Proyecto Fin de Carrera

Ingeniería de Telecomunicación

Generación automática de drivers de protocolo en VHDL para verificación

Autora: Maria Julia Graciani Mestre

Tutor: Hipólito Guzmán Miranda

Dpto. Ingeniería Electrónica

Escuela Técnica Superior de Ingeniería

Universidad de Sevilla

Sevilla, 2018

ÍNDICE

1. INTRODUCCIÓN

1.1. Hardware Description Languages (HDL)

Los lenguajes de descripción de hardware (HDL) permiten describir de forma muy directa circuitos electrónicos digitales, permitiendo modelar de forma completa su comportamiento, es decir, la estructura que siguen y la operación que realizan.

El concepto de los HDL puede resultar similar a los lenguajes de programación a nivel software. Sin embargo, tienen dos diferencias fundamentales:

- todas las operaciones son concurrentes, es decir, se ejecutan en paralelo
- existen diferentes tipos de definiciones para una misma función:
  1. Funcional: se asigna un valor determinado de forma constante
  2. Procedimental: los valores se asignan según un procedimiento establecido
  3. Estructural: es una descripción directa de los módulos del circuito. Sería el equivalente al propio esquema del circuito electrónico.

Desde el punto de vista práctico, la gran ventaja que aportan los HDL es que permiten simular la operación que realiza el circuito antes de su implementación, consiguiendo así asegurar que la operación del circuito es correcta antes de su fabricación.

Otra característica fundamental es que permiten la síntesis automática, es decir, convierten la descripción de un lenguaje HDL en una implementación de un circuito electrónico de forma totalmente automática.

Además estos lenguajes son independientes de la tecnología, es decir, que puede ser sintetizado en librerías de distintos vendedores. Así se consigue que un mismo diseño pueda ser útil en componentes tan diferentes como ASICs (Application-Especific Integrated Circuits) o FPGAs (Field-programmable Data Array) con un esfuerzo mínimo.

Existen dos ejemplos claros de HDL:

- VHDL (VHSIC Hardware Description Language): a pesar de tener una sintaxis más compleja, soporta una amplia diversidad de tipos de datos. Permite la

creación y adición de librerías, lo que permite empaquetar piezas de código nuevas o ya existentes y su reutilización en distintos diseños. Su uso es predominante en el diseño de FPGAs.

- Verilog: se caracteriza por su simplicidad, tanto en su sintaxis como en los tipos de datos. Sin embargo, esta sencillez tiene el coste añadido de la reducción de la tipología de datos, por ejemplo, no existe la declaración de componentes. A pesar de ser un código más compacto, tiene un esquema más pobre en lo que a la resolución de la concurrencia se refiere. Este factor resulta crucial en la elección de un HDL. Este es el lenguaje más usado en la actualidad en el diseño de ASICs

La verificación es el proceso con el que se comprueba si la realización de un producto se ajusta a las normas o especificaciones técnicas establecidas, es decir, si se ajusta a los requisitos fijados en el proyecto. En el ambiente de los diseños electrónicos, la verificación se realiza en especial previamente a la depuración en el laboratorio, para comprobar el cumplimiento de la funcionalidad y su rendimiento. Este paso se hace especialmente relevante debido al alto coste, económico y humano, que tiene el análisis en el laboratorio, además del alto riesgo que supone que pueda dilatarse indefinidamente en el tiempo. Así, si se realiza un análisis completo funcional en el simulador, se reducen notablemente los riesgos que surgirían en su análisis en el laboratorio. Además, pueden existir una serie de condiciones que resulten extremadamente difíciles de generar (o de visualizar) en el laboratorio, limitados por los equipos de los que se disponga, y la verificación nos da la posibilidad de estudiarlas.

Sin embargo, mientras las tareas de diseño crecen linealmente con la complejidad, el proceso de verificación crece de forma exponencial, consumiendo cada vez más tiempo y recursos. Los diseños actuales emplean bloques funcionales, procesadores, periféricos, protocolos e interfaces múltiples, por lo que la verificación debe probar cada elemento y su interacción con los demás, en todas las combinaciones y modos de operación posibles. La verificación incluye la realización de un entorno de verificación, crear un testbench, realizar la simulación lógica, analizar los resultados para detectar y aislar los problemas, sincronizar las señales, etc. Por todo ello y debido a la complejidad de los sistemas actuales, realizar este proceso a mano se hace prácticamente inviable.

Debido a todo esto, la verificación puede consumir entre el 60% y el 80% del esfuerzo total de desarrollo de un sistema digital, desde el concepto inicial hasta la realización final. Resulta entonces

absolutamente necesario optimizar el proceso, utilizando las mejores herramientas y metodologías. Además, cuanto más se automatice este proceso, mejores resultados obtendremos, ya que evita el error humano y nos permite ejecutar diversos tests ante cualquier cambio que hayamos realizado.

El objetivo final de toda verificación es la creación de un banco de pruebas que realice una verificación funcional, comprobando si se cumplen los objetivos que se propusieron al inicio del proceso: si se han cubierto la totalidad de los escenarios típicos, los casos de error, la totalidad de los protocolos, etc. Por ello, se hace necesario que dicho banco de pruebas cumpla con una serie de capacidades mínimas[2]:

1. Conseguir una total cobertura de código, o la máxima posible dentro de las posibilidades reales. Es decir, procurar la verificación del código fuente en su totalidad. El grado de cobertura de código que se consiga determinará la calidad de las pruebas que se llevan a cabo, además de permitir distinguir las partes del código que ya han sido comprobadas y las que no.
2. El uso de aserciones nos proporciona robustez. Son expresiones que comprueban un comportamiento específico y muestran un mensaje (normalmente de error) si este ocurre. Nos sirven para señalar los comportamientos indeseados e informar de la causa del fallo, para poder subsanarla.
3. La generación automática de los estímulos también es una capacidad a tener en cuenta. En cualquier sistema que se desee verificar se necesita generar los datos que simulen el comportamiento de las entradas al DUT (Device Under Test). El comportamiento de estas entradas debe estar basado en las especificaciones del sistema que se desea verificar, para luego poder describir ese comportamiento en VHDL o Verilog. Sin embargo también es relevante verificar el comportamiento del DUT ante entradas cuyos valores no sean los esperados.
4. Es fundamental el modelado a nivel de transacción (TLM Transaction-Level Modeling), que permite el uso de transacciones en lenguaje de alto nivel. Es necesario comprender muy bien este concepto, por lo que se desarrollará más en profundidad.

1.2. Transaction-Level Modeling (TLM)

El modelado a nivel de transacciones (Transaction-Level Modeling) se basa en abstraer las comunicaciones entre las distintas partes de un sistema. El objetivo que se persigue es desarrollar de forma independiente por un lado la funcionalidad de cada módulo y por otro, las comunicaciones. Esta abstracción se representa mediante transacciones entre módulos. Su uso tiene como finalidad facilitar la transferencia de datos o de secuencias de control entre dos o más componentes del sistema, ocultando detalles de implementación de los medios (canales) por la que la transacción se lleva a cabo. Un ejemplo concreto de una operación contribuirá a entender mejor la mejora que esto supone.

Por ejemplo, en el caso de querer realizar una operación de escritura de un dato concreto en una ubicación determinada, habría que ejecutar un comando parecido a éste:

```
write (data,address)
```

Se trata de una mera descripción de la operación que se quiere realizar, así como de los datos que se quieren transferir y la dirección donde están alojados. Se elabora de una forma muy comprensible e intuitiva. A continuación se muestra una figura del movimiento de pines que esta simple orden provocaría:

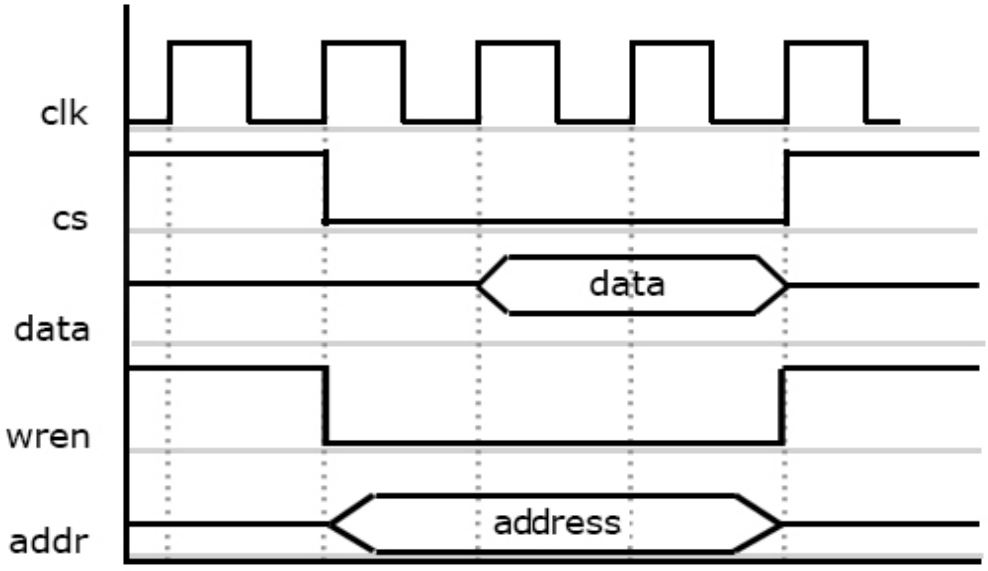


Ilustración 1: Señales generadas por el driver para una sola orden

La imagen demuestra que una orden muy simple produce un movimiento de pines que es necesario generar, y que debe cumplir una serie de requisitos. Además de ser operativo, es fundamental que todos los cambios que se produzcan en las señales generadas estén sincronizados con el flanco activo de reloj. Desde el punto de vista del desarrollador, tener que preocuparse por el sincronismo a nivel de bit de cada señal generada puede resultar tedioso. Por lo que se persigue el objetivo de independizar toda esta complejidad, y automatizarla en la medida de lo posible.

Este nivel de abstracción se ha convertido en un punto de partida en el diseño de sistemas, ya que debido a la creciente complejidad en los sistemas digitales, y las presiones inducidas del tiempo de desarrollo del proyecto, se hace fundamental reducir el esfuerzo de modelado, aumentar la productividad y reducir tiempos. Cuanto más alto sea el nivel en el que los módulos son diseñados, más rápido será el proceso de simulación y más fácil será la conexión entre ambas partes: funcionalidad y comunicación [1].

Todo esto se procura ilustrar en la siguiente figura. La gran ventaja de usar transacciones en los bancos de prueba radica en que nos permite elevar el nivel de abstracción, para que el desarrollador no tenga que preocuparse de todos los detalles de las señales a lo largo del banco de pruebas. Las transacciones nos permiten encapsular nuestra complejidad en 1 o 2 módulos, así el resto del testbench puede seguir un nivel de abstracción mayor.

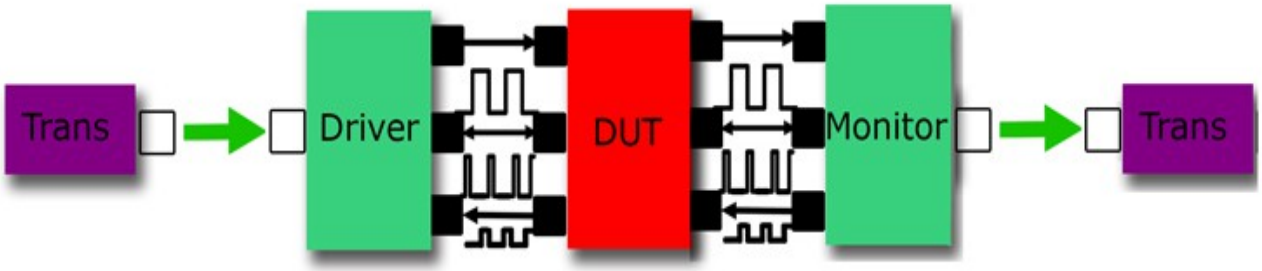


Ilustración 2: Testbench modelado a nivel de transacción

Al ejecutar un test usando un testbench de alto nivel usando transacciones, el driver es el encargado de recibir la transacción y traducirlas de algún modo en señales. Se consigue así aislar toda la complejidad de crear esas señales en un sólo módulo. Cuando el DUT responde ante estas

señales, otro módulo denominado monitor puede ver estas señales y convertirlas en una transacción. El resultado es que toda la complejidad de verificar el comportamiento del nivel de señal del DUT estudiado se mantiene en el driver y en el monitor, y el resto del testbench queda libre de ello, usando construcciones de más alto nivel como las transacciones.[2] Evidentemente los módulos del driver y el monitor estarán generados en un HDL, ya sea VHDL o Verilog.

Es importante resaltar las ventajas que aporta un testbench modelado a nivel de transacción, ya que facilita enormemente el proceso de creación del propio test, además de incrementar la productividad. También nos permitiría poder crear testbenches que generen su propio test y comparen los resultados obtenidos con los esperados, realizando un proceso completo de autoverificación.

2. ANTECEDENTES

Debido a la gran relevancia que ha ido tomando el proceso de verificación, en los últimos años se han ido desarrollando diversas metodologías. El enfoque principal de todas ellas se basa en definir una arquitectura modular que se adapte convenientemente al testbench y los estímulos. VHDL es un lenguaje idóneo para ello, ya que resulta útil tanto a ingenieros de diseño como de verificación al incluir construcciones de diseño RTL, aserciones y un amplio conjunto de construcciones de verificación. A continuación se describen las metodologías más relevantes escritas en Verilog o SystemVerilog:

– VVM (Verification Methodology Manual): fue la primera colección de metodologías de verificación que tuvo éxito. Creada por Synopsys en 2003, VMM utiliza funciones de lenguaje tales como programación orientada a objetos, aleatorización, restricciones, cobertura funcional... para permitir crear entornos de verificación poderosos. La contribución de VMM fue un factor determinante en la creación de UVM.[3]

– OVM (Open Verification Methodology): fue creada en 2008 por Cadence y Mentor Graphics, basada en la metodología de verificación existentes en ambas compañías. Disponible en código abierto, permite la creación sencilla de pruebas dirigidas o aleatorias utilizando comunicación a nivel de transacción y cobertura funcional. Igual que la anterior, OVM contribuyó significativamente al desarrollo de su sucesora, la UVM.[4]

– UVM (Universal Verification Methodology): fue creada en 2010 a partir de las dos anteriores [5]. Permite la creación de componentes de verificación flexibles y reutilizables, así como la generación de potentes bancos de prueba utilizando la generación de estímulos aleatorios y las metodologías de cobertura funcional. Sus principales ventajas son la mejora de la reutilización del banco de pruebas, así como un código de verificación más versátil. Esta metodología ha tomado tal popularidad que IEEE ha realizado el estándar IEEE 1800-2017, donde se describe la sintaxis y la semántica del lenguaje de verificación. [6]

A partir de esta última metodología, se ha desarrollado un nuevo mercado, la Verification IP (Intellectual Property), que en su mayoría ya no es de código abierto sino con propiedad intelectual y costosas licencias. Los bloques Verification IP (VIP) se insertan en el banco de pruebas para

verificar el funcionamiento de los protocolos y las interfaces, tanto individualmente como de forma combinada. Incluye módulos de verificación absolutamente reutilizables, y de diversa índole: BFM (Bus Functional Modes), generadores de tráfico, monitores de protocolo, y bloques de cobertura funcional, entre otros. Se desarrolla con la idea de otorgar licencias a múltiples proveedores para que los mismo bloque sean usados en diferentes diseños.

La Verification IP es increíblemente útil para los desarrolladores, ya que permite una exploración detallada del sistema diseñado. Este factor se está volviendo cada vez más importante debido al crecimiento en la complejidad de los diseños de sistema en chip (SoC). Otra ventaja que aporta es que acelera notablemente el desarrollo de un entorno de verificación completo, lo que permite reducir los tiempos de desarrollo.

A continuación se detallan algunos proveedores de VIP, que son los que poseen un catálogo más amplio de posibilidades, así como las diferentes estrategias que ambos han llevado a cabo:

- Synopsys VC Verification IP está completamente desarrollado en lenguaje SystemVerilog, además de adoptar la filosofía de la metodología UVM. [7] Es compatible con los principales simuladores del mercado. Proporciona medidas para la latencia y el rendimiento, un modelo para las conexiones totalmente configurable, así como verificaciones del protocolo a nivel de sistema, comprobaciones de la integridad de los datos y de coherencia de los mismos. Implementado en miles de proyectos, Synopsys VIP admite los protocolos Arm® AMBA®, CCIX, Ethernet, MIPI®, PCIe®, USB, DRAM y FLASH, además de protocolos específicos en muy diversos ámbitos: automóvil, almacenamiento y otros protocolos BUS / interfaz. También comercializan su propio depurador Synopsys Verdi® Protocol Analyzer. [8].
- Cadence es su gran competidor, con un amplio catálogo que incluye más de 40 protocolos de comunicación y 60 interfaces de memoria. La principal diferencia con el anterior es que permite el uso de diferentes lenguajes: Verilog, SystemVerilog, VHDL, incluso C/C++. Por lo que su principal fortaleza es que Cadence VIP se adapta a casi todos los entornos de verificación con soporte para los principales lenguajes de verificación, además de ser compatible con los principales simuladores del mercado.[9]
- El VIP de Mentor se integra perfectamente en entornos de verificación avanzados, incluyendo bancos de pruebas en distintos lenguajes: Verilog, VHDL y SystemC. Es el único del mercado que ofrece una arquitectura completa UVM en SystemVerilog,

pudiéndose adaptar a los lenguajes que se han comentado. [10]

Desde el año 2010 en que fue creada la metodología UVM, ha ido tomando relevancia y generalizándose su filosofía. Sin embargo todas las metodologías que se han comentado han sido tradicionalmente en Verilog. Esto suponía un esfuerzo adicional para los equipos que trabajaban con VHDL, ya que tenían que adaptarse a un nuevo HDL. Tanto es así que hace un par de años se creó UVVM (Universal VHDL Verification Methodology). UVVM es una librería de código abierto destinada a mejorar la legibilidad, el mantenimiento y la reutilización de los testbench. Incluye un entorno de verificación de VHDL completo, con las siguientes características:

– Ofrece una gran descripción y legibilidad del banco de pruebas

– Banco de pruebas basado en transacciones

– Permite extensibilidad y mantenimiento simple

– Promueve la reutilización de la verificación

– Es de código libre y abierto

– Incluye los componentes de verificación más básicos en código libre, como SPI, UART, I2C...

Está disponible en Github, una de las mejores plataformas de desarrollo colaborativo, disponible para cualquier usuario.[11]

La metodología UVVM se lanzó en marzo de 2016 como una arquitectura y biblioteca estandarizadas destinadas definir una forma común de controlar los componentes en la verificación. A lo largo de todo ese año adquirió los primeros usuarios, tomando cierta relevancia alrededor de todo el mundo. Ya fue en febrero de 2017 cuando se lanzó la segunda versión, incluyendo bastante mejoras. Entre otras, los componentes de verificación podrían controlarse desde múltiples secuenciadores simultáneamente, se agregaron características nuevas de sincronización, etc.[12] Durante el año 2017 se disparó el número de usuarios. Además, en el año 2018 se ha añadido soporte para los principales simuladores de VHDL.

UVVM se compone de dos partes:

- UVVM Utility Library (anteriormente Bitvis Utility Library), proporciona las funcionalidades básicas para ser utilizadas en cualquier testbench VHDL. Es una

infraestructura básica que permite un desarrollo mucho más rápido del banco de pruebas con un buen mecanismo de registro y control de alertas. Además aporta algunos procedimientos de control útiles, como el control del valor de la señal. También incluye soporte para el manejo de cadenas y BFM (Bus Functional Models), y un conjunto simple pero eficiente de funciones para la generación de valor aleatorio.

- UVVM VVC (VHDL Verification Component) Framework proporciona una funcionalidad más avanzada, permitiendo a los usuarios crear una arquitectura de banco de pruebas muy estructurada, incluso para dispositivos DUT grandes y complejos. Permite que múltiples interfaces sean estimuladas y controladas simultáneamente y de una manera muy estructurada

Gracias a UVVM se ha mejorado significativamente la eficiencia y la calidad en la verificación, y ha permitido acortar en la medida de lo posible los tiempos de desarrollo del producto. Hay que reconocerle el mérito de que ha permitido estandarizar la interfaz de comando, integración, depuración y arquitectura interna de los componentes de verificación de VHDL. Sin embargo, al ser un proyecto relativamente nuevo aún no integra gran cantidad de protocolos.

Resumiendo, las metodologías modernas de verificación han permitido estandarizar el proceso de generación del banco de pruebas. A pesar de ello, gran parte del tiempo de desarrollo sigue siendo innecesariamente consumido en la generación del testbench, especialmente en la generación de los drivers y los monitores de protocolo . En el caso de los protocolos más usados, suele haber disponible alguna solución de VIP, aunque no toda es de código libre. No obstante, para protocolo más específicos en aplicaciones más concretas, casi siempre resulta necesario terminar implementando estos drivers y monitores a mano. Surge entonces la necesidad de facilitar en la medida de lo posible este proceso, para que no resulte tan tedioso y no consuma tiempo y recursos innecesariamente.

3. SOLUCIÓN PROPUESTA

3.1. Descripción

Se trata de generar de forma automática los drivers en VHDL para diversos protocolos. Las ventajas que nos aporta, al tratarse de una generación automática son evidentes:

– Carecemos del riesgo del error humano

– Podemos realizar cualquier cambio en algún parámetro, y generar el nuevo driver y realizar los nuevos testbenches de forma inmediata

– El problema de la sincronización queda absolutamente resuelto. Como se genera automáticamente, la operatividad del sistema queda totalmente garantizada para cualquier transacción que se reciba.

– Es evidente que al generarse de forma instantánea se reduce notablemente el esfuerzo de desarrollo, así como el tiempo que éste supone.

Se trata de la creación de testbenches de alto nivel modelados a nivel de transacciones, donde el driver es el encargado de recibir la transacción. Se consigue así aislar en un solo módulo toda la complejidad de traducir dicha transacción en señales. Como se trata de un módulo generado automáticamente, las propiedades cuyo buen funcionamiento se hayan comprobado con una simulación, cabe esperar que se comporten de igual manera en el resto de ejecuciones, siempre que no cambien las condiciones. Esto permite ir incrementando la complejidad paso a paso, garantizando en cada uno de ellos la funcionalidad del código que ya se haya comprobado en etapas anteriores.

Otra ventaja que nos aporta el programa al que se refiere este documento, es que se generan simultáneamente diversos módulos o drivers, cada uno correspondiente a un protocolo diferente. Cada uno de ellos puede ser simulado de forma individual, o el usuario puede combinarlos y usarlos de forma conjunta. Por lo tanto, se consigue que múltiples interfaces sean estimuladas, simuladas y visualizadas simultáneamente, de una manera muy ordenada y estructurada.

Cabe resaltar que la filosofía de esta generación automática de diferentes drivers es que se

realice a través del mismo código. Es decir, que se generen de forma instantánea y automática varios módulos de diversa índole, pero que todos ellos se hagan compilando el mismo programa con distintas transacciones. Esto querría decir que responde a las demandas de adaptabilidad y reutilización que existen en la actualidad en este campo.

3.2 Alcance

El objetivo de este código no es simplemente generar una serie de drivers simulados a través de testbenches con transacciones. Tampoco es el de generar un sistema completo driver/monitor para diversos protocolos. No se trata de competir con metodologías de verificación de código abierto tan potentes como UVVM. Se trata de ir un paso más allá, es decir, de ver si es realizable la generación automática y simultánea de diferentes drivers en VHDL, todos ellos a través del mismo código. Podría entenderse como un generador de drivers genérico, que implementa un protocolo u otro dependiendo de la transacción que recibe.

En esta experiencia se van a modelar todas las señales necesarias para implementar los diversos protocolos en sentido al DUT. Esto quiere decir que se generarán todas las señales de salida que están implicadas en la comunicación, desde el punto de vista del maestro. No se van a contemplar las comunicaciones o las operaciones en sentido opuesto (del esclavo al maestro) o las bidireccionales.

Si esta experiencia resulta satisfactoria podría extenderse a la creación de monitores que realizaran la tarea opuesta, y tradujeran de alguna manera el movimiento de señales en una transacción de salida. Esta siguiente etapa ya nos permitiría la implementación de protocolos bidireccionales que se comentaba en el párrafo anterior. Con todo esto, se conseguiría aumentar completamente el nivel de abstracción, teniendo que manejar únicamente transacciones de más alto nivel tanto de entrada como de salida.

Por lo tanto, en este documento se trata de estudiar la viabilidad de la solución que se propone. Además, si se obtiene éxito en esta experiencia, se abriría un nuevo horizonte en lo que a la verificación se refiere. Teniendo en cuenta las herramientas que existen en la actualidad, esto podría significar una nueva línea de desarrollo en este campo.

4. ANÁLISIS

En este apartado se va a describir de forma detallada la solución propuesta, los requisitos que es indispensable que cumpla, así como el flujo de trabajo dentro del diseño.

La estructura de la solución propuesta es la siguiente:

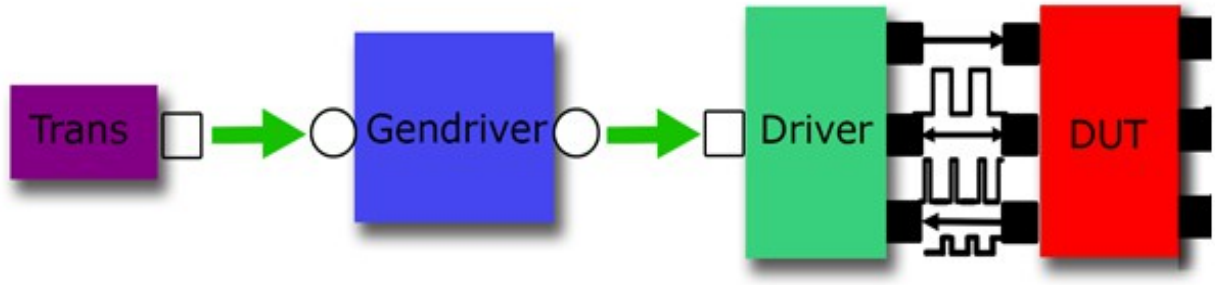


Ilustración 3: Estructura de la solución propuesta

Como puede verse en la figura anterior, se proporciona una descripción de la transacción a un módulo al que llamaremos gendriver, escrito en lenguaje de alto nivel. Éste será el encargado de capturar los campos de la transacción y generará automáticamente el driver que corresponda. Será este módulo del driver el encargado de traducir dichos campos de la transacción en movimiento de pines, que permiten la comunicación con el dispositivo que se está probando (DUT).

A continuación se detallan cada uno de los bloques, y los requisitos fundamentales que tienen que cumplir cada uno de ellos:

1. Transacción: el requisito fundamental que tiene que cumplir la transacción es que sea compatible con el lenguaje de alto nivel, además de ser lo más legible, intuitiva y comprensible. Al fin y al cabo esa es la finalidad de elevar el nivel de abstracción. Además, se requeriría un formato de transacción que facilitara en la medida de lo posible el proceso de captura de los datos. Evidentemente se trata de utilizar un formato abierto y que sea de uso generalizado. Las opciones que se presentan son las siguientes:

- csv (Comma-separated values): este formato es el más compacto de los tres. Su tamaño es aproximadamente la mitad que los otros dos. Esto lo hace adecuado para transferir

grandes conjuntos de datos, todos con la misma estructura. Sin embargo, si no se aporta documentación puede resultar casi imposible interpretar el significado de las diferentes columnas. Por lo que esto lo convierte en el formato menos versátil de los tres. Además, si la estructura de datos cambia, es tarea ardua actualizar la propia transacción, además de tener que replantear los analizadores. Esto tiene como consecuencia una falta de adaptabilidad que puede resultar clave. Otro enorme inconveniente es que no admite jerarquía de datos, lo que complica en gran medida el diseño del analizador.

- Xml (Extensible Markup Language): fue diseñado en 1996, aunque oficialmente se convirtió en un estándar en 1998. Fue creado con el objetivo de representar los formatos de datos con una estructura jerárquica. Esta es la principal ventaja que aporta respecto al anterior, que es totalmente compatible con las estructuras de datos jerárquicos y es muy apropiado cuando se reciben datos complejos como respuesta. Otro beneficio que aporta respecto al anterior es que es muy legible. Sin embargo, tiene un tamaño unas tres veces superior al mismo en csv, aunque esto no ha impedido que su uso esté extremadamente generalizado.

- Json (JavaScript Object Notation): Fue inventado en 2001 y se popularizó entre otros gracias a Yahoo y Google en 2005 y 2006. Fue creado como una alternativa a XML, aunque también representa datos jerárquicos con el uso de comas, llaves y corchetes. Por lo tanto, este formato de datos admite datos jerárquicos y además tiene un tamaño más reducido que XML.

  Como su nombre indica, también se creó para analizar datos más fácilmente en objetos JavaScript nativos, lo que lo hace muy útil para aplicaciones web. JSON combina las mejores características de los dos formatos anteriores. Es simple y compacto como CSV, pero admite datos jerárquicos como XML. Una ventaja que presenta respecto a este último, es que los formatos JSON son solo dos veces más grandes que los formatos CSV.

2. Gendriver: se ha denominado así al bloque encargado de recibir una descripción de la transacción y generar automáticamente el driver en código VHDL. Evidentemente un requisito fundamental es que se trate de un lenguaje de alto nivel, capaz de capturar los campos de la descripción de la transacción que recibe y, en consecuencia, generar el driver correspondiente.

Como con el mismo código tiene que ser capaz de generar distintos drivers dependiendo de la transacción recibida, se hace necesario que el módulo del generador reciba el fichero de la transacción como argumento al ejecutar el compilador en la línea de comandos. Este será un requisito indispensable en el proceso de diseño: que el programa sea capaz de responder de manera diferenciada ante cualquier tipo de transacción, por lo que debe recibir la descripción de ésta de forma externa. A efectos prácticos, se trata de escoger un lenguaje de alto nivel capaz de recibir un fichero como argumento en la llamada, requisito que cumplen todos los candidatos que se proponen.

También resulta obvio, que al tratarse de un generador automático de código, éste debe imprimir de alguna manera el driver a generar en lenguaje VHDL. Además, debe ser capaz de capturar esta salida en un archivo independiente.

Sería recomendable también incluir una serie de aserciones o respuesta frente a errores. Es decir, que dicho programa fuera capaz de responder a comportamientos indeseados o usos indebidos del mismo. Además resulta muy aconsejable que en el mensaje de error se recomiende el uso debido y correcto del mismo. Así se consigue que el propio mensaje de error provoque el cambio necesario para solucionarlo, no volviendo a ser generado.

Además se tiene que tratar de un lenguaje lo más versátil posible. El gran desafío de este documento es ser capaz de generar cualquier tipo de protocolo con la misma pieza de código. Por lo que se necesita un lenguaje de alto nivel lo más simple y adaptable, lo más flexible e intuitivo posible.

A continuación se detallan los posibles candidatos y las características más relevantes de ellos, que no harán decantarnos por una u otra opción:

- C/C++: Desarrollado por primera vez en 1969 en AT&T Bell Labs, y C++ apareció en 1983. C se considera un lenguaje de programación de propósito general. En sus inicios fue diseñado para tareas de administración de memoria de bajo nivel, que previamente se habían escrito en lenguaje ensamblador (código escrito en formato hexadecimal que accede directamente a las ubicaciones de memoria). Por esta razón, todavía se usa

ampliamente en la arquitectura del sistema operativo. C sigue siendo el lenguaje de programación más utilizado de todos los tiempos y como consecuencia su desarrollo y estandarización ha sido muy notable en las últimos tiempos. Además, la naturaleza estática de este lenguaje de programación ayuda a prevenir operaciones involuntarias.

A pesar de todo ello, también presenta algunas limitaciones. El carácter estático que ya se ha comentado provoca que no sea un lenguaje dinámico. Es decir, que son diseños más rígidos, por lo que puede resultar más complicado adaptarlos a la programación de scripts.

- Python: es un lenguaje de programación más reciente, ya que fue creado en 1989 por Guido van Rossum, aunque fue presentado en 1991. Desde entonces, se ha convertido en uno de los lenguajes de programación más populares. Todos los lanzamientos de Python son de código abierto y se pueden utilizar y distribuir libremente, incluso en proyectos comerciales.

  Igual que el anterior, se trata de un lenguaje de alto nivel de propósito general. La sintaxis enfatiza la legibilidad del código hasta tal punto que permite a los programadores realizar un diseño con sólo el 10% del código que sería necesario con otros lenguajes, como C. La importancia de la legibilidad no puede infravalorarse: cuando se trabaja en equipo, este aspecto es fundamental para el mantenimiento del código y la sincronización de los diferentes equipos de trabajo. Además la falta de legibilidad tiene una consecuencia directa sobre el tiempo de desarrollo, y por lo tanto también en los costes.

  Otra ventaja que aporta respecto al lenguaje anterior es que mientras que C / C ++ es más propenso a errores, Python es conocido por su capacidad de escritura y por la reducción de dichos errores. Además, la reutilización del diseño de Python supera ampliamente a la de C / C ++, y esta característica es fundamental para mantenerse a la vanguardia en este campo.

  Una de las circunstancias que han provocado el rápido crecimiento de este lenguaje es el desarrollo de sistemas cada vez más complejos, como redes neuronales. Este crecimiento provoca que la reducción del tamaño del código que se consigue con este lenguaje no sea ya un valor añadido sino un requisito indispensable.

  Además la existencia de numerosas bibliotecas aumentan drásticamente la productividad de los programadores, pudiendo importar dichas bibliotecas con un solo comando, y

luego reutilizar los módulos que contienen las veces que sean necesarias.

3. Driver: El driver indiscutiblemente estará escrito en VHDL. Entre los requisitos que debe cumplir está el de que debe ser capaz de encapsular una transacción completa como un sólo objeto. Hay que tener en cuenta que los campos de esta transacción pueden ser de diversos tipos: enteros, lógicos, vectores, etc. Aún así, el programa debe ser capaz tanto de tratarlo como un sólo objeto, como de discriminar los diversos campos que contiene.

Además, la filosofía del documento que nos atañe es la de que todas las señales generadas estén sincronizadas con el flanco activo de reloj. Como ya se ha comentado, el driver es el encargado de sincronizar todas estas señales, por lo que será necesario cumplir todos los requisitos de sincronización en este módulo.

Una vez vista la estructura de la solución que se propone, se quiere hacer énfasis en el beneficio que ésta aporta al diseño. El hecho de que el código generador del VHDL esté realizado en un lenguaje de alto nivel, permite que cada driver se genere con una sola compilación de un programa, es decir, con un solo comando. Por lo que se pueden generar un número prácticamente ilimitado de drivers simultáneamente. Éstos deben poder ser testeados, simulados y visualizados conjuntamente. Quedará a disposición del usuario la posibilidad de poderlos combinar entre sí para generar un solo driver. Sin embargo, es incuestionable que la modularidad siempre aporta una mayor flexibilidad, así como una mayor facilidad de uso de cara al usuario.

Este funcionamiento se detalla en el siguiente esquema:

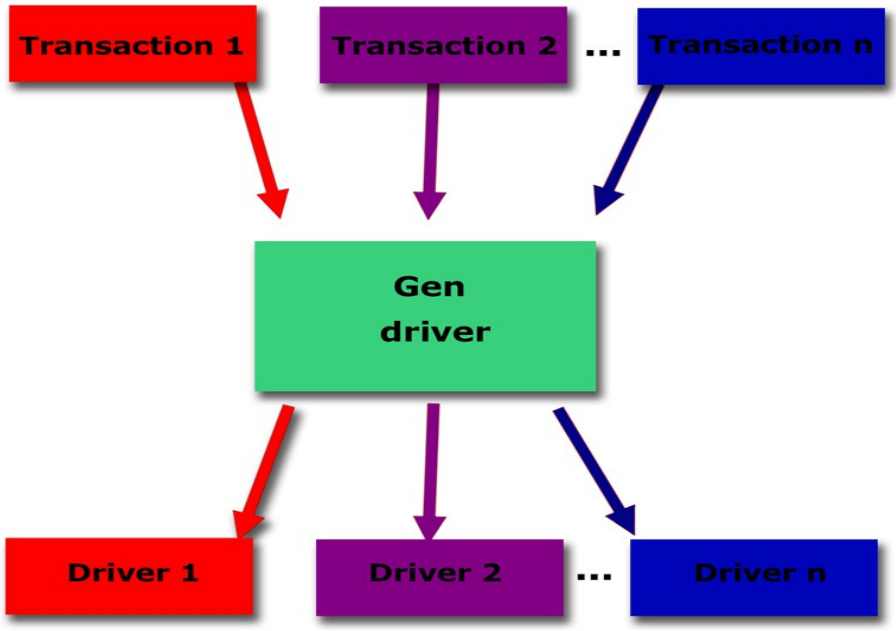


Ilustración 4: Creación simultánea de un número indefinido de drivers

Además, el beneficio de que sea una misma pieza de código la que genera los diversos drivers es evidente. La primera ventaja que supone compilar tantas veces como sea necesario una misma pieza de código, es que esto supone una garantía de robustez, fiabilidad y operatividad. Además, como ya se ha visto la reutilización del código es ya uno de los principios fundamentales de la verificación actual.

Ya se ha visto que este código es capaz de generar un número prácticamente ilimitado de drivers. Sin embargo, tener en un sólo instante un incremento de código tan grande puede resultar peligroso si no se tiene bien organizada la estructura de ficheros. Es necesario tener ordenado de una forma clara y precisa cada driver generado, por lo que se necesita una arquitectura de los ficheros jerarquizada. Sería recomendable tener clasificados, en carpetas diferentes, toda la información referente a cada uno de los protocolos estudiados. Dicha carpeta contendrá, al menos, el driver generado, una transacción y un testbench para simularlo. Además, habrá tantas carpetas como protocolos implementados. De esta forma además se facilita la experiencia del usuario. Al estar tan clara la estructura, siendo esta totalmente modular y con nombres representativos, se simplifica muchísimo el uso del programa desde el punto de vista del usuario.

5. DISEÑO

En el capítulo anterior se trató de detallar qué requisitos fundamentales debía cumplir la solución propuesta. En el capítulo actual se va a definir la implementación de los módulos propuestos, es decir, cómo se va a desarrollar en profundidad cada uno de ellos.

1. **Transacción:** aunque las primeras inspecciones se realizaron con un fichero csv, se hizo fundamental manejar estructuras de datos jerarquizadas, por lo que pronto fue desechado este formato. Se ha optado por el JSON, al ser el que ofrece mejores prestaciones.

   Se trata de tener una transacción diferente por cada protocolo. Además, cada transacción recibirá el nombre del protocolo que se refiere, para garantizar una mayor legibilidad. También cada una de ella estará almacenada en su carpeta correspondiente, como ya se ha comentado con anterioridad.

   Aunque cada transacción por definición ha de tener sus propias peculiaridades, hay una serie de características que tienen todas en común. La más importante es que han de tener una estructura jerarquizada e idéntica entre ellas, para garantizar que el programa encargado de ello sea capaz de traducirlas. Se describen a continuación los campos o elementos que son comunes a todas ellas:

   - 'generic': contiene los generic que serán declarados en la entidad VHDL
   - 'constants': almacena las constantes a declarar. La diferencia principal es que como el generic se declara dentro de la entidad, se le puede asignar un valor diferente cada vez que se instancia este componente. Las constantes permanecen inmutables durante toda la ejecución.
   - 'tran': en este campo se hace necesario definir toda la información que se ha de convertir en movimiento de pines. Evidentemente, su contenido dependerá del protocolo a definir tanto en número como en tipo de campos. Es decir, se creará un campo diferente dentro de este nivel por cada dato individual que se necesite transmitir en el movimiento de pines, tales como: byte a enviar, direcciones de registros, direcciones de dispositivos, y un largo etcétera.
     En principio, todos estos campos serán declarados como entrada en el fichero VHDL.

- 'Interface': se creará un campo diferente por cada señal que se quiera generar y enviar al DUT. En este caso todas ellas serán de tipo lógico, pero cada una de ellas tiene que ser capaz de gestionar un número indefinido de datos, es decir, transmitir señales de una longitud en principio indefinida, tan extensa como lo requiera la operación que se esté llevando a cabo.

Se ha comentado la estructura que siguen todas las transacciones que usa nuestro generador. Es necesaria que ésta sea uniforme para que pueda responder de la misma manera ante las distintas transacciones.

Otro punto en común de todas ellas es que en el interior de los elementos que ya se han definido, cada campo que lo integra, contiene al menos los dos siguientes atributos:

- 'name': indica el nombre de la señal a generar
- 'type': indica el tipo, que puede ser: entero, lógico, vector, etc...

Además, dependiendo de la naturaleza del campo, será necesario declarar también el siguiente atributo, siempre adaptándose a los requerimientos de cada protocolo:

- 'values': nos da a conocer los valores que van a tomar los distintos campos de la transacción. Dicho valor puede ser de distintos tipos: desde una constante a una variable tomada de otro campo de la transacción. Esta es una de las claves de la versatilidad de esta solución. Vamos a ver las distintas características que debe cumplir este campo, dependiendo del elemento de la transacción al que se refiera:

  - En 'constant' y en 'generic', casi siempre será un valor entero, aunque no tiene por qué ser necesariamente así.

  - El elemento 'tran' contiene tantos campos como haya que configurar para implementar cada uno de los protocolos. Como han de ser configurables de forma externa, y debido a la filosofía que tiene este proyecto y que ya se ha comentado, los valores a los campos de esta parte de la descripción de la transacción serán establecidos desde el testbench, y no desde la propia transacción. Por lo tanto este atributo no es necesario en la definición de este elemento.

  - El campo 'interface' puede contener datos de distintos tipos: constantes o

variables. Resulta especialmente interesante darle el valor de otros campos de la transacción, por ejemplo de los datos de entrada ubicados en el campo denominado 'tran'.

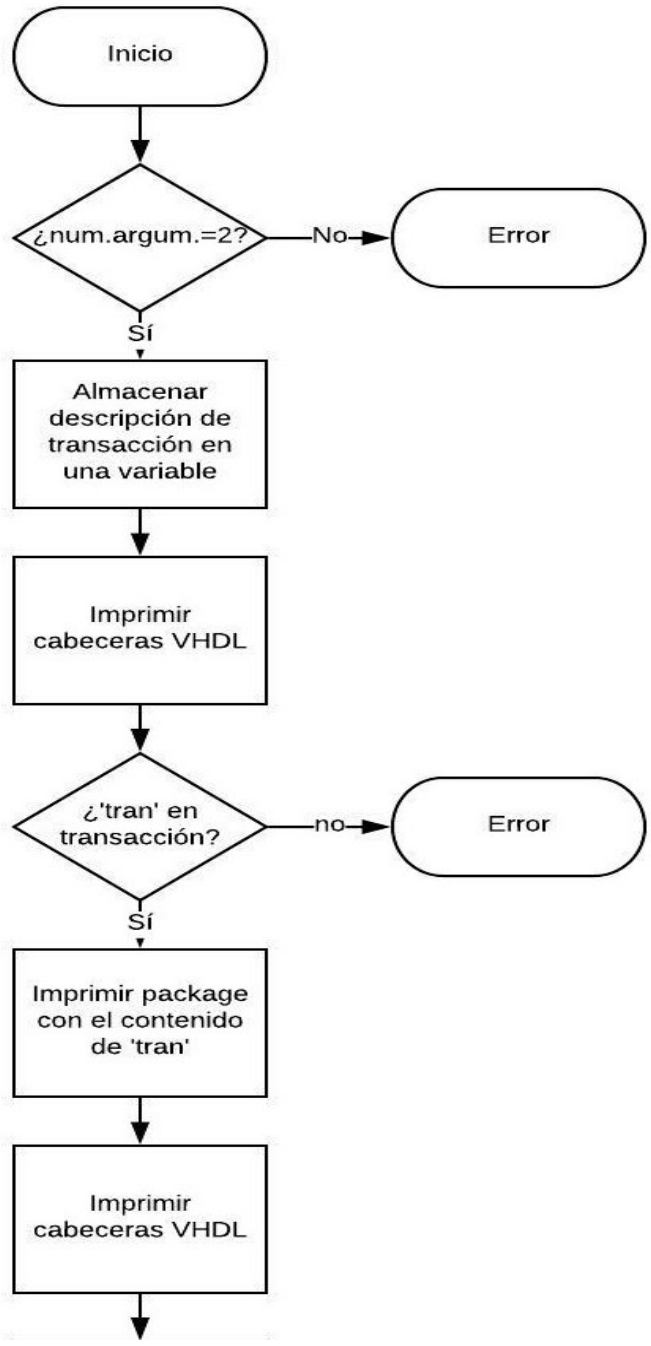
Una característica que tienen que cumplir los elementos que formen parte de este campo, es que algunas señales deberán poder cambiar su valor pasado un tiempo determinado. Y repetir este proceso de forma periódica, sincronizada con la señal de reloj. Debido a este requerimiento, se hace evidente la necesidad de indicar, además del valor que tienen que tomar las distintas señales, la duración que tienen que tener cada uno de esos valores. Por lo tanto, se hace necesario crear dentro de cada dato que contenga este campo un nuevo atributo, que indicará el número de ciclos de reloj que tienen que estar activos dichos valores en las señales generadas.

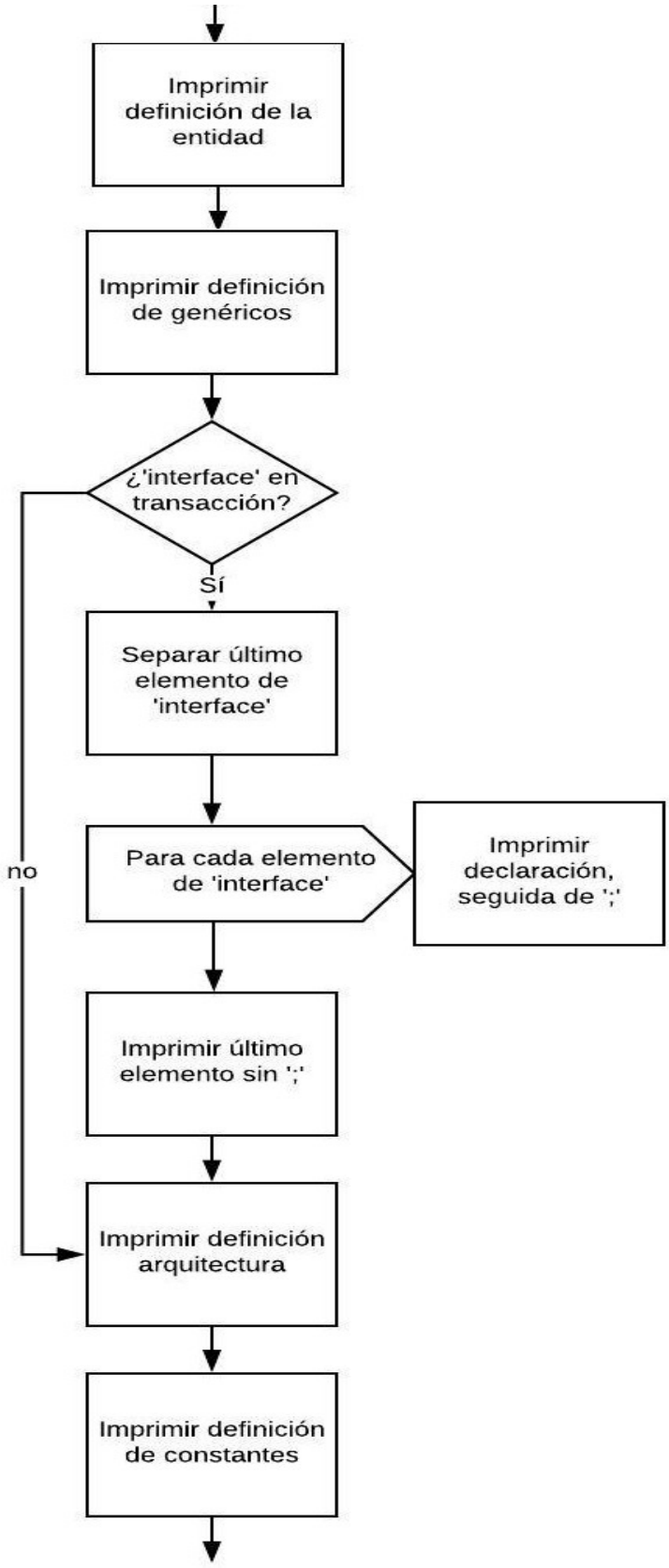
2. Gendriver: la primera versión de este módulo que se realizó fue en lenguaje C. Después de realizar las primeras pruebas sobre el primer driver, y a la hora de automatizar el proceso para el resto, resultó imposible exportarla tal y como se había realizado. El principal problema surgió al intentar capturar los distintos campos de la descripción de la transacción., que está en formato json Es cierto que existen librerías de código libre en que se incluyen este tipo de traductores de formato json a estructuras de datos. Sin embargo, no son lo suficientemente sencillos e intuitivos como para poder adoptarlos sin complicar en gran medida el código que se presenta. Por lo tanto la única opción que queda es generar un código especialmente para ello, y hacerlo a mano puede resultar engorroso. Y esa no es la filosofía de este proyecto. En ese momento se optó por adoptar el python como lenguaje de alto nivel, resultando ser la opción más satisfactoria. Hay que destacar que con python se consigue convertir los campos del fichero json en una estructura ordenada de datos con una sola línea de código. Es evidente que se hace a través de una librería que hay que importar, pero su uso es extremadamente simple, y la reducción de código y complejidad que genera es muy importante.

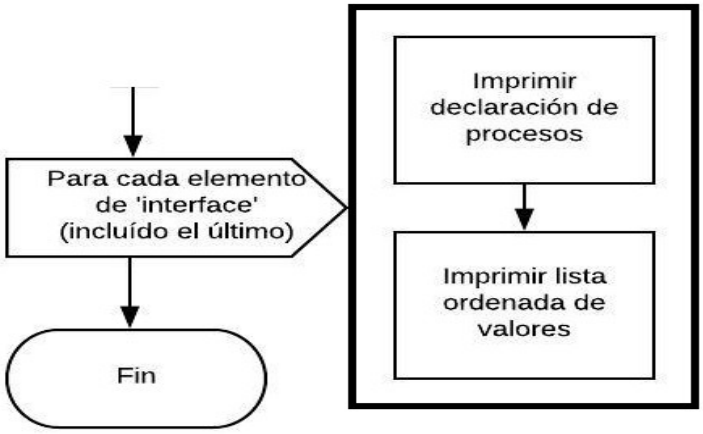
El bloque gendriver tiene el mismo código para todas las transacciones. Por lo tanto, es necesario indicarle de forma externa a qué transacción se refiere, recibiendo como argumento el fichero correspondiente. Así, se le permitirá generar el driver conveniente en cada caso.

Por lo tanto, por cada módulo a generar, será necesario realizar una compilación del programa. Cada una de ellas va a generar un fichero vhd diferente, nombrado con el protocolo que representa. Además cada uno estará alojado en su carpeta correspondiente, según la estructura jerarquizada de archivos que hemos comentado anteriormente.

Debido a que este módulo es el principal objeto de este proyecto, se procede a describirlo más detalladamente mediante un diagrama:




Imprimir definición de la entidad
Imprimir definición de genéricos
¿'interface' en transacción?
Sí
no
Separar último elemento de 'interface'
Para cada elemento de 'interface'
Imprimir declaración, seguida de ';'
Imprimir último elemento sin ';'
Imprimir definición arquitectura
Imprimir definición de constantes



El programa tendrá la siguiente estructura:

- tras importar las colecciones o librerías que sean necesarias, hay que comprobar que el número de argumentos recibidos ha sido el adecuado. Si no lo ha sido, se genera el error correspondiente, informando de cómo se ha de invocar el programa adecuadamente.
- Tras esta comprobación, se abre el fichero json que indique el argumento que se ha recibido. Como dicho fichero tiene una estructura jerarquizada, se ha de almacenar en una variable que también posea esta característica. Así, el contenido de los diversos campos podrá ser gestionado en el resto del programa. En python existe un tipo de estructura idónea para este caso: el diccionario ordenado. Es un tipo de estructura de datos que permite guardar una lista de parejas de elemento. Se trata de un conjunto ordenado de pares clave-valor, denominados tuplas. La clave se utiliza para acceder al valor correspondiente, por lo que tiene que ser un dato inmutable, como por ejemplo una cadena de caracteres. Es requisito indispensable que las claves sean únicas dentro de un mismo diccionario. El valor puede ser un número, una cadena de caracteres, un valor lógico, o incluso una lista o otro diccionario. Por lo tanto, al poder anidar varios diccionarios ordenados, se pueden crear estructura de datos más complejas. Además, gracias a las claves se pueden gestionar de una manera muy intuitiva y por lo tanto eficiente.
- Una vez se ha almacenado el contenido de la transacción que corresponda, se procede a imprimir el código VHDL necesario. Una de las dificultades que presenta esta etapa es la de introducir en el programa los campos de la transacción cada vez que sea necesario. Además en VHDL cuando se definen varias señales dentro de port, o varios genéricos, hay que separarlo entre ellos por un punto y coma. Sin embargo, el último elemento en

cada uno de estos apartados debe carecer de este separador. Por lo tanto, tiene que ser capaz de discriminar en cada campo de la transacción qué elemento es el último. Además de detectarlo, tendrá que imprimirlo de una manera diferenciada al resto.

3. Driver: el driver estará realizado completamente en VHDL, y seguirá la siguiente estructura:

- Inclusión de librerías necesarias en el programa
- Definición de un package denominado tran. Como tiene que ser tratado como un solo objeto, y además tiene que soportar campos de diversos tipos, se hace necesario declarar una variable tipo record. En este record se definen todos los elementos que contiene el campo 'tran' de la transacción. Cada uno de ellos se define con su tipo correspondiente, según indique el atributo asociado. El valor que tomarán todos los campos de este package no se le asignarñan desde este módulo, sino desde el testbench que corresponda en cada caso.

  Además todos los packages, independientemente del protocolo que implementen, deberán contener un campo al que denominaremos 'valid'. Este será el encargado de indicar que hay una transacción válida para ser gestionada, y comenzar con la generación de las señales. Por lo tanto, cuando esta señal se active desde el testbench, comenzará el movimiento de pines, sincronizado con la señal de reloj.
- En VHDL es necesario incluir las librerías que se van a usar cada vez que se declara una entidad. Por ello, una vez que se ha declarado el package, es necesario volver a incluir las librerías. Además habrá que añadir una nueva, la librería work, que contiene el package anterior.
- A continuación se declara la entidad. Aquí hay que definir:

  - la lista de todos los generic, discriminando el último elemento para no imprimirlo con el punto y coma.

  - en port hay que especificar todos los pines de entrada y salida. Necesariamente habrá que declarar como entradas una señal de reloj y una variable de entrada tipo transacción declarada en el package. También habrá que definir todas las señales de salida que se indiquen en el campo 'interface' de la transacción.
- Se comienza con la arquitectura y se declaran las constantes. Es necesario capturarlas del campo 'constant' de la transacción.

- A continuación habrá un proceso por cada elemento que haya en el apartado 'interface' de la transacción, es decir, por cada señal que haya que generar. Este proceso será el encargado de dar el valor que le corresponda a las señales de salida, la duración que indique el atributo correspondiente.

6. PRUEBAS Y RESULTADOS

6.1. Protocolos implementados

A continuación se detallan los protocolos que se han implementado, así como la descripción de las transacciones necesarias para ello. Estas distintas implementaciones se han realizado con el fin de demostrar que un mismo código es capaz de generar distintos protocolos, cada uno con sus peculiaridades. Aquí se van a definir cuáles son esas peculiaridades o requisitos de cada uno de ellos. Después habrá que comprobar en una simulación que todos ellos se han cumplido.

1. UART (Universal Asynchronous Receiver-Transmitter): la primera prueba se realizará con este protocolo debido a su sencillez, ya que hay que generar una sola señal, la línea de datos. Esto quiere decir que la arquitectura de este primer driver contendrá un solo proceso.

Por lo tanto, en este caso la transacción sólo tendrá una variable de entrada denominada data, de longitud 8 bits, que contendrá los datos a transmitir. Además tendrá una señal de salida denominada tx en la se irán colocando los bits, comenzando por el menos significativo. A continuación se adjunta parte de la descripción de la transacción que se usará, con el fin de ilustrar el funcionamiento del programa:

```
{
 "generic" :
      {
        "generic0":{ "name" : "UART_CYCLES", "type" : "integer", "value" : 1 }
      },
 "tran":
      {
        "field0" : { "name" : "data", "type" : "std_logic_vector(7 downto 0)" }
      },
 "interface":
      {
        "port0" :
            { "name"   : "tx",
              "type"   : "std_logic",
              "values" :
               {
                "val0" : { "val": "'0'", "cycles": "UART_CYCLES" },
```

```
                "val1" : { "val":"input_tran.data(0)","cycles": "UART_CYCLES" },
                "val2" : { "val":"input_tran.data(1)","cycles": "UART_CYCLES" },
                "val3" : { "val":"input_tran.data(2)","cycles": "UART_CYCLES" },
                           .. .. ..
            }
        }
    }
}
```

Se necesita definir un sólo campo en generic, que será el que se corresponda con el número de ciclos de reloj que debe permanecer cada bit en la señal de salida. En el campo 'tran' sólo tenemos una entrada, que es el byte a enviar, cuyo valor se inicializará desde el testbench. Por último se van asignando los valores correspondientes a las señales de salida en el atributo respectivo.

Una transmisión de este tipo sigue el siguiente esquema:

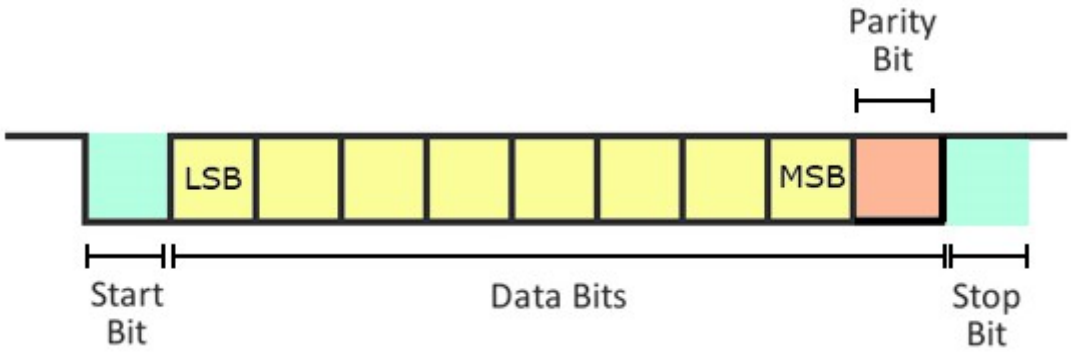


Ilustración 5: Transmisión UART

- La línea está inicialmente en reposo, con valor lógico 1
- A continuación el bit de start pone la línea a 0 para indicar el inicio de la transmisión
- Se envían 8 bits de datos, comenzando por el menos significativo
- El bit de paridad como método de detección de errores
- El bit de stop pone la línea a 1, dejándola así de nuevo en reposo

2. **I2C (Inter-Integrated Circuit) :** en este caso los campos que contiene la transacción son los siguientes:

```
"generic" :
   {
    "generic0" : { "name" : "SCL_CYCLES", "type" : "integer", "value" : 10 },
    "generic1" : { "name" : "SCL_PERIOD", "type" : "integer", "value" : 20 },
    "generic2" : { "name" : "SCL_SYNC", "type" : "integer", "value" : 15 }
   },
"constant":
   {
    "constant0": { "name" : "ACK", "type" : "std_logic", "value" :"'0'" },
    "constant1": { "name" : "NACK", "type" : "std_logic", "value" :"'1'" },
    "constant2": { "name" : "R_W", "type" : "std_logic", "value" :"'0'" }
   },
"tran":
   {
    "field0":{"name": "slave_address","type" : "std_logic_vector(6 downto 0)"},
    "field1":{"name": "reg_address", "type" : "std_logic_vector(7 downto 0)" },
    "field2":{"name": "data", "type" : "std_logic_vector(7 downto 0)" }
   },
```

- Se han definido dos genéricos adicionales, que serán los encargados de la sincronización particular de este protocolo.
- Se requieren tres constantes, debido a que sólo se implementará la comunicación en un solo sentido. Por ello, los bits de asentimiento por parte del esclavo (ACK y NACK) y el bit de operación (R/W) se mantendrán constantes durante todas las ejecuciones.
- Se requieren tres valores de entrada, a inicializar desde el testbench:
  - 'slave_address': contiene la dirección del esclavo con el que el maestro quiere establecer la comunicación
  - 'reg_address': dirección del registro interno del esclavo al que se refiere la operación
  - 'data': contiene os datos a transmitir. Tiene una longitud de o bits

Además este protocolo tiene además dos salidas a generar:

- SCL: señal de reloj, necesaria para el sincronismo con el esclavo
- SDA: línea por la que se transmiten los datos, enviando cada byte comenzando por el más significativo. Es decir, se sigue la lógica contraria al caso anterior.

Es importante resaltar algunas características de este protocolo. La transmisión de datos a través de la línea SDA se realiza byte a byte, comenzando por el bit más significativo. Después del envío de cada byte se recibe un bit de reconocimiento ACK (Acknowledge) procedente del esclavo, en el noveno pulso de reloj.

Los bits se transmiten de forma sincronizada con la señal de reloj. Estos bits se colocan en la línea SDA sólo cuando SCL tiene un valor lógico 0. Es decir, que los bits se actualizarán en la línea en los semiperiodos a nivel bajo de la señal de reloj. Este factor es muy importante, ya que la línea SDA debe permanecer estable, y jamás cambiar, mientras la señal SCL esté a nivel alto y se estén transmitiendo datos. Será en este semiperiodo a nivel alto de la señal de reloj cuando el esclavo capture el dato de la línea.

Las secuencias de inicio y la de parada son especiales, ya que son los dos únicos casos en que se permite que la línea de datos (SDA) cambie cuando la línea de reloj (SCL) está a nivel alto.

A continuación se detalla una transmisión completa de este tipo:

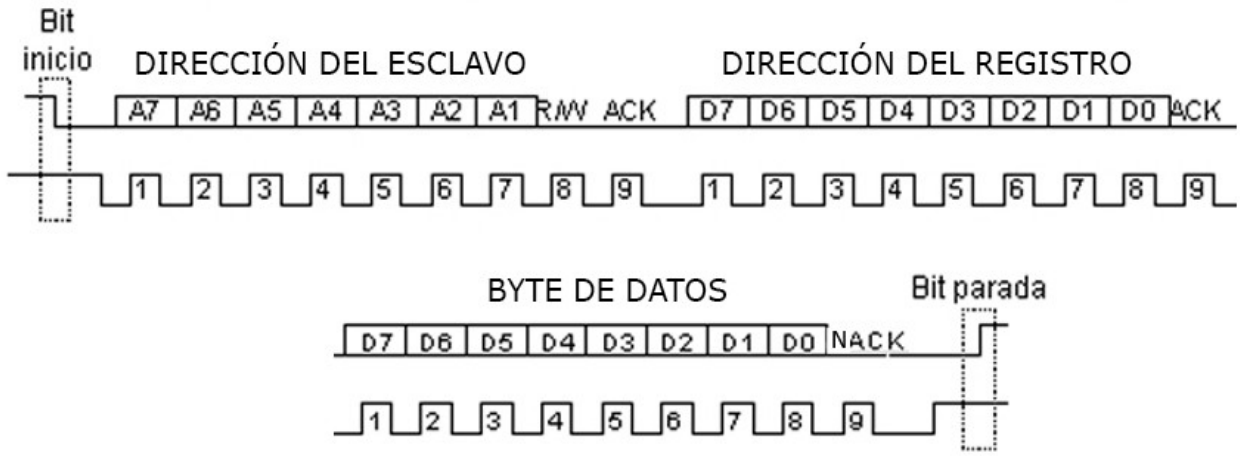


Ilustración 6: Ejemplo de comunicación con protocolo i2c

- La línea de datos está inicialmente en reposo, con valor lógico 1
- Cuando el dispositivo maestro quiere comunicarse con un esclavo, produce una secuencia de inicio en el bus. La secuencia de inicio marca el inicio de la transmisión. Ésta consiste en poner un valor lógico 0 en la línea de datos SDA mientras la señal de reloj SCL está a nivel alto.

- A continuación envían 7 bits que contiene la dirección del dispositivo esclavo con quien se quiere establecer la comunicación. Esto quiere decir que se pueden direccionar hasta 128 dispositivos en un bus.
- Después se envía un bit de lectura escritura (R/W) que será el que marque el sentido de la comunicación. Sí este bit está a nivel bajo se indica la operación de escritura, donde es el maestro el encargado de enviar los datos al esclavo. A nivel alto se indica la operación de lectura, es decir que será el esclavo el encargado de enviar los datos y el maestro de recibirlos
- Si el dispositivo cuya dirección se corresponde con la que se indica en los 7 primeros bits del byte anterior está presente en el bus, deberá enviar al maestro un bit de ACK o reconocimiento, activo a nivel bajo. Éste le indicará que el esclavo reconoce la solicitud y que está en condiciones de comunicarse. Aquí se establece la comunicación en firme y comienza el intercambio de información entre los dispositivos.
- A continuación el maestro indica la dirección de memoria a la que quiere acceder, es decir, la dirección del registro desde el que se quiere leer o en el que se quiere escribir. La cantidad de registros depende evidentemente del dispositivo, pudiendo variar en función de la complejidad del mismo.
- El esclavo envía un bit de ACK, activo a nivel bajo, si la dirección que ha recibido se corresponde con la ubicación de algún registro interno.
- A continuación, el maestro o el esclavo (a quien corresponda según el valor de R/W) envía un byte de datos, y espera el ACK de su interlocutor.
- Si el dispositivo que recibe envía un ACK a nivel bajo, indica que ha recibido el dato correctamente y que está preparado para recibir otro. Sin embargo, si se recibe un ACK a nivel alto, indica que no se pueden enviar o recibir más datos y el dispositivo maestro debería terminar la transferencia enviando una secuencia de parada.
- El maestro envía la secuencia de parada que consiste en provocar un cambio de 0 a 1 lógico en la línea SDA mientras la señal SCL está activa.

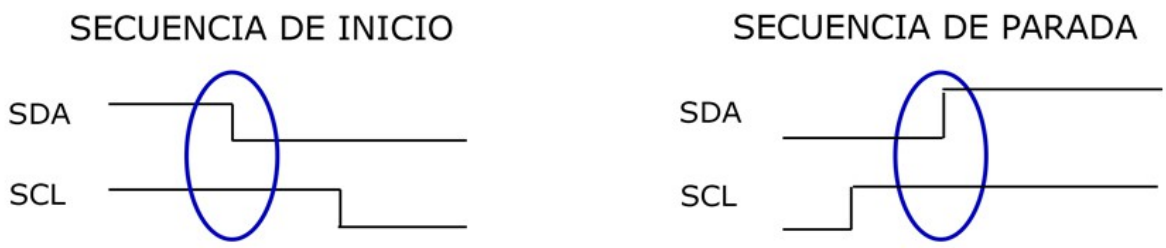


Ilustración 7: Secuencia de inicio y de parada en protocolo i2c

3. SPI (Serial Peripheral Interface): en este caso la descripción de la transacción ha de contener los siguientes campos:

```
"tran":
   {
   "field0" :{ "name"  : "data", "type": "std_logic_vector(15 downto 0)"},
   "field1" :{ "name"  : "cpol", "type" : "std_logic" }
    },
```

- 'data': que contiene los datos a transmitir, de 16 bits de longitud
- 'cpol': indica la polaridad de la señal de reloj que genera. El valor que toma este bit indica qué valor lógico toma la señal de reloj en reposo
- 'valid': presente en todas las transacciones. Encargada de iniciar el movimiento de pines

También, este protocolo va a tener tres salidas:

- SCLK: señal de reloj, necesaria para el sincronismo del maestro y el esclavo
- MOSI (Master Out Slave In): línea por la que se transmiten los datos, en sentido del maestro al esclavo. El número de bits consecutivos que se pueden enviar es arbitrario. Se comienza por el más significativo.
- SS: Señal activa a nivel bajo con la que el maestro indica al esclavo que se active

Además por la línea de datos se envía una cadena de bits, de forma sincronizada a través la señal de reloj. En cada ciclo de reloj el maestro transmite un sólo bit. Además la transmisión de cada bit puede realizarse de cuatro modos diferentes, dependiendo del valor que tomen los bits que se describen a continuación:

- CPOL: define la polaridad del reloj. A efectos prácticos esto de traduce en definir el flanco activo de reloj, si es el de bajada o el de subida. Si CPOL=1, entonces el estado de reposo de esta línea es a nivel alto, y en caso contrario será el nivel bajo.

- CPHA: define la fase del reloj, es decir, en qué instante concreto del periodo del reloj se van a actualizar los datos y en cuál se capturan. Esto quiere decir que si CPHA=0 el dato se captura en el primer flanco de reloj. En caso contrario, en el primer flanco de reloj se actualiza el dato y ya en el siguiente se capturará.

En la siguiente imagen se puede ver la diferencia entre los distintos modos:

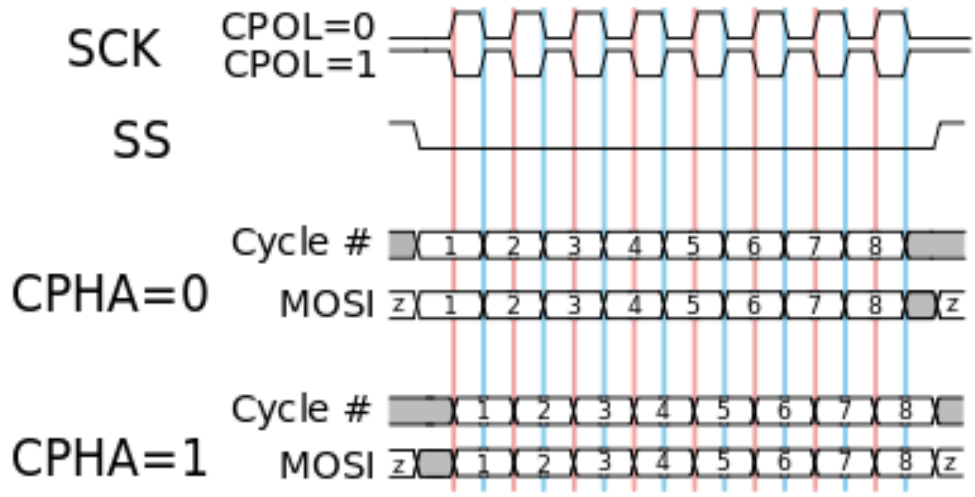


Ilustración 8: Cronograma del protocolo SPI. Fuente: wiki.org [14]

Si CPHA=0, en las líneas rojas se produce la captura del dato (que tenía que estar previamente actualizado), mientras que si CPHA=1, en ese mismo instante el maestro actualiza el dato.

Además, podremos generar los dos modos correspondientes a ambos valores de CPOL, simplemente dándole un valor a este bit desde el testbench, por lo que la descripción de la señal de reloj clk quedará definida en la descripción de la transacción de la siguiente manera:

```
"port1" : {
     "name"   :"SCLK",
     "type"   : "std_logic",
     "values" :
         {
         "val0" :{"val": "input_tran.cpol", "cycles":"SCLK_PERIOD"},
```

```
        "val1" :{"val": "not(input_tran.cpol)", "cycles":"SCLK_CYCLES"},
        "val2" :{"val": "input_tran.cpol", "cycles": "SCLK_CYCLES" },
      ...
```

A continuación se describe una transmisión con este protocolo:

- En primer lugar se activa a nivel bajo la señal SS (Slave Select), para indicarle al esclavo correspondiente que se quiere iniciar la comunicación.
- El maestro va enviando bit a bit los datos a través de MOSI. Cada bit estará activo un periodo de reloj. No hay un número determinado de bits que se puedan o no enviar. El maestro irá enviando hasta completar la transmisión de todos ellos
- La comunicación termina cuando el maestro desactiva la señal SS, es decir, poniéndola a nivel alto. Así se le indica al esclavo que la transmisión ha finalizado.

4. Protocolo ficticio: Como la filosofía de estas implementaciones no es la de generar los protocolos en sí, sino la de comprobar la funcionalidad del código para crear cualquier tipo de driver, vamos a experimentar con un protocolo ficticio.

Los campos que contendrá la descripción de transacción serán uno sólo:

```
"tran":
       {
        "field0" : {"name": "din", "type":"std_logic_vector(31 downto 0)"}
       },
```

- 'din' (Data In): contiene los datos a enviar. Tiene una longitud de 32 bits

En este protocolo debemos crear las siguientes señales de salida:

- 'ena' (Enable): señal de habilitación del protocolo. Es activa a nivel alto y debe permanecer así durante toda la comunicación
- 'startp' (Start Pulse): marca el inicio de la transmisión. Debe estar activa un ciclo de reloj
- 'endp' (End Pulse): marca el fin de la transmisión. Debe estar activa un ciclo de reloj
- 'data': es la señal de salida por donde viajarán los datos. Los datos se envían comenzando

por el bit menos significativo, que asimismo será alojado en el bit menos significativo de la variable de salida. Lo interesante respecto a los protocolos anteriores es que esta señal tiene 4 bits de anchura, por lo que el dato se transmite en bloques de 4 bits.

A continuación se detalla una transmisión completa con este protocolo

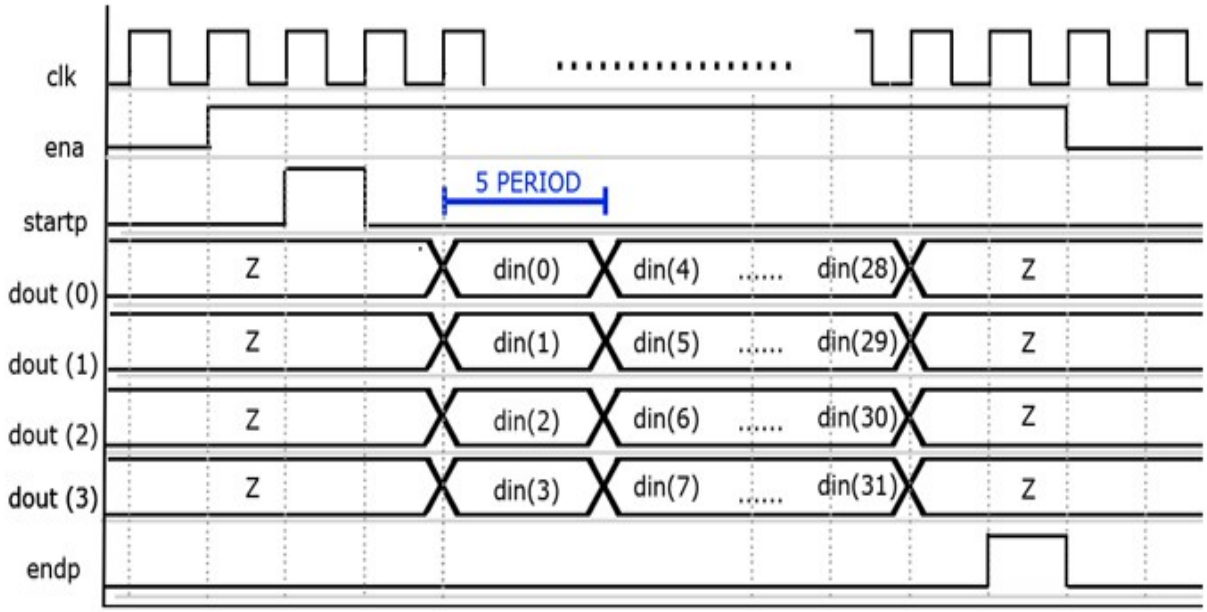


Ilustración 9: Cronograma correspondiente al protocolo ficticio

- En primer lugar se activa la señal 'ena', poniéndola a 1 lógico en un flanco de subida de la señal de reloj.
- En el siguiente flanco activo de reloj se activa a nivel alto la señal 'startp', que estará activa solo un ciclo
- Tras un periodo de reloj de retardo, en el siguiente flanco de subida se transmite el primer bloque de 4 bits en la variable 'dout', que se mantendrán durante 5 periodos de reloj
- Uno tras otro, se irán enviando los bloque de 4 bits, terminando por los más significativos.
- Tras enviar el último dato y después de un periodo de reloj de retardo, en el siguiente flanco activo de reloj se activará la señal 'endp' durante un ciclo de reloj, para indicar que la transmisión ha terminando

- Por último, en el siguiente flanco de subida del reloj, la señal 'ena' se desactiva, poniéndose a 0 y dando por terminada la comunicación.

5. Rotary Encoder: Una vez que se ha comprobado la funcionalidad con protocolos de comunicación de diversa índole, se va a intentar dar un paso más e implementar un sistema diferente.

Se ha pensado en un transductor de posición angular, es decir, un dispositivo electromecánico que convierte el movimiento de rotación en una señal, ya sea analógica o digital. En este caso se trata de adaptar un codificador incremental. La información que debe proporcionar es el sentido de giro y la distancia recorrida. Para ello, genera dos señales distintas de salida: una de referencia y otra, desfasada respecto a la anterior, cuyo desfase será proporcional a la distancia angular a la que se encuentren. Además, dependiendo de la dirección de giro, el desfase se producirá en un sentido o en otro.

Para ofrecer una mayor cobertura de código, las transacciones también implementarán de forma automática los casos de error en ambas líneas de salida. Además, se generarán dos transacciones (y por lo tanto dos drivers diferentes), uno por cada sentido de giro. Sin embargo, el extracto de descripción de transacción que se adjunta es idéntico para ambos casos:

```
"generic" :
    {
    "generic0" : { "name" : "DATA_CYCLES", "type" :"integer", "value":10},
    "generic1" : { "name" : "GAP_CYCLES", "type" : "integer", "value" : 3 }
    },
"tran":
    {
    "field0" :{ "name" : "test_signal","type":"std_logic_vector(7 downto 0)" }
    "field1" :{ "name" : "error_A", "type" : "std_logic" },
    "field2" :{ "name" : "error_B", "type" : "std_logic" }
     },
```

Los campos que contendrá en la transacción que reciba serán los siguientes:

- 'test_signal' : contiene la señal de prueba. Tiene una longitud de 8 bits
- 'error_A': modela que se haya producido un error en la salida de referencia

- 'error_B': modela que se haya producido un error en la señal de salida en la que se produce el desfase respecto a la de referencia.

En este protocolo debemos crear únicamente las dos señales de salida que se han comentado:

- A: Señal de referencia que se inicializa desde el testbench
- B: señal desfasada con respecto a la anterior, dependiendo de la dirección y la velocidad de giro.

A continuación se muestra una figura con la forma que tomarían dichas señales, tanto en sentido horario como en sentido antihorario:

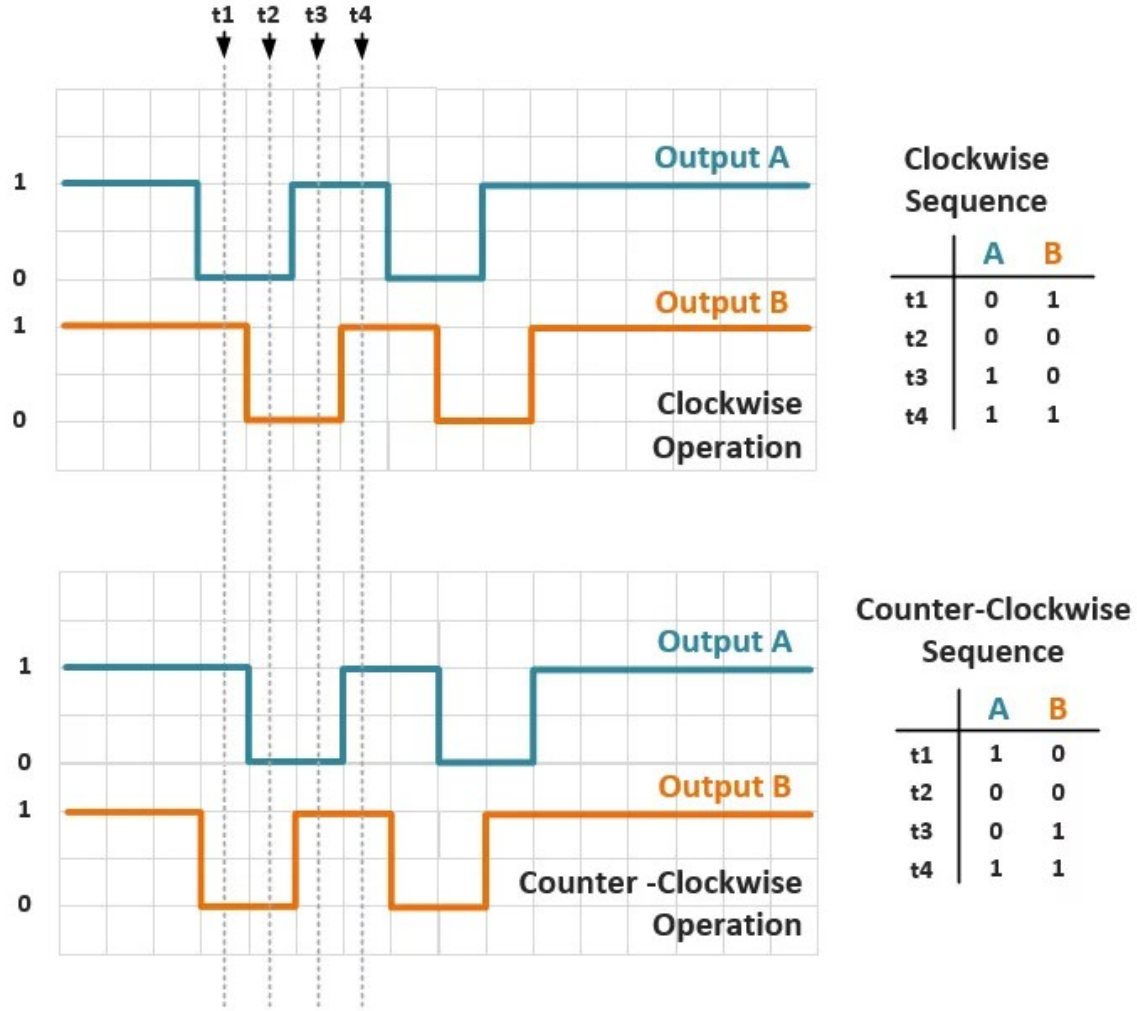


Ilustración 10: Señales de salida del codificador rotatorio [13]

Como se ve en la figura, la señal de salida A es la salida de referencia. La señal B marca el desplazamiento respecto a la primera. Si el transductor gira en el sentido de las agujas del reloj, la segunda señal aparece retrasada respecto a la primera. El desfase entre ambas señales está determinado por la posición angular en la que se encuentre, y como consecuencia, por la velocidad de rotación. Si gira en sentido opuesto, la señal B se adelanta respecto a la primera. Así, siempre se tiene información disponible no sólo de la posición en que se encuentre, sino también del sentido en que está girando. Esta última característica es especialmente relevante, ya que no todos los tipos de codificadores rotatorios cumplen con esta premisa

6.2. Resultados obtenidos

En la sección anterior se establecieron los requisitos que debían cumplir cada uno de los protocolos que se han tratado de implementar. Se han descrito con el objetivo de comprobar con la simulación en el presente apartado si todos ellos se cumplen.

A la hora de hacer las simulaciones, es muy importante tener en cuenta la característica que ya se ha comentado de que hay que ser capaz de generar un número indeterminado de drivers simultáneamente. Esto tiene dos consecuencias directas:

- Hay que tener una estructura jerarquizada y organizada de archivos, para favorecer la comprensión y la experiencia del usuario. Cada protocolo o implementación estará contenido en su propia carpeta, denominada con un nombre representativo. Además cada carpeta contendrá en principio tres archivos: un json con la descripción de la transacción, un testbench para su simulación, y un makefile cuyo fin se explicará más adelante. Hay que tener en cuenta que se generarán diversos archivos en varias etapas de forma automática. Es requisito indispensable que este proceso se realice de forma ordenada, alojando en la carpeta correspondiente los ficheros respectivos.
- Si se van a generar un número indeterminado de código harán falta un número indeterminado de compilaciones y simulaciones. Por lo tanto, según se va incrementando el número de protocolos a utilizar, se va haciendo más necesario automatizar todos estos procesos, con el fin de que el incremento de drivers no suponga un aumento de la complejidad a la hora de simular.

Para paliar los efectos de esta última consecuencia, se han automatizado todos los procesos: compilación en python, análisis y compilación en vhdl, y también la ejecución de los testbench. Todo esto se ha podido realizar gracias al uso de la herramienta make. Para automatizar el proceso completo ha sido necesario crear un archivo Makefile en cada una de las carpetas de la jerarquía de archivos.

Para comprender bien el proceso de automatización a través de la herramienta make, vamos a describirlo paso a paso:

- Inicialmente existen en cada carpeta, además de los Makefile, a los que ya no nos referiremos, una descripción de la transacción almacenada en un fichero json y un

testbench.

- Comenzamos al ejecutar el comando

  `make all`

  que compilará tantas veces como sea necesario el código python, que recibirá como argumento la descripción de la transacción adecuada en cada caso. Generará el driver correspondiente, cada uno alojado en su carpeta respectiva. Así, tras ejecutar esta orden, quedará alojado en cada carpeta el fichero vhd con el driver correspondiente, almacenado con un nombre que lo represente para favorecer la legibilidad.

- A continuación ejecutamos el comando siguiente

  `make test`

  que realiza en primer lugar el análisis sintáctico del driver generado y del test bench, y crea el fichero objeto y la librería de trabajo. A continuación se realiza la elaboración y se genera el ejecutable. Finalmente se ejecuta el testbench y se vuelca la salida en un fichero para poder visualizarlo. Por lo tanto, al finalizar todas las tareas automatizadas en este comando se habrán generado los siguientes archivos en cada una de las carpetas:

  - fichero objeto del driver y del testbench
  - librería de trabajo work
  - archivo ejecutable del testbench
  - archivo vcd para visualizar las formas de onda

- Es necesario dotar a la automatización de la posibilidad de realizar algún cambio en los parámetros, y poder volver a generar los drivers y las simulaciones inmediatamente. Por ello, es necesario ser capaz de borrar todos los elementos que se han generado en las dos etapas anteriores. Tecleando el comando:

  `make clean`

  se borran todos estos ficheros generados de forma automática, volviendo al estado inicial.

Gracias a este sistema de gestión, el usuario podrá realizar cualquier tipo de modificación y realizar un nuevo ciclo completo (compilación en ambos lenguajes y ejecución) de forma inmediata, tecleando tan sólo tres comandos.

A continuación se detallarán algunos resultados obtenidos en las simulaciones, con el fin de demostrar que todos los objetivos que se planteaban en los requisitos se han cumplido con éxito:

1. UART: En este primer ejemplo y con el fin de ilustrar el resultado final de la generación automática de la transacción, se comenzará incluyendo el inicio del driver generado, que se comentará a continuación.

```
-------------------------
--Generated by Gendriver--
-------------------------
library ieee;use ieee.std_logic_1164.all;

package tran is
  type tran_t is record
        data : std_logic_vector(7 downto 0) ;
        valid: std_logic;
  end record;
end tran;

library ieee; use ieee.std_logic_1164.all;
library work; use work.tran.all;

entity driver is
  generic(
      UART_CYCLES : integer := 1 ;
         );
  port(
      clk : in std_logic;
      input_tran: in tran_t;
      tx : out std_logic
      );
end driver;
```

En primer lugar se genera un package con todos los elementos que existieran en el campo 'tran' de la transacción, en este caso sólo existía el byte a enviar (data). Además todas las transacciones incluyen la señal 'valid', encargada de iniciar el movimiento de pines. En este caso, ha sido necesario declarar un generic que se usará para determinar el número de ciclos que permanece un valor en la señal de salida. En el port existirá en todos los protocolos una señal clk y otra de entrada del tipo input_tran, es decir, del package que se incluyó más arriba.

La simulación obtenida es la siguiente:

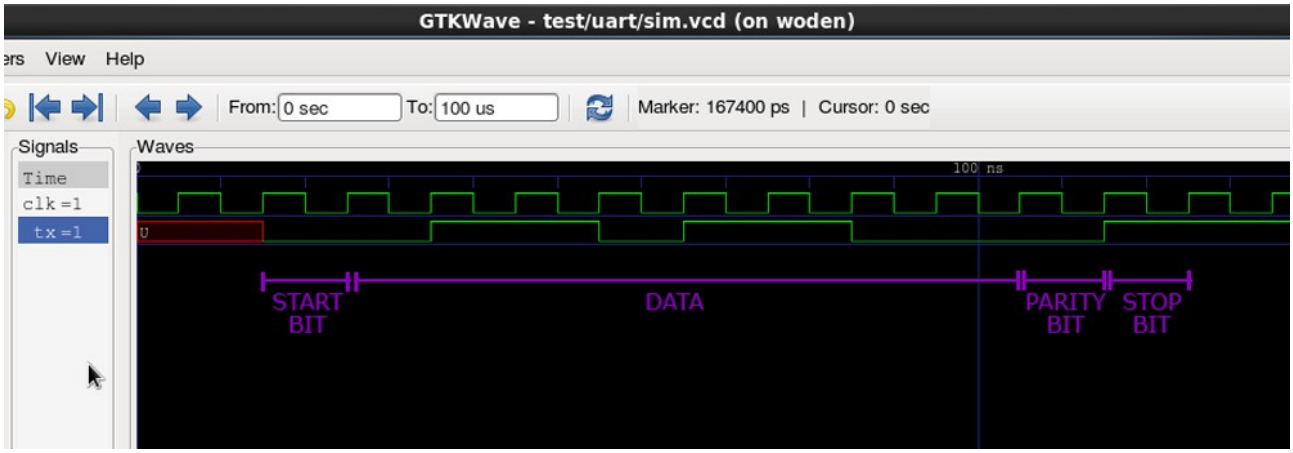


Ilustración 11: Simulación protocolo UART

- Se comprueba que el inicio del movimiento de pines está sincronizado con la señal de reloj del sistema. Todas las señales de salida tomarán un valor indefinido ('U' Undefined) hasta ese momento.
- Todos los cambios de la señal de salida están sincronizados con la señal de reloj.
- Todos los valores son los esperados. El campo 'data' coincide con el valor que se asignó en el testbench, en el sentido correcto. También se comprueba que el bit de paridad es el correcto.
- Tras la transmisión, la línea queda en reposo.

2. I2C: una vez comprobada la funcionalidad básica del programa, capaz de generar el protocolo anterior adecuadamente, se implementó este protocolo con el fin de confirmar nuevas propiedades.

En este caso resulta especialmente interesante la cuestión de la sincronización. La señal de datos no está sincronizada con la señal de reloj que genera el maestro, aunque evidentemente sí lo estará con la señal interna de reloj. Es requisito indispensable que la línea de datos sólo cambie mientras 'scl' está a nivel bajo. Se han declarado dos nuevas constantes a tal efecto. Para ilustrarlo, se adjunta tanto la transacción como la entidad generada aunque evidentemente no aparezcan consecutivas en el driver:

```
package tran is
  type tran_t is record
        slave_address : std_logic_vector(6 downto 0) ;
        reg_address : std_logic_vector(7 downto 0) ;
        data : std_logic_vector(7 downto 0) ;
        valid:std_logic;
  end record;
end tran;
…… …
entity driver is
  generic(
        SCL_CYCLES : integer := 10 ;
        SCL_PERIOD : integer := 20 ;
        SCL_SYNC : integer := 15
  );
  port(
        clk : in std_logic;
        input_tran: in tran_t;
        sda : out std_logic ;
        scl : out std_logic
  );
end driver;
```

Como se implementa la línea en sentido maestro-esclavo, se simulará una operación de escritura básica.. A continuación se muestra el inicio de la transmisión, y se enumeran las propiedades que se han conseguido verificar en esta ejecución.

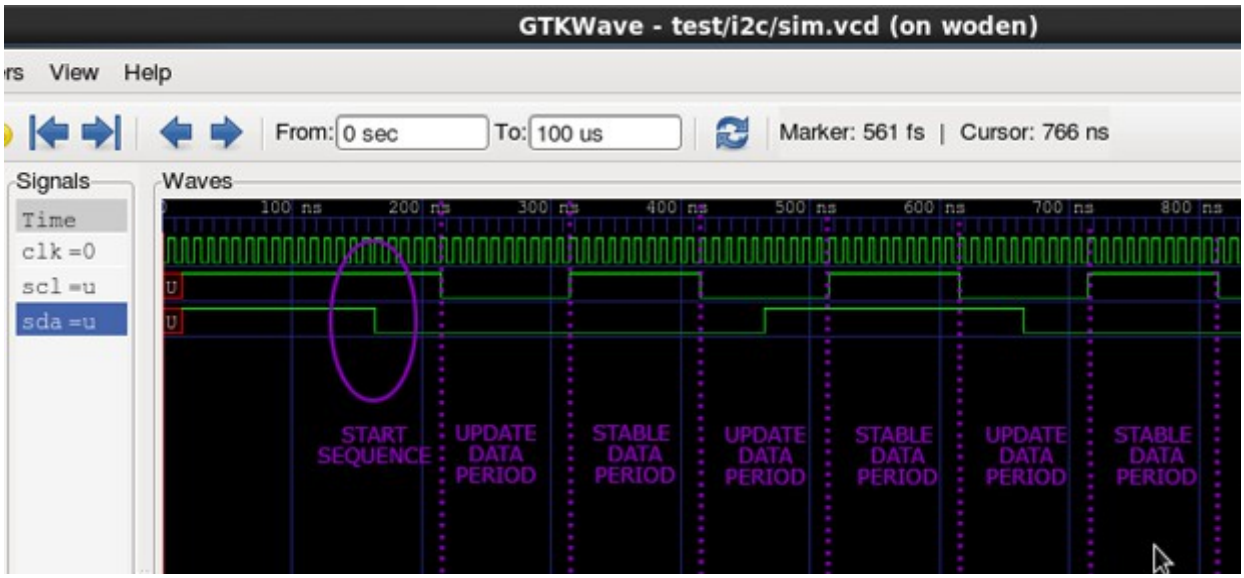


Ilustración 12: Simulación transmisión i2c

- Se comprueba que tanto la secuencia de inicio, que se aprecia en la figura, como la de parada, son las idóneas.
- La línea de datos se mantiene a un nivel constante siempre que la señal 'scl' está a nivel alto. Este requisito es fundamental ya que es cuando se produce la lectura del dato del bus por parte del esclavo. Si no se mantuviera el valor estable, provocaría un fallo en la comunicación.
- Al tratarse de una operación de escritura, al bit de selección de operación (R/W) se le da valor como una constante desde la propia transacción. Al tratarse la operación de lectura de un procedimiento en sentido opuesto, no se contemplará en este documento.
- Se verifica que tanto la dirección del esclavo, como la dirección del registro, así como el dato enviado se corresponden con los datos establecidos en el testbench.
- Los asentimientos que se recibirían por parte del esclavo se modelan como constantes, ya que no se verán modificados en sucesivas ejecuciones. Como ya se ha comentado solo se implementan señales en este sentido.

3. SPI: como ya se ha comentado en el apartado anterior, hay cuatro modos de funcionamiento en este protocolo. Se necesitan dos transacciones diferentes para generar los dos modos correspondientes a CPHA=0, o a CPHA=1. Hay que recordar que la única diferencia entre ambos es que en el segundo modo la señal está retrasada medio periodo de reloj con respecto al primero. Por lo que una leve modificación en la transacción nos permite cambiar de un modo a otro. En esta ejecución se contempla el caso en el que CPHA=1, es decir, el dato se carga en la señal MOSI con el primer flanco de reloj, y no antes.

La transacción y la entidad generada en este caso, ambas de forma automática son:

```
package tran is
  type tran_t is record
        data : std_logic_vector(15 downto 0) ;
        cpol : std_logic ;
        valid:std_logic;
  end record;
end tran;
    ...
entity driver is
  generic(
        SCLK_CYCLES : integer := 10 ;
        SCLK_PERIOD : integer := 20
  );
  port(
        clk : in std_logic;
        input_tran: in tran_t;
        mosi : out std_logic ;
        SCLK : out std_logic ;
        ss : out std_logic
  );
end driver;
```

A continuación se muestran dos simulaciones para sendos valor de CPOL. Ambas se consiguen rápidamente, con el simple hecho de cambiar el valor de un bit en el testbench.

Ilustración 13: Inicio de la transmisión SPI. Imagen superior CPOL=1. Imagen inferior CPOL=0

- La simulación mostrada en la parte de arriba se corresponde con un valor de CPOL=1, que marca el estado de reposo de la señal de reloj que genera el maestro 'sclk'. Se puede comprobar que la única diferencia entre las dos simulaciones es que la señal de reloj aparece invertida, pero el comportamiento del resto de señales es el mismo. Además se comprueba que el resultado es el correspondiente al valor del bit CPHA adecuado, ya que el valor se carga en la línea de datos con el primer flanco de la señal 'sclk'.
- Se verifica que el dato enviado por la línea MOSI se corresponde con el establecido en el testbench
- El bit 'ss' marca el inicio y el fin de la comunicación de la forma esperada.

4. PROTOCOLO FICTICIO: con el fin de verificar funcionalidades que no se han implementado hasta el momento, se plantea realizar un protocolo donde se pueda escoger los parámetros libremente.

La principal novedad que aporta es que los bits se transmiten en bloques de 4 bits. Esto requiere manejos de otro tipo de variables que las vistas hasta ahora. Las señales de control se establecen en este caso activas a nivel alto. El estado de reposo de las líneas de datos se establece en alta impedancia.

La transacción y la entidad generada en este caso, ambas de forma automática son las siguientes:

```
package tran is
  type tran_t is record
        din : std_logic_vector(31 downto 0) ;
        valid:std_logic;
  end record;
end tran;
...
entity driver is
  generic(
        CLK_PERIOD : integer := 1 ;
        DATA_PERIOD : integer := 5
  );
  port(
        clk : in std_logic;
        input_tran: in tran_t;
        dout : out std_logic_vector(3 downto 0) ;
        ena : out std_logic ;
        startp : out std_logic ;
        endp : out std_logic
  );
end driver;
```

A continuación se muestra el inicio y el fin de la transmisión:

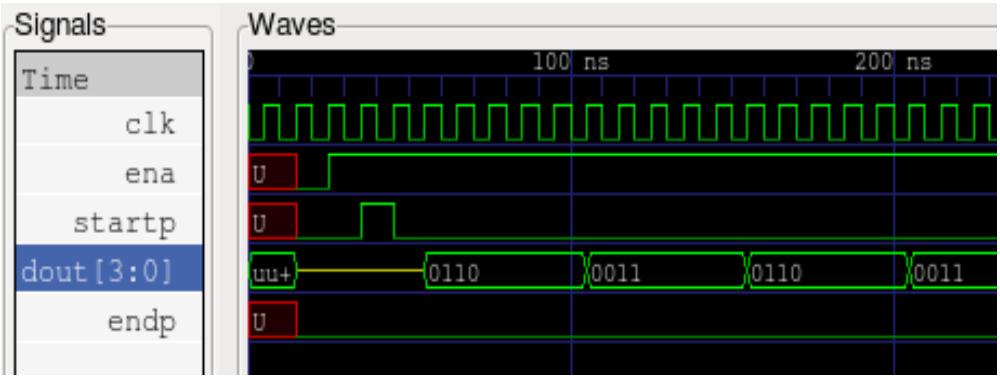


Ilustración 14: Protocolo ficticio. Inicio de la transmisión

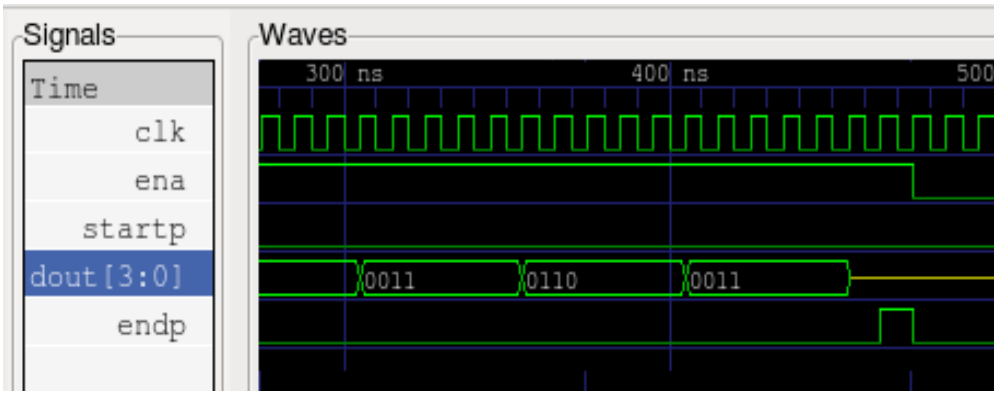


Ilustración 15: Protocolo ficticio. Fin de la transmisión.

- Se comprueba que la señal de enable marca el inicio y el fin de la transmisión, quedando a nivel alto durante el transcurso de la misma.
- Se verifica que los pulsos de comienzo y fin están sincronizados correctamente y que ambos tienen la duración de un periodo de reloj.
- La línea de datos permanece en alta impedancia cuando está inactiva. Además se comprueba que los bits enviados se corresponden con los inicializados en el testbench, en el orden que se esperaba. Se confirma que cada dato está activo en la línea durante 5 ciclos de reloj.

5. ROTARY ENCODER: en este caso se generarán dos tipos de transacciones diferentes, una para simular el giro en sentido horario y otra para el giro antihorario. Además también se va a modelar el comportamiento anómalo de alguna de las dos señales. Veamos que la declaración del package y la entidad es idéntica para las dos transacciones que se han descrito. Sólo diferirán entre ellas en la forma de sacar los datos por las señales A y B.

```
package tran is
  type tran_t is record
        test_signal : std_logic_vector(7 downto 0) ;
        error_A : std_logic ;
        error_B : std_logic ;
        valid:std_logic;
  end record;
end tran;
...
entity driver is
  generic(
        DATA_CYCLES : integer := 10 ;
        GAP_CYCLES : integer := 3
  );
  port(
        clk : in std_logic;
        input_tran: in tran_t;
        A : out std_logic ;
        B : out std_logic
  );
end driver;
```

A continuación se muestra una captura del inicio del resultado de la simulación:

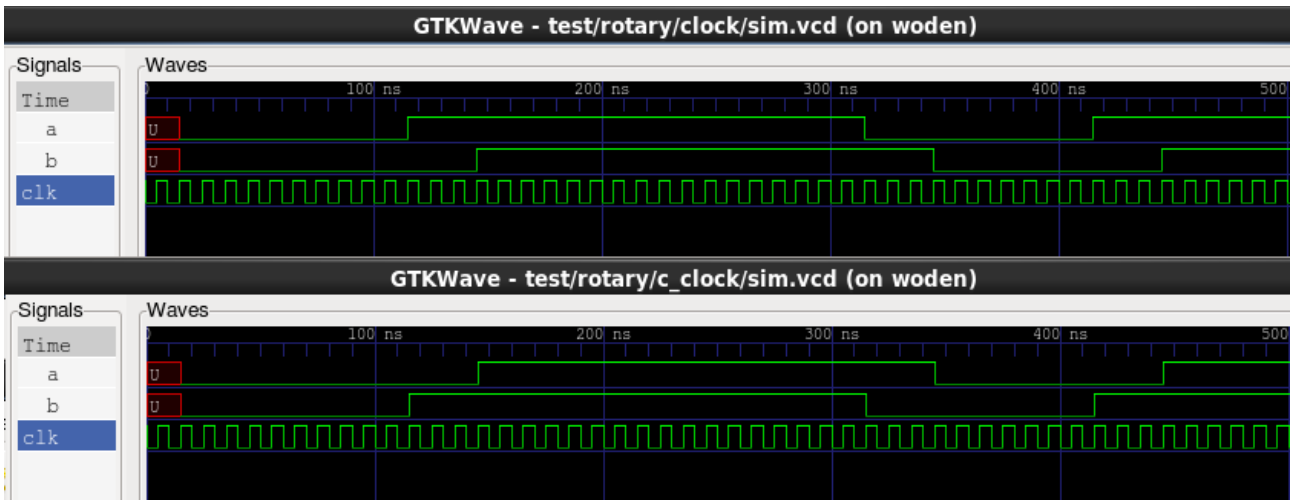


Ilustración 16: Codificador rotatorio. Imagen superior: sentido horario. Imagen inferior: sentido antihorario

- Como se ve en la figura, ambas señales de salida aparecen con un cierto desfase entre ellas. El signo del desfase entre ambas señales será positivo o negativo dependiendo del sentido de giro. En la imagen superior la segunda señal aparece retrasada con respecto a la primera, correspondiente al giro en sentido horario. En la imagen inferior, la segunda señal aparece adelantada respecto a la primera, lo que se corresponde con el giro en sentido antihorario. Cada sentido de giro tiene su propia descripción de transacción asociada.
- El desfase generado entre ambas señales debe ser un valor configurable, ya que se trata de modelar el transductor de la manera más completa posible. Por ello, debe declararse desde la transacción. Para esta simulación se ha tomado dicha cantidad como un genérico, ya que la hemos considerado una constante para cada ejecución.
- Además se han implementado los posibles errores que pudieran ocurrir en las líneas de salida. Para ello se han creado sendos campos en la transacción, a los que se le puede dar el valor correspondiente desde el testbench. Modelaremos dichos errores manteniendo una de las señales a un valor constante, simulando por ejemplo un error de conexión. En la siguiente figura muestra ambos casos.

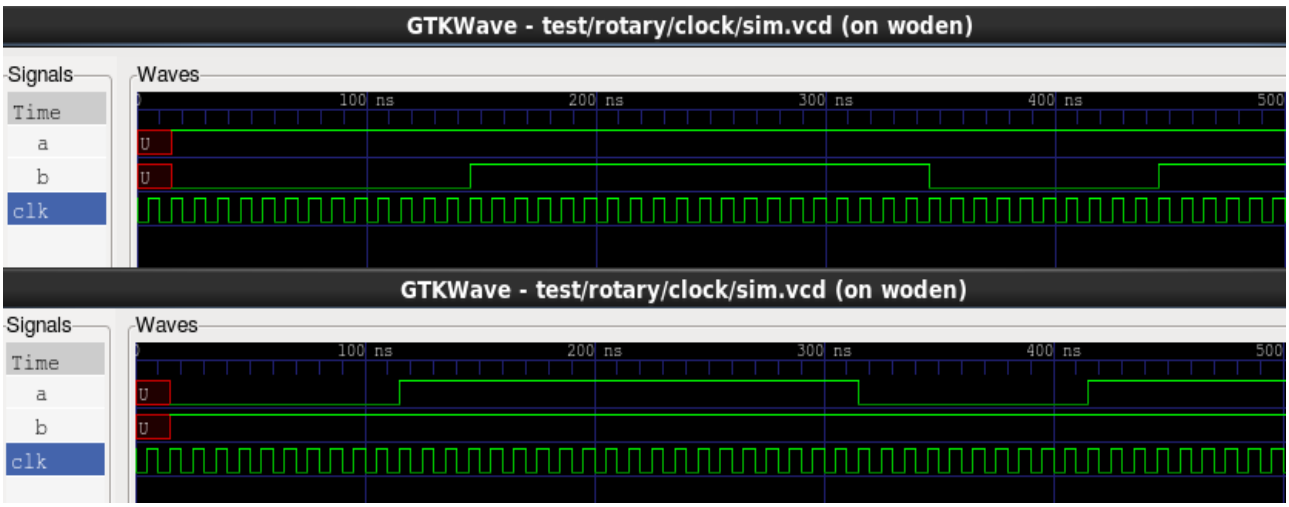


Ilustración 17: Simulated error signals in rotary encoder

- Hay que recordar que basta cambiar el valor de un bit en el testbench para generar una transacción con o sin error. En este caso vemos que una señal no reproduce a la otra. A pesar de tratarse de un comportamiento indeseado, los resultados obtenidos en la simulación son los esperados.

-

7. CONCLUSIONES Y LÍNEAS FUTURAS

7.1 Conclusiones

Se ha implementado un generador automático de protocolos capaz de generar con cada ejecución, el driver correspondiente para el manejo de las señales adecuadas. El compromiso que se adquirió en la descripción del alcance de este mismo documento fue el de conseguir con una misma pieza de código la generación totalmente automática de protocolos de muy diversa naturaleza. Todo esto se consigue con el mero hecho de modificar la descripción de la transacción, lo que nos permite elevar nuestro nivel de abstracción. Además se limitaría a generar una comunicación unidireccional partiendo del maestro, ya que se trata de una incursión innovadora cuyo resultado satisfactorio abrirá nuevas posibilidades.

Los primeros prototipos se realizaron el lenguaje C. Aunque las primeras pruebas fueron favorables, al tratar de verificar el código con nuevos tipos de transacciones, comenzaron a surgir las limitaciones propias de un código poco flexilble. Un ejemplo claro de que la elección de python ha sido la mejor opción posible es el código necesario para convertir la descripción de la transacción, proveniente de un archivo json. En lenguaje C no resultó satisfactoria la elección de ninguna librería ya existente, por lo que se escribió un código específicamente para ello. Este código, al probar distintos formatos de transacción generaba complicaciones, no irresolubles pero sí innecesarias. Todos estos problemas en python se solucionaron con una sola línea de código. Obviamente, no sólo la reducción de código es considerable, sino también el ahorro de tiempo y complicaciones, además de la flexibilidad que aporta este lenguaje.

Se comenzó implementando los protocolos más básicos y de uso generalizado de comunicación serie. El primero fue el UART, por tratarse del más simple. Una vez que se comprobó que el código respondía satisfactoriamente se realizó la prueba con protocolos ligeramente más complejos, como I2C o SPI. En ambos casos el resultado fue el esperado, como ha demostrado con las simulaciones del apartado anterior.

Como se trata de demostrar las garantías que ofrece el código que se presenta, a continuación se implementó un protocolo ficticio, con el objetivo de poder modificar parámetros que hasta ahora no se habían probado. Por ejemplo, que se envíen los datos por bloques de un número determinado

de bits en lugar de por una sola línea.

Además se ha considerado interesante verificar el código propuesto con una implementación que no resultara ser un protocolo en sí, ya fuese real o ficticio. En este caso se ha sugerido un transductor de posición angular o codificador rotatorio (Rotary Encoder). Al poder generarse también este tipo de sistema, queda más que comprobado la amplia adaptabilidad de la solución propuesta. Los resultados tan satisfactorios que se han obtenido abren un abanico de posibilidades a desarrollar.

7.2. Líneas futuras

Después del resultado tan favorable que se ha logrado, sería muy interesante extender la misma filosofía a la generación automática de un monitor de protocolos, que realizaran la operación inversa. Se trataría en ese caso de traducir el movimiento de pines en una transacción de salida más comprensible. Esto nos permitiría elevar el nivel de abstracción de forma completa, teniendo que manejar únicamente transacciones de entrada y salida de más alto nivel, consiguiéndose aislar por completo la ardua tarea del movimiento de pines.

Otra solución más a corto plazo para comprobar que el movimiento de pines que se está realizando es el adecuado podría consistir en incluir en VHDL el uso de las notificaciones. Éstas permiten advertir si las señales correspondientes han sido activadas, o incluso comprobar si toman un valor determinado. En caso de comportamiento indebido , se muestra un mensaje por pantalla con el error correspondiente.

Ahora que se ha demostrado la viabilidad de la solución propuesta y una vez se ha garantizado la adaptabilidad y la funcionalidad de este código, resultaría muy interesante implementar protocolos mucho más complejas. Para ello sólo se requeriría completar la descripción de transacciones, sin más requisito que seguir la estructura adecuada.

En conclusión, los resultados tan satisfactorios que se han obtenido con las distintas simulaciones que se han realizado, permiten abrir un abanico de posibilidades de desarrollo para trabajos futuros.

BIBLIOGRAFÍA

[1] M. Sedghi, A. Shahabi, Z. Navabi, “TLM Studio for Transaction Level Simulation and Synthesis”, University of Tehran, 2009.

[2] R. Salemi. “Evolving FPGA Verification Capabilities”, Verification Academy. Mentor Inc.

[3] J. Bergeron, E. Cerny, A. Hunter, A. Nightingale, ¨Verification Methodology Manual for SystemVerilog¨ , Ed. Springer, 2006

[4] Cadence. Open verification methodology. Cadence Design Inc.

[5] Accellera. Universal verification methodology. Accellera Organization Inc.

[6] IEEE std. 1800-2017, IEEE Standard for SystemVerilog--Unified Hardware Design, Specification, and Verification Language, 2017

[7] Synopsis VC Verification IP. Synopsys Inc. https://www.synopsys.com/verification/verification-ip.html

[8] Synopsis Press Releases. Synopsis Inc. Jan, 2018. https://news.synopsys.com/2018-01-31-Synopsys-Announces-Verification-IP-and-Test-Suite-for-Arm-AMBA-ACE5-and-AXI5

[9] Cadence VIP, Cadence Inc. https://ip.cadence.com/ipportfolio/verification-ip

[10] Mentor Verification IP, Mentor Inc. https://www.mentor.com/products/fv/verification-ip

[11] UVVM Github, https://github.com/UVVM/UVVM

[12] E. Tallasken, “Universal VHDL Verification Methodology” https://www.linkedin.com/pulse/universal-vhdl-verification-methodology-2-years-espen-tallaksen

[13] T.Youngblood, “How to use a rotary encoder in a MCU-based project” https://www.allaboutcircuits.com/projects/how-to-use-a-rotary-encoder-in-a-mcu-based-project/

[14] https://en.wikipedia.org/wiki/Serial_Peripheral_Interface_Bus

ANEXO:

MÓDULO GENDRIVER. CÓDIGO PYTHON

```
#Generador automatico de drivers en VHDL
import sys

import simplejson as json
from collections import OrderedDict
from pprint import pprint

#Se comprueba si el num.de argumentos es correcto. Si lo es,
#Carga el contenido de la descripcion de la transaccion en la variable data

if(len(sys.argv) != 2):
    print sys.exit('Error:Usage: gentran file.json')
else:

        with open(sys.argv[1]) as f:
                data = json.load(f, object_pairs_hook=OrderedDict)


        print '--------------------------'
        print '--Generated by Gendriver--'
        print '--------------------------'
        print ' '
        print 'library ieee;'
        print 'use ieee.std_logic_1164.all;'
        print ' '

#Imprime todos los elementos del campo tran
        if 'tran' in data:
                print 'package tran is'
                print '  type tran_t is record'
                for element in data['tran']:
                        print ' ',data['tran'][element]['name'], ':', data['tran']
[element]['type'], ';'
                print ' valid:std_logic;'
                print '  end record;'
                print 'end tran;'
        else:
                sys.exit('Error: mandatory tran field not found in input file')

        print ' '
        print 'library ieee;'
        print 'use ieee.std_logic_1164.all;'
        print 'library work;'
        print 'use work.tran.all;'
        print' '
```

```
          print'entity driver is'

  #Imprime todos los elementos del campo generic
  # Para eliminar el ';' del ultimo generic:
  #   1.- Se elimina el 'ultimo elemento de la lista de genericos
  #   2.- Imprime el resto de los elementos, añadiendo el ';'
  #   3.- Imprime el ultimo elemento, sin el ';'

  if 'generic' in data:
                  lastkey, lastelement = data['generic'].popitem(last=True)
                  print '  generic('
                  for element in data['generic']:
                          print ' ', data['generic'][element]['name'], ':',
data['generic'][element]['type'], ':=', data['generic'][element]['value'], ';'
                  print ' ', lastelement['name'], ':', lastelement['type'], ':=',
lastelement['value']

                  print '  );'

          print '  port('
          print ' clk : in std_logic;'
          print ' input_tran: in tran_t;'

  #Imprime todos los elementos del campo interface
  #Elimina el ';' del ultimo elemento de la lista

          if 'interface' in data:
                  lastkey, lastelement = data['interface'].popitem(last=True)
                  for element in data['interface']:
                          print ' ',data['interface'][element]['name'], ': out',
data['interface'][element]['type'], ';'
                  print ' ', lastelement['name'], ': out', lastelement['type']

          print '  );'
          print 'end driver;'
          print' '

    print'architecture transactional of driver is'

          if 'constant' in data:
                  for element in data['constant']:
                          print '  constant', data['constant'][element]['name'], ':',
data['constant'][element]['type'], ':=', data['constant'][element]['value'], ';'
```

```
        print 'begin'
        print' '

        if 'interface' in data:
                for element in data['interface']:
                # para cada puerto de la interfaz un proceso diferente
                        print data['interface'][element]['name'] + str('_proc: process')
                        print 'begin'
                  #sincroniza con la señal de reloj para iniciar el movimiento de pines
                        print " if (input_tran.valid='1')then"
                        print '         wait until rising_edge(clk);'
                        #imprime todos los valores correspondientes a un puerto
                        for value in data['interface'][element]['values']:
                              print '\t', data['interface'][element]['name'], '<=',
data['interface'][element]['values'][value]['val'], ';'
                                  print 'for i in 1 to', data['interface'][element]
['values'][value]['cycles'], 'loop'
                                  print 'wait until rising_edge(clk);'
                                  print 'end loop;'
                         print ' else'
                         print '     wait until rising_edge(clk);'
                         print ' end if;'
  print 'end process;'

                #repite el proceso con el ultimo elemento
                print lastelement['name'] + str('_proc: process')
                print 'begin'
                print " if (input_tran.valid='1')then"
                print '         wait until rising_edge(clk);'
                for value in lastelement['values']:
                        print '\t', lastelement['name'], '<=', lastelement['values']
[value]['val'], ';'
                        print 'for i in 1 to', lastelement['values'][value]['cycles'],
'loop'
                        print 'wait until rising_edge(clk);'
                        print '             end loop;'
                print ' else'
                print '    wait until rising_edge(clk);'
                print ' end if;'
                print 'end process;'


        print 'end transactional;'
```