RC-Mock: Mocking Framework para módulos hardware generados mediante HLS

Julián Caba[1], Fernando Rincón, Julio Daniel Dondo,
Jesús Barba, Manuel Abaldea y Juan Carlos López

Resumen— **La fase de verificación es una de las fases más importantes dentro del ciclo de vida de un producto, debido a que de ella depende en gran medida el *time-to-market* del mismo. En los últimos años el flujo de diseño hardware para sistemas basados en FPGAs (Field-Programmable Gate Arrays) ha evolucionado notablemente, permitiendo el uso de lenguajes de alto nivel para la descripción de aceleradores hardware. Este avance ha permitido disminuir el esfuerzo realizado por los desarrolladores a la hora de implementar sus diseños. Sin embargo, la etapa de verificación continúa siendo una tarea compleja de abordar, principalmente debido a que en la mayoría de entornos de verificación es necesario incluir componentes de terceros.**

Este trabajo propone como solución el uso de dobles de prueba propuesto en las metodologías de desarrollo ágil en un entorno puramente hardware, utilizando un dispositivo físico como plataforma de verificación hardware. El framework de mocks propuesto en este trabajo, RC-Mock, permite reemplazar la funcionalidad de terceros dentro de un diseño HLS (High-Level Synthesis) por un componente que imita el comportamiento real de ese tercero, eliminando del diseño original las dependencias. Esta verificación será realizada en un dispositivo físico con el fin de eliminar inexactitudes que introducen las herramientas HLS y reducir el tiempo que introduce la co-simulación. Sobre el dispositivo lógico reconfigurable o FPGA se ha diseñado una plataforma de verificación hardware disponible remotamente permitiendo el despliegue de diseños basados en FPGA y su posterior verificación con un framework de testing.

Palabras clave— **verificación, Field-Programmable Gate Array, Test-Driven Development, dobles de prueba**

I. Introducción

La continua evolución de los sistemas en chip (SoC) requiere de nuevas herramientas de diseño con el fin de reducir la brecha entre diseño y capacidades tecnológicas ofrecidas por estos tipos de dispositivos, así como disminuir la complejidad del proceso de diseño de hardware [1]. Un ejemplo de estas nuevas herramientas son las denominadas herramientas de Síntesis de Alto Nivel o HLS, que permiten a los ingenieros acelerar sus aplicaciones o construir sus propios componentes utilizando lenguajes de programación de alto nivel (HLL), como pueden ser C, C++ o SystemC. El uso de estas herramientas y de los HLL están cada vez más extendidos en el diseño hardware, permitiendo trabajar en un nivel de abstracción más alto siempre que los requisitos lo permitan y no sea necesario un diseño manual de bajo nivel. Por lo tanto, el uso de estas herramientas permite a los ingenieros construir módulos hardware a partir de algoritmos software y realizar una exploración del espacio de diseño en un tiempo reducido, disminuyendo el *time-to-market* del producto final [2].

[1]Universidad de Castilla-La Mancha, 13071 Ciudad Real, España, email: julian.caba@uclm.es

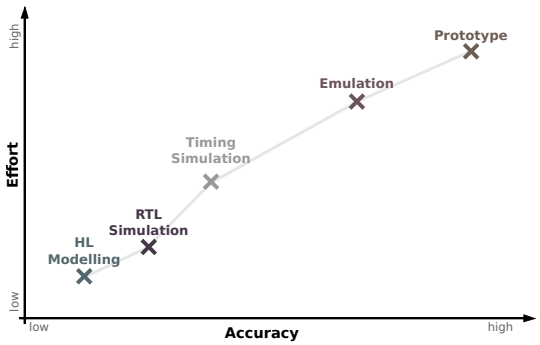

Fig. 1: Relación entre esfuerzo y precisión referente a la verificación

Desafortunadamente, y a pesar de los avances en el diseño digital hardware, aún persiste una significativa brecha entre las capacidades tecnológicas y las necesidades de verificación. Los últimos estudios afirman que la verificación de hardware es el cuello de botella en la mayoría de los proyectos, llegando a alcanzar hasta el 70 % del tiempo de desarrollo [3]. Esta situación se agrava por el uso de las herramientas HLS debido a las limitaciones que estas presentan y que aún no han sido resueltas, como es la inclusión de un dispositivo físico para completar el flujo de diseño hardware. Incluir un dispositivo físico proporciona una serie de ventajas como puede ser la precisión de resultados o la reducción del tiempo de verificación frente al tiempo que requiere una co-simulación [4]. La parte negativa de introducir un dispositivo físico como parte del proceso de verificación es el aumento de la complejidad o esfuerzo que requiere. La Figura 1 ilustra la relación entre esfuerzo y precisión de la verificación hardware de acuerdo con el nivel de abstracción. En la gráfica se puede observar que el menor esfuerzo de verificación se obtiene utilizando modelos de alto nivel, pero que normalmente conllevan inexactitudes. En el lado contrario, se puede observar que el uso de un dispositivo físico es el entorno perfecto para el proceso de verificación, pero esta técnica requiere mayores esfuerzos por parte de los desarrolladores [5].

Otro aspecto a destacar son las dependencias con terceros, como pueden ser el uso de una memoria, un componente no disponible, etc. Estas dependencias son un nuevo reto para los ingenieros de pruebas, puesto que deben ser incluidas en sus escenarios de verificación con el fin de comprobar el comportamiento del módulo en producción. Una solución propuesta

en el mundo software es el uso de dobles de prueba, de forma que aquellas partes de terceros son sustituidas por componentes que simulan el comportamiento real de terceros. Posteriormente, los dobles de prueba pueden ser interrogados para obtener información sobre el proceso de verificación. Sin embargo, sustituir el comportamiento de estos terceros en hardware no es nada sencillo y menos aún si se introduce un dispositivo físico como plataforma de verificación.

En este trabajo, se propone un framework de mocks hardware como ampliación del trabajo previo realizado en [4], el cual se basa en técnicas de testing software para módulos hardware generados con herramientas HLS. Nuestra propuesta considera como parte del proceso de verificación el despliegue del módulo en producción en un dispositivo físico para eliminar inexactitudes introducidas por las herramientas HLS. Por lo tanto, proponemos una solución a los desafíos relacionados con la validación de sistemas, eliminando las dependencias de terceros que presentan algunos diseños con el fin de construir un entorno de verificación con el menor número de entidades reales (dependencias). Para lograr nuestro objetivo principal, nuestra propuesta proporciona los siguientes aspectos:

- Un nuevo framework de mocks hardware, RC-Mock, capaz de crear dobles de prueba hardware, facilitando el proceso de verificación sobre dispositivos físicos.
- Una plataforma física de verificación para manejar el propio proceso de verificación, permitiendo su configuración desde el propio test.
- Una comunicación transparente entre el test y el módulo en producción y entre el test y el doble de prueba, es decir, desacoplar el framework propuesto del diseño bajo prueba o DUT (Design Under Test).

El documento está organizado de la siguiente forma. La segunda sección introduce las características de los test unitarios y define los dobles de prueba, listando sus tipos y justificando la elección de uno de ellos. La sección tercera muestra nuestra visión de componente hardware junto con la propuesta de doble de prueba hardware aplicando esa visión de componente hardware. La cuarta sección describe la plataforma de verificación hardware diseñada y sobre la que se ha desplegado el caso de estudio presentado en la quinta sección. Finalmente, la última sección resume este documento y propone las líneas futuras de este trabajo.

II. Dobles de prueba

Una propuesta para comprobar la validez de los diseños implementados por los ingenieros es el uso o definición de pruebas unitarias junto con otros tipos de pruebas. Las pruebas unitarias se relacionan con la técnica de comprobar cierta funcionalidad de nuestro diseño, esa funcionalidad generalmente se denomina *unidad* puesto que representa a la unidad mínima que puede ser probada, en software la unidad mínima es un método, mientras que en hardware se puede considerar una unidad funcional (*Functional Unit, FU*) como la unidad mínima a probar. La validez de la unidad se determina de acuerdo a la comparación entre el resultado esperado y el resultado producido por la ésta tras inyectarle un conjunto de estímulos. Estas pruebas unitarias deben cumplir varias características que se recogen en la conocida regla FIRST [6]: **F**ast, las pruebas deben ser lo suficientemente rápidas para no convertirse en un cuello de botella; **I**ndependent, una prueba no debe depender de otras pruebas; **R**epeatable, la prueba debe devolver los mismos resultados independientemente de las veces que se ejecute; **S**elf-validation, la prueba debe devolver un resultado booleano (pasar o fallar) sin ninguna interpretación subjetiva y; **T**imely, las pruebas unitarias deberían ser escritas justo antes del código de producción. A través de esta técnica y del uso de framework de testing el desarrollador puede identificar defectos en el diseño. Este trabajo hace uso de un framework de testing hardware, RC-Unity, desarrollado previamente en [4].

Por lo general, la mayoría de diseños, tanto software como hardware, presentan dependencias con otras entidades, lo que dificulta el proceso de test del diseño bajo prueba. Esta dificultad viene dada debido a que esas entidades de terceros no deben ser utilizadas en entorno de verificación con el fin de facilitar el proceso de verificación sin necesidad de construir todo el sistema, ya que es muy posible que esa entidad de terceros dependa a su vez de otra convirtiendo la verificación de un módulo en la construcción del sistema completo, algo que no es deseable. Por lo tanto, lo deseable es instanciar lo mínimo posible del sistema final, y para conseguir este objetivo se puede hacer uso de los dobles de prueba.

Un doble de prueba es una entidad capaz de simular la interfaz que un determinado componente ofrece a otros componentes, pero que realmente no implementa la funcionalidad del tercero, es decir, un doble imita el comportamiento del tercero haciendo creer al DUT que está interactuando con la entidad del tercero. Además, el doble tiene una serie de utilidades que permiten comprobar aquello que utilizó el DUT cuando invocó al tercero. Existen diferentes tipos de dobles de pruebas, los cuales son clasificados según su comportamiento [7] [8]:

- **Stub**. Reemplaza un objeto real por uno específico para alimentar al DUT mediante unos valores predefinidos, es decir retorna los valores configurados por el test cuando es invocado. Normalmente no responde a nada fuera de lo programado.
- **Fake**. Reemplaza un componente del que depende el DUT con una implementación mucho más ligera, pero funcionalmente equivalente. Por lo general son implementados de tal forma que no los hacen no aptos para el código final.
- **Mock**. Sustituye un objeto del que depende el DUT por un objeto específico que verifica si el DUT lo está utilizando correctamente. Están

programados con una serie de expectativas que deberían cumplirse durante la ejecución de la prueba. Si no se cumplen esas expectativas la prueba deberá fallar.

- **Spy**. Como su propio nombre indica, captura las llamadas realizadas por el DUT para su posterior verificación. Esa verificación se comprueba mediante aserciones para verificar que ciertas llamadas se realizaron. A diferencia del *mock*, no se comprueba la secuencialidad de las llamadas incluso pueden existir más llamadas de las esperadas, se podría decir que el *spy* es más benevolente.

Entre los diferentes tipos de dobles de prueba, el más interesante para el dominio hardware es el tipo **mock** debido a su comportamiento, ya que este tipo permite inspeccionar las invocaciones realizadas por el DUT, proporcionando la información necesaria para conocer si la comunicación con entidades de terceros es correcta, aspecto que el resto de dobles de prueba no permite.

III. RC-Mock framework: Dobles de prueba hardware

Para comprender la propuesta realizada en torno a los dobles de prueba hardware es necesario exponer previamente nuestra visión de componente hardware y como pueden ser modelados mediante herramientas HLS.

A. Encapsulado del hardware con HLS

Desde el punto de vista del progreso del diseño electrónico, el uso de lenguajes de programación de alto nivel junto con las herramientas HLS abre un gran abanico de posibilidades permitiendo adoptar técnicas o metodologías ampliamente aceptadas en la industria. Una de estas técnicas es el Paradigma Orientado a Objetos (POO), la cual aporta abstracción, encapsulado, modularidad, robustez entre otros a los sistemas desarrollados.

Desde el punto de vista del POO, un componente hardware puede modelarse como un conjunto de funciones, a ello le denominamos objeto hardware, por el alto grado de similitud con respecto a los objetos software. La diferencia radica en la necesidad de definir un único punto de entrada en el caso del diseño electrónico digital. Por otro lado, si el módulo que se está desarrollando ofrece varias operaciones cada una de esas operaciones será definida en una función distinta y deberá garantizarse su acceso por un tercero.

La solución propuesta consiste en definir una función *top* que encapsula el código del componente junto con un módulo denominado *facade* capaz de redirigir la entrada de datos a la función correspondiente. Mientras que la salida de cada una de las funciones que componen el módulo es redirigida a un componente denominado *collector* encargado de enviar el resultado. Por lo tanto, los módulos *facade* y *collector* serán los encargados de establecer el camino por el que deben fluir los datos con el objetivo de establecer un único punto de entrada y salida al objeto hardware. De esta forma, el objeto hardware puede recibir los datos de forma serializada y enviarlos de la misma forma (ver Figura 2).

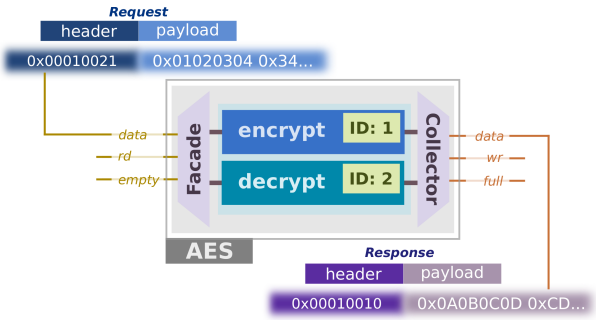

Fig. 2: Objeto hardware (AES)

Sin embargo, en este punto no hay forma de conocer qué función debe ser ejecutada cuando los datos son recibidos por el componente. La solución se basa en incluir un direccionamiento lógico que se enviará a través del canal de datos como si de una cabecera de un mensaje de red se tratase. Esta cabecera incluirá dos palabras de 16-bits que representan a un identificador único y al tamaño de datos enviados expresado en palabras de 32-bits, que a su vez corresponda al ancho del canal de entrada del objeto hardware. Además, cada una de las funciones del objeto hardware tendrá asignado de antemano un identificador único. Por consiguiente, el mensaje recibido por el objeto hardware será atendido por el módulo *facade* que extraerá los 32-bits correspondientes a la cabecera para determinar la función que debe ser ejecutada, a la vez dispondrá de la cantidad de palabras de 32-bits que debe transferir a dicha función. En caso de recibir un identificador no conocido, el objeto hardware rechazará el mensaje completamente, es decir leerá tantas palabras de 32-bits como indique el campo tamaño de la cabecera.

La Figura 2 muestra la estructura interna del objeto hardware correspondiente al algoritmo AES de 128 bits cuya funcionalidad queda definida por los métodos `encrypt` y `decrypt`, correspondientes a las operaciones de cifrar y descifrar, respectivamente. La interfaz del objeto hardware consta de dos canales comunicación, uno para los datos de entrada y otro para los datos de salida. Las señales de estos canales corresponden a los de una FIFO cuyo ancho de palabra es 32-bits. El funcionamiento del objeto AES se basa en tres sencillos pasos. En primer lugar se debe identificar la función a ejecutar (`encrypt` o `decrypt`), para ello el módulo *facade* obtiene una palabra de 32-bits del canal de entrada (cabecera de datos). En este instante se conoce qué función debe ser llamada y la cantidad de palabras de 32-bits que se deben leer del canal de entrada y enviar al módulo en cuestión que implementa la operación solicitada. Previamente a ese envío el módulo *facade* realizará un *casting hardware* para adaptar los datos serializados recibidos por el canal de entrada a la entrada del módulo en cuestión. Una vez que el resultado se en-

cuentra disponible, el *collector* construirá el mensaje de retorno incluyendo los propios datos del resultado junto con una pequeña cabecera de 32-bits al inicio del mensaje, que consta del identificador de la función y de la cantidad de palabras de 32-bits retornadas. Como se ha comentado, los datos enviados al módulo solicitado deben ser serializados en palabras del mismo tamaño que el canal de entrada, que corresponde a 32-bits. En los casos que la cantidad de datos enviados hacia o desde el módulo no completen palabras de 32-bits, se introducirá el padding necesario para corregir el tamaño de datos recibido o enviado.

Como se ha comentado anteriormente, la interfaz del objeto hardware se basa en señales FIFO, que es el uso más común a la hora de diseñar aceleradores hardware mediante herramientas HLS. Esta interfaz puede ser adaptada a un bus estándar como es el bus AXI de AMBA. Para ello, se debe diseñar un driver de comunicaciones con la funcionalidad suficiente para interpretar el protocolo de bus y traducirlo a los canales de comunicación propuestos (doble interfaz basada en FIFO).

B. RC-Mock objects

RC-Mock framework sigue el mismo enfoque que otros frameworks: ofrecer una funcionalidad extra para expresar con qué frecuencia, con qué argumentos y en qué momento fue invocada una operación o método mockeado, incluso retornar respuestas a esas llamadas con información almacenada previamente, es decir el objetivo es imitar una parte de la funcionalidad. Todo el código debe ser generado antes del proceso de síntesis, debido a que a los componentes de hardware no ofrece manipulación de código en tiempo de ejecución, pero su configuración si es posible realizarla posteriormente. Los dobles de prueba hardware se generan a partir de los ficheros de cabecera que contienen los métodos a mockear. El proceso para generar la funcionalidad del mock se realiza en dos pasos:

- El parser lee los archivos fuente del objeto a mockear y construye el árbol de sintaxis (AST). Este proceso se realiza mediante Clang, que se trata de un front-end que utiliza LLVM como back-end [9].
- A partir del AST se construye el objeto o función mockeada haciendo uso de una serie de plantillas predefinidas mediante Ctemplate que es un sencillo pero potente lenguaje de plantillas para C++ desarrollado por Google.

El resultado es un nuevo fichero de cabecera y archivos fuente con código ANSI C sintetizable que contiene la funcionalidad extra para configurar el doble de prueba y para posteriormente interrogar qué sucedió durante la ejecución de un test (funciones mock), también incluye la funcionalidad necesaria para imitar los métodos reales que han sido mockeados. Todo ello siguiendo la filosofía de objetos hardware presentada en la sección anterior

```
#ifndef RD_MEM_MOCK
#define RD_MEM_MOCK

#include <hls_stream.h>

typedef unsigned int tbus;

struct struct_readMem_PARAM
{
  unsigned int addr;
};
typedef struct_readMem_PARAM readMem_PARAM;


struct struct_readMem_FAIL
{
  unsigned int _callCount;
  unsigned int _time;
  readMem_PARAM _actual;
  readMem_PARAM _expect;
};
typedef struct_readMem_FAIL readMem_FAIL;


unsigned int readMem(unsigned int addr);
void readMem_CallCount(hls::stream<tbus> &dout,
             unsigned int &readMem_callCount);
void readMem_FailureCount(hls::stream<tbus> &dout,
             unsigned int &readMem_failureCount);
void readMem_callTimes(hls::stream<tbus> &dout,
             unsigned int &readMem_callTime);
void readMem_Expect(hls::stream<tbus> &dout,
             hls::stream<treadMem_PARAM> &readMem_expect);
void readMem_failures(hls::stream<tbus> &dout,
             hls::stream<treadMem_FAIL> &readMem_failures);

#endif
```

Fig. 3: Fichero de cabecera de la función mockeada *readMem*

(ver Sección III-A). La Figura 3 muestra el fichero de cabecera obtenido a partir de la signatura `unsigned int readMem(unsigned int addr)`, que abstrae una operación de lectura en memoria.

En primer lugar, el generador define dos estructuras para cada método mockeado; La primera estructura contiene los parámetros del método específico, en el ejemplo struct_readMem_PARAM está formado por la dirección de memoria a leer; la segunda estructura tiene como objetivo definir la información almacenada en caso de encontrar un error, es decir si los datos de la configuración del mock no corresponden con los datos recibidos por el diseño bajo prueba, en el ejemplo se comprobarán si la dirección de lectura es la esperada. Finalmente, para gestionar la información adicional, el generador define varias *funciones mock* que responden a las peticiones realizadas por las pruebas unitarias, proporcionando la información almacenada en el doble de prueba. Por ejemplo, el método `readMem_CallCount` devuelve el número de veces que fue invocada la función mockeada `readMem`. Por lo tanto, el doble de prueba dispondrá de un conjunto de variables internas donde se almacenará la información de lo que está sucediendo, así como de la configuración establecida por el test antes de estimular el diseño bajo prueba. Las variables con las que trabaja el doble de prueba son las siguientes:

- *callCount* Contador para el número de llamadas realizadas.
- *failCount* Contador para el número de fallos detectados.

- *return_values* FIFO que contiene los valores de retorno del método mockeado.
- *expected_values* FIFO que contiene los valores esperados por el método mockeado.
- *timestamp* FIFO que contiene la marca de tiempo en el que fue invocado el método mockeado.
- *failures* FIFO que contiene una traza de los fallos detectados.

Las funciones mock que contiene el doble de prueba dependerá del tipo de función mockeada. La Tabla I muestra las posibles combinaciones dependiendo del método a mockear. Estas funciones permitirán acceder a las variables internas que componen el doble de prueba, bien para rescatar información de ellas o bien para el caso de las variables `return_values` y `expected_values` almacenar los valores de retorno y esperados, respectivamente.

- **void func_return(PARAM)**. Esta función almacena los valores de retorno en la variable `return_values`. Cuando la función mockeada intenta leer datos de esta variable y no contiene ninguno, se repetirá el valor del último enviado.
- **void func_expect(PARAM)**. Esta función almacena los valores esperados en la variable `expected_values`. Su principal función es dotar a la función mockeada los valores esperados para posteriormente comprobar su similitud con el valor(es) recibido(s) como parámetro(s).
- **int func_callCount()**. Esta función retorna el número actual de veces que fue llamada la función mockeada, es decir, devuelve el valor de la variable *callCount*.
- **int func_failCount()**. Esta función devuelve el número de fallos encontrados, es decir, devuelve el valor de la variable *failCount*, que coincide con la cantidad de items almacenados en la variable *failures*.
- **int func_timestamp()**. Esta función devuelve una marca de tiempo correspondiente a las llamadas realizadas por el DUT a la función mockeada. Para obtener todas las marcas de tiempo se deberán realizar tantas llamadas a este método como indique la variable *callCount*. La información de tiempos es obtenida en el mismo orden que la función mockeada fue invocada.
- **FAIL func_failures()**. Esta función retorna las trazas de errores encontrados por la función mockeada. Para obtener todas las trazas se deberán realizar tantas llamadas a este método como indique la variable *failCount*. La información de los errores es rescatada en el mismo orden en el que fueron detectados.

Desde el punto de vista estructural la función mockeada del ejemplo quedaría representada en la Figura 4, la cual representa su diagrama de bloques. Este diagrama de bloques contiene las funciones mock listadas anteriormente y las variables responsables de almacenar la información necesaria para imitar la funcionalidad del tercero (*return* y *expect*), junto con las variables que almacenan información del comportamiento observado tras la ejecución de un test.

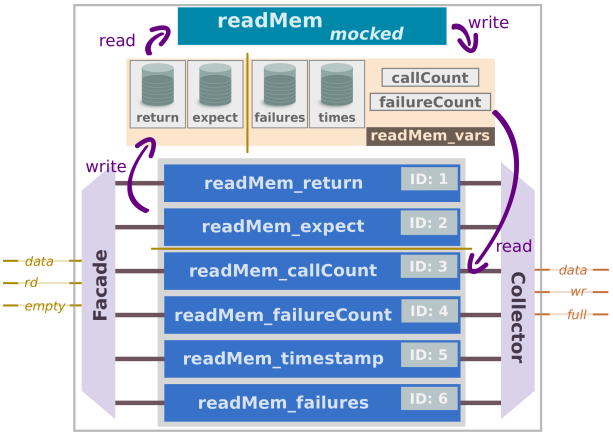

Fig. 4: Diagrama de bloques de *readMem*

```
unsigned int readMem(unsigned int addr)
{
  treadMem_FAIL auxFail;
  unsigned int _return;
  unsigned int _expect_addr;
  int _diff;
  unsigned int _time;

  _expect_addr = readMem_expect.read().addr;
  _time = timeClock.read();
  _diff = abs(addr - _expect_addr);

  if(_diff > DELTA){
    auxFail._callCount = readMem_callCount;
    auxFail._param.addr = addr;
    auxFail._expect.addr = _expect_addr;
    auxFail._time = _time;
    readMem_failures.write(auxFail);
    readMem_failureCount += 1;
  }
  readMem_timestamp.write(_time);
  _return = readMem_return.read()._return;
  readMem_callCount += 1;

  return _return;
}
```

Fig. 5: Cuerpo de la función mockeada *readMem*

En la parte superior de la figura se sitúa la función mockeada, *readMem*, que desde el punto de vista del comportamiento, la función comienza a leer el valor esperado y la marca de tiempo en el que es invocada dicha función por el diseño bajo prueba. A la vez se compara el valor actual con el esperado, cuyo resultado determinará si debe almacenarse la traza de error o no. Si los valores no coinciden, la función mockeada escribirá la información concerniente al fallo en la variable *failures*. Esta variable permite enviar una traza del fallo encontrado al test, indicando: la invocación que detectó el fallo, la marca de tiempo en que sucedió la invocación y los valores reales y esperados. A continuación, se aumenta el contador de fallos (*failureCount*). Por último, independientemente si se detectó un fallo o no, la función mockeada almacena la marca de tiempo leída en el primer paso, lee los valores de retorno (siempre que la función mockeada deba devolver algo) y aumenta la variable *callCount*. Finalmente, se devuelve el valor o los valores de retorno (ver Figura 5).

TABLA I: Relación entre funciones mockeadas y propiedades

Func. mockeada	**Propiedades Mock** return	expect	callCount	failureCount	timestamp	failures
void foo(void)	-	-	✓	-	✓	-
void foo(args)	-	✓	✓	✓	✓	✓
ret foo(args)	✓	✓	✓	✓	✓	✓

IV. Plataforma de verificación hardware

Uno de los objetivos en los que se centra este trabajo es proporcionar una plataforma de verificación hardware para probar los módulos generados por herramientas HLS y mejorar el análisis que proporcionan. Nuestra plataforma de verificación se basa en una plataforma SoC que integra una FPGA y un procesador embebido en el mismo dispositivo. En nuestro caso, utilizamos la plataforma de prototipado Zedboard de Xilinx.

La elección de esta plataforma se debe a la facilidades que proporciona de comunicación entre el procesador embebido y la parte lógica. Además, la plataforma Zedboard se puede conectar a una red a través de Ethernet con el objetivo de comunicarse con el equipo donde se han desarrollado módulos hardware. El framework de pruebas utilizado, RC-Unity, también es remoto, por lo que la FPGA puede desempeñar el papel de servicio de testing remoto.

A la derecha de la Figura 6 se muestra el diagrama de bloques de la parte lógica de la plataforma de verificación hardware. El módulo a verificar junto con los dobles de prueba están programados en este lado y su comunicación con el procesador embebido se realiza a través de dos canales FIFO conectados a una interfaz AXI.

La parte lógica está dividida en dos partes, una estática que contiene aquellos componentes que no cambian independientemente de la lógica que se verifique, mientras que esta parte, el DUT, se despliega en un área dinámica reconfigurable. Dos de las ventajas del uso de la reconfiguración dinámica parcial (Dynamic Partial Reconfiguration, DPR) es la reducción del tiempo de síntesis [10] y la posibilidad de insertar dinámicamente nuevas funcionalidades sin necesidad de rediseñar todo el sistema o cambiar el sistema desarrollado a un dispositivo más grande. Además, es posible adaptar una FPGA a diferentes escenarios, simplemente modificando la funcionalidad que contienen las zonas dinámicas. EL entono de verificación propuesto se basa en la característica DPR que permite reutilizar el entorno en diferentes proyectos.

El componente *zipFactory* es el encargado de programar (desplegar físicamente) un bitstream parcial en el área dinámica disponible de nuestra plataforma de verificación sin la intervención del procesador embebido. Para ello, recupera los datos del bitstream parcial, que previamente estará almacenado en la DDR. Este componente es capaz de reconocer la palabra de desincronización que contienen los bitstreams para dejar de leer en ese mismo instante. Los datos recuperados se almacenan en un pequeño buffer interno, asegurando la disponibilidad de datos a enviar al ICAP para el despliegue físico. El ancho de banda utilizado en el ICAP es de 32-bits, que coincide con el ancho del buffer interno. En el momento que se envía la palabra de desincronización, el proceso de reconfiguración finaliza, aunque algunos NOPs son enviados al ICAP.

El componente *Hardware Manager* realiza un conjunto de tareas hardware referentes a la verificación (cada una de estas tareas son controladas desde el test) que no serían factibles realizar desde el entorno de pruebas de software. Estas tareas van desde resetear la zona dinámica a configurar la frecuencia a la que trabajará la zona dinámica, es decir el DUT.

A la izquierda de la Figura 6 se muestra la parte del procesador embebido, junto con la comunicación con el equipo del desarrollador. El procesador embebido se trata de un procesador ARM que ejecuta un sistema operativo basado en Ubuntu (distribución Linaro). Para utilizar los componentes internos de la plataforma de verificación, se han desplegado sobre este sistema operativo tres servicios basados en middleware de comunicaciones ZeroC Ice. Por ejemplo, cuando se quiere desplegar un nuevo bitstream parcial, se debe enviar los datos que componen este fichero a la DDR de la FPGA, para ello se utilizará el servicio *transfer* que permite copiar datos a partir de una dirección de memoria. En ese instante se puede utilizar el servicio *dpr* para desplegar la nueva funcionalidad en la área dinámica. Por lo tanto, este trabajo proporciona un mecanismo transparente que permite compartir la plataforma de verificación hardware, aumentando su disponibilidad y accesibilidad. Finalmente, para inyectar estímulos y rescatar información de los dobles de prueba se puede hacer uso del servicio *GCommand* que traduce los mensajes enviados por la red en mensajes AXI cuya dirección hardware coincide con la del DUT.

Como puede observarse, gracias a estos tres servicios es posible hacer uso de la plataforma de verificación hardware de forma remota, haciendo uso de los adaptadores de estos tres servicios que permiten serializar los datos en mensajes ZeroC Ice. Estos servicios se muestran como funciones desde el punto de vista del test.

V. Caso de estudio

Nuestra propuesta ha sido desarrollada bajo un entorno GNU/Linux e implementada en la plataforma Zedboard de Xilinx. Por defecto, la plataforma funciona a 100 MHz, pero el área dinámica y otros componentes pueden modificar esta velocidad de reloj en tiempo de ejecución de acuerdo a las necesida-

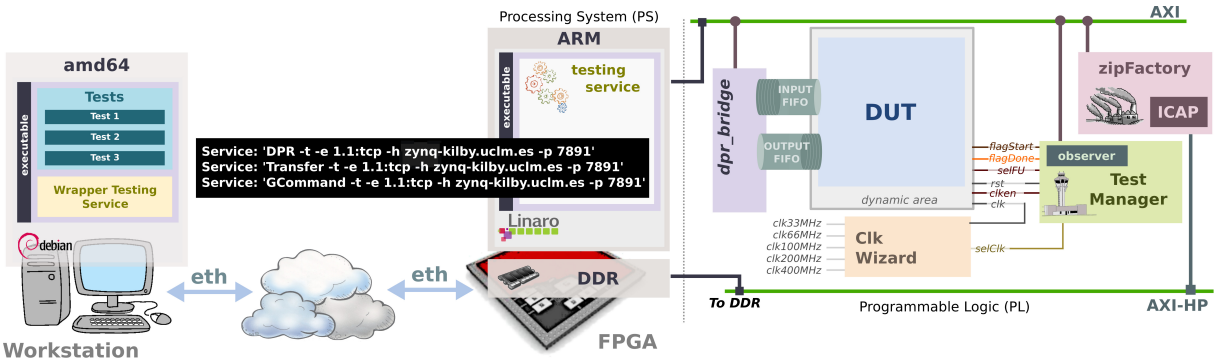

Fig. 6: Plataforma de verificación hardware

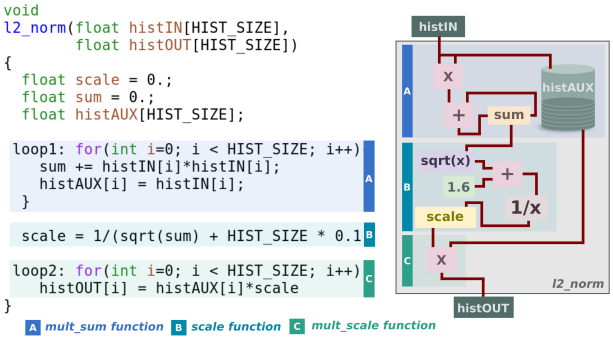

Fig. 7: Factor de normalización l^2-*norm*

des del proyecto. Como caso de estudio de este trabajo hemos seleccionado un algoritmo de descripción de características basado en gradientes (Histogram of Oriented Gradients, HOG). El HOG es un descriptor de características utilizado en visión por computador y en el procesamiento de imágenes para la detección de objetos, especialmente para la detección humana. La implementación del algoritmo se divide en diferentes pasos: cálculo de gradientes, normalización vectorial, entre otros. En este caso de estudio, la etapa elegida corresponde con la normalización vectorial con el factor de normalización l^2-*norm* [11]. La solución a esta etapa ha sido desarrollada en el lenguaje de programación C utilizando la herramienta Vivado HLS en su versión 2016.4 [12].

La Figura 7 muestra una posible solución para la etapa elegida como caso de estudio con el algoritmo l^2-*norm*. Para el caso de estudio presente, el factor de normalización toma como entrada un ventana de píxeles de 4x4 (16 píxeles) y es dividido en tres pasos más pequeños (A, B y C) antes que el resultado sea devuelto a través de *histOUT*, el cual es calculado por el sub-bloque C. Mientras tanto, los sub-bloques A y B calcula la suma de los cuadrados de los componentes del bloque de entrada y el factor de escala respectivamente (ver lado izquierdo de la Figura 7). Siguiendo la filosofía de los objetos hardware presentada en este trabajo, este módulo contendrá un único método que anida otros tres métodos correspondientes a los sub-bloques, que se encuentran interconectados entre sí. Por lo tanto, desde el punto de vista de un usuario que desee utilizar este bloque sólo podrá hacer uso de la única función que engloba al resto, pero ¿qué sucedería si el sub-bloque B dependiera de una tercera entidad? De acuerdo a la propuesta presentada en este trabajo, el sub-bloque B (la función que implementa B) deberá ser mockeada con el objetivo que los sub-bloques A y C puedan ser verificados sin la necesidad de instanciar la dependencia de B.

Por lo tanto, el sub-bloque B quedará definido como un doble de prueba que permitirá inspeccionar la comunicación entre los sub-bloques A y B, mientras que el sub-bloque C será estimulado por el doble de prueba y por lo tanto se podrá introducir aquellos estímulos que se deseen independientemente del resultado aportado por el sub-bloque A, acotando posibles errores producidos por el sub-bloque A, es decir si A devuelve un resultado erróneo no afectará a C, ya que el doble de prueba que imita al sub-bloque B permite detectar fallos en las invocaciones realizadas por A y producir estímulos correctos para el sub-bloque C.

A. Tests con dobles de prueba

Una vez definido el doble de prueba hardware queda por describir cómo se utilizará dicho doble y sus funciones mock para extraer la información almacenada tras estimular el diseño bajo prueba. Para ello, hacemos uso de una representación virtual del doble de prueba y del diseño bajo test. Desde el punto de vista del test estas representaciones virtuales pueden verse como llamadas a funciones que aportan transparencia de localización y comunicación con la implementación final. En la Figura 8 se muestra el caso de prueba para el factor de normalización l^2-*norm*. En primer lugar, antes de utilizar la macro `RCUNITY_SETUP()` y estimular el diseño, se debe configurar el doble de prueba (sub-bloque B o función *scale*). Para estas tareas se hará uso de las funciones mock *scale_expect* y *scale_return* que permiten configurar los valores esperados en una llamada y los valores de retorno, respectivamente. Posteriormente, el diseño bajo prueba es estimulado para comprobar su comportamiento sin una implementación real del sub-bloque B, como se indicó en su configuración se conoce de antemano que el sub-bloque A producirá el resultado 1240.0 que será utilizado como parámetro de entrada para el sub-bloque B, mientras que el sub-bloque B inyectará en el sub-bloque C el valor 0.027164. Se debe recordar que estos valores sólo

```
#include "rc-unity.h"
#include "scale.h"

void
test_caseA(){
#ifdef TIMING
  RCUNITY_SKIP_INPUT(18);
  RCUNITY_SKIP_OUTPUT(1);
#endif

  scale_return(0.027164);
  scale_expect(1240.0);

#ifdef TIMING
  RCUNITY_SETUP();
#endif

  RCUNITY_RESET();
  float out[HIST_SIZE];
  l2norm(input, out);


  for(int i = 0; i != HIST_SIZE; i++)
    TEST_ASSERT_EQUAL_FLOAT(ref[i], out[i]);

  printf("Calls %d\n",scale_callCount());
  printf("Failures %d\n",scale_failureCount());
  scale_print_failures();
}
```

Fig. 8: Test para el algoritmo l^2-*norm*

son válidos para este caso de prueba, para otro caso se debe volver a configurar el doble de prueba desde el test, el componente hardware referente al doble de prueba no variará. Cuando el diseño bajo prueba finaliza su ejecución, se comprobará el resultado producido por éste. Además, se podrá rescatar información del doble de prueba para conocer el comportamiento que tuvo el diseño, permitiendo conocer si el sub-bloque A devolvió un resultado correcto, las veces que fue llamado la función mockeada y los errores que se detectaron durante la ejecución de la prueba, junto con una traza de los errores detectados.

VI. Conclusión y Trabajo Futuro

RC-Mock es un framework para la generación de mocks hardware que aprovecha los avances realizados en el diseño electrónico con herramientas HLS, permitiendo la capacidad de crear objetos hardware y aplicar técnicas de testing ampliamente aceptadas en el mundo industrial. RC-Mock se centra en la última etapa del desarrollo hardware (verificación sobre el dispositivo físico) y complementa el trabajo realizado previamente con el framework hardware RC-Unity. Los dobles de prueba hardware ofrecen una solución clara para sustituir las dependencias de terceros de forma transparente, permitiendo un análisis post-estimulación. Este análisis. en cierta medida, permite realizar una introspección hardware de un diseño, ya que es posible sustituir parte del diseño por un doble de prueba y posteriormente analizar los valores con los que se le estimuló.

El trabajo futuro se centra en aserciones hardware sintetizables, con el fin de mejorar las capacidades de verificación en tiempo real de nuestra plataforma y aumentar la visibilidad de las señales internas, sin perturbar el rendimiento del diseño original o haciéndolo mínimamente.

Agradecimientos

Este trabajo ha sido financiado por el *Ministerio de Economía y Competitividad del Gobierno de España* bajo el proyecto `PLATINO (TEC2017-86722-C4-4-R)` y por la *Consejería de Educación, Cultura y Deportes de la Junta de Comunidades de Castilla-La Mancha* bajo el proyecto `SimbIoT (SBPLY-17-180501-000334)`.

Referencias

[1] A. Canis, J. Choi, B. Fort, R. Lian, Q. Huang, N. Calagar, M. Gort, J. J. Qin, M. Aldham, T. Czajkowski, S. Brown, and J. Anderson, "From software to accelerators with LegUp high-level synthesis," in *2013 International Conference on Compilers, Architecture and Synthesis for Embedded Systems (CASES)*, Sept 2013, pp. 1–9.

[2] J. Cong, B. Liu, S. Neuendorffer, J. Noguera, K. Vissers, and Z. Zhang, "High-Level Synthesis for FPGAs: From Prototyping to Deployment," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 30, no. 4, pp. 473–491, April 2011.

[3] W. Chen, S. Ray, J. Bhadra, M. Abadir, and L. C. Wang, "Challenges and Trends in Modern SoC Design Verification," *IEEE Design Test*, vol. 34, no. 5, pp. 7–22, Oct 2017.

[4] Julián Caba, João M. P. Cardoso, Fernando Rincón, Julio Dondo, and Juan Carlos López, "Rapid prototyping and verification of hardware modules generated using hls," in *Applied Reconfigurable Computing. Architectures, Tools, and Applications*, Nikolaos Voros, Michael Huebner, Georgios Keramidas, Diana Goehringer, Christos Antonopoulos, and Pedro C. Diniz, Eds., Cham, 2018, pp. 446–458, Springer International Publishing.

[5] Lingkan Gong and Oliver Diessel, *Verification Challenges*, chapter 2, pp. 15–40, Springer International Publishing, Cham, 2015.

[6] Robert Cecil Martin, *Agile Software Development: Principles, Patterns, and Practices*, Prentice Hall PTR, Upper Saddle River, NJ, USA, 2003.

[7] F. Moya, C. González, D. Villa, S. Pérez, M.A. Redondo, C. Mora, F.J. Villanueva, and M. García, *Desarrollo de Videojuegos: Técnicas Avanzadas*, Bubok, 2011.

[8] G. Meszaros, "Test double patterns," http://xunitpatterns.com, 2008.

[9] Apple, Inc., "clang: a C language family frontend for LLVM," http://clang.llvm.org/ ; accessed 16-Feb-2018.

[10] W. Lie and W. Feng-yan, "Dynamic partial reconfiguration in fpgas," in *2009 Third International Symposium on Intelligent Information Technology Application*, Nov 2009, vol. 2, pp. 445–448.

[11] N. Dalal and B. Triggs, "Histograms of oriented gradients for human detection," in *2005 IEEE Computer Society Conference on Computer Vision and Pattern Recognition (CVPR'05)*, June 2005, vol. 1, pp. 886–893 vol. 1.

[12] Xilinx, "Vivado Design Suite User Guide: High-Level Synthesis (UG902)," Tech. Rep., Xilinx Inc., 2017.