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

h_da
HOCHSCHULE DARMSTADT
UNIVERSITY OF APPLIED SCIENCES

Synchronous time base generation

Today's Agenda

Intended topics for today's session

- SystemVerilog for Design: An introduction to synchronous time base generation

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

Synchronous time base generation

Objectives

By the end of this lecture you will be able to

- implement SystemVerilog designs with custom and application specific time bases.

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

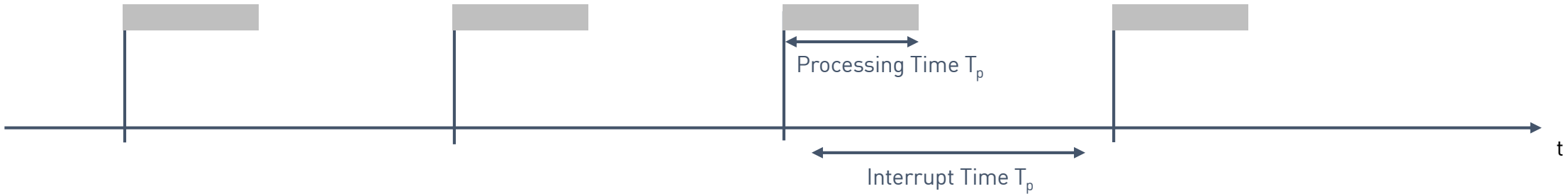
Synchronous Time Base Generation

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

- **Synchronous time base generation**: Ensure that a dedicated functionality is executed in periodic and equidistant time periods.



- Review: How do we do that using a µC?

A more sophisticated time base generation scheme ...

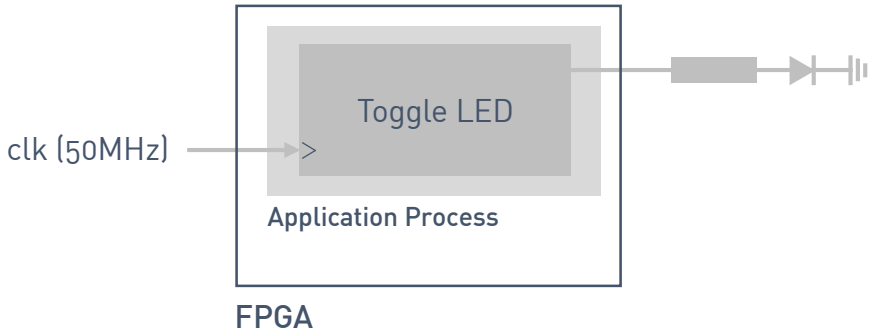
Introduction to SystemVerilog

- Ensure that a dedicated functionality is executed in equidistant time periods.
- Consider an FPGA as the implementation platform of interest.

A first attempt ...

```
1. module blinking_led_v1 (
2.     input logic clk,          // 50MHz oscillator
3.     input logic reset_n,      // active low reset
4.     output logic q
5.     );
6.
7.     // application process ...
8.     always_ff@(posedge clk)
9.         if(reset_n == 0)
10.            q <= 0;
11.        else
12.            q <= ~q;
13. endmodule
```

[SystemVerilog] Source Code: blinking_led_v1.sv



- LED Blink frequency? – 25MHz ... way too fast to be detectable by the human eye ...

The central question is how to generate application specific frequencies ...

- **Use Case #1**: The FPGA operates at a (main) clock of 50MHz but how can we blink an LED with just 5Hz?
- **Use Case #2**: The FPGA operates at a (main) clock of 50MHz but how can we operate a FPGA internal I2C controller at 100kHz/400kHz?

A second attempt: Consider again the LED example …

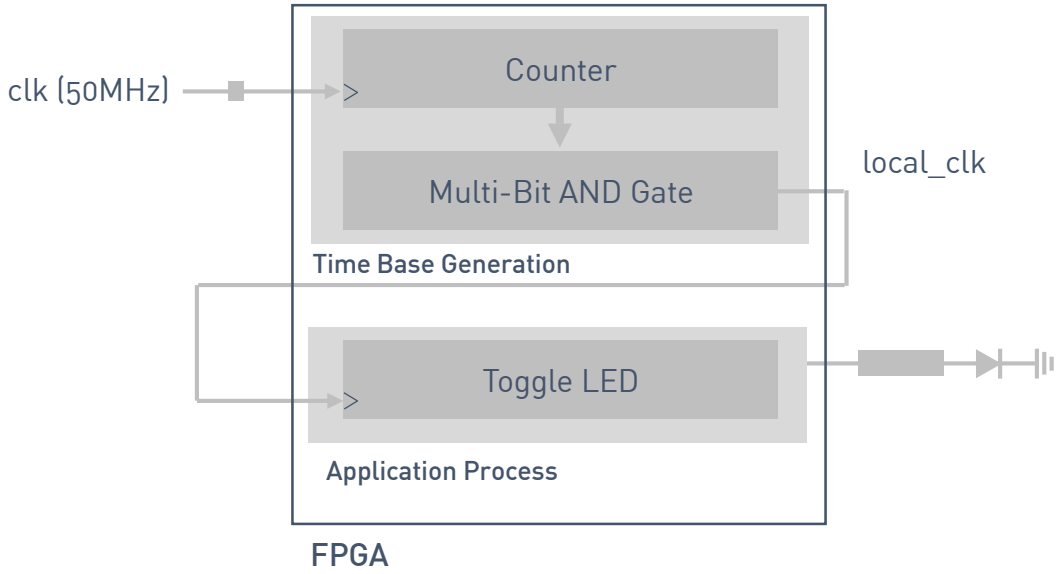
Introduction to SystemVerilog

- 50MHz on-board oscillator
- LED connected to general purpose FPGA I/O pin. Let's create a toggle frequency that is detectable by the human eye …

A second attempt ... however a pretty bad approach

```
1.  module blinking_led_v2 (
2.     input logic clk, input logic reset_n,
3.     output logic q_led
4.     );
5.     // time base generation ...
6.     logic [22:0] time_base = 0;
7.     always_ff@(posedge clk)
8.         if(reset_n == 0)
9.             time_base <= 0;
10.        else
11.            time_base <= time_base + 1;
12.
13.    logic local_clk; assign local_clk = &time_base;
14.    // application process ...
15.    always_ff@(posedge local_clk)
16.        if(reset_n == 0)
17.            q <= 0;
18.        else
19.            q <= ~q;
20. endmodule
```

here, we create a new clock signal ...



- **In conclusion:** Never use such an approach! In this example, we use the overflow of counter as a new clock signal. This might lead to clock skew and thus to severe timing problems ...

[SystemVerilog] Source Code: blinking_led_v2.sv

Another ‘more useful’ attempt …

Introduction to SystemVerilog

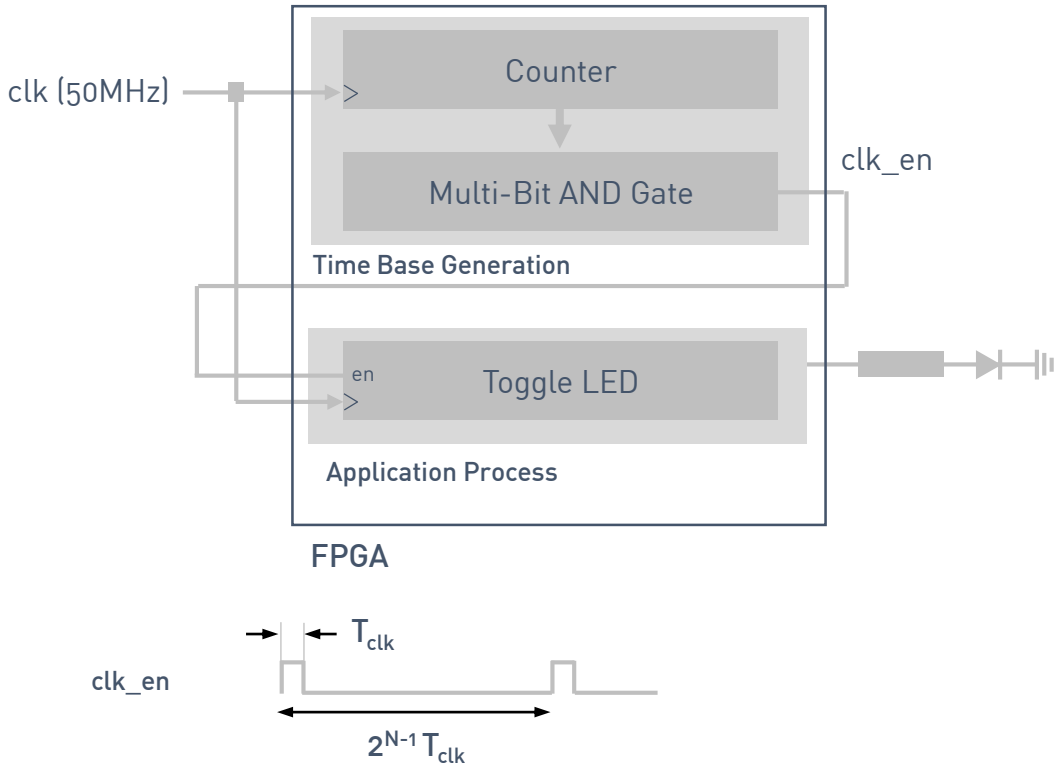
- 50MHz on-board oscillator
- LED connected to general purpose FPGA I/O pin

Another ‘more useful’ attempt …

```
1.  module blinking_led_right_approach_v2 (
2.     input logic clk, input logic reset_n,
3.     output logic q_led
4.     );
5.     // time base generation ...
6.     logic [22:0] time_base = 0; 23 bit counter
7.     always_ff@(posedge clk)
8.        if(reset_n == 0)
9.           time_base <= 0;
10.       else
11.          time_base <= time_base + 1;
12.
13.    logic enable; assign enable = &time_base;
14.    // application process ...
15.    always_ff@(posedge clk)
16.       if(reset_n == 0)
17.          q <= 0;
18.       else if(enable)
19.          q <= ~q;
20. endmodule
```

here, we create a clock-synchronous enable signal …

[SystemVerilog] Source Code: blinking_led_right_approach_v2.sv



- The output of the **time base generator** is high for single clock period and remains low for another $2^{N-1}T_{clk} - 1$ clock cycles.
- The application process is only active if this signal is high!
- It therefore acts as a clock synchronous **prescaler** for the following application process

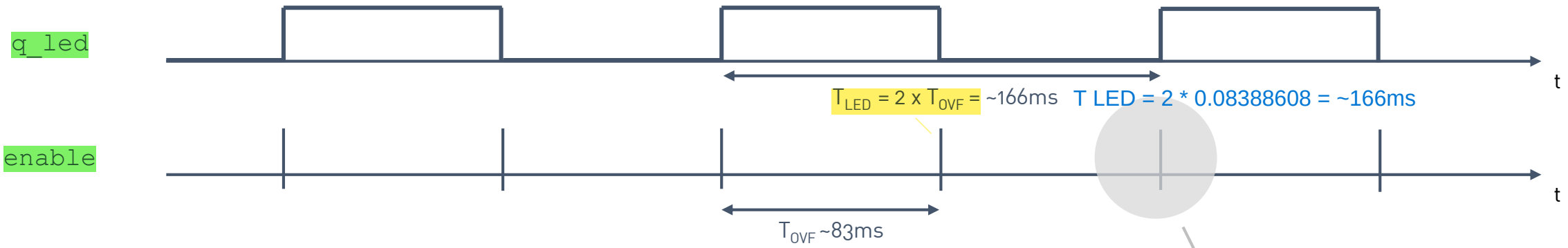
IMP

Another ‘more useful’ attempt …

Tclk = 1/ f
1/ 50MHz
20 ns

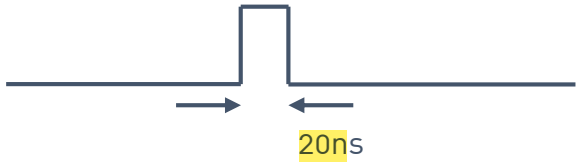
2^(N-1) * Tclk = 2^(23-1) * 20ns = ~83ms

- **Example:** 23-bit time base counter, operating at 50MHz
- Overflow period: 2^{22} x 20ns = 0,08388608s



- Resulting LED toggle frequency: T_{LED}^{-1} = ~ 6 Hz

f = 1 / TLED
1 / 166ms
~6Hz



Design Reuse - Let’s put the time base generation process into a separate module …

Introduction to SystemVerilog

- The intention is to reuse the time base generation process in various designs and to configure it with an application specific setting …

A simple synchronous time base generation module contd.

```
1. module time_base_gen #(parameter L = 23) (
2.    input logic clk, input logic reset_n,
3.    output logic q
4.    );
5.    // time base generation ...
6.    logic [L-1:0] time_base = 0;
7.    always_ff@(posedge clk)
8.        if(reset_n == 0)
9.            time_base <= 0;
10.       else
11.           time_base <= time_base + 1;
12.
13.   assign q = &time_base;
14.
15. endmodule
```

[22:0]

- A more general approach using a parameter statement to specify the counter bit-width
- The parameter value can be overwritten at compile time

23 bit time base counter

[SystemVerilog] Source Code: time_base_gen.sv

Something in between – Module Instances and Hierarchy ...

Introduction to SystemVerilog

Module Instances and Hierarchy

- Complex designs are partitioned into smaller blocks that are connected together.
- Each sub-block is represented as a module.

Syntax of a Module Instance

```
1.  module some_top_level_module (
2.        // I/O port declarations
3.  );
4.
5.  module_name #(parameter_values) instance_name (connection_to_ports);
8.  ...
9.
```

[SystemVerilog] Source Code Excerpt: Module Instantiation

- A module instance is a reference to the name of a module.
- It is optional to define parameter values. Using parameters to make modules configurable is discussed in the next section.
- The module instance name is required, and must be a unique identifier within the context of the module.
- Inside the following parenthesis, the port to signal connections are specified

Module Instantiation

- SystemVerilog provides two ways to define the connections to the signals of the instantiating module: by **port order**, or by port name.

Module Instantiation – Port Order Connections

```
1. module simple_pwm_8bit (
2.    input logic clk, input logic reset_n,
3.    input logic [7:0] duty_cycle,
4.    output logic pwm
5.    );
6.
7.    logic [7:0] count;   // intermediate wire
8.
9.    counter_8bit pwm_counter(clk, reset_n, count);
10.   comparator_8bit pwm_comp(duty_cycle, count, pwm);
11.
12. endmodule
```

[SystemVerilog] Source Code: simple_pwm_8bit.sv

Notes:

- Connections between modules are **implicit** through the use of i/o port signal names such as clk, reset_n or pwm …

```
1. module counter_8bit (
2.    input logic clk, reset_n,
3.    output logic [7:0] q
4.    );
5.    always_ff@(posedge clk)
6.        if(reset_n == 0)
7.            q <= 0;
8.        else
9.            q <= q + 1'b1;
10. endmodule
```

[SystemVerilog] Source Code: counter_8bit.sv

```
1. module comparator_8bit (
2.    input logic [7:0] a, b, output logic q
3.    );
4.    assign q = (a > b) ? 0 : 1;
5. endmodule
```

[SystemVerilog] Source Code: comparator_8bit.sv

Module Instantiation – Port Order Connections

- Port order connections are error prone.
- A simple coding mistake of listing a connection in the wrong order can result in design bugs that are difficult to debug …
- SystemVerilog provides two ways to define the connections to the signals of the instantiating module: by port order, or by **port name**.

Module Instantiation – Named Port Connections

```
1. module simple_pwm_8bit (
2.    input logic clk, input logic reset_n,
3.    input logic [7:0] duty_cycle,
4.    output logic pwm
5.    );
6.
7.    logic [7:0] count;   // intermediate wire
8.
9.    counter_8bit pwm_counter(.clk(clk),
10.                            .reset_n(reset_n)
11.                            .q(count)
12.                           );
13.   ...
14.
15. endmodule
```

[SystemVerilog] Source Code: simple_pwm_8bit.sv

Explicit named connection: Named port connections associate a port name with a local signal name connected to that port.

Module Instantiation – Named Port Connections

- Named port connections can be listed in any order.
- Unused ports can be explicitly listed, but with no local signal name in the parentheses …
- The code is self-documenting.
- In conclusion: Use named port connections for all module instances.

Module Instantiation – Named Port Connections - Disadvantage

```
1. module simple_pwm_8bit (
2.    input logic clk, input logic reset_n,
3.    input logic [7:0] duty_cycle,
4.    output logic pwm
5.    );
6.
7.    logic [7:0] count;   // intermediate wire
8.
9.    counter_8bit pwm_counter(.clk(clk),
10.                            .reset_n(reset_n)
11.                            .q(count)
12.                           );
13.   ...
14.
15. endmodule
```

[SystemVerilog] Source Code: simple_pwm_8bit.sv

Disadvantage: Named port connections are verbose, and can require considerable replication of names.

Module Instantiation – Named Port Connections - Shortcut

```
1. module simple_pwm_8bit (
2.    input logic clk, input logic reset_n,
3.    input logic [7:0] duty_cycle,
4.    output logic pwm
5.    );
6.
7.    logic [7:0] count;   // intermediate wire
8.
9.    counter_8bit pwm_counter(.clk,
10.                           .reset_n,
11.                           .q(count)
12.                          );
13.   ...
14.
15. endmodule
```

[SystemVerilog] Source Code: simple_pwm_8bit.sv

Dot-name shortcuts: Eliminates the need to type the same name twice to connect a net to a port.

Design Reuse - Instantiating the previously designed Time-Base module ...

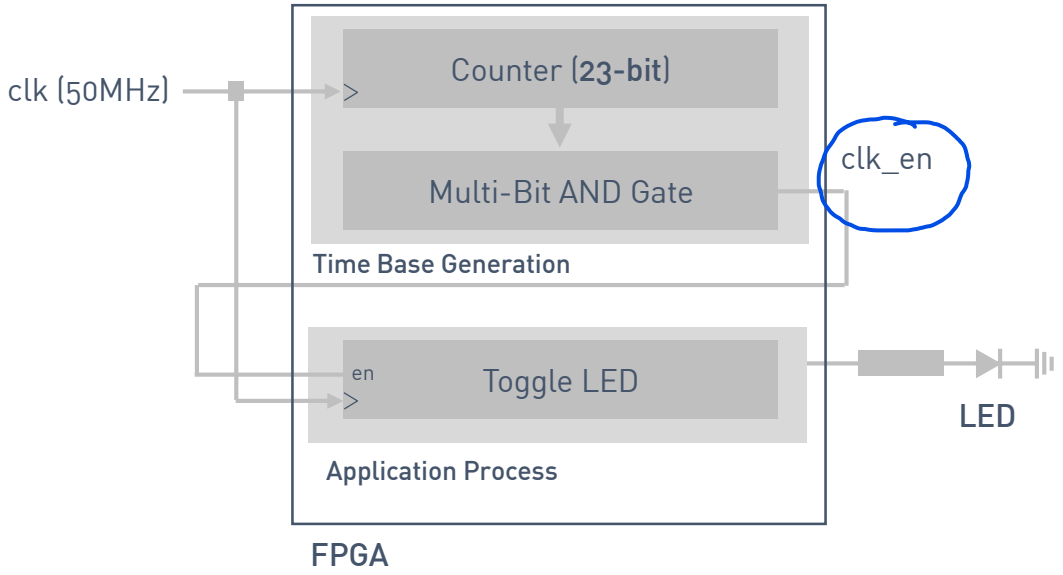
Introduction to SystemVerilog

- **How to parameterize a module** in SystemVerilog or many ways lead to Rome: **Solution #1.**

Instantiating a generic module in SystemVerilog ...

```
1. module blinking_led_v3 (
2.    input logic clk, input logic reset_n,
3.    output logic q
4.    );
5.    // time base generation ...
6.    logic enable;
7.    time_base_gen #(23) inst_0 (.clk,
8.                                .reset_n,
9.                                .q(enable));
10.   // application process ...
11.   always_ff@(posedge clk)
12.      if(reset_n == 0)
13.          q <= 0;
14.      else if(enable)
15.          q <= ~q;
16.
17. endmodule
```

[SystemVerilog] Source Code: blinking_led_v3.sv



- LED Blink frequency? ~6Hz

- How to parameterize a module in SystemVerilog or many ways lead to Rome: **Solution #1.**

Design Reuse - Instantiating the previously designed Time-Base module ...

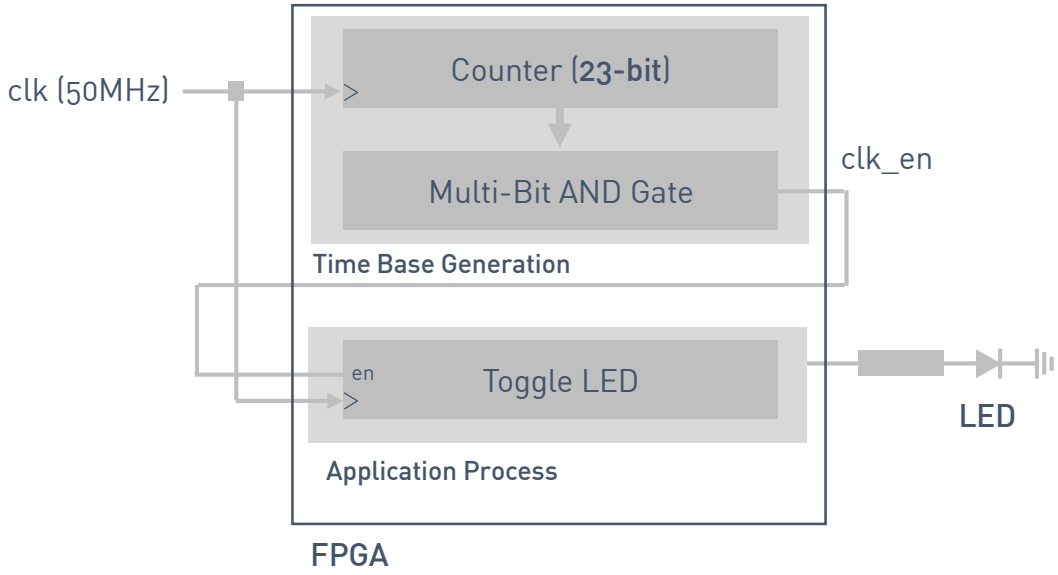
Introduction to SystemVerilog

- **How to parameterize a module** in SystemVerilog or many ways lead to Rome**: Solution #2.**

Instantiating a generic module in SystemVerilog ...

```
1. module blinking_led_v4 (
2.    input logic clk, input logic reset_n,
3.    output logic q
4.    );
5.    // time base generation ...
6.    logic enable;
7.    time_base_gen #(.L(23)) inst_0 (.clk,
8.                                    .reset_n,
9.                                    .q(enable));
10.   // application process ...
11.   always_ff@(posedge clk)
12.       if(reset_n == 0)
13.           q <= 0;
14.       else if(enable)
15.           q <= ~q;
16.
17. endmodule
```

[SystemVerilog] Source Code: blinking_led_v4.sv



- LED Blink frequency?

- How to parameterize a module in SystemVerilog or many ways lead to Rome**: Solution #2.**
- This approach is especially useful if multiple parameters must be defined ...

Design Reuse - Instantiating the previously designed Time-Base module ...

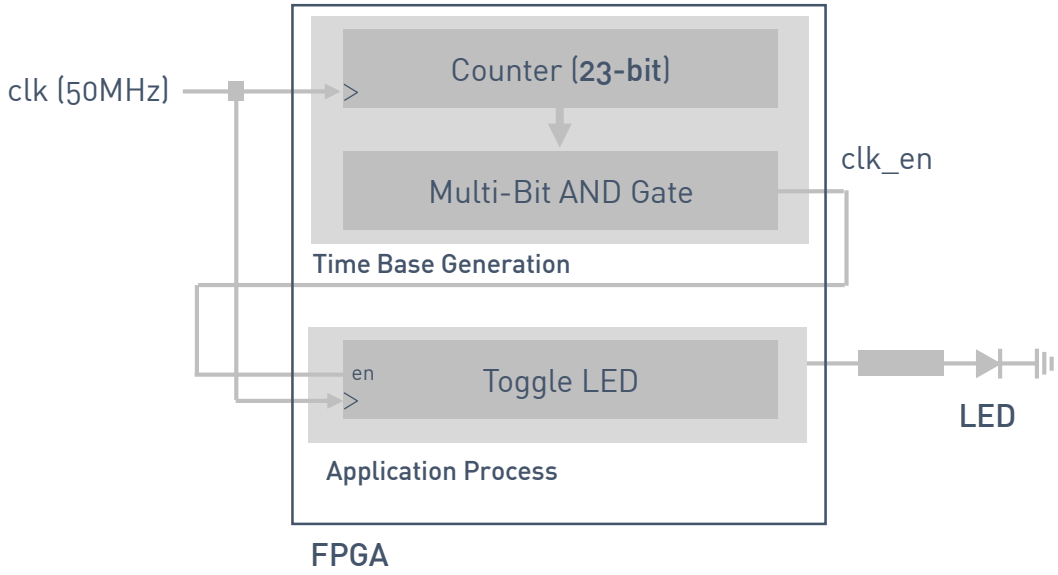
Introduction to SystemVerilog

- **How to parameterize a module** in SystemVerilog or many ways lead to Rome: **Solution #3.**

Instantiating a generic module in SystemVerilog ...

```
1. module blinking_led_v5 (
2.    input logic clk, input logic reset_n,
3.    output logic q
4.    );
5.    // time base generation ...
6.    logic enable;
7.    time_base_gen inst_0 (.clk,
8.                          .reset_n,
9.                          .q(enable));
10.   defparam inst_0.L = 23;
11.
12.   // application process ...
13.   always_ff@(posedge clk)
14.      if(reset_n == 0)
15.          q <= 0;
16.      else if(enable)
17.          q <= ~q;
18.
19. endmodule
```

[SystemVerilog] Source Code: blinking_led_v5.sv



- LED Blink frequency?

- How to parameterize a module in SystemVerilog or many ways lead to Rome: **Solution #3.**

Instantiating a generic module in SystemVerilog ...

- The `parameter` statement is used in the context of the module description.
- Whereas the `defparam` statement is used in the context of the module instantiation.

A more sophisticated time base generation scheme ...

Introduction to SystemVerilog

A more sophisticated time base generation scheme ...

```
1. module time_base_gen #(parameter L = 23) (
2.    input logic clk, input logic reset_n,
3.    output logic q
4.    );
5.    // time base generation ...
6.    logic [L-1:0] time_base = 0;
7.    always_ff@(posedge clk)
8.       if(reset_n == 0)
9.           time_base <= 0;
10.      else
11.          time_base <= time_base + 1;
12.
13.   assign q = &time_base;
14.
15. endmodule
```

[SystemVerilog] Source Code: time_base_gen.sv

- The code shows the previously developed time base generation scheme.
- What are the disadvantages associated with that approach?

- **Review Question:**
  a) How does the overflow period changes when increasing or decreasing the counter size by a single bit.
  b) How does the enable frequency changes when increasing or decreasing the counter size by a single bit.

Tovf = 2^(N-1) * Tclk

enable frequency = 1 / Tovf

?

A more sophisticated time base generation scheme …

- How does the overflow period changes when increasing or decreasing the counter size by a single bit? We assume a fixed clock frequency of 50MHz.
  - Time base counter width of **23 bits**: Overflow period of 2^{23} x 20ns = **0,16777216s**, which again results in a frequency of **5,960Hz**.
  - Time base counter width of **22 bits**: Overflow period of 2^{22} x 20ns = **0,0838861s**, which again results in a frequency of **11,921Hz**.
  - Time base counter width of **24 bits**: Overflow period of 2^{24} x 20ns = **0,3355442s**, which again results in a frequency of **2,980Hz**.
- The overflow based time base generation scheme has the disadvantage of a **coarse-grained** frequency resolution.
- How to generate a frequency of exactly 10Hz without changing the operating frequency of 50MHz?

Decide size of time base counter

A more sophisticated time base generation scheme …

- The range of possible application frequencies can be drastically increased by replacing the regular counter by a **modulo based counting** scheme …
  - A 10Hz frequency results in period of 100ms, which is again equivalent to 100ms/20ns = 5.000.000 - 50Mhz clock periods.
  - Therefore, by setting the module counter to 5.000.000 cycles, we precisely synthesize a frequency of 10Hz …
- How to easily determine the **required counter width** for a given counting value?
  - Use the binary logarithm (log2 n) = (log10 n/log10 2)
  - Always round the result towards **+infinity** …
  - log2(5.000.000) = 22,25349 …
  - So, we obviously need a 23-bit wide counter …

f =10Hz
T =100ms

modulo counter: 100ms/20ns = 5MHz cycles

Select counter:

log2(5MHz) = 22.25349

23bit wide counter

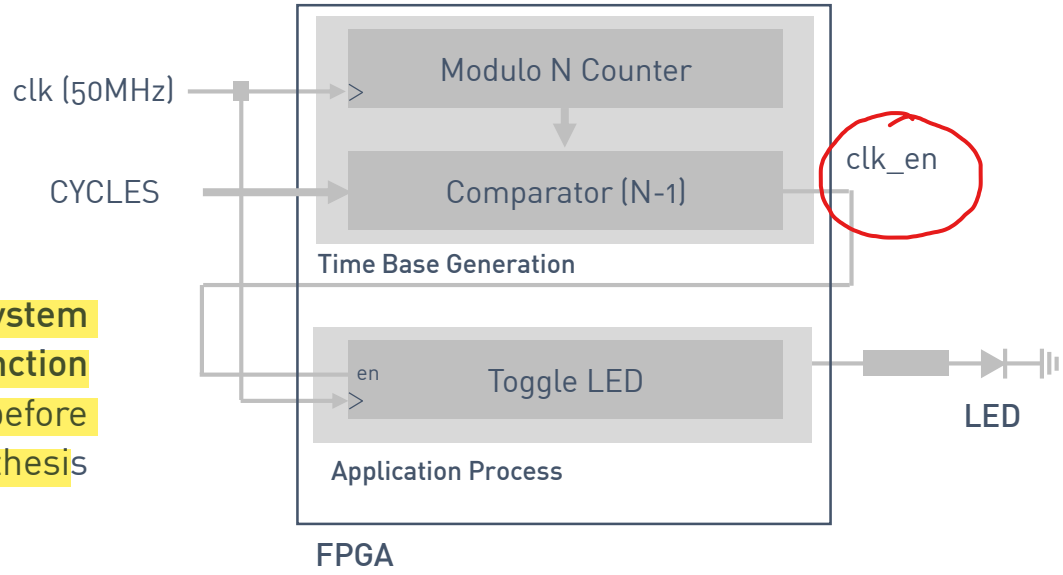
A more sophisticated time base generation scheme

```
1. module time_base_gen #(parameter CYCLES = 5000000) (
2.    input logic clk, reset_n,
3.    output logic q
4.    );
5.
6.    localparam BITWIDTH = $clog2(CYCLES);
7.
8.    logic [BITWIDTH-1:0] time_base = 0;
9.    always_ff@(posedge clk)
10.       if(reset_n == 0)
11.          time_base <= 0;
12.       else
13.          time_base <= (time_base + 1) % CYCLES;
14.
15.   assign q = (time_base == (CYCLES - 1)) ? 1 : 0;
16.
17. endmodule
```

SystemVerliog system function evaluated before synthesis

[SystemVerilog] Source Code: time_base_gen.sv

q = f time-base



- The following time base generation scheme is based on a **modulo-N counter** structure. The modulo N-value is provided as a module parameter
- The counter bit-width is derived using the Verilog $clog2 system function (binary logarithm).
- **We are specifying the counting interval, not the bit-width of the counter** ...