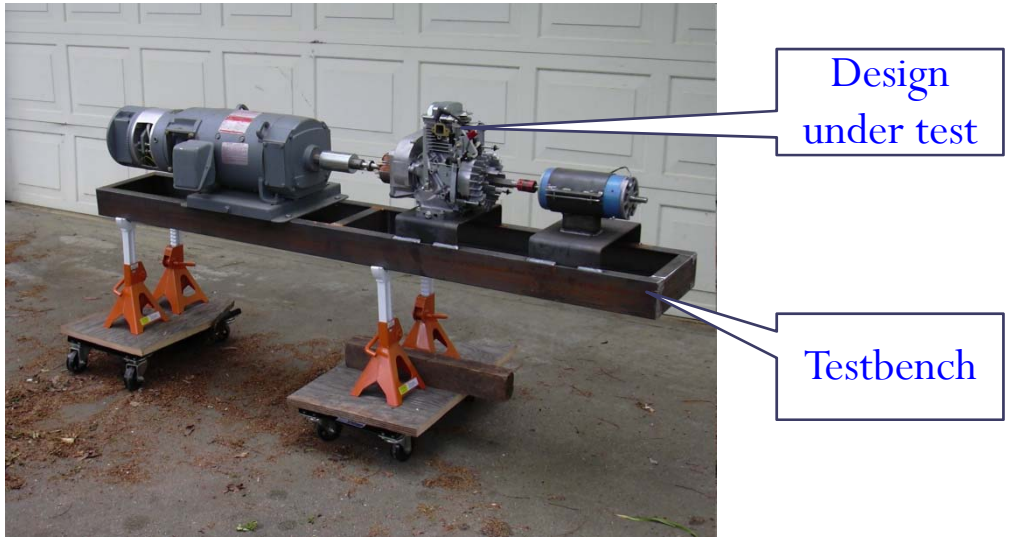
TKT-1212
Digitaalijärjestelmien toteutus

Lecture 7: VHDL Testbenches

Erno Salminen 2012



Contents

- Purpose of test benches
- Structure of simple test bench
  - Side note about delay modeling in VHDL
- Better test benches
  - Separate, better reusable stimulus generation
  - Separated sink for the response
  - File handling for stimulus and response
- Example and conclusions

Introduction

- Verification is perhaps the most difficult aspect of any design
  - That's not an excuse for omitting it…
  - Multiple levels: single component, module with multiple sub-components, and system-level
- In synchronous design, we verify the *functionality at cycle-level accuracy*
  - Not detailed timing, which will be checked with static timing analysis (STA) tools
- Note that term *testing* is usually used for manufacturing test of Ics whereas *verification* ensures functional correctness
  - However, terms are often used interchangeably and most people talk about *testbenches*
  - E.g. DUT (design under test) should actually be DUV

What is a VHDL Test Bench (TB)?

- VHDL test bench (TB) is a piece of code meant to verify the functional correctness of HDL model
- The main objectives of TB is to:
  1. Instantiate the design under test (DUT)
  2. Generate stimulus waveforms for DUT
  3. Generate reference outputs and compare them with the outputs of DUT
  4. Automatically provide a pass or fail indication
- Test bench is a part of the circuits specification
- Sometimes it's a good idea to design the test bench before the DUT
  - Functional specification ambiguities found
  - Forces to dig out the relevant information of the environment
  - Different designers for DUT and its tb!

Testbench benefits

- Unit is inspected outside its real environment
  - Of course, TB must resemble the real environment
  - Making TB realistic is sometimes hard, e.g. interface to 3rd party ASIC which does not have a simulation model
- Isolating the DUT into TB has many desirable qualities
  - Less ”moving parts”, easier to spot the problem
  - Easy to control the inputs, also to drive illegal values
  - Easy to see the outputs
  - Small test system fast to simulate
  - Safer than real environment, e.g. better to test mergency shutdown first in laboratory than in real chemical factory

Stimulus and Response

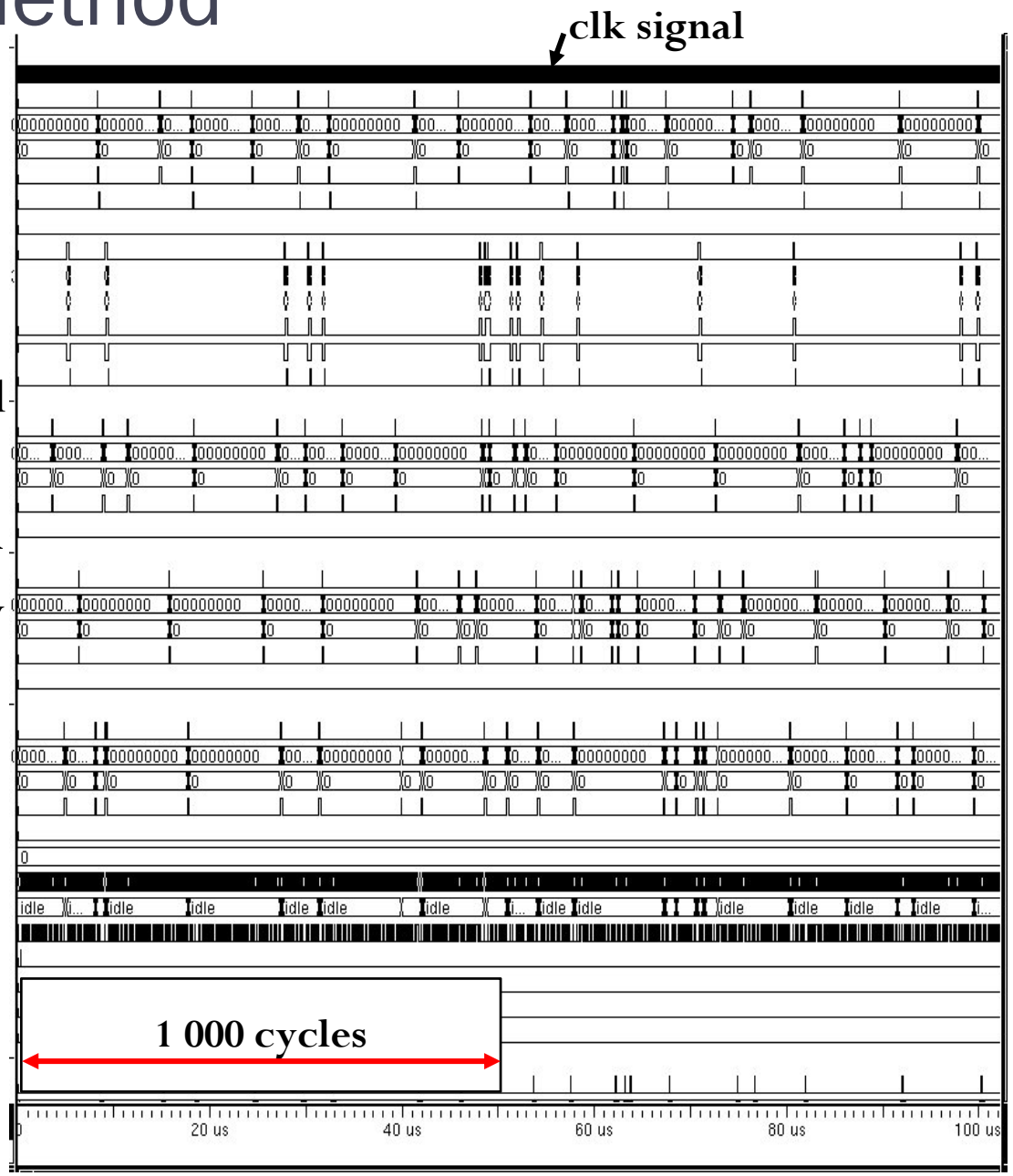
- TB can **generate the stimulus** (input to DUT) in several ways:
  a) Read vectors stored as constants in an array
  b) Read vectors stored in a separate system file
  c) Algorithmically "on-the-fly"
  d) Read from C through Foreign Language Interface (FLI, Modelsim)
- **The response** (output from DUT) **must be automatically checked.**
  - Expected response must be known exactly
  - Response can be stored into file for further processing.
- Example:
  - Stimulus can be generated with Matlab and TB feeds it into DUT.
  - DUT generates the response and TB stores it into file.
  - Result are compared to Matlab simulations automatically, no manual comparison!

Choosing test method

A. Manual
   - generate test with modelsim `force` command
   - check the wave forms

B. Automated test generation and response checking

- **B is the only viable option**
- This real-life figure shows only
  - 1/3 of signals
  - 1/50 000 of time line
- This check should be repeated few times a day during development…

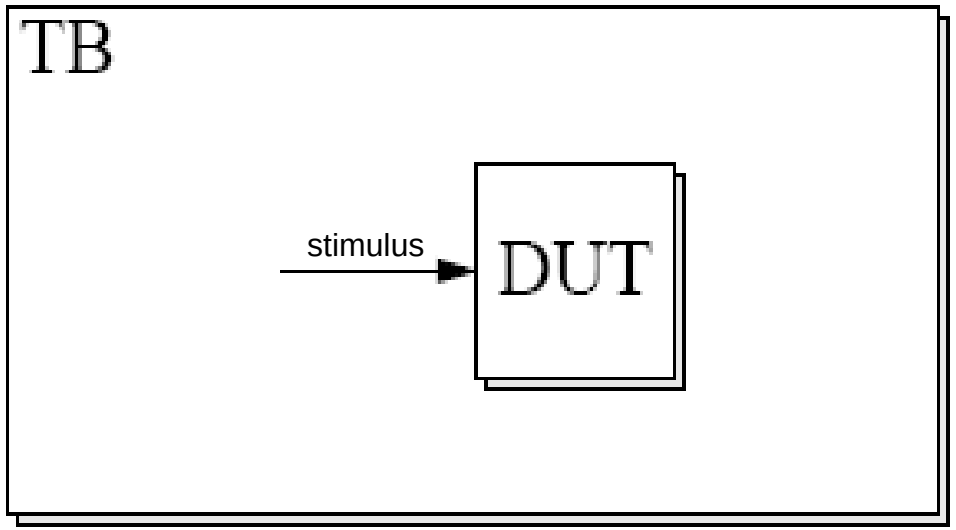


Test Bench Structures

- TB should be reusable without difficult modifications
  - ➔ Modular design
- The structure of the TB should be simple enough so that other people understand its behaviour
- It has to be easy to run
  - Not much dependencies on files or scripts
- Good test bench propagates all the generics and constants into DUT
- Question: How to verify that the function of the test bench is correct? A: “Sanopa muuta kato”

Simple Test Bench

- Only the DUT is instantiated into test bench.
- Stimulus is generated inside the test bench
  - Not automatically – handwritten code trying to spot corner cases
  - Poor reusability.
- Suitable only for very simple designs, if at all
- However, such "tb" can be used as an *usage example* to *familiarize new user* with DUT
  - "See, driving input like this makes the DUT do something useful…".

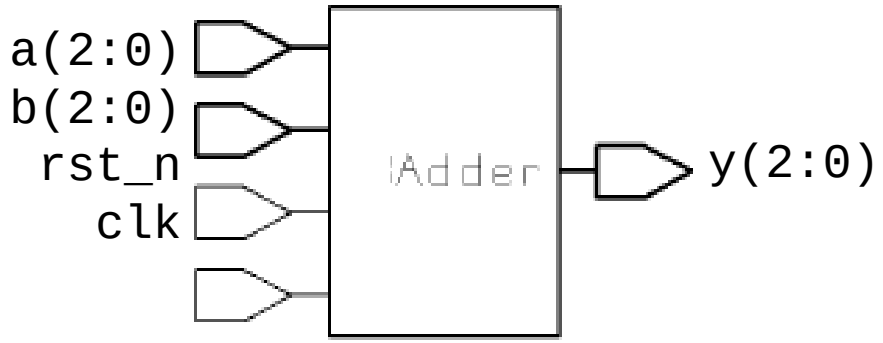


Better than none, but not reliable

A Simple Test Bench (2)

- DUT: Synchronous adder, entity, architecture

```
ENTITY adder IS
    PORT (
        clk   : IN STD_LOGIC;
        rst_n : IN STD_LOGIC;
        a, b  : IN UNSIGNED(2 DOWNTO 0);
        y     : OUT UNSIGNED(2 DOWNTO 0));
END adder;
ARCHITECTURE RTL OF adder IS
BEGIN -- RTL
    PROCESS (clk, rst_n)
    BEGIN -- process
        IF rst_n = '0' THEN -- asynchronous reset (active low)
            y <= (OTHERS => '0');
        ELSIF clk'EVENT AND clk = '1' THEN -- rising clock edge
            y <= a + b;
        END IF;
    END PROCESS;
END RTL;
```



A Simple Test Bench (3)

- Test bench
  - Simplest possible entity declaration:

```
ENTITY simple_tb IS
END simple_tb;
```

  - Architecture:

```
ARCHITECTURE stimulus OF simple_tb IS
```

  - DUT:

```
COMPONENT adder
  PORT (
        clk   : IN STD_LOGIC;
        rst_n : IN STD_LOGIC;
        a, b  : IN UNSIGNED(2 DOWNTO 0);
        y     : OUT UNSIGNED(2 DOWNTO 0)
       );
 END COMPONENT;
```

A Simple Test Bench (4)

- Clock period and connection signals:

```
CONSTANT period : TIME := 50 ns;
SIGNAL clk      : STD_LOGIC := '0'; -- init values only in tb
SIGNAL rst_n    : STD_LOGIC;
SIGNAL a, b, y  : unsigned(2 downto 0);
```

- Begin of the architecture and component instantiation:

```
begin
DUT : adder
  PORT MAP (
    clk   => clk,
    rst_n => rst_n,
    a     => a,
    b     => b,
    y     => y);
```

- Clock generation:

```
generate_clock : PROCESS (clk)
BEGIN -- process
   clk <= NOT clk AFTER period/2; -- this necessitates init value
END PROCESS;
```

A Simple Test Bench (5)

- Stimuli generation and the end of the architecture:

```
rst_n <= '0',
         '1' AFTER 10 ns;
a <= "000",
     "001" AFTER 225 ns,
     "010" AFTER 375 ns;
b <= "000",
     "011" AFTER 225 ns,
     "010" AFTER 375 ns;
end stimulus; -- ARCHITECTURE
```

Not very comprehensive

- Configuration:

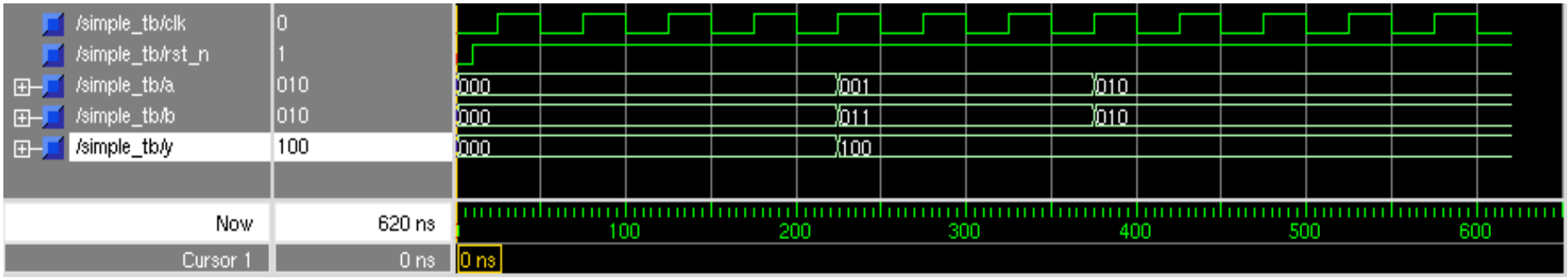
```
CONFIGURATION cfg_simple_tb OF simple_tb IS
  FOR stimulus
    FOR DUT : adder
          USE ENTITY work.adder(RTL);
    END FOR;
  END FOR;
END cfg_simple_tb;
```

A Simple Test Bench (6)

- Simulation:

How to check correcteness?

Take your eye into hand and watch.
Not too convenient…

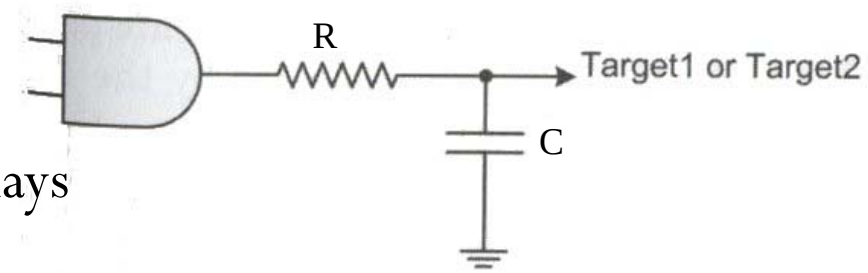


You notice that is *does something* but validity is hard to ensure.

VHDL delay modeling

- Signal assignments can have delay (as in prev example)

1. Inertial delays
   - Used for modeling propagation delay, or RC delay
   1. `after`
   2. `reject-inertial`
   - Useful in modeling gate delays
   - Glitches filtered
2. `transport` delay
   - transmission lines
   - testbench stimuli generation
   - glitches remain



VHDL delay modeling (2)

```
-- Inertial delay
target1 <= waveform AFTER 5 NS;

-- Inertial with reject
target2 <= REJECT 3 NS INERTIAL waveform AFTER 5 NS;

-- Illustrating transport delay
target3 <= TRANSPORT waveform AFTER 5 NS;
```

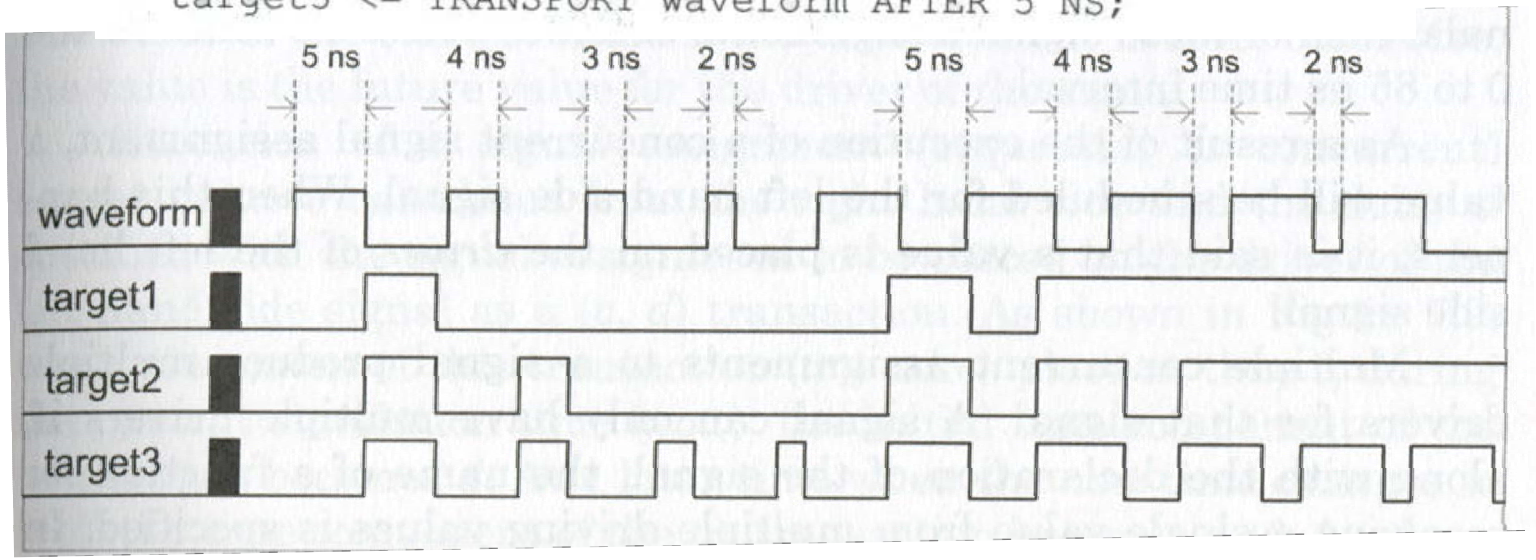
- `After 5 ns` propagates signal change IF the signal value stays constant for > 5 ns, and change occurs 5 ns after the transition
- `reject inertial` propagates signal change if value is constant for > 3 ns and change occurs 5 ns after transition
- `transport` propagates the signal *as is* after 5 ns

VHDL delay modeling (3)

```
-- Inertial delay
target1 <= waveform AFTER 5 NS;

-- Inertial with reject
target2 <= REJECT 3 NS INERTIAL waveform AFTER 5 NS;

-- Illustrating transport delay
target3 <= TRANSPORT waveform AFTER 5 NS;
```

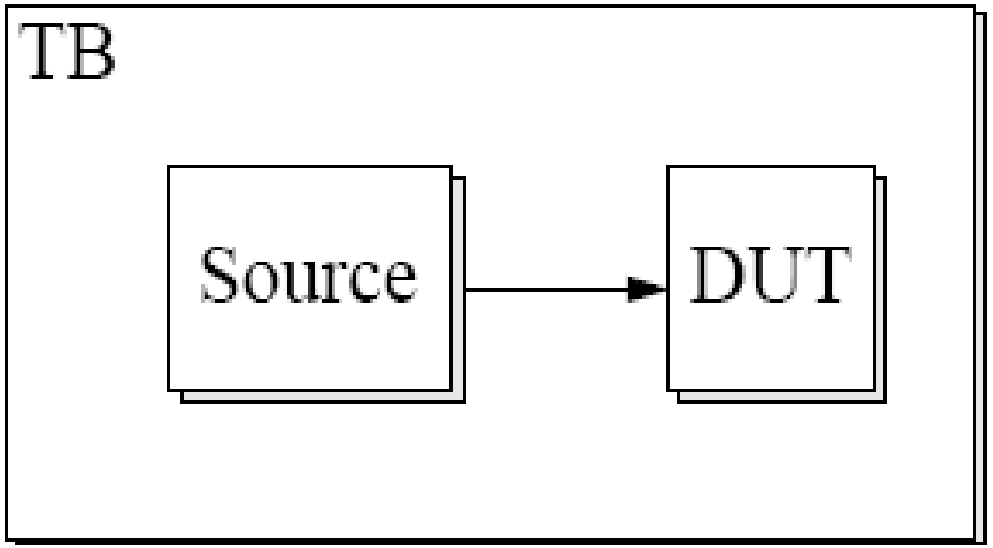


Be careful! Behavior will be strange if the edges of the clk and signal generated this way are aligned.

More elegant test benches

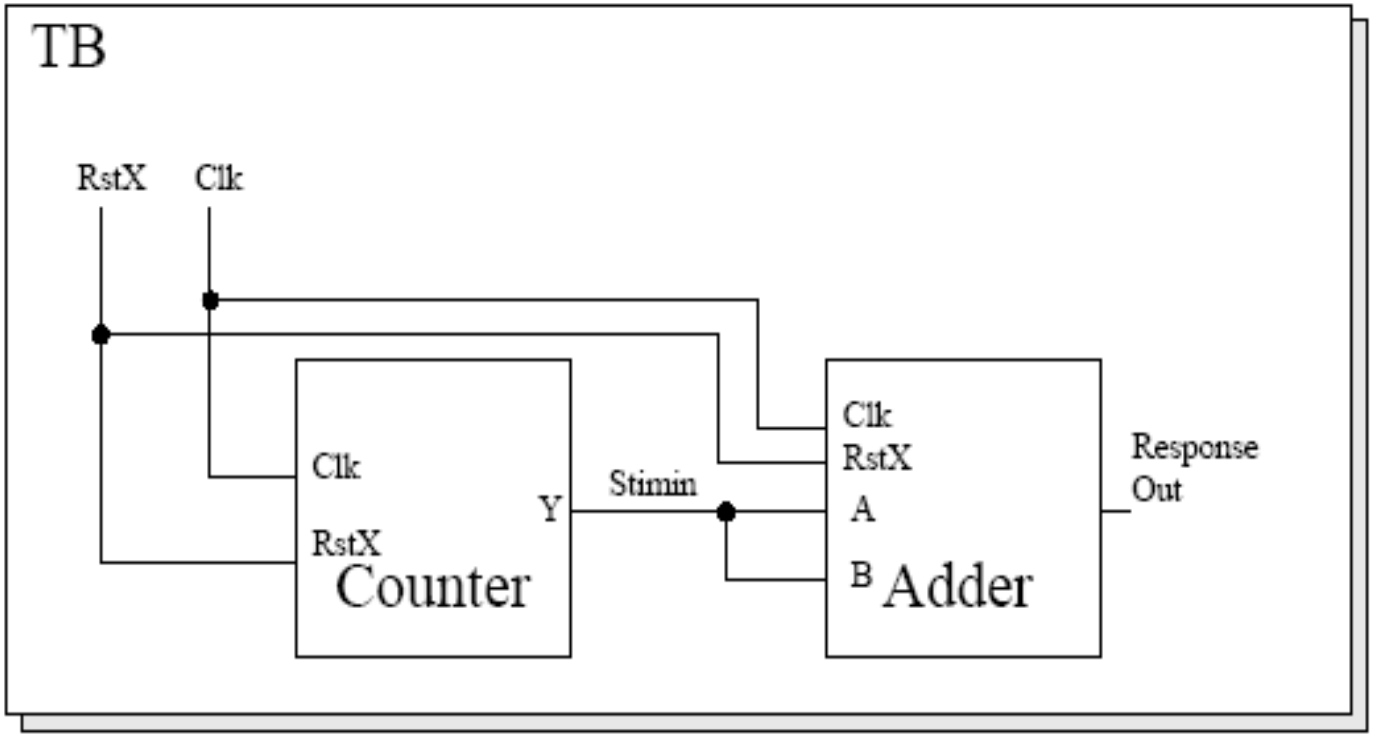
Test Bench with a Separate Source

- Source and DUT instantiated into TB
- For designs with complex input and simple output
- Source can be e.g. another entity or a process



A Test Bench with a Separate Source (2)

- Input stimuli for "adder" is generated in a separate entity "counter".



A Test Bench with a Separate Source (3)

- Stimulus source is a clock-triggered up counter
- Entity of the source:

```
ENTITY counter IS
    PORT (
      clk   : IN STD_LOGIC;
      rst_n : IN STD_LOGIC;
      Y     : OUT STD_LOGIC_VECTOR(2 DOWNTO 0)
      );
END counter;
```

A Test Bench with a Separate Source (4)

- Architecture of the source component:

```
ARCHITECTURE RTL OF counter IS
   SIGNAL Y_r : unsigned(2 downto 0)
BEGIN -- RTL
    PROCESS (clk, rst_n)
    BEGIN -- process
      IF rst_n = '0' THEN -- asynchronous reset (active low)
        Y_r <= (OTHERS => '0');
      ELSIF clk'EVENT AND clk = '1' THEN -- rising clock edge
        Y_r <= Y_r+1;
      END IF;
    END PROCESS;
Y <= std_logic_vector(Y_r);
END RTL;
```

Note: overflow not checked

A Test Bench with a Separate Source (5)

- Test bench:
  - Architecture

```
ARCHITECTURE separate_source OF source_tb IS
```

  - Declare the components DUT and source

```
COMPONENT adder
  PORT (
          clk   : IN STD_LOGIC;
          rst_n : IN STD_LOGIC;
          a, b  : IN UNSIGNED(2 DOWNTO 0);
          y     : OUT UNSIGNED(2 DOWNTO 0)
        );
 END COMPONENT;
 COMPONENT counter
 PORT (
          clk   : IN STD_LOGIC;
          rst_n : IN STD_LOGIC;
          y     : OUT UNSIGNED(2 DOWNTO 0)
        );
 END COMPONENT;
```

A Test Bench with a Separate Source (6)

- Clock period and connection signals

```
CONSTANT period : TIME      := 50 ns;
SIGNAL clk      : STD_LOGIC := '0'; -- init value allowed only in simulation!
SIGNAL rst_n    : STD_LOGIC;
SIGNAL response_dut_tb, a_cntr_dut, b_cntr_dut  : unsigned(2 downto 0);
```

- Port mappings of the DUT and the source

```
begin
  DUT : adder
    PORT MAP (
      clk   => clk,
      rst_n => rst_n,
      a     => a_cntr_dut,
      b     => b_cntr_dut,
      y     => response_dut_tb
   );
  i_source : counter
    PORT MAP (
      clk   => clk,
      rst_n => rst_n,
      y     => a_cntr_dut
   );
  b_cntr_dut <= a_cntr_dut;
-- simplication, generally should be different from stim_a_in
```

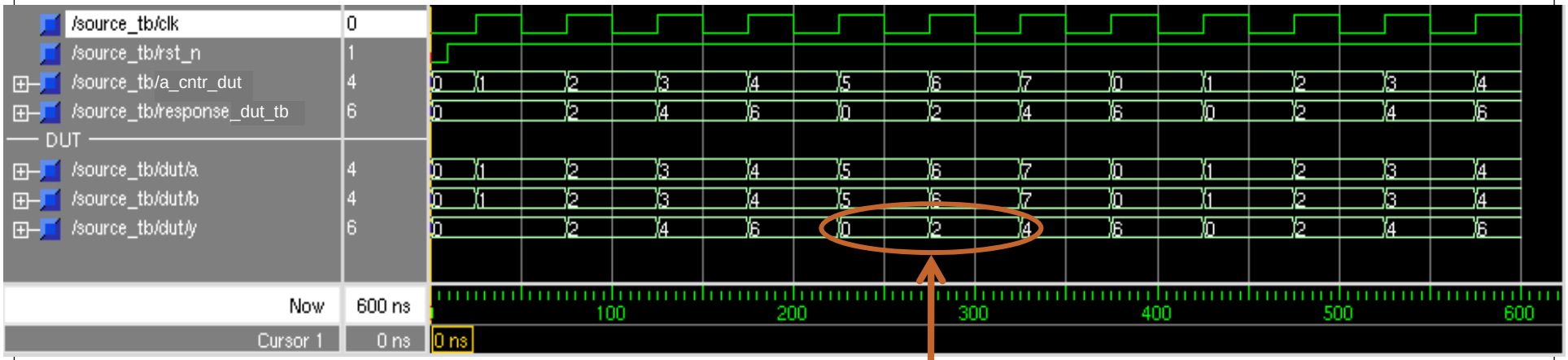
A Test Bench with a Separate Source (7)

- Clock and reset can be generated also without processes

```
clk   <= NOT clk AFTER period/2; -- this style needs init value
rst_n <= '0', '1' AFTER 10 ns;
END separate_source
```

A Test Bench with a Separate Source (8)

- Simulation:



Overflow in adder, three bits → unsigned value range 0..7

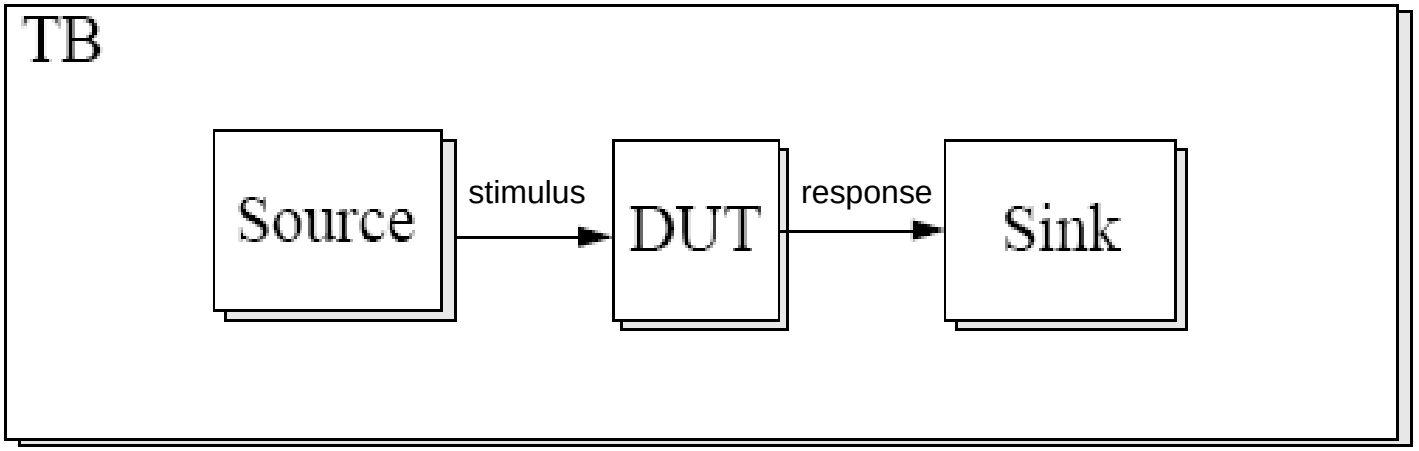
The checking is still inconvenient.

Better than previous tb.
Easy to scale the stimulus length for wider adders.
Quite straightforward to test all values by instantiating two counters.

Response handling

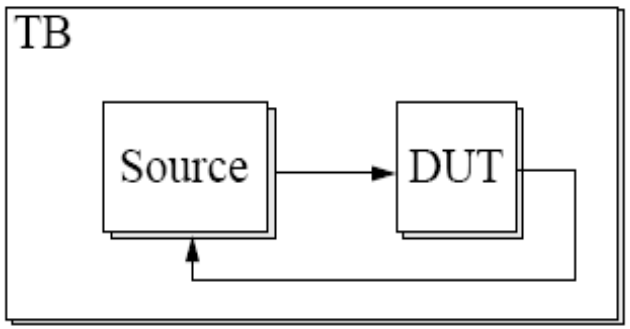
Test Bench with a Separate Source and Sink

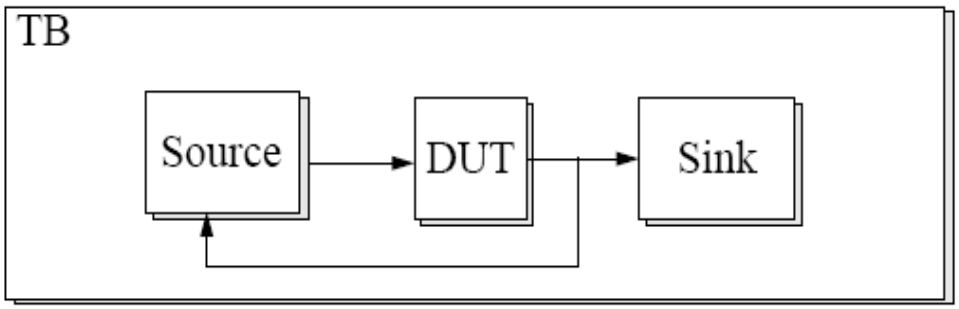
- Both the stimulus source and the sink are separate instances.
- Complex source and sink without response-source interaction.
- Sink uses assertions of some sort



Smart Test Bench

- Circuit's response affects further stimulus.
- In other words, TB is *reactive*
  - E.g. DUT requests source to stall, if DUT cannot accept new data at the moment
  - E.g. Source fwrites to FIFO (=DUT) until it is full, then it does something else…
  - Non-reactive TB with fixed delays will break immediately, if DUT's timing changes



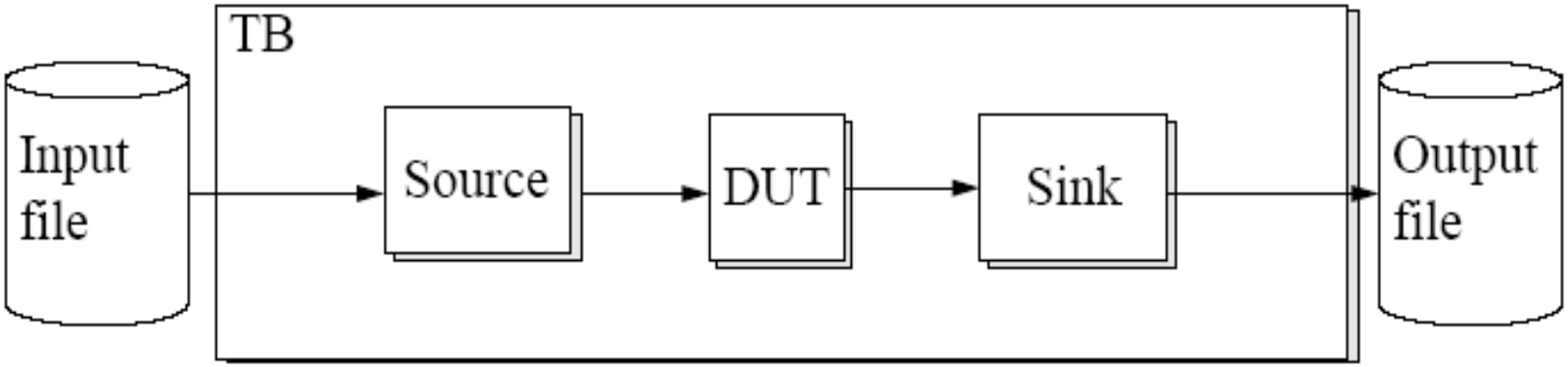


Presence of expected results + non-existence of bad side-effects

- It is evident to check that the *main result happens*
  - E.g. having inputs `a=5, b=6` yields an output `sum=11`
  - For realistic system, even this gets hard
- It is much more subtle and harder to check that *nothing else happens*
  - E.g. the previous result is not an overflow, not negative, not zero, valid does not remain active too long, all outputs that should remain constant really do so, unit does not send any extra data…
  - E.g. SW function does not corrupt memory (even if it calculates the correct result), SW function does not modify HW unit's control registers unexpectedly, function does not wipe the GUI screen blank…
  - This is tricky if the consequence is seen on primary outputs much later (e.g. later operation fails since ctrl regs were modified unexpectedly)
- Usually there is 1 or perhpas few entirely correct results and infinite number of wrong ones

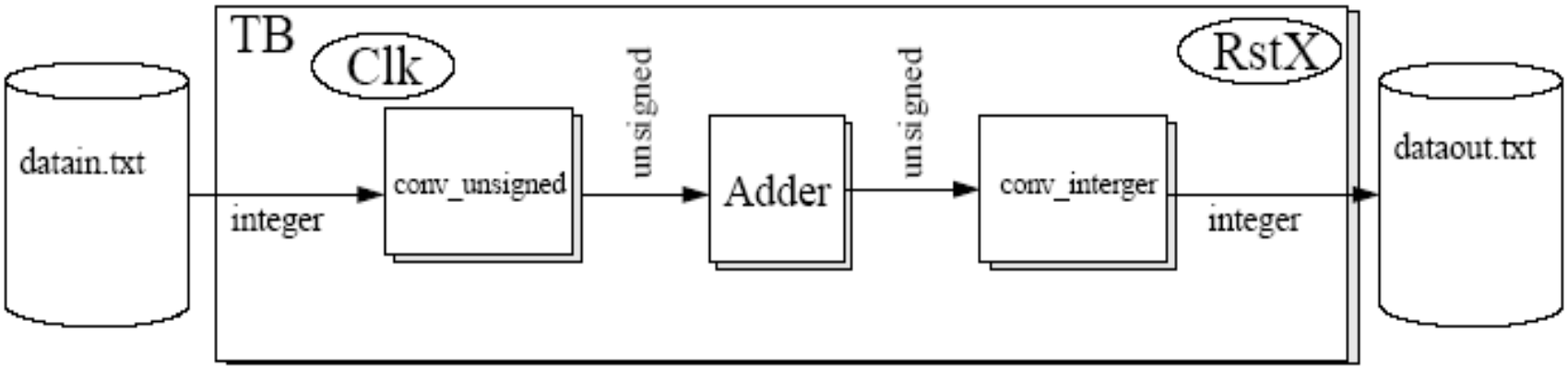
Test Bench with Text-IO

- Stimulus for DUT is read from an input file and modified in the source component
- The response modified is in the sink and written to the output file



A Test Bench with Text-IO (2)

- Test case:



A Test Bench with Text-IO (3)

- Test bench:
  - Libraries, remember to declare the textio-library!

```
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_arith.all; -- old skool
use std.textio.all;
use IEEE.std_logic_textio.all;
```

A Test Bench with Text-IO (4)

- Architecture

```
ARCHITECTURE text_io OF source_tb IS
  COMPONENT adder
  PORT (
         clk   : IN STD_LOGIC;
         rst_n : IN STD_LOGIC;
         a, b  : IN UNSIGNED(2 DOWNTO 0);
         y     : OUT UNSIGNED(2 DOWNTO 0)
         );
   END COMPONENT;

  CONSTANT period : TIME := 50 ns; -- even value
  SIGNAL clk      : STD_LOGIC := '0';
  SIGNAL rst_n    : STD_LOGIC;
  SIGNAL a, b, y  : unsigned(2 downto 0);
```

A Test Bench with Text-IO (5)

- In achitecture body

```
begin
  DUT : adder
      PORT MAP (
        clk   => clk,
        rst_n => rst_n,
        a     => a,
        b     => b,
        y     => y);

   clk   <= NOT clk AFTER period/2;
   rst_n <= '0',
            '1' AFTER 75 ns;
```

A Test Bench with Text-IO (6)

- Create process and declare the input and output files (VHDL'87)

```
process (clk, rst_n)
FILE file_in  : TEXT IS IN  "datain.txt";
FILE file_out : TEXT IS OUT "dataout.txt";
```

  - File paths are relative to *simulation directory* (the one with modelsim.ini)
- Variables for one line of the input and output files

```
VARIABLE line_in  : LINE;
VARIABLE line_out : LINE;
```

  - Value of variable is updated immediately. Hence, the new value is visible on the same execution of the process (already on the next line)
- Variables for the value in one line

```
VARIABLE input_tmp  : INTEGER;
VARIABLE output_tmp : INTEGER;
```

A Test Bench with Text-IO (7)

- Beginning of the process and reset

```
BEGIN -- process
    IF rst_n = '0' THEN -- asynchronous reset
      a <= (OTHERS => '0');
      b <= (OTHERS => '0');

  ELSIF clk'EVENT AND clk = '1' THEN -- rising clock edge
```

- Read one line from the input file to the variable "line_in" and read the value in the line "line_in" to the variable "input_tmp"

```
IF NOT (ENDFILE(file_in)) THEN
  READLINE(file_in, line_in);
  READ    (line_in, input_tmp);
```

A Test Bench with Text-IO (8)

- "input_tmp" is fed to both inputs of the DUT

```
a <= CONV_UNSIGNED(input_tmp, 3); -- old skool conversion
b <= CONV_UNSIGNED(input_tmp, 3);
```

- The response of the DUT is converted to integer and fed to the variable "output_tmp"

```
output_tmp := CONV_INTEGER(y);
```

- The variable "output_tmp" is written to the line "line_out" that is written to the file "file_out"

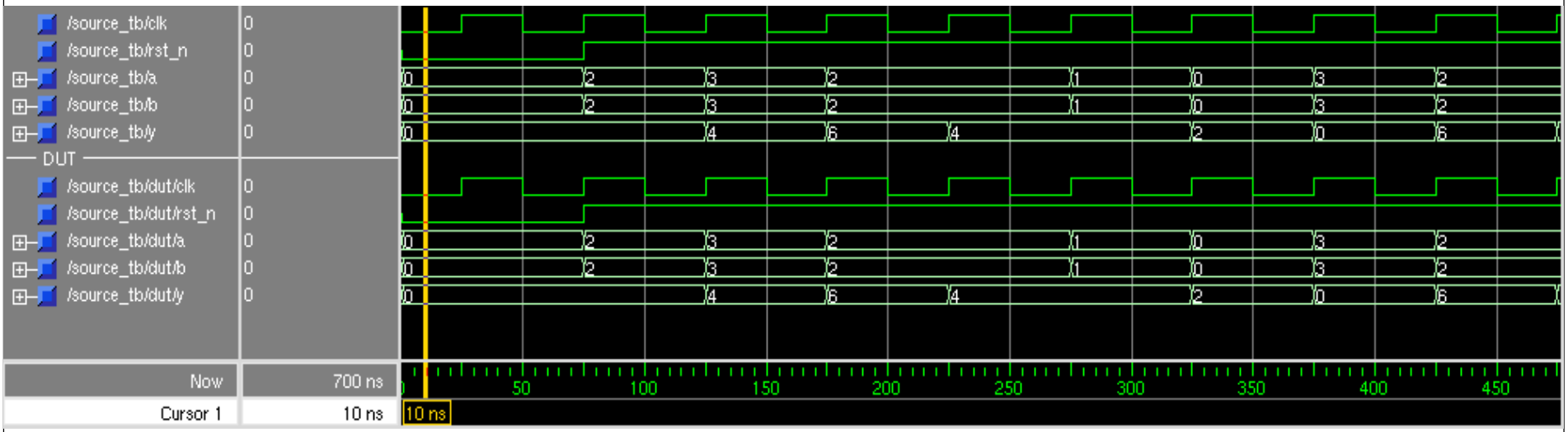
```
WRITE    (line_out, output_tmp);
WRITELINE(file_out, line_out);
```

- At the end of the input file the note "End of file!" is given

```
ELSE
  ASSERT FALSE
    REPORT "End of file!"
    SEVERITY NOTE;
END IF;
```

A Test Bench with Text-IO (9)

- Simulation:

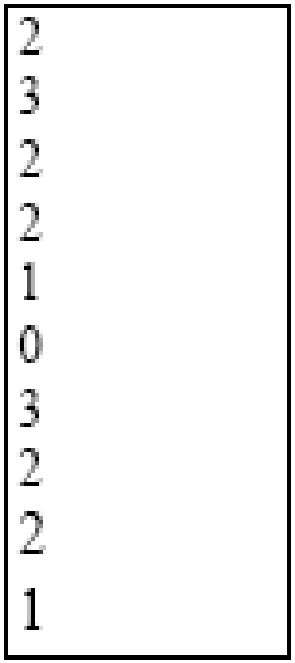


Now, the designer can prepare multiple test sets for certain corner cases (positive/negative values, almost max/min values, otherwise interesting) . However, the VHDL is not modified.

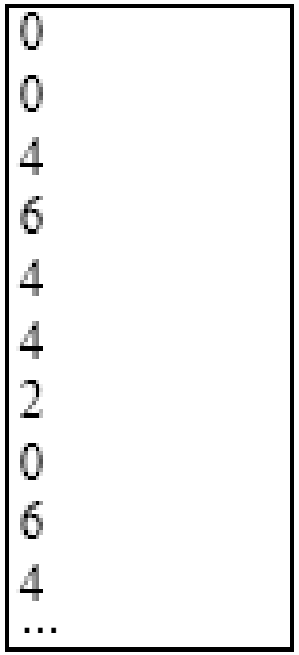
This version does not check the response yet.

A Test Bench with Text-IO (10)

- Input file provided to test bench



- Output file produced by the test bench:



Two cycle latency:
1 cycle due to file read
+1 cycle due to DUT

Comments:
For each stimulus file, the designer also prepares the *expected output trace*. It can be automatically
compared to the response of DUV, either in VHDL or using command line tool *diff* in Linux/Unix.
It is good to allow comments in stimulus file. They can describe the structure:
e.g. There is 1 line per cycle, 1st value is… in hexadecimal format, 2nd value is…

Text-I/O types supported by default

- READ and WRITE procedures support
  - Bit, bit_vector
  - Boolean
  - Character
  - Integer
  - Real
  - String
  - Time
  - Source: textio_vhdl93 library, Altera
- For other types, use `txt_util.vhd` from http://www.stefanvhdl.com/vhdl/vhdl/txt_util.vhd

Hint: std_logic_1164, numeric_std etc are all available in VHDL, you can always check those for reference.

Test prints

- Printing a constant text string is easy inside processes, just report what's going on

```
process (…)
…
report (”Thunder!”);
```

- Some people use

```
assert false report ”Bazinga!” severity…
```

- Whereas some prefer

```
write (line_v, string'(”Halibatsuippa!”));
writeline(output, line_v);
-- output is a reserved word for stdout
```

- On the other hand, signal and variable values are a bit tricky

Test print examples (2)

- Write function example for 1-bit std_logic

```
write (line_v, string'("Enable "));
write (line_v, to_bit(en_r));
writeline(output, line_v);
```

- Integers and enumerations can be converted to string, for example

```
write (line_v, string'("I= ")& integer'image(5));
```

or

```
report "Value is "
& integer'image(to_integer(unsigned(data_vec)));
```

Test print layout

- Default layout for report/assert uses two lines which is somewhat inconvenient

```
** Note: Thunder!
#    Time: 60 ns  Iteration: 0  Instance: /tb_tentti
```

- Modify modelsim.ini and restart vsim

```
; AssertionFormat       = "** %S: %R\n  Time: %T  Iteration: %D%I\n"
AssertionFormat        = "** %S: %R  Time: %T  %I\n"
```

- Then

```
# ** Note:Thunder!  Time: 60 ns  Instance: /tb_tentti
# ** Note: hojo hojo  Time: 88 ns  Instance: /tb_tentti
```

  - Me likes! For example, using `grep` is much much easier now

File handling in VHDL'87 and '93

- [HARDI VHDL handbook's page 71]

```
-- VHDL'87:
FILE f1 : myFile IS IN "name_in_file_system";
FILE f2 : mySecondFile IS OUT "name_in_file_system";
-- VHDL'93:
FILE f1 : myFile OPEN READ_MODE IS "name_in_file_system";
FILE f2 : mySecondFile OPEN WRITE_MODE IS "name_in_file_system";
```

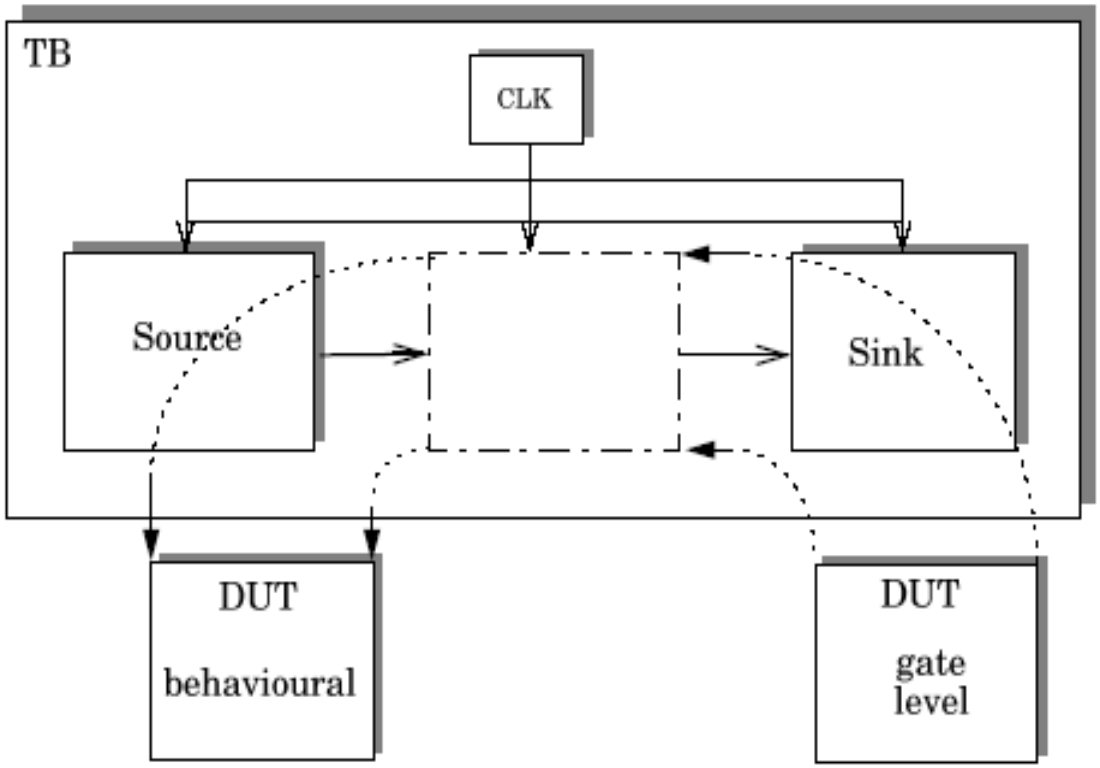
- Input files may be written compatible with both VHDL'87 and VHDL'93, but for output files that is not possible:

```
-- Declaration of an input file both for VHDL'87 and VHDL'93
FILE f : myFile IS "name_in_file_system";
```

- The predefined subprograms `FILE_OPEN` and `FILE_CLOSE` do not exist in VHDL'87.

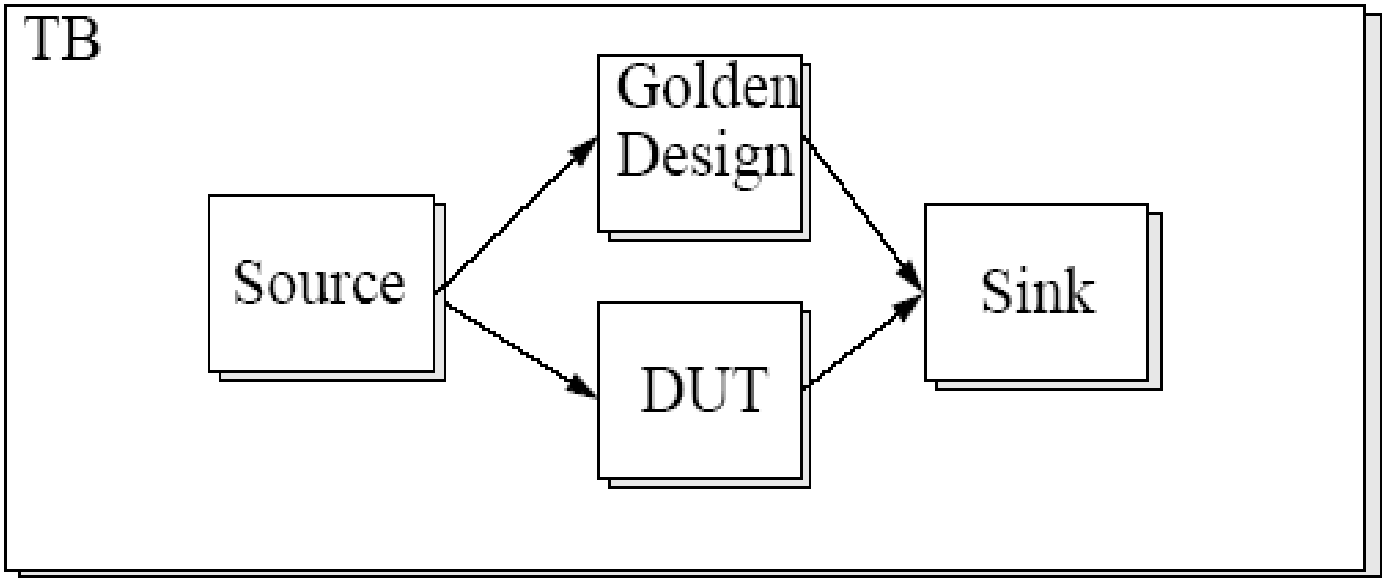
Same Test Bench for Different Designs

- Architecture of the DUT can be changed
- Should always be the objective
- Creating a separate testbench for gate-level will likely introduce bugs in TB

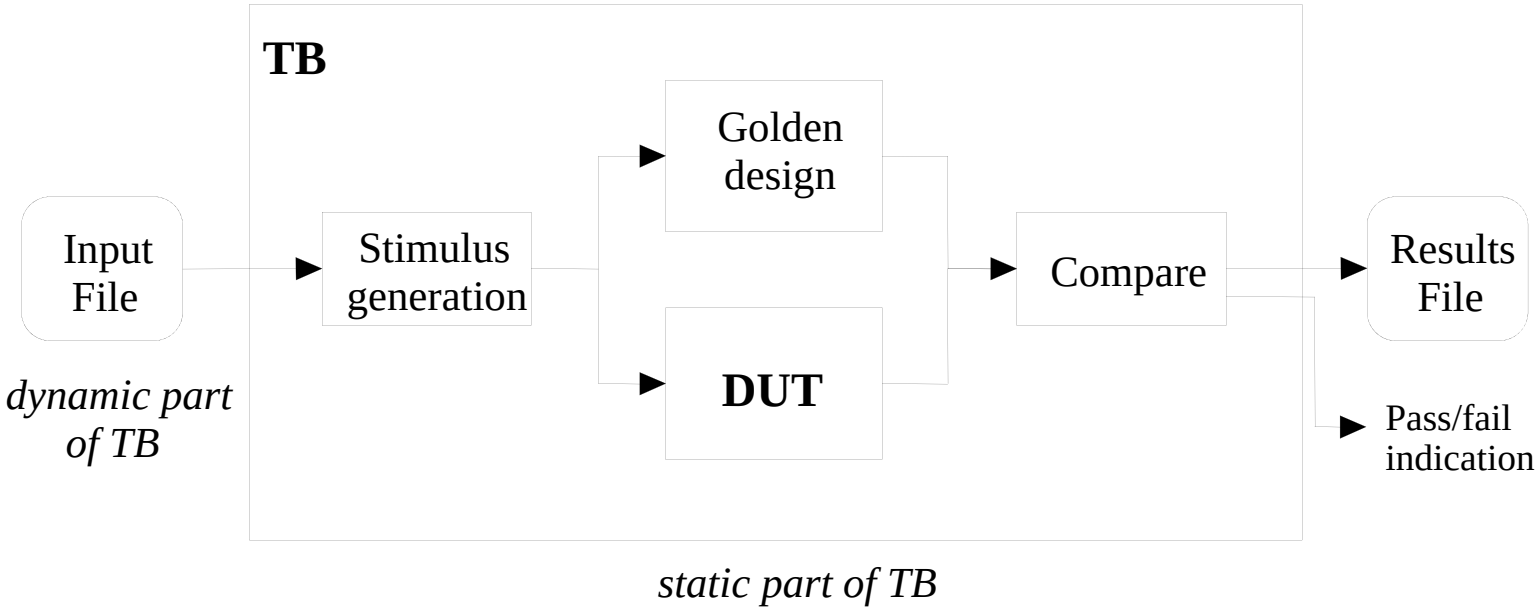


Golden Design

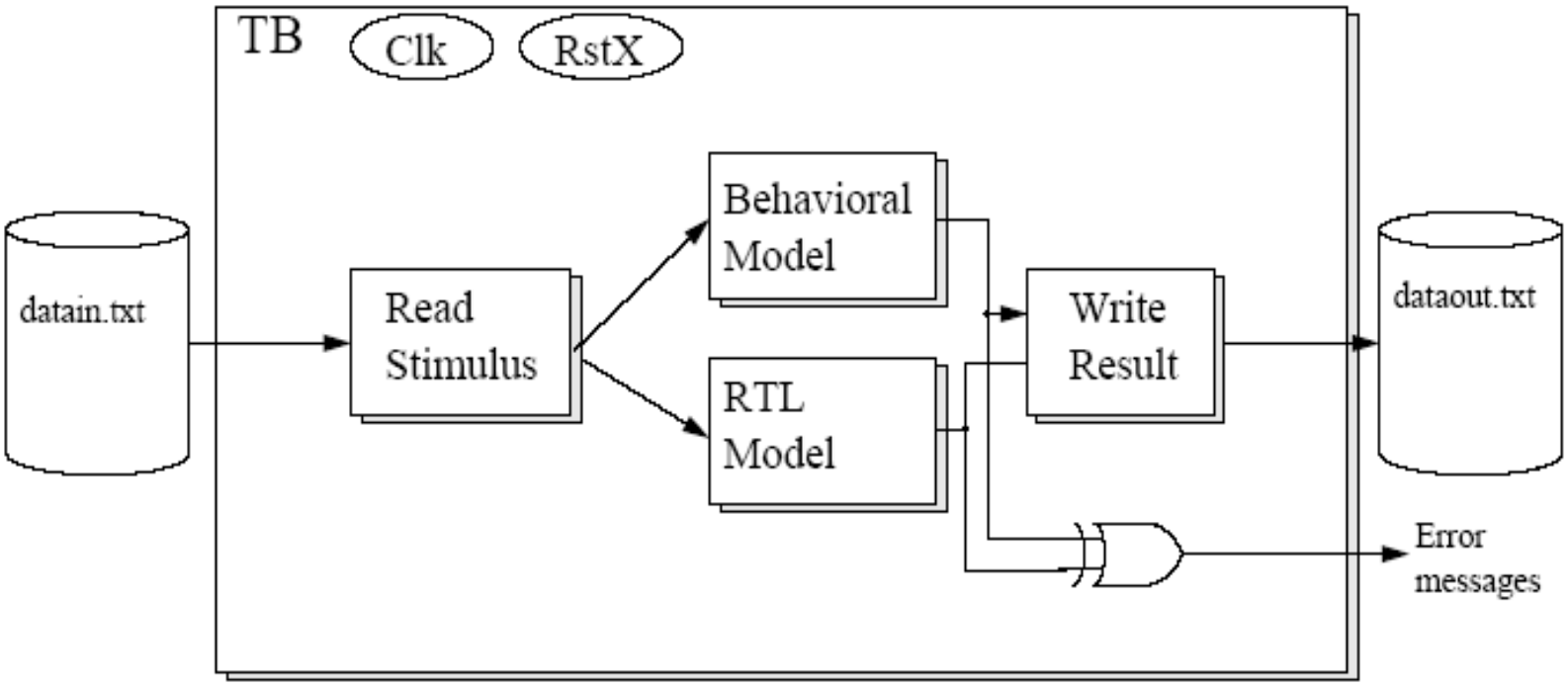
- DUT is compared to the specification i.e. the golden design.
  - Something that is agreed to be correct
  - E.g. Non-synthesizable model vs. fully optimized, pipelined, synthesizable DUT
- Special care is needed if DUT and Golden Design have different timing



Complex Test Bench



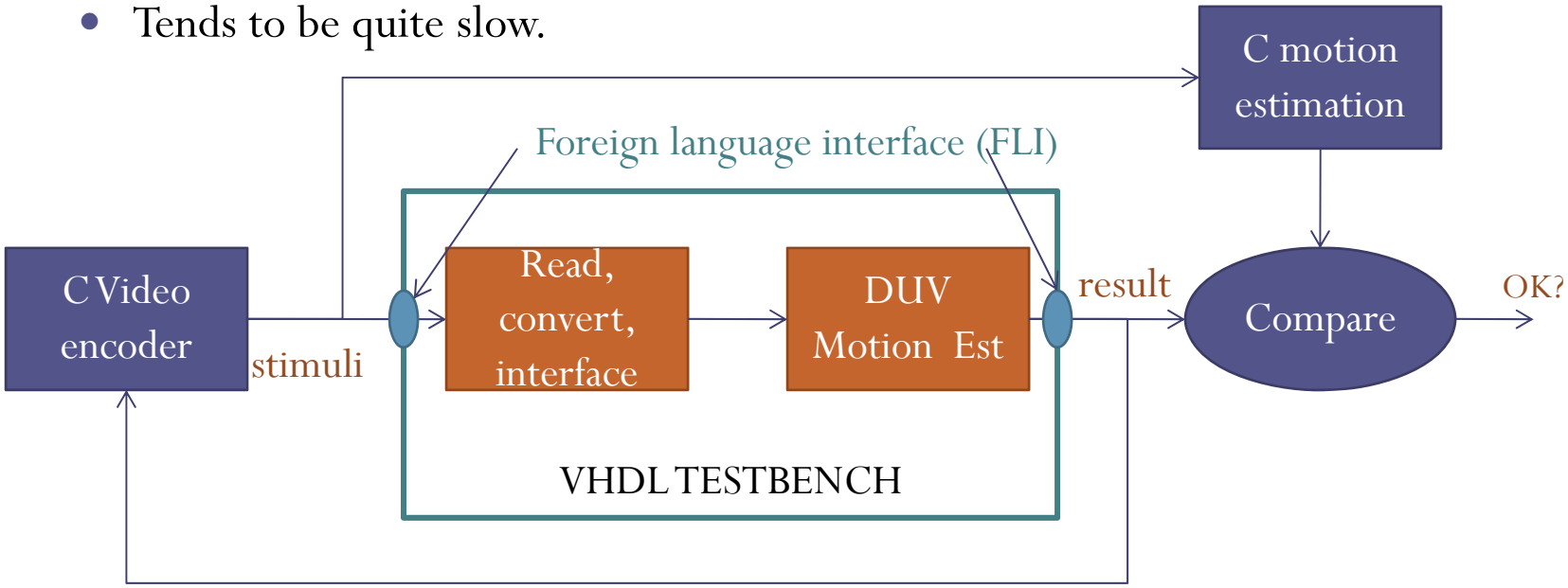
The Simulation with A Golden Design



For example, ensure that two models are equivalent. E.g. behavioral model for fast simulation and RTL model for efficient synthesis.

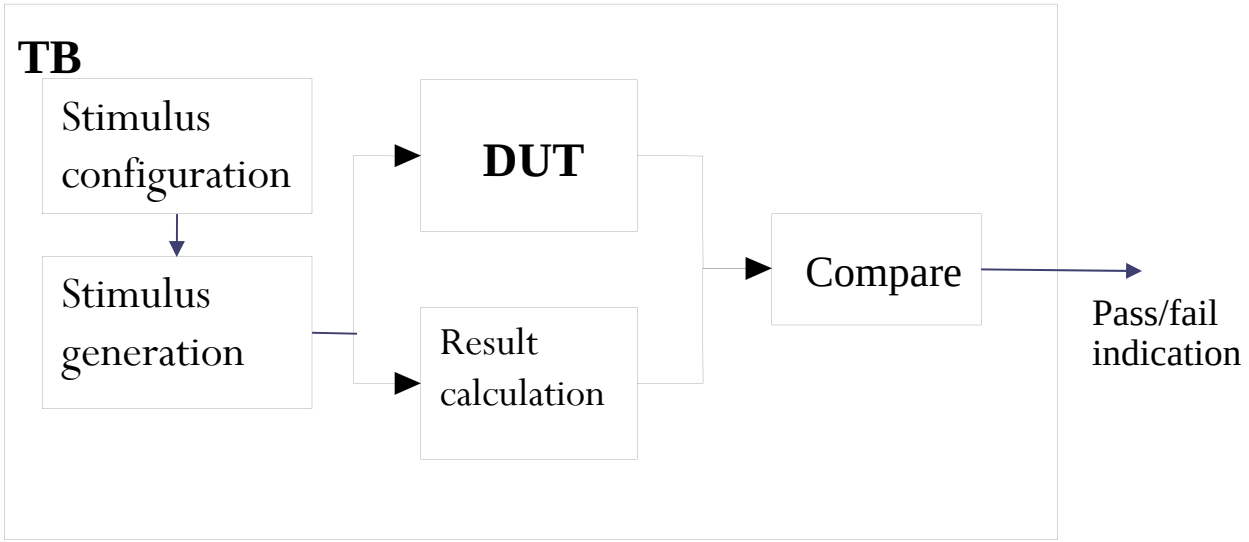
Example of golden design testbench

- Often, a system is first modeled with software and then parts are hardware accelerated
  - Software implementation serves as a golden reference
- E.g. Video encoder implemented with C, motion estimation accelerated
- Tends to be quite slow.



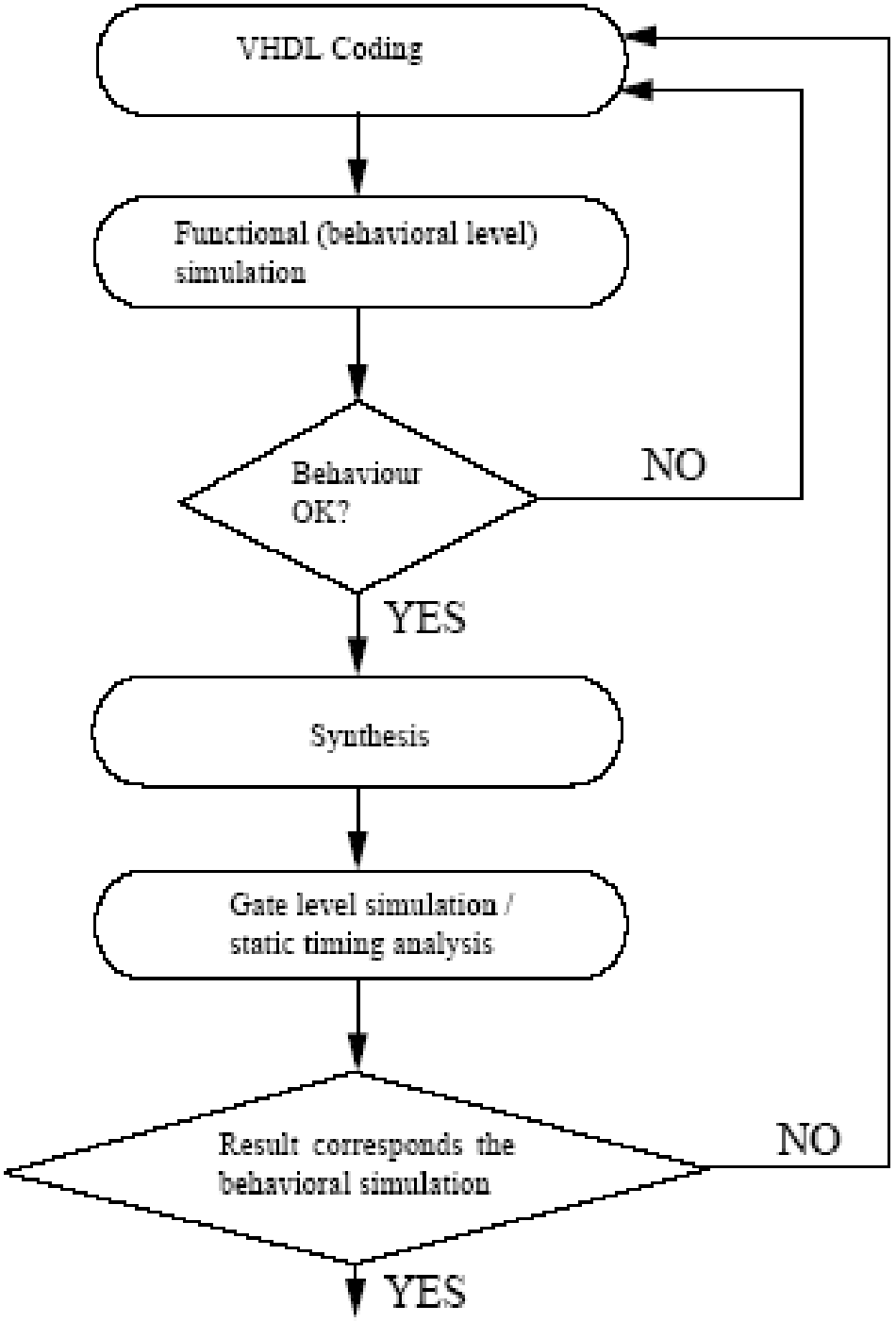
Autonomous test bench

- Does not need input file stimulus
- Determines the right result ”on-the-fly”
- Very good for checking if simple changes or optimizations broke the design
- Note that some (pseudo-)randomization on the stimuli must be done in order to make sure that the unusual cases are covered
  - Check the code, statement, and branch coverages!



static part of TB

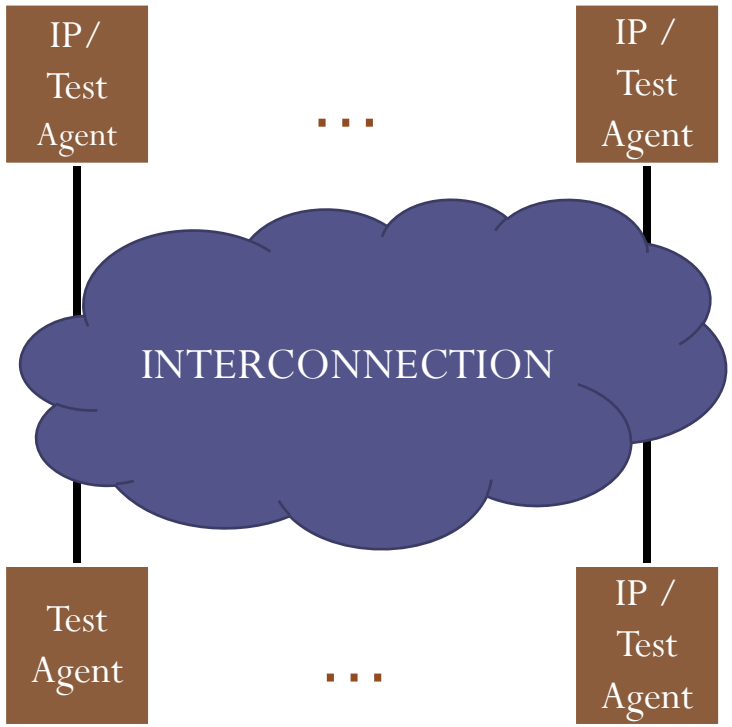
Design Flow


VHDL Coding
Functional (behavioral level) simulation
Behaviour OK?
NO
YES
Synthesis
Gate level simulation / static timing analysis
Result corresponds the behavioral simulation
NO
YES

Example and conclusions

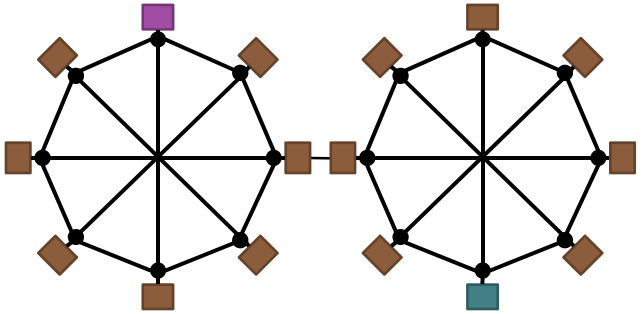
The test scenario

- DUV system-level interconnection network between IP blocks (CPUs, memories, accelerators…)
- TB must be expandable to very large configurations

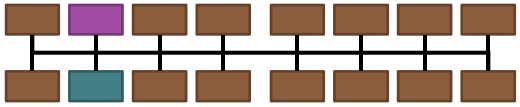


What is the interconnection?

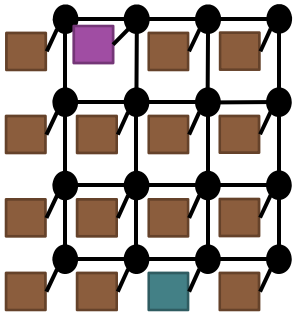
- **The interconnection topology does not matter** since we did not make assumptions about it, *only the functionality of the interconnection*
  a) Data arrives at correct destination
  b) Data does not arrive to wrong destination
  c) Data does not change (or duplicate)
  d) Data B does not arrive before A, if A was sent before it
  e) No IP starves (is blocked for long periods of time)



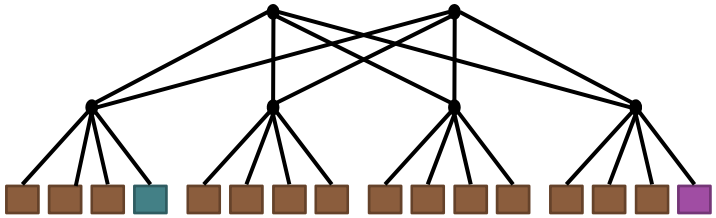
b) Hierarchical octagon



a)Typical: Single shared bus



c) Mesh



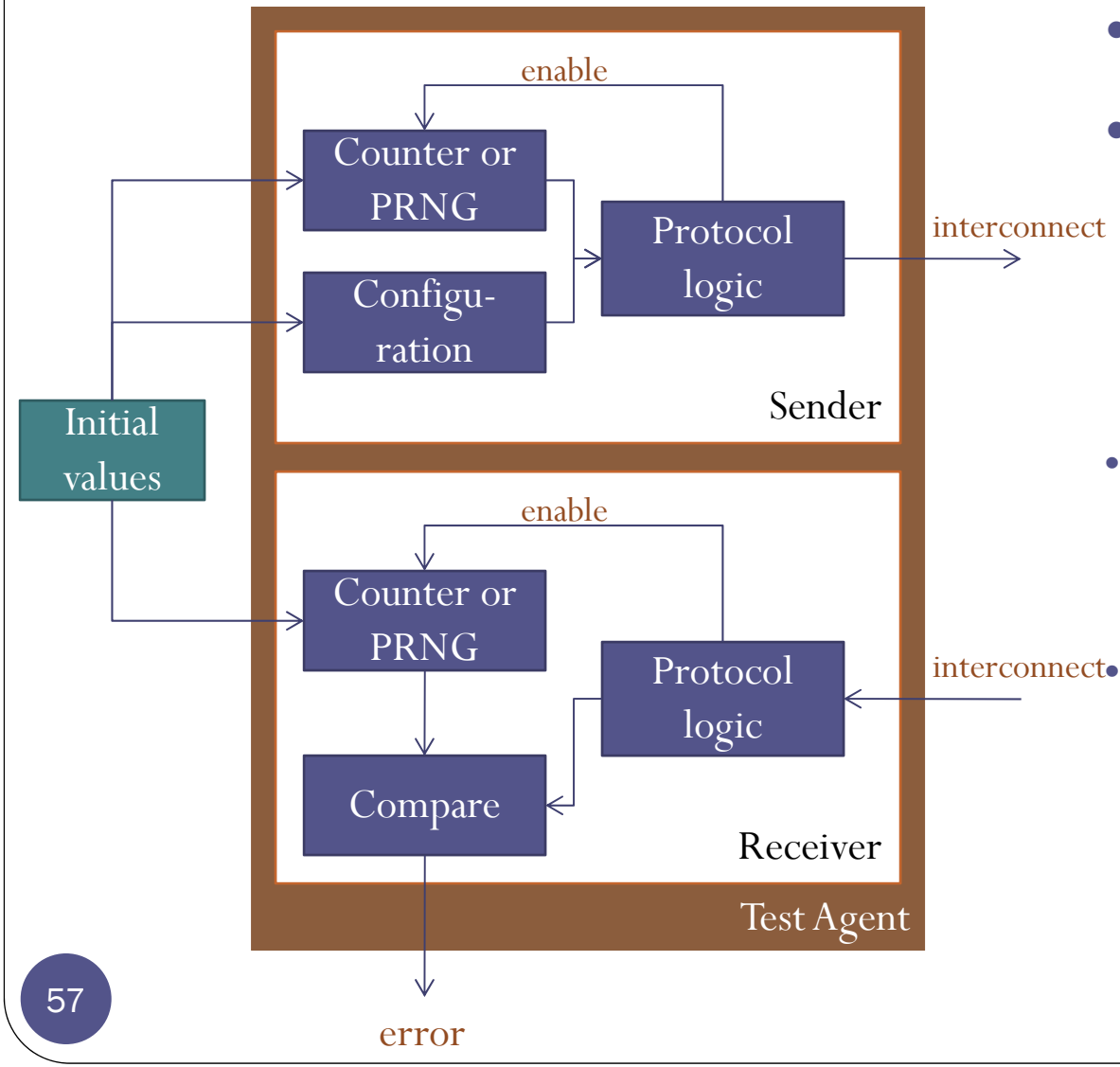
d) Butterfly

Source
Destination

Example: verifying an interconnection

- Tested interconnections delivers data from a single source to destination
  - Same principle, same IP interface, slightly different addressing
- Note that the only the transferred data integrity is important, not what it represents - Running numbers are great!
- The testbench should provide a few assertions (features a-d in prev slide)
- *When checking these assertions, you also implicitely verify the correctness of the interface!*
  - i.e. read-during-write, write to full buffer, write with an erroneous command etc.
- All of these can be fairly simply accomplished with an automatic test bench requiring no external files
- TB is pseudo-random numbers (
  - Longer simulation provides more thoroughness
  - The same run can be repeated because it is not really random
  - (Note that even if pseudo-random sequence is exactly the same, any change in DUV timing might mask out the bug in later runs)

Hierarchical interconnection testbench



- Separate the stimuli and verification
- Sender configuration per test agent-basis
  - Burst length (ie.sending several data in consecutive clock cycles)
  - Idle times
  - Destination
- Initial values:
  - Seed for counter / LFSR
  - Number of components
  - Addresses of the components
- Sender and Receiver
  - Counter or PRNG needed for each source and/or desstination!
  - (PRNG = pseudo-random number generator)

Autonomous and complex test benches

- Always a preferred choice - Well designed, resusable testbench pays back
- Use modular design
  - Input (stimuli) separated from output (check) blocks in code
  - Arbitrary number of test agents can be instantiated
  - Interconnection-specific addressing separated from rest of the logic
- All testbenches should **automatically check for errors**
  - No manual comparison in any stage
- Code coverage must be high
  - However, high code coverage does not imply that the TB is all-inclusive, but it is required for that!
  - Autonomous test benches must include long runs and random patterns to make sure that corner cases are checked
- Designing the test benches in a synchronous manner makes sure that the delta delays do not mess things up
  - Synchronous test bench also works as the real environment would
  - More on the delta delay on next lecture about simulators

Example VHD:

- Traffic light test bench
- Statement, Branch, Condition and expression coverage 100%
- However, the test bench is not perfect!
- Example VHDL code shown (available at web page)
- General test bench form

```
begin  -- tb

  -- component instantiation
  DUV : traffic_light ...

  input : process (clk, rst_n)
  begin  -- process input
   ...
  end process input;

  output: process (clk, rst_n)
  begin  -- process output
   ...
  end process output;

Clock generation
Reset generation

end tb;
```

Synthesizable testbenches

- Synchronous, synthesizable test benches are a good practice if e.g. the design includes clock domain crossings
- Can be synthesized to a FPGA and tested in a real environment
- Run-time error checking facilities to a real product may be extracted from the test bench easily
- Then, assertion should be written this way:

```
If (a > b) then
  assert false "error" …
  error_out(0) <= '1';
End if;
```

  - Or one can make own assertions for simulation and synthesis, e.g.

    `Assert_gen(cond, level, error_sig);`

  - In simulation, regular assertion, in synthesis, assigns to `error_sig`

Summary and general guidelines

- **Every entity you design has an own test bench.**
- Automatic verification and result checking
  - Input generated internally or from a file
  - Output checked automatically
  - The less external files we rely on, the easier is the usage
    - Somebody else will also be using your code! e.g.
      - ”Vsim my_tb; run 10ms; ” ➔ ”tests ok”
      - or just type ”make verify”
- Timeout counters detect if the design does not respond at all!
  - You must not rely that the designer checks the waves