Soporte para depuración en *FPGA*

Autor: Christian Meléndez Núñez

Director: Daniel Jiménez González

Departamento de Arquitectura de Computadores

Grado en Ingeniería informática

Mención en Ingeniería de computadores

Facultat d'Informàtica de Barcelona (FIB)

Universitat Politècnica de Catalunya (UPC) – BarcelonaTech

Fecha: 04-04-2018

Resumen/abstract

Describiremos la implementación de un soporte a la depuración llamado **hgdb** basado en la idea que sigue *LegUp*, un entorno de depuración orientado a la síntesis de alto nivel (HLS) que permite inspeccionar circuitos generados por *HLS* de la misma manera que GDB (utilizando *breakpoints*, ejecuciones paso a paso y permitiendo consultar el valor de las variables de código en alto nivel). Se proporciona un entorno sencillo que permite depurar el hardware paso a paso y comparar su funcionamiento al del programa en alto nivel, con tal de ver en qué momento el hardware no genera los mismos resultados que la versión del programa en software.

We describe the implementation of an FPGA debugging support called **hgdb** which is based on *LegUp*'s idea, an HLS-oriented debugging environment that allows to inspect circuitry generated by *HLS* the same way as *GDB* does (by using breakpoints, step-by-step execution and allowing to read the variables on the high-level version of a program). We provide a simple environment that enables the user to perform step-by-step hardware debugging and compare the result to the generated by the high-level version, making it possible to detect discrepancies between both versions.

Índice

Tablas

Figuras

A. Alcance del proyecto y contextualización

1. Contextualización

1.1. Marco conceptual

1.1.1. FPGA

Una *FPGA* [1] (de Field-Programmable Gate Array) es un dispositivo prefabricado que puede ser programado eléctricamente para que funcione como casi cualquier tipo de sistema o circuito digital.

Éstas ofrecen una serie de tentadoras ventajas respecto a los circuitos de función fija *ASIC* (**A**pplication **S**pecific **I**ntegrated **C**ircuit). Se suelen tardar meses en fabricar un *ASIC* y su coste gira en torno a los cientos de miles y/o millones de dólares; las *FPGA* se configuran en menos de un segundo (y asimismo pueden ser reconfiguradas si se comete un error) y cuestan desde unos pocos dólares hasta unos pocos miles.

La naturaleza flexible de las *FPGA* se consigue a expensas de un coste mayor por área, un mayor retardo de señales y una mayor potencia consumida.

Una *FPGA* consiste en una matriz de bloques lógicos programables de distintos tipos, incluyendo lógica general, memoria y bloques multiplicadores, rodeados de mecanismos que permiten programar la interconexión de estos bloques. Esta misma matriz está rodeada por bloques de E/S programables que conectan el chip al mundo exterior. El término “programable” indica la capacidad de programar la *FPGA* después de su fabricación. Esta personalización es posible debido a la tecnología usada para programarlas, la cual permite cambiar el comportamiento del chip prefabricado.

1.1.2. Historia reciente de la programación lógica

Los orígenes de las *FPGA* van ligados al desarrollo de los circuitos integrados a principios de los 60. Las *"cellular arrays"* consistían en una matriz bidimensional de celdas lógicas dotadas de comunicación fija punto a punto.

- 1967: se introduce la primera *FPGA* basada en memoria estática. Su arquitectura permitía configurarla e interconectar sus bloques usando un conjunto de bits de configuración. A diferencia de sus contrapartes contemporáneas basadas en *"cellular arrays",* la *E/S* y el almacenamiento podían implementarse en cualquier celda lógica. Además, las conexiones intercelda podían cambiarse fácilmente, lo que permitía la implementación de una gran variedad de circuitos. No obstante, aunque la memoria estática ofrece un enfoque más flexible en cuanto a la programación de estos dispositivos, requiere un incremento significativo en área por unidad programable en comparación a las implementaciones en memoria *ROM.*

- Años 70: se introdujeron dispositivos programables basados en memorias *ROM.* Los primeros *PLA* (Programmable Logic Array) usaban planos lógicos AND-OR de dos niveles que encajan con la estructura de funciones lógicas comunes y son significativamente más eficientes en área.

- Año 1984: La primera *FPGA* moderna fue introducida por *Xilinx*. Contenía la actualmente clásica matriz de bloques lógicos configurables. Desde la primera *FPGA*, que contenía 64 bloques lógicos y 58 entradas y salidas, la complejidad de las *FPGA* ha crecido enormemente. Actualmente pueden contener aproximadamente 330.000 bloques lógicos configurables y 1000-1100 entradas y salidas, además de un gran número de bloques más específicos que han expandido notablemente las capacidades de las *FPGA.*

1.2. Estado del arte

La versatilidad de las *FPGA* ha contribuido a que se utilicen en el ámbito de la computación de altas prestaciones (*HPC*, High-Performance Computing). Recientemente, ha habido iniciativas tanto por parte de empresas como del ámbito académico para explorar la aceleración *hardware* de determinadas aplicaciones mediante *FPGA* en plataformas de altas prestaciones, dado que los supercomputadores basados en *clusters* con *CPU*'s genéricas no llegan a cubrir la creciente demanda de aplicaciones que explotan al máximo la potencia computacional, y ello es debido a limitaciones en consumo de energía y costes. La investigación ha demostrado que un sistema heterogéneo basado exclusivamente en *FPGA* que utiliza una combinación de diferentes tipos de nodos de computación (incluyendo procesadores embebidos y aceleradores *hardware* específicos para determinadas aplicaciones) es una forma escalable de usar *FPGA* en el ámbito de la computación de altas prestaciones.

No obstante, aunque el *hardware* especializado tiene el potencial de proporcionar la aceleración de programas y reducir el consumo de energía, el principal inconveniente reside en su diseño. Es necesario que el diseñador especifique las funcionalidades del *hardware* a un muy bajo nivel de abstracción en el que se especifica completamente el comportamiento ciclo a ciclo de ese diseño. El uso de estos lenguajes requiere avanzados conocimientos en *hardware*, lo cual lleva a procesos de desarrollo más largos que pueden influir negativamente en el tiempo que se tarda en sacar un producto al mercado. Por otra parte, y como dato importante, depurar circuitos puede resultar tedioso debido a que los errores, en los peores casos, pueden ser casi indetectables (debido a la gran cantidad de señales a analizar, a que los errores puedan deberse al mal funcionamiento de más de una señal, o simplemente a que se produzcan durante un muy corto lapso de tiempo). También cabe destacar el hecho de que los programas generadores de *hardware* se muestran ante los usuarios como cajas negras, en el sentido de no poder saber de qué manera lo generan y por consiguiente cómo optimizar ese proceso.

Una solución interesante para lograr esa heterogeneidad y, al mismo tiempo, solventar el problema de la falta de tiempo al poner un producto en venta, es la combinación de *FPGA's* y herramientas de síntesis de alto nivel (*HLS*, *High-Level Synthesis*). Las herramientas de *HLS* parten de un código escrito en un lenguaje de programación de alto nivel (C, C++…) y generan la especificación de un circuito en *HDL* que realiza la misma función.

1.3. Actores

La *HLS* ofrece ventajas para los **ingenieros de software**, permitiéndoles aprovecharse de los beneficios de la aceleración *hardware* sin ser necesariamente expertos en la materia. También son ventajas para los **ingenieros de *hardware***, pues pueden permitirse diseñar sistemas más rápidamente y desde un nivel de abstracción superior.

2. Objetivo y alcance del TFG

Nuestro objetivo consiste en **implementar un sencillo depurador HLS, junto con la *API* e infraestructura necesarias para su posterior desarrollo y extensión.** Se partirá de un software de síntesis de alto nivel existente, utilizado para simular diseños *hardware* y cargarlos en una *FPGA*, y se desarrollará un soporte inspirado en funcionalidades de depuración ya disponibles por parte de la misma herramienta.

El entorno/software en el que nos basaremos se llama *LegUp* [2][3], una herramienta de *HLS* (ver apartado 2.2) de código abierto implementada por la Universidad de Toronto. Su entorno de depuración, denominado *Inspect*, ofrece una perspectiva *software* a la hora de depurar. Permite, mediante una interfaz gráfica, desplazarse a través del código fuente en C, establecer puntos de corte (*breakpoints*), inspeccionar variables y más cosas. Se permiten tres modos de ejecución [4][5]:

-Simulación del *RTL* producida mediante herramientas de síntesis de alto nivel

Se lleva a cabo mediante *ModelSim*; en este caso se analiza el *hardware* ciclo a ciclo y se extrae el valor de las señales.

-Ejecución del mismo *hardware* en una *FPGA* (depuración en silicio)

Se utiliza *SignalTap* para extraer los valores de las señales. La depuración en silicio permite analizar errores que son invisibles a nivel *RTL*, como por ejemplo: errores debidos a tiempos, errores de ejecución pasajeros, errores en la interfaz entre un módulo *hardware* generado por *HLS* y otros módulos en el sistema, o incluso errores en la síntesis *RTL*, mapeo y herramientas de enrutamiento del fabricante.

-Modo híbrido

Mezcla de los anteriores. Se simulan ciertas partes del código en *ModelSim* y el resto se ejecutan en la FPGA.

Además, se proporcionan herramientas de depurado estrecha y únicamente relacionadas con el contexto *HLS*:

-La capacidad de ver la relación entre el código fuente y el *RTL* generado mediante *HLS* (relacionando líneas de código en alto nivel y líneas de *Verilog*). La base de datos de depuración [6] contiene la relación entre tres vistas del programa/circuito a diferentes niveles de abstracción:

1) El código fuente en C
2) *IR* de *LLVM*
3) *Verilog*

-Detección de discrepancias *SW*/*HW*, mediante la cual *Inspect* se comunica con procesos en ejecución de ***gdb*** [7] y ***ModelSim*** para identificar, en tiempo de ejecución, el punto en que la ejecución software no corresponde a la ejecución *hardware*. Por lo tanto, tenemos detección de discrepancias entre la simulación *RTL* y su implementación en silicio, correlacionadas con el código fuente original en C.

La parte que desarrollaremos contempla las siguientes funcionalidades:

- Comunicación con *GDB* para depurar la versión en alto nivel de un programa
- Comunicación con la *FPGA*
- Depuración de la versión en circuito junto con la versión en alto nivel del programa

Mientras que la parte que dejamos abierta es la de compilación de código en C y generación de la versión especificada en *Verilog.* Lo que se pretende con este trabajo es crear una herramienta de *HLS* aprovechando conocimientos tanto de *hardware* como de software (ya que la *HLS* mezcla distintas áreas dentro de la ingeniería informática). Teniendo en cuenta que en el ámbito de la computación de altas prestaciones son muchas las disciplinas que aprovechan la potencia computacional de supercomputadores para realizar infinidad de cálculos, resulta positivo que existan aplicaciones que permitan brindar un extra de rendimiento sin que sean necesarios conocimientos profundos en materia de ingeniería de *hardware*.

3. Metodología

La metodología de trabajo no seguirá un procedimiento demasiado complejo, tratando tan sólo de compilar las pequeñas partes del depurador que pudieran sufrir modificaciones y a continuación haciendo uso de nuestro método de validación.

El método de validación consiste en utilizar ejemplos implementados por nosotros mismos que servirán para comprobar que el depurador funciona correctamente. Dado que no implementamos la compilación y generación de diseños en *HDL*, crearemos códigos en C y también sus equivalentes en circuito. A la hora de probar nuestro entorno, tan solo tendremos que utilizar estos ejemplos para probar una depuración completa.

Una posible herramienta de seguimiento a utilizar es *GitHub*, mediante la creación de un repositorio al que se vayan subiendo los cambios en el código fuente de manera progresiva. De esta manera, se podrá tener constancia de los últimos cambios en el caso de que surgiera un error inesperado durante la ejecución del entorno, y así actuar en consecuencia.

B. Planificación temporal

4. Fases del proyecto

Hemos subdividido el proyecto en tres fases bien definidas, que como posteriormente veremos siguen una relación de precedencia entre ellas.

- Documentación e investigación

  Antes de implementar nuestro producto es necesario saber con se va a trabajar, estudiar su funcionamiento y averiguar qué posibilidades ofrece.

- Implementación

  Proceso durante el que se espera implementar un código que permita una depuración sencilla (principalmente el código del depurador y también los ejemplos que permitirán comprobar su correcto funcionamiento).

- Pruebas

  Será necesario reunir evidencias de que la implementación realizada proporciona ese soporte que esperábamos.

5. Planificación y organización

5.1. Fases del proyecto

Valor proyecto	Ciclo de vida			
		Documentación e investigación	**Implementación**	**Prueba**
	Tiempo	Desde febrero hasta septiembre de 2017 = **60 horas efectivas**	40 días laborables x 3 horas/día = **120 horas efectivas**	18-20 días laborables x 3 horas/día = **60 horas efectivas**
	Calidad	Artículos de investigación	Aprovechar todo el código ya implementado posible	Probar todos los tests que hemos generado
	Riesgos		Encontrar errores en *hgdb* cuya complejidad dificulte terminar de desarrollar el soporte elegido	Que no se pasen el 50% de los test o más

Tabla 1. Matriz de identificación de requerimientos

Nótese que la fase de documentación e investigación ha sido llevada a cabo previamente a la redacción de este documento.

Las distintas fases del proyecto quedan ordenadas acorde al orden que se les ha dado en el apartado anterior:

1. Documentación e investigación

2. Implementación

3. Pruebas

La relación de dependencia entre todas estas fases es estrictamente de precedencia. Si bien la consecución de las distintas fases implicará revisar documentación continuamente y/o reimplementar ciertas partes del código (o, en el peor de los casos, replantearse la actual implementación), no concebimos estos saltos hacia atrás en las fases como algo que necesariamente deba suceder. Por esa razón nos ceñimos a la definición de relación de precedencia como forma en que están conectadas las distintas fases del proyecto.

5.2. Tiempo y duración

Clasificamos las fases del proyecto según dificultad. Los tiempos expresados en la tabla anterior se basan en los siguientes razonamientos:

- Documentación e investigación

  Se parte de un conocimiento y experiencia en *FPGA* que consideraremos intermedios. El conocimiento va ligado a la implementación y carga de diseños *hardware* en *FPGA* y depuración a nivel de señales. La introducción a una nueva 'tecnología' llamada síntesis de alto nivel ha requerido leer artículos de investigación e indagar en el código fuente de *LegUp* con tal de comprender su esencia y también las funcionalidades que ofrece el entorno. Nos ha tomado 7 meses, teniendo en cuenta los periodos de vacaciones y la cantidad de horas que hemos podido dedicar en combinación con jornadas laborales a tiempo completo. Dentro de estos 7 meses, las horas efectivas dedicadas giran en torno a las 2 por semana. Teniendo unas 30 semanas desde febrero-marzo, las horas totales efectivas dedicadas son 30 x 2 = 60 h.

- Implementación (dificultad: alto)

  La consideramos la parte que más tiempo llevará debido a que, además de implementar código desde cero, hay que familiarizarse con el entorno Xilinx. Consideramos 40 días un tiempo razonable (sin contar fines de semana, festivos y vacaciones), dedicando 3 horas por día.

- Pruebas (dificultad: medio)

  Serán suficientes 18 o 20 días para pasar todos los tests posibles e ir corrigiendo errores sobre la marcha (suponiendo que topemos con algún error en cierto modo difícil de arreglar) y dedicando el mismo número de horas que durante la implementación, 3 horas por día. Para corregir errores menores necesitaríamos menos tiempo, entre 5 y 10 días

5.3. Recursos

Para llevar a cabo nuestro propósito, serán necesarios los recursos que a continuación especificamos:

Materiales

- Hardware:
  - Portátil *HP EliteBook 840 G3*
  - *ZedBoard development kit (Xilinx Zynq®-7000 All Programmable SoC)*

- Software:

  - SO: *Windows 10 Enterprise*

  - Entorno:

    - *Ubuntu* 14.04.2 LTS de 64 bits
    - *LegUp high-level synthesis tool*, versión 4.0
    - Ubuntu Linaro 14.04

Humanos

- Autor del trabajo, cuyo rol será el de **ingeniero técnico de I+D**

C. Gestión económica y sostenibilidad

6. Gestión económica

6.1. Gestión de los costes

Dentro de nuestro proyecto, podemos contemplar costes de estructura. Desglosándolos, tenemos:

- Costes de estructura:
    - Costes directos: RRHH, hardware y software
    - Costes indirectos: consumo de energía

Tal y como podemos ver, teniendo en cuenta que no generaremos gastos de actividad (ni por compra de materias primas, ni por alquileres de oficinas o seguros, etc), tan sólo tenemos que fijarnos en los costes de estructura.

6.2. Planificación

Dentro de proyectos informáticos, los gastos más comunes tienen que ver con:

- Recursos humanos
- Hardware
- Software
- Consumo de energía

La única parte que no generará gastos es el software, ya que se trata de software de código abierto en este caso gratuito y disponible en la web de la Universidad de Toronto y en sus respectivas páginas web (en el caso del SO y de la máquina virtual). En cuanto al hardware y los recursos humanos, tienen que ver con todas y cada una de las fases de nuestro proyecto.

Estimaciones

Basándonos en el diagrama de Gantt del anexo, tenemos:

Hardware

Para cada tarea, el material necesario para realizarla será el ordenador portátil utilizado para ejecutar el software, realizar pruebas y redactar documentación. Los gastos son los siguientes:

- Ordenador portátil HP EliteBook 840 G3 = 1245 €
- ZedBoard development kit (Xilinx Zynq®-7000 All Programmable SoC) = 400 €

Recursos humanos

Para calcular el precio a pagar al autor del proyecto por hora, nos hemos basado en las cifras recogidas por *MichaelPage* en el siguiente enlace:

https://www.michaelpage.es/sites/michaelpage.es/files/ER_ING_PG.pdf

Considerando un único rol, el de *ingeniero técnico de I+D*, y las cifras que aportan como máximo y mínimo a cobrar (28.000 y 20.000 euros respectivamente), haremos a continuación la estimación de presupuesto destinado a remunerar la actividad del autor del proyecto:

- Documentación e investigación: 60 horas
- Implementación: 40 días laborables
- Pruebas: 18-20 días laborables

Suponiendo un mínimo de 3 horas por cada día laborable, tenemos un total de:

40*3+20*3+60 = **240 horas**

Hemos de tener en cuenta que el autor cobrará más por hora en aquellas etapas que requieren más esfuerzo: documentación e implementación. Primero haremos los siguientes cálculos:

1 mes = 4 semanas

1 semana = 40h

Por lo tanto, 1 mes = 160h

28000€/14 pagas = 2000€/mes

2000€/mes / 160 h/mes = **12,5 €/h**

20000€/14 pagas = 1428,57€/mes

1428,57€/mes / 160 h/mes = 8,92€/hora => **9 €/h**

Ahora, podemos calcular lo que cobrará el autor en base al precio máximo y mínimo por hora suponiendo fases del proyecto de mayor y menor dificultad en comparación:

180 horas (documentación e implementación) * 12,5€/h = **2.250€**

60 horas (pruebas) * 9€/h = **540€**

Total = 2250+540 **= 2.790 €**

Consumo de energía

Debido a que en este proyecto se trabaja con dispositivos electrónicos tales como un ordenador portátil y en ocasiones con una *FPGA,* haremos una estimación en Kwh del consumo energético derivado de nuestra actividad.

En el informe de planificación temporal advertimos las siguientes fases del proyecto y su correspondiente estimación temporal:

- Documentación e investigación: 60 horas
- Implementación: 40 días laborables x 3 horas/día
- Pruebas: 18-20 días laborables x 3 horas/día

Para obtener una cantidad en horas, supondremos que por cada día laborable dedicaremos como mínimo 3 horas. Eso significa que a la implementación le dedicaremos unas 120 horas y a las pruebas unas 60 horas. Para la fase de documentación e investigación hemos dedicado unas 60 horas.

Tras consultar el enlace http://www8.hp.com/es/es/products/laptops/product-detail.html?oid=7815294#!tab=specs de la página oficial de HP, encontramos que la batería del portátil que utilizaremos para este proyecto apotra 46 W por cada hora. Hagamos números:

46 W * (120 + 60 + 60) = 46 * 240 = 11.040 W*h = **11,040 Kwh totales.**

Vamos a suponer un precio medio por Kwh que contemple el precio en horas valle y el precio en horas punta. Cifras consultadas en http://www.tarifadeluz.com/

Tenemos 0,13€/Kwh x 11,040 Kwh = **1,44 €.**

Contingencias

Dado que en la estimación de costes hemos contemplado los peores casos en cuanto a horas dedicadas por parte de la mano de obra, en este caso contemplaremos aquellas situaciones en que sea necesario:

Reparar los dispositivos utilizados

- Reemplazar alguno o ambos dispositivos

Teniendo presentes los costes del hardware mencionado:

- Ordenador portátil: 1245 €
- FPGA: 400 €

Suponemos un 50% de probabilidades de que en algún momento el ordenador se estropee, y un 20% de posibilidades de que algún elemento de la FPGA no funcione correctamente. Esto significa que como partida de contingencia, apartaríamos una cantidad de:

1245 x 0,5 + 400 x 0,2 = 204,5 € -> **205€**

Con este importe deberíamos poder cubrir las posibles reparaciones del ordenador portátil o la hipotética compra de un dispositivo que sustituya la actual FPGA.

Imprevistos

Para cubrir gastos imprevistos dispondremos de un fondo de reserva equivalente al 15% de la suma de los costes de hardware, recursos humanos y consumo de energía.

3.038,94 x 0,15 = **455,84€**

Amortizaciones

Para los equipos electrónicos que utilizaremos, utilizaremos los coeficientes de amortización publicados y disponibles en el siguiente enlace de la página web de la Agencia Tributaria:

http://www.agenciatributaria.es/AEAT.internet/Inicio/_Segmentos_/Empresas_y_profesionales/Empresas/Impuesto_sobre_Sociedades/Periodos_impositivos_a_partir_de_1_1_2015/Base_imponible/Amortizacion/Tabla_de_coeficientes_de_amortizacion_lineal_.shtml

Considerando equipo electrónico la FPGA y un sistema informático el portátil, vamos a calcular las correspondientes amortizaciones suponiendo que hemos utilizado estos equipos durante **1 año.**

Total

	Unidades	Precio unitario (euros)	Vida útil (años)	Amortización estimada (euros/año)	Precio (euros)
Costes directos					
RRHH	1	2790	-	-	2790
Hardware (Portátil)	1	1245	6	208	208
Hardware (FPGA)	1	400	10	40	40
Costes indirectos					
Consumo de energía	11,040 (Kwh)	0,13 (por cada Kwh)	-	-	1,44 (1)
Imprevistos	-	455,84	-	-	456
Contingencias	-	205	-	-	205
Total sin IVA	-	-	-	-	**3700**
Total con IVA	-	21%	-	-	**4477**

Tabla 2. Presupuesto

6.3. Control de gestión

En el caso que nos ocupa, las desviaciones se producen comúnmente en la duración de las diferentes fases. Es recomendable llevar un registro específico y periódico de las horas reales invertidas y elaborar una lista de verificación al final de cada fase o actividad. Se calculará el coste real en horas y se comparará con el coste de las horas que hemos estimado.

Podrían surgir desviaciones de tiempo durante las etapas de implementación y pruebas. En el primer caso, debemos tener presente que estudiaremos un código ajeno y en la mayor parte de ocasiones no dispone de suficientes comentarios como para comprender la totalidad de su estructura. Esto puede acarrear demoras en susodicha fase. En el segundo caso, las pruebas pueden dar resultados erróneos y ello supondría revisar, asimismo, nuestra implementación (suponiendo que ninguno de los tests disponibles contenga errores). Es por esta razón que se ha establecido una partida para imprevistos que cubra las posibles horas extra dedicadas a corregir errores y/o a reimplementaciones.

Los indicadores que utilizaremos serán los siguientes:

- Desviaciones en la realización de tareas (en coste): (coste estimado – coste real)* consumo horas real
- Desviaciones de un recurso hardware (en coste): (coste estimado – consumo real) * coste real
- Desviaciones en la realización de tareas (en horas): (consumo estimado – consumo real) * coste real
- Desviaciones totales en la realización de tareas: (coste estimado total – coste real total)
- Desviaciones totales de recursos: (coste estimado total – coste real total)

7. Sostenibilidad

A continuación exponemos una matriz de sostenibilidad cuyos valores reflejan el impacto ambiental, económico y social que creemos que corresponden a la realización de este proyecto:

	PPP	Vida útil	Riesgos
Ambiental	10	5	-5
Económico	5	5	-10
Social	10	10	0
Total	25	20	-15
	30		

Tabla 3. Matriz de sostenibilidad

Los valores de cada celda quedan justificados en los subsiguientes apartados.

7.1. Estudio del impacto ambiental

Tal como hemos visto en el apartado de gestión económica, el hardware es un recurso que se utilizará durante toda la realización del proyecto. El consumo será de aproximadamente unos 11 Kwh (11,040). Tal y como hemos podido ver, pese a que se consuma energía durante todas las fases del proyecto, la cantidad de energía consumida es ínfima en comparación al gasto que podría derivarse de una cierta actividad en oficina.

Durante las distintas fases del proyecto, los recursos a utilizar han sido los mismos. La fases de documentación, implementación y pruebas requieren el uso de nuestro computador. Es en la fase de implementación y la de pruebas en las que necesitamos también la *FPGA*.

Aparte de *LegUp, se* han podido reaprovechar recursos de otros proyectos de software libre / código abierto / propietario que listamos a continuación:

- **libmigdb**: librería que implementa un protocolo llamado *GDB/MI*, que permite establecer comunicación con un proceso de *GDB* sin necesidad de interactuar con éste a bajo nivel. Escrita por Salvador Eduardo Tropea [8].

- **OmpSs4FPGA:** versión de Ubuntu Linaro 14.04 para nuestra placa facilitada por Daniel Jimenez-González, director de este proyecto.

El impacto ambiental generado por actividades similares a las que se realizarían sin la existencia de este TFG sería el mismo. El hecho de implementar las funcionalidades de nuestro programa no tiene nada que ver con los medios hardware que vayan a utilizarse a posteriori.

7.2. Estudio del impacto económico

Cabe destacar que, aunque el material hardware especificado es el que utilizaremos dada su disponibilidad, podrían utilizarse equipos de sobremesa o portátiles de menor coste y con características hardware similares a las de nuestro equipo. De todos modos se espera que, dada la calidad de nuestro hardware, este no se estropee y por consiguiente no sea necesario utilizar el presupuesto destinado a imprevistos y contingencias.

Por otra parte, el hecho de utilizar un software de código abierto disponible en la web de la misma universidad donde se ha desarrollado, implica que el coste de utilizarlo sea 0. Esto, además, permite hacerle al software las modificaciones deseadas y también probarlas.

7.3. Estudio del impacto social

El inicio y finalización de este proyecto, para su autor, ha supuesto profundizar en el ámbito de las *FPGA* al mismo tiempo que se ha sumergido en toda una nueva tecnología (síntesis de alto nivel) que hasta el momento le era desconocida y que cuyo conocimiento le ha aportado un valor de gran importancia teniendo en cuenta que ha ampliado los conocimientos relacionados con su especialidad. Ha descubierto que dentro de su especialidad existen más vías de investigación de las que creía. Asimismo, ha aprendido una forma de permitir que otros profesionales, no necesariamente dedicados a la ingeniería del *hardware*, puedan aprovecharse de las nuevas tecnologías en supercomputación. Con este proyecto, se intenta dar un pequeño paso adelante hacia la adaptación de una herramienta de código abierto existente a una mayor gama de productos presentes en el mercado, con lo cual trata de que el soporte actual no quede limitado a una determinada marca. Eso implica que aquellos que trabajen con productos de alguna de las dos marcas puedan beneficiarse de esta tecnología.

Uno de los factores que se podría pensar que hipotéticamente afecta al sector de la ingeniería del hardware es el hecho de que herramientas de este tipo reduzcan la cantidad de empleos en el ámbito de la investigación en *FPGA* y supercomputación. Creemos que no sucederá, ya que estas herramientas simplemente conforman una especie de 'puente' que permite omitir (no sustituir) ciertos conocimientos que un ingeniero especializado en ello posee (y a nivel avanzado) y que suponen un obstáculo para otros ingenieros y científicos a la hora de aprovechar estos avances en materia de *hardware.* Sostenemos firmemente que en determinados contextos y tipologías de proyecto, estos conocimientos avanzados seguirán siendo necesarios (independientemente de que el software de *HLS* existente esté lo suficientemente perfeccionado).

A grandes rasgos, el autor prevé que la síntesis de alto nivel abre un camino hacia no sólo el desarrollo de la misma tecnología sino al descubrimiento y desarrollo e investigación de tecnologías similares cuyo objetivo sea acercar la potencia de la computación moderna a un amplio espectro de profesionales.

D. Análisis/diseño

8. Funcionamiento de *LegUp*

En esta sección trataremos de consolidar los pasos que se siguen desde que disponemos de un código escrito en C hasta que depuramos el modulo hardware generado a partir de este y especificado en *Verilog*. Cabe destacar que esta depuración se llevara a cabo mediante ***GDB***, por lo que finalmente analizaremos cómo se lleva a cabo la comunicación entre *GDB* y el depurador ***Inspect*** de *LegUp.*

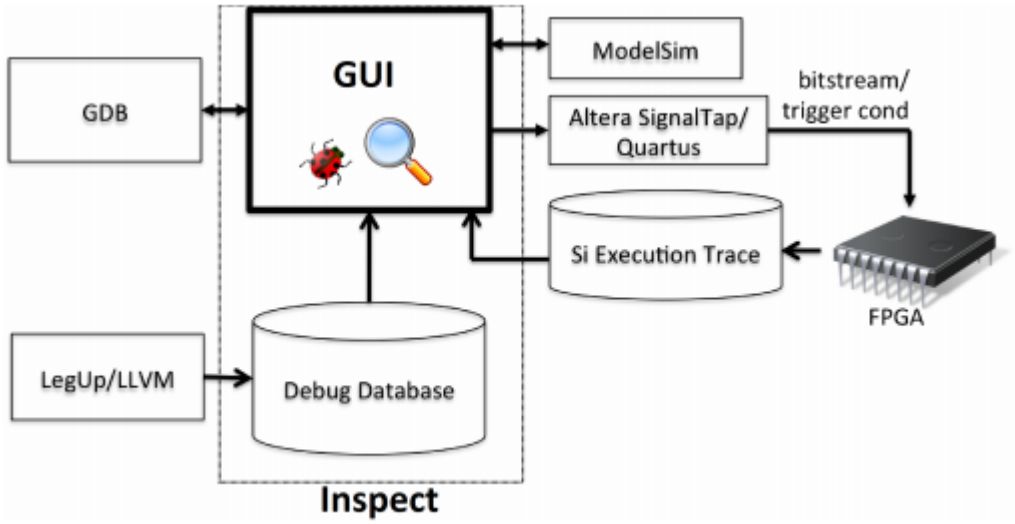


Figura 1. Estructura interna de *LegUp*

8.1. Generación del diseño

Backend

LegUp implementa una **etapa final** (ver apartado 'Etapas de *LLVM*') a la hora de compilar con *LLVM.* La clase de nivel superior se denomina ***'LegupPass'***. Esta clase es ejecutada por el gestor de *LLVM,* el cual llama al método *runOnModule()* pasando como parámetro de entrada el código *IR* y esperando como salida el código en *Verilog.*

Los **pasos** que se siguen a la hora de producir un código en *Verilog* son los siguientes:

-Asignación:

-Se lleva a cabo a través de una **clase** llamada **'*Allocation*'**.

-La **clase *'Allocation'*** lee un script *TCL* en el que se especifica el dispositivo para el que se compila, restricciones temporales y opciones de síntesis de alto nivel.

-La misma **clase *'Allocation'*** lee otro script *TCL* que contiene datos como los retardos específicos de la *FPGA* para la que se compila, entre otros.

-Los datos anteriormente mencionados se guardan en una instancia de la **clase *'LegupConfig'*** accesible a lo largo del código. Esta instancia se pasa hacia las últimas etapas de *LegUp.* Este objeto también se encarga de **mapear las instrucciones *LLVM* como señales en *Verilog***, y se asegura de que estos nombres no se solapen con palabras reservadas del propio lenguaje.

-Aquellos ajustes generales que deban ser usados en otras etapas posteriores de *LegUp*, deben almacenarse en una instancia de la **clase *'Allocation'.***

-Planificación:

-El planificador se implementa con la **clase *'SDCScheduler'***. Esta utiliza la **clase *'SchedulerDAG'***, la cual mantiene todas las dependencias entre instrucciones para una determinada función. La planificación definitiva se almacena en una instancia de la **clase *'FiniteStateMachine'*** que especifica el estado inicial y final de cada instrucción *LLVM.*

-Ensamblado (*binding*):

-Se utiliza la **clase *'BipartiteWeightedMatchingBinding'***, que lleva a cabo (bipartite weighted matching). Se almacenan los resultados en una estructura de datos que **mapea cada instrucción *LLVM* con el nombre de la unidad funcional hardware que debería implementar** la respectiva instrucción.

-**Generación de *RTL*:**

-Una instancia de la clase ***'GenerateRTL'*** ejecuta un bucle sobre cada instrucción LLVM del programa y, utilizando la información de planificación y ensamblado, **crea una instancia de la clase *'RTLModule'* (para cada instrucción) que representa el circuito hardware final**.

Frontend

-Para el modo híbrido:

-Se eliminan todas las funciones del código *IR* que deberían implementarse por software mediante la clase ***'HwOnly'.***

-Hacemos lo mismo con las que deberían implementarse en hardware mediante la clase ***'SwOnly'.***

-Lo obtenido con *HwOnly* se le pasa al *backend* de *HLS* y lo obtenido con *SwOnly* se le pasa al *backend* de compilación *MIPS/ARM*

Etapas de LLVM

Todos los **pasos** llevados a cabo dentro de *LegUp* (*LegUp* visto como una etapa (pass) o conjunto de etapas backend-frontend que forma parte de *LLVM*) tienen una función de entrada llamada:

-bool runOnFunction(Function &F);

Cuando se hace una llamada a esta función, *LLVM* ya ha construido el *IR* asociado al fichero de código C de entrada. Sobre este *IR* llevamos a cabo todos los pasos hasta generar código *RTL Verilog* válido.

-Bool runOnModule(Module &M);

En LLVM, un 'Module' tiene una lista de *'Functions'*. Una *'Function'* tiene una lista de *'BasicBlocks'*. Un *'BasicBlock'* tiene una lista de instrucciones.

8.2. Arquitectura del diseño generado por LegUp

A grandes rasgos, hemos descubierto que en el **autómata de estados finitos** del módulo especificado en *Verilog* que *LegUp* genera automáticamente se distinguen **tres partes principales**:

- Registro de estado

  -Circuito secuencial que pone al autómata en su estado inicial cuando la señal '***reset'*** toma valor **'1'.**

  -Mientras haya una petición a memoria en curso, se mantiene al autómata en el mismo estado. Si no ocurre nada de lo anterior, se hace pasar al autómata al estado siguiente.

- Lógica de próximo estado

  -Circuito secuencial que, en función del estado actual, decide a qué estado debemos ir a continuación (*Moore*). En ocasiones, el estado al que vayamos puede depender también del valor de las señales de entrada *(Mealy).*

- Lógica de salida en función del estado

  -Circuitos secuenciales que determinan qué valores deben tomar las señales asociadas a las variables del código en alto nivel en función del estado. Dentro de esta lógica se toman los valores de los puertos de lectura y se escribe en los puertos de escritura del controlador de memoria en el caso de operaciones *'**load'*** y *'**store'*** respectivamente.

  -Asimismo, determinan qué valores deben tomar las señales de control del controlador de memoria en función del estado para determinar en qué BRAM se va a realizar una operación de lectura/escritura. Por lo que se refiere al **controlador de memoria**, a los distintos ***BRAM*** y al **módulo ligado a la función '*main*'**, hemos analizado y encontrado lo siguiente.

Jerarquía de memoria

LegUp contempla una **jerarquía de memoria de 4 niveles (**de los cuales mostramos tan sólo los 2 comunes para todos los modos de ejecución ofrecidos por *LegUp***)**. Tal y como vamos a ver, la traducción de un código en alto nivel por lo que se refiere a variables se lleva a cabo asignando, a cada parte del código, un hardware que implementa la misma funcionalidad:

1) Memoria local (variables locales)

> **Por cada variable local del código en alto nivel, se crea un *BRAM*.** Por lo tanto, puede haber varias memorias locales dentro de un módulo y estas pueden accederse en paralelo. Distintos *BRAM* se conectan a través de **señales intermedias** dentro del módulo y no mediante los puertos del controlador de memoria.

La señal ***'enable'*** tomará valor **'1'** cuando la etiqueta que recibamos coincida con la asignada por defecto al *BRAM* que queremos acceder. Las etiquetas de cada *BRAM* se encuentran al principio del fichero de código *Verilog*.

```
`define MEMORY_CONTROLLER_ADDR_SIZE 32
`define MEMORY_CONTROLLER_DATA_SIZE 64
// Number of RAM elements: 4
`define MEMORY_CONTROLLER_TAG_SIZE 9
//   %a = alloca i32, align 4
`define TAG_main_0_a `MEMORY_CONTROLLER_TAG_SIZE'd2
`define TAG_main_0_a_a {`TAG_main_0_a, 23'd0}
//   %b = alloca i32, align 4
`define TAG_main_0_b `MEMORY_CONTROLLER_TAG_SIZE'd3
`define TAG_main_0_b_a {`TAG_main_0_b, 23'd0}
//   %c = alloca i32, align 4
`define TAG_main_0_c `MEMORY_CONTROLLER_TAG_SIZE'd4
`define TAG_main_0_c_a {`TAG_main_0_c, 23'd0}
//   %d = alloca i32, align 4
`define TAG_main_0_d `MEMORY_CONTROLLER_TAG_SIZE'd5
`define TAG_main_0_d_a {`TAG_main_0_d, 23'd0}
```

Tras especificar cuáles serán los tamaños de las direcciones, de los datos y de las etiquetas, se especifican las etiquetas para cada uno de los *BRAM* creados.

2) Memoria global (variables globales)

- También **se implementan en *BRAM* separados.**
- La etiqueta 0x0 se reserva para punteros nulos.
- La etiqueta 0x1 se reserva para memoria del procesador.
- Los bits de menor peso se usan para punteros. Incrementar estos bits no afecta a los bits de etiqueta.

Controlador de memoria

Se necesita un controlador para compartir memoria entre módulos. Por lo tanto, el controlador sólo se crea si hay memoria compartida entre funciones o si se falla al determinar a qué memorias hacen referencia los punteros de un determinado programa.

- El controlador de memoria tan sólo dispone de **dos puertos, A y B,** y de un camino de lectura y otro de escritura para cada puerto.

- La latencia de lectura es de 1 ciclo, por lo que debemos utilizar la etiqueta para determinar qué *BRAM* está proporcionando los datos pedidos en el ciclo anterior.

Conexión de los *BRAM* con el controlador de memoria

Los accesos a los distintos *BRAM* se encaminan a través del controlador de memoria, utilizando los dos puertos de los que este dispone. La asignación de puertos de cara a acceder a determinados *BRAM* se hace de forma arbitraria, lo que significa que puede haber X variables a las que se les haya asignado el puerto A e Y variables a las que se les haya asignado el puerto B, siendo X e Y valores iguales o distintos.

Conexión del módulo *'main'* con el controlador de memoria

A la hora de conectar el módulo 'main' con el controlador de memoria, conectamos las señales de control del controlador de memoria con las señales del módulo 'main' destinadas a este menester.

No obstante, si al compilar nuestro código hemos habilitado la depuración, *LegUp* añadirá en el módulo *Verilog* resultante una instancia del módulo ***'hlsd'***, utilizado precisamente para tareas de depuración

Comunicación entre el controlador de memoria y el sistema anfitrión

Tal y como se especifica en el manual de *LegUp*, existe un módulo llamado '**top'** cuya instancia se encarga de proporcionar al sistema anfitrión el valor de señales relacionadas con el protocolo *UART*, con el proceso de depuración y con el inicio y finalización del funcionamiento del autómata de estados finitos.

```
top top_inst (
    .clk (clk),                          → señal de reloj del sistema
    .reset (reset),                      → resetear el sistema
    .start (start),                      → iniciar el sistema
    .waitrequest (waitrequest),          → mantener máquina de estados al estado actual (valor 1)
    .finish (finish),                    → finalizar el funcionamiento del sistema
    .return_val (return_val),            → valor de retorno de la función
    .dbg_active_instance (pc_module),    → instancia de depurador activa
    .dbg_current_state (pc_state),       → estado actual de la máquina de estados
    .uart_rx (uart_rx),                  → puerto de lectura UART
    .uart_tx (uart_tx),                  → puerto de escritura UART
    .hlsd_state (debugger_state));       → estado actual del módulo de depuración
```

8.3. Capa de depuración

Para comprender como funciona Inspect, analizaremos los ficheros de código fuente disponibles en los siguientes directorios:

../legup-4.0/dbg/debugger/src

../legup-4.0/dbg/rtl

En el primer directorio encontramos el **depurador** escrito íntegramente en *Python*. En el segundo directorio encontramos los módulos hardware escritos en *Verilog* que se incorporarán a nuestro proyecto para llevar a cabo la **depuración interactiva**.

En referencia a la depuración interactiva, hemos de distinguir qué dos tipos de comunicación se llevan a cabo:

-Comunicación *FPGA* – sistema anfitrión

-Comunicación con *GDB*

En este apartado sólo trataremos todo lo relativo a la comunicación FPGA – sistema anfitrión, dejando la comunicación con GDB para apartados posteriores.

Comunicación FPGA – sistema anfitrión

Desde la *FPGA*

En este apartado nos referimos a la comunicación desde la *FPGA* hacia el sistema anfitrión.

El **módulo hardware principal** a la hora de depurar se llama **'hlsd'**. Este, a su vez, instancia los siguientes módulos. En la tabla se muestran sus nombres y el fichero donde están declarados. Posteriormente especificamos sus funcionalidades:

Módulo	Fichero(s) de código fuente	Funcionalidades
uart_control	uart_control.v (uart_altera.v y uart_xilinx.vhd)	Controlador del protocolo de comunicación serie UART. En función del fabricante (Altera o Xilinx), se crea una instancia de un módulo distinto (para Altera, una instancia del módulo '**rs232'** y para Xilinx una instancia del módulo '**uartlite_core'**)
comm	comm.v	Módulo que gestiona todas las comunicaciones entre el núcleo de Inspect y nuestro módulo hardware.
memory_supervisor	hlsd.v	Controlador de memoria
startfinish	hlsd.v	Módulo utilizado para controlar el inicio y la finalización de la ejecución del autómata de estados finitos que implementa el módulo 'hlsd'

Tabla 4. Módulos hardware de LegUp y funcionalidades

Desde el sistema anfitrión

En este apartado nos referimos a la comunicación desde el sistema anfitrión hacia la *FPGA*. Para ello analizaremos los ficheros de código fuente que componen la estructura de ***Inspect:***

Fichero de código fuente/clase	Funcionalidades
main.py	Clase que inicia la ejecución del depurador y su interfaz gráfica
gui.py	Clase que engloba todos los elementos de la interfaz gráfica. Una instancia de esta se crea en main.py
manager.py	Clase que conforma el núcleo del depurador. La clase gui.py se utiliza como capa superior que capta órdenes mediante la interfaz y se las envía a esta misma clase.
design.py	Clase que se encargará de gestionar la información del diseño hardware
modelsim.py	Clase que se encargará de comunicarse con un proceso de Modelsim en ejecución.
paths.py	Fichero en el que hay declaradas funciones utilizadas para obtener rutas hacia determinados programas o ficheros
signals.py	Clase que se encarga de emitir y recibir señales (callbacks)
util.py	Fichero en el que sólo hay declarada una función, *roundupIntToNearest,* que sirve para redondear un numero entero.

Tabla 5. Módulos del depurador de LegUp y funcionalidades

De todas estas partes, y que no está incluida en la tabla, es el fichero **comm.py** el que se encarga de establecer comunicación con la placa. A su vez, cada una de las funciones que contiene se encarga de enviar a la placa un determinado **mensaje,** en función de lo que el usuario elija hacer. A continuación, especificaremos la utilidad de cada función dentro de este fichero, con lo cual indirectamente especificamos qué tipo de mensajes podemos enviar a la placa.

Mensaje depurador - placa	Funcionalidad
step	Utiliza la función **run_cycles** para avanzar un paso la ejecución del autómata de estados que previamente se ha cargado en la placa.
run_cycles	Hace funcionar el circuito durante N ciclos.
Run	Hace funcionar el circuito indefinidamente.
Reset	Reinicia el circuito.
clr_breakpoint	Elimina un breakpoint hardware.
SetBreakpoint	Añade un breakpoint hardware.
verifySystemID	Comprueba que el ID obtenido es el del sistema.
VarWrite	Escribe un valor en una variable.
RamRead	Lee una variable.
verifyIsConnected	Comprueba que la placa esté conectada.
TraceVarEnable	Habilita el rastreo de una variable.
TraceFunctionEnable	Habilita el rastreo de un módulo.
SerialSleep	Poner el circuito en hibernación durante el tiempo especificado.

Tabla 6. Mensajes depurador – placa de LegUp

Comunicación con GDB

En Internet, hay a disposición de los usuarios una librería llamada '**libmigdb**' que implementa una interfaz (llamada GDB/MI, que significa 'GNU debugger machine interface') en C y C++ que sirve para comunicarse con GDB. Se expresa explícitamente que esta librería está pensada para la comunicación entre programas, no entre usuarios y programas, dada la poca legibilidad de las respuestas que emite GDB (de ahí el término 'machine interface'). Cabe destacar que esta librería **no ha sido escrita específicamente para *LegUp*** ni tiene que ver con el desarrollo del proyecto, es un proyecto aparte.

Para entender cómo Inspect conecta com un proceso en ejecución de GDB, conviene analizar el contenido del fichero **main.cpp** del directorio ../InspectDebugger/Inspect/, ya que se trata de la función principal de Inspect. Tal y como podemos ver, en el código se inicializa una instancia de la clase **GDBWrapper,** disponible en el fichero GDBWrapper.cpp. Esta clase iniciará todas las estructuras de datos necesarias para comunicarse con un proceso de GDB y así depurar el programa deseado.

9. Valoración de alternativas

9.1. Objetivos iniciales

El planteamiento inicial de este proyecto se basaba en extender una herramienta de *HLS* de código abierto denominada *LegUp,* añadiéndole soporte para placas de la marca *Xilinx* cuando inicialmente el entorno tan sólo proporciona soporte para placas de la marca *Altera*.

- La etapa de documentación giró en torno al estudio de los artículos disponibles en la red sobre *HLS* y la documentación del mismo entorno *LegUp.*
- Asimismo, fue necesario estudiar en profundidad el código fuente de los distintos módulos que lo conforman, para tener una idea de cómo extender sus funcionalidades.
- Las pruebas iban a consistir en utilizar los tests que por defecto proporcionaba el mismo entorno, y que se basaban en comparar la ejecución de los circuitos cargados en placa a la simulación de los mismos en *ModelSim* o al resultado del código en alto nivel compilado y ejecutado.

No obstante, y por razones que especificamos en la memoria del proyecto, se terminó decidiendo elaborar una implementación propia (en otras palabras, un depurador de *HLS* propio). Pese a ello, **la idea principal (desarrollar funcionalidades de *HLS* para *Xilinx*) se ha mantenido.**

9.2. Posibles implementaciones

Durante la etapa de documentación se pensó en **extender el soporte para depuración en placa** orientado a productos de *Xilinx* que ofrece *LegUp*. Surgieron dos alternativas que, como veremos seguidamente, **se descartaron para elaborar una implementación propia:**

- Com. *PC-PL*: uso de *UART* por cable para comunicar ambos dispositivos
- Com. *PS-PL*: uso del *PS* para controlar el *UART*

Con tal de descartar ***UART*** como interfaz de comunicación *PS-PL (*lo cual nos ahorraría implementar un controlador *UART* para la *FPGA)*, nos pusimos a estudiar el entorno de Xilinx y las capacidades que posee nuestra placa. Descubrimos que existía una interfaz *GPIO* denominada ***AXI-GPIO*** que podía utilizarse para la comunicación *PS-PL.* Su implementación se detalla a posteriori en este mismo capítulo.

Habiendo escogido implementar un producto desde cero, pensamos en elaborar ejemplos a mano de códigos en C y sus supuestos equivalentes en *Verilog*. En ese mismo momento se nos presentó la idea de utilizar ***Vivado HLS*** para automatizar la generación de estos ejemplos.

Decidimos utilizar el entorno de *Vivado HLS debido* a la facilidad y rapidez de implementación e incorporación de los *IP* generados en un diseño ***Vivado.***

En cuanto al entorno software, hemos listado las razones por las cuales hemos decidido elaborar una implementación propia:

- Documentación:
    - En ocasiones ambigua, impidiendo comprender el funcionamiento de *LegUp* en su totalidad.
- Código:
    - Debido a la calidad de la documentación, presencia de errores a la hora de hacer funcionar *Inspect.*
    - Algunos ficheros de configuración *TCL* y *Makefiles* fueron dejados con parámetros de configuración determinados. Los scripts no han sido escritos para contemplar todos los posibles parámetros, configuraciones y tipos de placa, dificultando la compilación y la utilización de *LegUp.* También resulta complejo aplicar parches o arreglos, dado que es difícil saber de dónde proceden los errores.
    - Carencia de comentarios en el código, obligando al ingeniero a hacer ingeniería inversa para descubrir cómo se han implementado las distintas funcionalidades.

- Soporte para *Xilinx*
    - El soporte beta nos parece insuficiente como base de la que partir a la hora de extender ese soporte.
    - La generación de *bitstreams* para la Virtex-6 se lleva a cabo compilando con *Makefiles*. A la hora de compilar, aparecen errores que hacen dudar de si el proceso de generación del *bitstream* ha finalizado correctamente.
    - La máquina virtual de *LegUp* no dispone de entornos Vivado/ISE instalados.

E. Implementación final

El software que hemos implementado consiste en un depurador *HLS* llamado **hgdb**, junto con una *API* que de soporte a su posterior extensión y desarrollo. La idea gira en torno a proporcionar una herramienta cuya funcionalidad sea igual o parecida a la que proporciona *LegUp,* pero tratando de generar el mínimo código fuente y añadirle comentarios de tal manera que la herramienta sea fácil de comprender, modificar y mantener.

Detallamos brevemente el flujo de depuración de nuestro depurador **hgdb** (que, del siguiente diagrama, es la parte que hemos implementado desde cero).

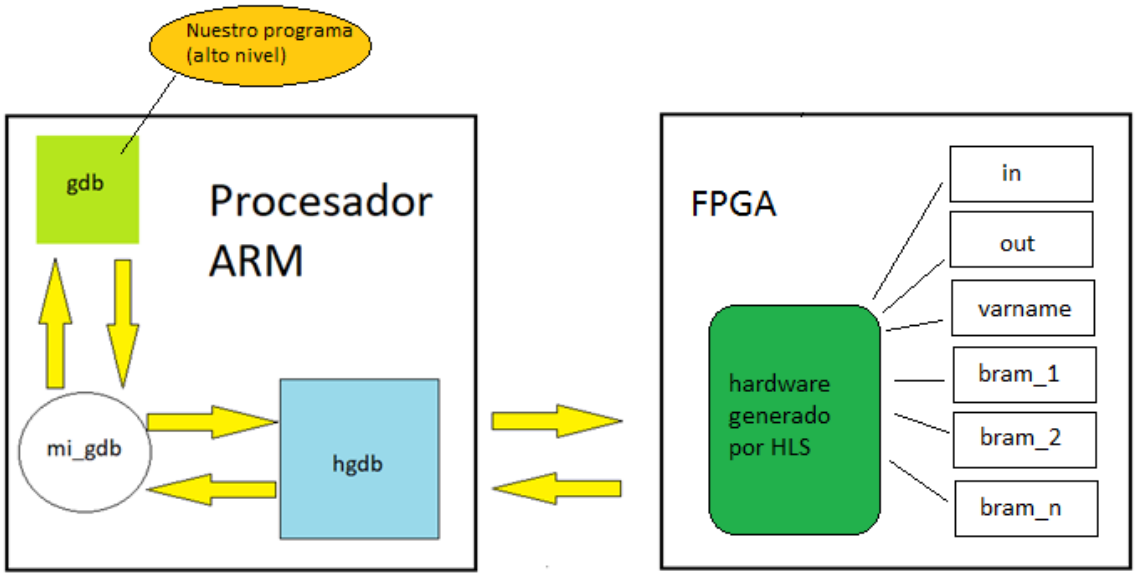


Figura 2. Flujo de depuración de hgdb

Haremos una descripción de todos y cada uno de los **módulos** que forman parte de nuestro sistema. Asimismo, describiremos el **entorno de trabajo** y cómo prepararlo.

10. Módulos

10.1. Funcionamiento de *HGDB*

Intérprete de comandos

Teniendo en cuenta que *GDB* implementa un intérprete de comandos sin interfaz gráfica, hemos decidido seguir con el mismo diseño por cuestiones de consumo de recursos y de facilidad de implementación. No obstante, y como describimos en el apartado 'trabajo futuro', será necesario añadir una interfaz para mejorar su usabilidad y facilitar el uso a personal no tan familiarizado con entornos de línea de comandos.

Si bien el propio *GDB* dispone de una interfaz llamada *GDBTUI*, mediante la cual se puede mostrar el recorrido a través del código fuente en terminal dividida y de forma gráfica, hemos decidido no utilizarlo debido a que en la misma página web de este *plugin* se especifica que la mayoría de comandos propios de *GDBTUI* no funcionarán correctamente si se ha establecido comunicación previa con un proceso de *GDB* mediante GDB/MI (que es precisamente lo que hacemos). Nuestro depurador dispone de dos modos de ejecución (HGDB y HWDEBUG) y acepta los siguientes comandos:

```
WELCOME TO HGDB!
Author: Christian Meléndez Núñez
Director: Daniel Jimenez-Gonzalez
2018, Universitat Politècnica de Catalunya


LIST OF COMMANDS FOR EACH EXECUTION MODE:

HGDB: GDB's 'built-in' functions (hgdb mode)

-break [line]   :       set a breakpoint at a certain line of the source code
-step           :       execute a line of source code
-run            :       execute the whole program until the end or until a breakpoint is found
-frame          :       get info about the current stack frame
-locals         :       show local variables at a certain point of execution (a breakpoint)
-curstate       :       show info about the current state of the program
-exit           :       exit the debugger
-reload         :       restart the chosen program's execution

HWDEBUG: hardware debugging functions

-trace          :       get the execution trace of a program
-strace         :       print the execution trace of a program
-hwdebug        :       perform in-system debug between a source code file and a VHDL file.

 NOTICE         :       HWDEBUG mode commands' help will be available by typing 'help' once you have switched to this mode.

hgdb > hwdebug

LIST OF COMMANDS FOR HWDEBUG MODE:

-step           :       perform a step both within the source code and the hardware
-read           :       read a hardware variable and automatically compare its value to the one in the source code (discrepancy detection)
-restart        :       restart both the software and the hardware versions
-hwinfo         :       show information about the harware variables in our system.
-curstate       :       show info about the current state of the program

hwdebug >
```

Figura 3. Vista de la ayuda de *hgdb*

HGDB: consta de las funcionalidades propias de *GDB*

- *break [línea]* → establece un *breakpoint* en una determinada línea de código fuente.
- *step* → ejecuta una línea de código fuente.
- *run* → ejecuta el programa entero hasta el final o hasta que se encuentra un *breakpoint*.
- *frame* → obtiene información sobre el *stack frame* actual.
- *locals* → muestra el valor de las variables locales en un punto determinado de la ejecución.
- *curstate* → muestra información sobre el estado actual del programa
- *exit* → salir del depurador
- *reload* → reiniciar el depurador y la ejecución del programa elegido

HWDEBUG: funcionalidades implementadas por nosotros

- *hwdebug* → inicia la depuración en silicio de un determinado *stub.*
- *trace* → obtiene la traza de ejecución de un programa.
- *strace* → muestra por pantalla la traza de ejecución de un programa.
- *step* → ejecuta un 'step' del código fuente y de la versión hardware
- *read [var_name]* → lee una variable hardware y automáticamente compara su valor con el de la misma variable en el código en alto nivel en el punto de ejecución en el que se encuentra (detección de discrepancias).
- *hwinfo* → muestra información de las variables hardware de nuestro sistema
- *restart* → reinicia tanto el programa como el circuito

El código fuente deseado y previamente traducido a *VHDL* se depurará al mismo tiempo que se establece comunicación con la *FPGA*.

Al iniciar el modo de depuración en silicio, se dispondrá de una estructura de datos adaptada a cada *stub* llamada FSM_states_struct, que contiene la cantidad de ciclos que debe funcionar el diseño elegido para pasar de una determinada instrucción del código en alto nivel a la siguiente en secuencia:

```
struct FSM_states_struct{

        int inst_line;
        int num_states;

};
```

Dado que una instrucción puede tener asociados uno o más estados del autómata de estados finitos generado a partir del código C, es posible que sean necesarios uno o más ciclos para pasar de una instrucción a otra. De esta manera, se podrá avanzar a través de nuestro código en C mientras el diseño *VHDL* funcionará a la par.

Asimismo, cada stub tiene asociada una estructura de datos que relaciona los nombres de las variables del código (y también presentes en el hardware) con un identificador que las diferencia y con las direcciones de memoria a las que el hardware correspondiente (en este caso hablamos de los controladores de memoria que gestionan los accesos a los *BRAM* de cada variable) está mapeado en la memoria de la *FPGA*. He aquí la estructura de datos base '***addrmap_struct'*** y una instancia de esta:

```
struct addrmap_struct{

        int id;
        char *var_name;
        char *address;
};

addrmap suma_addrmap[] = {
        { .id = 1,   .var_name = "in",  .address = "0x40000000"},
        { .id = 2,   .var_name = "out", .address = "0x42000000"},
        { .id = 3,   .var_name = "v",   .address = "0x4A000000"},
        { .id = 97,  .var_name = "a",   .address = "0x44000000"},
        { .id = 98,  .var_name = "b",   .address = "0x46000000"},
        { .id = 99,  .var_name = "c",   .address = "0x48000000"}
};
```

Comunicación ARM – FPGA

Para este propósito no ha sido necesario utilizar ningún protocolo de comunicación serie como hace *LegUp.* A través del protocolo de comunicación *AXI,* utilizado por *Xilinx*, podemos comunicarnos con el hardware que hemos cargado en la *FPGA* y que cuyos puertos de E/S quedan mapeados en memoria en el momento de cargar el *bitstream* en la *FPGA.*

Nuestro depurador utiliza la llamada a sistema ***mmap*****,** propia de la librería *POSIX,* que permite establecer un mapeo de memoria entre el espacio de direcciones de un proceso y un fichero u objeto de memoria compartida. Tras realizar este mapeo, se escribe o se lee el dato deseado en/de una determinada posición de memoria. Esto permite enviar comandos u órdenes a nuestro diseño para controlar su funcionamiento y obtener información relevante, como por ejemplo el valor de las variables locales de nuestro equivalente en circuito en un instante concreto. A continuación mostramos los códigos que permiten enviar y recibir mensajes (resaltamos en amarillo las líneas de código en las que se produce la lectura/escritura de/en la posición de memoria en cuestión) y asimismo los mensajes que es posible enviar a la FPGA:

```
#define D_STEP      1     → hace avanzar un estado dentro de la FSM
#define D_F_STEP    2     → indica que se ha terminado de pedir avanzar un paso
#define D_READVAR   3     → leer una variable
#define D_RESTART   4     → reiniciar el circuito
```

```
int send(int message, char * address){

	int fd;
	void *ptr;
	unsigned val;
	unsigned addr, page_addr, page_offset;
	unsigned page_size=sysconf(_SC_PAGESIZE);
```

Apertura del fichero en /dev/ que representa la memoria del sistema

```
	fd=open("/dev/mem",O_RDWR)
	if(fd<1) {
	perror("Error trying to open /dev/mem. Accessing memory was not possible");
	exit(-1);
	}
```

Se establece la región de memoria en la que se mapeará el puerto de la FPGA en el que se desea escribir un valor.

```
	addr=strtoul(address,NULL,0);
	page_addr=(addr & ~(page_size-1));
	page_offset=addr-page_addr;
```

Se hace el mapeo de memoria del Puerto de la FPGA en la memoria del sistema

```
	ptr=mmap(NULL,page_size,PROT_READ|PROT_WRITE,MAP_SHARED,fd,(addr &
	~(page_size-1)));

	if((int)ptr==-1) {
		perror("Error writing to main memory");
		exit(-1);
	}

	*((unsigned *)(ptr+page_offset))=(unsigned)message;
	return 0;

}
```

```
unsigned receive(char *address){

	int fd;
	void *ptr;
	unsigned addr, page_addr, page_offset;
	unsigned page_size=sysconf(_SC_PAGESIZE);
```

Apertura del fichero en /dev/ que representa la memoria del sistema

```
	fd=open("/dev/mem",O_RDONLY);
	if(fd<1) {
	perror("Error trying to open /dev/mem. Accessing memory was notpossible.");
	exit(-1);
	}
```

Se establece la región de memoria en la que se mapeará el puerto de la FPGA del que se desea leer un valor.

```
	addr=strtoul(address,NULL,0);
	page_addr=(addr & ~(page_size-1));
	page_offset=addr-page_addr;
```

Se hace el mapeo de memoria del Puerto de la FPGA en la memoria del sistema

```
	ptr=mmap(NULL,page_size,PROT_READ,MAP_SHARED,fd,(addr & ~(page_size-1)));
	if((int)ptr==-1) {
		perror("Error reading from main memory.");
		exit(-1);
	}

	printf("0x%08x\n",*((unsigned *)(ptr+page_offset)));
	return *((unsigned *)(ptr+page_offset));

}
```

10.2. GDB e interfaz MI_GDB

Para seleccionar el programa a depurar con *GDB*, nos serviremos del script ***config.sh*** que va junto al resto de código fuente de nuestro entorno:

```
#!/bin/bash


if [ "$#" -eq 0 ]; then
   usage
   return
fi

#Primero, compilamos el depurador
make dbg

#Dependiendo del ejemplo elegido, se compila lo necesario
if [ "$1" = "suma" ]; then
   make suma
elif [ "$1" = "vars" ]; then
   make vars
fi

#Aquí se preparan las variables de entorno usadas por hgdb
#--SOURCE_PATH: needed by our debugger in order to find the source code file
#--EXEC_PATH: needed in order to find the executable file
if [ "$1" != "" ]; then
   export PATH=$PATH:.
   export EXEC_PATH=$PWD/examples/$1.out
   export SOURCE_PATH=examples/$1.c
fi
```

Antes de empezar a utilizar *hgdb,* tan sólo hay que ejecutar:

```
. ./config.sh nombre_ejemplo
```

para tener listo el entorno.

El uso que normalmente se le da a GDB consiste en interactuar con este mediante comandos. No obstante, dado que esta vez hemos querido implementar una capa que permitiera depurar un programa en su versión en alto nivel y en hardware al mismo tiempo, era necesario interactuar con GDB a más bajo nivel.

La librería ***libmigdb*** se sirve de un protocolo llamado *GDB/MI*, que permite establecer comunicación con un proceso de *GDB* sin necesidad de interactuar con éste a bajo nivel. Escrita por Salvador Eduardo Tropea []

10.3. Hardware y BRAM

Para generar hardware nos hemos servido de las herramientas de Xilinx siguientes. Las nombramos y explicamos el uso que se les ha dado:

-***Vivado***

A través de Vivado hemos creado los *block designs* que incluyen nuestros ejemplos generados a partir de *HLS.* Como se expresa en el flujo de depuración de más arriba, se utiliza un módulo ***BRAM*** para cada una de las variables que contemplamos (las del código en alto nivel, la de entrada de comandos y la de salida de resultados).

-***Vivado HLS***

Con este entorno hemos generado ejemplos especificados en VHDL a partir de un código C. Los ejemplos consisten en:

- La función principal que se traducirá a hardware
- Un conjunto de puertos de E/S comunes:
    - In: para enviarle órdenes al circuito
    - Out: para obtener el valor de la variable que previamente se ha pedido leer
    - Varname: para darle al circuito el identificador de la variable que se desea leer
- Puertos E/S para cada variable del código en alto nivel

Estos puertos se conectarán a posteriori con los *BRAM* que incluyamos en el 'block design' de nuestro ejemplo.

-***Xilinx SDK***

Este entorno nos ha servido para escribir códigos que permitieran probar nuestro hardware antes de pasar a utilizar un sistema operativo en el que poder ejecutar nuestro depurador.

En el apartado siguiente se detalla el proceso a seguir para preparar el entorno de trabajo y, posteriormente, empezar a depurar.

11. Preparación del entorno de trabajo

Antes de hacer funcionar *hgdb*, es necesario preparar el hardware a cargar en la *FPGA* y permitir que el sistema operativo reconozca ese hardware. Detallamos los pasos a seguir, usando un ejemplo al que hemos llamado 'suma' cuyo código en C es el siguiente:

```
#include <stdio.h>

int main(){

	int a = 2;
	int b = 3;
	int c = a + b;

}
```

11.1. Generación del hardware por *HLS*

En este paso utilizaremos el programa **Vivado HLS.** Partimos del **código en C** de nuestro ejemplo, que **simula una máquina de estados finitos**, donde **cada uno de los estados ejecuta una línea de la versión de código en alto nivel**. Es obvio que esto podría ser distinto, si se aplicasen ciertas optimizaciones hipotéticas podríamos ejecutar las tres líneas en sólo dos estados (como ejemplo), pero hemos tratado de simplificar el ejemplo para facilitar la comprensión total del proceso.

A lo largo de este código comentamos todas y cada una de las partes de la función 'suma', que es la que única presente en nuestro código y la que se traduce a hardware.

Código de ejemplo

```
#include <stdio.h>
#include <stdint.h>
```

Nos encontramos con la cabecera de la función, cuyos parámetros representan lo siguiente:

- In
  puerto de entrada, que será usado para dar órdenes al circuito generado
- out
  puerto de salida a través del cual obtendremos el valor de las variables cuando lo necesitemos
- v_a, v_b, v_c
  puertos de E/S hacia el hardware que representa las variables de nuestro código
- varname
  puerto de entrada por el que llegará el identificador de la variable que se quiere leer

```
void suma(volatile uint32_t in[2],volatile uint32_t out[2],volatile uint32_t
v_a[2],volatile uint32_t v_b[2],volatile uint32_t v_c[2],volatile uint32_t
varname[2]) {
```

Con las directivas Vivado HLS que mostramos a continuación, indicamos que los parámetros de la función son puertos de lectura y escritura a *BRAM's.* Aquí se asume que el sistema que hemos diseñado posee *BRAM's* para todas y cada una de estas variables.

```
#pragma HLS INTERFACE ap_ctrl_none port=return
#pragma HLS INTERFACE bram port=in
#pragma HLS INTERFACE bram port=out
#pragma HLS INTERFACE bram port=v_a
#pragma HLS INTERFACE bram port=v_b
#pragma HLS INTERFACE bram port=v_c
#pragma HLS INTERFACE bram port=varname
```

Este es el estado inicial, con valor 1

```
int state = 1;
```

El bucle se ejecutará de forma infinita, por el puerto ‘in’ nunca llegará un -2

```
while (in[0] != -2) {
```

Para la sentencia *switch-case* que viene a continuación, llevaremos a cabo distintas acciones en función del estado y el valor que llegue por el puerto ‘in’. Recordemos cuáles son los mensajes que podemos enviar hacia la FPGA:

```
D_STEP      1      → hace avanzar un estado dentro de la FSM
D_F_STEP    2      → indica que se ha terminado de pedir avanzar un paso
D_READVAR   3      → leer una variable
D_RESTART   4      → reiniciar el circuito
```

Podemos ver que en cada estado se asigna el valor original del código en C al puerto de la variable correspondiente. Asimismo, se incrementa el estado. Ambas cosas suceden sólo cuando por el puerto ‘in’ llega el valor ‘1’, que significa que queremos hacer avanzar el circuito un *‘step’*.

```
switch(state){
case 1:
	if(in[0] == 1){
		state++;
		v_a[0] = 2;
	}
	break;
case 2:
	if(in[0] == 1){
		state++;
		v_b[0] = 3;
	}
	break;
case 3:
	if(in[0] == 1){
		v_c[0] = v_a[0] + v_b[0];
	}
	break;
}
```

En el caso de haber recibido un ‘1’, nos esperamos hasta recibir un valor distinto. En caso de no hacer esto, la máquina de estados terminaría sin nuestro consentimiento.

```
if(in[0] == 1){
	while(in[0] == 1){}
}
```

Esto sirve para reiniciar el estado de la *FSM*, poniendo todos los *BRAM* de las variables a 0 y el estado a 1, que es el estado inicial.

```
if(in[0] == 4){
	state = 1;
	v_a[0] = 0;
	v_b[0] = 0;
	v_c[0] = 0;
}
```

En esta parte del código tratamos la lectura de variables. Los identificadores de variable (que entran por el puerto ‘varname’) corresponden a los valores *ASCII* de los nombres de las variables, pero sólo por casualidad. Cualquier valor sería válido como identificador.

```
if(in[0] == 3){
	if(varname[0] == 97){
		out[0] = v_a[0];
	}
	else if(varname[0] == 98){
		out[0] = v_b[0];
	}
	else if(varname[0] == 99){
		out[0] = v_c[0];
	}
} //Fin del bucle

}//Fin suma
```

Síntesis y exportado de RTL

El siguiente paso consiste en generar el hardware correspondiente y marcar que debe exportarse, para luego poder importarlo dentro de nuestro diseño/sistema hardware.

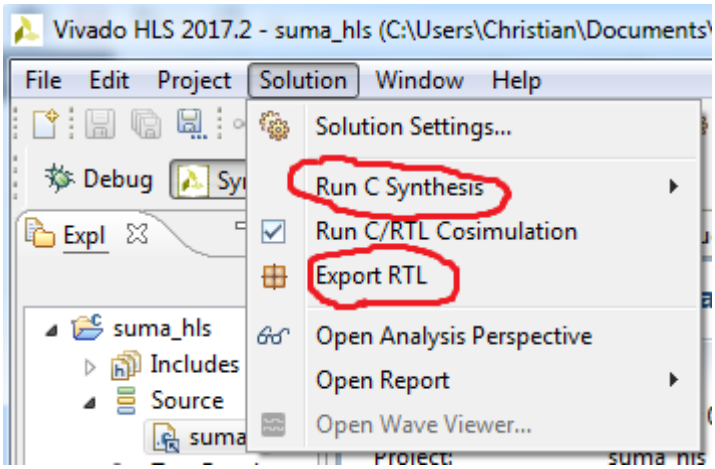


Figura 4. Síntesis y exportado de nuestro diseño

Si todo ha ido bien, tanto durante la síntesis como la exportación, aparecerán los siguientes mensajes en la consola de ***Vivado HLS:***

-Finished C synthesis

-Finished export RTL

11.2. Importación del hardware generado

Tal y como vimos en el apartado 10, los diseños hardware se crean con **Vivado**. Para el ejemplo 'suma', disponemos del diseño de bloques siguiente:

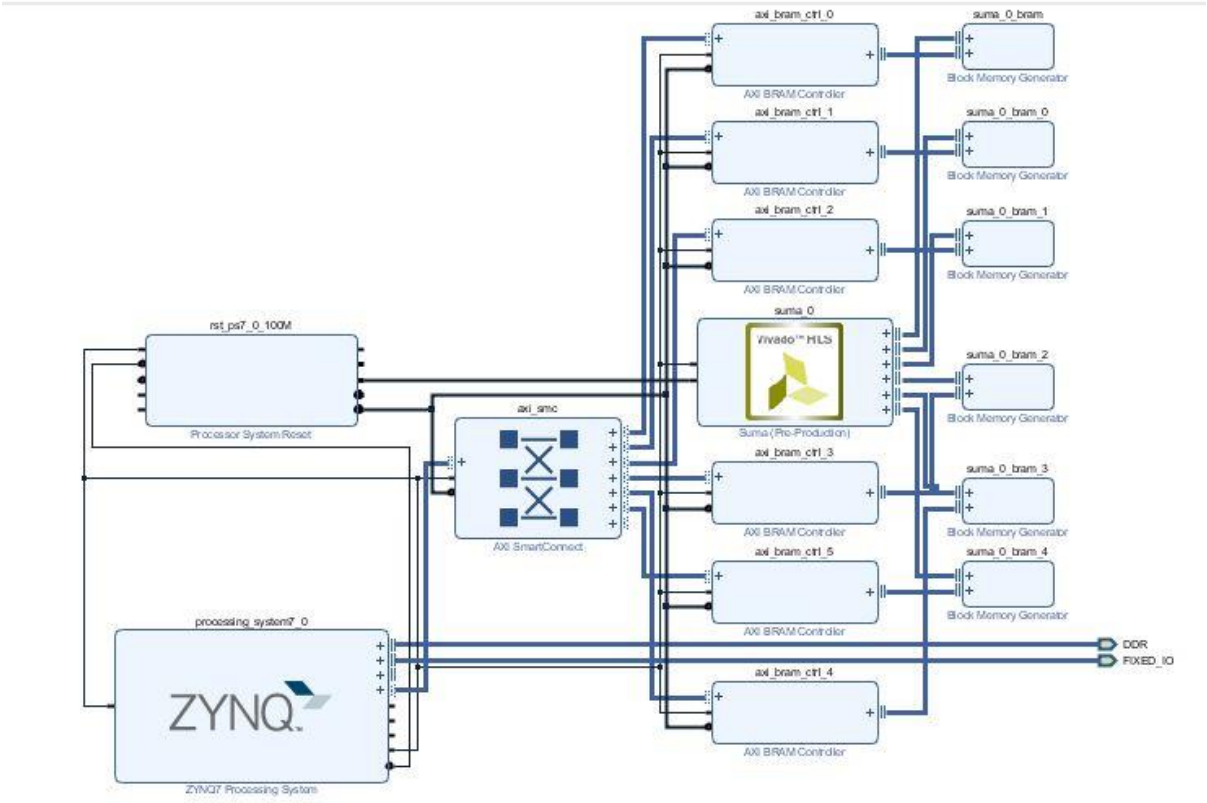


Figura 5. Diagrama de bloques del ejemplo 'suma'.

Como podemos ver, hay seis módulos *BRAM* con sus correspondientes controladores de *BRAM (*son 3 variables específicas (a,b,c) y 3 comunes (in,out,varname)).

Para incluir el hardware generado a partir de *HLS,* importamos un fichero .xml creado por *Vivado HLS* durante la compilación. Una vez seguidos los pasos para incluirlo con el resto de *IP* de las librerías de *Vivado*, el nuevo IP que hemos generado se incluye como cualquier otro dentro de nuestro diseño. Todos los puertos previamente declarados en *Vivado HLS* irán conectados a un *BRAM, y cada uno de los BRAM* a un controlador. Para leer/escribir los *BRAM,* bastará con interactuar con las posiciones de memoria a las que estén mapeados sus respectivos controladores.

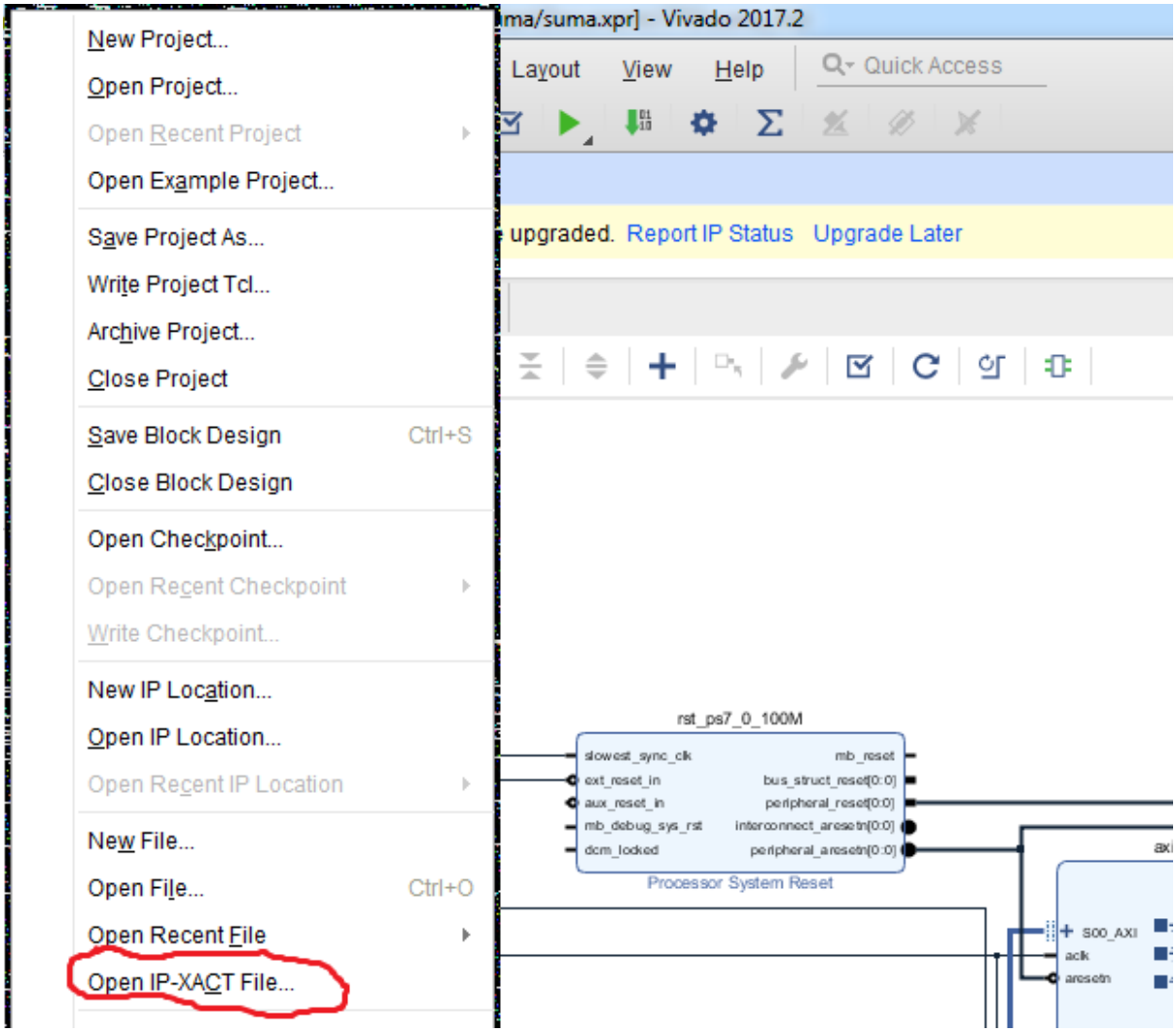


Figura 6. Importando un diseño generado por *HLS*

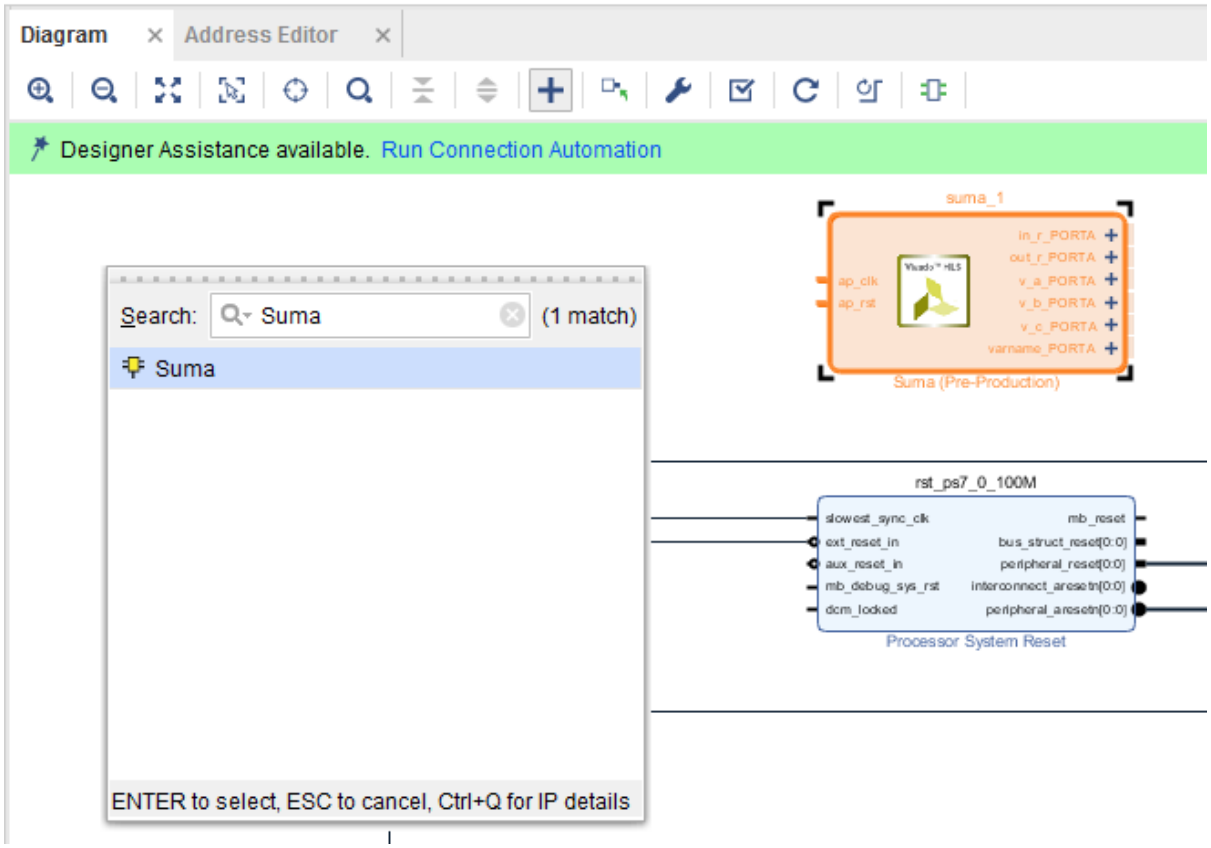


Figura 7. Insertando el IP generado por HLS en el *block design*

Una vez tenemos nuestro diseño, el siguiente paso es generar el **bitstream** que se cargará en la FPGA. Para ello seleccionamos la opción *'generate bitstream'* y esperamos a que *Vivado* nos confirme que se ha generado correctamente.

En el siguiente subapartado comentamos las especificaciones de la placa que utilizamos, el sistema operativo y lo necesario para que este reconozca el hardware que hemos añadido en nuestro sistema.

11.3. Entorno hardware: placa y SO

La **placa** utilizada es una *ZedBoard* la cual, junto al resto de componentes (alimentación, cable USB, tarjeta SD) conforman el *ZedBoard Development Kit.* He aquí las especificaciones técnicas que nos interesan:

- Zynq-7000 SoC XC7Z020CLG484-1
  - Dual ARM Cortex A9
- Memoria
  - 512 MB DDR3
  - 4 GB SD card
- Conectividad
  - Ethernet
  - UART
  - OTG
- Depuración/programación
  - Puerto JTAG USB

En la **figura 5** se muestra todo el sistema que se cargará en la FPGA. Este diseño, como se puede ver, también incluye un IP llamado *ZYNQ* que es el que contempla todo el hardware incluído en la placa. Los periféricos y protocolos de comunicación pueden activarse o desactivarse a conveniencia en *Vivado*. En este caso tan sólo tenemos activados los puertos *UART* y Ethernet.

El **sistema operativo** consiste en una versión de Ubuntu Linaro 14.04 para nuestra placa facilitada por Daniel Jimenez-González, director de este proyecto. Usaremos una tarjeta SD para arrancarlo en placa, y esta deberá contener los siguientes ficheros (previamente generados):

- boot.bin → fichero de configuración de la Zynq que se genera usando estos tres ficheros:
    - system.bit: *bitstream* usado para programar la FPGA
    - u-boot.elf → gestor de arranque de segunda etapa, usado para arrancar Línux
    - zynq_fsbl.elf → gestor de arranque de primera etapa, usado para configurar el *PS.*
- *devicetree.dtb* → *device tree* de nuestro sistema
- uImage → kernel de Linux precompilado
- .config → configuración del kernel
- uEnv → configuración de arranque

Dado que se nos han facilitado todos los ficheros, tan sólo tendremos que generar el nuevo *device tree.* Para ello, en el mismo *Vivado* exportaremos el hardware a través de la opción **File -> Export hardware,** marcando la casilla '**include bitstream'**.

Se nos generará un fichero llamado **system.hdf** (hdf = hardware description file), que es del que partiremos para **crear el *device tree***. Usaremos **cualquier imagen de Ubuntu** a la que se le haya instalado **el entorno de desarrollo de Xilinx y todas las herramientas asociadas**. Tras sincronizar el sistema de ficheros de nuestro sistema anfitrión con el de la imagen virtual, transferimos el fichero .hdf a Ubuntu y ejecutamos los siguientes comandos en un terminal, que sirven para compilar y generar el *device tree*:

- hsi
    - open_hw_design system.hdf
    - set_repo_path /directorio_codigo_device_tree → el código fuente necesario para generar el *device tree* se encuentra en http://github.com/Xilinx/device-tree-xlnx.git
    - create_sw_design device-tree -os device_tree -proc ps7_cortexa9_0
    - generate_target –dir my_dts
- dentro del directorio my_dts (fuera de hsi):
    - dtc –I dts –O dtb –o devicetree.dtb system.dts

Una vez ejecutado y obtenido el *device tree* (fichero de extensión .dtb), deberá guardarse en la tarjeta SD junto al resto de ficheros. El siguiente paso será insertar la tarjeta SD en la placa y encenderla.

11.4. Arrancando el sistema operativo

Para poder visualizar lo que está ocurriendo en nuestra placa, será necesario conectarla a la red o conectarla por cable serie a un ordenador. Recomendamos el primer método. El cliente SSH que hemos utilizado durante el desarrollo es *PuTTY.*

Si se conecta la placa por serie, veremos esta vista durante 3 segundos antes de que el SO arranque por defecto:

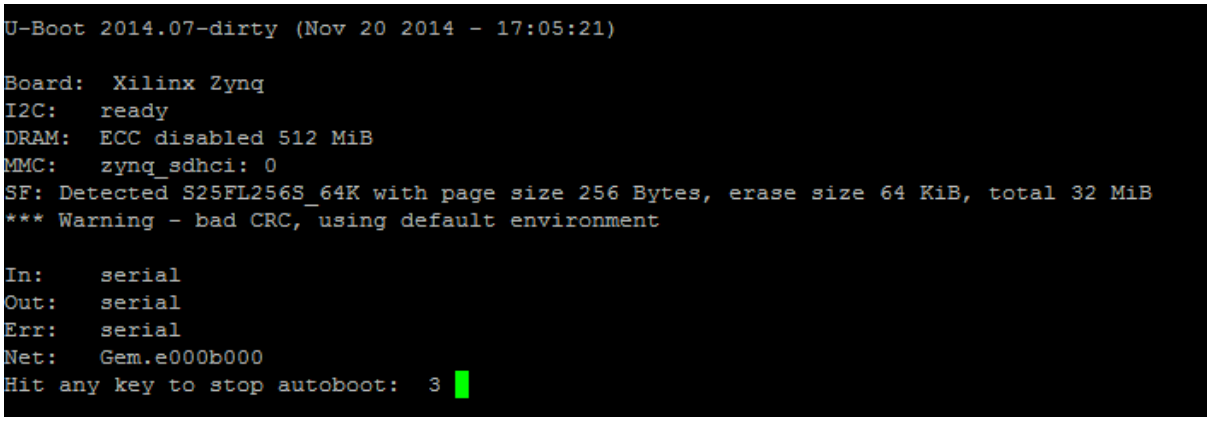


Figura 8. Gestor de arranque, antes de iniciar el SO

A partir de este punto, se nos pasará la ejecución en modo terminal (sin interfaz gráfica). Todo está listo para cargar el *bitstream* de nuestro sistema y empezar a utilizar nuestro depurador *hgdb*. Para cargar el bitstream, podemos hacerlo desde *Vivado* a través de **Open hardware manager – Program device,** habiendo seleccionado 'auto-connect' previamente para que encuentre automáticamente la *FPGA*. En cuanto la placa queda en este estado (con un LED azul encendido), el bitstream se ha cargado correctamente y todo está listo para depurar.

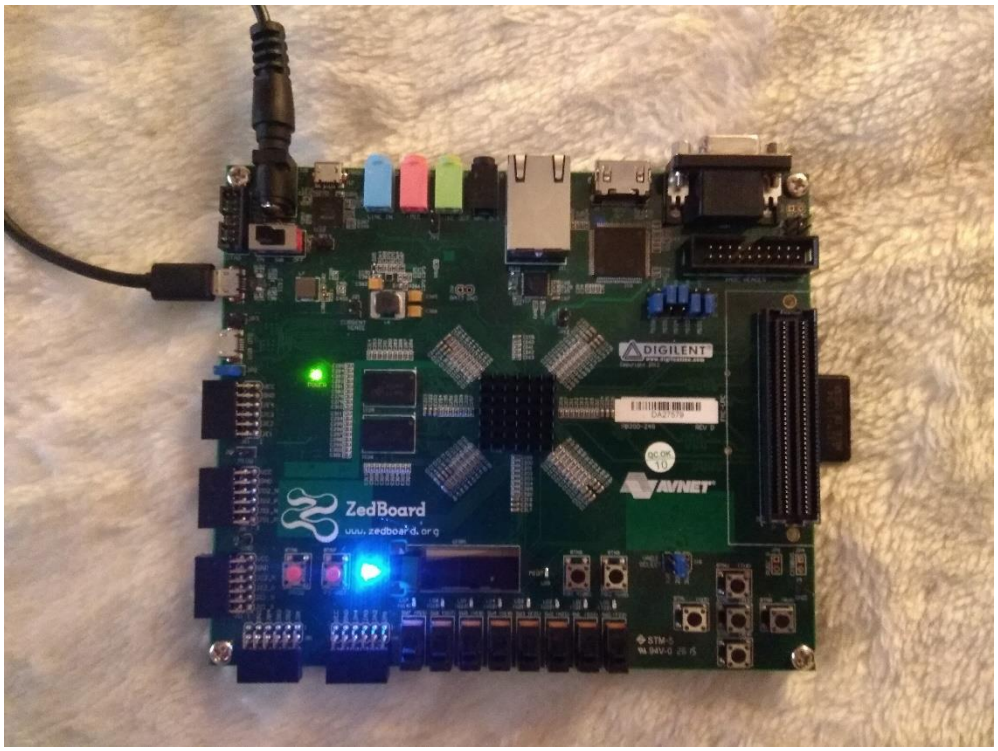


Figura 9. Placa lista para depurar

F. Resultados

Ejemplo ‘suma’

La ejecución del ejemplo ‘suma’ ha resultado satisfactoria. La comunicación con la FPGA se ha establecido sin problemas. La prueba se ha llevado a cabo con una versión del depurador que dispone de funcionalidad de detección de discrepancias entre variables del código y del hardware. Ilustramos el procedimiento de prueba del ejemplo, una vez seguida la preparación del entorno explicada en el apartado anterior.

Antes de empezar a trabajar con *hgdb*, debemos ejecutar el script *config.sh* con el argumento ‘suma, de la siguiente manera:

. ./config.sh suma

Lo primero que vemos al iniciar el modo ‘*hwdebug’* es lo que aparece en la **figura 3.**

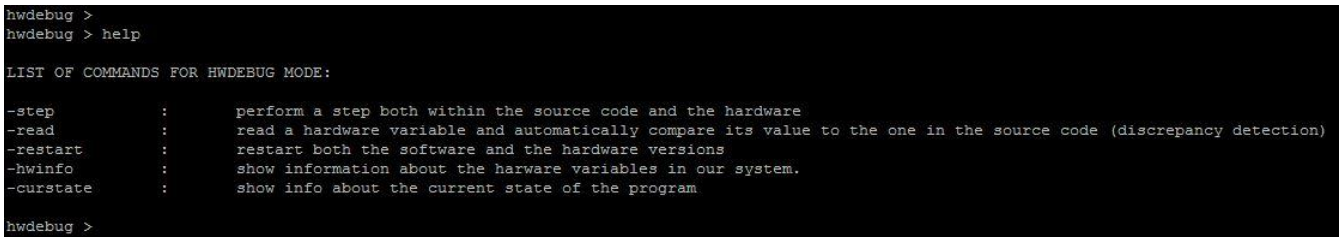


Figura 10. Comandos del modo *hwdebug*

Habiendo tecleado ‘*help’* una vez dentro del modo ‘*hwdebug’,* se nos muestran los comandos disponibles (**figura 10**).

Lo que proporcionaremos como resultado será la ejecución del ejemplo ‘suma’, mostrando como se lleva a cabo la detección de discrepancias al mismo tiempo que se avanza paso a paso no solo en el código en alto nivel, sino también en el circuito.

Para comprobar qué variables podemos consultar, mostraremos la información sobre las variables en el hardware (comando ***hwinfo, figura 11*****)**:

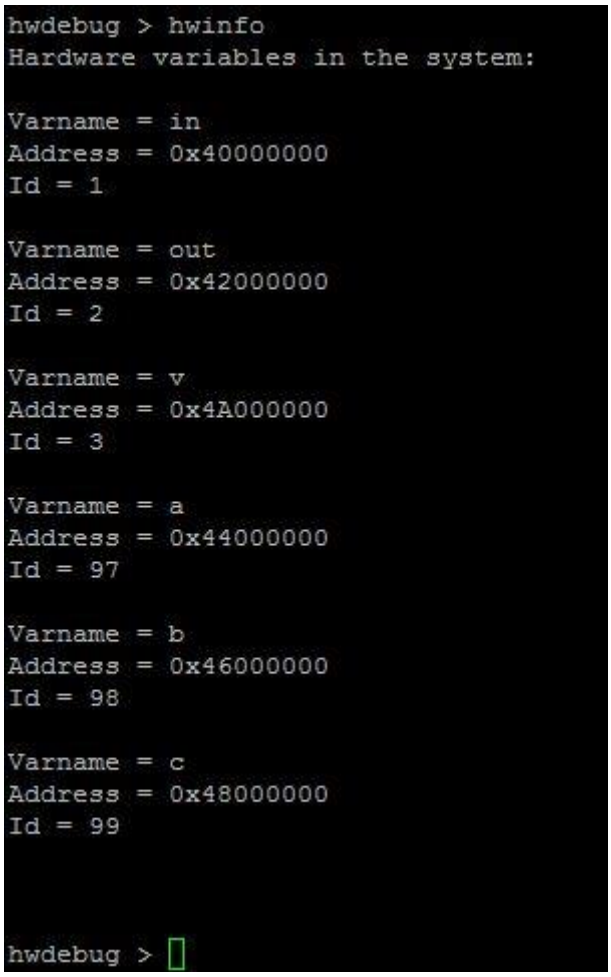


Figura 11. Información sobre las variables hardware

Si mostramos el estado actual de nuestro programa (comando ***curstate,* figura 12 a***)*, veremos que se encuentra al inicio de la función main (variables a 0). Si bien una de las variables parece no estar inicializada, no supondrá un problema para el funcionamiento del depurador ya que al ejecutar el primer *step* todo quedará sincronizado.

Lo que aparece como información son las variables declaradas en nuestro programa, su tipo y su valor en ese instante. Seguidamente, se muestra información sobre el

stack frame actual (información relevante como la función actual, la línea de código y el fichero de código, entre otros).

Mostramos el resultado de ejecutar el comando '*step'* y después mostrar el estado actual (*curstate) (***figura 12, a y b)**

```
hwdebug > step
Software step performed
Hardware step performed
hwdebug >
hwdebug >
hwdebug >
hwdebug > read a
0x00000002
Ok, var name a and line 8 state found, the var has value 2
hwdebug >
hwdebug >
hwdebug >
hwdebug > read b
0x00000000
Ok, var name b and line 8 state found, the var has value 0
hwdebug >
hwdebug >
hwdebug > read c
0x00000000
Ok, var name c and line 8 state found, the var has value 0
hwdebug >
hwdebug >
hwdebug > curstate


 Local variables (name, type, value):

        a       int     2
        b       int     0
        c       int     0
```

```
hwdebug >
hwdebug > curstate


 Local variables (name, type, value):

        a       int     2
        b       int     0
        c       int     0


 Frame info (level, address, function, file, line, args):

        Level 0, addr 0x83c6, func main, where: examples/suma.c:8 args? n

hwdebug >
```

Figura 12 (a y b, orden descendente). Estado del programa (1r step)

En la figura 12 puede verse que ejecutamos el comando *'read'* tres veces, una para cada variable (a, b, c). Podemos comprobar que no nos indica que haya ninguna discrepancia entre los valores del código y del circuito, ya que las variables 'b' y 'c' todavía siguen a 0 en este punto de la ejecución y la variable 2 contiene el valor esperado 2. Ejecutemos el siguiente 'step' (**figura 13, a y b)**.

```
hwdebug > step
Software step performed
Hardware step performed
hwdebug >
hwdebug >
hwdebug > read b
0x00000003
Ok, var name b and line 9 state found, the var has value 3
hwdebug >
hwdebug >
hwdebug > read c
0x00000000
Ok, var name c and line 9 state found, the var has value 0
hwdebug >
```

```
hwdebug >
hwdebug > curstate


 Local variables (name, type, value):

        a       int     2
        b       int     3
        c       int     0


 Frame info (level, address, function, file, line, args):

        Level 0, addr 0x83ca, func main, where: examples/suma.c:9 args? n
```

Figura 13 (a y b, orden descendente). Estado del programa (2º step)

Tras el segundo step, la variable b ha tomado valor 3 y la variable c continúa con valor 0. Todo correcto, falta ejecutar un step más (**figura 14).**

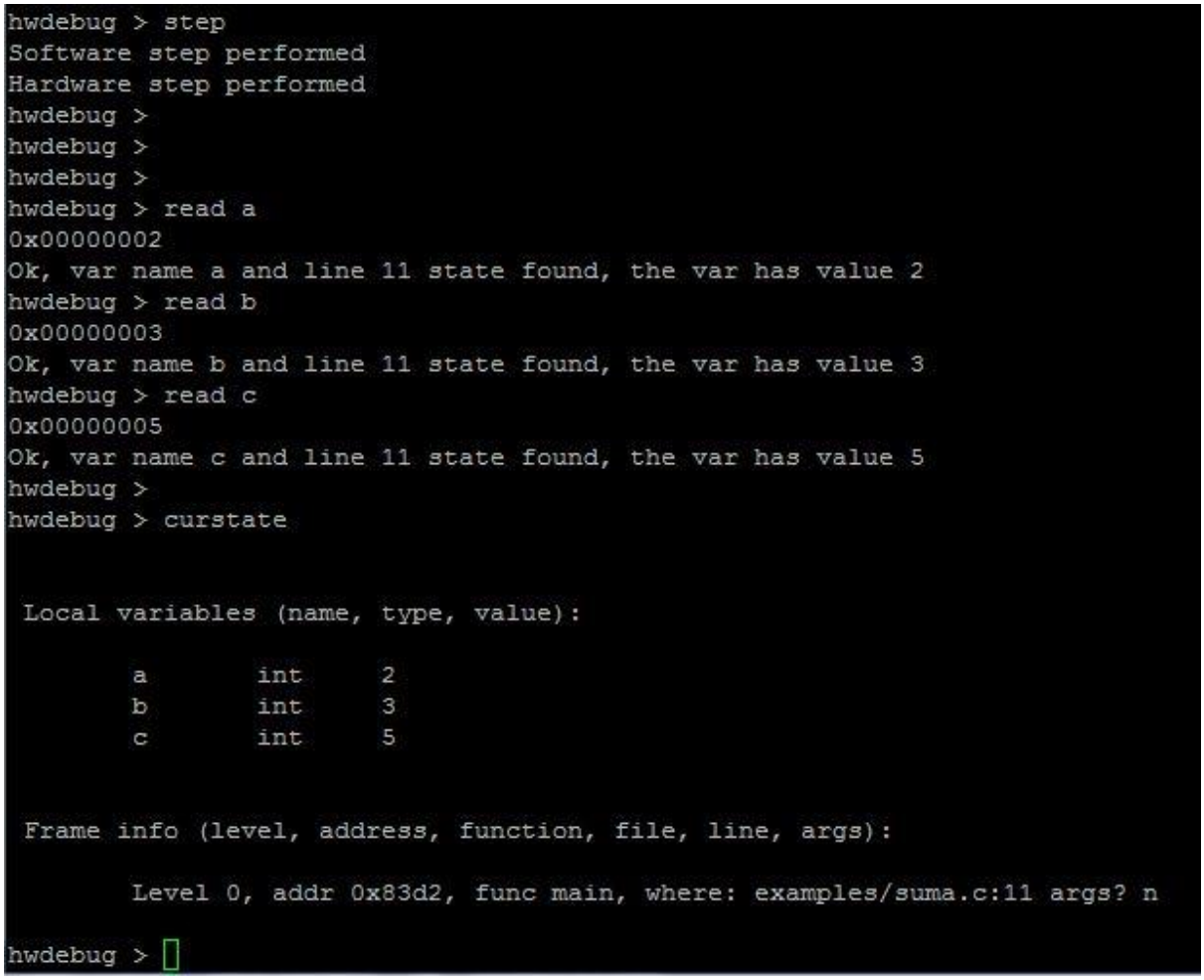


Figura 14. Detección de discrepancias y estado del programa (3r step)

La ejecución del circuito ha transcurrido sin problemas, las variables a,b,c tienen los valores 2,3,5 respectivamene, tal y como indica el código del principio del apartado 11. En un escenario 'real', esto nos indicaría que el hardware que se ha generado por *HLS* no presenta errores en su funcionamiento. En la siguiente figura mostraremos qué mensaje aparecería si por alguna razón una variable no contuviera el valor que le corresponde en un punto determinado de la ejecución del código en alto nivel. Para ello, vamos a utilizar los códigos mostrados en el apartado anterior para escribir un valor en la posición de memoria correspondiente a la variable a. Reiniciaremos el depurador y leeremos esta variable.

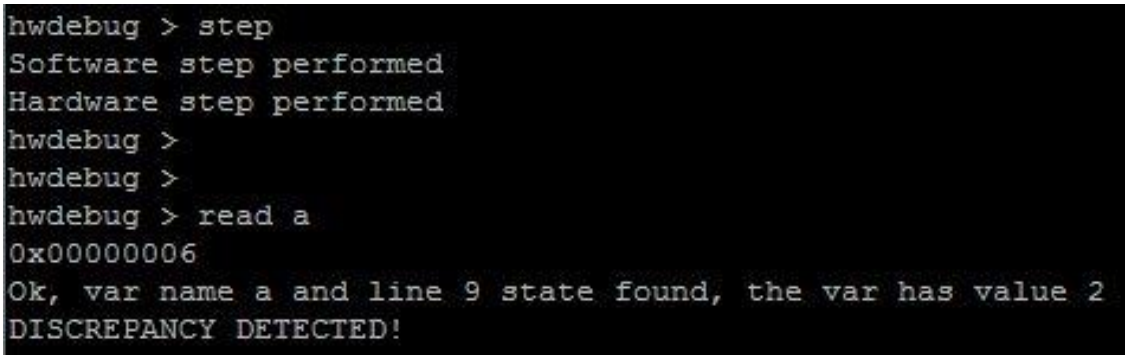


Figura 15. Detección de discrepancias en la variable 'a'. Los valores de la variable hardware y la del programa no tienen el mismo valor

Dado que la variable hardware 'a' ha tomado valor 6 (y en el código tiene valor 2 nada más empezar la depuración), el depurador nos muestra que hay una discrepancia entre ambos valores. En un caso 'real', convendría revisar el hardware generado en busca de errores.

Tras terminar la ejecución del programa, el hardware se quedará en el último estado. Para reiniciar ambos, se ha de introducir el comando '*restart'*. Esto permitirá iniciar la ejecución del programa y del hardware desde cero.

Lo que hemos mostrado es la ejecución de un programa y de un hardware a la par (modo *hwdebug)*.

Ejemplo ‘func’

Además del ejemplo anterior, hemos probado con éxito un código que no sólo **llama a una función**, sino que **contiene una sentencia *if-else.***

Por lo referente a la llamada a función, la misma llamada devuelve el resultado en el mismo ciclo. Si quisiéramos construir un cuerpo de función más complejo, se podría implementar una máquina de estados finitos dentro de esta misma función con una sentencia *switch-case* (igual que en la función principal).

En cuanto a la sentencia *if-else*, nótese que se ha tenido en cuenta que *GDB* interpreta la condición de un *if* como una línea de código más. Por ello, la simple evaluación de la condición se ha mantenido en un estado aparte, mientras que la acciones derivadas de cumplirse (o no) la condición ocupan estados distintos. El contador de estado se incrementa en 1 o 2 unidades en función del resultado de evaluar la condición.

El código, que mostraremos a continuación, mantiene la misma estructura (contiene partes dedicadas a reiniciar el circuito, a mantenerse en bucle hasta que la petición de *step* ha finalizado, a proporcionar el valor de las variables que se quieren leer, etc). Lo único que cambia es la sentencia *switch-case* que simula la máquina de estados finitos.

```
#include <stdio.h>


int mult(int a, int b){
	return a*b;
}


int main(){

	//int d = 0;
	int a = 2, d = 0;
	int b = 3;
	int c = mult(a,b);
	if (c == 6){
	 d = 1;
	}
	else{
	 d = 2;
	}
}
```

```
#include <stdio.h>
#include <stdint.h>

uint32_t mult(uint32_t a, uint32_t b){
	return a*b;
}

void func(volatile uint32_t in[2],volatile uint32_t out[2],volatile uint32_t
v_a[2],volatile uint32_t v_b[2],volatile uint32_t v_c[2], volatile uint32_t
v_d[2], volatile uint32_t varname[2]) {

#pragma HLS INTERFACE ap_ctrl_none port=return
#pragma HLS INTERFACE bram port=in
#pragma HLS INTERFACE bram port=out
#pragma HLS INTERFACE bram port=v_a
#pragma HLS INTERFACE bram port=v_b
#pragma HLS INTERFACE bram port=v_c
#pragma HLS INTERFACE bram port=v_d
#pragma HLS INTERFACE bram port=varname

int state = 1;

while (in[0] != -2) {

	switch(state){
	case 1:
		if(in[0] == 1){
			state++;
			v_a[0] = 2;
		}
		break;
	case 2:
		if(in[0] == 1){
			state++;
			v_b[0] = 3;
		}
		break;
	case 3:
		if(in[0] == 1){
			state++;
			v_c[0] = mult(v_a[0], v_b[0]);
		}
		break;
	case 4:
		if(in[0] == 1){
			if(v_c[0] == 6){
				state++;
			}
			else{
				state = state + 2;
			}
		}
		break;
	case 5:
		if(in[0] == 1){
			v_d[0] = 1;
			state = state + 2;
		}
		break;
```

```
        case 6:
               if(in[0] == 1){
                      v_d[0] = 2;
                      state++;
               }
               break;
        }//Fin FSM

        //Se espera en el bucle hasta que hemos finalizado cualquier petición desde
        //el procesador
        if(in[0] == 1){
               while(in[0] == 1){}
        }

        //Para reiniciar el estado de la FSM
        if(in[0] == 4){
               state = 1;
               v_a[0] = 0;
               v_b[0] = 0;
               v_c[0] = 0;
               v_d[0] = 0;
        }

        //Para leer las variables hardware
        if(in[0] == 3){
               if(varname[0] == 97){
                      out[0] = v_a[0];
               }
               else if(varname[0] == 98){
                      out[0] = v_b[0];
               }
               else if(varname[0] == 99){
                      out[0] = v_c[0];
               }
               else if(varname[0] == 100){
                      out[0] = v_d[0];
               }
        }

}//Fin bucle

}//Fin función
```

Para entrar en la parte *else* en lugar de la *if,* bastaría con escribir en el *BRAM* asociado a la variable ‘c’ un valor distinto a 6 tal y como hemos hecho antes para mostrar el mensaje de detección de discrepancia.

G. Trabajo futuro

El punto fuerte de este trabajo (cuya implementación dejamos abierta a propósito, dado el alcance de este) gira en torno a la traducción automática de C a *HDL.* Existen distintas opciones:

- Modo de ejecución híbrido
- Compilador C-*HDL*
- Interfaz gráfica de usuario

13. Modo de ejecución híbrido

Una mejora interesante consiste en permitir que el mismo compilador se encargue de generar el hardware correspondiente a determinadas funciones de nuestro programa. En otras palabras, permitir un modo de ejecución híbrido (ni puramente software ni puramente hardware, ciertas partes del código ejecutadas en un ordenador convencional y el resto en hardware). Supongamos el siguiente código en alto nivel con las directivas necesarias para indicar que esas porciones de código deben convertirse en circuito:

```
#define size 10
int suma_vector (int n, int *vector){
        Int res = 0;
        for(int i = 0; I < n; i++)
                res += vector[i];
        return res;
}
```

```
int main(){
        int vector[size];
        init_vector(size, &vector);
        #pragma HLS hardware
        int res = Suma_vector(size,&vector);
}
```

La función suma_vector se convertiría a hardware, quedando por lo tanto traducida a circuito tan sólo una parte del código original. Se entiende que para esto sería necesario controlar qué parte del código analiza GDB para que tanto el programa como el circuito funcionen a la par.

14. Compilador C-HDL

Posiblemente la opción que más trabajo implique. Hasta ahora, los ejemplos en *HDL* se han generado de forma automática a través del entorno *Vivado HLS* de Xilinx. Asimismo, los diseños hardware se han generado mediante el entorno *Vivado*, montando un diagrama de bloques compuesto de nuestro diseño *HLS* y un conjunto BRAM-controlador para cada variable de nuestro código en alto nivel.

El hecho de construir un compilador que generase un circuito especificado en HDL a partir de un código en C, podría conllevar asimismo la implementación/automatización de las siguientes funcionalidades:

- Compilación C-*HDL*:

-Tal como comentamos en apartados previos, *LegUp* añade a la infraestructura *LLVM* un *backend* que permite traducir de *IR* a *Verilog*. Se podría tanto implementar un *backend* propio como escribir un compilador C-*HDL* desde cero.

-Siguiendo la estructura de *LegUp* como inspiración, el circuito generado podría seguir el esquema de autómata de estados finitos con un registro de estado, una lógica de próximo estado y una lógica de salida en función del estado.

-En este caso se ha imitado el sistema de *LegUp* para mantener relación entre instrucciones del código en alto nivel y número de estados a avanzar dentro del autómata. En lugar de almacenar estas y más relaciones en esta base de datos, la simplicidad de nuestro diseño tan sólo ha requerido implementar una pequeña estructura de datos dentro dentro de **stub.c** a la que se accede en tiempo de ejecución. Una posible idea a la hora de reimplementar esta parte (si se desea) consistiría en crear y llenar una base de datos en tiempo de compilación con todas las relaciones que puedan interesar a la hora de depurar en silicio (es decir, tener en cuenta instrucciones del código, variables, estados del autómata y las relaciones que se establezcan entre ellos).

- Compilación y carga de ejemplos en placa

-Si se continúa implementando soporte para la placa *ZedBoard*, una buena idea consistiría en automatizar la compilación del ejemplo que se desea probar y su carga en placa (versión circuito). Mediante la utilización de *scripts* y *Makefiles*, junto con un conjunto de variables de entorno definido por el programador, implementarlo debería ser relativamente fácil.

15. Interfaz gráfica de usuario

Actualmente nuestro depurador consiste en un intérprete de comandos. Esto podría mejorar si, al igual que hace *LegUp*, se implementase una capa de más bajo nivel que funcionase a base de comandos, y una capa a más alto nivel que permitiera manipularlos mediante una interfaz gráfica dotada de todos los elementos que nos son familiares (visores, botones, campos de texto, etc).

Una buena forma de agilizar el proceso de implementación sería utilizando el entorno Qt mediante el cual se facilita, de forma visual e intuitiva, la creación de interfaces y el añadido de elementos. En el caso de *LegUp,* se utilizan librerías diseñadas para programar interfaces Qt en Python.

H. Conclusiones

En este trabajo hemos diseñado y desarrollado un plugin de *GDB* y el soporte hardware necesario para poder depurar código acelerado en una *FPGA*. Este depurador se ha realizado para un sistema Linux en una máquina *Zynq*, que consta de dos núcleos *ARM* y una *FPGA* totalmente integradas y que comparten la memoria principal. La parte de soporte hardware se ha probado con códigos desarrollados directamente en C que han sido pasados a hardware usando una herramienta de High Level Synthesis (*HLS*).

Finalmente presentamos un entorno de depuración de *HLS* y una *API* que permita desarrollarlo y extenderlo *a posteriori*. Nos hemos encargado de implementar la funcionalidad principal, que consiste en depurar un programa en alto nivel y sincronizar esta depuración con la de su versión en hardware, para de este modo poder detectar si hay algo en el hardware que no funcione correctamente. Los resultados muestran que se puede llegar a depurar un programa acelerado al igual que se depura un programa en un procesador de carácter general.

El grado de ingeniería informática nos ha ayudado a trabajar mucho la parte de búsqueda de información y programación, ambas indispensables tanto para implementar nuestro depurador como para hacer la investigación previa sobre *HLS* (cuya gran parte de soporte está disponible en forma de artículos de investigación). Asimismo, nos ha enseñado a valorar posibilidades y esto ha sido de gran ayuda al deliberar sobre la opción final a implementar. También hemos podido tener en cuenta el grado de sostenibilidad del proyecto, gracias a los conocimientos adquiridos durante los estudios.

Glosario

- **ASIC** (Application-specific Integrated Circuit): circuito integrado hecho a medida para un uso en particular, en vez de ser concebido para propósito general. P.E: un chip diseñado únicamente para ser usado en un teléfono móvil.

- **Bitstream:** secuencia de bits que contiene la información necesaria para programar una FPGA.

- **Block design:** representación esquemática de los módulos que conforman el hardware de un sistema.

- **BRAM:** módulo de memoria configurable generado por HDL.

- **Cluster:** conglomerado de computadoras construido mediante la utilización de un *hardware* común y que se comportan como si fuesen una única computadora.

- **Device tree:** estructura de datos que describe los componentes hardware de un determinado sistema (procesador, memoria, buses y periféricos), usado por el kernel del sistema para identificarlos y gestionarlos.

- **GPIO (**General Purpose Input/Output**):** pin genérico en un chip cuyo comportamiento se puede programar en tiempo de ejecución.

- **HPC** (High-Performance Computing): utilización de *clusters*, supercomputadoras y computación paralela para conseguir mucho más rendimiento en comparación al conseguido con computadoras de sobremesa, para resolver problemas complejos de distintas ramas tales como la ciencia, la ingeniería, los negocios, etc.

- **IP** (Intellectual Property): módulo o bloque lógico preconfigurado que puede añadirse a un determinado diagrama de bloques.

- **IR** (Intermediate Representation): lenguaje diseñado para traducir el código fuente de un programa a algo más apropiado de cara a transformaciones, mejora y optimización de dicho código fuente, antes de generar el código objeto o el código máquina correspondiente.

- **LLVM** (Low-Level Virtual Machine): infraestructura para desarrollar compiladores que se centra en la optimización de tiempos de compilación, enlazado y ejecución en cualquier lenguaje que el usuario quiera definir. Se utiliza ampliamente en proyectos comerciales y en investigación académica.

- **ModelSim:** simulador de lenguajes de descripción de hardware

- **PL:** siglas de 'Programmable Logic', referente a la parte programable de la placa (FPGA).

- **PLA** (**P**rogrammable **L**ogic **A**rray): dispositivo lógico programable que se utiliza para implementar circuitos lógicos combinacionales.

- **POSIX** (**P**ortable **O**perating **S**ystem **I**nterface (Uni**X**)): norma escrita por la IEEE que define una *API* que incluye un intérprete de comandos y herramientas para apoyar la portabilidad de las aplicaciones a nivel de código fuente entre variantes de *Unix* y otros sistemas operativos.

- **PS:** siglas de 'Processing System', en nuestro caso referente al procesador ARM Cortex-A9 del que está dotada nuestra placa.

- **RTL** (Register-Transfer Level): abstracción de diseño que modela un circuito digital en términos del flujo de sus señales entre registros hardware, y las operaciones lógicas aplicadas a estas señales. Se usa en lenguajes de descripción de hardware para crear representaciones de un circuito a un nivel superior de abstracción.

- **SignalTap:** herramienta de depuración que permite monitorizar el estado de las señales de un diseño cargado en FPGA en tiempo de ejecución.

- **Stack frame:** estructura de datos dependiente de la máquina que contiene información de estado de una subrutina. Cada *stack frame* corresponde a una llamada a una subrutina que todavía no ha terminado con un retorno a quién la llamó. Por lo general, contienen la dirección de la siguiente instrucción a ejecutar tras terminar la subrutina, los parámetros con que fue llamada, y un espacio reservado para sus variables y sus constantes locales.

- **UART:** protocolo de comunicación serie asíncrona que permite configurar el formato de los datos a enviar/recibir y las velocidades de transmisión.

- **Valgrind:** conjunto de herramientas que ayuda en la depuración de problemas de memoria y rendimiento de programas.

- **Verilog:** junto a VHDL, uno de los lenguajes de descripción de hardware más conocidos.

Referencias

[1] I.Kuon, R.Tessier, J.Rose. ***FPGA Architecture: Survey and Challenges*** [en línea]**.** Foundations and Trends® in Electronic Design Automation, 2007, vol. 2, no. 2, p.135-253 [Consulta: 20-09-2017]**.** Disponible en: http://www.eecg.toronto.edu/~jayar/pubs/kuon/foundtrend08.pdf

[2] A.Canis, J. Choi, M. Aldham, V. Zhang, A. Kammoona, T. Czajkowski, S. D. Brown, and J. H. Anderson. ***LegUp: An Open Source High-Level Synthesis Tool for FPGA-Based Processor/Accelerator Systems*** [en línea]**.** ACM Transactions on Embedded Computing Systems (TECS), 2013, vol. 13, no. 2, p.24 [Consulta: 20-09-2017]**.** Disponible en: http://legup.eecg.utoronto.ca/ACM_TECS_journal.pdf

[3] A.Canis, J. Choi, M. Aldham, V. Zhang, A. Kammoona, J.H. Anderson, S. Brown, T. Czajkowski. ***LegUp: High-level synthesis for FPGA-based processor/accelerator systems*** [en línea]. ACM/SIGDA International Symposium on Field Programmable Gate Arrays (FPGA), Monterey, CA, February 2011, pp. 33-36 [Consulta:16-09-2017]. Disponible en: http://legup.eecg.utoronto.ca/fpga60-legup.pdf

[4] University of Toronto**.** ***LegUp documentation*** [en línea]. Toronto, 17 de octubre de 2015 [Consulta: 18-09-2017]. Disponible en: http://legup.eecg.utoronto.ca/docs/4.0/legup-4.0-doc.pdf

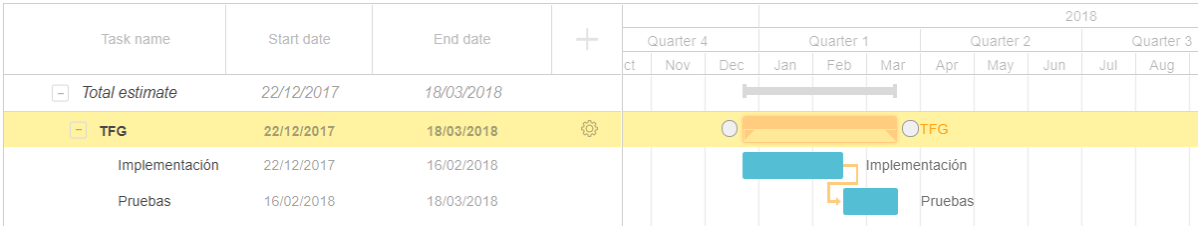
[5] N. Calagar, S. Brown, J.H. Anderson. **Source-Level debugging for FPGA high-Level synthesis** [en línea]. IEEE International Conference on Field-Programmable Logic and Applications (FPL), Munich, Germany, September 2014. [Consulta: 13-09-2017]. Disponible en: http://legup.eecg.toronto.edu/debugger_fpl_2014.pdf

[6] ***MySQL commands*** [en línea]. Fecha de publicación: 25-07-2007. Fecha de actualización: 21-01-2010 02:40:23. [Consulta: 16-04-2017] Disponible en: https://www.pantz.org/software/mysql/mysqlcommands.html

[7] GDB Wiki. ***GDB internals*** [en línea]. Fecha de actualización: 11-01-2016 10:30:21, por usuario MikeFrysinger [Consulta: 18-04-2017]. Disponible en: https://sourceware.org/gdb/wiki/Internals

[8] Tropea, Salvador E. ***Currículum Vitae*** [en línea]. [Consulta: 20-02-2017]. Disponible en: http://www.electron.frba.utn.edu.ar/CV/Tropea.pdf

Anexo: diagrama de Gantt



Se muestra la relación de precedencia entre fases del proyecto y las fechas de inicio y fin para cada una de ellas. Estas son las fechas que permiten tener el trabajo listo antes de su entrega y presentación. Tengamos presente que el tiempo requerido para implementar podría verse aumentado debido a contratiempos relacionados con las pruebas. Eso significa que el período de tiempo asignado a ambas fases quedaría solapado o repartido de distinta manera, teniendo en cuenta que si las pruebas fuesen fallidas se invertiría tiempo de pruebas en corregir los cambios.

Podemos señalar que la tarea clave es la implementación, ya que una implementación pobre puede conducir a infinidad de errores en cuyo arreglo se invierta más tiempo que en susodicha.

Anexo: árbol de directorios del código fuente

../hgdb/

-**hgdb.c**: módulo principal que se encarga de gestionar la comunicación con el proceso de *GDB* que lleva a cabo la depuración del programa elegido, y la comunicación con la *FPGA.*

-**hgdb.h:** contiene estructuras de datos usadas por el módulo principal.

-**config.sh**: script utilizado para compilar un determinado ejemplo y preparar el entorno adaptado al ejemplo que se quiera probar.

-**Makefile**: permite compilar tanto el depurador como los ejemplos. También permite depurar fugas de memoria con *Valgrind.*

/hgdb/examples/

-*.c: programas en C de los que se dispone de una traducción a *VHDL*.

/hgdb/hwdebug/

-**hwdebug.c:** contiene todas las funciones destinadas a comunicar nuestro depurador con la *FPGA*.

-**hwdebug.h**: en este fichero se especifican los posibles mensajes a enviar a la *FPGA* que hemos decidido implementar y que todos nuestros ejemplos (en formato circuito) saben interpretar.

/hgdb/stub/

-**stub.c:** contiene todas las estructuras de datos y funciones necesarias para simular ejemplos C-HDL.

-**stub.h**: aquí se encuentran las definiciones de estructuras de datos utilizadas en stub.c.