Functional verification

Hipólito Guzmán Miranda
Departamento de Ingeniería Electrónica
Universidad de Sevilla
hguzman@us.es

Acknowledgement to Ray Salemi

Teaching context

B02: Advanced Programmable Logic Systems

- Tema 1: FPGA architecture
- Tema 2: Advanced digital design methodologies
- Tema 3: Advanced VHDL
- Tema 4: Verification capabilities for digital circuits

Required prior knowledge:

- Basic VHDL
- Advanced VHDL

Learning objectives

- Understand the limitations of traditional testbenches
- Learn the most common verification metrics, such as code coverage and functional coverage
- Understand the concept of Transaction-Level Modeling
- Acquire the conceptual capabilities to build a structured testbench step by step

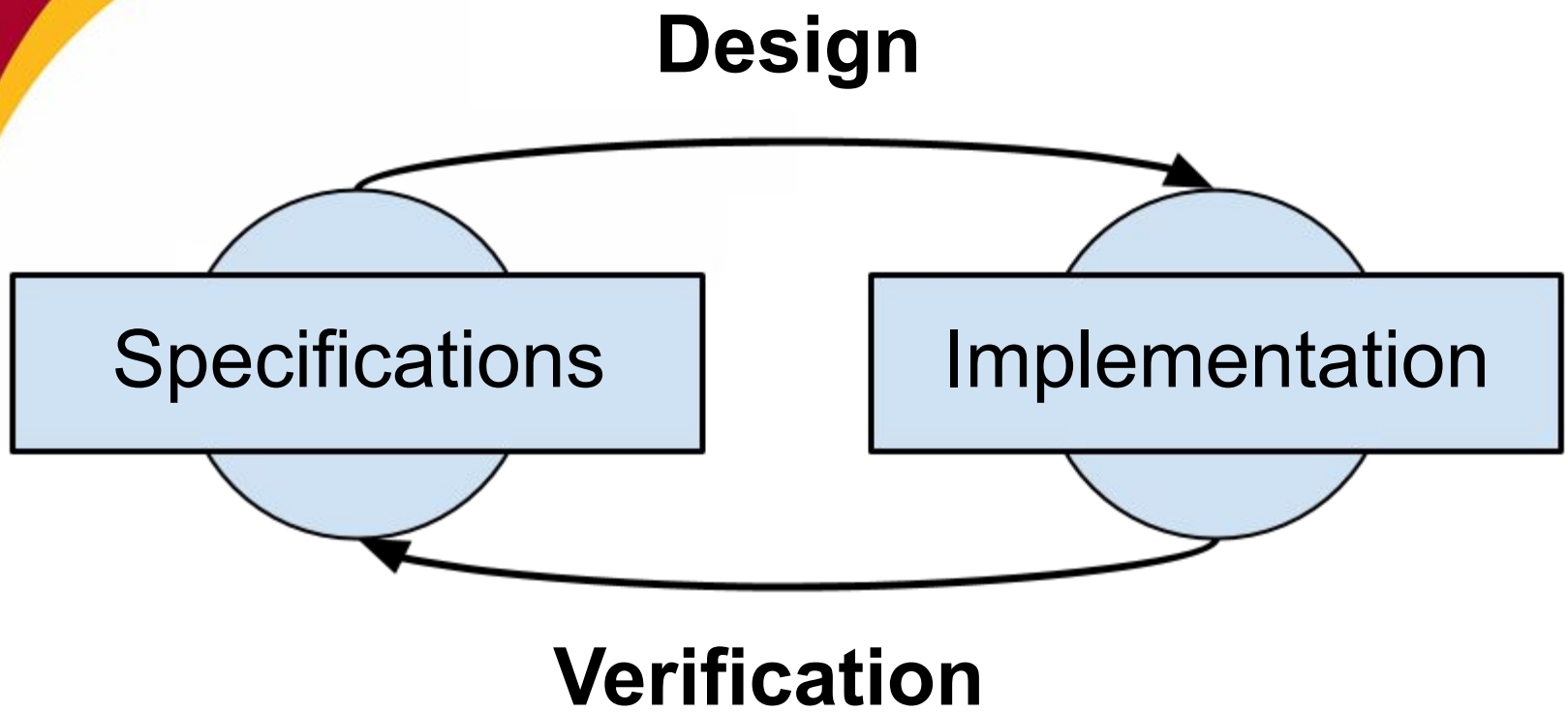
Contents

- What is verification?
- Why verify?
- 0.- Directed testing
- 1.- Code coverage
- 2.- Assertions
- 3.- Transaction-Level Modeling
- 4.- Self-checking testbenches
- 5.- Automatic stimuli
- 6.- Functional coverage
- Bibliography

Contents

- What is verification?
- Why verify?
- 0.- Directed testing
- 1.- Code coverage
- 2.- Assertions
- 3.- Transaction-Level Modeling
- 4.- Self-checking testbenches
- 5.- Automatic stimuli
- 6.- Functional coverage
- Bibliography

What is verification?



Ensure that the developed implementation is actually compliant with the specifications

Contents

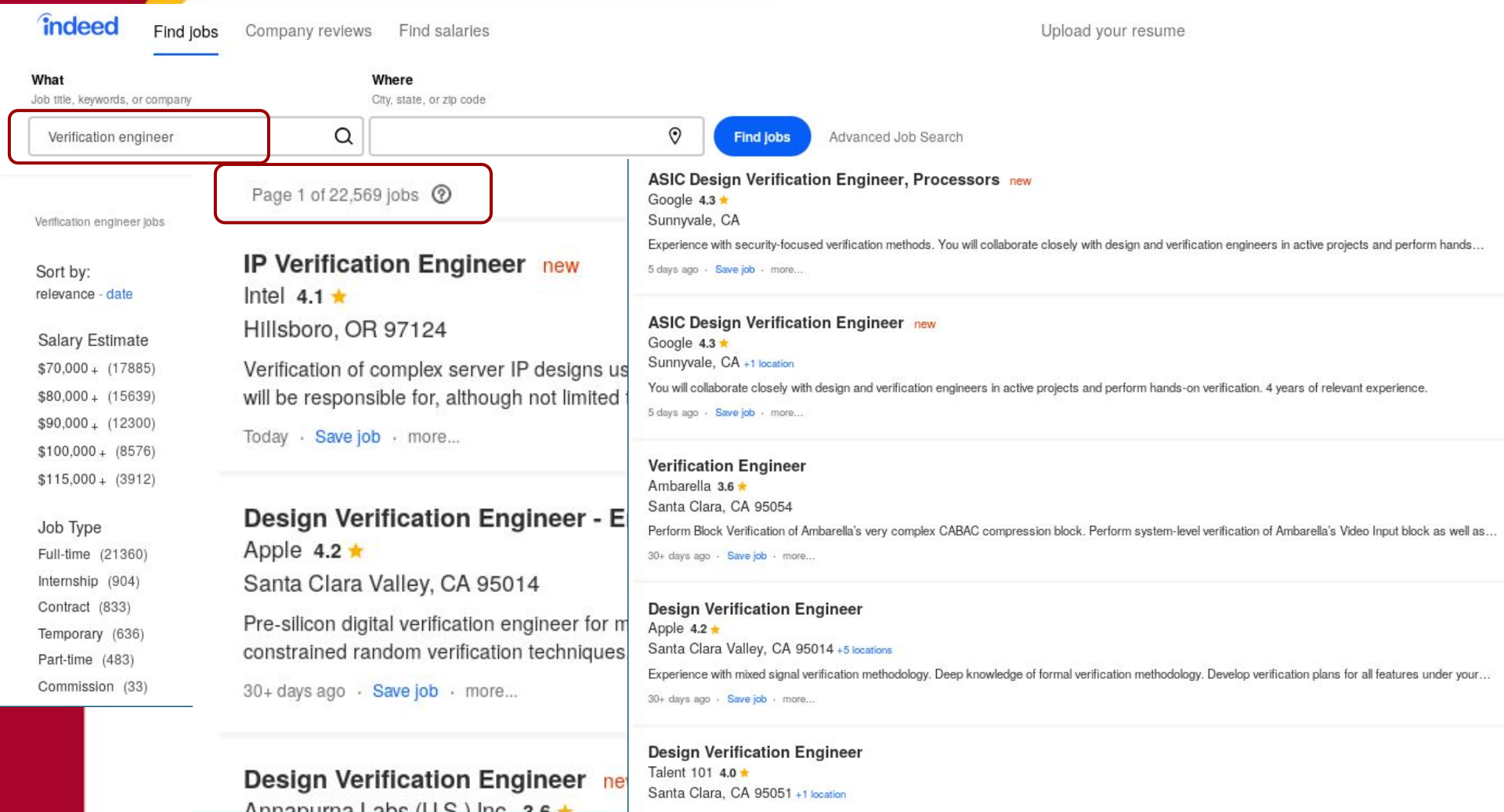
- What is verification?
- Why verify?
- 0.- Directed testing
- 1.- Code coverage
- 2.- Assertions
- 3.- Transaction-Level Modeling
- 4.- Self-checking testbenches
- 5.- Automatic stimuli
- 6.- Functional coverage
- Bibliography

Some good reasons…

- Verification gap
- Mental health
- Verification can be between 50% and 80% of total development time
- ASIC manufacturing costs
- Industries where bugs and errors must be avoided at all costs: space, aeronautics, biomedicine, nuclear …
- Difficulty in diagnosing and fixing faults on the FPGA prototype
- Actually finishing projects on time

Motivation

Why learn verification?



Contents

- What is verification?
- Why verify?
- 0.- Directed testing
- 1.- Code coverage
- 2.- Assertions
- 3.- Transaction-Level Modeling
- 4.- Self-checking testbenches
- 5.- Automatic stimuli
- 6.- Functional coverage
- Bibliography

‘Traditional’ testing

Typical testbench:

- Manually defined stimuli
- Always the same stimuli
- Verification by visually inspecting the waveform

This approach does not scale for complex tests

E.g., 200K stimuli and 1M clock cycles of waveforms to check

Directed vs random testing

- Tests can be of two types:
  - Directed
    - Predetermined stimuli
    - (We may have pre-calculated the expected output)
  - Random
    - Stimuli generated each time the simulation is run
    - (How do I know if the output is correct?)

When to stop?

- In both cases, it's hard to be sure we've tried everything!
- How do we know when we have finished verifying?

Contents

- What is verification?
- Why verify?
- 0.- Directed testing
- 1.- Code coverage
- 2.- Assertions
- 3.- Transaction-Level Modeling
- 4.- Self-checking testbenches
- 5.- Automatic stimuli
- 6.- Functional coverage
- Bibliography

An automated technique

- The simulator does this by itself, by using certain options when compiling the simulation executables.

- Supported by ModelSim/Questa, Aldec, Vivado XSim, GHDL (partially), etc.

Identify code that has not been tested

- Mnemonic:
  "**S**ome **B**eers **F**or **E**xtra **C**ourage"

- Statement
- Branch
- FSM
- Expression
- Condition

Statement Coverage

- Which statements were executed and which were not
- A statement is anything that ends with a semicolon.
- One statement (at most) per line makes calculating coverage easier for the simulator.

Questa Coverage Report - Mozilla Firefox

File Edit View History Bookmarks Tools Help

Questa Coverage Report

file:///home/hipolito/devel/edelweiss/build_vsim/coverage/pages/__frametop.h

Testplan | Design | DesUnits

- tb_bit2symb
- edelweiss_common
- vhdl_verification (no covera
- image_pkg
- txt_util
- tb_d_ff
- tb_fadapt
  - clkmanager_inst
  - datagen_inst
  - uut
  - datacompare_inst
  - throughputchecker_inst
- tb_fifo
- tb_pulse_shaping
- tb_qdelay
- tb_symb2chip
- tb_top_tx
- tb_upsampling
- tb_dem_filter
- tb_downsampling
- tb_tap

```
144       32          n_ppdu <= PHR (bit_count);
145       32          n_state <= count_cycles_header;
146                   if (bit_count = 7) then
147        4            n_state <= wait_for_data;
148        4            n_byte_count <= 0;
149        4            n_bit_count <= 0;
150                   end if;
151                 end if;
152               when wait_for_data =>
153                 if (remaining = 0) then
154        3          n_state <= idle;
155        3          n_count <= THROUGHPUT - 3; -- conserve throughput between frames
156                 elsif (count > 1) then
157  1275534          n_count <= count -1;
158                 elsif (not empty = '1') then
159     3080          rden <= '1';
160     3080          n_count <= THROUGHPUT -1;
161     3080          n_state <= data;
162                 end if;
163               when data =>
164     5048        n_ppdu <= fifo_ppdu;
165     5048        n_ppdu_valid <= '1';
166     5048        n_remaining <= remaining-1;
167     5048        n_state <= wait_for_data;
168               when others =>
169           0     n_state <= idle;
170           0     n_bit_count <= 0;
171           0     n_byte_count <= 0;
172           0     rden <= '0';
173           0     n_count <= THROUGHPUT -1;
174             end case;
175           end process;
176
177
178           sinc: process(clk, rst)
179             begin
180             if (rst = '1') then
181       22      state <= idle;
182       22      bit_count <= 0;
183       22      byte_count <= 0;
184       22      ppdu <= '0';
185       22      ppdu_valid <= '0';
```

Branch Coverage

- We might enter an if statement, but which `if/elsif/else` statement do we exit through? Which '`when X =>`' are we actually entering into?

- Evaluates whether the different branches of our code have been reached

Branch Coverage

case state is			83.33%
Branch	**Source**	**Hits**	**Status**
TRUE	when idle =>	13	Covered
TRUE	when count_cycles_header =>	79676	Covered
TRUE	when header =>	192	Covered
TRUE	when wait_for_data =>	1278618	Covered
TRUE	when data =>	5048	Covered
TRUE	when others =>	0	ZERO

if ((Looped = true) or (Head /= Tail)) then			50.00%
Branch	**Source**	**Hits**	**Status**
IF	if ((Looped = true) or (Head /= Tail)) then	385	Covered
ALL FALSE	if ((Looped = true) or (Head /= Tail)) then	0	ZERO

FSM Coverage

- When verifying state machines, we want to know if we have covered:
  - The states
  - The possible transitions between states¡
- For the ‘`when others =>`’ an exception is typically needed
  - This is called “code coverage exclusion”

state	83.33%

States / Transitions	Hits	Status
State: idle	390214	Covered
Trans: idle -> count_cycles_header	4	Covered
Trans: idle -> idle	390209	Covered
State: count_cycles_header	79676	Covered
Trans: count_cycles_header -> header	192	Covered
Trans: count_cycles_header -> idle	0	ZERO
Trans: count_cycles_header -> count_cycles_header	79484	Covered
State: header	192	Covered
Trans: header -> count_cycles_header	188	Covered
Trans: header -> wait_for_data	4	Covered
Trans: header -> idle	0	ZERO
Trans: header -> header	0	ZERO
State: wait_for_data	1673819	Covered
Trans: wait_for_data -> idle	3	Covered
Trans: wait_for_data -> data	3080	Covered
Trans: wait_for_data -> wait_for_data	1670735	Covered
State: data	3080	Covered
Trans: data -> wait_for_data	3080	Covered
Trans: data -> idle	0	ZERO
Trans: data -> data	0	ZERO

Expression Coverage

When we assign:

```
output <= a OR (b AND c);
```

If output = ‘1’...

- Is it because a = ‘1’ ?
- Is it because b = ’1’ and c = ‘1’ ?

If output = ‘0’...

- Is it because a, b = ‘0’?
- Is it because a, c = ‘0’?

We want to make sure that we have tested all cases

Condition Coverage

Similar to Expression Coverage, but in conditions instead of assignments:

```
if(a='1' OR (b='1' AND c='1')) then
```

- ¿a = ‘1’?
- ¿b=’1’ and c=’1’?

else

- ¿a = ‘0’ and b = ‘0’?
- ¿a = ‘0’ and c = ‘0’?

Condition Coverage

FEC : Focused Expression Coverage

FEC Condition: if (i_index = 30 AND q_index = 31) then			50.00%
Input Term	**Covered**	**Reason For No Coverage**	**Hint**
(i_index = 30)	**Yes**		
(q_index = 31)	**No**	'_0' not hit	Hit '_0'

Rows	FEC Target	Hits	Matching Input Patterns
Row 1	(i_index = 30)_0	2	{ 0- }
Row 2	(i_index = 30)_1	2	{ 11 }
Row 3	(q_index = 31)_0	0	{ 10 }
Row 4	(q_index = 31)_1	2	{ 11 }

Contents

- What is verification?
- Why verify?
- 0.- Directed testing
- 1.- Code coverage
- 2.- Assertions
- 3.- Transaction-Level Modeling
- 4.- Self-checking testbenches
- 5.- Automatic stimuli
- 6.- Functional coverage
- Bibliography

Notifies us if a condition does not hold

```
assert condition report string
severity severity_level;
```

4 severity levels:

- note
- warning
- error
- failure (stops simulation)

Are they synthesizable?

- The are not synthesizable, but they don't impede synthesis
- The synthesizer, in general, doesn't consider assertions
  - It can only consider those assertions whose condition is static (for example, to avoid synthesis with invalid GENERIC values), and makes the check in synthesis time
- They are only considered by the simulator

Assertion types

- Firewall assertions
  - To ensure your blocks are being used correctly
  - Typically added by the design engineer
- Protocol monitor
  - To ensure different blocks are communicating correctly (in compliance to a specific protocol)
  - Typically added by the verification engineer
  - A protocol monitor is more than assertions

(Both use the same VHDL `assert` statement)

Examples

- **En VHDL:**

```
assert (cont >= 0 and cont <= 7)
   report "cont overflow, should
never happen!" severity failure;
```

There are assertions in other languages such as PSL or SystemVerilog

```
assert DATA_LENGTH > 0
report "fadapt : DATA_LENGTH must be a positive
non-zero integer"
severity failure;

assert (NOT (ifull = '1' and wr_en='1' and
falling_edge(clk)))
report "fadapt : Trying to write in a full fifo: data
will be lost. Check throughput of blocks"
severity failure;

assert (NOT (empty = '1' and rd_en='1' and
falling_edge(clk)))
report "fadapt : Trying to read from an empty fifo:
invalid data will be processed"
severity failure;
```

If the condition is complex, better inside an `if`

- We can also use `report` without `assert`:

```
if (output /= expected) then
   report (“error in data”)
   severity error;
end if;
```

Contents

- What is verification?
- Why verify?
- 0.- Directed testing
- 1.- Code coverage
- 2.- Assertions
- 3.- Transaction-Level Modeling
- 4.- Self-checking testbenches
- 5.- Automatic stimuli
- 6.- Functional coverage
- Bibliography

Transaction-Level Modelling (TLM)

Is to elevate the abstraction level in verification, separating:

- Data that move through interfaces

of

- The pin movement of data and associated control signals

Transaction-Level Modelling (TLM)

For example, when we send data through a FIFO, we:

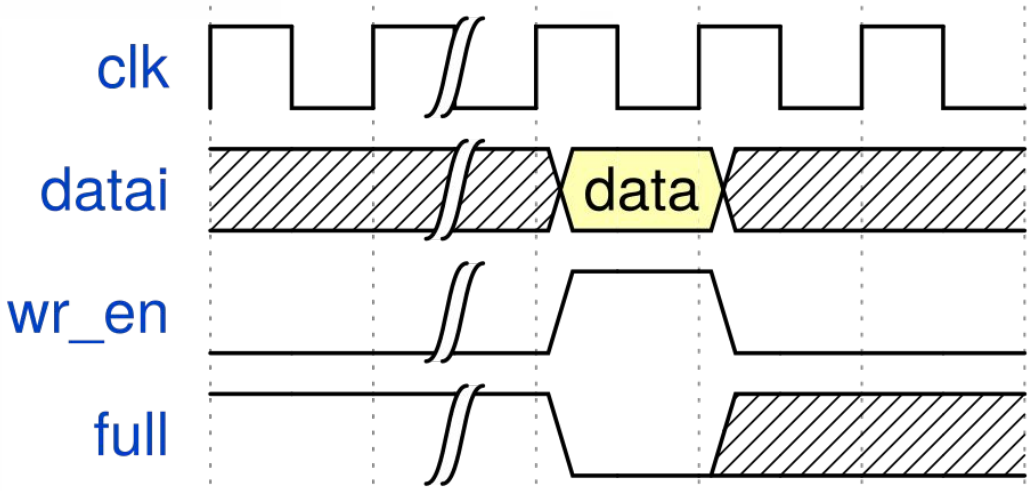
1. Wait until the FIFO is not full
2. Drive the data
3. Drive write_enable
4. Wait for a single clock cycle
5. Deactivate write_enable

We want to separate the sending of the data (fifo_write) from the pin movement (full, write_enable, datai)

Transaction-Level Modelling (TLM)

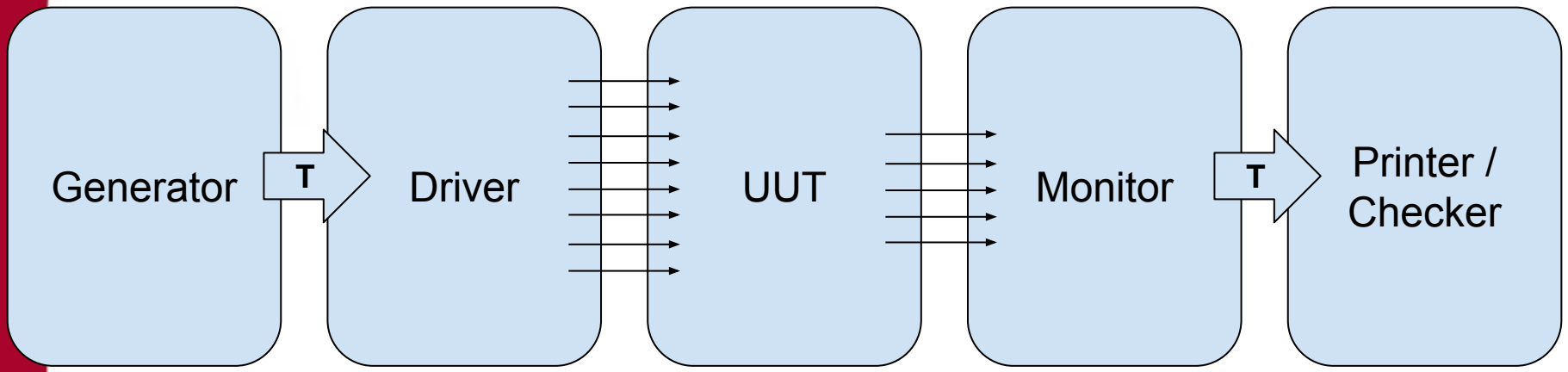
```
fifo_write(data);
```

← `data` propagates through the interface



← associated pin movement

Architecture of a TLM testbench



How to do it?

Define a record with the data associated to each channel:

```
type input_tran is
  record
    a  : std_logic_vector (7 downto 0);
    b  : std_logic_vector (7 downto 0);
    op : op_type;
  end record;
```

How to do it?

- Describe an entity (based on processes and/or procedures) to convert transactions into pin movement
  - Driver
- Describe an entity (based on processes and/or procedures) to convert pin movement into transactions
  - Monitor

Test plan

Now, defining a test plan is easier:

- Given X incoming transaction(s), Y outgoing transaction(s) are expected

More information in the lesson “Designing test plans”

Contents

- What is verification?
- Why verify?
- 0.- Directed testing
- 1.- Code coverage
- 2.- Assertions
- 3.- Transaction-Level Modeling
- 4.- Self-checking testbenches
- 5.- Automatic stimuli
- 6.- Functional coverage
- Bibliography

Self-checking Testbenches

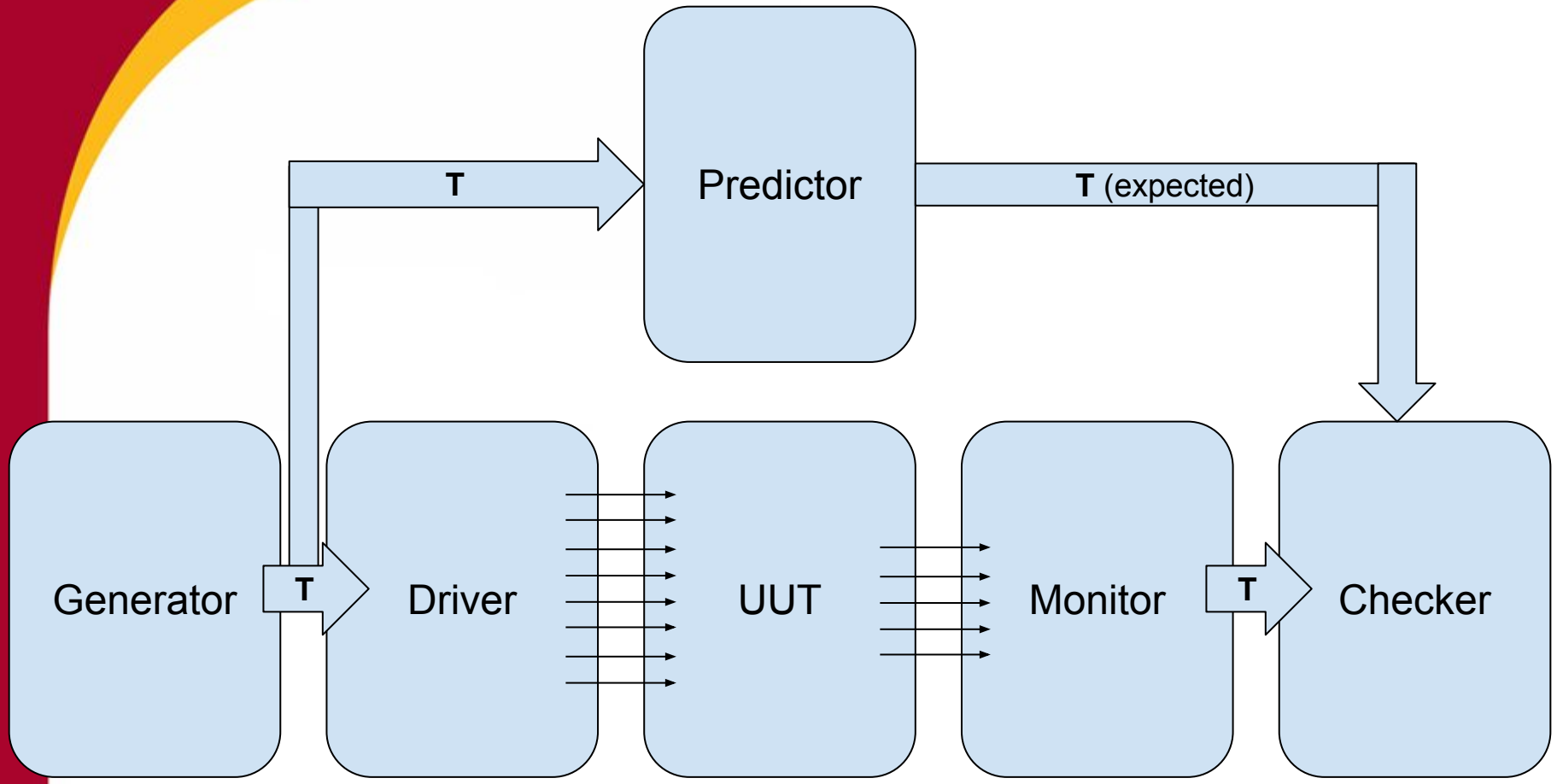
- Instead of manually checking the waveforms, we insert the following checks into the testbench:
  - Whether the pin movements are correct (assertions from the protocol monitor)
  - Whether the output data is correct
- Predictor: predicts expected output transactions
- Checker: verifies if the transactions are correct

Design of predictor+checker

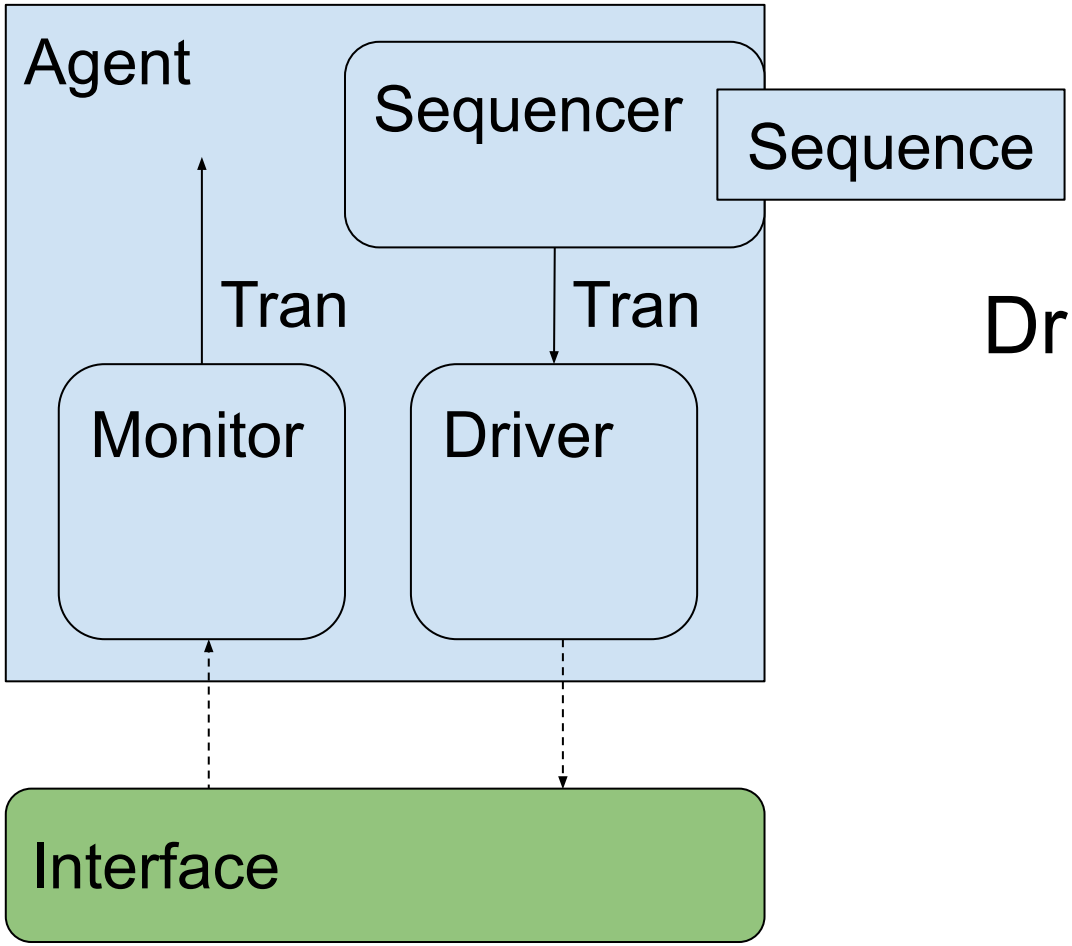
Two options:

1. Generate 'gold' output files from a high-level model
   - For example, for crosscheck with Matlab/octave
2. Integrate the high-level model into the simulation
   - Model implemented in VHDL or (System)Verilog
   - QuestaSim-Matlab interface
   - VHDL-C interface (GHDL, QuestaSim)
   - Python (CoCoTb)

Self-checking Testbenches



Verification agent



Driver + Monitor

Contents

- What is verification?
- Why verify?
- 0.- Directed testing
- 1.- Code coverage
- 2.- Assertions
- 3.- Transaction-Level Modeling
- 4.- Self-checking testbenches
- 5.- Automatic stimuli
- 6.- Functional coverage
- Bibliography

Automatic stimulus

If we have a testbench that includes a high-level model which we can compare against:

- We can generate random stimuli.
- “Random test” as opposed to “directed test”
- Actually, it’s “constrained random” because restrictions are applied to the generated stimuli.

Contents

- What is verification?
- Why verify?
- 0.- Directed testing
- 1.- Code coverage
- 2.- Assertions
- 3.- Transaction-Level Modeling
- 4.- Self-checking testbenches
- 5.- Automatic stimuli
- 6.- Functional coverage
- Bibliography

Functional coverage

Code coverage is very useful, but it does NOT tell us:

- Whether the execution was successful or not
- Whether we have tested all the 'corner cases': values, ranges, etc.
- Whether we are applying the stimuli in the correct sequences

Functional coverage indicates whether we are covering the entire *test plan*

Functional Coverage

Example:

- 16-bit multiplier, 4G possible cases
- At least we should test:
  - positive * positive
  - positive * negative
  - negative * positive
  - negative * negative
  - positive * zero
  - negative * zero
  - zero * positive
  - zero * negative
  - zero * zero

How to do it?

- Bins are defined (think of them as containers/categories)
- When an input transaction is generated, the bin to which the generated transaction belongs is noted
- At the end of the simulation, a report is generated showing the coverage of each bin (number of times each bin is reached)

Typically, third-party packages that already provide this functionality are used (OSVVM CoveragePkg in VHDL)

Contents

- What is verification?
- Why verify?
- 0.- Directed testing
- 1.- Code coverage
- 2.- Assertions
- 3.- Transaction-Level Modeling
- 4.- Self-checking testbenches
- 5.- Automatic stimuli
- 6.- Functional coverage
- Bibliography

Bibliography

- Ray Salemi, *FPGA Simulation: A Complete Step-by-Step Guide.* Boston Light Press, 2009
- Ray Salemi, 'Evolving FPGA verification capabilities', available at www.verificationacademy.com

Learning outcomes

- What are verification metrics used for?
- Differences between code coverage and functional coverage
- Why does it make sense to test with constrained random inputs?
- Understanding the importance of assertions
- Understanding what transaction-level modeling is and how it influences the construction of structured testbenches