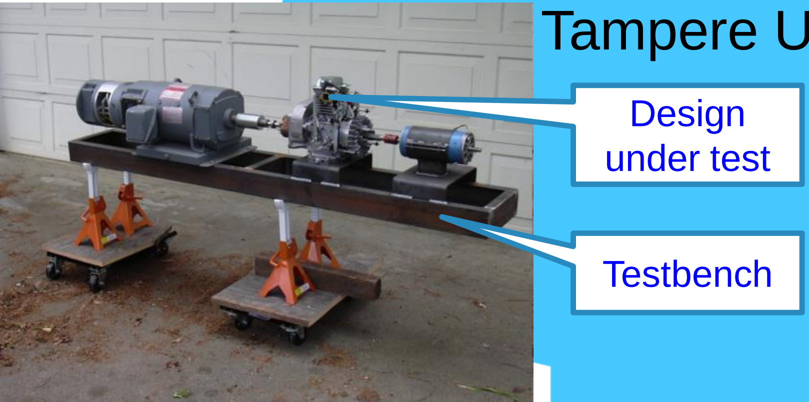
TAMPERE UNIVERSITY OF TECHNOLOGY

Lecture 8: VHDL Test Benches

TIE-50206 Logic Synthesis
Arto Perttula
Tampere University of Technology
Fall 2017



Contents

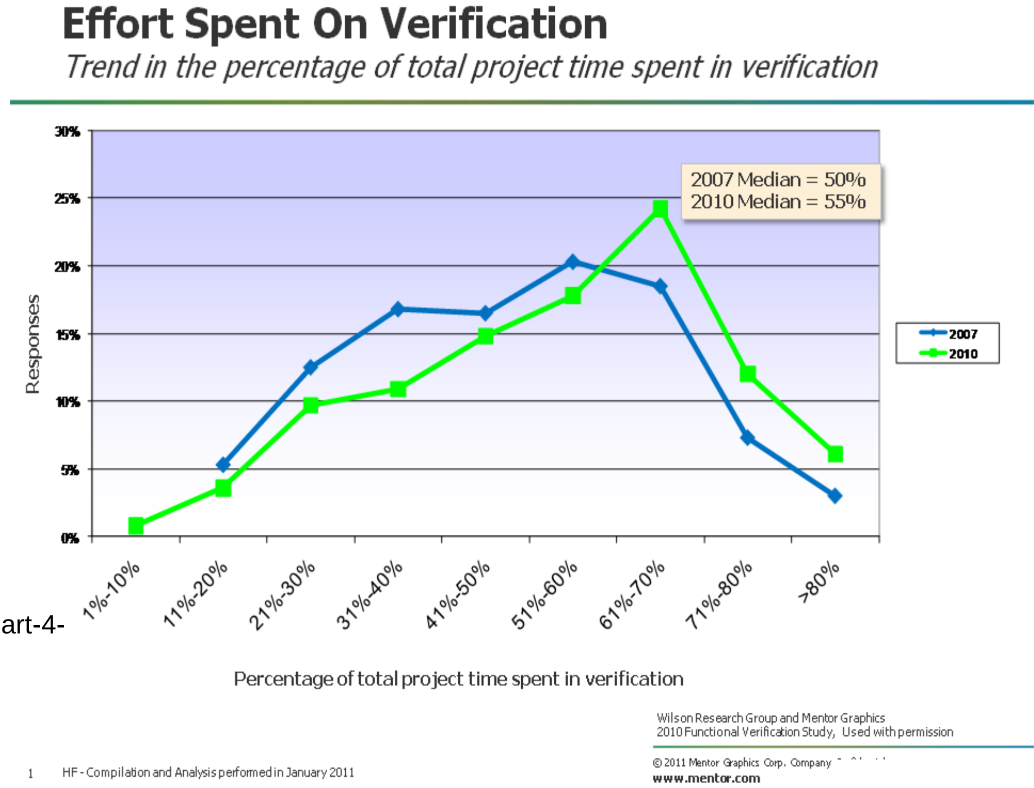
- Purpose of test benches
- Structure of simple test bench
    - Side note about delay modeling in VHDL
- More elegant test benches
    - Separate, more reusable stimulus generation
    - Separate sink from the response
    - File handling for stimulus and response
- Example and conclusions

- Lots of miscellaneous self-study material

Introduction

- Verification is perhaps the most difficult aspect of any design
  - That’s not an excuse for omitting it or leaving to others…
  - Multiple levels: single component, module with multiple sub-components, and system-level
- Multiple abstraction levels
- In synchronous design, we verify the *functionality at cycle-level accuracy*
  - Not detailed timing, which will be checked with static timing analysis (STA) tools

[http://blogs.mentor.com/verificationhorizons/blog/2011/04/03/part-4-the-2010-wilson-research-group-functional-verification-study/slide13-2-2/}



Effort Spent On Verification

Trend in the percentage of total project time spent in verification

2007 Median = 50%
2010 Median = 55%

Wilson Research Group and Mentor Graphics 2010 Functional Verification Study, Used with permission

1 HF - Compilation and Analysis performed in January 2011

© 2011 Mentor Graphics Corp. Company

www.mentor.com

Introductory Question

Q: What’s the difference between theory and practice?

A1: In theory there’s no difference…

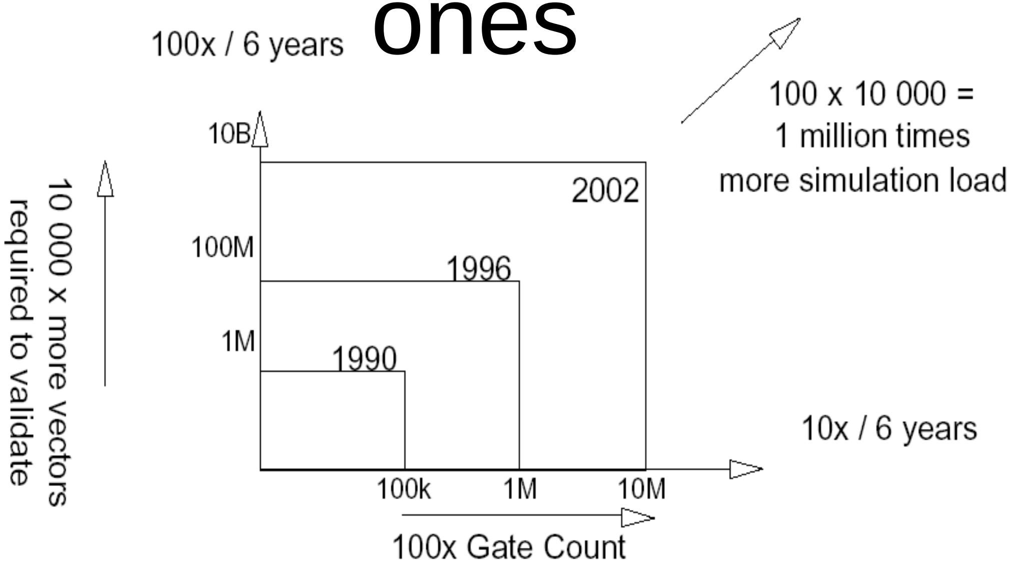
A2: In theory everything works. In practice nothing works.

Validation versus Verification versus Testing

- **Validation**: Does the product meet customers’ wishes?
  - Am I building the right product?
- **Verification**: Does the product meet the specification?
  - Am I building the product right?
  - Debugging begins when error is detected
- **Testing**: Is chip fabricated as meant to?
  - No short-circuits, open connects, slow transistors etc.
  - Post-manufacturing tests at the silicon fab
  - Accept/Reject
- However, sometimes these are used interchangeably
  - Most people talk about *test benches*
  - E.g., both terms are used: DUT (design under test) and DUV (design under verification)

At First

Make sure that simple things work before even trying more complex ones

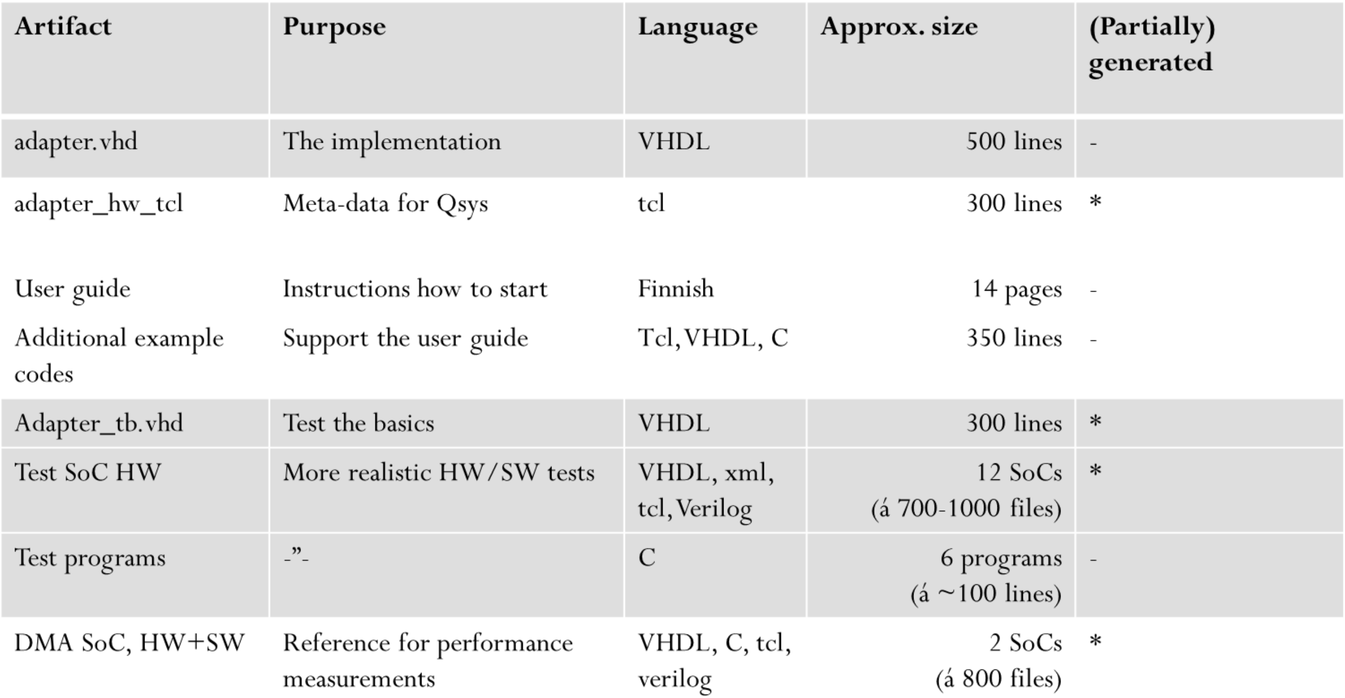


Verification effort growth outpaces design

[P. Magarshack, SoC at the heart of conflicting, Réunion du comité de pilotage (20/02/2002), trendshttp://www.comelec.enst.fr/rtp_soc/reunion_20020220/ST_nb.pdf]

a.k.a. ”state explosion”

Amount of Test Codes And Support Material Exceeds Implementation

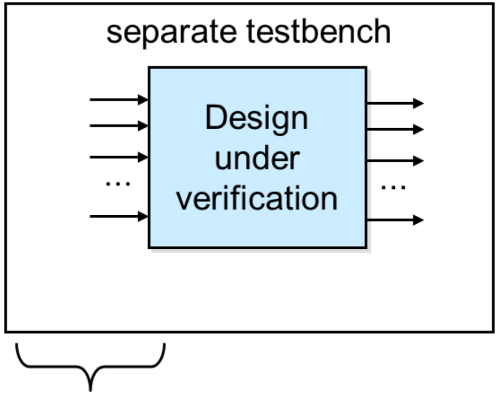


Artifact	Purpose	Language	Approx. size	(Partially) generated
adapter.vhd	The implementation	VHDL	500 lines	-
adapter_hw_tcl	Meta-data for Qsys	tcl	300 lines	*
User guide	Instructions how to start	Finnish	14 pages	-
Additional example codes	Support the user guide	Tcl, VHDL, C	350 lines	-
Adapter_tb.vhd	Test the basics	VHDL	300 lines	*
Test SoC HW	More realistic HW/SW tests	VHDL, xml, tcl, Verilog	12 SoCs (á 700-1000 files)	*
Test programs	-”-	C	6 programs (á ~100 lines)	-
DMA SoC, HW+SW	Reference for performance measurements	VHDL, C, tcl, verilog	2 SoCs (á 800 files)	*

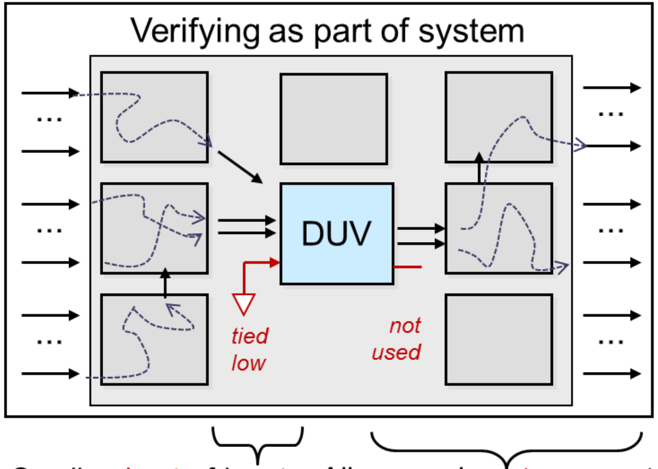
Table 6.1 Developed artifacts [T. Korpela, Adapter for distributed and shared memory between HIBI and Avalon, MSc thesis, TUT, 2014]

Controllability and Observability

- How to provide inputs and see the results
- Both properties favor the *low-level verification*
  - Exception: mimicking realistic input patterns might be more difficult than using real neighbour block



All input sequences can be generated. Also illegal!



Small subset of input sequences achievable. Designer must control DUV's inputs via other units

All cases do not propagate to observable outputs, or it takes time and goes through several units

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!

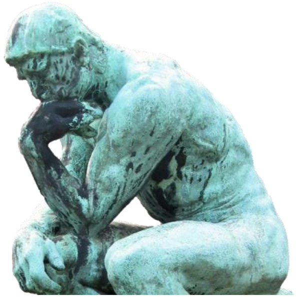
Test Bench Benefits

- 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 emergency shutdown first in laboratory than in real chemical factory

Stimulus and Response

- TB may **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
  - Results are compared to Matlab simulations automatically, no manual comparison!

Philosophical Note [after Keating]



- Verification can find bugs, prove equivalence, and prove interesting properties of a design
- Verification *cannot prove correctness*
  - It can show the *existence of bugs*, but not their non-existence
  - We can show cases when the design works
- Nevertheless, we do achieve high quality through verification
- Quality needs to be designed in (and then verified), not verified in
- Completion condition should be explicitly stated
  - E.g., statement/coverage/state coverage > 98%, #new bugs/week <0.5, all time/money spent, we're bored with it…

Other Philosophical Note

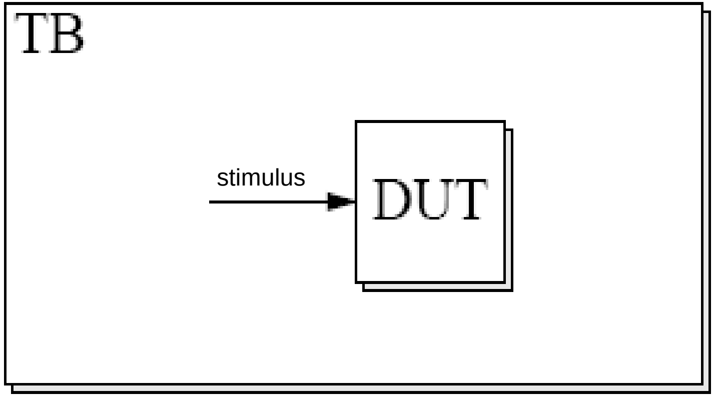
- TB tests the DUT against the *interpretation of specification (which is interpreted from requirements)*
  - Specification may have flaws
    - Ambiguity
    - Not meeting the customer's desires
  - Test bench may have mistakes
    - False interpretation
    - Test bench codes may have bugs
- Good to have different persons writing the actual code and test bench
  - Less likely that both make the same miss-interpretation

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: “That is a good question indeed”

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 usefull…”

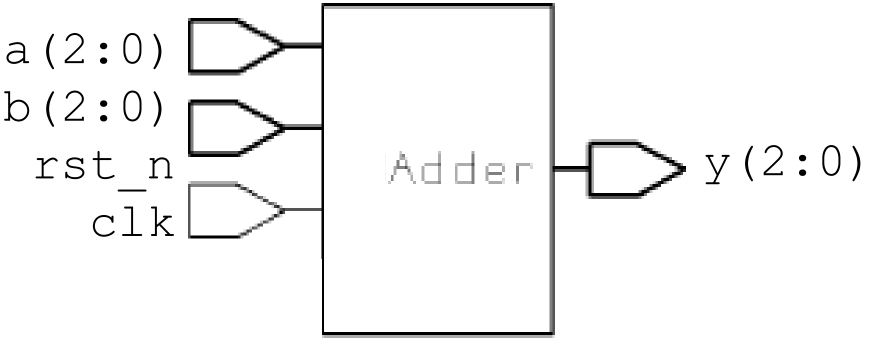


Better than none, but not reliable

Example DUT

- 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;
```



Simple TB (3): Entity without Ports

- 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;
```

Simple TB (4): Instantiate DUT And Generate Clock

- 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;
```

Simple TB (5): Stimulus And Config

- 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;
```

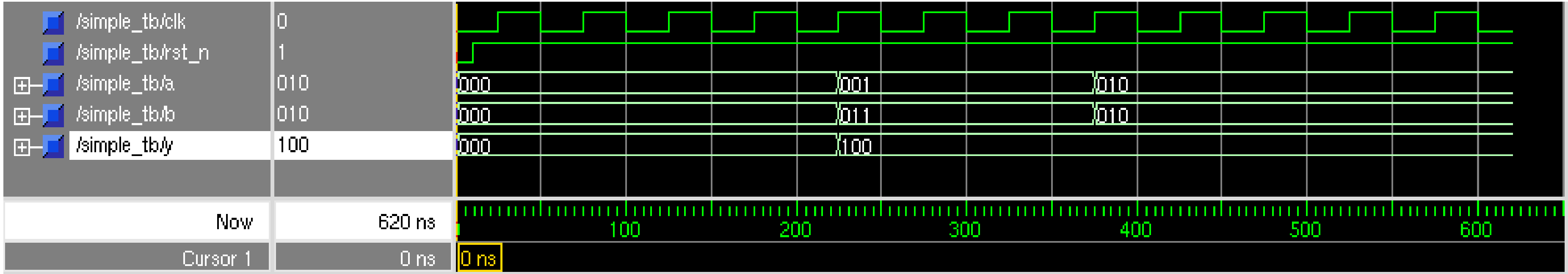
Simple TB (6): Simulation Results

- Simulation:

How to check correcteness?

Take a look!

Not too convenient…



/simple_tb/clk 0
/simple_tb/rst_n 1
/simple_tb/a 010
/simple_tb/b 010
/simple_tb/y 100

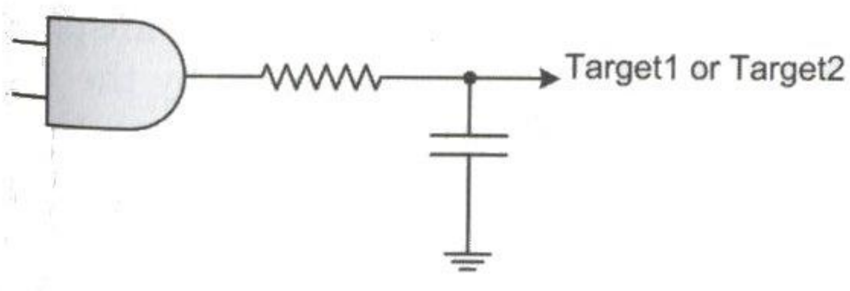
Now 620 ns
Cursor 1 0 ns

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

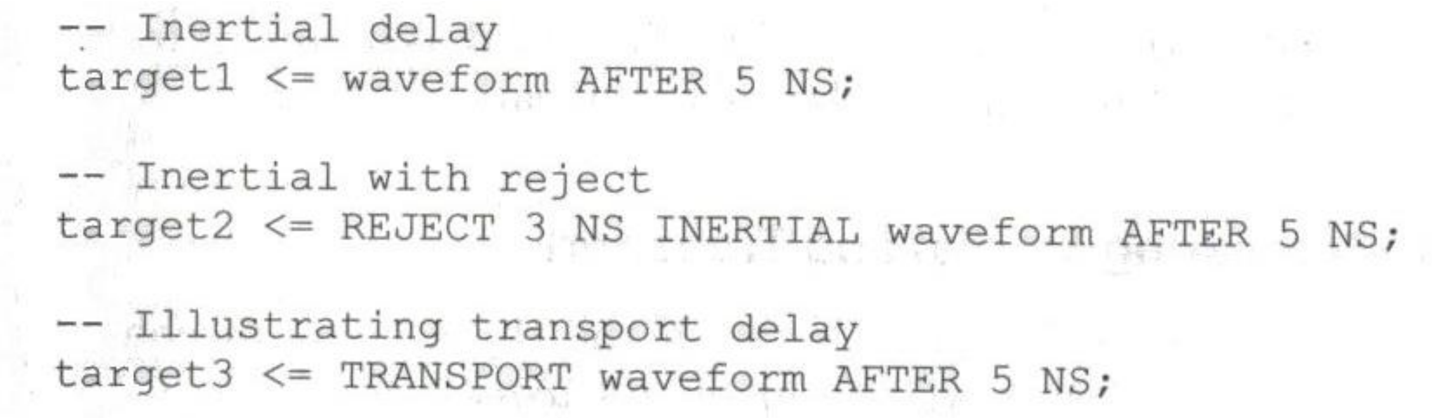
VHDL Delay Modeling

- Signal assignments can have delay (as in previous 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
   - test bench stimuli generation
   - glitches remain



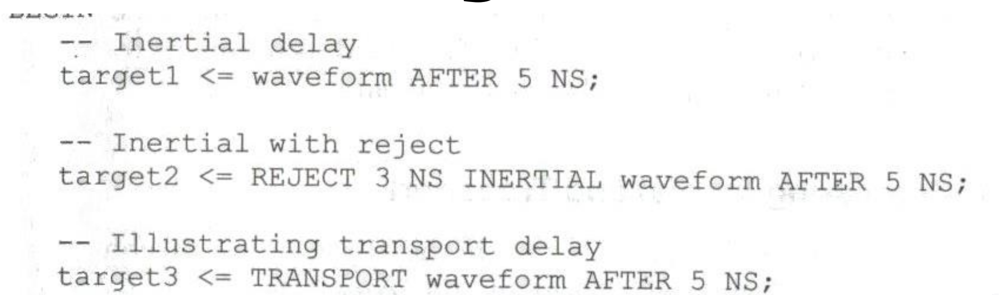
VHDL Delay Modeling (2)



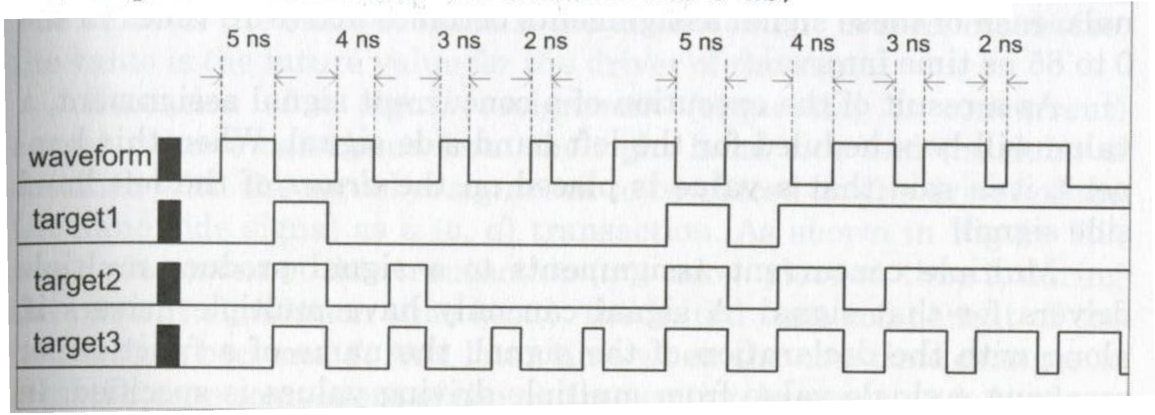
```
-- Inertial delay
target1 <= waveform AFTER 5 NS;

-- 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)



```
BEGIN
    -- 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;
```

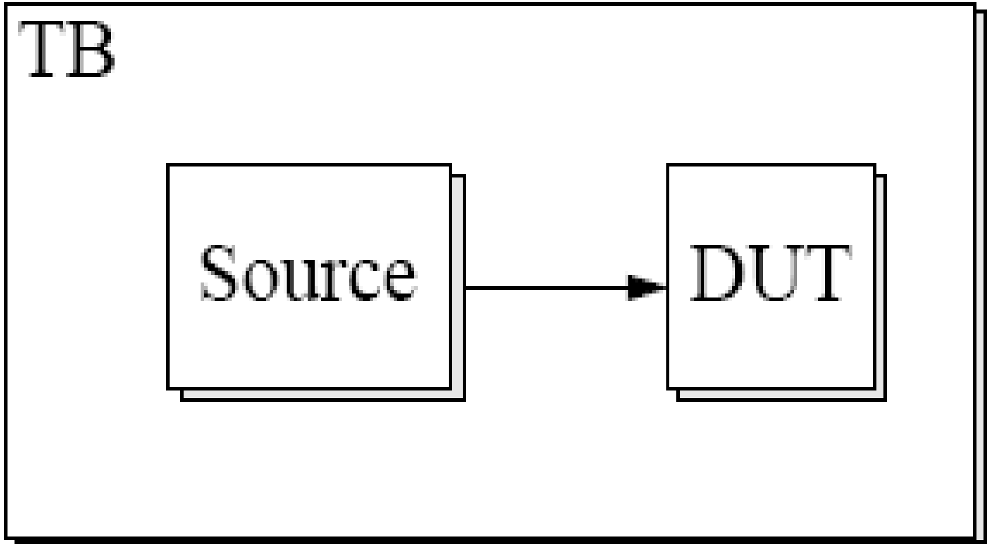


Be careful! Behavior will be strange if the edges of the clk and signal generated this way are aligned.

MORE ELEGANT TEST BENCHES

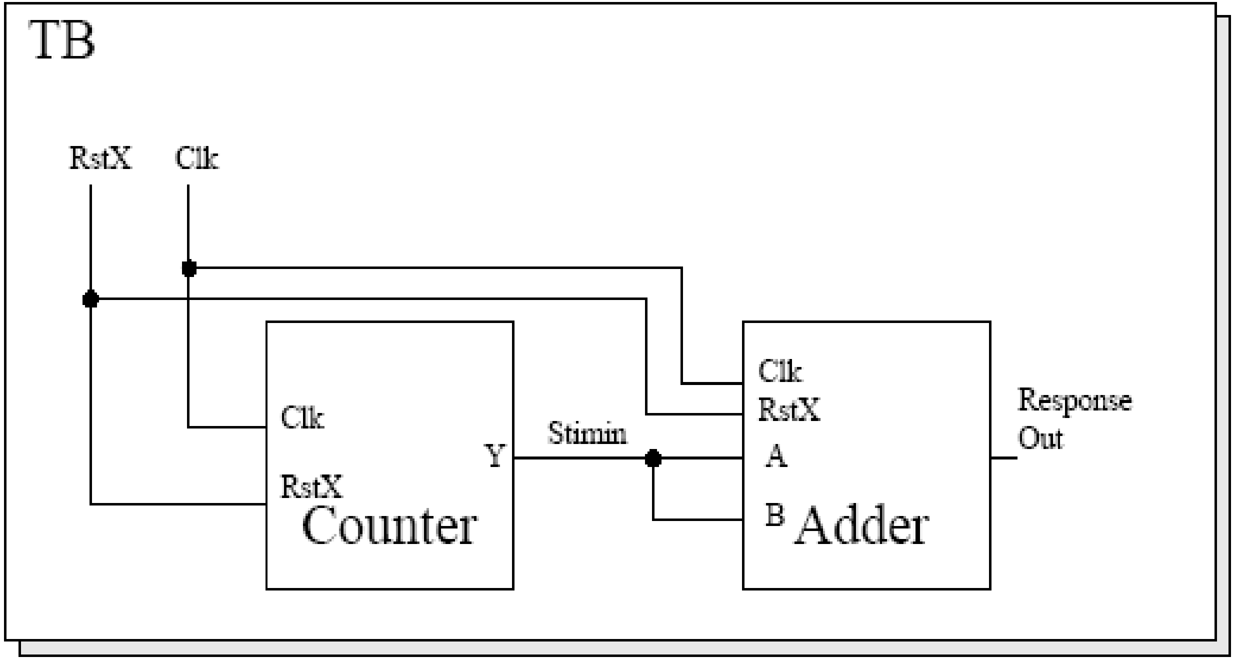
Test Bench with a Separate Source

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



Test Bench with a Separate Source (2): Structure

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



Test Bench with a Separate Source (3): Create Counter

- 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;
```

Test Bench with a Separate Source (4): Create Counter

- 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

Test Bench with a Separate Source (5): Declare Components

- 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;
```

Test Bench with a Separate Source (6): Instantiate

- 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;
-- simplification, generally should be different from stim_a_in
```

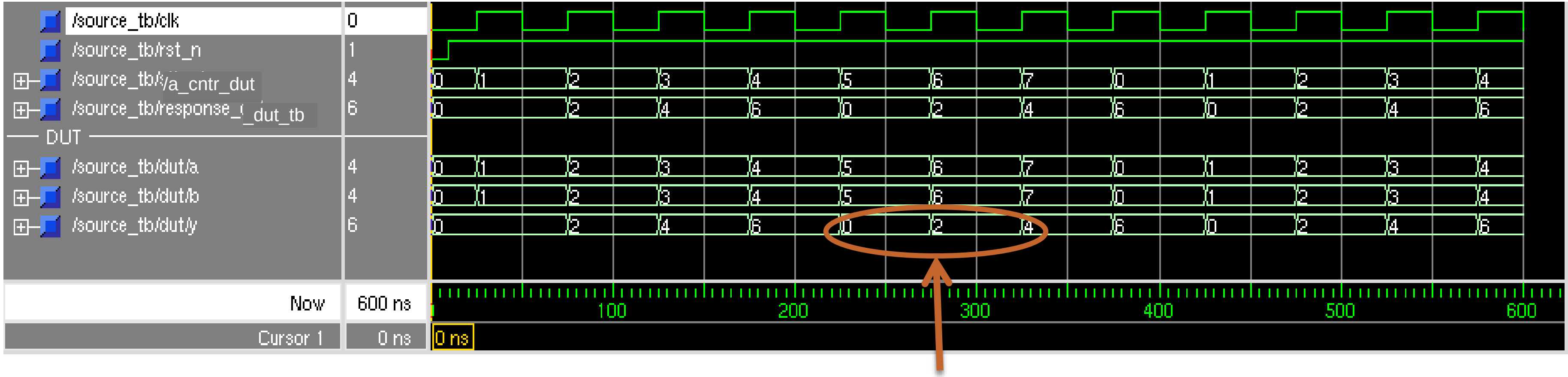
Test Bench with a Separate Source (7): Clock And Reset

- 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
```

Test Bench with a Separate Source (8): Simulation Results

- Simulation:



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

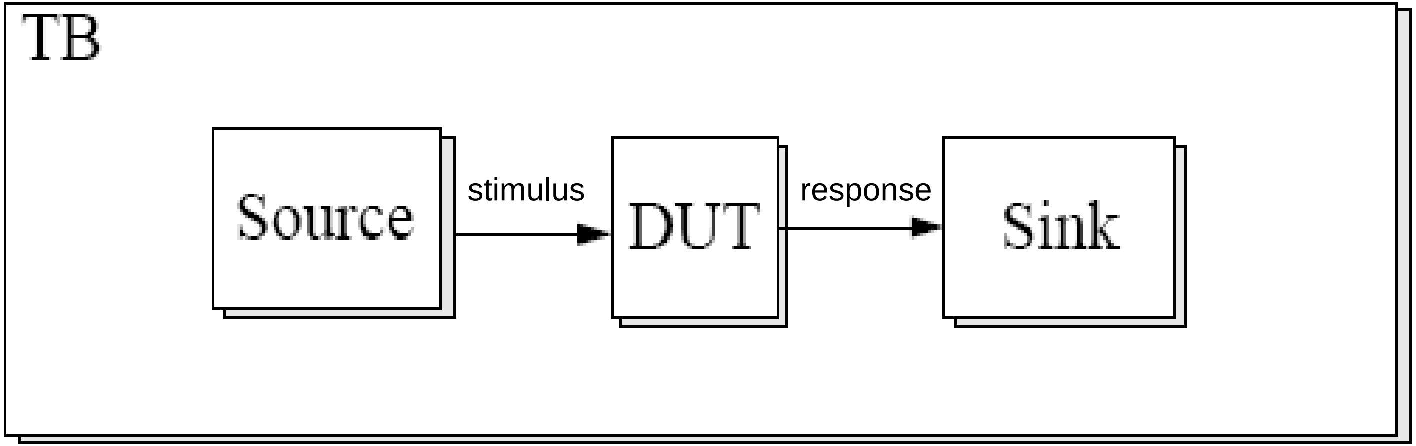
Overflow in adder, three bits → unsigned value range 0..7

The checking is still inconvenient.

RESPONSE HANDLING

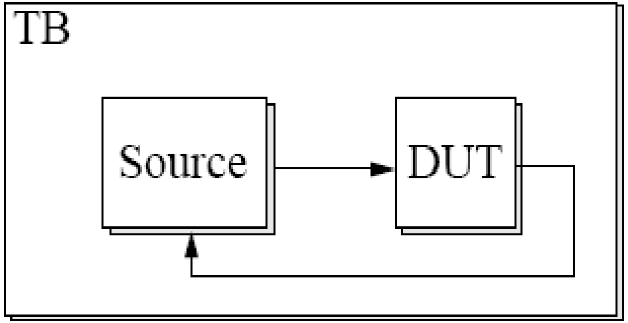
Test Bench with a Separate Source And Sink

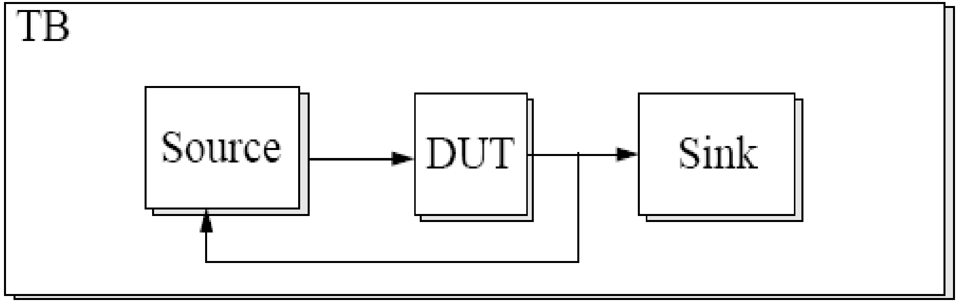
- Both the stimulus source and the sink are separate instances
- Complex source and sink without response-source interaction
- Sink uses assertions of some sort



Smart Test Bench

- Circuit’s response affects further stimulus
- In other words, TB is *reactive*
  - E.g., DUT requests to stall, if DUT cannot accept new data at the moment
  - E.g., source writes 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





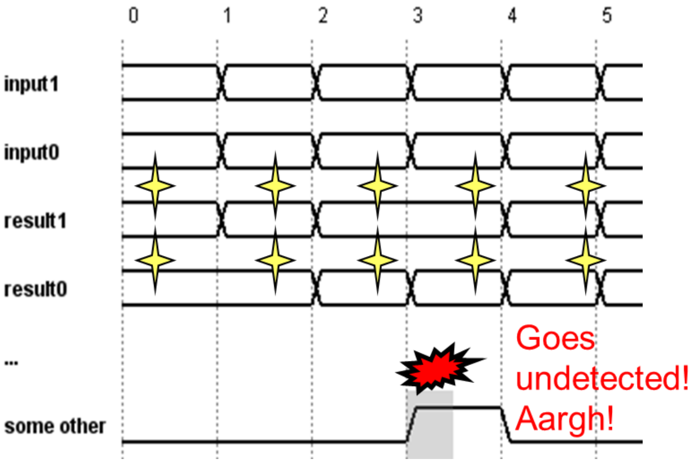
Presence of Expected Results + Non-Existence of Bad Side-Effects

- 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 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 control registers were modified unexpectedly)
- Usually there is one or perhaps few entirely correct results and infinite number of wrong ones

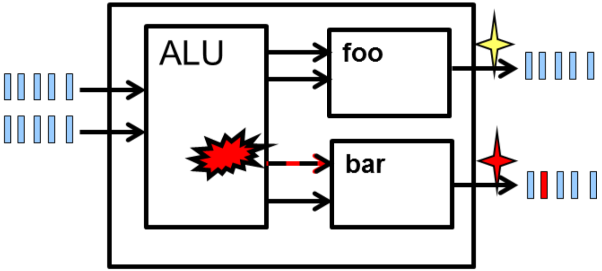
Check Absence of Side-Effects! (2)

- TB correctly checked the upper outputs of the ALU (those affected with given stimulus)
- Implicit, false assumption was not verified!
- Glitch in lower signals goes undetected in TB but causes problems in real SoC
  - E.g., divide-by-zero rises during add operation

TB waves:



SoC fails



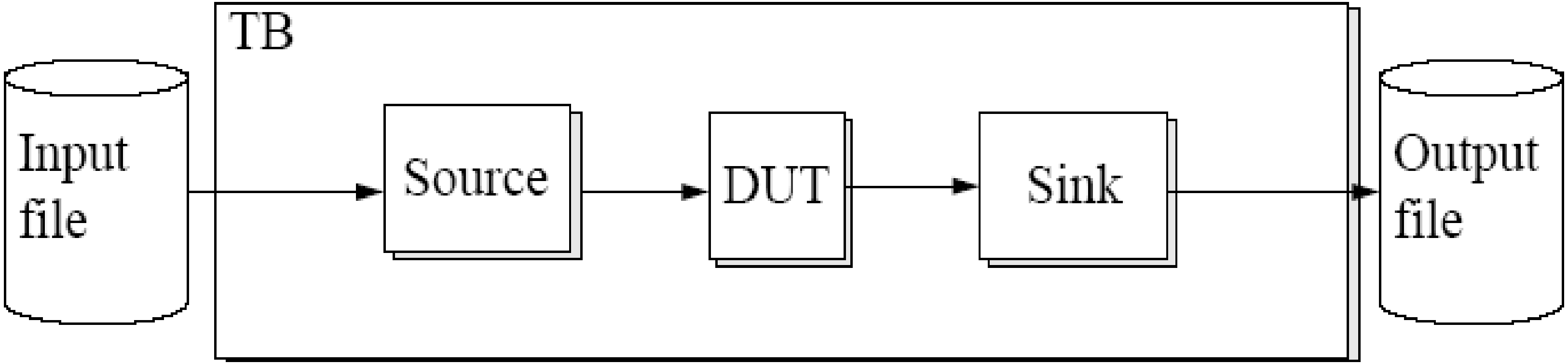
Component *Bar* is innocent but becomes the suspect at first when SoC fails!

Mutation Testing

- Q: How do you know if your TB really catches any bugs?
- A: Create intentional bugs and see what happens
- Examples
  - Some output is stuck to ’0’ or ’1’
  - Condition `if (a and b)` then becomes `if (a or b)`
  - Comparison > becomes >=
  - Assignment `b <= a` becomes b `<= a+1`
  - Loop iterates 1 round less or more
  - State machine starts from different state or omits some state change
- Automated mutation tools replace manual work
- If mutated code is not detected, the reason could be
  a) Poor checking
  b) Too few test cases
  c) Codes were actually functionally equivalent

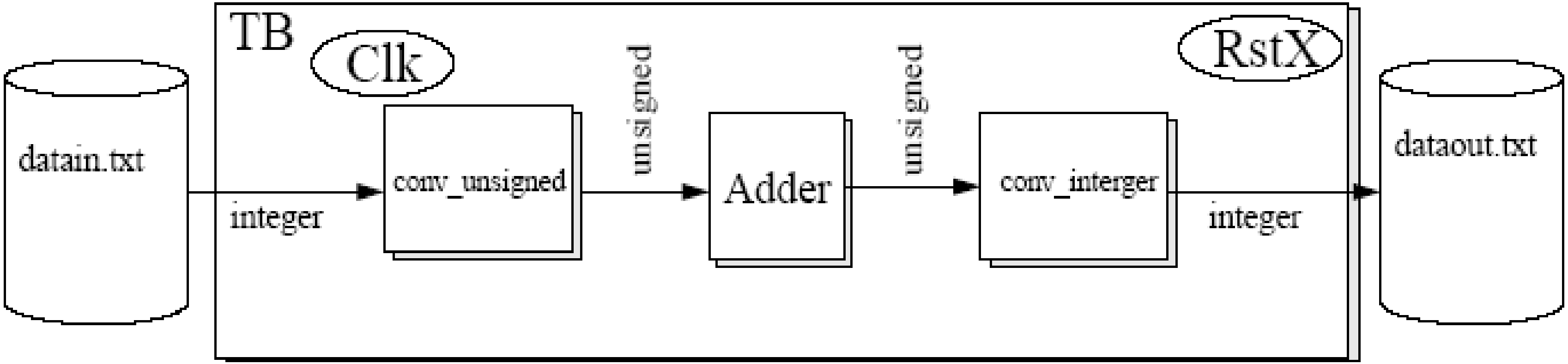
Test Bench with Text-IO

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



Test Bench with Text-IO (2): Structure

- Test case:



Test Bench with Text-IO (3): Libraries

- 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;
```

Test Bench with Text-IO (4): Declarations

- 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);
```

Test Bench with Text-IO (5): Clock, Reset, Instantiation

- In architecture 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;
```

Test Bench with Text-IO (6): Process for File Handling

- 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;
```

Test Bench with Text-IO (7): Main Process

- 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);
```

Test Bench with Text-IO (8): Handle I/O Line by Line

- ”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”

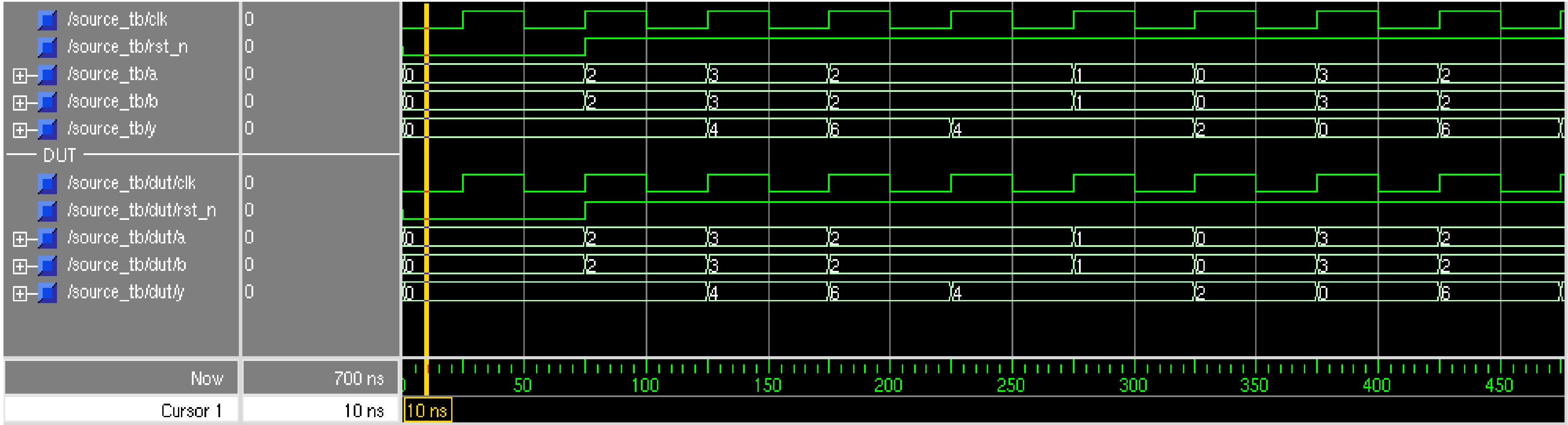
```
WRITE    (line_out, output_tmp);
WRITELINE(file_out, line_out);
```

- 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;
```

Test Bench with Text-IO (9): Simulation Results

- Simulation:

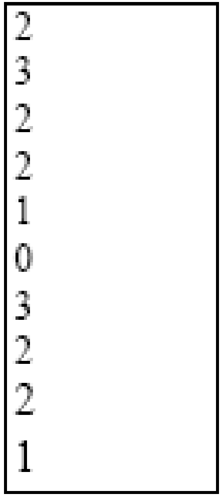


Now, the designer can prepare multiple test sets for certain corner cases (positive/negative values, almost max/min values, otherwise interesting) . However, the VHDL is not modified.

This version does not check the response yet.

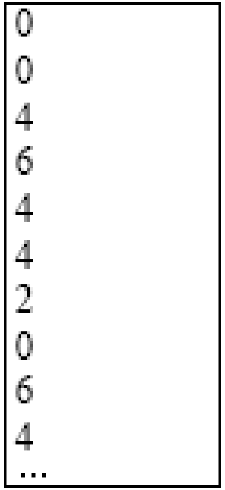
Test Bench with Text-IO (10): Files

- Input file provided to test bench



```
2
3
2
2
1
0
3
2
2
1
```

- Output file produced by the test bench



```
0
0
4
6
4
4
2
0
6
4
...
```

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…

Use Headers in Input And Output Files

- Fundamental idea is to have many input, reference output, and result files
  - Provide clear links between these
- Use headers!
- E.g., input file
- # File: ov1_in.txt
- # Purpose: Test overflow…
- # Designer: Xavier Öllqvist
- # Date: 2013-12-24 16:55:02
- # Version: 1.1
- # Num of cases: 430
- # Line format: hexadecimal…

- E.g., DUV output log could provide summary
- # File: ov1_out.txt
- # Input File: ov1_in.txt
- # Time: 2014-02-03 14:24:33
- # User: xavi
- # Format:…
- 0
- 0
- 4…
- # Sim ends at: 100 ms
- # Num of cases: 430
- # Errors: 0
- # Throughput: 25.4 MB/s
- . . .

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
- Tip: In ModelSim, double-clicking a print message takes the cursor in wave window to the correct time instant

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));
```

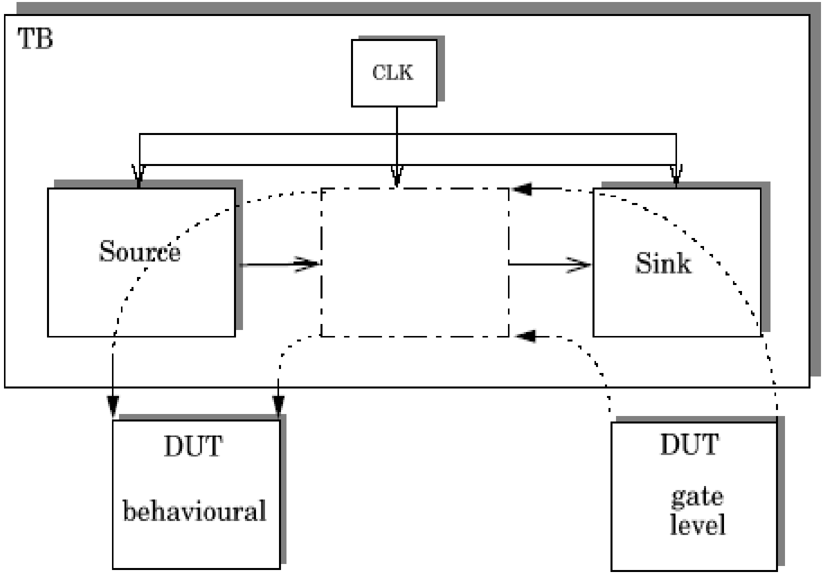
or

```
report “Value is “
& integer‘image(to_integer(unsigned(data_vec)));
```

- It is recommended to print which value the test bench *expected* in addition to what is *actually got* (”Exp: 512, Got: 511”)
  - Easier to see, e.g., off-by-one errrors (value differs ±1), wrong timing (±1 clock cycle), overflow, tb bugs

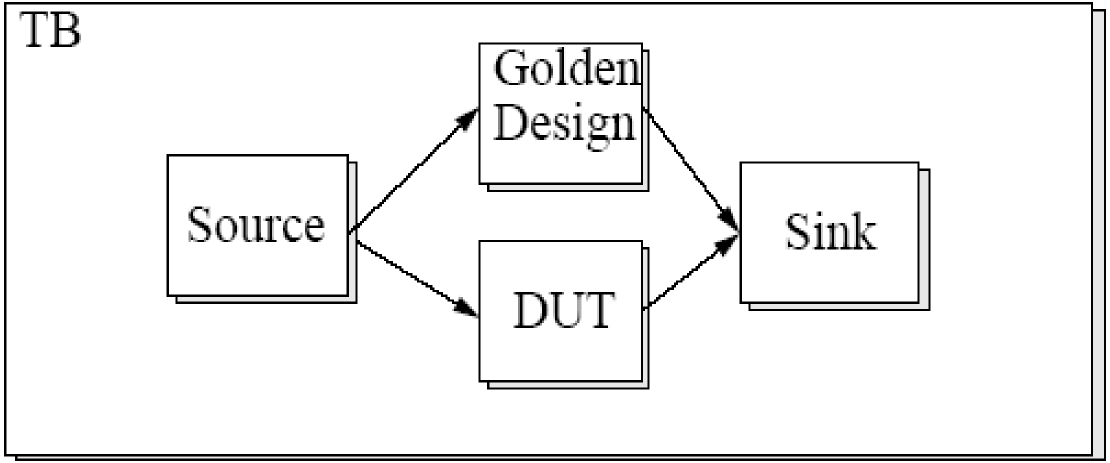
Same Test Bench for Different Designs

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

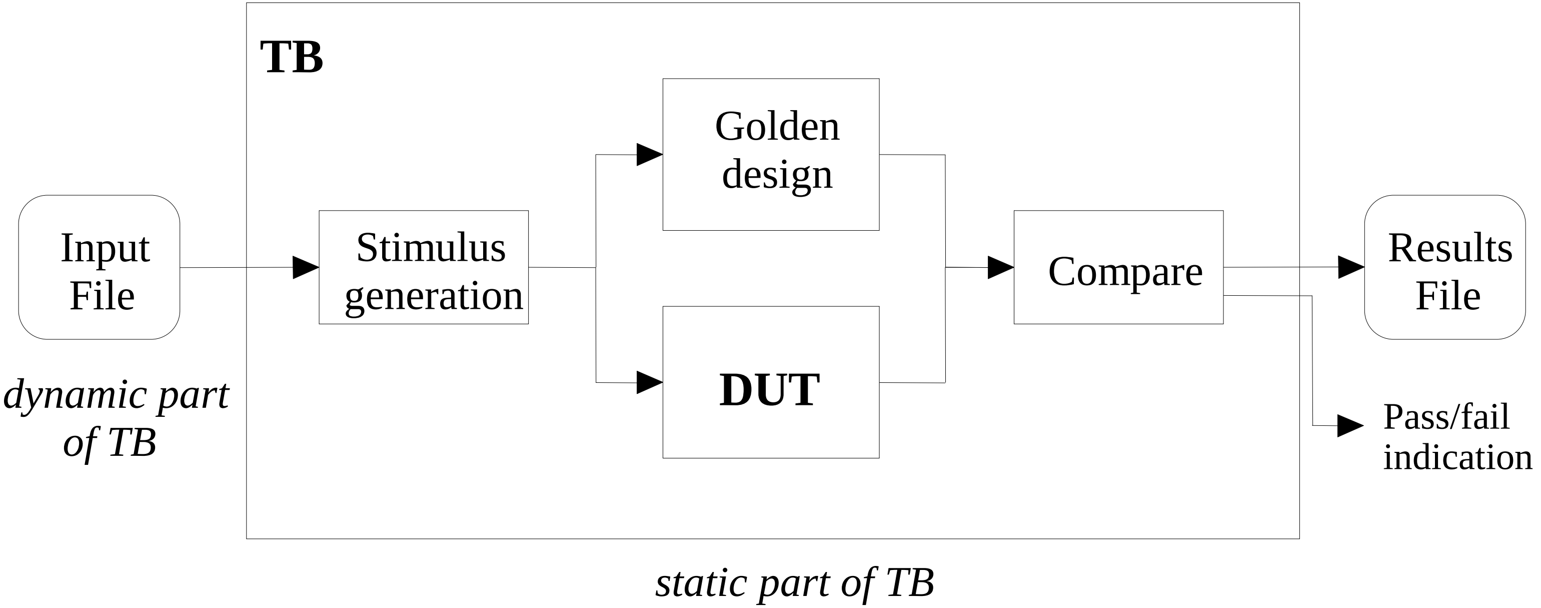


Golden Design

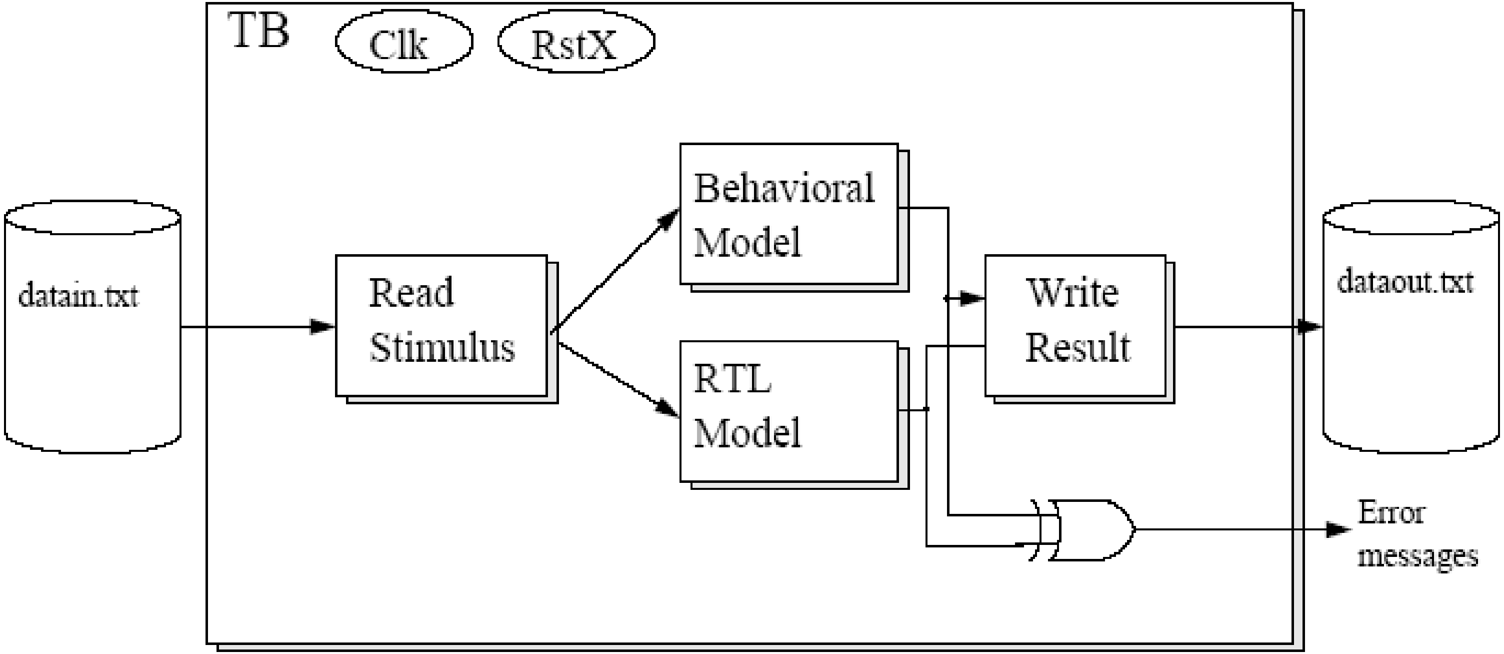
- DUT is compared to the specification, i.e., the golden design
  - Something that is agreed to be correct
  - E.g., non-synthesizable model vs. fully optimized, pipelined, synthesizable DUT
- Special care is needed if DUT and Golden Design have different timing



Complex Test Bench



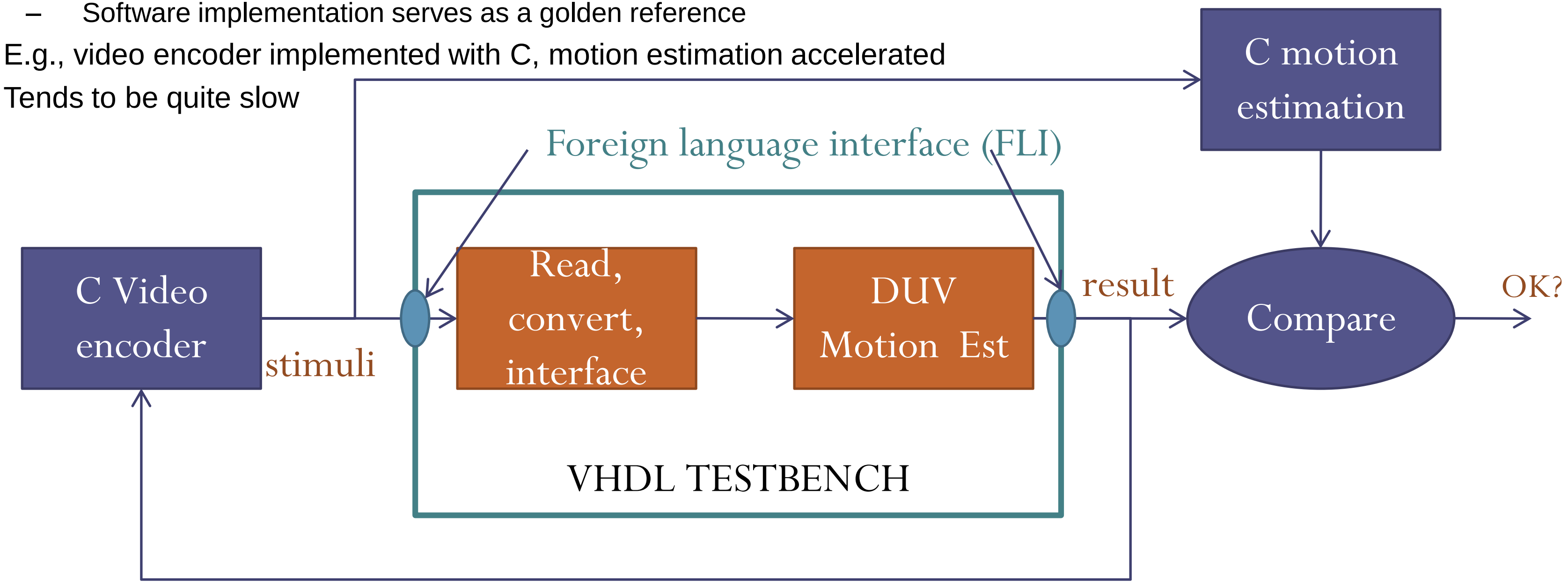
The Simulation with a Golden Design



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 Test Bench

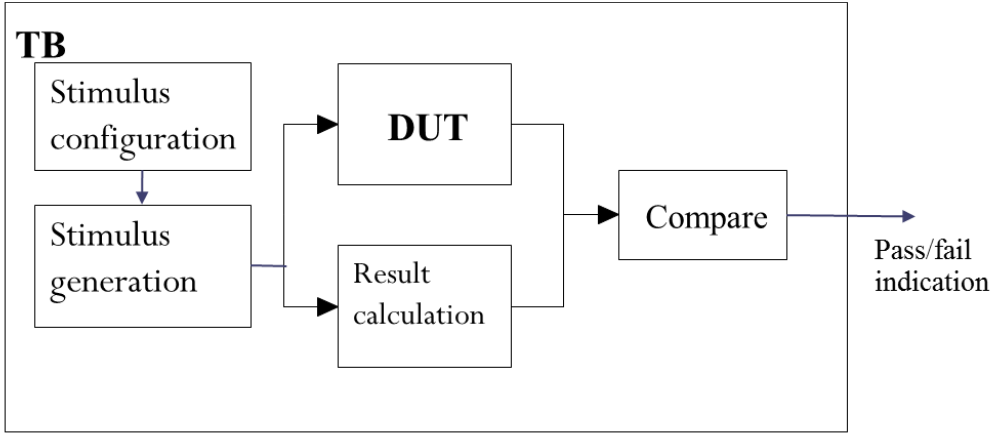
- Often, a system is first modeled with software and then parts are hardware accelerated
  - Software implementation serves as a golden reference
- E.g., video encoder implemented with C, motion estimation accelerated
- Tends to be quite slow



Feedback for the video encoder

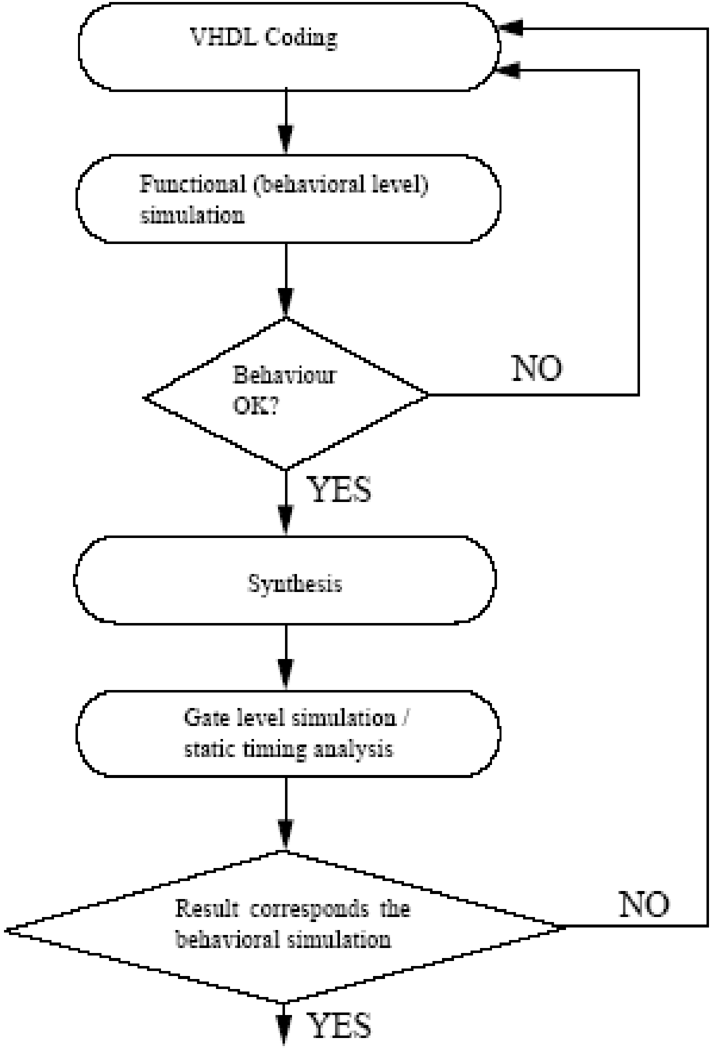
Autonomous Test Bench

- 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

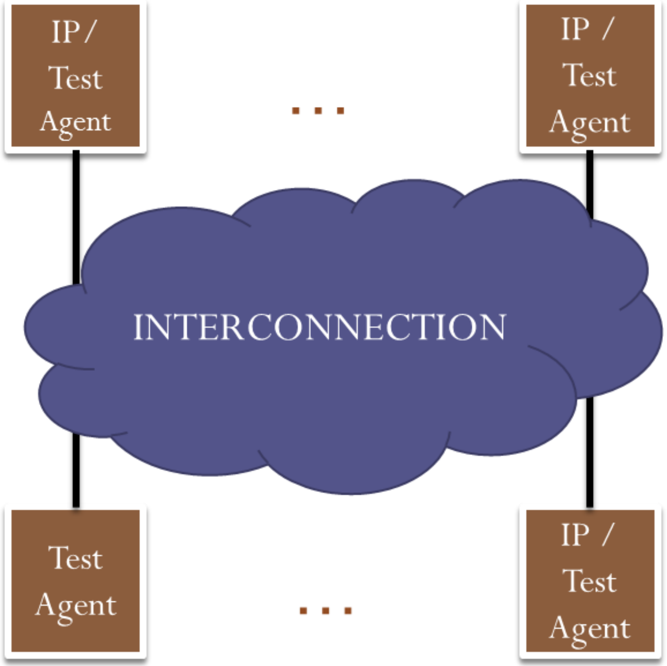
Design Flow


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

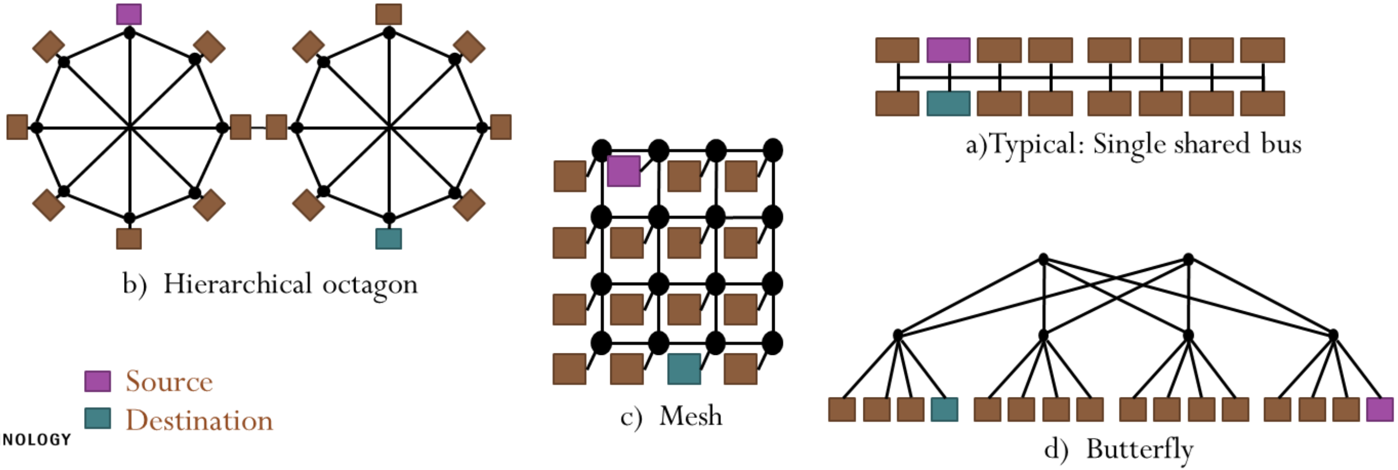
The Test Scenario

- 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)



a)Typical: Single shared bus

b) Hierarchical octagon

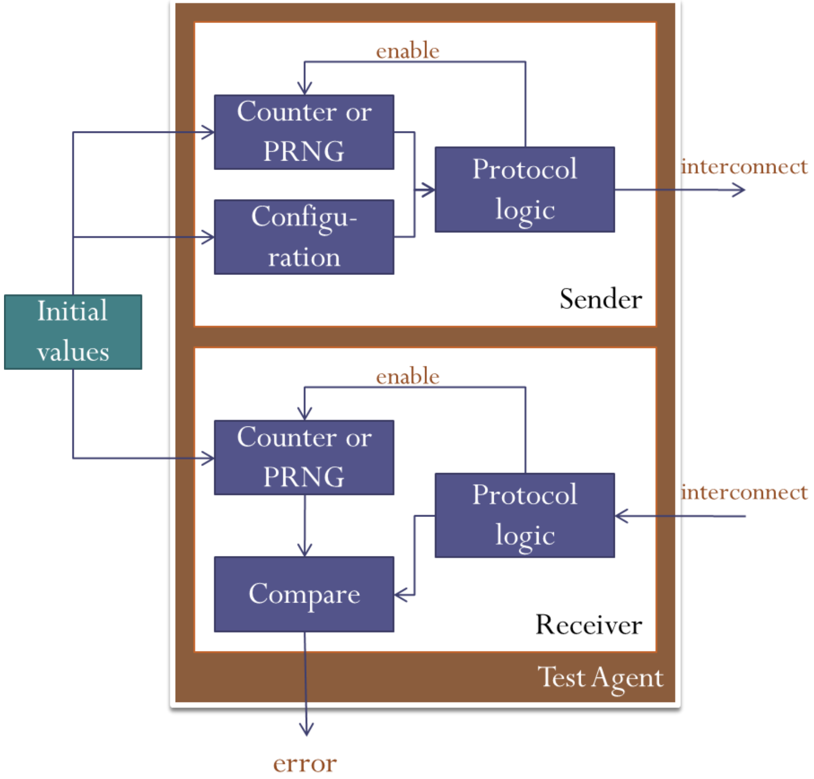
c) Mesh

d) Butterfly

Example: Verifying an Interconnection

- Tested interconnections delivers data from a single source to destination
  - Same principle, same IP interface, slightly different addressing
- Note that only the transferred data integrity is important, not what it represents – Running numbers are great!
- The test bench should provide few assertions (features a-d in previous 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 Test Bench



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

  - (PRNG = pseudo-random number generator)

Autonomous And Complex Test Benches

- Always a preferred choice – Well designed, reusable test bench 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 test benches 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
  - Syncronous test bench also works as the real environment would
  - More on the delta delay on next lecture about simulators

Example VHDL

- 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 Test Benches

- Synchronous, synthesizable test benches are a good practice if, e.g., the design includes clock domain crossings
- Can be synthesized to an FPGA and test 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.,

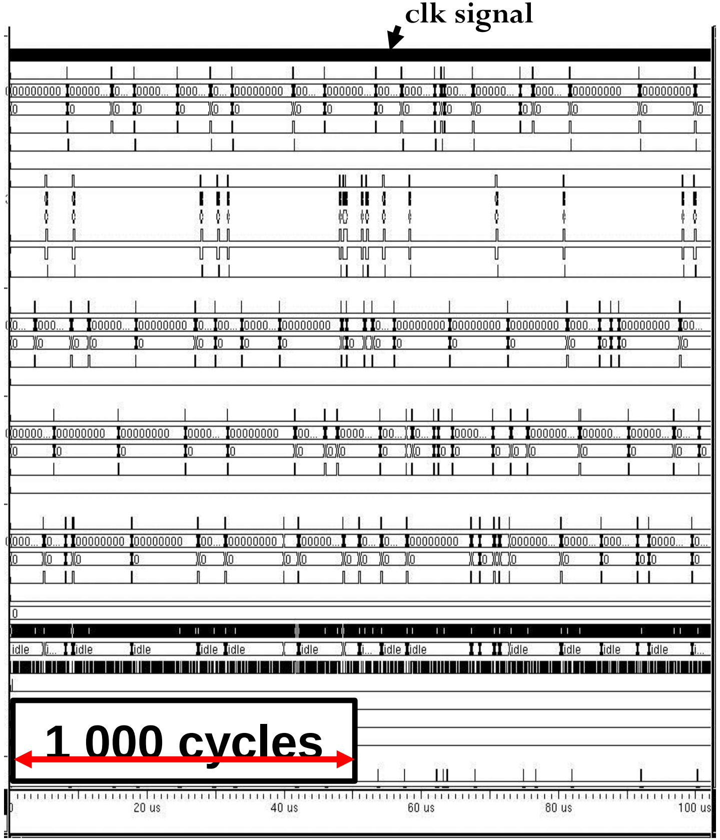
    `Assert_gen(cond, level, error_sig);`
  - In simulation, regular assertion, in synthesis, assigns to `error_sig`

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…



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!
      - ”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
- Test infrastructure is often of similar size and sometimes larger than the actual design
  - TB must be easy to run and analyze the results
  - TB may have bugs
  - TB may be reused and modified. That must be easy.

SELF-STUDY

Correct Bugs Early

[B.W. Boehm, Software Engineering,
IEEE Trans. Computers, 1976]

- Earlier the bugs fixed, the cheaper
  - Shorter design time
  - Smaller personel cost
  - Bigger market share
- Worst case: already sold devices must be returned to manufacturer
- Similarly, a bug in specification affects all other phases
- May require changes in many places

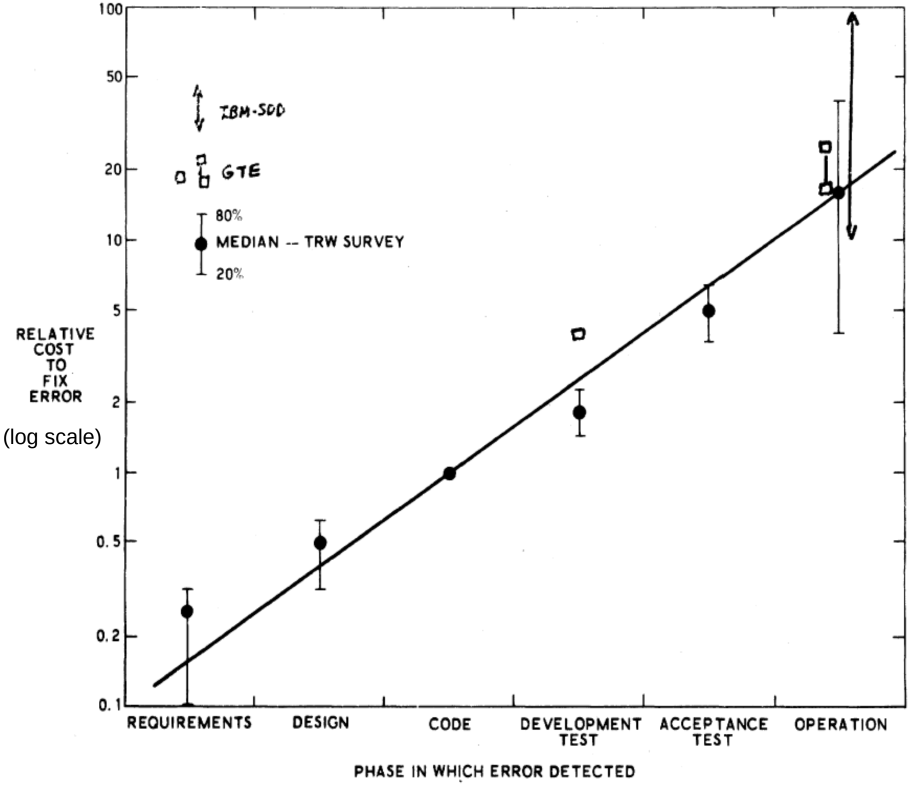
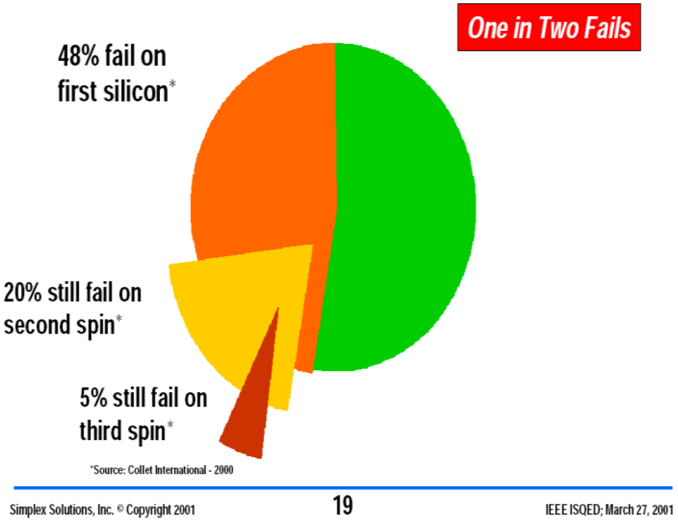


Fig. 3. Software validation: the price of procrastination.

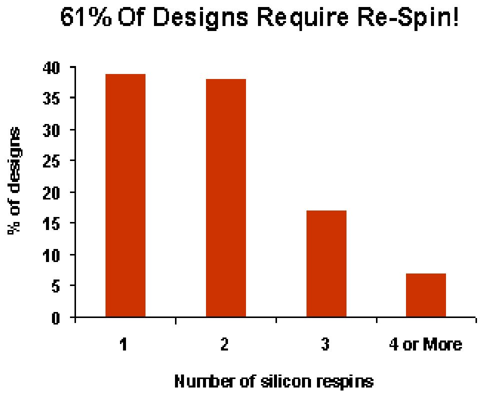
SoC Failures Cause Production Delays

- Incresed NRE (new mask, chip redesing)
- Lost revenue in the market (due to delay)



*Source: Collet International - 2000

Simplex Solutions, Inc. © Copyright 2001 19 IEEE ISQED; March 27, 2001

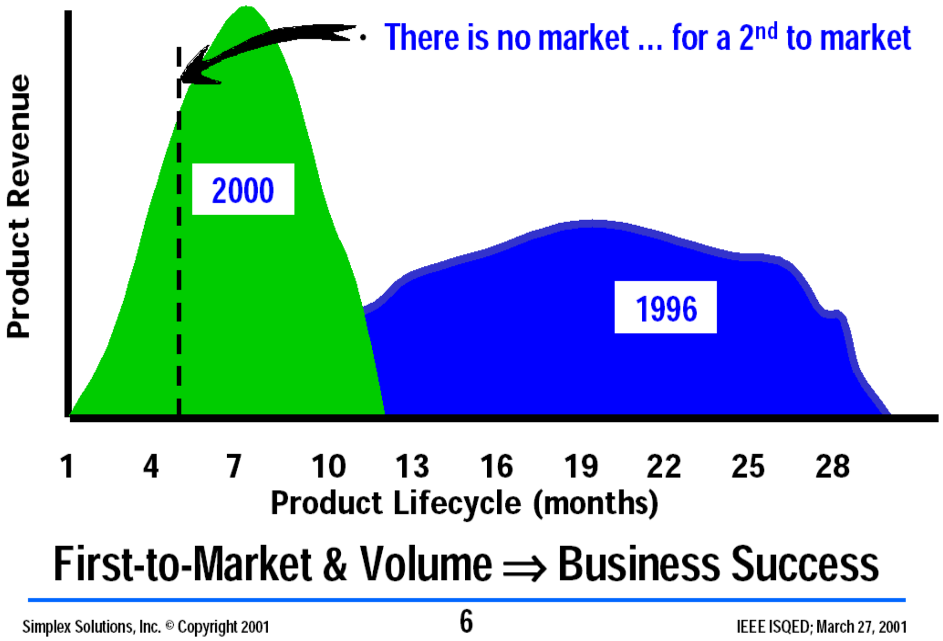


Source: Collett International Research Inc.

Left: [J. Costello, Delivering Quality Delivers Profits, IEEE ISQED, March 27, 2001]

Right: [P. Woo, Structured ASICs - A Risk Management Tool, Design&Reuse, Sep. 2005, [online] Available: http://www.design-reuse.com/articles/11367/structured-asics-a-risk-management-tool.html]

Time-To-Market (TTM)



First-to-Market & Volume ⇒ Business Success

Simplex Solutions, Inc. © Copyright 2001 6 IEEE ISQED; March 27, 2001

[J. Costello, Delivering Quality Delivers Profits
IEEE ISQED, March 27, 2001]

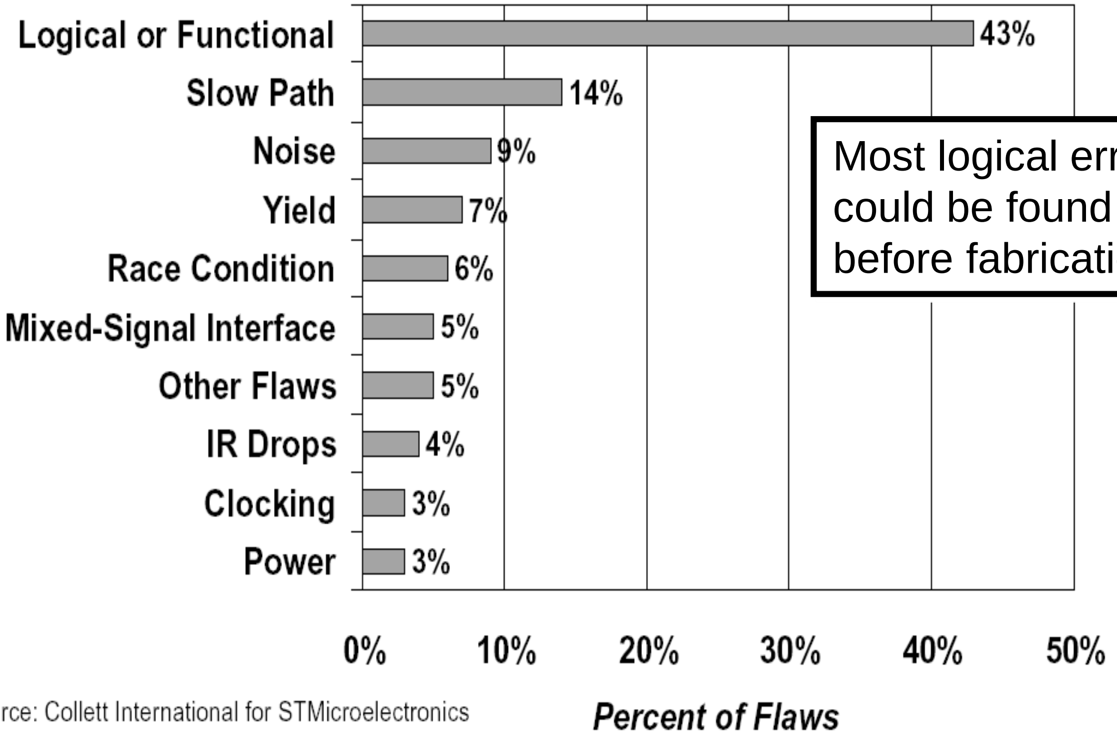
Table 1. Time-to-market matters

Time-To-Market	Potential Sales Achieved
First-To-Market	100%
3 Months Late	73%
6 Months Late	53%
9 Months Late	32%
12 Months Late	9%

[E. Clarke, FPGAs and Structured ASICs: Low-Risk SoC for the Masses, Design & Reuse, http://www.design-reuse.com/articles/13080/fpgas-and-structured-asics-low-risk-soc-for-the-masses.html]

TAMPERE UNIVERSITY OF TECHNOLOGY

Source of Failures

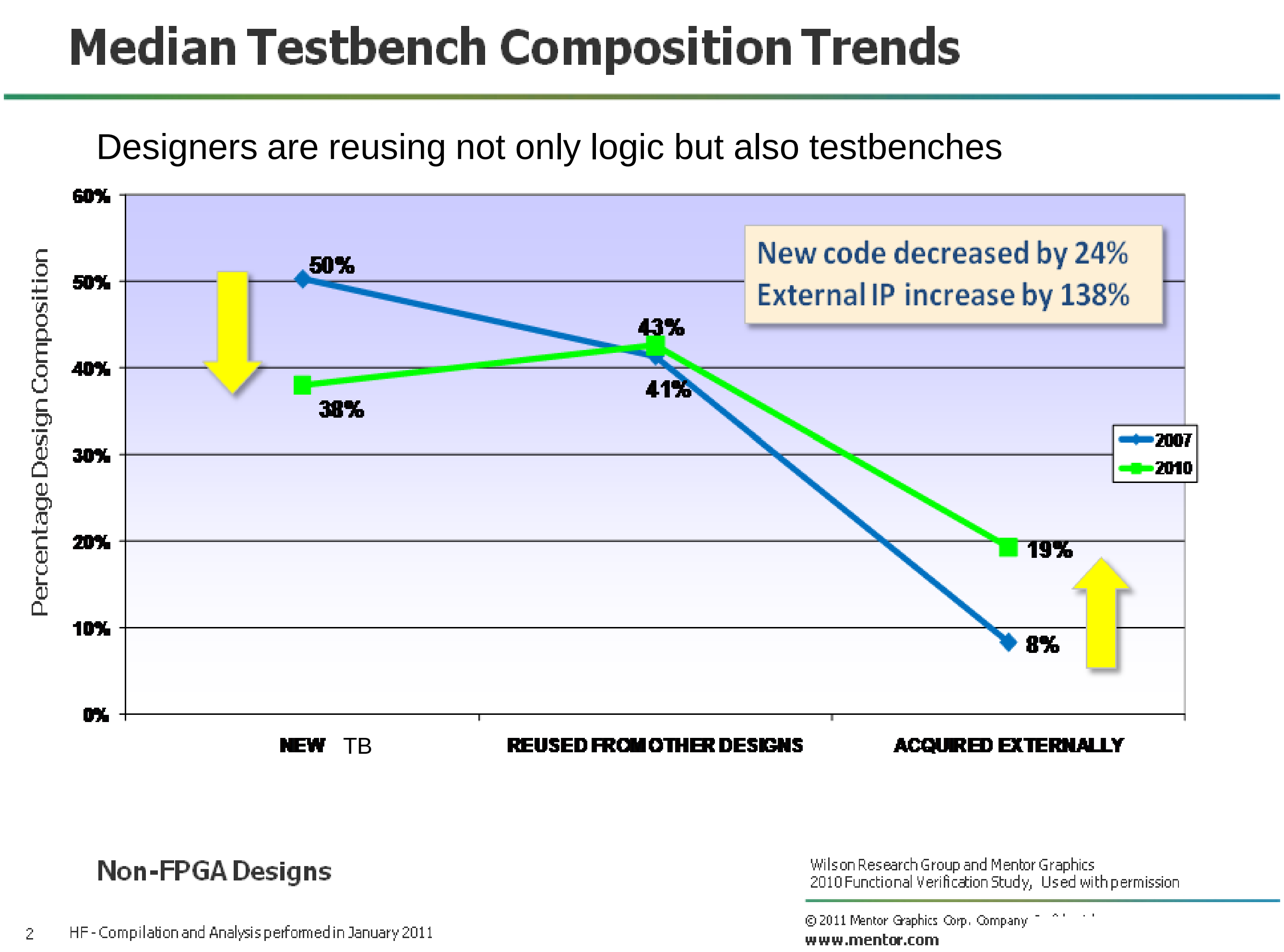


Most logical errors could be found before fabrication

Source: Collett International for STMicroelectronics

[P. Magarshack, SoC at the heart of conflicting, Réunion du comité de pilotage (20/02/2002), trendshttp://www.comelec.enst.fr/rtp_soc/reunion_20020220/ST_nb.pdf]

[http://blogs.mentor.com/verificationhorizons/blog/2011/04/01/part-3-the-2010-wilson-research-group-functional-verification-study/slide21-2/]



Median Testbench Composition Trends

Designers are reusing not only logic but also testbenches

New code decreased by 24%
External IP increase by 138%

Non-FPGA Designs

Wilson Research Group and Mentor Graphics
2010 Functional Verification Study, Used with permission

2 HF - Compilation and Analysis performed in January 2011

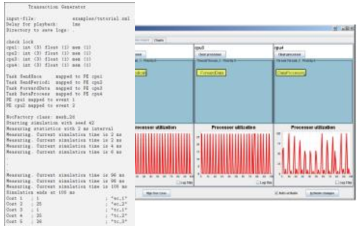
www.mentor.com

Verification Methods (1)

1. Reviews come in 2 flavors
   1. Specification reviews are especially useful
      - Remove ambiguities
      - Define how certain aspects of specification are recognized and analyzed in final product
      - Be sure to capture customer’s wishes correctly
   2. Code review
      - Designer explains the code to others
      - Surprisingly good results even if others do not fully understand
      - Good for creating clear and easy-to-understand code
      - Limited to small code sizes
      - Define and adopt coding guidelines
        - Automated checkers (lint) tools available
   - ”I’ve made many many product mistakes over the years. I should at least help make sure we make new mistakes this time around”
      - Eric Hahn on code reviews

Verification Methods (2)

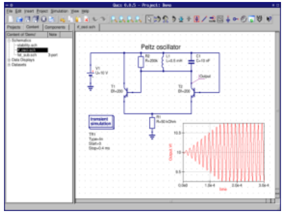
2. Simulation-based
   - Behavior is simulated in simulator program
   - Relies on test data
     - Test bench creation takes time
   - Cannot prove correctness
   - Slow, 100 Hz - 100kHz
   - Many levels of abstraction (algorithm vs. RTL vs. gate-level)
   - Availability of models might be a problem
   - Most widely used method
   - System simulation in **lecture 5**



```
        Transaction Generator

input-file:            examples/tutorial.xml
Delay for playback:    1ms
Directory to save logs: .

check lock
cpu1: int (3) float (1) mem (1)
cpu2: int (3) float (1) mem (1)
cpu3: int (3) float (1) mem (1)
cpu4: int (3) float (1) mem (1)

Task SendRace     mapped to PE cpu1
Task SendPeriodi  mapped to PE cpu2
Task ForwardData  mapped to PE cpu3
Task DataProcess  mapped to PE cpu4
PE cpu1 mapped to event 1
PE cpu2 mapped to event 2

NocFactory class: mesh_2d
Starting simulation with seed 42
Measuring statistics with 2 ms interval
Measuring. Current simulation time is 2 ms
Measuring. Current simulation time is 2 ms
Measuring. Current simulation time is 4 ms
Measuring. Current simulation time is 6 ms
.
.
.
Measuring. Current simulation time is 96 ms
Measuring. Current simulation time is 98 ms
Measuring. Current simulation time is 100 ms
Simulation ends at 100 ms
Cost 1   : 1                    : "ec_1"
Cost 2   : 25                   : "ec_2"
Cost 3   : 1                    : "tc_1"
Cost 4   : 25                   : "tc_2"
Cost 5   : 26                   : "tc_3"
```

Example 1: SystemC TLM + visualization

Example 2: RTL



Example 3:
Transistor-level

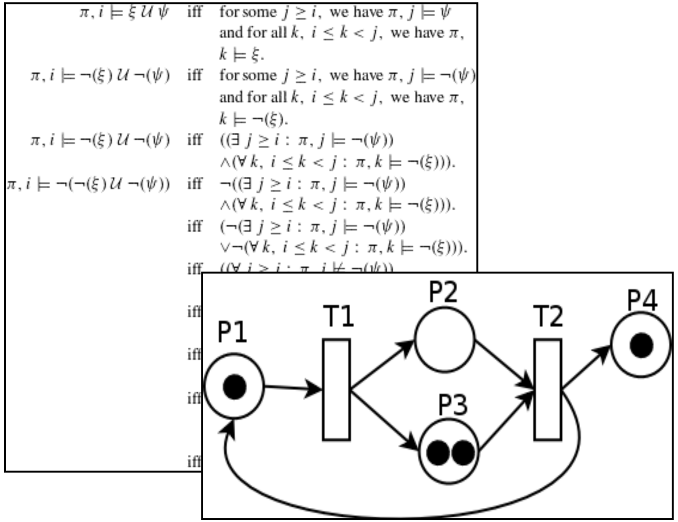
Verification Methods (3)

3. HW emulation
   - (Part of the) system is executed on programmable HW (FPGA)
     - ”FPGA prototype”, no mask costs as in ASIC proto
   - Nearly real-time execution (~1 MHz - 100 MHz)
     - But no regard of real logic delays!
   - Can connect to real external HW, such as radio
   - Rough GUI testing possible
   - Setup time may be long, e.g., few hours
     - Needs synthesis and place-and-route
   - Traditionally quite expensive systems
   - Reduced visibility compared to simulation
   - Relies on test data, cannot prove correctness

Example: BEE4 system contains 4 Xilinx Virtex-6 LX550 FPGAs (20 Million system gates per FPGA)

Verification Methods (4)

4. Formal methods
   - Correctness proven mathematically
   - Does not require test data
   1. Equivalence checkers
      - Check that two versions (e.g., RTL vs. gate-level) are functionally identical
      - Supported by many synthesis tools
   2. Model checking
      - Compare behaviour with formal specification
        - Proves that something good eventually happens (e.g., every request receives acknowledgement)
        - Proves that something bad will never happen (e.g., no deadlock)
      - Practically nobody writes such specifications currently…
   3. Semi-formal is combination of formal and simulation
      - **Assertions**



$$
\begin{array}{rll}
\pi, i \models \xi\,\mathcal{U}\,\psi & \text{iff} & \text{for some } j \geq i,\ \text{we have } \pi, j \models \psi \\
 & & \text{and for all } k,\ i \leq k < j,\ \text{we have } \pi, \\
 & & k \models \xi. \\
\pi, i \models \neg(\xi)\,\mathcal{U}\,\neg(\psi) & \text{iff} & \text{for some } j \geq i,\ \text{we have } \pi, j \models \neg(\psi) \\
 & & \text{and for all } k,\ i \leq k < j,\ \text{we have } \pi, \\
 & & k \models \neg(\xi). \\
\pi, i \models \neg(\xi)\,\mathcal{U}\,\neg(\psi) & \text{iff} & ((\exists\, j \geq i : \pi, j \models \neg(\psi)) \\
 & & \wedge(\forall\, k,\ i \leq k < j : \pi, k \models \neg(\xi))). \\
\pi, i \models \neg(\neg(\xi)\,\mathcal{U}\,\neg(\psi)) & \text{iff} & \neg((\exists\, j \geq i : \pi, j \models \neg(\psi)) \\
 & & \wedge(\forall\, k,\ i \leq k < j : \pi, k \models \neg(\xi))). \\
 & \text{iff} & (\neg(\exists\, j \geq i : \pi, j \models \neg(\psi)) \\
 & & \vee\neg(\forall\, k,\ i \leq k < j : \pi, k \models \neg(\xi))). \\
 & \text{iff} & ((\forall\, j \geq i : \pi, j \not\models \neg(\psi)) \\
 & \text{iff} & \\
 & \text{iff} & \\
 & \text{iff} & \\
 & \text{iff} &
\end{array}
$$

Levels of Verfication

- Level 0: Designer / macro, lowest level
  - Verification done by the designer (one who wrote the VHDL)
  - **Ensures that the design will load into simulator and that basic operation work**
  - Often many changes in specification expected this level
  - Small block size, perhaps just a single HDL file, suitable also for formal verification
- Level 1: Unit /core
  - **Combines few low-level blocks together,** DMA, ALU…
  - More stable interfaces and functions compared to level 0
    - Test suite remains mostly unchanged
  - Reusable component (a *core*) necessitates more thorough verification
    - Pro: Once verified, work always
    - Con: Can be used in arbitrary environment, hard to verify all corner cases

Levels of Verification (2)

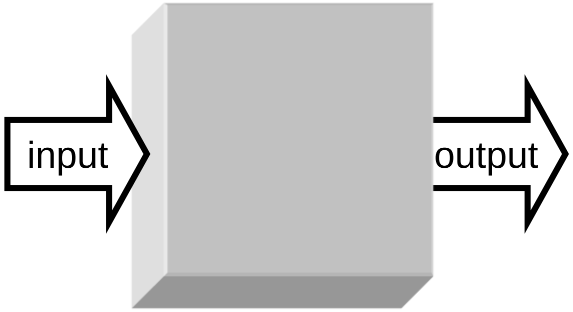
- Level 1: Unit / core continued
  - Most important level for functional verification
  - Q: How to gain customer's confidence when selling core?
  - A: Well-defined verification process, regression suite, proper documentation, coverage reports, good reputation based on previous cores…
- Level 2+: Chip, Board, System
  - Multiple units, stable interfaces
    - Possibly glue logic
  - Some functions cannot be verified a unit level
    - For example, reset and start-up sequence
  - ***Interaction* rather than particular functions are important at system level**

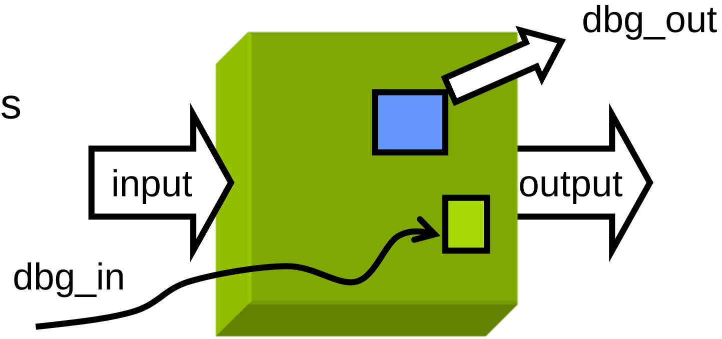
Choose the Lowest Possible Level

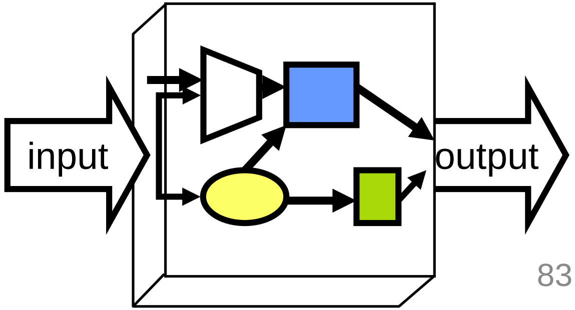
- Always choose the lowest level that completely contains the targeted function
  - Smallest state space, fast simulation
  - 79% of bugs were found at BLOCK LEVEL [S. Switzer, Using Embedded Checkers to Solve Verification Challenges, Designcon, 2000]
- Each verifiable piece should have its own specification document
- Every VHDL entity must have its own test bench, at least the simple macro-level TB
- New and/or complex functions need extra focus
- Bugs seldom live alone
- *Controllability* and *observability* define the correct level
- **The lower the level, the more control/visibility**

Visibility of DUT

- Black box
  - Contents invisible
  - Access only through primary inputs and outputs
  - E.g., proto-chip, SW binary
- Gray box
  - Some parts visible, perhaps touchable
  - E.g., proto-chip with test IO, some FPGAs
- White box, Glass box
  - All parts fully visible and touchable
  - E.g., RTL





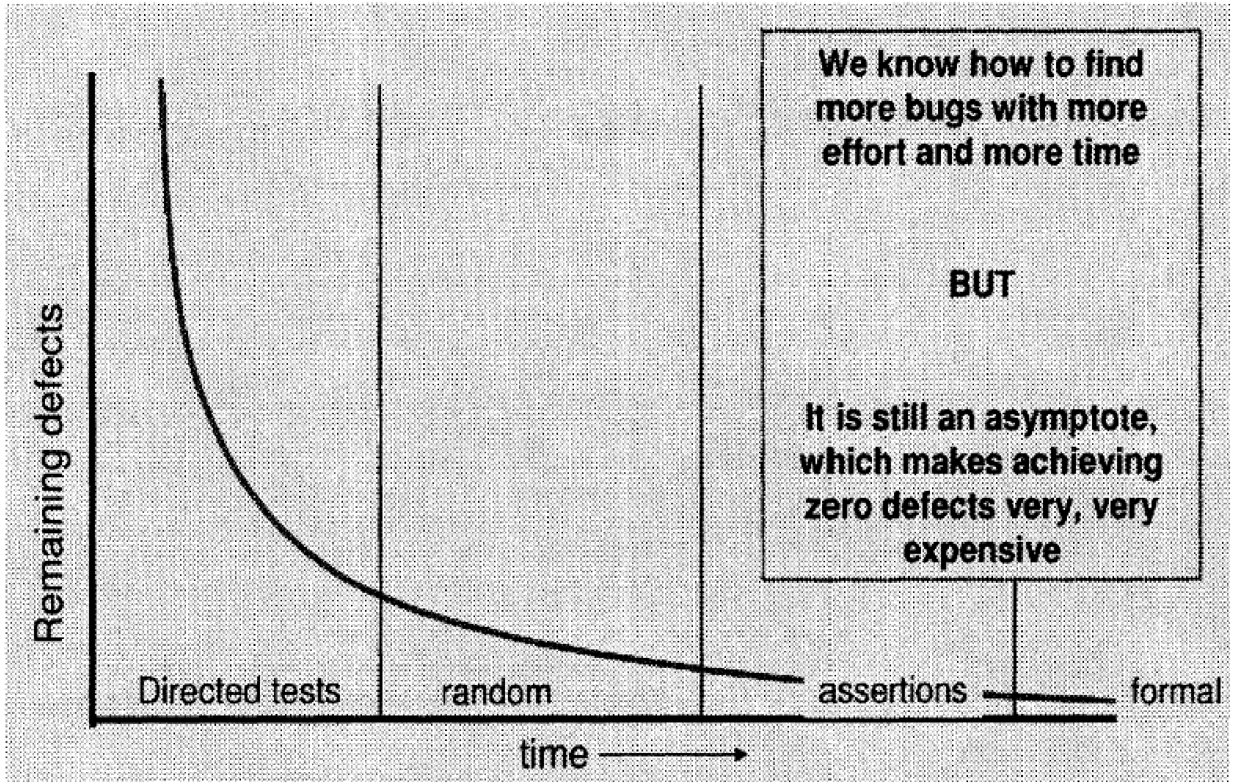


Repeating Tests

- Same error must be repeated to see if fix works
  - – Same test data, same timing
  - ➢ **Automated test generation**
- Must ensure that ”fix” does not break any other part of system
  - ➢ **Automated checking**
  - Manual checking suitable only for TB creation
- Preferably same TB during the design refinement
  - E.g., RTL and gate-level use same TB
- Keep all the test cases that have failed
  - Already fixed errors sometimes reappear later
  - Partition test cases into smaller sets

Bug-Free Behaviour Not Guaranteed

- [M. Keating, Toward Zero-Defect Design: Managing Functional Complexity in the Era of Multi-Million-Gate Design Architecture, DCAS '05]



Random Test Input

- Good for finding corner cases
- Easy to produce, allow large test vector sets
  - Luckily, **pseudo-random** number generators produce same series if seed is same
  - Running numbers are sometimes enough!
  - Can generate random input to file ➔ reproducible (but space-hungry)
- Randomness makes it harder to track **error source**
  - Output with running numbers: 1,2,3,4,**888**,6,7...
  - Output with random data: 701,123,-987,2,**3**,4,5,..

- Should not be used without test cases with ’known values’

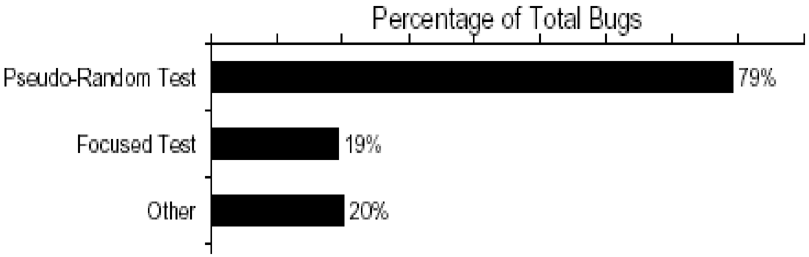


Figure 4. Effectiveness of Test Category

[S. Taylor, DAC 1998]

Tracking the Error Source, Assertions

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

Visibility on FPGA: Logic Analyzer (1)

- Provide easily accessible test points to the PCB
  - Route interesting signals to FPGA output/PCB test point
  - Clock, reset, state register, write enable, error flag
- Still only few (~2%) of signals on PCB are accessible to external logic analyzer
  - Number of analyzer inputs also restricted
- Integrated Logic Analyzer (ILA) used
  - Synthesized into FPGAs
    - Takes few logic resources just like communication networks
  - Monitors selected signals

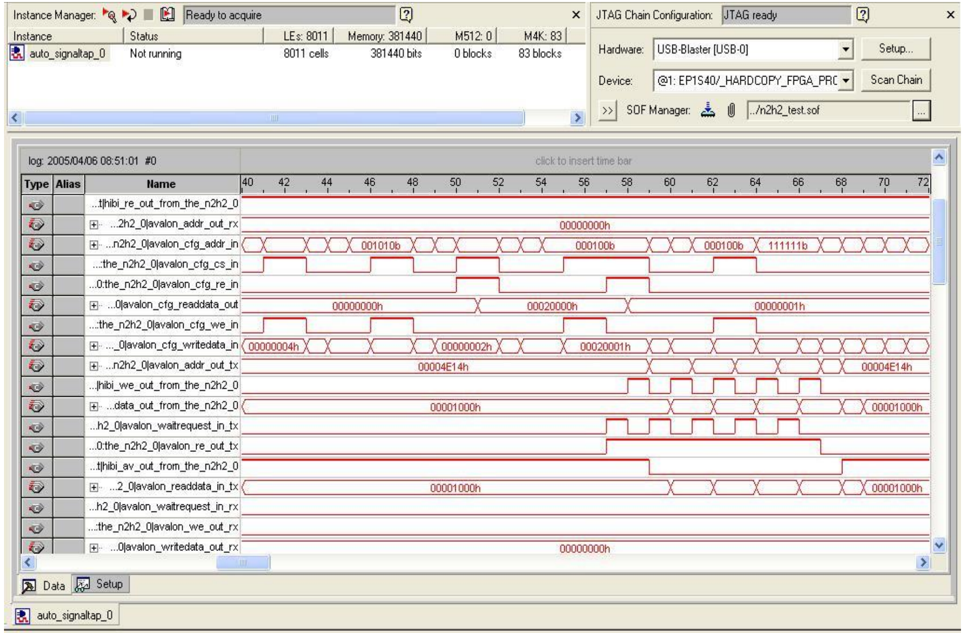
  a) Transmitting signal values to workstation via JTAG is slow (< 56Kb/s)

  b) Signal values stored in memory (1MB, 133MHz, 32b) and read after emulation run

  - Waveform displayed on workstation (just like in RTL simulation)
  - Not emulator-specific – can be used in any FPGA

Visibility on FPGA: Logic Analyzer (2)

- Only short (e.g., < 2048 cycles) traces collected if only on-chip memory utilized
- ➢ Hard to define correct trigger condition



Instance Manager: Ready to acquire

Instance	Status	LEs: 8011	Memory: 381440	M512: 0	M4K: 83
auto_signaltap_0	Not running	8011 cells	381440 bits	0 blocks	83 blocks

JTAG Chain Configuration: JTAG ready

Hardware: USB-Blaster [USB-0]

Device: @1: EP1S40/_HARDCOPY_FPGA_PRC

SOF Manager: ../n2h2_test.sof

log: 2005/04/06 08:51:01 #0

Type	Alias	Name
		...t|hibi_re_out_from_the_n2h2_0
		...2h2_0|avalon_addr_out_rx
		...n2h2_0|avalon_cfg_addr_in
		...:the_n2h2_0|avalon_cfg_cs_in
		...0:the_n2h2_0|avalon_cfg_re_in
		...0|avalon_cfg_readdata_out
		...:the_n2h2_0|avalon_cfg_we_in
		..._0|avalon_cfg_writedata_in
		...n2h2_0|avalon_addr_out_tx
		...|hibi_we_out_from_the_n2h2_0
		...data_out_from_the_n2h2_0
		...h2_0|avalon_waitrequest_in_tx
		...0:the_n2h2_0|avalon_re_out_tx
		...t|hibi_av_out_from_the_n2h2_0
		...2_0|avalon_readdata_in_tx
		...h2_0|avalon_waitrequest_in_rx
		...:the_n2h2_0|avalon_we_out_rx
		...0|avalon_writedata_out_rx

Tracking Error Source (1)

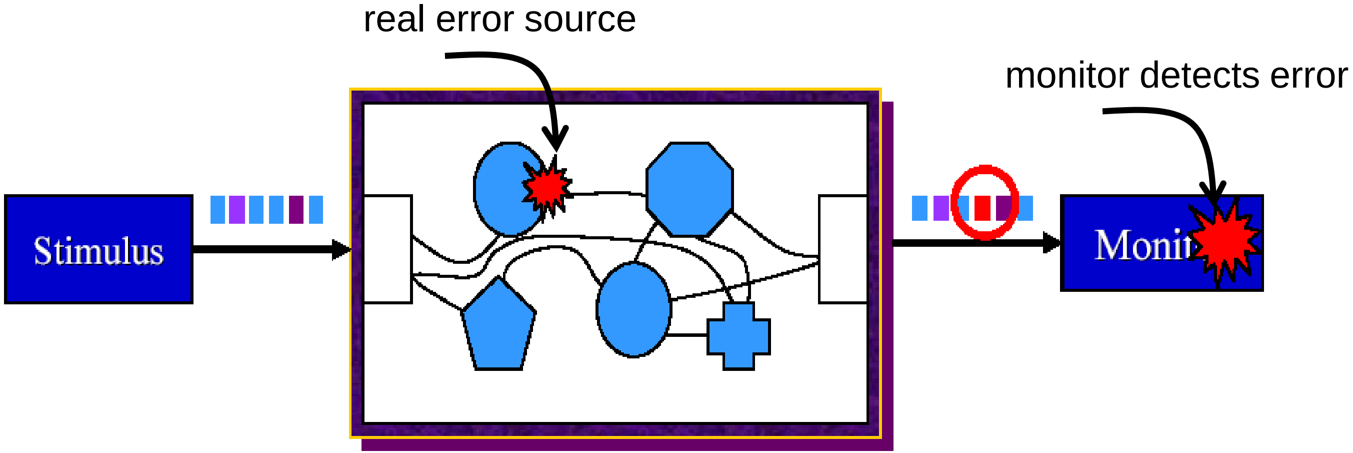
a) Pass / no pass
  - When errors cannot be corrected, source is irrelevant
  - E.g., manufacturing test on chips (black box)
    - Faulty chips are thrown away

b) Usually the errors should be corrected
  - Locating error is necessary
    - SW or HW
    - Which component
    - Which internal state
    - Line of code
    - Which test case (input sequence)

- Locating errors is easier if
  - smaller the system being verified
  - fewer changes have been made to functioning system

Tracking Error Source (2)

- In large design,
  - error is hard to repeat in controlled way
  - error at output is hard to find
  - error source **extremely** difficult to find
- **Common case: spend 1 month for finding the bug, fix it in 5 minutes**


real error source

monitor detects error

[B. Bailey, Property Based Verification for SoC, Tampere SoC, 2003]

Tracking Error Source (3)

- Sad example of bus testing:
  - Transmit 50 times value `0x7` and check that 50 values are received. What if:

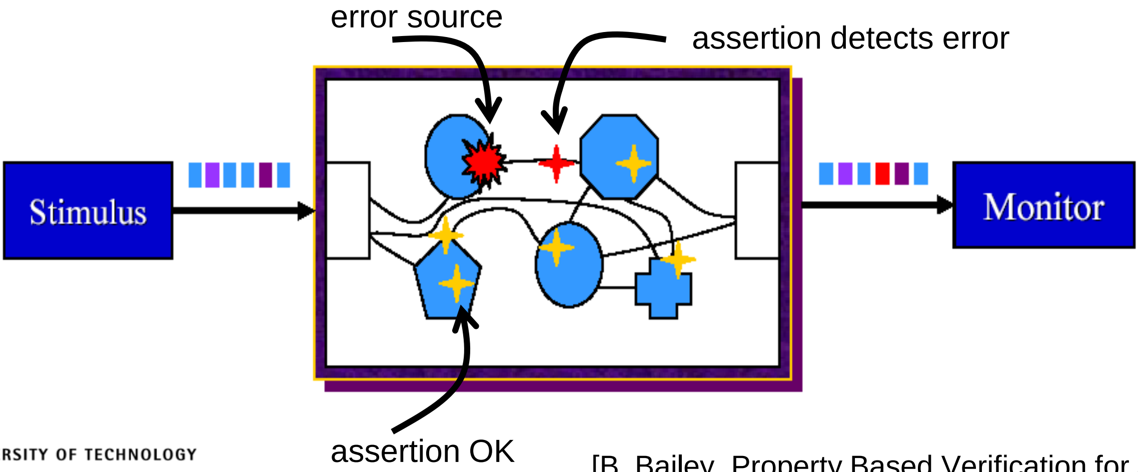
  a) <50 data received: which were missing?

  b) 50 received: is some data duplicated and some missing?

  c) >50 received: which one is duplicated?
- **Selection of test data may simplify finding error**
  - Consider transmitting sequential numbers (`1,2,3`...) over bus instead on constant value `7`
  - Locating duplicated/missing data is trivial
  - Of course, sequential numbers are not that useful with, e.g., arithmetic components
  - Using *unique values* in test input helps to track the error from wave format or trace
    - Differentiate sources: (`1,2,3`...); (`101,102,103`...); ...(`901,902,903`...)

Tracking Error Source (4)

- **Assertions** can be added to modules or interfaces
- Brings the point detection closer to the point of problem injection
  - Close in time (clock cycle)
  - Close in place (module, code line)
- Simplifies TB – no longer necessary to propagate all effects to outputs



[B. Bailey, Property Based Verification for SoC, Tampere SoC, 2003]

Assertions (1)

- Express the design intent
  - Best captured by the designer
  - ”Built-in implementation specification”
- Check that certain properties always hold
  - FSM won’t enter illegal state, one-hot encoding is always legal etc.
- Checked during simulation, not synthesizable
  - Synthesizable HDL assertion would be über-cool
- E.g., signals A and B must NEVER be ’1’ at the same time
  - VHDL: `assert (A and B = 0) report ”A and B simultaneously asserted” severity warning;`
  - ➢ VHDL simulator: `Warning: A and B simultaneously asserted, Time: 500 ns, component : /tb_system/tx_ctrl/`

Assertion (2): Classification

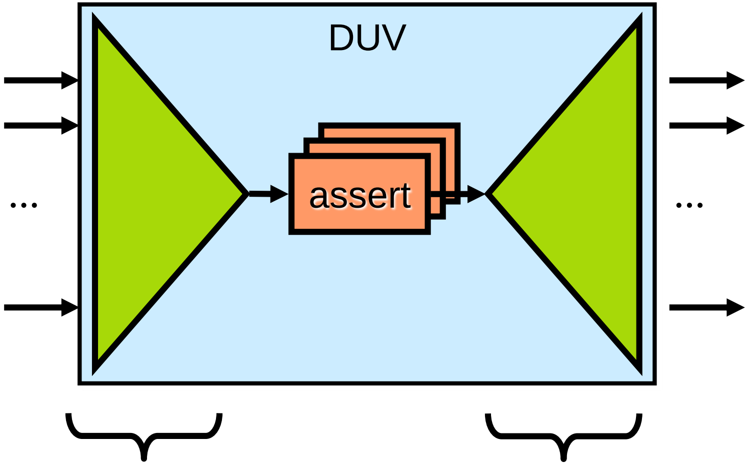
a) Event detection
   - Simplest form
   - Checks the absence of a specific event, which is a sign of failure
     - E.g., FIFO seems empty and full at the same time
   - Static – not related to any other event

b) Temporal event detection
   - Refer to sequence of events
   - VHDL assertions have restricted means for expressing timing (sequences)
     - E.g., check that after 2 cycles…
   - Possible to build specific logic that creates simple Boolean check for the assert
   - Addressed by formal methods, such as PSL/Sugar

Assertions (3)

c) Pre-defined event detection bulding blocks
   - Library for checking often occurring events
   - Data structures (stack, buffer, FIFO) or control structures (handshake)

- Figure: Defensive HDL design
- Special tools can process VHDL assertions and try to violate them



This logic ”cone” is checked by assertion(s). Errors will be be caught

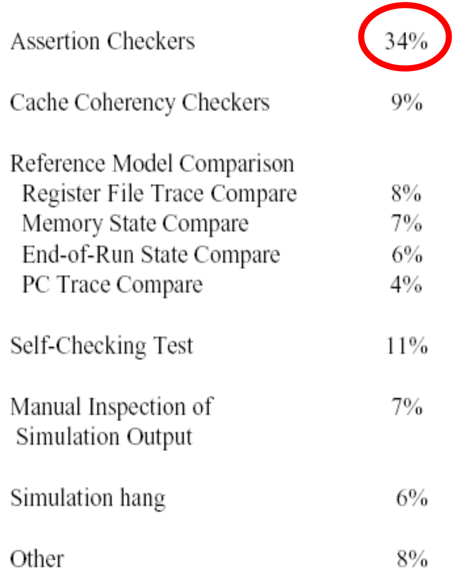
This logic is protected by asserion(s). Erroroneous values won’t pass the asertion(s).

Assertions (4)

- In C language:
- Degines in header file `assert.h`
- Common error outputting is in the form:
  - `Assertion failed:` *`expression`*`, file` *`filename`*`, line` *`line-number`*
- Example:

```
#include<assert.h>
void open_record(char *record_name)
{
    assert(record_name!=NULL);
/* Rest of code */
}
```

Assertions (5)



Assertion Checkers	34%
Cache Coherency Checkers	9%
Reference Model Comparison	
Register File Trace Compare	8%
Memory State Compare	7%
End-of-Run State Compare	6%
PC Trace Compare	4%
Self-Checking Test	11%
Manual Inspection of Simulation Output	7%
Simulation hang	6%
Other	8%

Figure 2: Effectiveness of Bug Detection Mechanisms

[M. Kantrowitz L.M. Noack, I'm Done Simulating; Now What? Verification Coverage Analysis and Correctness Checking of the DECchip 21164 Alpha microprocessor, DAC, June 1996, pp. 325-330. http://www.sigda.org/Archives/ProceedingArchives/Compendiums/papers/1996/dac96/pdffiles/23_5.pdf]

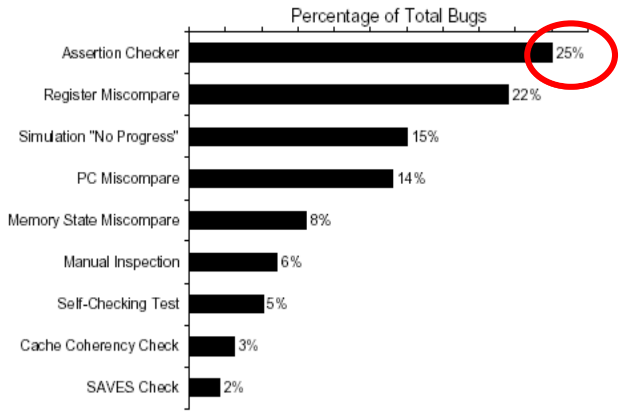


Figure 6. Bug Detection Mechanisms

[S. Taylor *et al.*, Functional Verification of a Multiple-issue, Out-of-Order, Superscalar Alpha Processor—The DEC Alpha 21264 Microprocessor, DAC 98, pp. 638-644. http://www.sigda.org/Archives/ProceedingArchives/Compendiums/papers/1998/dac98/pdffiles/39_1.pdf]

Assertions (6)

- Powerful in finding errors *in simulation*
- Assertions can be used as a base for generating test cases automatically
- Naturally, cannot be synthesized
  - Synthesizable HW monitors may be developed case-by-case
    - Error tracking
    - Performance measurement
    - Trigger the trace collection for off-line analysis
- **USE ASSERTIONS**