- The overflow signal is assigned within the procedural always_ff block. This creates an additional output flip-flop that registers the actual overflow signal.

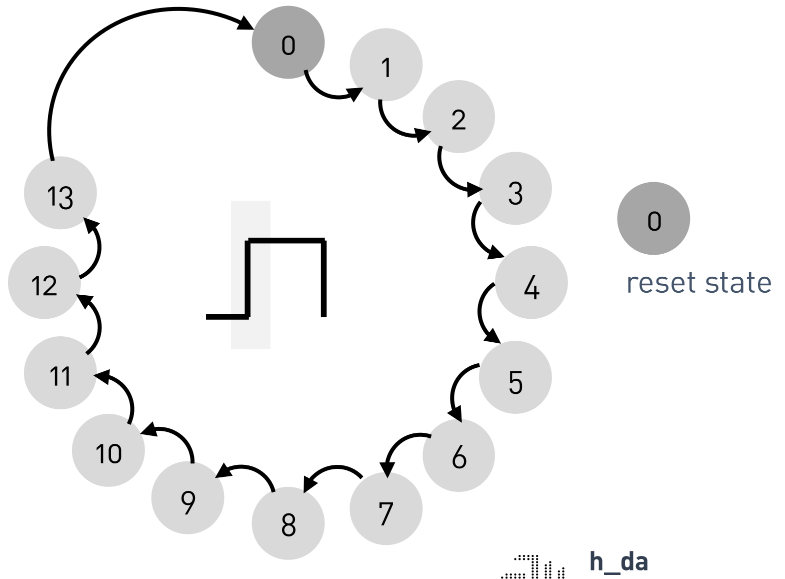
Synchronous Up-Counter Version ´6´

1/1

```
module sync_counter_v6(
   input logic clk, input logic reset_n,
   output logic [3:0] q
   );
   always_ff@(posedge clk, negedge reset_n)
      if(reset_n == 0)
         q <= 0;
      else if(q < 13)
         q <= q + 1;
      else
         q <= 0;
endmodule
```

13 is part of the counting sequence





[SystemVerilog] Source Code: sync_counter_v6.sv

Notes

- The resulting counting sequence is as follows: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 0, 1, 2, 3, 4, ...

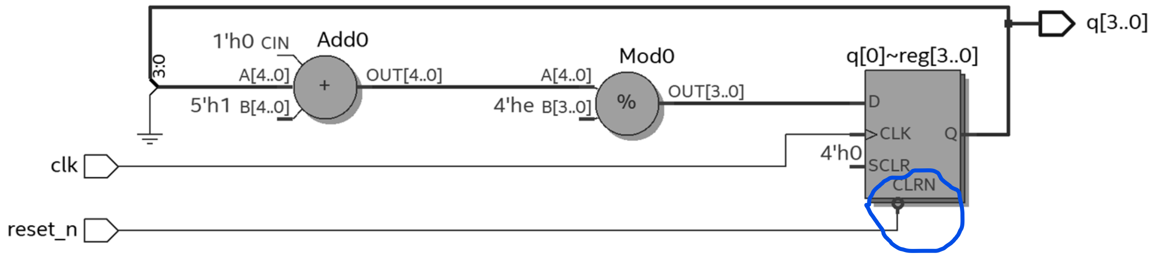
Synchronous Up-Counter Version ´7´

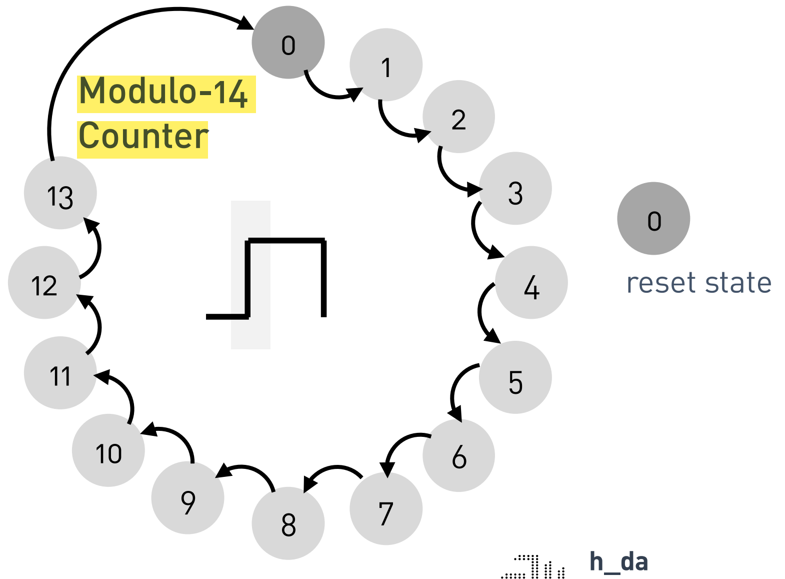
1/1

```
module sync_counter_v7(
   input logic clk, input logic reset_n,
   output logic [3:0] q
   );
   always_ff@(posedge clk, negedge reset_n)
      if(reset_n == 0)
         q <= 0;
      else
         q <= (q + 1) % 14;
endmodule
```

→ the counting sequence comprises of 14 numbers …

[SystemVerilog] Source Code: sync_counter_v7.sv





Notes

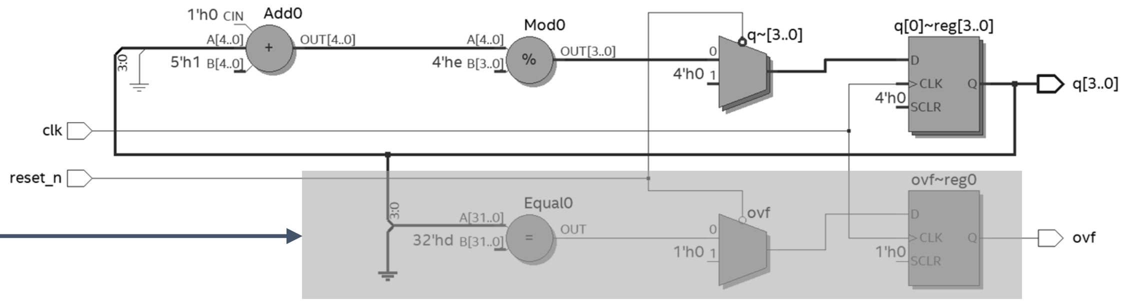
- The resulting counting sequence is as follows: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 0, 1, 2, 3, 4, …

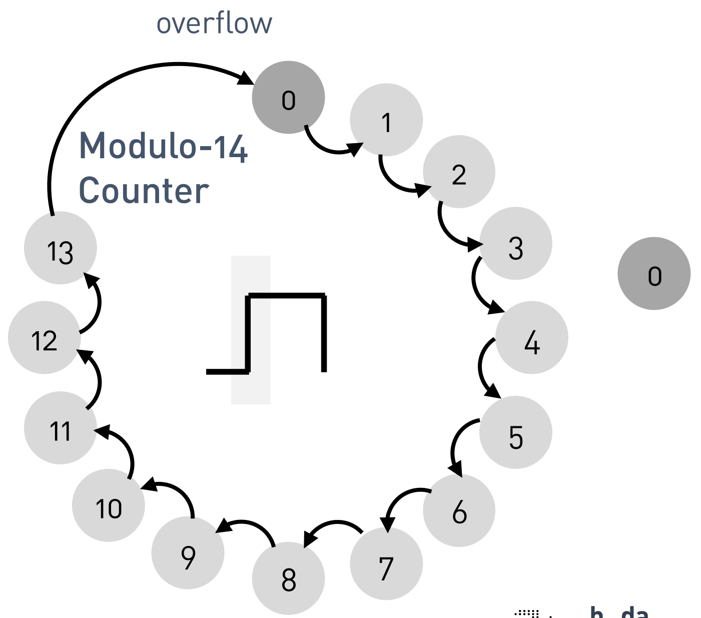
Synchronous Up-Counter Version ´8´

1/1

```
module sync_counter_v8(
   input logic clk, reset_n,
   output logic [3:0] q, output logic ovf
   );
   always_ff@(posedge clk)
      if(reset_n == 0) begin
         q <= 0; ovf <= 0;
      end
      else begin
         q <= (q + 1) % 14; ovf <= (q == 13) ? 1:0;
      end

endmodule
```

comparator with registered output





[SystemVerilog] Source Code: sync_counter_v8.sv

Notes

- Here, a **conditional assignment** is used to generate the overflow signal …

?

Gray Counter

1/1

```
module gray_count_4bit(
   input logic clk, reset_n,
   output logic [3:0] q
);
logic [3:0] bin_count = 0;

always_ff@(posedge clk)
   if(reset_n == 0)
      bin_count <= 0;
   else
      bin_count <= bin_count + 1;

 assign q = (bin_count >> 1) ^ bin_count;

endmodule
```

[SystemVerilog] Source Code: gray_count_4bit.sv

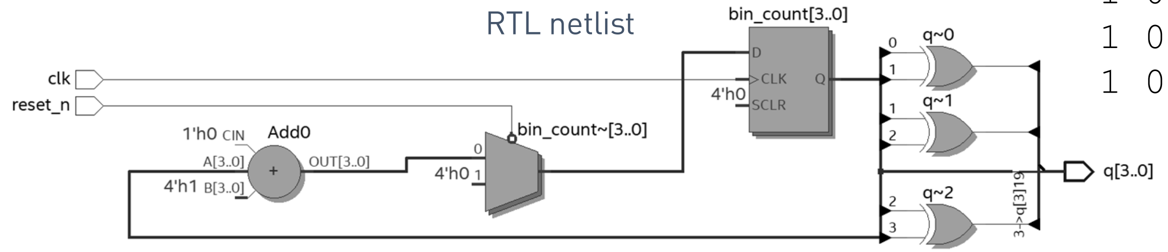
Notes

- Gray code after Frank Gray, is an ordering of the binary numeral system such that two successive values differ in only one bit:

0	0	0	0	0
0	0	0	1	1
0	0	1	1	3
0	0	1	0	2
0	1	1	0	6
0	1	1	1	7
0	1	0	1	5
0	1	0	0	4
1	1	0	0	12
1	1	0	1	13
1	1	1	1	15
1	1	1	0	14
1	0	1	0	10
1	0	1	1	11
1	0	0	1	9
1	0	0	0	8

two successive values differ in only one bit

RTL netlist



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

A first set of coding guidelines

always_ff block
always_ff@(poesdge ckl)
<=

- 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.