Instituto Tecnológico de Costa Rica

Escuela de Ingeniería Electrónica

Ambiente de verificación funcional empleando los estándares UVM y PSS

Documentación

Estudiante:

Mario A. Montero Marenco

2019197658

Cartago, Costa Rica

27 de octubre de 2024

Índice

1. Introducción

En este documento se presentan los resultados más importantes de la implementación del ambiente de verificación funcional con UVM (Universal Verification Methodology) y PSS (Portable Test and Stimulus Standard). El DUT (Device Under Test) se trata del bloque del bus digital del microcontrolador Siwa, el cual cuenta con tres terminales: MBC, SPI y UART, por lo que se pueden enviar y recibir paquetes de información de 65 bits. Para ello, cada paquete tiene una estructura definida, que incluye campos de destino, fuente y carga útil. Adicionalmente, el bus es capaz de realizar operaciones de tipo difusión (broadcast), las cuales se tratan de paquetes que han sido enviados por un terminal y que deben ser recibidas en los otros dos terminales restantes.

UVM permite simplificar el proceso de diseño y construcción del banco de pruebas del bus, puesto que muchas de las clases base ya han sido definidas en una biblioteca. Por otro lado, PSS busca simplificar la generación de estímulos de prueba en todos los niveles de abstracción de un circuito integrado (i.e. desde bloque hasta sistema o SoC). En el caso actual, el modelo PSS genera estímulos que se asocian con secuencias UVM.

2. Plan de pruebas

A continuación se muestran los escenarios que conforman el plan de pruebas del bus digital del microcontrolador Siwa. Para ello, se tomaron en cuenta escenarios generales, así como condiciones de esquina, mostrando el objetivo, los recursos y el nombre de la acción raíz utilizada para generar las pruebas PSS. Sin embargo, el plan puede ser extendido para así obtener un porcentaje de cobertura estructural del 100 % en todos los módulos principales que conforman al bus.

1. **Prueba: Envío de múltiples transacciones aleatorias válidas**
   Objetivo: Comprobar el funcionamiento del bus ante el ingreso de altos volúmenes de paquetes de información.
   Descripción: Este escenario debe generar una cantidad aleatoria de transacciones, de manera que se envíen paquetes desde varias o todas las terminales del bus digital.
   Restricciones: El modelo de estímulos debe ser capaz de aleatorizar:

   - Número de transacciones/secuencias entre 10 y 30.
   - Destino [2:0].
   - Fuente [1:0].
   - Carga útil [59:0].
   - Retardo entre secuencias de 0 a 10 ciclos del reloj.

   Criterio de cumplimiento: Cada una de las transacciones generadas de manera aleatoria llegan al destino asignado aleatoriamente. Para esto, el *scoreboard* debe contar con un banco de datos (ya sean colas o arreglos asociativos) para verificar en tiempo de corrida que lo que se introdujo por un terminal es exactamente igual a lo que salió desde el terminal de destino.
   Nombre de la acción raíz: `random_xfers_a`
   Número de variaciones de la acción raíz: 3

2. **Prueba: Envío de múltiples transacciones desde un dispositivo a todos los demás**
   Objetivo: Comprobar que se pueden enviar paquetes desde un solo dispositivo a todos los demás. Además, se busca generar un *overflow* en las FIFOs de entrada del dispositivo seleccionado.
   Descripción: Este escenario debe generar una cantidad aleatoria de transacciones, de manera que se envíen paquetes desde una terminal seleccionada a todos los demás dispositivos.
   Restricciones: El modelo de estímulos debe ser capaz de aleatorizar:

- Número de transacciones/secuencias entre 10 y 30.
- Destino [2:0].
- Carga útil [59:0].
- Retardo entre secuencias de 0 a 10 ciclos del reloj.

Criterio de cumplimiento: Cada una de las transacciones generadas de manera aleatoria llegan al destino asignado aleatoriamente. Para esto, el *scoreboard* debe verificar en tiempo de corrida que lo que se introdujo por un terminal fuente es exactamente igual a lo que salió desde el terminal de destino.
Nombres de las acciones raíz: `mbc_snd_a`, `spi_snd_a` y `uart_snd_a`
Número de variaciones de cada acción raíz: 3

3. **Prueba: Envío de múltiples transacciones de tipo *broadcast***
Objetivo: Comprobar que se pueden enviar paquetes de tipo *broadcast* desde cada uno de los dispositivos.
Descripción: Este escenario debe generar una cantidad aleatoria de transacciones, de manera que se envíen paquetes de tipo *broadcast* desde cada una de las terminales del bus digital.
Restricciones:

   - Número de transacciones/secuencias entre 10 y 30.
   - Destino fijado a 0x7, según la descripción del diseñador.
   - Fuente [1:0].
   - Carga útil [59:0].
   - Retardo entre secuencias de 0 a 10 ciclos del reloj.

   Criterio de cumplimiento: Cada una de las transacciones de tipo *broadcast* es recibida en todos los terminales, con excepción del terminal que la generó. Por lo que el *scoreboard* debe ser capaz de discernir entre una transacción normal y una de tipo difusión para realizar la comparación de paquetes correctamente.
   Nombres de la acción raíz: `solo_brcst_a`
   Número de variaciones de la acción raíz: 3

4. **Prueba: Envío de múltiples transacciones aleatorias a un solo dispositivo**
Objetivo: Llenar la FIFO de salida del dispositivo que se encarga de recibir los paquetes desde las otras terminales.

Descripción: En este escenario se crea un conjunto aleatorio de transacciones desde dos de los terminales con el fin de llenar y desbordar la FIFO de salida del otro dispositivo disponible.
Restricciones:

- Número de transacciones/secuencias entre 10 y 30.
- Destino fijado a un valor predeterminado.
- Fuente [1:0].
- Carga útil [59:0].
- Retardo entre secuencias de 0 a 10 ciclos del reloj.

Criterio de cumplimiento: Cada una de las transacciones generadas de manera aleatoria llegan al destino seleccionado. En caso de existir *overflow*, el *scoreboard* debe reportar en qué terminal se originó el desborde.
Nombres de las acciones raíz: `solo_a_mbc_a`, `solo_a_spi_a` y `solo_a_uart_a`
Número de variaciones de cada acción raíz: 3

5. **Prueba: Todos los dispositivos envían paquetes en orden de terminal**
Objetivo: Comprobar que se pueden enviar paquetes en orden de terminal al destino correcto.
Descripción: En esta prueba se crean 9 transacciones. En las que se toman en cuenta todas las combinaciones válidas de fuente y destino (incluyendo *broadcast*). Se empieza desde el terminal de MBC, luego el de SPI y por último UART.
Restricciones:

- Destino fijado a un valor predeterminado.
- Fuente fijada a un valor predeterminado.
- Carga útil [59:0].
- Retardo entre secuencias de 0 a 10 ciclos del reloj.

Criterio de cumplimiento: Cada una de las transacciones generadas llegan al destino asignado. El *scoreboard* se encarga de realizar la comprobación mediante el banco de datos.
Nombre de la acción raíz: `todas_a_todas_a`
Número de variaciones de la acción raíz: 1

3. Diseño del ambiente UVM

3.1. Estructura del ambiente

En la Figura 1 se muestran las clases que componen el ambiente de verificación funcional. Se muestran las conexiones entre ellos, mas no así el nombre de los puertos de análisis.

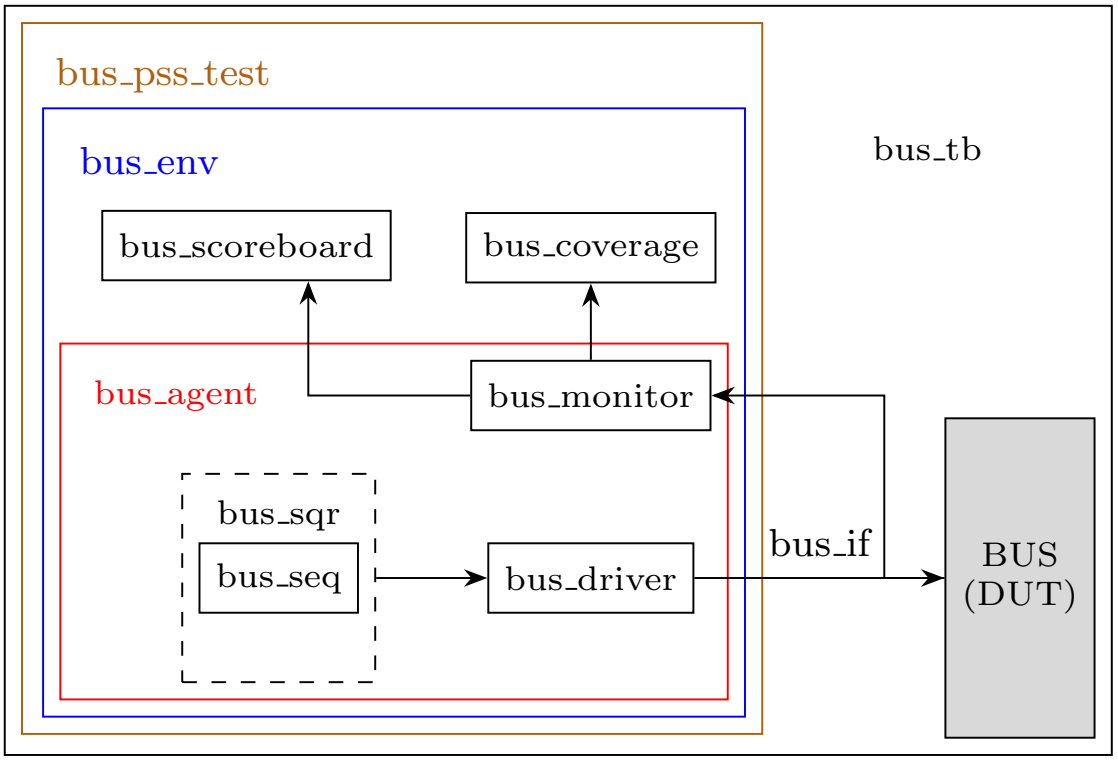


Figura 1: Diagrama del ambiente de verificación. Los componentes contienen los mismos nombres que fueron usados al codificar.

3.2. Interfaz de comunicación

La interfaz del bus digital se define de la forma mostrada en la Figura 2. Al solo tener una interfaz, se creó un solo driver y un solo monitor para manejar todo el protocolo y, así simplificar la creación del ambiente.

```
interface bus_if(
  input bit clk,
  input bit reset
);
// Datos de entrada al DUT
  logic [64:0] D_push_mbc;
  logic [64:0] D_push_spi;
  logic [64:0] D_push_uart;
// Señales que indican si las FIFOs de salida no están vacías
  logic pndng_mbc;
  logic pndng_spi;
  logic pndng_uart;
// Señales utilizadas para introducir datos en el DUT
  logic push_mbc;
  logic push_spi;
  logic push_uart;
// Señales utilizadas para extraer datos del DUT
  logic pop_mbc;
  logic pop_spi;
  logic pop_uart;
// Datos de salida del DUT
  logic [64:0] D_pop_mbc;
  logic [64:0] D_pop_spi;
  logic [64:0] D_pop_uart;
...
```

Figura 2: Definición de la interfaz del bus digital.

3.3. Objeto de secuencia o sequence item

El paquete que utiliza el driver para conducir la información en el bus tiene el formato de la Figura 3. Después, se verá que este formato es el mismo que utiliza el modelo PSS para generar las secuencias.

```
class bus_seq_item extends uvm_sequence_item;
  bit [64:0] rcv_pkt;        // paquete recibido
  bit [64:0] snt_pkt;        // paquete enviado
  rand bit [2:0] target;     // destino
  rand bit [1:0] source;     // fuente
  rand bit [59:0] pkt;       // payload
  rand trans_code op_type;   // tipo de operación
  rand bit [3:0]  delay;     // retardo entre transacciones
...
```

Figura 3: Definición del sequence item.

3.4. Secuencia

La secuencia se definió de una manera distinta debido a que los datos que esta contiene (sequence items) son generados por el modelo PSS. Por ende, se tuvo que crear un task para que la secuencia fuera activada por sí misma y, además, los datos o variables le sean otorgados desde donde se genera la instancia de la secuencia. La Figura 4 muestra el método utilizado para activar la secuencia, nótese que también se debe pasar una instancia de un secuenciador para poder correr la secuencia en el driver.

```
task bus_seq::write(
 input bit [2:0]   target,  // destino
 input bit [1:0]   src,     // fuente
 input bit [59:0]  pkt,     // carga útil
 input trans_code  op_type, // tipo de transacción
 input bit [3:0]   delay,   // retardo
 input bus_sqr     seqr,    // secuenciador
 input uvm_sequence_base parent = null); // secuencia padre
 // asignación de parámetros
 this.target  = target;
 this.src     = src;
 this.pkt     = pkt;
 this.op_type = op_type;
 this.delay   = delay;
 this.start(seqr, parent); // se inicia la secuencia
endtask : write
```

Figura 4: Ejecución de la secuencia `bus_seq` a través del *task* `write()`.

3.5. Driver

El driver puede considerarse un componente clave, puesto que él se encarga de hacer tanto push como pop al DUT. Por ende, si sucede algún error en la comunicación, el driver es lo primero que se debe realizar. En la Figura 5 se muestra el método que se utiliza para hacer push a cada paquete, nótese la importancia del código de operación en la codificación del driver, ya que este es el que restringe/delimita el valor de fuente y destino. Asimismo, se puede intuir que este driver no es capaz de hacer dos o tres push al mismo tiempo, lo cual no presenta ningún inconveniente, ya que el DUT no envía todos los paquetes de una sola vez, él va muestreando las FIFOs una por una para verificar si no están vacías.

El driver cuenta con otro proceso o thread para realizar el pop de los paquetes de las FIFOs de salida, como se puede ver en la Figura 6.

```
virtual task drive_item(bus_seq_item trans);
  @(posedge vif.clk);
  repeat(trans.delay) begin
    @(posedge vif.clk);
  end
  case (trans.source)
    2'b00: begin // FUENTE 0 == HACER PUSH EN MBC
        `DRIV_IF.D_push_mbc <= trans.snt_pkt;
        `DRIV_IF.push_mbc <= 1'b1;
        @(posedge vif.clk);
        `DRIV_IF.push_mbc <= 1'b0;
        `DRIV_IF.D_push_mbc <= 65'b0;
      end
      2'b01: begin // FUENTE 1 == HACER PUSH EN SPI
        `DRIV_IF.D_push_spi <= trans.snt_pkt;
        `DRIV_IF.push_spi <= 1'b1;
        @(posedge vif.clk);
        `DRIV_IF.push_spi <= 1'b0;
        `DRIV_IF.D_push_spi <= 65'b0;
      end
      2'b10: begin // FUENTE 2 == HACER PUSH EN UART
        `DRIV_IF.D_push_uart <= trans.snt_pkt;
        `DRIV_IF.push_uart <= 1'b1;
        @(posedge vif.clk);
        `DRIV_IF.push_uart <= 1'b0;
        `DRIV_IF.D_push_uart <= 65'b0;
      end
      default: begin // SI ES DISTINTO A LOS CASOS ANTERIORES =>
   // REPORTAR ERROR DE FUENTE
        `uvm_fatal("DRV",$sformatf("Fuente␣Inválida,␣valor␣de␣source␣=␣%h",
    trans.source))
      end
    endcase
    trans.print();
  endtask : drive_item
```

Figura 5: Forma en la que se hace push a los paquetes que genera la secuencia. Este case se puede expandir para permitir enviar transacciones con error en el campo de fuente.

```
1  virtual task capture_item();
2    forever begin
3      repeat(5) begin
4           @(posedge vif.clk);
5        end
6      @(posedge vif.clk);
7      if (`DRIV_IF.pndng_mbc) begin
8          `DRIV_IF.pop_mbc <= 1; //Le hace pop al dato del mbc
9          @(posedge vif.clk);
10         `DRIV_IF.pop_mbc <= 0; //pone pop en bajo despues de un ciclo
11     end
12     if (`DRIV_IF.pndng_spi) begin
13       `DRIV_IF.pop_spi <= 1; //Le hace pop al dato del spi
14         @(posedge vif.clk);
15         `DRIV_IF.pop_spi <= 0;
16     end
17     if (`DRIV_IF.pndng_uart) begin
18       `DRIV_IF.pop_uart <= 1; //Le hace pop al dato del uart
19         @(posedge vif.clk);
20         `DRIV_IF.pop_uart <= 0;
21     end
22   end
23 endtask : capture_item
```

Figura 6: Forma en la que se hace pop a los paquetes desde las FIFOs de salida del DUT. Las líneas 3-5 emulan el comportamiento con retardo para hacer pop a un dato, de esta manera se da cabida a generar escenarios de overflow o desborde.

3.6. Monitor

Al igual que el driver, el monitor tiene dos procesos para capturar todos los paquetes que entran y salen del DUT. Posteriormente, toma estos paquetes y los envía al scoreboard por sus dos puertos de análisis. En las Figuras 7 y 8 se puede observar la forma en la que se ejecuta cada proceso. La variable entera `num_xfers` de la Figura 7 se encarga de almacenar la cantidad de transacciones que se han introducido al bus, de esta forma se tiene un control extra del proceso de extracción de paquetes (i.e. no se deben recibir paquetes que no han sido enviados).

Como los paquetes tardan algo de tiempo en ser recibidos, además de que no necesariamente se reciben en el orden en el que se introducen, se tuvo que investigar una manera en la que pueda otorgar cierto tiempo extra a la fase de corrida de UVM para dar chance a que el driver los pueda procesar y extraer. Si bien una opción era poner

tiempo en la fase de corrida del test, se queria poner una solución más "elegante". Por ende, se encontró la solución en [1], la cual se utilizó en la Figura 9, la forma en la que funciona es la siguiente: si el monitor aún tiene elementos en la variable `num_xfers` cuando la secuencia de la prueba finaliza, puede plantear una objeción para retrasar el fin de la simulación. Una vez que la variable sea cero, puede retirar la objeción y permitir que la prueba termine.

Si al final de la simulación, la variable `num_xfers` sigue siendo mayor que cero, en la fase de reporte se indica al usuario que algunas de las transacciones no fueron recibidas (i.e. no se les hizo pop) debido a algún overflow. La tarea de emular el overflow en todas las seis FIFOs no fue implementada en este trabajo debido a que implicaba entrar a las señales internas del DUT, algo que se salía del propósito del proyecto.

```
1  virtual task in_monitor();
2  // captura los datos que se enviaron al DUT (REQUEST == IN)
3      bus_seq_item request;
4      forever begin
5      @(posedge vif.clk);
6      if(vif.push_mbc) begin // hubo push en fifo del MBC
7        `uvm_info("MON", $sformatf("PUSH EN MBC"), UVM_DEBUG)
8        request = bus_seq_item::type_id::create("request");
9        request.snt_pkt = vif.D_push_mbc;
10       request.target = vif.D_push_mbc[64:62];
11       request.source = vif.D_push_mbc[61:60];
12       in_port.write(request); // se envia al scoreboard
13       num_xfers = num_xfers + 1;
14       if (request.target == 3'b111)
15 // sumar dos veces porque en broadcast
16 // el paquete se recibe dos veces (i.e. una vez en cada terminal)
17             num_xfers = num_xfers + 1;
18       end
19     ...
20     if(vif.push_uart) begin // hubo push en fifo del UART
21       `uvm_info("MON", $sformatf("PUSH EN UART"), UVM_DEBUG)
22           request = bus_seq_item::type_id::create("request");
23       request.snt_pkt = vif.D_push_uart;
24       request.target = vif.D_push_uart[64:62];
25       request.source = vif.D_push_uart[61:60];
26       in_port.write(request);
27       num_xfers = num_xfers + 1;
28       if (request.target == 3'b111)
29         num_xfers = num_xfers + 1;
30       end
31     end
32 endtask : in_monitor
```

Figura 7: Tarea encargada de capturar los paquetes que se envían o introducen en las FIFOs de entrada del bus. Se nota, en las líneas 27-28, que cuando hay un broadcast se suma dos veces el contador de transacciones introducidas al bus. De esta forma, cuando se haga pop dos veces al mismo dato, no se generen errores en la otra tarea de captura (ver Figura 8).

```
virtual task out_monitor();
// captura los datos que salen del DUT (LEGACY == OUT)
bus_seq_item legacy; // paquete que sale
bus_seq_item aux; // paquete auxiliar
forever begin : forever_block
  @(negedge vif.clk);
    if ((num_xfers == 0) &&
  (vif.pop_mbc || vif.pop_spi || vif.pop_uart)) begin
// checks if transaction's queue is empty and
// if any pop signal has been triggered
    `uvm_error("MON", $psprintf("Se realizó pop con %0d transacciones
  por procesar", num_xfers))
    end else begin
      if (vif.pop_mbc && (vif.D_pop_mbc[64:60] != 5'b00000)) begin
   // pop en MBC
        `uvm_info("MON", $sformatf("POP EN MBC"), UVM_DEBUG)
        legacy = bus_seq_item::type_id::create("legacy");
        num_xfers = num_xfers - 1;
        legacy.target = 3'b000;
        legacy.source = vif.D_pop_mbc[64:62];
        legacy.rcv_pkt = vif.D_pop_mbc;
        out_port.write(legacy);
      end
    ...
  end
@(posedge vif.clk);
end
endtask : out_monitor
```

Figura 8: Tarea encargada de capturar los paquetes a los que el driver hace pop.

```
virtual function void phase_ready_to_end(uvm_phase phase);
    if (phase.get_name != "run")
      return;
    if (num_xfers != 0) begin
// no se termina la prueba hasta que se reciban
// todos los paquetes generados
      phase.raise_objection(this);
      fork
        delay_phase_end(phase);
      join_none
    end
  endfunction

  virtual task delay_phase_end(uvm_phase phase);
    fork
// se termina la prueba hasta que la cola trans_q este vacía
// o bien hayan pasado 100 tiempos de simulación x transacción
      begin
        wait (num_xfers == 0);
      end
      begin
        #100;
      end
    join_any
    phase.drop_objection(this);
endtask : delay_phase_end
```

Figura 9: Tarea encargada de retardar la fase de corrida del monitor, lo que permite que el driver siga haciendo pop a los paquetes. Puede consultar en [1] para más información acerca de este método

3.7. Scoreboard

La Figura 10 muestra las funciones que utiliza el scoreboard para guardar los paquetes o transacciones provenientes de los dos puertos de análisis del monitor. Por otro lado, en la Figura 11 se observa la fase de corrida de este componente. Lo que se hace basicamente es: esperar a que las colas tengan transacciones, luego se verifica que las transacciones estén en el arreglo asociativo, en caso contrario, se genera un error debido a que hay una inconsistencia en el paquete recibido.

El scoreboard es capaz de discernir cuales de las transacciones enviadas no fueron recibidas, para ello hace uso del arreglo asociativo `exp_trans_array` como se ve en la

Figura 11.

```
function void write_out(bus_seq_item trans);
  `uvm_info("SCB", "Received␣transaction␣from␣FIFO_OUT", UVM_DEBUG)
  trans.print();
  act_trans_q.push_back(trans);
  act_trans_array[trans.rcv_pkt] = trans;
 endfunction : write_out

function void write_in(bus_seq_item trans);
  `uvm_info("SCB", "Received␣transaction␣from␣FIFO_IN", UVM_DEBUG)
  trans.print();
  exp_trans_q.push_back(trans);
  exp_trans_array[trans.snt_pkt] = trans;
endfunction : write_in
```

Figura 10: Métodos encargados de guardar los paquetes en los bancos de datos (i.e. arreglos asociativos y colas) del scoreboard.

```
task compare_bus_trans();
    bus_seq_item act_trans; // actual
    bus_seq_item exp_trans; // expected
    bus_seq_item aux;       // auxiliar
    bit [64:0] idx;         // index
    bus_seq_item aux_asoc_array [bit[64:0]];
    forever begin
      wait((exp_trans_q.size() > 0) && (act_trans_q.size() > 0));
      aux = bus_seq_item::type_id::create("aux");
      aux = act_trans_q.pop_front();
      idx = aux.rcv_pkt;
      if(exp_trans_array.exists(idx)) begin
// si el indice de paquete recibido existe
// en el arreglo de paquetes transmitidos, PASS
        exp_trans = bus_seq_item::type_id::create("exp_trans");
`uvm_info("SCB",$sformatf("----------␣::␣DATA␣Match::␣-----------------"),
UVM_LOW)
`uvm_info("SCB",$sformatf("Expected␣Data:␣%0h", exp_trans.snt_pkt),
UVM_LOW)
`uvm_info("SCB",$sformatf("Actual␣Data␣␣:␣%0h", aux.rcv_pkt),
UVM_LOW)
`uvm_info("SCB",          "-------------------------------------------",
UVM_LOW)
        ...
      end else begin
// si los datos no coinciden,
// se advierte al usuario,
// dado que se generó un error
`uvm_error("SCB", "----------␣::␣DATA␣Mismatch::␣-------------")
`uvm_info("SCB","Expected␣Data:␣???", UVM_LOW)
`uvm_info("SCB",$sformatf("Actual␣Data␣␣:␣%0h", idx), UVM_LOW)
`uvm_info("SCB", "Actual␣data␣isn't␣in␣Expected␣assoc␣array", UVM_LOW)
`uvm_info("SCB", "-------------------------------------------", UVM_LOW)
      end
    end
  endtask
```

Figura 11: Métodos encargados de validar el funcionamiento del DUT. Puede ser visto como un pequeño modelo de referencia.

3.8. Coverage

La cobertura funcional se mide de dos formas: la primera se realiza a nivel de señal en la interfaz del bus, para verificar que las señales de pop, push y pending se activan al menos una vez durante las pruebas, la segunda se mide a nivel de transacción por medio de un subscriptor de UVM. En las Figuras 12 y 13 se muestran los grupos que conforman cada una de ellas.

```
covergroup cg_bus_sig @(posedge clk);
// PUSH SIGNALS
    cp_bus_push_mbc : coverpoint push_mbc {
      bins high = {1};
      bins low  = {0};
    }
    cp_bus_push_spi : coverpoint push_spi {
      bins high = {1};
      bins low  = {0};
    }
    cp_bus_push_uart : coverpoint push_uart {
      bins high = {1};
      bins low  = {0};
    }
// POP SIGNALS
    cp_bus_pop_mbc : coverpoint pop_mbc {
      bins high = {1};
      bins low  = {0};
    }
    cp_bus_pop_spi : coverpoint pop_spi {
      bins high = {1};
      bins low  = {0};
    }
    cp_bus_pop_uart : coverpoint pop_uart {
      bins high = {1};
      bins low  = {0};
    }
// PENDING SIGNALS
  cp_bus_pending_mbc : coverpoint pndng_mbc {
      bins high = {1};
      bins low  = {0};
    }
    cp_bus_pending_spi : coverpoint pndng_spi {
      bins high = {1};
      bins low  = {0};
    }
    cp_bus_pending_uart : coverpoint pndng_uart {
      bins high = {1};
      bins low  = {0};
    }
  endgroup : cg_bus_sig
```

Figura 12: Grupo de cobertura funcional encargado de asegurar que las señales de push, pop y pending se activan al menos una vez. Este grupo fue declarado dentro de la interfaz del bus digital.

```
covergroup cg_bus_xfer;
    cp_bus_target: coverpoint trans.rcv_pkt[64:62] { // TARGET
      bins tg_mbc   = {3'b000};
      bins tg_spi   = {3'b001};
      bins tg_uart  = {3'b010};
      bins tg_brcst = {3'b111};
    }
    cp_bus_src: coverpoint trans.rcv_pkt[61:60] { // SOURCE
      bins src_mbc  = {2'b00};
      bins src_spi  = {2'b01};
      bins src_uart = {2'b10};
    }
    cc_bus_xfer: cross cp_bus_target, cp_bus_src {
    // CRUCE ENTRE TARGET Y SOURCE CON EXCEPCIÓN DE TARGET == SOURCE
      ignore_bins cc_1 = cc_bus_xfer with
      (cp_bus_target == {3'b000} && cp_bus_src == {2'b00});
      ignore_bins cc_2 = cc_bus_xfer with
      (cp_bus_target == {3'b001} && cp_bus_src == {2'b01});
      ignore_bins cc_3 = cc_bus_xfer with
      (cp_bus_target == {3'b010} && cp_bus_src == {2'b10});
    }
endgroup : cg_bus_xfer
```

Figura 13: Grupo de cobertura funcional encargado de asegurar que se producen todas las combinaciones de destino y fuente en los paquetes enviados al bus digital. Este grupo fue declarado dentro de la clase `bus_coverage`.

3.9. Test

Todos los métodos de la clase `bus_pss_test` mantienen una estructura similar a la de un banco de pruebas generado únicamente con UVM, con exclusión de la fase de corrida, como se ve en la Figura 14. Ya que aquí se llama al planificador de VCPS desde la fase de corrida, por medio de la función `pss_run_solution()`. Esto permite que el código de SystemVerilog, generado por el proceso de mapeo de los escenarios, pueda ser utilizado posteriormente por VCS (*Verilog Compiler Simulator*) a la hora de simular.

```
task bus_pss_test::run_phase (uvm_phase phase);
 super.run_phase(phase);
 phase.raise_objection(this);
 pss__pkg::pss_run_solution();
 phase.drop_objection(this);
endtask : run_phase
```

Figura 14: Llamado de la función `pss_run_solution()` del paquete `pss__pkg` desde la fase de ejecución de una prueba UVM. Se recomienda consultar la guía de usuario de VC PS en [2] para obtener más detalles de la razón de esta implementación.

Adicionalmente, con el fin de ver que prueba falló o no, se creó una fase de reporte en el test. En la Figura 15 se muestra lo que se imprimirá en consola y en un archivo de texto llamado `results.txt` en caso de que la prueba falle o no.

```
function void report_phase(uvm_phase phase);
...
  svr = uvm_report_server::get_server();
  if(svr.get_severity_count(UVM_FATAL)+svr.get_severity_count(UVM_ERROR)>0)
 begin
  `uvm_info(get_type_name(), "----------------------------------------", UVM_N
  `uvm_info(get_type_name(), "----␣␣␣␣␣␣␣␣␣␣␣␣TEST␣FAIL␣␣␣␣␣␣␣␣␣␣␣----", UVM_N
  `uvm_info(get_type_name(), "----------------------------------------", UVM_N
  file = $fopen ("results.txt", "a");
  $fdisplay(file, "Test␣name:␣%s",pss_test);
    $fdisplay(file, "FAIL");
    $fdisplay(file, "-----------------------------------------------");
    $fdisplay(file, "␣");
    $fclose(file);
  end
  else begin
  ...
  end
endfunction

function string obtener_pss_test_name();
// esta funcion se encarga de obtener el nombre de la prueba PSS
  string argumentos[$];
// cola con todos los argumentos pasados en la linea de comandos
// con ./simv
    uvm_cmdline_processor clp; // clase de tipo definido por UVM
    clp = uvm_cmdline_processor::get_inst();
// obtengo la instancia de la linea de comandos
    clp.get_args(argumentos);
// se retorna una cola con la linea de comandos y
// se guarda en argumentos
  foreach (argumentos[i]) begin
 // recorro todas las posiciones de la cola
      if (argumentos[i] == "-pss_test") begin
// si encuentro el switch -pss_test
        return argumentos[i+1];
// retorno el valor del siguiente argumento
      end
// dado que corresponde SI o SI con el nombre del test
    end
endfunction : obtener_pss_test_name
```

Figura 15: Función para reportar que la prueba pasó o falló.

3.10. Otros componentes

Los componentes secuenciador, agente y ambiente siguen un esquema genérico, por lo que no se incluye código de ninguno de ellos. Asimismo, tampoco se incluye nada del testbench porque su construcción es muy genérica y no se considera necesario explicar nada de ello en este reporte.

4. Diseño del modelo PSS

Con el ambiente y la estructura de la secuencia definida, se creó el modelo PSS. Este tiene como finalidad generar el contenido o los estímulos que son transportados por la secuencia UVM.

Como al principio no se tenía idea de cómo utilizar la herramienta VC PS, se tuvo que acudir tanto al manual como a un ejemplo que otorga Synopsys. Sin embargo, el ejemplo de Synopsys corresponde a un SoC, que integra múltiples módulos y es verificado mediante una combinación de C y SystemVerilog. Por lo que nunca se logró hacer funcionar la simulación porque se carece de esos conocimientos. Por ende, el modelo PSS desarrollado en este TFG solo corresponde a una pequeña muestra de lo que se puede lograr con el estándar y esta nueva herramienta.

4.1. Paquete con definiciones base

Para crear un diseño modular y más ordenado, PSS define paquetes como en SystemVerilog. En la Figura 16 se muestra la definición del paquete que contiene definiciones base para poder generar los escenarios de prueba. Notese como se debe importar las bibliotecas base definidas por el equipo de Synopsys, así como unos métodos estándar para declarar ejecutores y, también, se define el enum con los tipos de operación del bus digital.

Por otra parte, en este paquete se definen también el struct del mensaje o transacción con sus respectivas restricciones, así como el buffer como objeto de flujo para conectar las acciones y dos acciones abstractas que otras hereden de ellas, todo esto se puede observar en la Figura 17.

```
1  package user_executor_pkg {
2    import executor_pkg::*; // importa el paquete con las funciones base
         definidas por Synopsys
3
4    struct bus_executor_trait : executor_trait_s {
5  // este struct hereda de otro struct definido por Accellera/Synopsys
6      rand int id; // ID para identificar el ejecutor <=> sequencer
7    };
8
9    resource bus_executor_r : executor_claim_s<bus_executor_trait> {
10 // esto es estándar, es para asignar un número de instancia al executor
11     constraint trait.id == instance_id;
12   }
13
14   enum op_type_e {
15   // enum con los tipos de operación, esto es utilizado por
16   // el UVM driver para saber a cual FIFO de entrada del bus
17   // le debe hacer push
18     MBC2SPI    = 0,
19     MBC2UART   = 1,
20     SPI2MBC    = 2,
21     SPI2UART   = 3,
22     UART2MBC   = 4,
23     UART2SPI   = 5,
24     MBCBRCST   = 6,
25     SPIBRCST   = 7,
26     UARTBRCST = 8
27   };
28   ...
```

Figura 16: Paquete con definiciones base para generar los escenarios.

```
1  ...
2  struct message_s { // estructura de la transaccion, emula al sequence
       item
3      rand bit [59:0] pkt; // paquete aka payload
4      rand op_type_e op_type; // tipo de transacción
5      rand bit [2:0] tgt; //destino aka target
6      rand bit [1:0] src; // fuente aka source
7      rand bit [3:0] delay; // retardo aka delay
8      constraint delay in [0..10]; // retardo entre transacciones de entre
           0 y 10 ciclos del reloj
9      constraint { // valores de source y target en función del tipo de
           operación
10       if (op_type == MBC2SPI)    { tgt == 1; src == 0;} ;
11       if (op_type == MBC2UART)   { tgt == 2; src == 0;} ;
12       if (op_type == SPI2MBC)    { tgt == 0; src == 1;} ;
13       if (op_type == SPI2UART)   { tgt == 2; src == 1;} ;
14       if (op_type == UART2MBC)   { tgt == 0; src == 2;} ;
15       if (op_type == UART2SPI)   { tgt == 1; src == 2;} ;
16       if (op_type == MBCBRCST)   { tgt == 7; src == 0;} ;
17       if (op_type == SPIBRCST)   { tgt == 7; src == 1;} ;
18       if (op_type == UARTBRCST) { tgt == 7; src == 2;} ;
19     }
20   };
21
22   buffer xfer_b { // declaración de un buffer (flow object) para conectar
         las acciones
23     rand message_s msg; // tipo de datos que puede manejar el buffer
24   }
25 // ACCIONES ABSTRACTAS: SOLO SIRVEN PARA HEREDAR, NO PUEDEN SER
26 // INSTANCIADAS POR SI SOLAS, COMO LAS CLASES VIRTUALES EN SYSTEMVERILOG
27   abstract action bus_prod_xfer_a {
28     output xfer_b dat_o;   // esta acción genera el mensaje y lo
           transmite a través del buffer
29   }
30
31   abstract action bus_copy_xfer_a { //
32     input xfer_b dat_i;  // esta acción permite mapear el mensaje aka
           struct en una secuencia, recibe un mensaje a traves de un buffer
           predefinido
33   }
34 }
```

Figura 17: Definición de estructuras, objetos de flujo y acciones abstractas dentro del paquete `user_executor_pkg`.

4.2. Componente `bus_c`

Para crear las acciones que proveen los estímulos del bus digital, se creó un componente por aparte llamado `bus_c`. En él se importan las bibliotecas base de Synopsys, así como los llamados `pool`. Lo cual permite que todas las acciones puedan acceder al contenido del buffer. Posteriormente, se definen todas las acciones básicas con los tipos de transacción, como se observa en la Figura 18.

```
1  ...
2  action mbc2spi_a: bus_prod_xfer_a { // MBC TO SPI ACTION
3      constraint {
4        dat_o.msg.op_type == MBC2SPI;
5      };
6    }
7  ...
8    action spi2uart_a: bus_prod_xfer_a { // SPI TO UART ACTION
9      constraint {
10       dat_o.msg.op_type == SPI2UART;
11     }
12   }
13 ...
14   action uart2mbc_a: bus_prod_xfer_a { // UART TO MBC ACTION
15     constraint {
16       dat_o.msg.op_type == UART2MBC;
17     }
18   }
19 ...
20   action uartbrcst_a: bus_prod_xfer_a { // UART BROADCAST ACTION
21     constraint {
22       dat_o.msg.op_type == UARTBRCST;
23     }
24   }
25 ...
26   action rand_xfer_a: bus_prod_xfer_a { // RANDOM TRANSFER ACTION
27     //output xfer_b dat_o;
28   }
29 ...
```

Figura 18: Definición de acciones base para modelar estímulos. Esto equivale a tener varios tipos de secuencias de UVM.

En la Figura 19 muestra la plantilla utilizada por la acción atómica `bus_copy_a` para el mapeo de la transacción. Se puede observar el uso de expresiones *Mustache* (envueltas con dobles llaves {}) para que la herramienta VCPS se encargue de reemplazarlas dinámicamente por los valores reales de las rutas y los datos generados para cada

prueba.

```
extend action bus_copy_a { // acción atómica
  exec body SV = """
  string path;
  bus_seq seq;
  bus_sqr sqcr;
  $sformat(path, "%s", "{{pss_top.ex_sv.get_path()}}");
  if (!$cast(sqcr, uvm_top.find(path)))
    `uvm_fatal("ENV_SV", $sformatf("PATH does not exist:%s",path));
  seq = bus_seq::type_id::create("seq");
  seq.write({{dat_i.msg.tgt}}, {{dat_i.msg.src}}, {{dat_i.msg.pkt}}, {{
      dat_i.msg.op_type}}, {{dat_i.msg.delay}}, sqcr);
  """;
}
```

Figura 19: Definición de la acción atómica `bus_copy_a` con plantilla para mapear código en secuencias UVM en SystemVerilog.

Por otro lado, note que la acción de la Figura 19 utilizará un handler del secuenciador y la secuencia. Por ende, se debe tener otra acción que defina o importe las definiciones de estos componentes para que, a la hora de compilar el banco de pruebas con VCS, no se generen errores debido a que no encuentran las definiciones de estos elementos. Esto se realizó mediante la acción `initialize_a`, como se puede ver en la Figura 20.

```
action initialize_a { // ATOMIC ACTION 2 (HEADER), esta solo quiero que
    se "imprima" una vez para que no hayan errores a la hora de compilar
    en VCS, por ende la pongo en una acción aparte
    (* instance *)
    exec declaration SV = """
      import bus_env_pkg::*;
      import bus_seq_pkg::*;
    """;
  }
```

Figura 20: Definición de la acción atómica `initialize_a` para importar los paquetes que definen a la secuencia y al secuenciador. La palabra `instance` se utiliza para que el código se imprima una sola vez cada vez que sea instanciada la acción atómica (veáse [2]).

4.2.1. Extensión del componente `bus_c`

Otra de las ventajas que otorga el estándar PSS es la extensión de componentes y acciones. Esto permite que el código del componente `bus_c` sea trabajado en otro archivo, y posteriormente, el compilador tome esa extensión y la interprete junto a la definición base del componente. Cabe recalcar que, a diferencia de SystemVerilog, extender no es lo mismo que heredar en PSS. Extender solo permite añadir definiciones extra que complementan a las básicas.

En la Figura 21 se muestran las acciones compuestas que fueron añadidas al extender el componente `bus_c`. Se les dice acciones compuestas debido a que tienen un bloque de actividad dentro de ellas. Cabe resaltar que las acciones dentro del bloque activity se ejecutan de manera secuencial por defecto, uno puede hacer también que se ejecuten de forma paralela mediante el bloque `parallel`.

```
1  extend component bus_c { // extiendo del componente base
2  // definición de acciones compuestas, los bloques activity
3  // se ejecutan de manera secuencial por defecto
4    action mbc2spi_xfers_a {
5      activity {
6        do mbc2spi_a;
7        do bus_copy_a;
8      }
9    } // action
10 ...
11   action spi2mbc_xfers_a {
12     activity {
13       do spi2mbc_a;
14       do bus_copy_a;
15     }
16   }
17 ...
18   action uart2mbc_xfers_a {
19     activity {
20       do uart2mbc_a;
21       do bus_copy_a;
22     }
23   }
24 ...
25   action spibrcst_xfers_a {
26     activity {
27       do spibrcst_a;
28       do bus_copy_a;
29     }
30   }
31 ...
32   action rand_xfers_a {
33     activity {
34       do rand_xfer_a;
35       do bus_copy_a;
36     }
37   }
38 } // component
```

Figura 21: Definición de las acciones compuestas para generar los estímulos de prueba. Todas siguen una estructura similar, puesto que el driver solo necesita diferenciar el valor del tipo de operación declarado en el enum de la Figura 17.

4.3. Componente top o `pss_top`

En el componente de mayor jerarquía, llamado `pss_top` en el código fuente, se deben definir las acciones raíz (*root actions*) para que la herramienta sepa a partir de cuáles debe empezar a generar los estímulos o secuencias. En la Figura 22 se muestran algunas de esas acciones, las cuales son llamadas a la hora de resolver (solve) con VC PS.

```
1  // modulo top del modelo PSS
2  component pss_top {
3  ...
4      action mbc_snd_a {
5      rand int in [10..30] num_of_xfers; // defino un número aleatorio de
           secuencias, lo que se traduce en transacciones, ya que van en
           regla 1 a 1.
6      activity {
7        do bus_c::initialize_a; // initialize
8        repeat(i:num_of_xfers){
9          select { // selección aleatoria
10           do bus_c::mbc2spi_xfers_a;
11           do bus_c::mbc2uart_xfers_a;
12           do bus_c::mbcbrcst_xfers_a;
13         }
14       }
15     } // activity
16   } // action
17
18   action todas_a_todas_a {
19     activity {
20       sequence {
21         do bus_c::initialize_a; // initialize
22         // MBC TRANSFERS
23         do bus_c::mbc2spi_xfers_a;
24         do bus_c::mbc2uart_xfers_a;
25         do bus_c::mbcbrcst_xfers_a;
26         // SPI TRANSFERS
27         do bus_c::spi2mbc_xfers_a;
28         do bus_c::spi2uart_xfers_a;
29         do bus_c::spibrcst_xfers_a;
30         // UART TRANSFERS
31         do bus_c::uart2mbc_xfers_a;
32         do bus_c::uart2spi_xfers_a;
33         do bus_c::uartbrcst_xfers_a;
34       } // sequence
35     } // action
36   } // action
37 ...
38 } // component
```

Figura 22: Definición de las acciones raíz o escenarios de prueba.

5. Indicaciones para generar las pruebas

Como se ha mencionado varias veces, las pruebas son generadas con VC PS de Synopsys, por lo que se recomienda que consulte los comandos y switches que son utilizados para compilar, resolver y mapear. A modo de ejemplo, se mencionan a continuación los comandos utilizados para generar las 25 pruebas PSS.

Para correr la prueba primero debe cargar los punteros a las herramientas de Synopsys, por lo tanto, en el directorio `trabajo-final-graduacion` debe ejecutar:

```
source setup.sh
```

Posteriormente, debe moverse hasta la carpeta donde está el código fuente del modelo PSS, entonces:

```
cd ./src/tb/pss
```

Dentro de ese directorio, se creó un Makefile para facilitar todo el proceso. Por lo tanto, para compilar haga lo siguiente

```
gmake compile_pss
```

Para resolver, debe indicar cuál acción raíz quiere resolver, entre las opciones están: `mbc_snd_a , spi_snd_a, uart_snd_a, solo_brcst_a, random_xfers_a, todas_a_todas_a, solo_a_mbc_a, solo_a_spi_a, solo_a_uart_a`. Por ende, ejecute lo siguiente

```
gmake solve_pss ROOT_ACTION=<nombre de la acción raíz>
```

Adicionalmente, usted puede cambiar tanto la semilla que utiliza VC PS para resolver como la cantidad de variaciones por escenario. Esto se realiza en el Makefile, antes de resolver, bajo las variables `SEED` y `NUM_CASES`, respectivamente.

Para mapear, tiene que especificar el nombre del test, aquí tiene dos opciones: la primera es especificar el nombre específico del test, el cual tiene la forma `ROOT_ACTION_NAME_#`; la otra opción es poner "*" (con las comillas) para mapear todas las pruebas que hayan sido resueltas. Entonces debe ejecutar el siguiente comando

```
gmake map_pss TEST_NAME=<nombre de la prueba>
```

Todo el proceso mencionado en esta sección le creará un directorio, por defecto nombrado `snps_vcps_dir`, en el cual se encuentran todas las pruebas generadas a partir del modelo PSS descrito en la sección anterior. Para comprender la estructura de ese directorio, consulte el manual [2].

De manera adicional, debe compilar un pequeño código en lenguaje C. Por alguna razón que hasta el momento se desconoce (puede ser un bug de la herramienta, ya que

se encuentra todavía en desarrollo), este código es necesario a la hora de simular con VCS, ya que el planificador de VCPS debe imprimir unos mensajes mientras se ejecutan las acciones, o al menos así se define en [2]. Por ende, debe ejecutar lo siguiente:

```
gmake compile_target_functions
```

6. Indicaciones para ejecutar las pruebas

Una vez haya generado todas las pruebas, debe moverse al directorio principal/raíz conocido como `trabajo-final-graduacion`. Aquí tendrá dos opciones, si quiere ejecutar una prueba en específico puede moverse al directorio `vcps`, si quiere correr la "regresión" dirijase a `sim`. A continuación se mencionan los dos posibles caminos:

- **Para una prueba en específico:**
  Primero debe compilar todo el banco de pruebas, por lo que ejecute:

  ```
  gmake vcs_compile
  ```

  Para simular, ejecute el comando

  ```
  gmake simulate_pss_sol PSS_TEST=<nombre de la prueba>
  ```

  En donde `PSS_TEST` tiene el mismo nombre que los utilizados nombres usados para las acciones raíz al crear las pruebas; es decir, `ROOT_ACTION_NAME_#`. También es posible cambiar la verbosidad de la prueba mediante la variable `VERBOSIDAD` (por defecto está en `UVM_DEBUG`).
  Lo anterior, además de generarle la salida de la prueba, le crea también una base de datos de cobertura, la cual puede visualizar en Verdi a través del comando

  ```
  gmake verdi
  ```

  Cabe resaltar que se pueden correr todas las pruebas a través del mismo ejecutable debido a que, en el banco de pruebas o módulo `bus_tb`, no se especificó cuál prueba compilar (i.e. se dejó la tarea `run_test() vacía`). Además, todas las pruebas PSS son importadas en un paquete que se llama en la fase de ejecución de la prueba `bus_pss_test`, por lo que ya todas deben estar compiladas.

- **Para la regresión:**
  Una vez en el directorio `sim`, ejecute el script mediante el comando

  ```
  source cmd.sh
  ```

  En caso de que no se haya clonado como ejecutable, ejecute los siguientes comandos

```
chmod +x cmd.sh
source cmd.sh
```

Como alternativa a source, puede utilizar

```
./cmd.sh
```

Este script tiene como finalidad generar tres bases de datos que se pueden importar en Verdi: `COV_DB`, `FS_DB` y la última se genera en el directorio

```
trabajo-final-graduacion/src/tb/pss/snps_vcps_dir/debug_dir
```

`COV_DB` contiene la base de datos de cobertura de cada prueba, por lo que si quiere visualizar los resultados de una prueba específica en Verdi ejecute

```
verdi -cov -covdir COV_DB/cov_dir_<nombre de la prueba>.vdb &
```

Si quiere visualizar el porcentaje de cobertura global (merge de todas las pruebas) ejecute

```
verdi -cov -covdir COV_DB/* &
```

Las bases de datos incluidas en los directorios `FS_DB` y `debug_dir` se pueden utilizar para visualizar en Verdi las formas de onda y la duración de las acciones principales que conforman cada prueba. Para ello puede utilizar el comando

```
verdi -lca -ssf <nombre de la prueba>.fsdb
```

Adicionalmente, el script también le genera un URG report para cada prueba, dentro del directorio `sim/REPORTES`.

7. Resultados

En la presente sección se muestran los resultados más relevantes de todo el proceso.

La Figura 23 muestra la forma en la que se comunica el driver con el DUT.

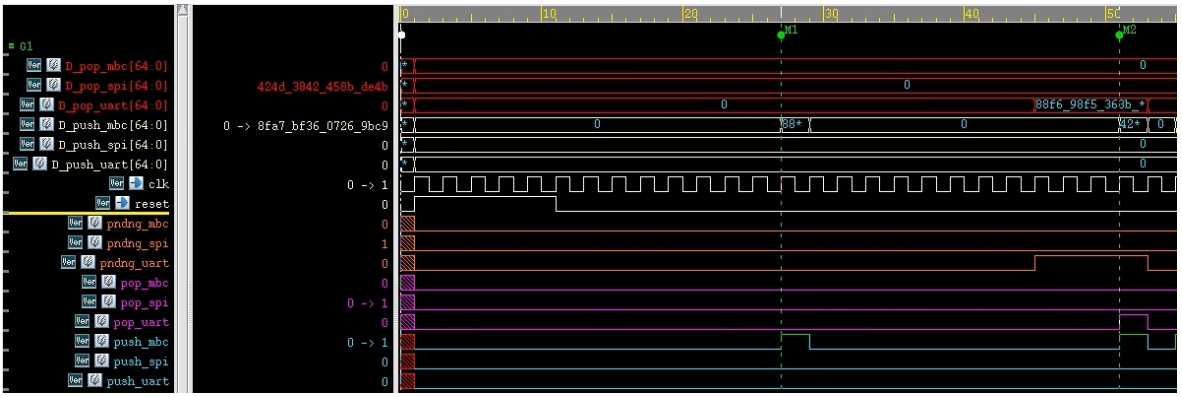


Figura 23: Forma de onda del protocolo de envío y recepción de los paquetes de una prueba en específico.

Cuando el scoreboard decide que la transacción fue enviada y recibida correctamente, se muestra una salida en la consola como la de la Figura 24. En caso contrario, saltará una advertencia como en las Figuras 25 y 26.

```
UVM_INFO /mnt/vol_NFS_rh003/estudiantes/mario_montero/trabajo-final-graduacion/src/tb/uvm_tb/bus_scoreboard.sv(66) @ 68: uvm_test_top.e0.sb0 [SCB] ---------- :: DATA Match:: ----------
UVM_INFO /mnt/vol_NFS_rh003/estudiantes/mario_montero/trabajo-final-graduacion/src/tb/uvm_tb/bus_scoreboard.sv(67) @ 68: uvm_test_top.e0.sb0 [SCB] Expected Data: 1c8fd846030d900d3
UVM_INFO /mnt/vol_NFS_rh003/estudiantes/mario_montero/trabajo-final-graduacion/src/tb/uvm_tb/bus_scoreboard.sv(68) @ 68: uvm_test_top.e0.sb0 [SCB] Actual Data  : 1c8fd846030d900d3
UVM_INFO /mnt/vol_NFS_rh003/estudiantes/mario_montero/trabajo-final-graduacion/src/tb/uvm_tb/bus_scoreboard.sv(69) @ 68: uvm_test_top.e0.sb0 [SCB] ----------------------------------------------
UVM_INFO /mnt/vol_NFS_rh003/estudiantes/mario_montero/trabajo-final-graduacion/src/tb/uvm_tb/bus_scoreboard.sv(66) @ 70: uvm_test_top.e0.sb0 [SCB] ---------- :: DATA Match:: ----------
UVM_INFO /mnt/vol_NFS_rh003/estudiantes/mario_montero/trabajo-final-graduacion/src/tb/uvm_tb/bus_scoreboard.sv(67) @ 70: uvm_test_top.e0.sb0 [SCB] Expected Data: 1c8fd846030d900d3
UVM_INFO /mnt/vol_NFS_rh003/estudiantes/mario_montero/trabajo-final-graduacion/src/tb/uvm_tb/bus_scoreboard.sv(68) @ 70: uvm_test_top.e0.sb0 [SCB] Actual Data  : 1c8fd846030d900d3
UVM_INFO /mnt/vol_NFS_rh003/estudiantes/mario_montero/trabajo-final-graduacion/src/tb/uvm_tb/bus_scoreboard.sv(69) @ 70: uvm_test_top.e0.sb0 [SCB] ----------------------------------------------
UVM_INFO /mnt/vol_NFS_rh003/estudiantes/mario_montero/trabajo-final-graduacion/src/tb/uvm_tb/bus_scoreboard.sv(66) @ 84: uvm_test_top.e0.sb0 [SCB] ---------- :: DATA Match:: ----------
UVM_INFO /mnt/vol_NFS_rh003/estudiantes/mario_montero/trabajo-final-graduacion/src/tb/uvm_tb/bus_scoreboard.sv(67) @ 84: uvm_test_top.e0.sb0 [SCB] Expected Data: 9747419l2d771e62
UVM_INFO /mnt/vol_NFS_rh003/estudiantes/mario_montero/trabajo-final-graduacion/src/tb/uvm_tb/bus_scoreboard.sv(68) @ 84: uvm_test_top.e0.sb0 [SCB] Actual Data  : 9747419l2d771e62
UVM_INFO /mnt/vol_NFS_rh003/estudiantes/mario_montero/trabajo-final-graduacion/src/tb/uvm_tb/bus_scoreboard.sv(69) @ 84: uvm_test_top.e0.sb0 [SCB] ----------------------------------------------
```

Figura 24: Salida mostrada cuando el paquete se recibe correctamente.

```
UVM_WARNING /mnt/vol_NFS_rh003/estudiantes/mario_montero/trabajo-final-graduacion/src/tb/uvm_tb/bus_monitor.sv(139) @ 2423: uvm_test_top.e0.a0.monitor [MON] No se recibieron 3 trans. debido a OVERFLOW
 en alguna FIFO
```

Figura 25: Salida mostrada cuando se detecta que se genera un overflow.

```
UVM_INFO /mnt/vol_NFS_rh003/estudiantes/mario_montero/trabajo-final-graduacion/src/tb/uvm_tb/bus_scoreboard.sv(105) @ 2423: uvm_test_top.e0.sb0 [SCB] ----------------------------------------------
UVM_WARNING /mnt/vol_NFS_rh003/estudiantes/mario_montero/trabajo-final-graduacion/src/tb/uvm_tb/bus_scoreboard.sv(106) @ 2423: uvm_test_top.e0.sb0 [SCB] The Expected asoc array hasn't completed 3
UVM_INFO /mnt/vol_NFS_rh003/estudiantes/mario_montero/trabajo-final-graduacion/src/tb/uvm_tb/bus_scoreboard.sv(107) @ 2423: uvm_test_top.e0.sb0 [SCB] Transfers in Expected assoc array:
UVM_INFO /mnt/vol_NFS_rh003/estudiantes/mario_montero/trabajo-final-graduacion/src/tb/uvm_tb/bus_scoreboard.sv(108) @ 2423: uvm_test_top.e0.sb0 [SCB] ----------------------------------------------
UVM_INFO /mnt/vol_NFS_rh003/estudiantes/mario_montero/trabajo-final-graduacion/src/tb/uvm_tb/bus_scoreboard.sv(110) @ 2423: uvm_test_top.e0.sb0 [SCB] Transfer  : 82bf390140091db1
UVM_INFO /mnt/vol_NFS_rh003/estudiantes/mario_montero/trabajo-final-graduacion/src/tb/uvm_tb/bus_scoreboard.sv(110) @ 2423: uvm_test_top.e0.sb0 [SCB] Transfer  : 94b702b5e9bec53f
UVM_INFO /mnt/vol_NFS_rh003/estudiantes/mario_montero/trabajo-final-graduacion/src/tb/uvm_tb/bus_scoreboard.sv(110) @ 2423: uvm_test_top.e0.sb0 [SCB] Transfer  : 9bcdaaa6e89ffa3b
UVM_INFO /mnt/vol_NFS_rh003/estudiantes/mario_montero/trabajo-final-graduacion/src/tb/uvm_tb/bus_scoreboard.sv(112) @ 2423: uvm_test_top.e0.sb0 [SCB] ----------------------------------------------
UVM_INFO /mnt/vol_NFS_rh003/estudiantes/mario_montero/trabajo-final-graduacion/src/tb/uvm_tb/bus_pss_test.sv(66) @ 2423: uvm_test_top [bus_pss_test] ---------------------------------------
UVM_INFO /mnt/vol_NFS_rh003/estudiantes/mario_montero/trabajo-final-graduacion/src/tb/uvm_tb/bus_pss_test.sv(67) @ 2423: uvm_test_top [bus_pss_test] ----           TEST PASS           ----
UVM_INFO /mnt/vol_NFS_rh003/estudiantes/mario_montero/trabajo-final-graduacion/src/tb/uvm_tb/bus_pss_test.sv(68) @ 2423: uvm_test_top [bus_pss_test] ---------------------------------------
Total coverage 74.07
```

Figura 26: Salida mostrada cuando no se reciben paquetes debido a overflow.

Por otro lado, cuando se ejecuta una prueba, se puede visualizar también la duración y el nombre de las acciones que se ejecutan en cada prueba, como se ve en la Figura 27.

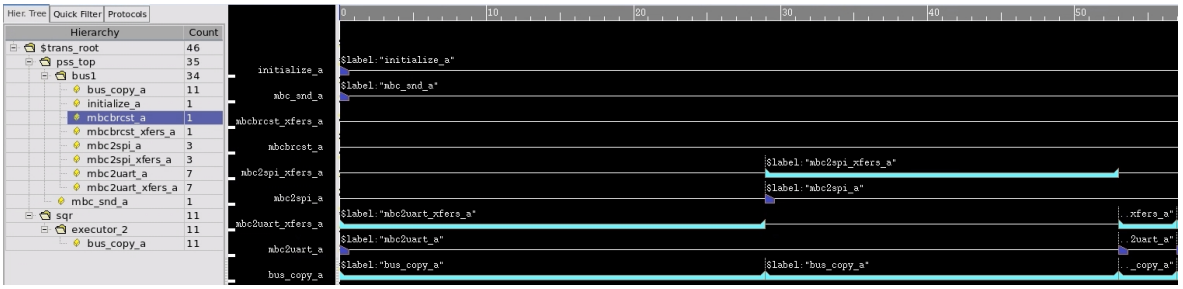


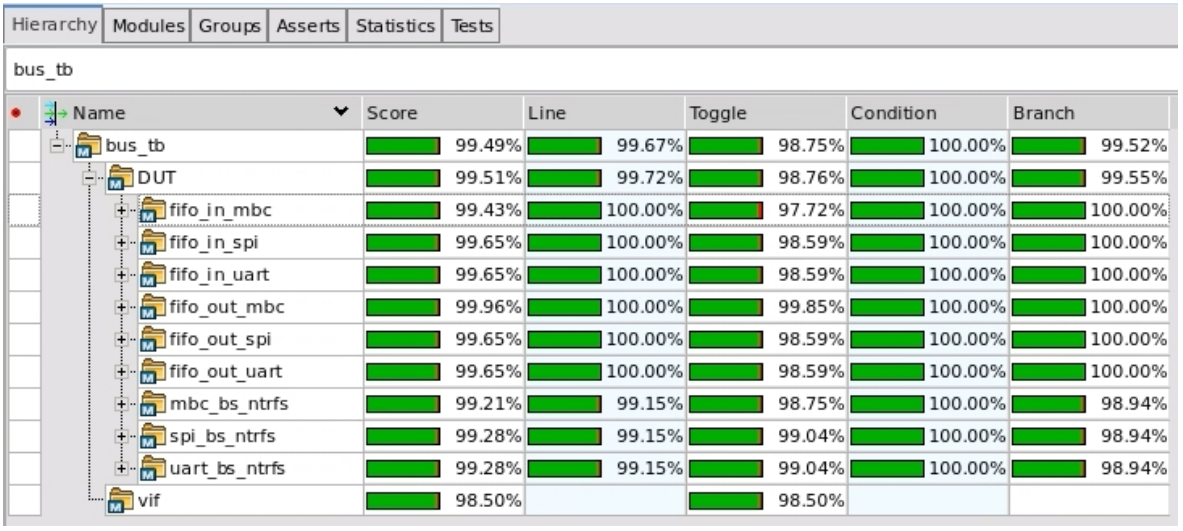
Figura 27: Salida del analizador de acciones.

Con respecto a la cobertura, se muestra en las Figuras 28 y 29 los porcentajes de la de tipo funcional y estructural, respectivamente.

Avg. Group Score:100.00% U+C:34 U:0 C:34 X:0
Avg. Group Inst. Score:100.00% U+C:34 U:0 C:34 X:0

Group	Score	U+C	U	C	X	Goal	Weight
CG bus_env_pkg::bus_coverage::cg_bus_xfer	100.00%	16	0	16	0	100%	1
CP cp_bus_src	100.00%	3	0	3	0	100%	1
CP cp_bus_target	100.00%	4	0	4	0	100%	1
CR cc_bus_xfer	100.00%	9	0	9	0	100%	1
CG bus_tb.vif::cg_bus_sig	100.00%	18	0	18	0	100%	1
CP cp_bus_pending_mbc	100.00%	2	0	2	0	100%	1
CP cp_bus_pending_spi	100.00%	2	0	2	0	100%	1
CP cp_bus_pending_uart	100.00%	2	0	2	0	100%	1
CP cp_bus_pop_mbc	100.00%	2	0	2	0	100%	1
CP cp_bus_pop_spi	100.00%	2	0	2	0	100%	1
CP cp_bus_pop_uart	100.00%	2	0	2	0	100%	1
CP cp_bus_push_mbc	100.00%	2	0	2	0	100%	1
CP cp_bus_push_spi	100.00%	2	0	2	0	100%	1
CP cp_bus_push_uart	100.00%	2	0	2	0	100%	1

Figura 28: Porcentaje de cobertura funcional luego de aplicar la regresión.



Hierarchy | Modules | Groups | Asserts | Statistics | Tests

bus_tb

Name	Score	Line	Toggle	Condition	Branch
bus_tb	99.49%	99.67%	98.75%	100.00%	99.52%
DUT	99.51%	99.72%	98.76%	100.00%	99.55%
fifo_in_mbc	99.43%	100.00%	97.72%	100.00%	100.00%
fifo_in_spi	99.65%	100.00%	98.59%	100.00%	100.00%
fifo_in_uart	99.65%	100.00%	98.59%	100.00%	100.00%
fifo_out_mbc	99.96%	100.00%	99.85%	100.00%	100.00%
fifo_out_spi	99.65%	100.00%	98.59%	100.00%	100.00%
fifo_out_uart	99.65%	100.00%	98.59%	100.00%	100.00%
mbc_bs_ntrfs	99.21%	99.15%	98.75%	100.00%	98.94%
spi_bs_ntrfs	99.28%	99.15%	99.04%	100.00%	98.94%
uart_bs_ntrfs	99.28%	99.15%	99.04%	100.00%	98.94%
vif	98.50%		98.50%		

Figura 29: Porcentaje de cobertura estructural luego de aplicar la regresión.

En la Figura 30 se muestra una tabla de cobertura generada a partir del URG report.

```
Dashboard

Date: Wed Sep 25 10:37:42 2024

User: Montero_Marenco_I_2024_vlsi

Version: U-2023.03-SP2
Command line: urg -full64 -dir simv.vdb -format text

Number of tests: 1

------------------------------------------------------------------------------
Total Coverage Summary
SCORE  LINE   COND   TOGGLE BRANCH GROUP
 52.47  46.80  48.40  35.54  43.20  88.43


------------------------------------------------------------------------------
Hierarchical coverage data for top-level instances
SCORE  LINE   COND   TOGGLE BRANCH NAME
 21.45  70.59  15.22   0.00   0.00 Arbiter
 33.31  42.93  45.79   0.00  44.52 bs_ntrfs
 32.04  41.40  43.73   0.00  43.04 bs_ntrfs_n_rbtr
 96.25  99.46  97.92  88.51  99.10 bus_tb
  0.00   0.00   0.00   0.00   0.00 fifo_flops
 20.34  48.95   0.00   0.00  32.41 fifo_ltch
  1.50   3.00   0.00   0.00   3.00 fifo_ltch_no_rst
 27.04  50.00  11.11   0.00  47.06 mem_latch
  2.94   4.53   5.28   0.00   1.95 ntrpt_cam_fifo
  1.24   2.95   2.00   0.00   0.00 prll_bs_ntrfs
  0.00   0.00 --       0.00   0.00 prll_rgstr


------------------------------------------------------------------------------
Total Module Definition Coverage Summary
SCORE  LINE   COND   TOGGLE BRANCH
 34.24  33.97  21.47  55.89  25.65


------------------------------------------------------------------------------
Total Groups Coverage Summary
SCORE  WEIGHT
 88.43 1
```

Figura 30: Tabla de cobertura de la prueba `random_xfers_a_2`.

Por último, en el siguiente listing se muestran los resultados de ejecutar las 25 pruebas. Se puede ver que el ambiente reporta tanto el nombre de la prueba PSS, si la prueba

falló o no y el porcentaje de cobertura funcional para los dos grupos antes definidos.

```
Test name: mbc_snd_a_1
PASS
Total coverage 62.50
--------------------------------------------------------------------------------

Test name: mbc_snd_a_2
PASS
Total coverage 62.50
--------------------------------------------------------------------------------

Test name: mbc_snd_a_3
PASS
Total coverage 62.50
--------------------------------------------------------------------------------

Test name: spi_snd_a_1
PASS
Total coverage 62.50
--------------------------------------------------------------------------------

Test name: spi_snd_a_2
PASS
Total coverage 62.50
--------------------------------------------------------------------------------

Test name: spi_snd_a_3
PASS
Total coverage 62.50
--------------------------------------------------------------------------------

Test name: uart_snd_a_1
PASS
Total coverage 62.50
--------------------------------------------------------------------------------

Test name: uart_snd_a_2
PASS
Total coverage 62.50
--------------------------------------------------------------------------------

Test name: uart_snd_a_3
PASS
Total coverage 62.50
--------------------------------------------------------------------------------

Test name: solo_brcst_a_1
PASS
```

```
48 Total coverage 76.39
49 ----------------------------------------------------------------
50
51 Test name: solo_brcst_a_2
52 PASS
53 Total coverage 76.39
54 ----------------------------------------------------------------
55
56 Test name: solo_brcst_a_3
57 PASS
58 Total coverage 76.39
59 ----------------------------------------------------------------
60
61 Test name: random_xfers_a_1
62 PASS
63 Total coverage 98.15
64 ----------------------------------------------------------------
65
66 Test name: random_xfers_a_2
67 PASS
68 Total coverage 88.43
69 ----------------------------------------------------------------
70
71 Test name: random_xfers_a_3
72 PASS
73 Total coverage 94.44
74 ----------------------------------------------------------------
75
76 Test name: solo_a_mbc_a_1
77 PASS
78 Total coverage 66.67
79 ----------------------------------------------------------------
80
81 Test name: solo_a_mbc_a_2
82 PASS
83 Total coverage 74.07
84 ----------------------------------------------------------------
85
86 Test name: solo_a_mbc_a_3
87 PASS
88 Total coverage 74.07
89 ----------------------------------------------------------------
90
91 Test name: solo_a_spi_a_1
92 PASS
93 Total coverage 66.67
94 ----------------------------------------------------------------
95
96 Test name: solo_a_spi_a_2
```

```
97  PASS
98  Total coverage 74.07
99  ------------------------------------------------------------------------------
100
101 Test name: solo_a_spi_a_3
102 PASS
103 Total coverage 74.07
104 ------------------------------------------------------------------------------
105
106 Test name: solo_a_uart_a_1
107 PASS
108 Total coverage 66.67
109 ------------------------------------------------------------------------------
110
111 Test name: solo_a_uart_a_2
112 PASS
113 Total coverage 74.07
114 ------------------------------------------------------------------------------
115
116 Test name: solo_a_uart_a_3
117 PASS
118 Total coverage 74.07
119 ------------------------------------------------------------------------------
120
121 Test name: todas_a_todas_a_1
122 PASS
123 Total coverage 100.00
124 ------------------------------------------------------------------------------
```

8. Ruta al directorio del proyecto

`/mnt/vol_NFS_rh003/estudiantes/mario_montero/trabajo-final-graduacion`

Referencias

[1] T. Timi. (s.f.) An overview of uvm end-of-test mechanisms. Accedido: 27-Sept-2024. [Online]. Available: https://blog.verificationgentleman.com/2016/03/25/an-overview-of-uvm-end-of-test-mechanisms.html

[2] *Verification Continuum Portable Stimulus (VC PS) User Guide*, Synopsys, Jun. 2023.