Electronics Systems (938II)

Lecture 2.5

Building Blocks of Electronic Systems – Simulation with testbench

Luca Crocetti

Simulation

- We have seen how to perform simulations with Modelsim
  - By using the commands
    - force
    - run

- This approach was useful to get familiarity the SV code, the simulation of modules designed in SV, and Modelsim

- However, in the typical design flow of electronic circuits with HDL, the simulation is performed using a testbench

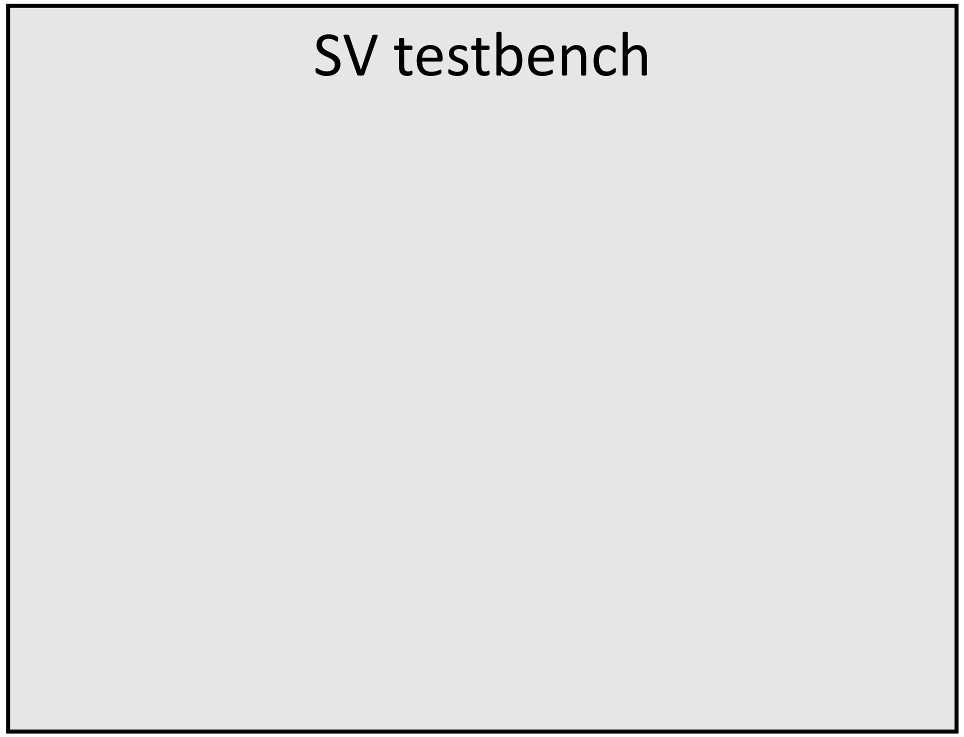
Testbench

- Another ‘module’ without ports used to verify the functionality of the designed circuit(s) by
  - Providing stimuli
  - Checking the output(s)

- The stimuli generation and the output(s) verification are performed with commands that are not-synthesizable
  - Such commands do not describe a realistic circuit, but only describe procedures
  - Or, in other words, such commands cannot be converted in a hardware circuit

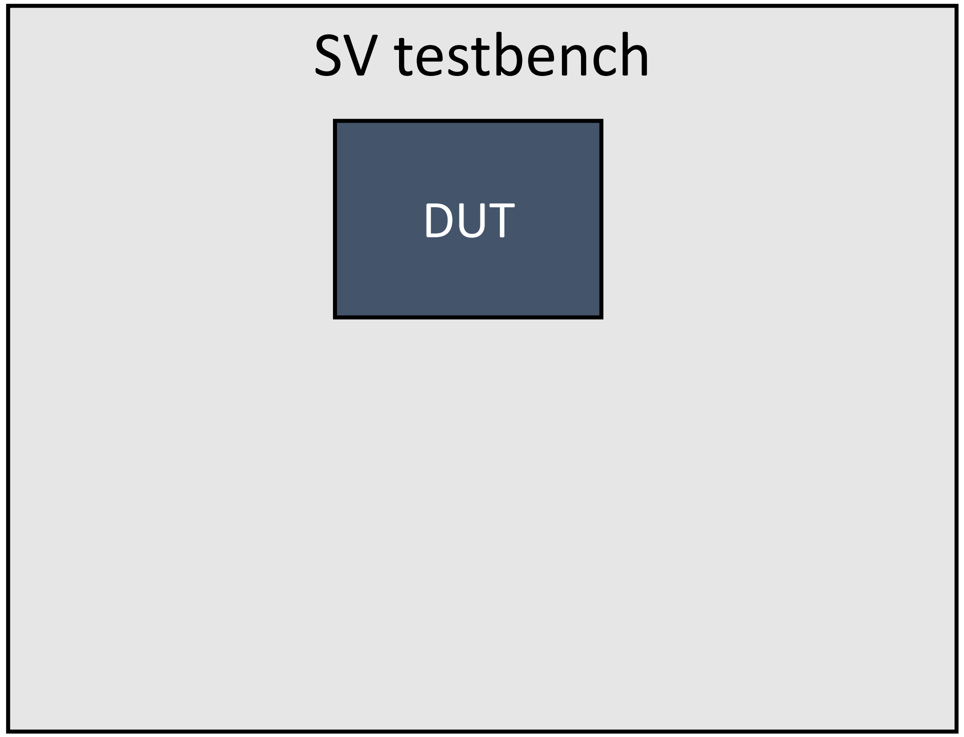
Testbench

- Main components



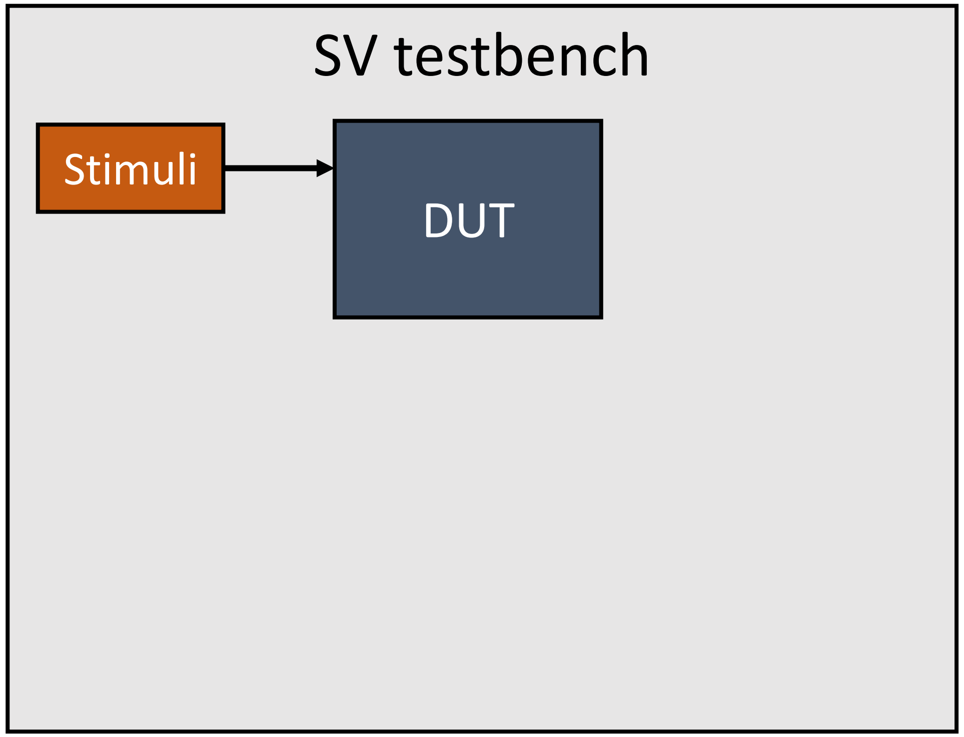
Testbench

- Main components
  - Device Under Test (DUT)
    - I.e., the instance of (the top-level of) your module



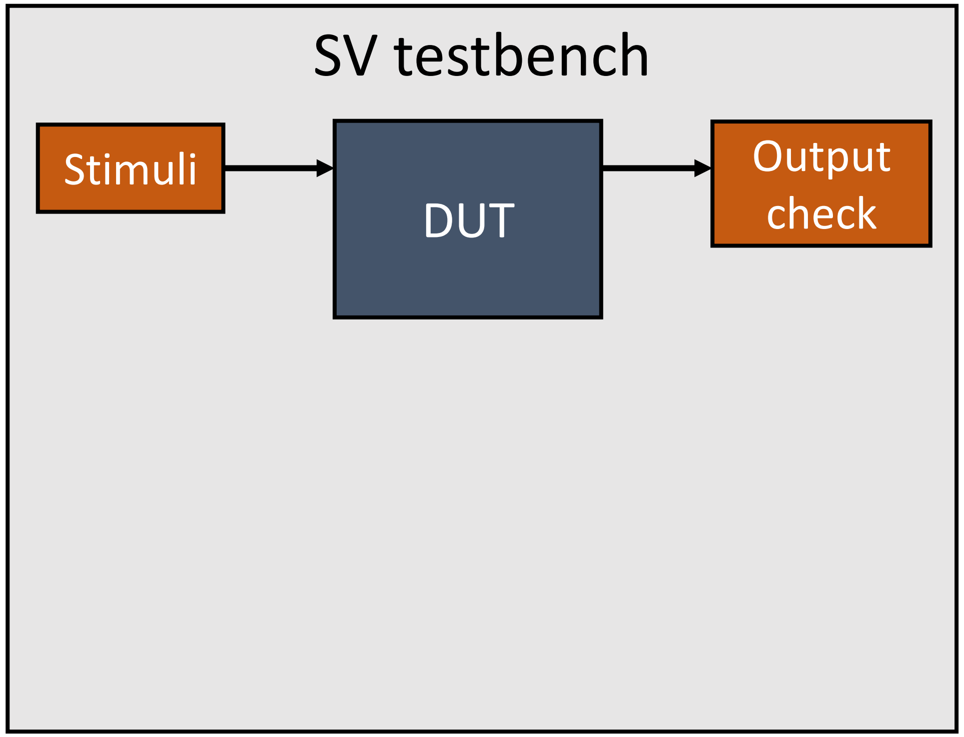
Testbench

- Main components
  - Device Under Test (DUT)
    - I.e., the instance of (the top-level of) your module
  - The stimuli procedure(s)



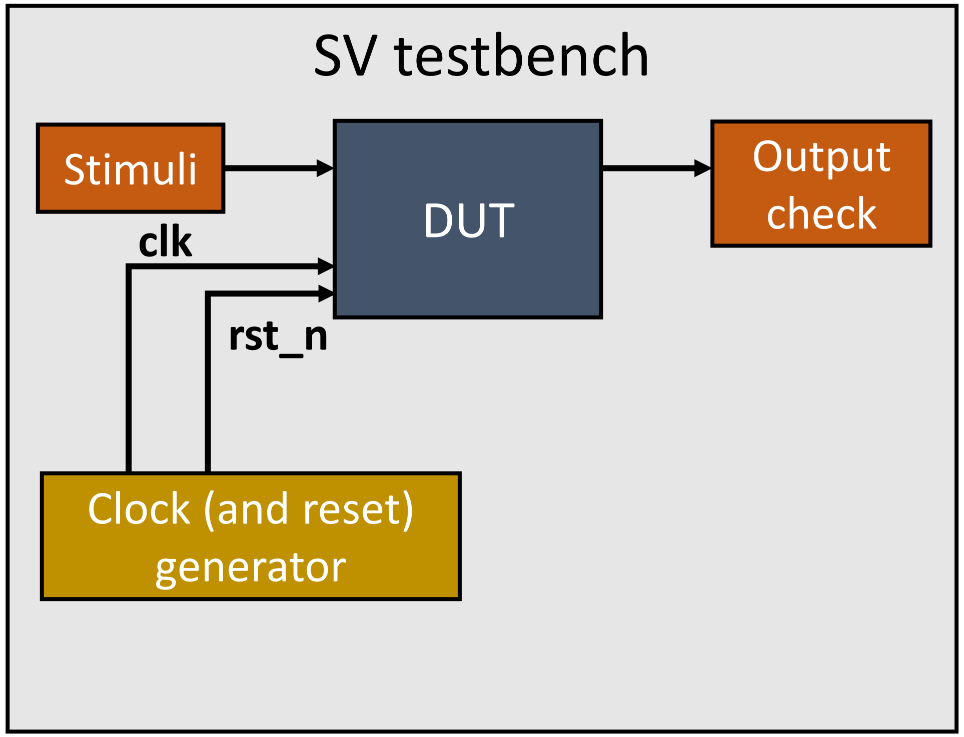
Testbench

- Main components
  - Device Under Test (DUT)
    - I.e., the instance of (the top-level of) your module
  - The stimuli procedure(s)
  - The output(s) check procedure(s)



Testbench

- Main components
  - Device Under Test (DUT)
    - I.e., the instance of (the top-level of) your module
  - The stimuli procedure(s)
  - The output(s) check procedure(s)
  - The clock (and reset) generator
    - In case the DUT is a sequential circuit



Testbench in SystemVerilog

- Main structure

```
module tb;

  reg         clk   = 1'b0;
  reg         rst_n = 1'b0;
  reg  [31:0] data_in;

  wire [31:0] data_out;

  my_module U0 (
    .clk    (clk)
    ,.rst_n (rst_n)
    ,.din   (data_in)
    ,.dout  (data_out)
  );

  initial begin
    // ...
    // ...
    // ...
  end

endmodule
```

Testbench in SystemVerilog

- Main structure
  - Module without ports

```
module tb;

  reg          clk   = 1'b0;
  reg          rst_n = 1'b0;
  reg   [31:0] data_in;

  wire  [31:0] data_out;

  my_module U0 (
     .clk    (clk)
    ,.rst_n  (rst_n)
    ,.din    (data_in)
    ,.dout   (data_out)
  );

  initial begin
    // ...
    // ...
    // ...
  end

endmodule
```

Testbench in SystemVerilog

- Main structure
  - Module without ports
  - Declaration of signals for stimuli and probes

```
module tb;

  reg          clk    = 1'b0;
  reg          rst_n = 1'b0;
  reg   [31:0] data_in;

  wire  [31:0] data_out;

  my_module U0  (
     .clk    (clk)
    ,.rst_n  (rst_n)
    ,.din    (data_in)
    ,.dout   (data_out)
  );

  initial begin
    // ...
    // ...
    // ...
  end

endmodule
```

Testbench in SystemVerilog

- Main structure
  - Module without ports
  - Declaration of signals for stimuli and probes
  - Instance of module to be tested (DUT)

```
module tb;

  reg        clk    = 1'b0;
  reg        rst_n  = 1'b0;
  reg [31:0] data_in;

  wire [31:0] data_out;

  my_module U0 (
     .clk    (clk)
    ,.rst_n  (rst_n)
    ,.din    (data_in)
    ,.dout   (data_out)
  );

  initial begin
    // ...
    // ...
    // ...
  end

endmodule
```

Testbench in SystemVerilog

- Main structure
  - Module without ports
  - Declaration of signals for stimuli and probes
  - Instance of module to be tested
  - Procedural blocks for driving stimuli and checking outputs

```
module tb;

  reg          clk   = 1'b0;
  reg          rst_n = 1'b0;
  reg   [31:0] data_in;

  wire  [31:0] data_out;

  my_module U0 (
     .clk    (clk)
    ,.rst_n  (rst_n)
    ,.din    (data_in)
    ,.dout   (data_out)
  );

  initial begin
    // ...
    // ...
    // ...
  end

endmodule
```

Testbench in SystemVerilog

- Module instance – example

```
module_name instance_name (
    clk
   ,rst_n
   ,din
   ,done
   ,dout
);
```

```
module_name instance_name (
    .clk        (clk)
   ,.rst_n      (rst_n)
   ,.data_out   (dout)
   ,.done       (done)
   ,.data_in    (din)
);
```

Testbench in SystemVerilog

- Module instance – example
  - Module name
    - The same used for the declaration of the module (in the corresponding SV file)

```
module_name instance_name (
   clk
  ,rst_n
  ,din
  ,done
  ,dout
);
```

```
module_name instance_name (
   .clk        (clk)
  ,.rst_n      (rst_n)
  ,.data_out   (dout)
  ,.done       (done)
  ,.data_in    (din)
);
```

Testbench in SystemVerilog

- Module instance – example
  - Instance name
    - Any arbitrary single word, e.g., DUT
    - The same module name can be used

```
module_name instance_name (
    clk
   ,rst_n
   ,din
   ,done
   ,dout
);
```

```
module_name instance_name (
    .clk        (clk)
  ,.rst_n       (rst_n)
  ,.data_out    (dout)
  ,.done        (done)
  ,.data_in     (din)
);
```

Testbench in SystemVerilog

- Module instance – example
  - Delimiters

```
module_name instance_name (
    clk
   ,rst_n
   ,din
   ,done
   ,dout
);
```

```
module_name instance_name (
    .clk        (clk)
   ,.rst_n      (rst_n)
   ,.data_out   (dout)
   ,.done       (done)
   ,.data_in    (din)
);
```

Testbench in SystemVerilog

- Module instance – example
  - Ports connection

```
module_name instance_name (
   clk
  ,rst_n
  ,din
  ,done
  ,dout
);
```

```
module_name instance_name (
   .clk         (clk)
  ,.rst_n       (rst_n)
  ,.data_out    (dout)
  ,.done        (done)
  ,.data_in     (din)
);
```

Testbench in SystemVerilog

- Module instance – example
  - Ports connection
    - By ordered list: the same ports order used in the module declaration

```
module_name instance_name (
     clk
    ,rst_n
    ,din
    ,done
    ,dout
);
```

```
module_name instance_name (
    .clk        (clk)
  ,.rst_n       (rst_n)
  ,.data_out    (dout)
  ,.done        (done)
  ,.data_in     (din)
);
```

Testbench in SystemVerilog

- Module instance – example
  - Ports connection
    - By port name: **.<port name> (<signal>)**
      - The order of the ports can be modified

```
module_name instance_name (
    clk
   ,rst_n
   ,din
   ,done
   ,dout
);
```

```
module_name instance_name (
    .clk         (clk)
   ,.rst_n       (rst_n)
   ,.data_out    (dout)
   ,.done        (done)
   ,.data_in     (din)
);
```

Testbench in SystemVerilog

- Module instance – example
  - Ports connection – Pay attention!
    - These are the signals (internal to the testbench) that are connected to the module ports

```
module_name instance_name (
   clk
  ,rst_n
  ,din
  ,done
  ,dout
);
```

```
module_name instance_name (
   .clk        (clk)
  ,.rst_n      (rst_n)
  ,.data_out   (dout)
  ,.done       (done)
  ,.data_in    (din)
);
```

Testbench in SystemVerilog

- Module instance – example
  - Ports connection – Pay attention!
    - These are module ports
      - Only in case of the connection by port name

```
module_name instance_name (
    clk
   ,rst_n
   ,din
   ,done
   ,dout
);
```

```
module_name instance_name (
    .clk          (clk)
   ,.rst_n        (rst_n)
   ,.data_out     (dout)
   ,.done         (done)
   ,.data_in      (din)
);
```

Testbench in SystemVerilog

- Module instance – example
  - Ports connection – In both cases:
    - Connecting signals can have any name
      - The same module port name can be used, but it is not mandatory

```
module_name instance_name (
     clk
    ,rst_n
    ,din
    ,done
    ,dout
);
```

```
module_name instance_name (
     .clk         (clk)
    ,.rst_n       (rst_n)
    ,.data_out    (dout)
    ,.done        (done)
    ,.data_in     (din)
);
```

Testbench in SystemVerilog

- Module instance – example
  - Ports connection – In both cases:
    - Connections must be separated by comma

```
module_name instance_name (
    clk
   ,rst_n
   ,din
   ,done
   ,dout
);
```

```
module_name instance_name (
    .clk        (clk)
   ,.rst_n      (rst_n)
   ,.data_out   (dout)
   ,.done       (done)
   ,.data_in    (din)
);
```

Testbench in SystemVerilog

- Module instance – example
  - Ports connection – In both cases:
    - It is not mandatory, but strictly recommended, that the module port and the connecting signal have the same bit width (otherwise the two vectors are aligned starting from the LSB)

```
module_name instance_name (
    clk
   ,rst_n
   ,din
   ,done
   ,dout
);
```

```
module_name instance_name (
    .clk        (clk)
   ,.rst_n      (rst_n)
   ,.data_out   (dout)
   ,.done       (done)
   ,.data_in    (din)
);
```

Testbench in SystemVerilog

- Module instance – example
  - Ports connection – In both cases:
    - Ports can be left unconnected

```
module_name instance_name (
    clk
   ,rst_n
   ,din
   ,done
   ,dout
);
```

```
module_name instance_name (
    .clk        (clk)
  ,.rst_n       (rst_n)
  ,.data_out    (dout)
  ,.done        (     )
  ,.data_in     (din)
);
```

Testbench in SystemVerilog

- Module instance – example
  - Ports connection – In both cases:
    - As rule of thumb, use **reg** signals to connect to input ports of the module
    - And **wire** signals to connect to the output ports of the module

```
module_name instance_name (
    clk
   ,rst_n
   ,din
   ,done
   ,dout
);
```

```
module_name instance_name (
    .clk        (clk)
   ,.rst_n      (rst_n)
   ,.data_out   (dout)
   ,.done       (done)
   ,.data_in    (din)
);
```

Testbench in SystemVerilog

- Stimuli and probes
  - **Stimuli** are signals that you are going to drive with specific values
    - **reg** type (because assigned inside an initial block – that we will see later)
    - Connected to the **input ports** of the module to be tested
  - **Probes** are used as reading signals to check the values of outputs
    - **wire** type (just a connection)
    - Connected to the **output ports** of the module to be tested

Testbench in SystemVerilog

- Clock generation

```
reg clk = 1'b0;

always #10 clk = !clk;
```

Testbench in SystemVerilog

- Clock generation
  - Declaration of clock signal (clk) as reg

```
reg clk = 1'b0;

always #10 clk = !clk;
```

Testbench in SystemVerilog

- Clock generation
  - Declaration of clock signal (clk) as reg
  - With initialization value
    - Attention: initialization values are not synthesizable, so do not use them in design modules

```
reg clk = 1'b0;

always #10 clk = !clk;
```

Testbench in SystemVerilog

- Clock generation
  - Definition of clock waveform
    - invert signal clk

```
reg clk = 1'b0;

always #10 clk = !clk;
```

Testbench in SystemVerilog

- Clock generation
  - Definition of clock waveform
    - invert signal clk
    - after 10 (time units)

```
reg clk = 1'b0;

always #10 clk = !clk;
```

Testbench in SystemVerilog

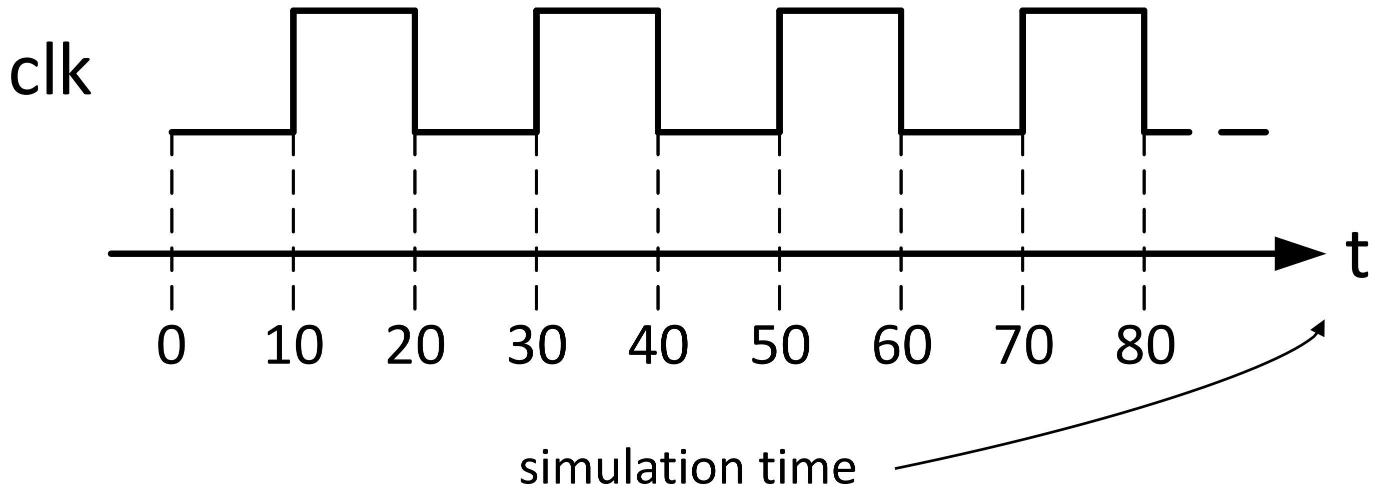
- Clock generation
  - Definition of clock waveform
    - invert signal clk
    - after 10 (time units)
    - always

```
reg clk = 1'b0;

always #10 clk = !clk;
```

Testbench in SystemVerilog

- Clock generation
  - Definition of clock waveform
    - invert signal clk
    - after 10 (time units)
    - always

```
reg clk = 1'b0;

always #10 clk = !clk;
```



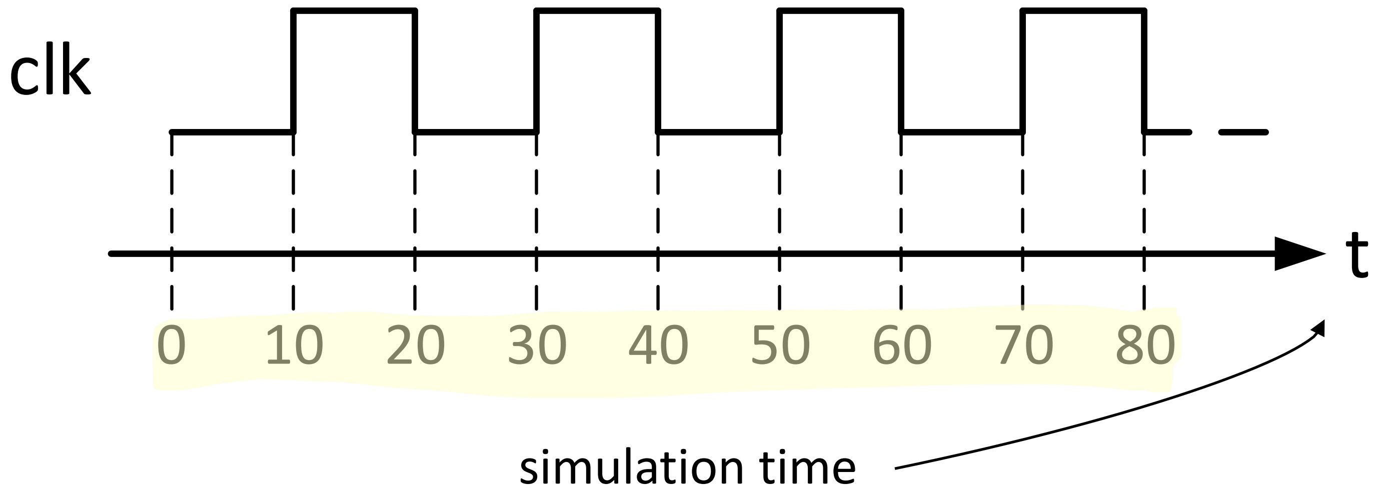
Testbench in SystemVerilog

- Clock generation
  - Definition of clock waveform
    - invert signal clk
    - after 10 (time units)
    - always

```
reg clk = 1'b0;

always #10 clk = !clk;
```

Time unit

- By default, it depends on the tool (Modelsim) settings
  - nanosecond (ns) or picosecond (ps)
- However, you can also explicitly specify with the delay
  - #10ns, #10ps, #10us, …



Testbench in SystemVerilog

- Reset and Power-on Reset (PoR)
  - Typically, a reset condition is imposed at power-up to bring the system/circuit (and all its parts) to a known state
    - Reset state
      - All zeros (typically)
      - Registers and other sequential logic elements
    - It is defined Power-on Reset (PoR)

Testbench in SystemVerilog

- Reset and Power-on Reset (PoR) – active-low reset

```
reg rst_n = 1'b0;

initial #35.4ns rst_n = 1'b1;
```

Testbench in SystemVerilog

- Reset and Power-on Reset (PoR) – active-low reset
  - Declaration of reset signal (rst_n) as reg

```
reg rst_n = 1'b0;

initial #35.4ns rst_n = 1'b1;
```

Testbench in SystemVerilog

- Reset and Power-on Reset (PoR) – active-low reset
  - Declaration of clock signal (clk) as reg
  - With initialization value
    - To impose the PoR, you must specify the active value
      - In this case (active-low), it is 0

```
reg rst_n = 1'b0;

initial #35.4ns rst_n = 1'b1;
```

Testbench in SystemVerilog

- Reset and Power-on Reset (PoR) – active-low reset
  - Definition of reset waveform
    - after 35.4 (time units)

```
reg rst_n = 1'b0;

initial #35.4ns rst_n = 1'b1;
```

Testbench in SystemVerilog

- Reset and Power-on Reset (PoR) – active-low reset
  - Definition of reset waveform
    - after 35.4 (time units)
    - from the beginning (of the simulation)

```
reg rst_n = 1'b0;

initial #35.4ns rst_n = 1'b1;
```

Testbench in SystemVerilog

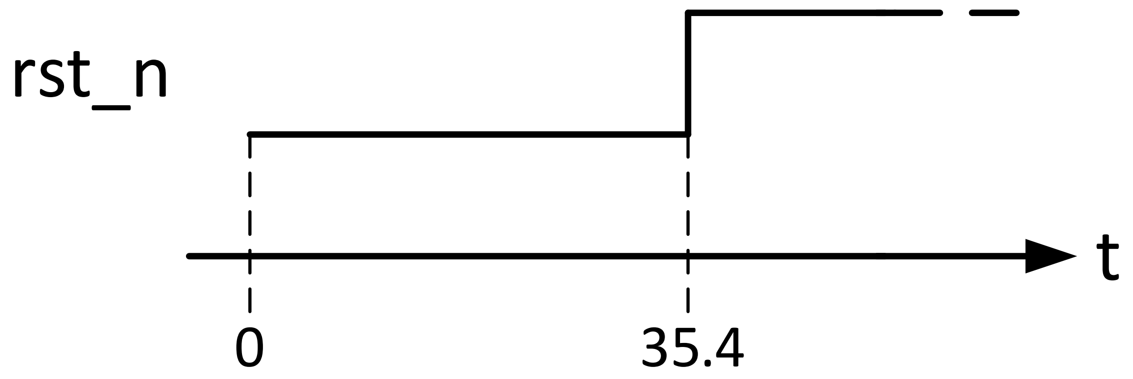
- Reset and Power-on Reset (PoR) – active-low reset
  - Definition of reset waveform
    - after 35.4 (time units)
    - from the beginning (of the simulation)
    - set the signal rst_n to 1 (inactive value)

```
reg rst_n = 1'b0;

initial #35.4ns rst_n = 1'b1;
```

Testbench in SystemVerilog

- Reset and Power-on Reset (PoR) – active-low reset
  - Definition of reset waveform
    - after 35.4 (time units)
    - from the beginning (of the simulation)
    - set the signal rst_n to 1 (inactive value)

```
reg rst_n = 1'b0;

initial #35.4ns rst_n = 1'b1;
```



Testbench in SystemVerilog

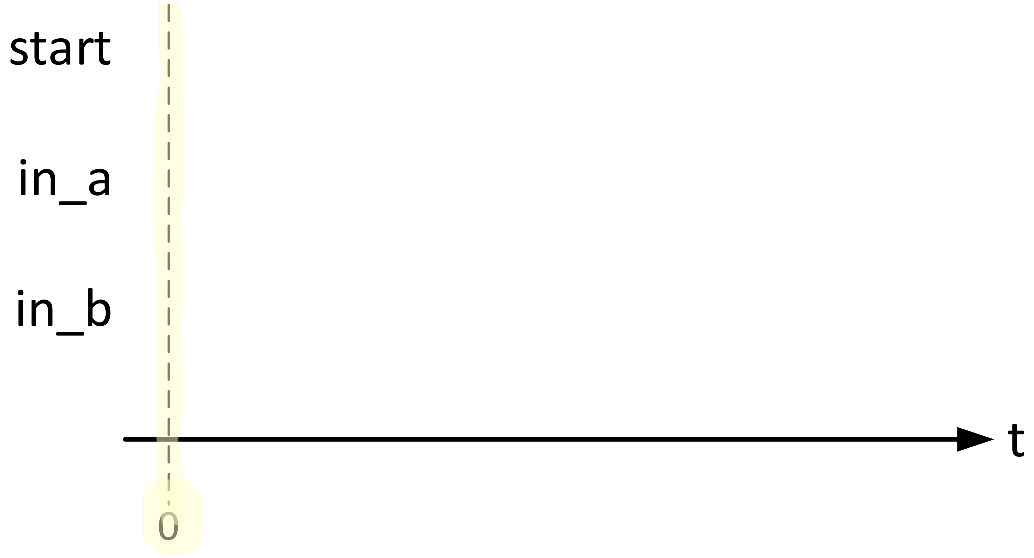
- The **initial** block
  - It is used to drive the stimuli (mainly)
    - The signal used as stimuli must be of **reg** type and assigned with operator **=** (blocking assignment)
  - It is not synthesizable

```
initial begin

  #10 start = 1'b1;

  #13 in_a  = 4'd2;
      in_b  = 4'd3;

  #37 in_a  = 4'd15;
      in_b  = 4'd7;
  //…
end
```

Testbench in SystemVerilog

- The **initial** block
  - It starts at the beginning of the simulation
    - Time 0

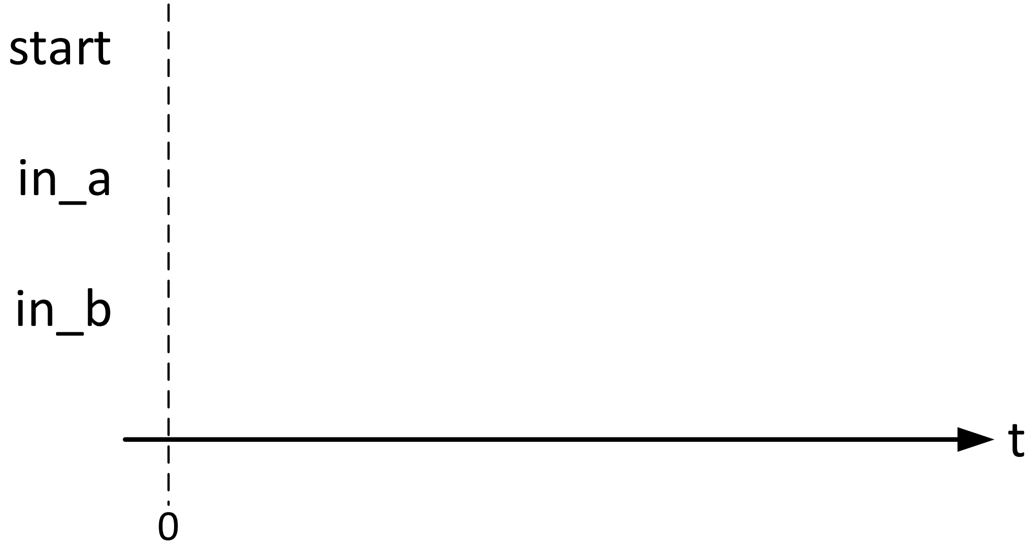
```
initial begin

  #10 start = 1'b1;

  #13 in_a  = 4'd2;
      in_b  = 4'd3;

  #37 in_a  = 4'd15;
      in_b  = 4'd7;
  //…
end
```



Testbench in SystemVerilog

- The **initial** block
  - begin – end delimiters if more than one statement

```
initial begin

  #10 start = 1'b1;

  #13 in_a  = 4'd2;
      in_b  = 4'd3;

  #37 in_a  = 4'd15;
      in_b  = 4'd7;
  //…
end
```



Testbench in SystemVerilog

- The **initial** block
  - Stimuli assignment – example

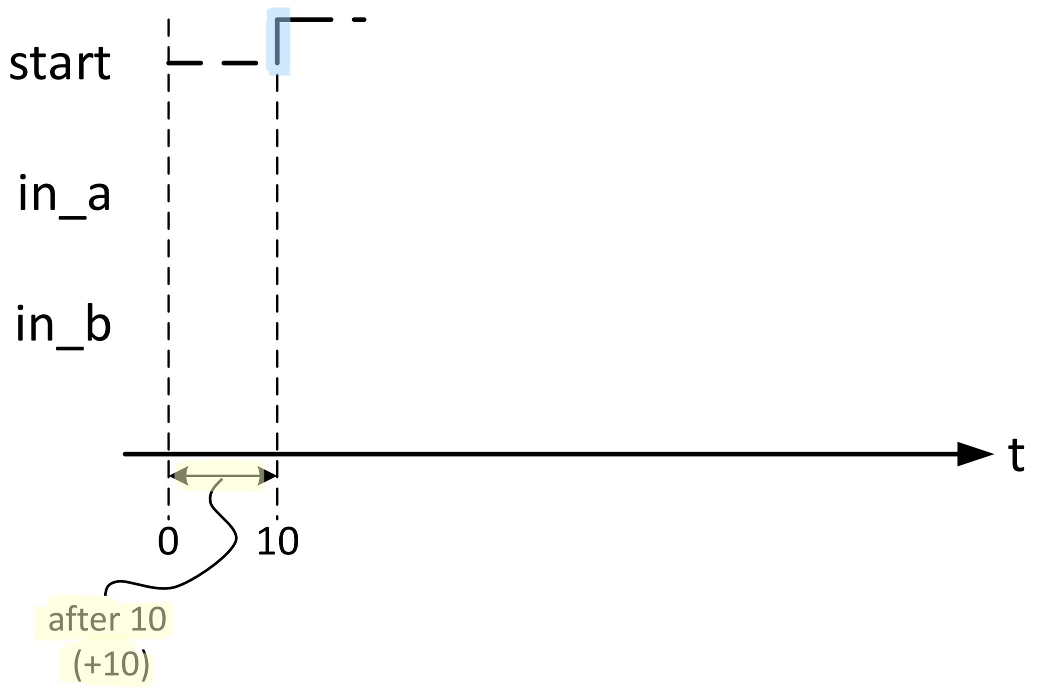
```
initial begin

  #10 start = 1'b1;

  #13 in_a  = 4'd2;
      in_b  = 4'd3;

  #37 in_a  = 4'd15;
      in_b  = 4'd7;
  //…
end
```

start
in_a
in_b
t
0

Testbench in SystemVerilog

- The **initial** block
  - Stimuli assignment – example

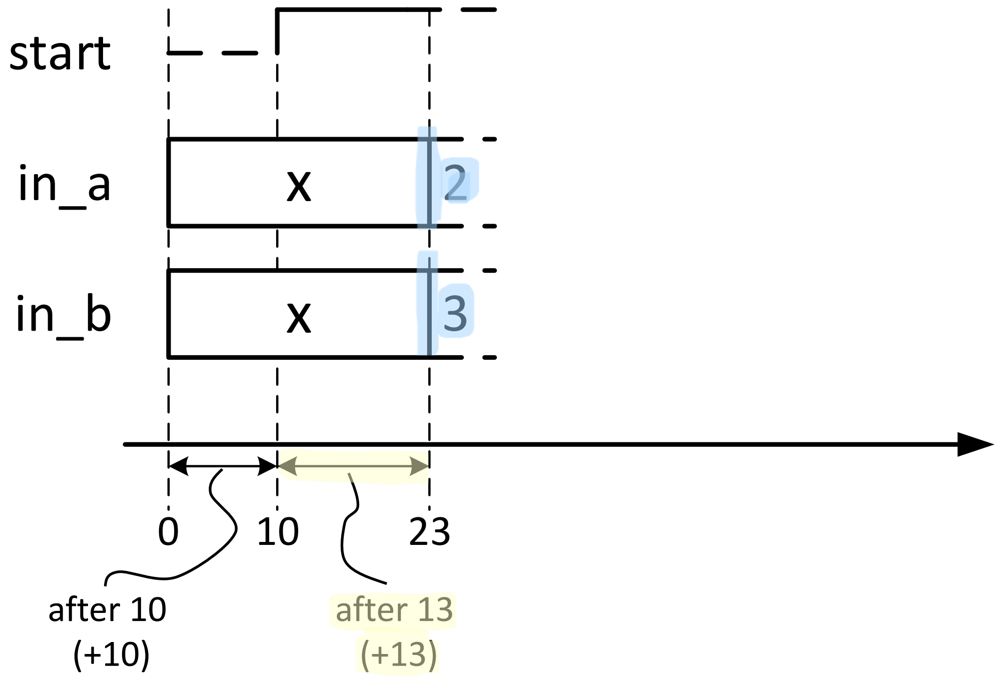
```
initial begin

  #10 start = 1'b1;

  #13 in_a  = 4'd2;
      in_b  = 4'd3;

  #37 in_a  = 4'd15;
      in_b  = 4'd7;
  //…
end
```



Testbench in SystemVerilog

- The **initial** block
  - Stimuli assignment – example

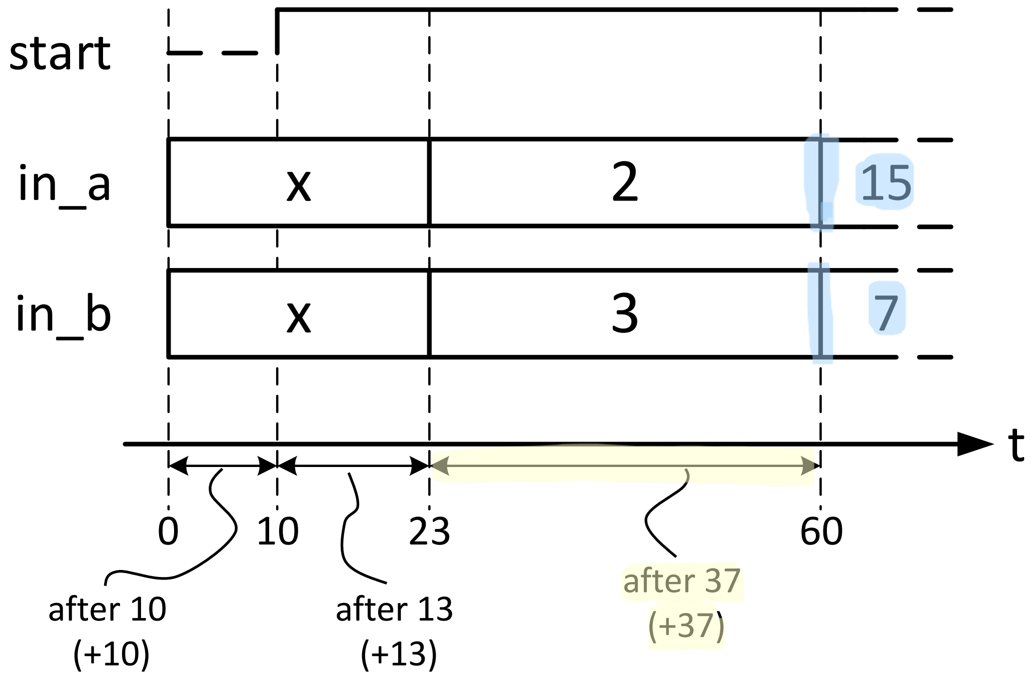
```
initial begin

  #10 start = 1'b1;

  #13 in_a  = 4'd2;
      in_b  = 4'd3;

  #37 in_a  = 4'd15;
      in_b  = 4'd7;
  //…
end
```



Testbench in SystemVerilog

- The **initial** block
  - Stimuli assignment – example

```
initial begin

  #10 start = 1'b1;

  #13 in_a  = 4'd2;
      in_b  = 4'd3;

  #37 in_a  = 4'd15;
      in_b  = 4'd7;
  //…
end
```



Testbench in SystemVerilog

- The **initial** block
  - A useful command for stimuli assignment: **$random**() / **$urandom**()
    - **$random**() → returns a signed 32-bit value
    - **$urandom**() → returns an unsigned 32-bit value

```
initial begin

  // …

   #15 a = $urandom();

  // …


end
```

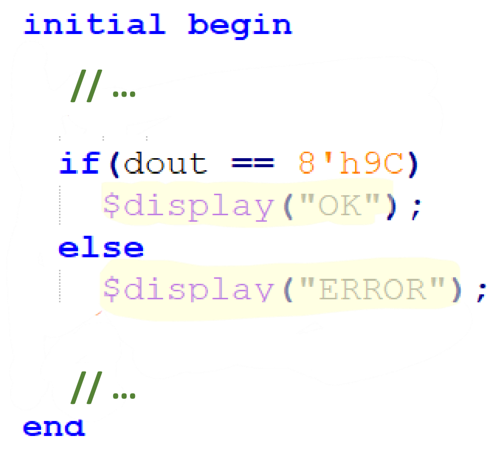
Testbench in SystemVerilog

- The **initial** block
  - But used also for output(s) check

```
initial begin
  // …

  if(dout == 8'h9C)
    $display("OK");
  else
    $display("ERROR");

  // …
end
```

Testbench in SystemVerilog

- The **initial** block
  - Command **$display**(<…>)
    - Used to print message(s) on the Transcript terminal (of Modelsim)



```
initial begin
  // …

  if(dout == 8'h9C)
    $display("OK");
  else
    $display("ERROR");

  // …
end
```

Testbench in SystemVerilog

- The **initial** block
  - Command **$display**(<…>)
    - Used to print message(s) on the Transcript terminal (of Modelsim)
    - Examples:
      - $display("Output value is %h" , dout); → print 'dout' in hexadecimal format
      - $display("Output value is %8h" , dout); → as above, but printing on 8 digits
      - $display("Output value is %08h" , dout); → as above, but filling head with 0s
      - $display("Output value is %10d" , dout); → print 'dout' in decimal format on 10 digits
      - $display("Output value is %32b" , dout); → print 'dout' in binary format on 32 digits
      - $display("Output value is %032b", dout); → as above, but filling head with 0s

Testbench in SystemVerilog

- The **initial** block
  - Also **$write**(<…>) command can be used
    - Like **$display**(<…>), but it does not include the new line
    - Examples:
      - $write("Output value is %h" , dout); → print 'dout' in hexadecimal format
      - $write("Output value is %8h" , dout); → as above, but printing on 8 digits
      - $write("Output value is %08h" , dout); → as above, but filling head with 0s

      - $write("Output value is %10d" , dout); → print 'dout' in decimal format on 10 digits

      - $write("Output value is %32b" , dout); → print 'dout' in binary format on 32 digits
      - $write("Output value is %032b", dout); → as above, but filling head with 0s

Testbench in SystemVerilog

- The **initial** block
  - Control commands for simulation
    - The simulation lasts as long as an event is scheduled
    - For instance, if using clock as shown before, simulation never stops, unless…

    - … you manually stop the simulation with the command of the simulation tool (Modelsim)
    - … you automatically stop the simulation with command **$stop** (inside an initial block)

Testbench in SystemVerilog

- The **initial** block
  - Control commands for simulation: **$stop** command
    - Automatically stops the simulation (when the simulation reaches that line of code) …

```
initial begin
  // …

  $stop;


  // …
end
```

Testbench in SystemVerilog

- The **initial** block
  - Control commands for simulation: **$stop** command
    - Automatically stops the simulation (when the simulation reaches that line of code) …
    - … when you run all the simulation: [Transcript terminal] run -all

```
initial begin
  // …

  $stop;


  // …
end
```

Simulation with testbench

- In practice, the **run –all** (instead of run 1, run 10, run 10ns, …) run all the simulation until its end (if any)
  - The **$stop** command sets the (automatic) end of the simulation

Simulation with testbench

- In conclusion, to perform the simulation (on Modelsim) with a testbench
  1. Include also the testbench file (.sv) in the Modelsim project
  2. Compile also the testbench file
  3. When launching the simulation, choose the testbench module
  4. Run the simulation
     - [Transcript terminal] >> run –all (if you included the $stop command)
     - Or >> run 1, run 10, …
     - Or >> run 100ps, run 100ns, …

Simulation with testbench

- In conclusion, to perform the simulation (on Modelsim) with a testbench
  1. Include also the testbench file (.sv) in the Modelsim project
  2. Compile also the testbench file
  3. When launching the simulation, choose the testbench module
  4. Run the simulation
     - [Transcript terminal] >> run –all (if you included the $stop command)
     - Or >> run 1, run 10, …
     - Or >> run 100ps, run 100ns, …

* If useful, please remember to create the waveform after launching the simulation (3), but before running it (4)

Reset vs. Initialization

- It may happen to confuse the reset with the initialization, since …
  - … the reset is a mechanisms to bring the system to a known state (typically at the power-on), which is why it is sometimes referred to as the initial state
  - … initialization means assign an initial (specific) value

Reset vs. Initialization

- It may happen to confuse the reset with the initialization, since …
  - … the reset is a mechanisms to bring the system to a known state (typically at the power-on), which is why it is sometimes referred to as the initial state
  - … initialization means assign an initial (specific) value
- But they are not the same: they differ in several ways!!!

Reset vs. Initialization

- Let’s assume the following example to better appreciate the difference
  - In a computer, the reset occurs when you press the reboot button
    - Everything is “turned off”
      - You lose the current content of RAM , the current state of programs, all unsaved progress, …
    - The computer (re-)starts from an initial state, which is well known and expected
  - Initialization may occur, for example, if you want to open a new blank document (to start writing it or filling it from the “beginning”)
    - All other programs are left untouched
    - The overall computer state does not restart from the initial state
      - At most, only the document management program starts from an “initial state” (i.e., the blank document)

Reset vs. Initialization

- Assume you want to execute an algorithm that generates a ciphertext C made of bytes $C[i]$ computed as:
  - $C[i] = P[i] \oplus K[i]$
  - $K[i] = K[i-1] + 1$
  - for i = 1, 2, …

  - $P[i]$ is the ith byte of plaintext
  - $K[0] = K$, the initial key

- Assume that the reset state for $K[i]$ corresponds to K (and not all zeros)
  - If you reset, you can restart the algorithm to encrypt a new message, because it re-initialize the $K[i]$ value

Reset vs. Initialization

- Assume you want to execute an algorithm that generates a ciphertext C made of bytes $C[i]$ computed as:
  - $C[i] = P[i] \oplus K[i]$
  - $K[i] = K[i-1] + 1$
  - for i = 1, 2, …

  - $P[i]$ is the ith byte of plaintext
  - $K[0] = K$, the initial key

- Assume that the reset state for $K[i]$ corresponds to K (and not all zeros)
  - Now assume that at a certain point you agree on a different initial key value, K_2, because the other initial key is no longer valid (it has been compromised, for example)
  - You encrypt the first message, and everything is ok. And if you want to start encrypting a new second message? what do you do? Do you reset the system?
    - If you reset the system, K_2 is lost and you will start encrypting the second message using K instead of K_2

Reset vs. Initialization

- To conclude, reset and initialization are different because of
  - Timing/occurrence
    - The reset is aimed to recover the system from a failing state, bringing it to a known (initial) state, so it is used at the power-on and only if necessary
    - The purpose of initialization is to assign an initial value, and its use falls within the typical operation of a program, or circuit, ...
  - Scope
    - The reset is global: when you reset, you reset the whole system
    - The initialization is local: it involves only a few specific parts (of a program, or of a circuit, …)

Reset vs. Initialization

- This to say that …
  - Don’t assume reset logic/resources to be used to perform initialization
  - Also reset logic/resources can be used for initialization, but not only them
  - Initialization requires specific dedicated logic/resources
    - For example, when you design a circuit using HDL

Exercise with SystemVerilog

- Implementation of a sequential circuit and simulation with Modelsim and testbench

Exercise with SystemVerilog

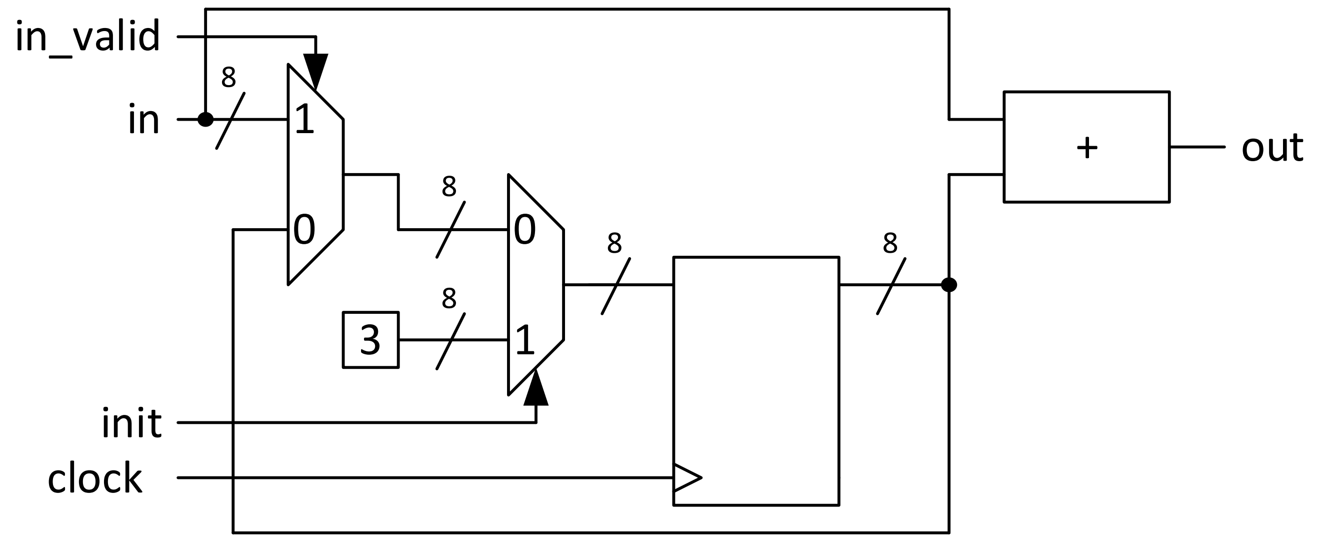
- Implementation of a sequential circuit and simulation with Modelsim and testbench
  - Design a module that perform the following operation at every clock cycle, if it receives a valid data input
    - $\mathrm{out}[\mathrm{i}] = \mathrm{in}[\mathrm{i}] + \mathrm{in}[\mathrm{i}-1]$
  - i = ith clock cycle (with a valid input data)
  - $\mathrm{in}[0]$ = 3
  - $\mathrm{out}[\mathrm{i}]$ and $\mathrm{in}[\mathrm{i}]$ are bytes

Exercise with SystemVerilog

- Implementation of a sequential circuit and simulation with Modelsim and testbench
  - Assume that your module gets the following valid input bytes: $B_1, B_2, B_3, B_4, \ldots$
  - So
    - At the first clock cycle (with a valid input byte, i = 1): $\text{out} = \text{B}_1 + 3$
    - At the second clock cycle (i = 2): $\text{out} = B_2 + \text{B}_1$
    - At the third clock cycle (i = 3): $\text{out} = \text{B}_3 + \text{B}_2$
    - At the fourth clock cycle (i = 4): $\text{out} = \text{B}_4 + \text{B}_3$
    - …

Exercise with SystemVerilog

- Implementation of a sequential circuit and simulation with Modelsim and testbench
  - In other words, a circuit like this
    - Omitting reset (asynchronous, active-low)



Exercise with SystemVerilog

- Implementation of a sequential circuit and simulation with Modelsim and testbench
  - Implement also the testbench
    - Clock generator
    - Power-on reset
    - Initialize the module, give some inputs, then re-initialize (without resetting), and give other inputs
  - Using the waveform, check that after the re-initialization the register contains the value 8’d3

Exercise with SystemVerilog

- Implementation of a sequential circuit and simulation with Modelsim and testbench
  - You can find all the files about this exercise in the dedicated folder on the Team of the course
    - File > Electronics Systems module > Crocetti > Exercises > 2.5
  - Try to simulate on your own

Thank you for your attention

Luca Crocetti
(luca.crocetti@unipi.it)