Design Verification

Today's Agenda

Intended topics for today's session

- Introduction to SystemVerilog – Basic Testbench Design
- Testbench based Functional Design Simulation using Intel Quartus Prime Lite, SystemVerilog HDL and Mentor ModelSim
- Live Demonstration

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

Design Verification

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 – Part#4

International Master of Science in Electrical Engineering

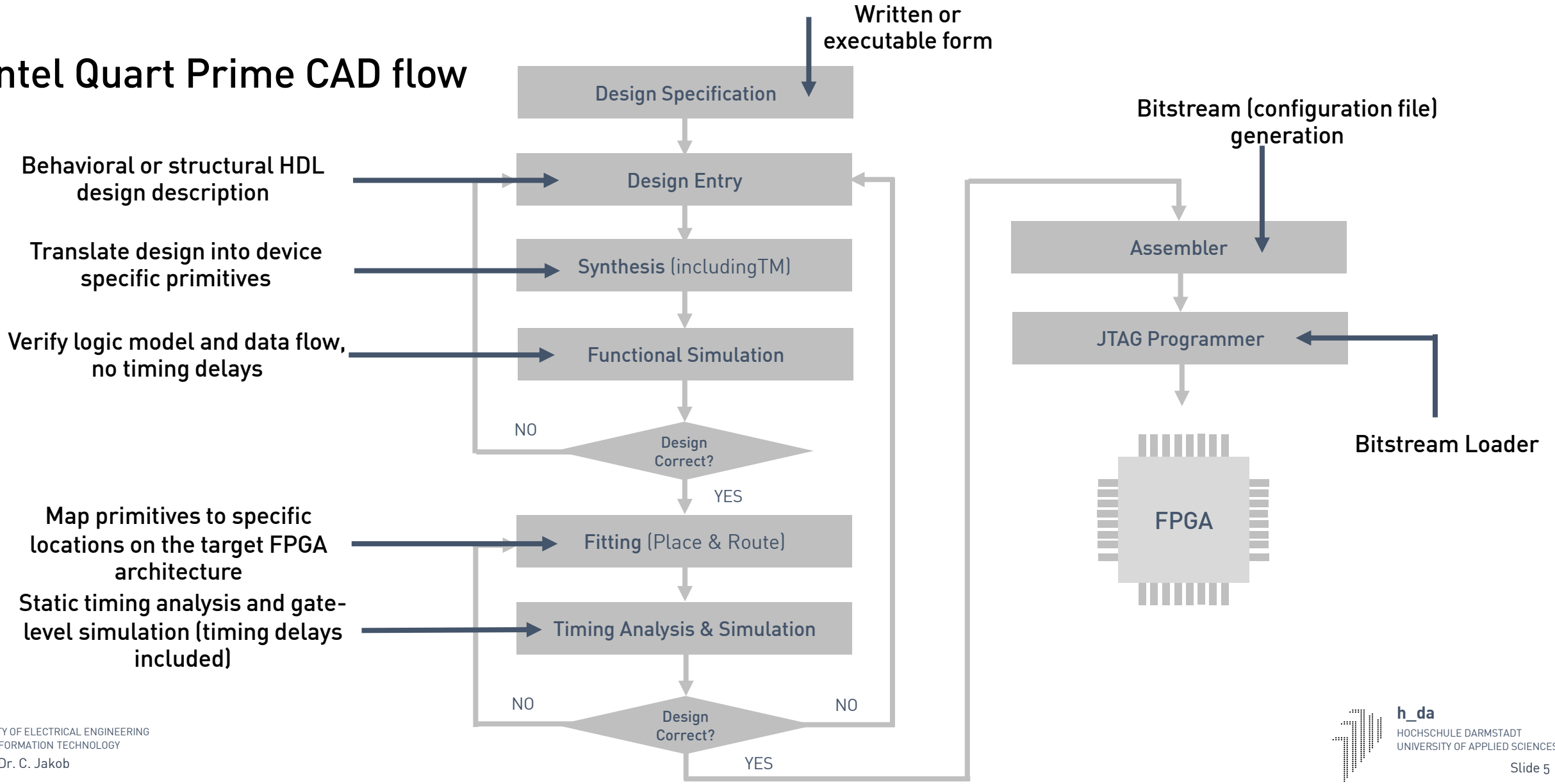
Prof. Dr. C. Jakob

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

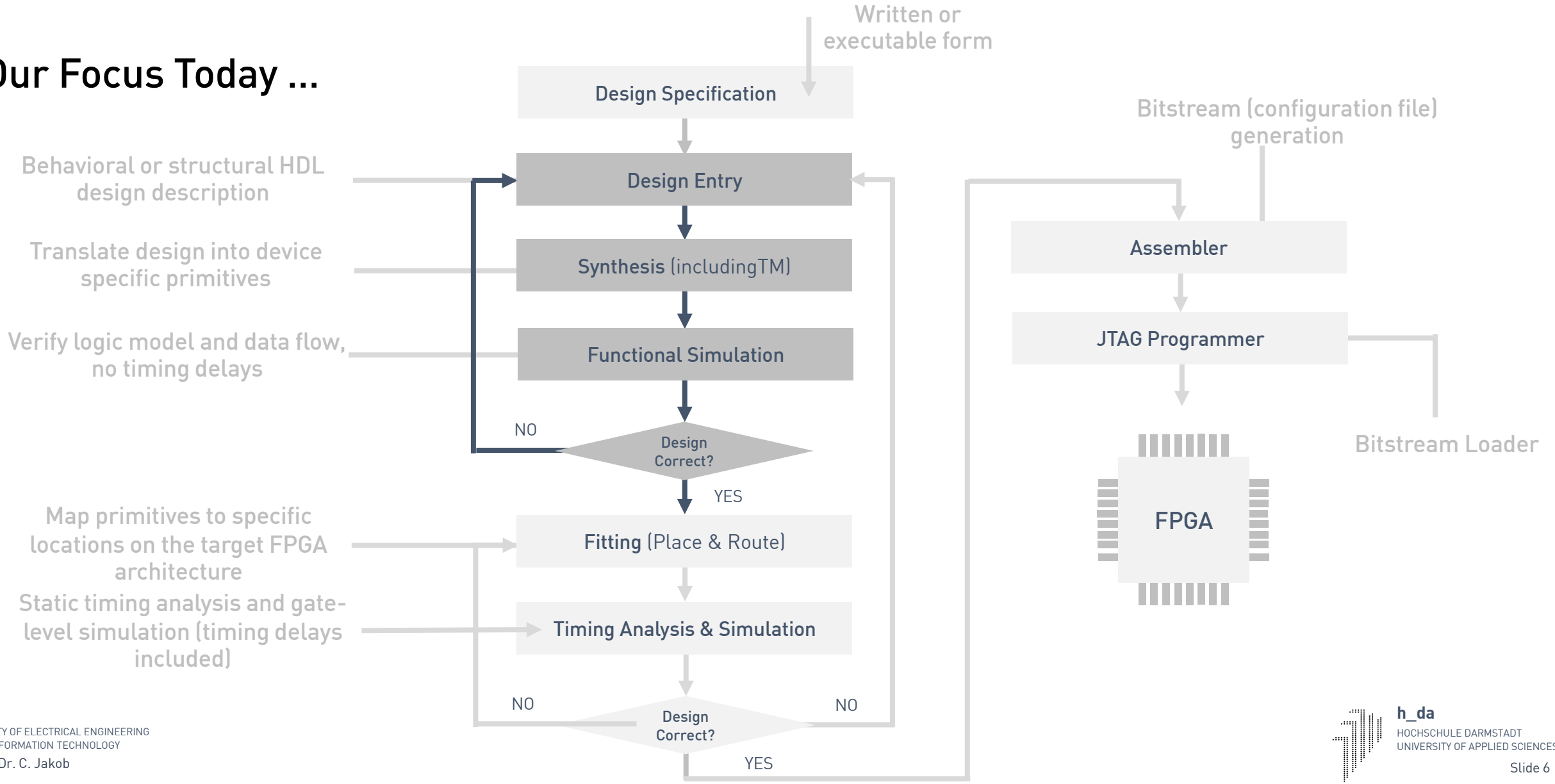
Motivation - How Do You Know That A Circuit Works?

- You have written the SystemVerilog code for a dedicated circuit …
  - Does it work correctly?
  - Even if the syntax is correct, does it do what you expect it to do?
  - Trial and error in-system can be costly, you need to ‘test’ your circuit in advance
- As digital systems become more complex, it becomes increasingly important to verify the functionality of a design before implementing it in a system.
- Before time is spent on fitting a design, the RTL description (or the synthesized netlist) is simulated to assure correct functionality

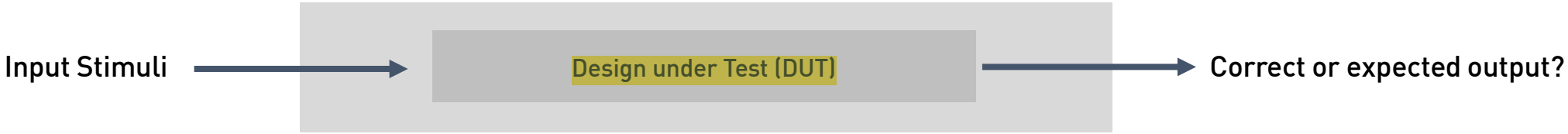
Intel Quart Prime CAD flow



Our Focus Today ...



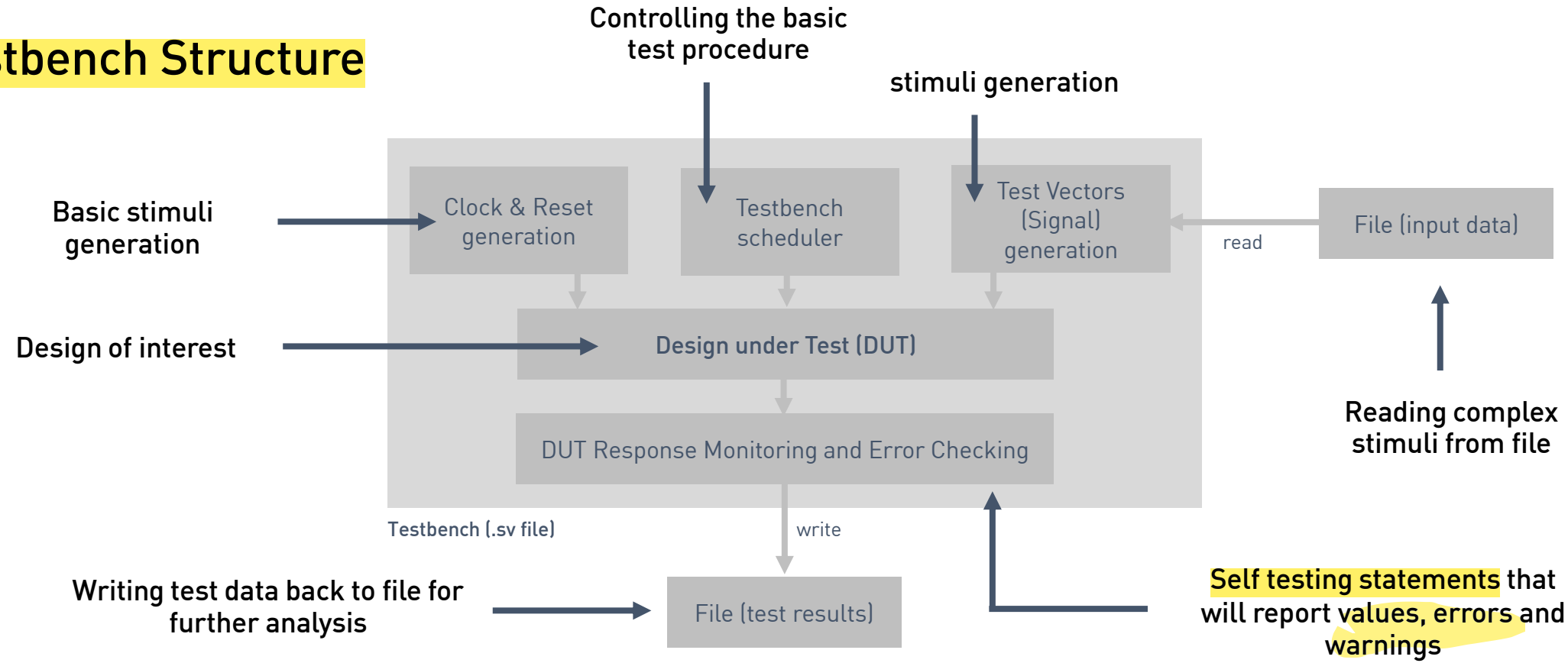
The Idea Behind a Testbench ...



Notes

- Using a computer simulator to test your circuit, you instantiate your design, you supply the circuit with some inputs and you finally observe the respective output behavior ...
- Does it return the "correct" outputs?

Testbench Structure



A **testbench** is a program written for the purpose of exercising and verifying the functional correctness of a hardware model (DUT) during simulation

Basic Testbench Design using SystemVerilog

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 simple Time Base Generation Scheme ...

```
1. module time_base_generator #(parameter CLK_CYCLES = 500000)(
2.    input logic clk, reset_n,
3.    output logic q
4.    );
5.
6.    localparam BITWIDTH = $clog2(CLK_CYCLES);
7.
8.    logic [BITWIDTH-1:0] time_base = 0;
9.    always_ff@(posedge clk)
10.       if(reset_n == 1`b0)
11.          time_base <= '0;
12.       else
13.          time_base <= (time_base + 1'b1) % CLK_CYCLES;
14.
15. assign q = (time_base == (CLK_CYCLES - 1'b1)) ? 1'b1 : 1'b0;
16.
17. endmodule
```

1/1

[SystemVerilog] Source Code: time_base_generator.sv

- 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 SystemVerilog $clog2 system function (binary logarithm)

Corresponding SystemVerilog Testbench ...

1/2

```
1.`timescale 1ns/1ps
2.`define HALF_CLOCK_PERIOD    10
3.`define RESET_PERIOD         200
4.`define SIM_DURATION         50000
5.
6. module time_base_generation_tb();
7.    logic tb_q; logic tb_q;
8.    // ### clock generation process ...
9.    logic tb_local_clock = 0;
10.   initial
11.      begin: clock_generation_process
12.         tb_local_clock = 0;
13.         forever
14.            #`HALF_CLOCK_PERIOD tb_local_clock = ~tb_local_clock;
15.   end
                  10
```

[SystemVerilog] Source Code: time_base_generator_tb.sv

Sets up the time scale and as well as the operating precision for a testbench module. A unit delay therefore causes a nanosecond delay

- The statement has the following structure:

  ```
  `timescale time_unit/time_precision
  ```

- Example:

  ```
  assign reset_n = 1'b0;
  #20
  assign reset_n = 1'b1;
  ```

The resolution time is the smallest possible time unit that can be resolved by the simulator.

Example: If a pulse is smaller than the resolution time it won't get sampled by the simulator.

Corresponding SystemVerilog Testbench ...

1/2

```
1.`timescale 1ns/1ps
2.`define HALF_CLOCK_PERIOD    10
3.`define RESET_PERIOD         200
4.`define SIM_DURATION         50000
5.
6. module time_base_generation_tb();
7.    logic tb_q;
8.    // ### clock generation process ...
9.    logic tb_local_clock = 0;
10.   initial
11.      begin: clock_generation_process
12.         tb_local_clock = 0;
13.         forever
14.            #`HALF_CLOCK_PERIOD tb_local_clock = ~tb_local_clock;
15.   end
```

[SystemVerilog] Source Code: time_base_generator_tb.sv

Define section. Used to improve the readability of the testbench module ...

Corresponding SystemVerilog Testbench ...

1/2

```
1.`timescale 1ns/1ps
2.`define HALF_CLOCK_PERIOD    10
3.`define RESET_PERIOD         200
4.`define SIM_DURATION         50000
5.
6. module time_base_generation_tb();
7.    logic tb_q;
8.    // ### clock generation process ...
9.    logic tb_local_clock = 0;
10.   initial
11.     begin: clock_generation_process
12.        tb_local_clock = 0;
13.        forever
14.           #`HALF_CLOCK_PERIOD tb_local_clock = ~tb_local_clock;
15.   end
```

A testbench usually does not have any ports!

[SystemVerilog] Source Code: time_base_generator_tb.sv

Corresponding SystemVerilog Testbench ...

1/2

```
1.`timescale 1ns/1ps
2.`define HALF_CLOCK_PERIOD    10
3.`define RESET_PERIOD         200
4.`define SIM_DURATION         50000
5.
6. module time_base_generation_tb();
7.    logic tb_q;
8.    // ### clock generation process ...
9.    logic tb_local_clock = 0;
10.   initial
11.     begin: clock_generation_process
12.        tb_local_clock = 0;
13.        forever
14.           #`HALF_CLOCK_PERIOD tb_local_clock = ~tb_local_clock;
15.   end
```

[SystemVerilog] Source Code: time_base_generator_tb.sv

Internal signals, which are monitoring the response of the DUT

- The signal **tb_q** represents the output signal of the DUT. This signal can be printed to the screen or can be captured in a waveform

Corresponding SystemVerilog Testbench ...

1/2

```
1.`timescale 1ns/1ps
2.`define HALF_CLOCK_PERIOD    10
3.`define RESET_PERIOD         200
4.`define SIM_DURATION         50000
5.
6. module time_base_generation_tb();
7.    logic tb_q;
8.    // ### clock generation process ...
9.    logic tb_local_clock = 0;
10.   initial
11.     begin: clock_generation_process
12.       tb_local_clock = 0;
13.       forever
14.         #`HALF_CLOCK_PERIOD tb_local_clock = ~tb_local_clock;
15.   end
```

[SystemVerilog] Source Code: time_base_generator_tb.sv

- Initial blocks start executing at simulation time 0. Each line within an initial block is sequentially executed until a delay is reached!
- When a delay is reached, the execution of this block waits until the delay time has passed and then picks up execution again ...
- Multiple initial blocks are running concurrently

Internal signals (clock signal), which is used to drive the DUT (reg type). When simulation starts it's important that all DUT driving signals are initialized!

Corresponding SystemVerilog Testbench ...

2/2

```
16.  // ### testbench scheduler ...
17.  logic tb_local_reset_n = 0;
18.
19.  initial
20.    begin: reset_generation_process
21.      $display ("Simulation starts ...");
22.      #`RESET_PERIOD tb_local_reset_n = 1'b1;
23.      #`SIM_DURATION
24.      $stop();
25.    end
26.    // ### design under test ...
27.  time_base_generator #(.CLK_CYCLES(1025)) inst_0 (
28.                        .clk(tb_local_clock),
29.                        .reset_n(tb_local_reset_n),
30.                        .q(tb_q)
31.                       );
32. endmodule
```

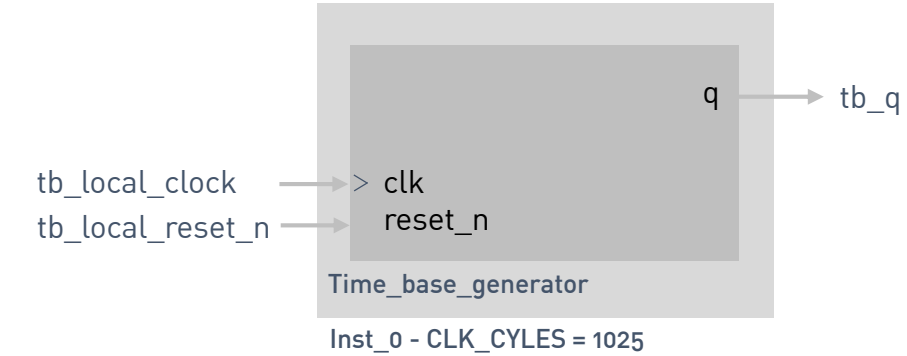
Testbench scheduler using system functions

[SystemVerilog] Source Code: time_base_generator_tb.sv

Corresponding SystemVerilog Testbench ...

2/2

```
16.  // ### testbench scheduler ...
17.  logic tb_local_reset_n = 0;
18.
19.  initial
20.    begin: reset_generation_process
21.       $display ("Simulation starts ...");
22.       #`RESET_PERIOD tb_local_reset_n = 1'b1;
23.       #`SIM_DURATION
24.       $stop();
25.    end
26.    // ### design under test ...
27.  time_base_generator #(.CLK_CYCLES(1025)) inst_0 (
28.                        .clk(tb_local_clock),
29.                        .reset_n(tb_local_reset_n),
30.                        .q(tb_q)
31.                       );
32. endmodule
```



Instantiated design under test (DUT). The DUT can be a behavioral or gate level representation of a design

[SystemVerilog] Source Code: time_base_generator_tb.sv

Basic Testbench Design using SystemVerilog

Introduction to SystemVerilog

SystemVerilog System Tasks and Functions

- A system tasks performs an operation, and does not have a return value.
- A system function calculates a value and returns that value. Some system functions perform an operation like a task, an return a pass or fail status value.
- Most system functions or task are for simulation only, and do not represent actual gate-level behavior.
- Simulation specific tasks are ignored by the synthesis compiler.

Simulation Control Tasks

```
1. $stop(arg)          // halts the simulator and puts it in the interactive mode
2.                     // where the user can enter commands
```

Note

- The `$stop()` system task is used to suspend the actual simulation. When invoked it suspends simulation, prints simulation time and prints location. An optional argument can determine the type of printed message:

```
3. // 0 - No message
4. // 1 - Simulation time and location
5. // 2 - Simulation time, location, memory consumption and CPU time used
```

- The `$finish()` system task ends the simulation, exits the simulator and passes control to the operating system. It can be invoked with the same arguments as `$stop()` and has the same default value.

```
6.  $finish(arg)       // exists the simulator
```

- The `$reset()` system task sets the simulator back to its "Time 0" state, i.e. at the beginning of simulation.

```
7.  $reset()           // resets the simulation time back to '0'
```

Simulation Time Tasks

```
1. $time()
```

Note

- This particular system task provides access to the current simulation time. When `$time()` is called, it returns the current time as a 64-bit integer value. Please note that this value is scaled to the `` `timescale `` unit.

```
2. $stime()
```

- The `$stime()` system function returns the current time as a 32-bit unsigned integer value. If the current simulation time is too large and the value does not fit in 32 bits, the function only returns the 32 low order bits of the value. The returned value is also scaled to the `` `timescale ``. unit.

```
3. $realtime()
```

- The `$realtime()` system function returns a value as a real number. As with the other time tasks, the returned value is scaled to the respective `` `timescale `` value.
- Store the return values in corresponding variables. Use the `$display()` system function to show print their values to the console.

Formatted Write to Display

```
1. $display("<format>", exp1, exp2, ...);
```

Note

- This system task is most often used to display debug messages in the console. It adds a newline character at the end of each output. Possible format strings:

```
2. // %b %B binary
3. // %c %C ASCII character (low 8 bits)
4. // %d %D decimal  %0d for minimum width field
5. // %e %E E floating point format %15.7E
6. // %f %F F format floating point %9.7F
7. // %h %H hexadecimal
8. // %s %S string, 8 bits per character,
9. // %t %T simulation time, expression is  $time
10. // \n newline
11. // \t tab
12. // \\ backslash
12. // \” quote
13. // %% percent
```

More Tasks for Display Writing

```
14. $write      // same as $display except no automatic insertion of newline
15. $strobe     // same as $display except waits until all events finished
16. $monitor    // same as $display except only displays if an expression changes
17. $monitoron  // only one $monitor may be active at ant time,
18. $monitoroff // turn current $monitor off
19. $displayb   // same as $display using binary as default format
20. $writeb     // same as $write using binary as default format
21. $strobeb    // same as $strobe using binary as default format
22. $monitorb   // same as $monitor using binary as default format
23. $displayo   // same as $display using octal as default format
24. $writeo     // same as $write using octal as default format
25. $strobeo    // same as $strobe using octal as default format
26. $monitoro   // same as $monitor using octal as default format
27. $displayh   // same as $display using hexadecimal as default format
28. $writeh     // same as $write using hexadecimal as default format
29. $strobeh    // same as $strobe using hexadecimal as default format
30. $monitorh   // same as $monitor using hexadecimal as default format
```

Reference: Verilog Language Reference Guide (http://verilog.renerta.com/)

File Opening and Closing Commands

```
1. integer fd;
2. fd = $fopen("<file name>", file_type);
```

Note

- *$***fopen**() opens a file for accessing and returns a 32-bit (integer) file handle.
- The `file_type` argument is a character string that indicates the way t the file is accessed.

```
3. // "r" or "rb"   - open text or binary file for reading
4. // "w" or "wb"   - open text or binary file for writing (file data deleted)
5. // "a" or "ab"   - open text or binary file for appending
6. // "r+" or "rb+" - open text or binary file for reading and writing
```

- The following task closes the file specified by the respective file descriptor.

```
7. $fclose(fd);
```

while the following system task closes all opened files …

```
8. $fclose();
```

Using File I/O in SystemVerilog Testbench ...

1/1

```
11. // ### (active low) reset generation process ...
12. logic tb_local_reset_n = 1'b0;
13.
14. integer testfile, i;
15. initial
16.    begin: reset_generation_process
17.       testfile = $fopen("C:/projects/dsp/corr_data.txt","w");
18.       #`RESET_PERIOD tb_local_reset_n = 1'b1;
19.       #`SIM_DURATION
20.       for (i = 0; i < 64; i = i+1)
21.          begin
22.             $fwrite(testfile,"%d \n",tb_q[i]); //tb_q[i]);
23.          end
24.       $fclose(testfile);
25.       $stop();
26.     end
```

[SystemVerilog] Source Code Excerpt: SystemVerilog Testbench using File I/O

Reading Data File Commands

```
9.  integer char;
10. char = $fgetc(fd);
```

Note

- The system function `$fgetc()` reads and returns a single byte (character) from the file specified with the respective file descriptor.
- It returns the integer `EOF` (`-1`) if the end of the file has been reached. Therefore, it is meaningful to use a variable larger than 8-bit to distinguish between `EOF` and `0xFF`.

```
11.`define EOF 32'hFFFF_FFFF
12. ...
13. integer char;
14. ...
15. char = $fgetc(fd);
16. while(tmp_char != `EOF)
17.     begin
18.         ...
```

- A valid value will be zero extended to the width of the variable, whereas `-1` will be sign extended.

Reading Data File Commands

```
19. string line;
20. fgets(line,fd);
```

Note

- The system function `$fgets()` reads a string of characters into a dedicated string variable.
- Reading stops by reaching a newline character or the end of file.

```
21. integer cnt;
22. cnt = fgets(line,fd);
```

- It returns the number of characters read if successful whereas it returns 0 if unsuccessful

```
23. while(!$feof(fd)) begin
24.   cnt = $fgets(line, fd);
25.   $display("line: %s",line);
26.   $display("cnt: %",cnt);
27. end
```

- The system task `$feof()` returns true when the end of the file is reached, othersise 0.

Reading Data File Commands

```
28. $fscanf(fd, format, args);
```

Note

- The system function `$fscanf()` reads formatted text from a file. It reads characters from the file specified by the file descriptor, interprets them according to a given format, and stores the results in its arguments. It uses C-style formatting. Moreover, it returns the number of successfully assigned items.

```
29. integer a, b;
30. logic [7:0] tmp;
31.
31. while(fscanf(fd,"%d %d %b \n", a, b, tmp) == 3) begin
32.     $display("a = %d % b = d tmp = %b", a, b, tmp);
33. end
```

- If there are insufficient arguments for the respective format, the behavior is undefined.

Some Additional File Commands

```
1. integer position;
2. position = $ftell(fd);
```

Note

- Returns the byte position (offset from the beginning of file) of the next character to be read from or written to the respective file.

```
3. $fseek(fd, < position >, < operation >);
```

- Random file access is possible using the `$fseek()` system function. Normally, files and streams are read and written sequentially. By using `$fseek()`, it is possible to get to any position in the file. The offset argument represents the signed distance from the actual operation point which is specified as follows:

```
4. // 0: Beginning of the file;
5. // 1: Current byte location
6. // 2: End of file
```

- This function is most often used in conjunction with the `$ftell()` system function. Here, `$ftell()` is used to get the current file position to which we would like go back at a later stage.

Some Additional File Commands

```
1. module file_read_test();
2.   integer fd, test_char, status;
3.   ...
4.
5.    initial begin
6.        fd = $fopen("data.rbf", "rb");
7.        status = $fseek(fd,4,0);
8.        assert(status);
9.        do begin
10.           test_char = $fgetc(fd);
11.           $display("test char: %h",file_char);
12.       end
13.       while(test_char != -1)
14.   end
15. endmodule
```

Note

- Here, the system function `$fseek()` is used to skip the first four bytes before start reading.
- The expression `assert(status)` represents an immediate assertion that generates an error if status is equal to -1

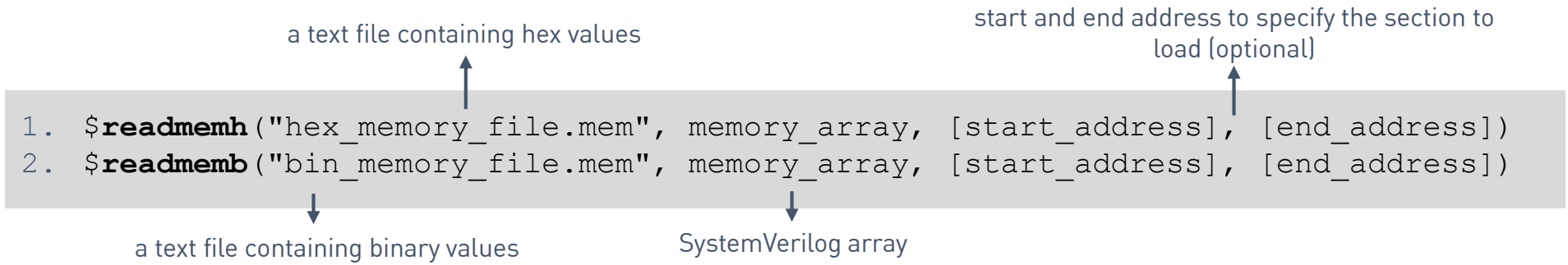
File Output Commands

```
1.  $fdisplay(fd,args)
2.  $fwrite(fd,args)
3.  $fmonitor(fd,args)
4.  $fstrobe(fd,args)
```

Note

- The upper system tasks as well as the respective variants (binary, hex, octal) work like the corresponding display system tasks, only that they are writing to a file instead of the console.

More SystemVerilog Tasks



Note

- SystemVerilog allows you to initialize memory from a text file with either hex or binary values.
- Values might be separated by **whitespace**, **tab** or a **newline**

```
1. // Declare a ROM variable
2. logic [15:0] rom [0:3];
3.
4. initial begin
5.     $readmemh("single_port_rom_init.txt", rom);
6. end
```

```
// this is a comment
DEAD
BEEF
DEAD
BEEF
```

[SystemVerilog] Source Code Excerpt