Capacidades de Verificación en Circuitos Digitales

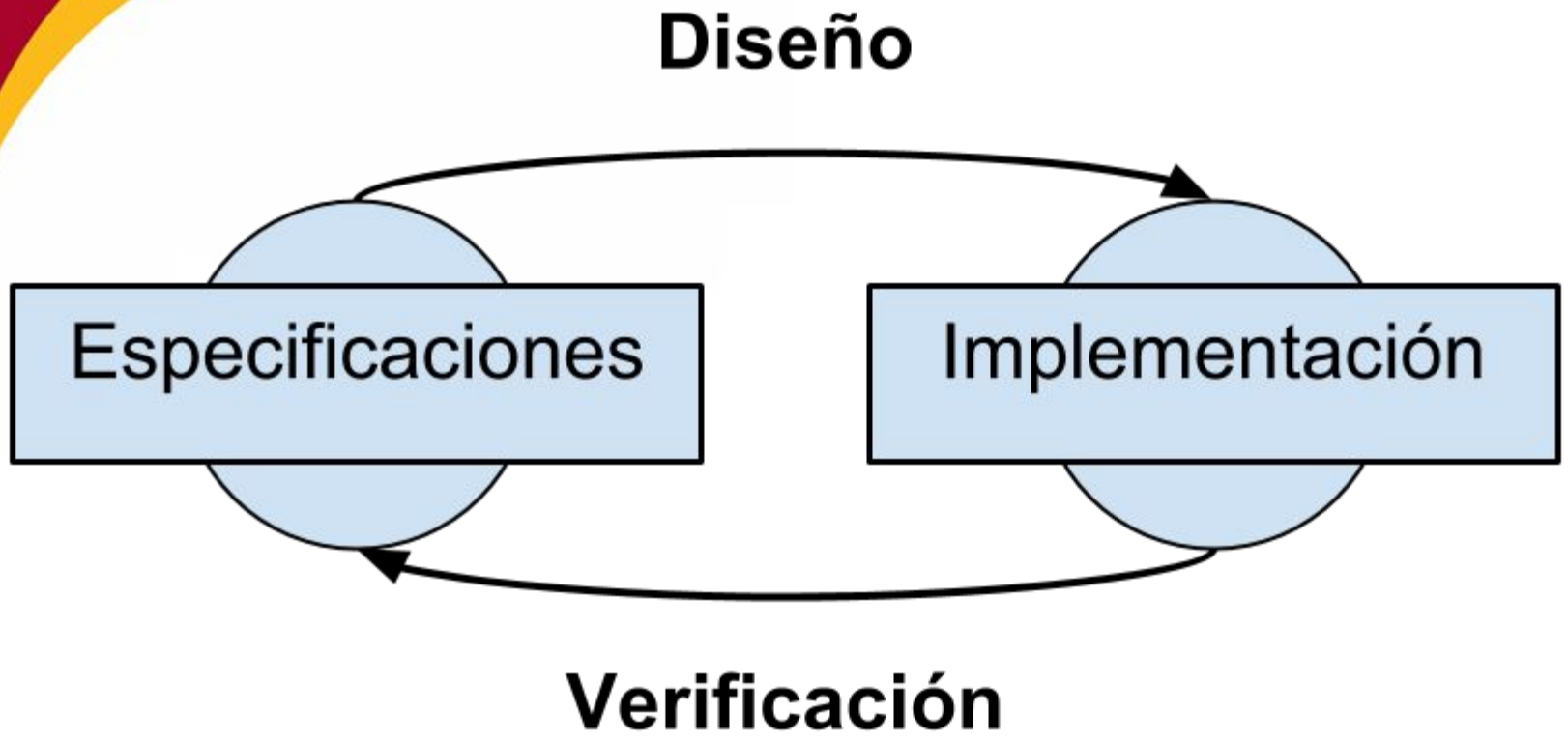
Hipólito Guzmán Miranda
Profesor Contratado Doctor
Universidad de Sevilla

Acknowledgement to Ray Salemi, Mentor Graphics

Capacidades de Verificación

- ¿Qué es la verificación?
- ¿Por qué verificar?
- 0.- Directed Testing
- 1.- Code Coverage
- 2.- Assertions
- 3.- Transactions
- 4.- Self-checking Testbenches
- 5.- Automatic Stimulus
- 6.- Functional Coverage

¿Qué es la verificación?



Comprobar que la implementación realizada realmente cumple con las especificaciones

¿Por qué verificar?

Algunas buenas razones...

- Verification gap
- Salud mental
- Verificación puede ser entre el 50% y el 80% del tiempo total del desarrollo
- Costes de fabricación de ASIC
- Dificultad de diagnosticar y arreglar fallos sobre el prototipo FPGA
- Terminar (de verdad) los proyectos a tiempo

Capacidades de Verificación

- 0.- Test tradicional
- 1.- Code Coverage
- 2.- Assertions
- 3.- Transactions
- 4.- Self-checking Testbenches
- 5.- Automatic Stimulus
- 6.- Functional Coverage

Capacidades de Verificación

- 0.- Test tradicional
- 1.- Code Coverage
- 2.- Assertions
- 3.- Transactions
- 4.- Self-checking Testbenches
- 5.- Automatic Stimulus
- 6.- Functional Coverage

Test ‘tradicional’

Típico testbench:

- Estímulos a mano
- Siempre los mismos estímulos
- Comprobación mirando “a ojo” la forma de onda

Este enfoque no escala para tests complejos

Ej: 200K estímulos y 1M ciclos de reloj de formas de onda que comprobar

Test dirigido vs aleatorio

- Los tests pueden ser de dos tipos:
  - Directed
    - Estímulos determinados de antemano
    - (puedo haber pre-calculado la salida esperada)
  - Random
    - Estímulos generados cada vez que se lanza la simulación
    - (¿cómo sé si la salida es correcta?)

Test dirigido

- En ambos casos, ¡es difícil estar seguro de que lo hemos probado todo!
- ¿Cuándo sé que he acabado de verificar?

Capacidades de Verificación

- 0.- Test tradicional
- 1.- Code Coverage
- 2.- Assertions
- 3.- Transactions
- 4.- Self-checking Testbenches
- 5.- Automatic Stimulus
- 6.- Functional Coverage

Es una técnica automática

- La hace el simulador, compilando los ejecutables de simulación con ciertas opciones
- No soportado por Xilinx ISIM :(
- Soportado por ModelSim/Questa, Aldec, GHDL (parcialmente), etc...

Identificar código que no ha sido probado

- Mnemónico :
  “**S**ome **B**eers **F**or **E**xtra **C**ourage”

- Statement
- Branch
- FSM
- Expression
- Condition

Statement Coverage

- Qué sentencias se han ejecutado y cuáles no
- Sentencia es cualquier cosa que termine en punto y coma
- Una sentencia (como mucho) por línea facilita la labor de cálculo del coverage al simulador

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

- Puede ser que entremos en un if, pero, ¿por cuál `if` / `elsif` / `else` salimos? ¿por cuál '`when X =>`' ?

- Evalúa si se han alcanzado las distintas ramificaciones de nuestro código

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

- Al verificar máquinas de estado, nos interesa saber si hemos cubierto:
  - Los estados
  - Las posibles transiciones entre estados

- Para el '`when others =>`' normalmente hay que añadir una excepción

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

Cuando asignamos:

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

Si salida = ‘1’...

- ¿Es porque a = ‘1’ ?
- ¿Es porque b = ’1’ y c = ‘1’ ?

Si salida = ‘0’...

- ¿Es porque a, b = ‘0’?
- ¿Es porque a, c = ‘0’?

Queremos asegurarnos de que hemos probado todos los casos

Condition Coverage

Como Expression Coverage, pero en las condiciones en lugar de las asignaciones:

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

- ¿a = ‘1’?
- ¿b=’1’ y c=’1’?

else

- ¿a = ‘0’ y b = ‘0’?
- ¿a = ‘0’ y 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 }

Capacidades de Verificación

- 0.- Test tradicional
- 1.- Code Coverage
- 2.- Assertions
- 3.- Transactions
- 4.- Self-checking Testbenches
- 5.- Automatic Stimulus
- 6.- Functional Coverage

Notifica si una condición no se cumple

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

4 niveles de gravedad:

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

¿Son sintetizables?

- No son sintetizables, pero tampoco impiden la síntesis
- El sintetizador en general no mira los assertions
  - Sólo puede mirar aquellos assertions cuya condición sea estática (por ejemplo para evitar síntesis con GENERICS inválidos), y realiza el chequeo en tiempo de síntesis
- Sólo los tiene en cuenta el simulador

Tipos de assertions

- Firewall assertions
  - Para asegurar que tus bloques están siendo usados correctamente
  - Los suele añadir el ingeniero de diseño
- Protocol monitor
  - Para asegurar que diferentes bloques se están comunicando entre sí correctamente (están cumpliendo el protocolo)
  - Los suele añadir el ingeniero de verificación
  - Un protocol monitor tiene más que assertions

(Pero se usan las mismas sentencias VHDL)

Ejemplos

- **En VHDL:**

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

También hay assertions en otros lenguajes como PSL o SystemVerilog

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

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

Si la condición es compleja, mejor en un `if`

- También se puede usar `report` sin `assert`:

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

Capacidades de Verificación

- 0.- Test tradicional
- 1.- Code Coverage
- 2.- Assertions
- 3.- Transactions
- 4.- Self-checking Testbenches
- 5.- Automatic Stimulus
- 6.- Functional Coverage

Transaction-Level Modelling (TLM)

Es elevar el nivel de abstracción en verificación, separando:

- Los datos que se mueven por los interfaces

de

- El movimiento de pines y señales de control asociado

Transaction-Level Modelling (TLM)

Por ejemplo si enviamos datos por una fifo:

1. Ponemos el dato
2. Activamos write_enable
3. Esperamos un ciclo de reloj
4. Desactivamos write_enable

Queremos separar el envío del dato del movimiento de pines (write_enable)

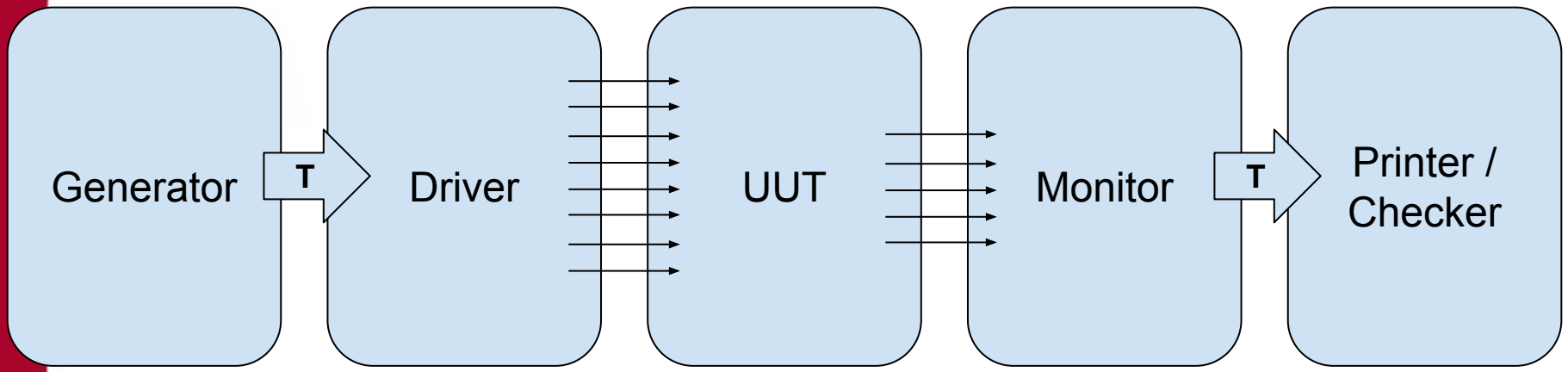
Transaction-Level Modelling (TLM)

Por ejemplo si enviamos datos por una fifo:

1. Ponemos el dato
2. Activamos write_enable
3. Esperamos un ciclo de reloj
4. Desactivamos write_enable

Queremos separar el envío del dato del movimiento de pines (write_enable)

Arquitectura de un testbench TLM



¿Cómo se hace?

Definimos un record (o un protected type en VHDL-2002/2008) con los datos asociados a cada canal:

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

¿Cómo se hace?

- Definimos entidad (basada en processes/procedures) para convertir las transacciones en movimiento de pines (‘driver’)
- Describimos entidad (basada en processes/procedures) para convertir el movimiento de pines en transacciones (‘monitor’)

Plan de pruebas

Ahora es más fácil definir un plan de pruebas:

- Ante X transacción(es) de entrada, se espera Y transacción(es) de salida

Capacidades de Verificación

- 0.- Test tradicional
- 1.- Code Coverage
- 2.- Assertions
- 3.- Transactions
- 4.- Self-checking Testbenches
- 5.- Automatic Stimulus
- 6.- Functional Coverage

Self-checking Testbenches

- En lugar de comprobar a mano las formas de onda, insertamos en el testbench comprobaciones de:
  - Si los movimientos de pines son correctos (assertions del protocol monitor)
  - Si los datos de salida son correctos

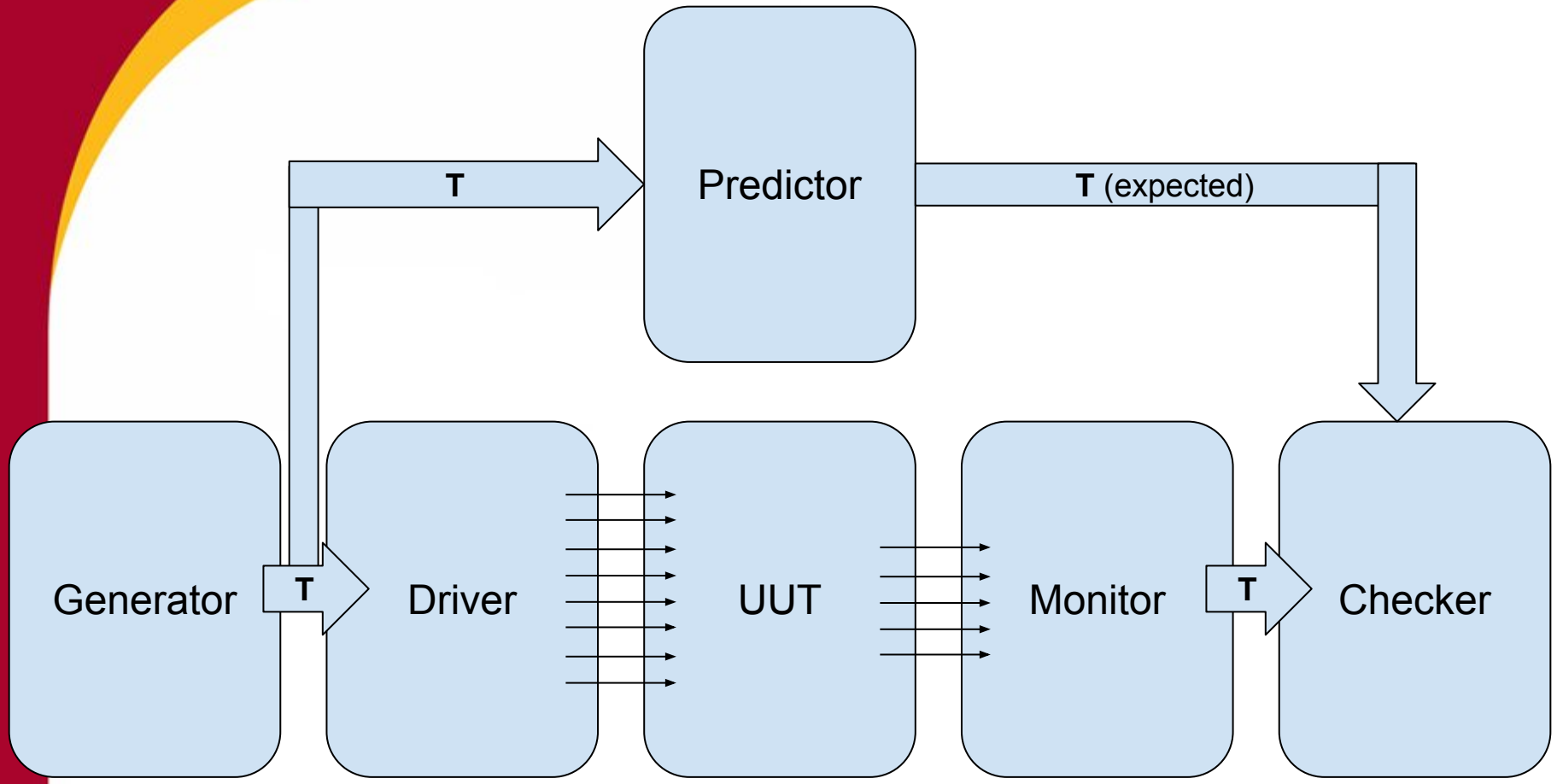
- Predictor: predice las transacciones de salida esperadas
- Checker: comprueba si las transacciones son correctas.

Diseño del checker

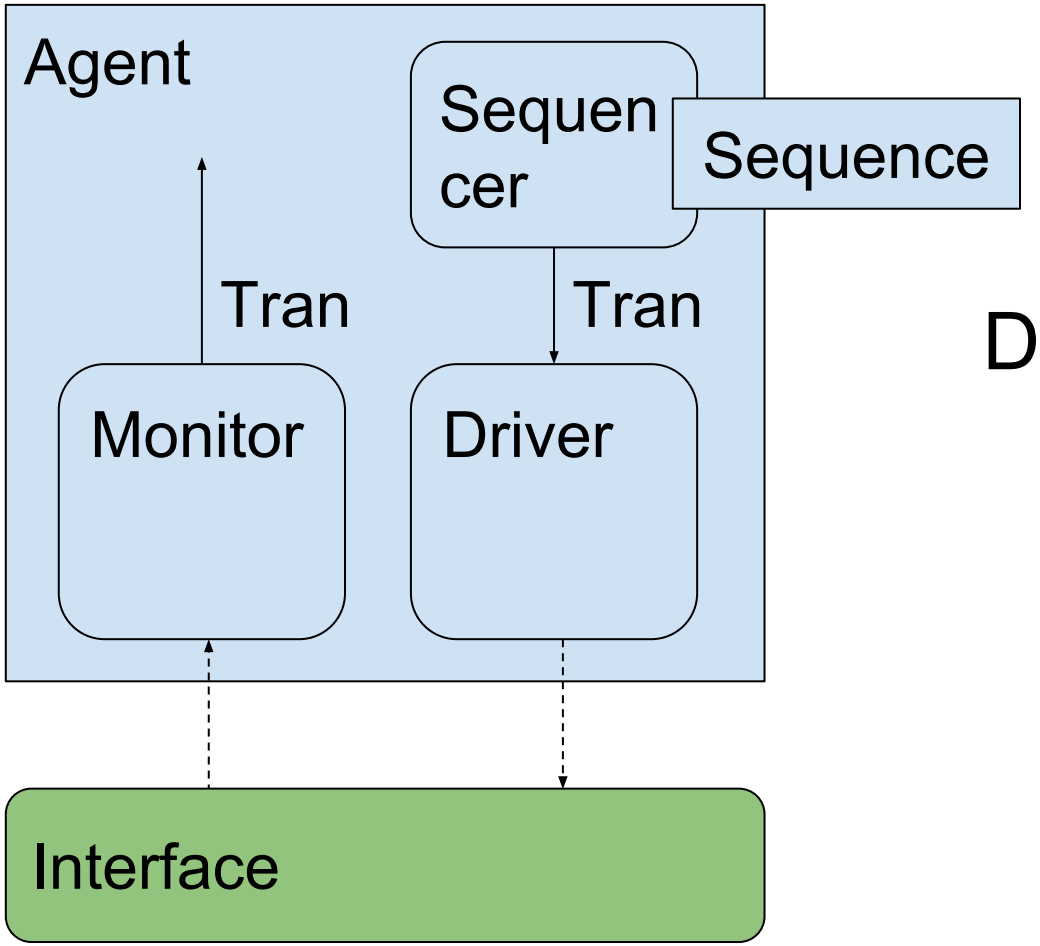
Dos opciones:

1. Generar antes de simular ficheros de salida ‘gold’ provenientes de un modelo de alto nivel
   - Más parecido a un test dirigido, y al diagrama anterior
2. Integrar modelo de alto nivel en la simulación
   - Modelo realizado en SystemVerilog
   - Interfaz Modelsim-Matlab
   - Interfaz VHDL-C (GHDL, QuestaSim)

Self-checking Testbenches



Agente de verificación



Driver + Monitor

Capacidades de Verificación

- 0.- Test tradicional
- 1.- Code Coverage
- 2.- Assertions
- 3.- Transactions
- 4.- Self-checking Testbenches
- 5.- Automatic Stimulus
- 6.- Functional Coverage

Estímulos automáticos

Si tenemos un ‘checker’ que incluye un modelo de alto nivel con el que comparar:

- Podemos generar estímulos aleatorios
- ‘Test random’ como contraposición a ‘test dirigido’
- En realidad es ‘constrained random’ porque se aplican restricciones a los estímulos generados

Capacidades de Verificación

- 0.- Test tradicional
- 1.- Code Coverage
- 2.- Assertions
- 3.- Transactions
- 4.- Self-checking Testbenches
- 5.- Automatic Stimulus
- 6.- Functional Coverage

Alcance Funcional

Code coverage es muy útil pero NO nos dice:

- Si la ejecución fue correcta o no
- Si hemos probado todos los ‘corner cases’: valores, rangos, etc
- Si estamos aplicando los estímulos en secuencias correctas

Functional coverage indica si estamos cubriendo todo el plan de pruebas

Alcance Funcional

Ejemplo:

- Multiplicador 32 bits, 4G casos posibles
- Al menos deberíamos probar:
  - positivo * positivo
  - positivo * negativo
  - negativo * positivo
  - negativo * negativo
  - positivo * cero
  - negativo * cero
  - cero * positivo
  - cero * negativo
  - cero * cero

¿Cómo se hace?

- Se definen ‘bins’ (contenedores)
- Cuando se genera la transacción de entrada se anota a qué bin pertenece la transacción generada
- Al final de la simulación se genera un informe del coverage de cada bin (número de veces que se alcanza cada una)

Normalmente se utilizan packages de terceros que ya dan esta funcionalidad.

Referencias

- Ray Salemi, ‘Evolving FPGA verification capabilities’, disponible en www.verificationacademy.com
- Ray Salemi, ‘FPGA simulation: A Complete Step-by-Step Guide’