ESCUELA TÉCNICA SUPERIOR DE INGENIEROS
INDUSTRIALES Y DE TELECOMUNICACIÓN

UNIVERSIDAD DE CANTABRIA

Trabajo Fin de Grado

CARACTERIZACIÓN DE TIEMPOS DE EJECUCIÓN DE PLATAFORMAS CON PROCESADORES LEON PARA LA GENERACIÓN DE MODELOS DE SIMULACIÓN PRECISOS

Execution time characterization of LEON-based platforms to generate accurate, high-level simulation models

Para acceder al Titulo de

Graduado en
Ingeniería de Tecnologías de Telecomunicación

Autor: Seila Vega Villegas

Diciembre – 2015

GRADUADO EN INGENIERÍA DE TECNOLOGÍAS DE TELECOMUNICACIÓN

CALIFICACIÓN DEL TRABAJO FIN DE GRADO

Realizado por: Seila Vega Villegas
Director del TFG: Héctor Posadas Cobo
Título: "Caracterización de tiempos de ejecución de plataformas con procesadores LEON para la generación de modelos de simulación precisos"
Title: "Execution time characterization of LEON-based platforms to generate accurate, high-level simulation models"

Presentado a examen el día:

para acceder al Título de

GRADUADO EN INGENIERÍA DE TECNOLOGÍAS DE TELECOMUNICACIÓN

Composición del Tribunal:
Presidente (Apellidos, Nombre): Eugenio Villar Bonet
Secretario (Apellidos, Nombre): Héctor Posadas Cobo
Vocal (Apellidos, Nombre): Pablo Sánchez Espeso

Este Tribunal ha resuelto otorgar la calificación de: .....

Fdo.: El Presidente

Fdo.: El Secretario

Fdo.: El Vocal

Fdo.: El Director del TFG
(sólo si es distinto del Secretario)

Vº Bº del Subdirector

Trabajo Fin de Grado Nº
(a asignar por Secretaría)

A mi familia, especialmente a mis padres, por ser quienes me han apoyado en todo momento, sobre todo en las caídas, durante los cuatro años del grado. Nadie mejor que la familia para hacerte saber lo que vales.

A Sara e Irene, esas amigas que se hacen al comienzo de la carrera, por su apoyo incondicional en todos los momentos en los que crees que has tocado fondo.

A Daniel, ese compañero que acaba convirtiéndose en amigo tras tantas horas de clase juntos, por su paciencia y sus ánimos en los momentos más necesarios.

A Héctor, mi tutor del trabajo fin de grado, por darme esta oportunidad y por ayudarme con cada problema que ha surgido durante la realización del mismo.

A Luis Díaz, por su ayuda y disposición cuando se han encontrado problemas con la herramienta Vippe.

Y a todos los profesores que me han enseñado y apoyado a lo largo de la carrera.

ÍNDICE

1 – INTRODUCCIÓN 1

1.1 – CONTEXTO 1

1.2 – OBJETIVOS 3

1.3 – ORGANIZACIÓN 4

2 – ESTADO DEL ARTE 5

2.1 – ISS 5

2.2 – TRADUCCIÓN BINARIA 6

2.3 – SIMULACIÓN BASADA EN TRAZAS 6

2.4 – SIMULACIÓN NATIVA 7

3 – INFRAESTRUCTURA Y HERRAMIENTAS 8

3.1 – LEON 8

3.1.1 – ELEMENTOS CONVENCIONALES 9

3.1.2 – PERIFÉRICO L3STAT 10

3.2 – PLACA VIRTEX-5 ML506 11

3.3 – COMPILADORES 13

3.3.1 – GCC 13

3.3.2 – LLVM 14

3.4 – LINUX-BUILD 15

3.5 – IMPACT 15

3.6 – GRMON 15

3.7 – SIMULADORES 16

3.7.1 – TSIM 16

3.7.2 – VIPPE 17

4 – DESARROLLO DEL TRABAJO 20

4.1 – ETAPAS DESARROLLADAS 20

4.2 – DESARROLLO DE LAS ETAPAS 21

4.2.1 – PRIMERA ETAPA 21

4.2.2 – SEGUNDA ETAPA 24

4.2.3 – TERCERA ETAPA 30

4.2.4 – CUARTA ETAPA 38

5 – RESULTADOS 40

6 – CONCLUSIONES 49

REFERENCIAS *51*

ANEXO 1 ..... *52*

Execution time characterization of LEON-based platforms to generate accurate, high-level simulation models

KEYWORDS

Processor, LEON-3, Simulation, Vippe

PALABRAS CLAVE

Procesador, LEON-3, Simulación, Vippe

1 – INTRODUCCIÓN

1.1 – CONTEXTO

Se entiende por sistema embebido un sistema basado en la combinación de hardware y software, diseñado y optimizado para resolver una función específica de manera eficiente, formando parte de un sistema más amplio. En su diseño se busca maximizar parámetros como el rendimiento y la fiabilidad mientras se minimizan otros como el consumo, el tamaño o el coste.

Para conseguir que la función a realizar por el sistema embebido funcione de manera más eficiente y reducir los recursos necesarios, es decir, obtener un mayor rendimiento, se necesita optimizar el código conforme a la plataforma destino y configurar dicha plataforma hardware eligiendo la mejor alternativa posible. Para ello es necesario el uso de herramientas que simulen la ejecución del código en la plataforma.

Las simulaciones nos aportan datos aproximados a la realidad a través de modelos. Estos datos pueden ser: tiempo de ejecución, consumo, temperatura, uso de memoria, etc. Utilizando estas simulaciones, se puede hacer un estudio del rendimiento del código desarrollado sobre dicha plataforma, permitiendo adelantar las características del prototipo.

El análisis del modelo del sistema debe permitir la identificación de posibles mejoras del sistema real, así como detectar las variables que más influyen en el rendimiento del mismo. Además, cuanto más temprana es la simulación en las diferentes fases del desarrollo de un proyecto, menos costosa es la solución de errores.

Los sistemas embebidos se pueden encontrar en múltiples campos de aplicación, entre los que cabe destacar:

- Equipos de comunicaciones.
- Dispositivos dedicados al sector de consumo, por ejemplo, electrodomésticos, juguetes, etc.
- Automoción.
- Equipos industriales.
- Equipos para domótica.
- Sector aeroespacial.
- Etc.

De estos campos de aplicaciones, este trabajo se centra en el sector aeroespacial. Este sector tiene ciertas características, como no poder disipar calor ni permitir pesos elevados, así como ser resistente a las radiaciones. Para cumplir estas características se utilizan sistemas que incluyen procesadores diseñados específicamente para estas necesidades.

La Agencia Espacial Europea (ESA) utiliza en sus satélites el LEON, un procesador con arquitectura SPARC-V8 del cual se hablará en detalle más adelante. Este procesador se puede utilizar con distintas configuraciones, la simulación de esas configuraciones es donde se encuentra situado este trabajo.

Los sistemas embebidos en los que se va a trabajar en este proyecto son, por tanto, sistemas basados en procesadores, lo que implica que la parte más importante y la que ejecuta la mayor parte de la funcionalidad es el procesador. Por este motivo la simulación se debe centrar, principalmente, en ellos.

1.2 – OBJETIVOS

El objetivo de este trabajo es obtener un simulador con el que se pueda estimar el rendimiento de aplicaciones ejecutadas sobre plataformas con uno o varios procesadores LEON-3, con un error de estimación entorno al 10 % en códigos grandes.

Para conseguir este simulador se ha partido de un simulador previo creado por el Grupo de Ingeniería Microelectrónica de la Universidad de Cantabria, cuyo nombre es Vippe. Este simulador proporciona varias estimaciones durante la ejecución del código en la plataforma.

El mayor problema para nuestros objetivos es que fue creado para procesadores ARM, lo que implica que para poder utilizarle con procesadores LEON se tiene que modificar.

Para ello, se necesita conocer una serie de características como el número de ciclos que lleva ejecutar cada instrucción y cada acceso a caché. Esto no es un dato directo: para conocer su valor es necesario realizar una serie de pruebas sobre la plataforma con procesador LEON-3. Por otra parte, se tiene que conocer el porcentaje de error introducido por otros elementos particulares del LEON, como la carga que suponen las funciones extras que incluye el compilador utilizado.

Una vez conocidos los ciclos que lleva cada instrucción y los accesos a caché, se efectuará una simulación de varios programas en el simulador aplicando dichos datos y se comparará el resultado con el obtenido directamente sobre la plataforma con procesador LEON-3.

1.3 – ORGANIZACIÓN

La memoria del trabajo se divide en seis capítulos:

Introducción
Es el presente capítulo. Se explica el contexto en el que se encuentra este trabajo y los objetivos marcados.

Estado del arte
Es el capítulo que se encuentra a continuación, en el que se puede observar de manera detalla las técnicas de simulación más habituales.

Infraestructura y herramientas
En este capítulo se definen brevemente la infraestructura y las herramientas que han sido necesarias para la realización del trabajo.

Desarrollo del trabajo
Resume el trabajo desarrollado durante la realización del proyecto. Explicando cada paso de forma justificada y cómo se ha realizado la modificación del simulador.

Resultados
En este capítulo se verá la eficacia del simulador con varios ejemplos. Comparando los datos de la plataforma con los del simulador modificado.

Conclusiones
Por último, se realizarán algunos comentarios finales acerca del trabajo y continuidad de este proyecto.

2 – ESTADO DEL ARTE

Tradicionalmente, para simular un sistema electrónico se utilizan lenguajes como VHDL o SystemC. VHDL describe todos los detalles internos, totalmente necesarios para generar los chips pero no para simular porque produce simulaciones muy lentas. Por otra parte, SystemC no requiere tantos detalles de bajo nivel de cada componente porque trabaja a nivel alto, pero necesita mecanismos adicionales que permitan simular con precisión componentes complejos como los procesadores.

Teniendo en cuenta que el objetivo de este trabajo es simular sistemas basados en procesador, es necesario utilizar técnicas más específicas. Entre estas técnicas, las más habituales son: ISS, traducción binaria, simulación basada en trazas y simulación nativa. Estas cuatro técnicas se detallan a continuación.

2.1 – ISS

Es una de las técnicas más extendidas. Los ISS (Instruction Set Simulator) implementan todo el conjunto de instrucciones de un cierto procesador, de forma que pueden ejecutar los binarios finales que han sido compilados para esa plataforma [1].

Esta técnica suele ser muy precisa. Debido a esta precisión son capaces de garantizar la corrección funcional y no funcional del diseño. Sin embargo, en línea con el VHDL, también son lentos.

Otro inconveniente de esta técnica, es que para realizar la simulación requieren todo el conjunto de software, el software dependiente del hardware (HdS), el sistema operativo y las herramientas de compilación. Por lo tanto, solo se pueden aplicar una vez que el proceso de desarrollo está bastante avanzado.

2.2 – TRADUCCIÓN BINARIA

La traducción binaria es una técnica cercana a los ISS. Se basa en la traducción de software compilado para la plataforma destino (target) a instrucciones propias de la máquina en la que se realizará la simulación anfitrión (host) [1].

Con esto se logra conseguir una mejora en el rendimiento sobre los ISS, y se mantiene una buena precisión. Los tiempos de ejecución de la simulación son menores que con los ISS, pero aún pueden ser un poco elevados.

Comparte el inconveniente de requerir el conjunto de software, el software dependiente del hardware (HdS) y el sistema operativo para realizar la simulación. Por lo que, al igual que los ISS, solo se pueden aplicar una vez que el proceso de desarrollo está casi terminado y no en etapas tempranas del proceso.

2.3 – SIMULACIÓN BASADA EN TRAZAS

La simulación basada en trazas es la simulación del sistema realizada mediante la exploración de las trazas de ejecución del programa o accesos al componente del sistema como propósito de predecir el rendimiento.

La simulación basada en trazas suele tener dos componentes: una que ejecuta las acciones y almacena los resultados, es decir trazas, y otra que lee los archivos de registros de las trazas y los interpola en unos nuevos (a veces más complejos) escenarios. Esto hace que pueda ser lenta y de dudosa precisión, ya que no puede incluir efectos que dependan del análisis de las trazas.

2.4 – SIMULACIÓN NATIVA

La simulación nativa es una técnica de simulación que se basa en ejecutar la aplicación en el PC [2]. Por eso tiene menos dependencias de herramientas intermedias que permitan ejecutar el código software en la plataforma destino, ya que el código se compila para ser ejecutado como un programa normal de ordenador donde se realiza la simulación. Además la simulación es bastante rápida.

Para obtener estimaciones de rendimiento de dicho código se realizan determinadas modificaciones al código mediante técnicas de instrumentación y anotación. Para poder modelar sistemas hardware/software completos, la simulación del software se enlaza con modelos virtuales de la plataforma hardware. Todas estas extensiones permiten que la estimación de rendimiento del sistema tenga la precisión suficiente para permitir la evaluación y exploración de los sistemas embebidos en desarrollo.

3 – INFRAESTRUCTURA Y HERRAMIENTAS

Para realizar este trabajo ha sido necesario utilizar una infraestructura basada en el procesador LEON y varias herramientas auxiliares. Cada una de ellas será explicada a continuación de forma detallada.

3.1 – LEON

El LEON, tal y como se ha mencionado anteriormente, es un procesador SPARC-V8, que fue diseñado originalmente por el Centro Europeo de Investigación y Tecnología del Espacio (European Space Research and Technology Centre, ESTEC), que es parte de la Agencia Espacial Europea (ESA). Actualmente, el procesador y las herramientas necesarias para su uso son desarrolladas y mantenidas por la empresa Gaisler Research.

El LEON se diseñó con el objetivo de disponer de un procesador capaz de presentar resistencia a las radiaciones del espacio exterior, que fuese fácilmente 'portable', con interfaces estándar y que ejecutase hasta 100 MIPS (Millones de Instrucciones Por Segundo). Con este propósito, se desarrolló funcionalidad del mismo implementando un mínimo de interfaces y de funciones. A este primer prototipo se le llamó LEON-1.

Posteriormente, a esta primera versión se le añadió una unidad de punto flotante para conseguir un procesador más completo y un mayor número de periféricos, dando lugar a las nuevas versiones LEON-2 y LEON-3. Este último procesador es el más utilizado actualmente por la Agencia Espacial Europea (ESA) en sus satélites y en el cual se centra este trabajo.

3.1.1 – ELEMENTOS CONVENCIONALES

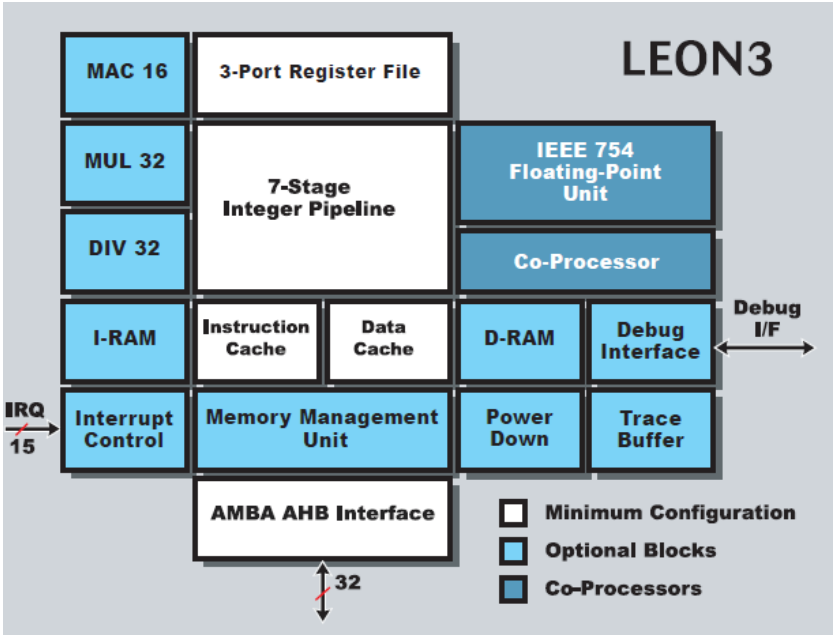

Figura 1. Arquitectura LEON-3

En la *Figura 1* se puede observar la arquitectura del LEON-3, un procesador de 32 bits implementado en lenguaje VHDL [3]. En ella se representan la mayor parte de los elementos que conforman la arquitectura interna de este procesador. Entre sus características se puede apreciar que:

- dispone de un pipeline de 7 etapas,
- dispone de cachés para datos e instrucciones (arquitectura Harvard),
- tiene una unidad de punto flotante (FPU) IEEE-754 que proporciona soporte para operaciones de punto flotante de simple y doble precisión,
- incluye un soporte avanzado para depuración dentro del chip,

- tiene coprocesadores que le proporcionan compatibilidad con el conjunto de instrucciones SPARC-V8 completo, incluyendo las instrucciones MUL, MAC y DIV.

El procesador puede ser utilizado en configuraciones de multiprocesamiento asimétrico (AMP) o multiprocesamiento simétrico (SMP). Por otra parte, el núcleo básico del procesador puede ser implementado en tecnologías ASIC y FPGA.

3.1.2 – PERIFÉRICO L3STAT

Además de los componentes descritos anteriormente, en este proyecto ha sido importante el uso de otro componente diseñado para el LEON. El L3STAT (LEON3 Statistics Unit) es un periférico que va conectado al bus y que se utiliza para contar los eventos que ocurren en el procesador y en el bus AHB, con el fin de crear estadísticas de rendimiento para aplicaciones software [4].

El L3STAT consta de un número configurable de contadores de 32-bit, que se incrementan en un determinado evento. Cada contador tiene un registro de control donde se selecciona el tipo de evento. Los contadores vuelven a cero cuando alcanzan su valor máximo, pero también pueden borrarse de forma automática leyéndolos para facilitar la creación de estadísticas durante periodos más largos. En los sistemas con múltiples procesadores, los registros de control también indican que núcleo del procesador se está controlando.

En la *Tabla 1* se encuentran algunos de los eventos del periférico L3STAT, con sus respectivos indicadores.

ID	Event description
Processor events:	
0x00	Instruction cache miss
0x01	Instruction MMU TLB miss
0x02	Instruction cache hold
0x03	Instruction MMU hold
0x08	Data cache (read) miss
0x09	Data MMU TLB miss
0x0A	Data cache hold
0x0B	Data MMU hold
0x10	Data write buffer hold
0x11	Total instruction count
0x12	Integer instructions
0x13	Floating-point unit instruction count
0x14	Branch prediction miss
0x15	Execution time, excluding debug mode
0x17	AHB utilization (per AHB master) (implementation depedent)
0x18	AHB utilization (total, master/CPU selection is ignored) (implementation dependent)
0x22	Integer branches
0x28	CALL instructions
0x30	Regular type 2 instructions
0x38	LOAD and STORE instructions
0x39	LOAD instructions
0x3A	STORE instructions
AHB events (only available if core is connected to a LEON3 Debug Support Unit):	
0x40	AHB IDLE cycles.
0x41	AHB BUSY cycles.
0x42	AHB NON-SEQUENTIAL transfers. Filtered on CPU/AHBM if SU(1) = '1'
0x43	AHB SEQUENTIAL transfers. Filtered on CPU/AHBM if SU(1) = '1'
0x44	AHB read accesses. Filtered on CPU/AHBM if SU(1) = '1'
0x45	AHB write accesses. Filtered on CPU/AHBM if SU(1) = '1'

Tabla 1. Captura del documento GRLIB IP Core User's Manual con la lista de eventos

3.2 – PLACA VIRTEX-5 ML506

Como el LEON3 se distribuye en VHDL, una de sus aplicaciones principales es integrarlo en una FPGA junto con elementos de memoria y entrada salida.

En el proyecto se ha utilizado para este fin la placa Virtex-5 ML506. Por ello, este es el sistema real que se tiene, y será el modelo de referencia que usaremos para configurar y probar el simulador que desarrollaremos. Por lo tanto, las medidas se realizan sobre esta placa.

En la *Figura 2* se pueden observar los componentes por los que está formada dicha placa [5].

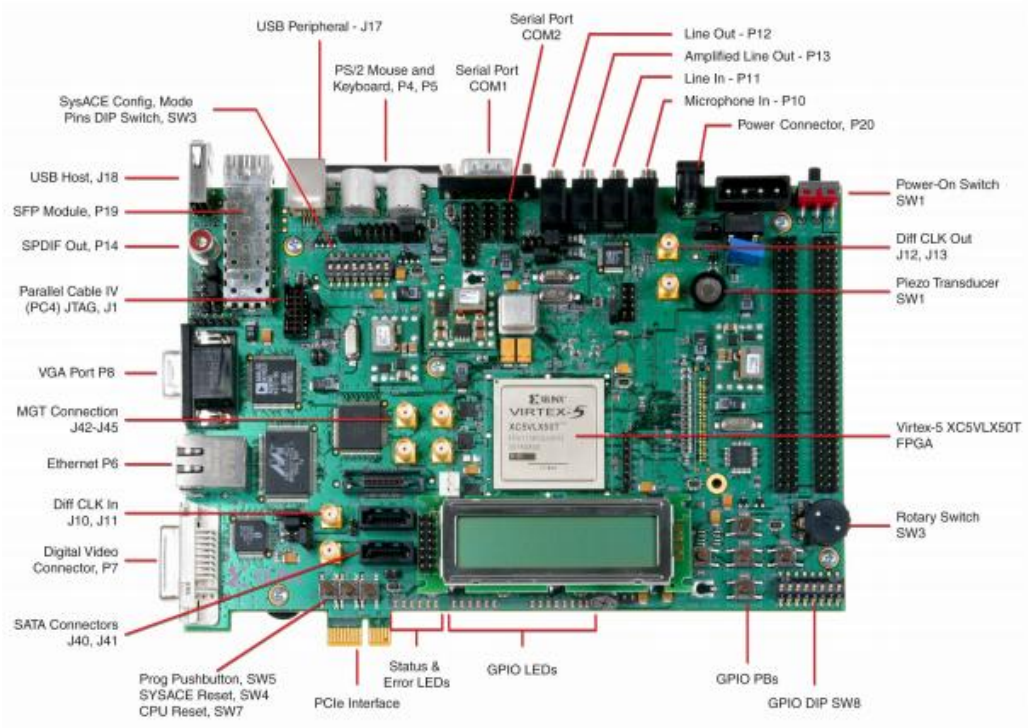

Figura 2. Componentes de la placa ML506

Sus principales características son:

- FPGA Xilinx Virtex-5 XC5VSX50T,
- dos plataformas PROM Flash XCF32P, de 32 Mb cada una, para almacenar grandes configuraciones de los dispositivos,
- ancho de 64-bit, DDR2 de 256 MB, compatible con el soporte EDK y controladores de software,
- reloj
    - chip de generador de reloj de sistema programable,
    - sincronización externa a través de SMA (dos pares diferenciados),
- interruptores DIP (8) de uso general, LEDs (8) y pulsadores,
- ZBT SRAM, 9 Mb en bus de datos de 32 bits con cuatro bits de paridad,
- Chip Intel P30 StrataFlash (32 MB),
- SPI (Serial Peripheral Interface) de 2MB.
- Puerto de configuración JTAG para su uso con cable paralelo III, IV o cable de descarga USB.

En este trabajo se han realizado medidas sobre esta placa con uno y dos procesadores LEON-3.

3.3 – COMPILADORES

Para ejecutar el código de aplicación en esta placa es necesario compilarlo para ella. Sin embargo, dependiendo del compilador, la versión y las opciones que utilicemos, los resultados de rendimiento pueden ser distintos. Por ello en el proyecto se han probado distintas alternativas, que se describen a continuación.

3.3.1 – GCC

GCC (**G**NU **C**ompiler **C**ollection) es un compilador para C, C++, Objective C y Fortran rápido, muy flexible y riguroso con el estándar de C ANSI. Puede funcionar como compilador cruzado para un gran número de arquitecturas distintas. Se encarga de realizar (o encargar el trabajo a otras utilidades) el preprocesado del código, la compilación y el enlazado. Dicho de otra manera, nosotros proporcionamos un código fuente en cualquiera de los lenguajes mencionados anteriormente y él nos devuelve un archivo binario compilado para nuestra arquitectura.

A continuación se describirán las opciones más habituales ('flags') al usar este compilador:

- -help: indica a gcc que muestre su salida de ayuda (muy reducida).
- -o <fichero>: el archivo ejecutable generado por gcc es por defecto a.out. Mediante este modificador, se especifica el nombre del ejecutable.
- -g: incluye en el archivo binario información necesaria para utilizar un depurador posteriormente.
- -S: preprocesa y compila, pero no ensambla ni enlaza.

- -O <nivel>: indica a gcc que utilice optimizaciones en el código. Los niveles posibles van desde 0 (no optimizar) hasta 3 (optimización máxima). Utilizar el optimizador aumenta el tiempo de compilación, pero suele generar ejecutables más rápidos.

En este trabajo se han utilizado dos versiones diferentes de gcc propocionadas por Gaisler para el LEON-3. Estas versiones son la 4.4.2 y la 3.4.4. Además, con la versión 4.4.2 hay dos alternativas, que permiten que las medidas se realicen con y sin sistema operativo (Bare C o Bare metal). La versión 3.4.4, al igual que cuando se tienen dos cores, se realizan las medidas sin sistema operativo.

El procesador LEON-3 admite una serie de atributos de compilación, los utilizados en este trabajo son los siguientes:

- -msoft-float: compila sin utilizar FPU
- -mcpu=v8: se cambia la versión de sparc, por defecto v7, a una v8

3.3.2 – LLVM

LLVM (**L**ow **L**evel **V**irtual **M**achine) es una infraestructura para desarrollar compiladores, que está diseñada para optimizar el tiempo de compilación, el tiempo de enlazado y el tiempo de ejecución de cualquier lenguaje de programación que el usuario quiera definir.

LLVM suministra las capas intermedias de un sistema de compilado completo, tomando el código en lenguaje intermedio llamado forma intermedia (Intermediate Form, IF) de un compilador y emitiendo un IF optimizado. Este nuevo IF puede ser convertido y enlazado en un código ensamblador dependiendo de la máquina concreta para una plataforma objetivo.

En este trabajo se utiliza un compilador llvm llamado Clang, utilizado una versión que es la 3.5.0. En diferencia a gcc, en la versión disponible este

compilador no dispone de los atributos específicos para LEON. Como consecuencia de esto, solo se puede obtener un código genérico para Sparc.

3.4 – LINUX-BUILD

Linux-build es una utilidad de configuración gráfica, archivos de configuración y un conjunto de pequeños scripts Makefile para construir y configurar una distribución de Linux que pueda ejecutar en la placa.

Linux-build proporciona una manera rápida de empezar el desarrollo de Linux para una arquitectura LEON. Vincula diferentes componentes para construir un completo entorno Linux. Cada componente es diseñado para utilizarse por separado el uno del otro o juntos desde Linux-build. Si se configura por separado hay que establecer las mismas opciones de configuración en cada uno.

3.5 – IMPACT

IMPACT es una herramienta de Xilinx que realiza dos funciones básicas. La primera de ellas es la generación de ficheros de configuración en diferentes formatos, y la segunda es descargar el programa en la placa a través de un cable de comunicación.

3.6 – GRMON

GRMON monitor general de depuración para el procesador LEON, y para diseños SOC basados en la librería GRLIB IP, que sirve para descargar y ejecutar aplicaciones para LEON en placa. Incluye las siguientes funcionalidades:

- Acceso de lectura/escritura a todos los registros del sistema y memoria.
- Desensamblador incorporado y almacenamiento de seguimiento.
- Gestión de puntos de interrupción.

- Conexión remota al depurador GNU (gdb).

Al igual que con TSIM, se ha tenido que trabajar con una versión de evaluación.

3.7 – SIMULADORES

3.7.1 – TSIM

TSIM es un simulador a nivel de instrucción capaz de emular sistemas basados en ERC32 o LEON [6]. Sus características son:

- ejecuta hasta 60 MIPS en ordenadores de alta gama,
- modo de espera del procesador acelerado (velocidades de simulación más rápidas que en tiempo real),
- funcionamiento autónomo,
- conexión remota al depurador GNU (gdb),
- módulos cargables que incluyen dispositivos Entrada/Salida definidos por el usuario,
- buffer de seguimiento de instrucciones,
- número ilimitado de puntos de interrupción (breakpoints),
- emulación MMU (LEON-3/4).

Está disponible en cuatro configuraciones, que son: ERC32, LEON-2, LEON-3 y emulación LEON-4. Y con él son compatibles cuatro plataformas anfitrión (host): Linux, Linux x86-64, Solaris y Windows.

Tal y como se ha mencionado en las características, se puede ejecutar en modo autónomo o conectado a través de una toma de red para el depurador gdb. En modo autónomo, una variedad de comandos de depurado están disponibles para permitir la manipulación del contenido de la memoria y los registros, punto de interrupción o de observación y medidas del rendimiento. Conectado a gdb actúa como un destino remoto y apoya las peticiones de depuración. La comunicación entre gdb y TSIM se

lleva a cabo utilizando el protocolo extendido gdb, pudiendo utilizar depuradores de terceros siempre que estén basados en este protocolo.

Como los ISS se caracterizan por proporcionar medidas precisas, este simulador también se utilizará como modelo de referencia para complementar a la placa.

Para hacer este trabajo se ha tenido que trabajar con una versión de evaluación de TSIM. Como consecuencia de trabajar con esta versión se tiene un tamaño limitado en memoria, por eso se han tenido que adaptar los datos de manera que se pudiese ejecutar en este simulador.

3.7.2 – VIPPE

Es una herramienta creada por el Grupo de Ingeniería Microelectrónica de la Universidad de Cantabria, y su nombre viene de "**V**irtual **P**latform **P**arallel performance **E**valuation" [1].

Se sitúa dentro de la simulación nativa. El código de aplicación se anota con código adicional durante la compilación, el cual proporciona varias estimaciones durante la ejecución del código en la plataforma anfitrión (host). Estas estimaciones pueden ser, por ejemplo, tiempo de ejecución y el número de accesos a memoria.

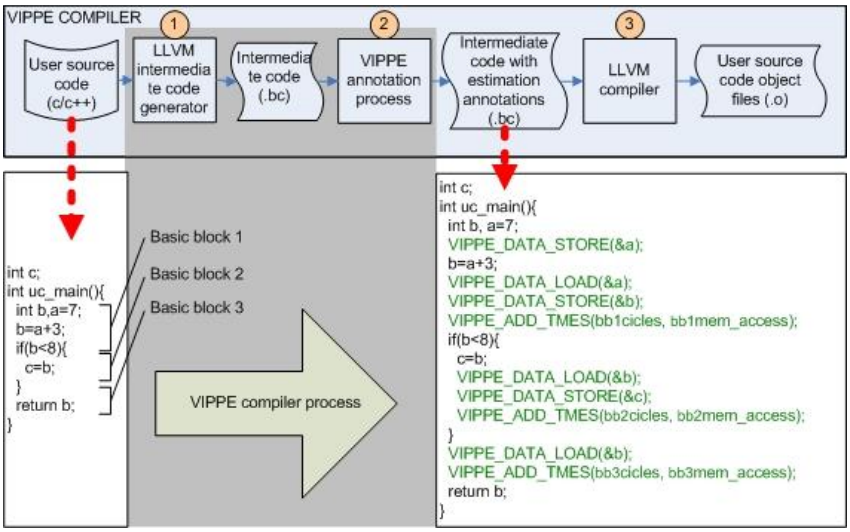

Figura 3. Instrumentación de código

En la *Figura 3* se puede observar el código adicional introducido. Estas líneas de código adiciones se insertan en cada bloque básico del programa durante el proceso de compilación. Se consideran bloques básicos a las secciones del código que se ejecutan sin ninguna sentencia de control (condicionales y bucles). Para ejecutar este código en la plataforma anfitrión (host) se emplean librerías específicas de simulación.

Tiene una sincronización asíncrona, en la que los diferentes hilos (threads) concurrentes de la aplicación pueden simularse de forma independiente, sincronizándose solo durante las operaciones de comunicación. Los hilos sólo se bloquean en caso de que se vaya a realizar una lectura de una variable compartida o un elemento de sincronización (como semáforos), este enfoque minimiza el bloqueo y mejora la ejecución en paralelo.

Los hilos destino (target) simulados son responsables del modelado de la funcionalidad del software y de informar al hilo anfitrión (host), situado en el kernel, de las estimaciones de hilos individuales como el tiempo y el acceso a memoria. El hilo anfitrión, conociendo dicha información, es responsable de modelar el sistema.

Para compartir esta información entre el hilo anfitrión y los hilos destino se utilizan variables compartidas. Finalmente, el kernel maneja el tiempo físico de la simulación o el tiempo global.

Esta herramienta contabiliza el número de instrucciones de un código cualquiera. A partir de este dato, la herramienta permite conocer otros datos del código como el número de ciclos y fallos de datos e instrucciones. Sin embargo el número de ciclos no era correcto, debido a que el valor era igual al número de instrucciones. Por eso, este trabajo tiene por objetivo modificar este simulador para que el número de ciclos sea preciso.

En un primer momento, en este tipo de simuladores se implementaron mecanismos para realizar este proceso analizando y anotando desde código fuente. Esto da mucha flexibilidad a la herramienta, ya que se puede usar el mismo compilador para realizar la estimación que el que se usará en la

realidad, pero puede dar problemas con alguna optimización del compilador.

Para solucionar este problema, Vippe utiliza por defecto un análisis de código intermedio del compilador llvm (clang). Esto reduce el problema de las optimizaciones, pero puede dar errores de estimación si el compilador utilizado en la realidad no es clang, por ejemplo gcc. Además esta alternativa aún está en desarrollo y puede presentar pequeños problemas de precisión.

Por ello en el proyecto se ha trabajado con ambas versiones: la alternativa basada en código intermedio que incluye Vippe por defecto y una adaptación de la versión de código fuente que ha sido integrada en Vippe.

4 – DESARROLLO DEL TRABAJO

4.1 – ETAPAS DESARROLLADAS

En la realización de este trabajo se pueden diferenciar claramente cuatro etapas.

La primera etapa tiene como objeto crear la plataforma y familiarizarse con el entorno y las herramientas, realizando lo necesario para poder realizar las primeras medidas del sistema real.

En la segunda etapa se realiza la evaluación de distintas configuraciones en la arquitectura hardware y el compilador (Aplicación, SO, openmp…), para lo que se ha ejecutado distintas versiones de código en el simulador Tsim de Gaisler y en la placa real Virtex-5 ML506.

La tercera etapa se centra en obtener la información que necesitará el simulador Vippe para modelar correctamente una plataforma basada en el procesador LEON-3. Para ello se ha utilizado la placa, obteniendo la información del número de ciclos, instrucciones y fallos de cache con el periférico l3stat.

Y, por último, en la cuarta etapa se ha procedido a aplicar esta información en Vippe, estimando códigos de aplicación reales. Para comprobar las estimaciones obtenidas se han aplicado las dos alternativas del punto anterior para las medidas de tiempos reales y los dos mecanismos de anotación para obtener las estimaciones de tiempos en Vippe, descritos en el punto 3. Estos cuatro mecanismos se han utilizado ejecutando ejemplos pequeños y un código de codificación de vídeo según el estándar CCSDS, todo ello sin sistema operativo (bare C).

4.2 – DESARROLLO DE LAS ETAPAS

4.2.1 – PRIMERA ETAPA

Tal y como se ha comentado en el apartado anterior en esta parte se ha realizado lo necesario para obtener las primeras medidas sobre la plataforma, familiarizándose con el entorno y las herramientas.

En primer lugar se tiene que realizar la configuración del procesador y de la plataforma hardware. Una vez que se tiene la configuración hay que generar un .bit, el archivo binario que contiene el hardware necesario para la plataforma. Esto se ha realizado para una plataforma con un "core" LEON3 y para otra con dos.

Ambos pasos se pueden realizar utilizando los comandos incluidos en los Makefiles proporcionados por Gaisler junto con los códigos VHDL. Estos comandos abren una interfaz gráfica para configurar la plataforma de forma sencilla (*Figura 4*) y permiten ejecutar las instrucciones necesarias para sintetizar un .bit para la placa correspondiente.

A continuación, y teniendo el archivo .bit, se abre la herramienta IMPACT para programar la FPGA de la placa con el .bit que se necesite en cada momento, en nuestro caso el de uno o dos "cores".

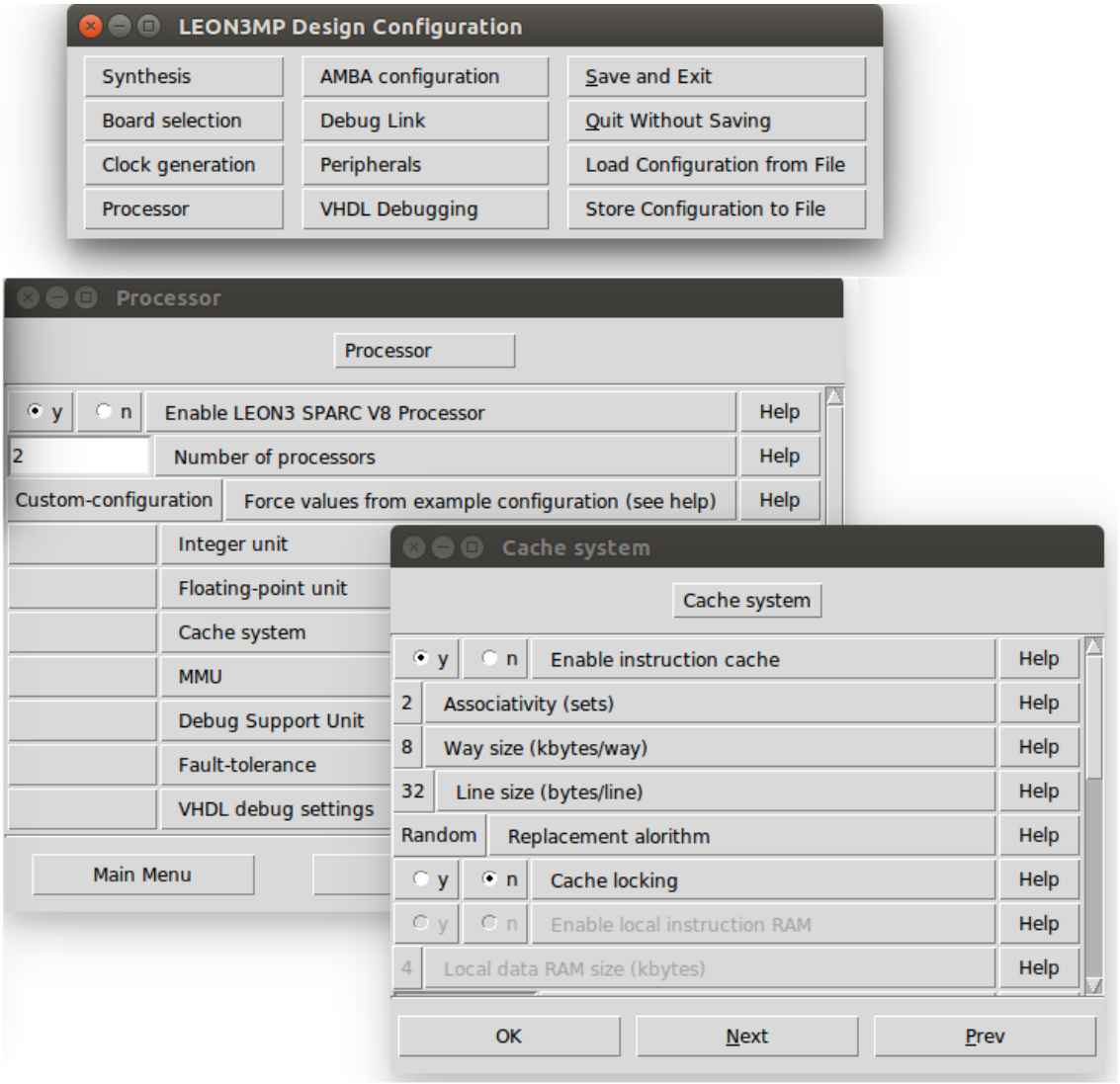

Figura 4. Configuración de la plataforma física

Teniendo la plataforma programada, se utiliza la herramienta Grmon para conectar con la placa. En esta herramienta, la opción 'infosys' informa de los componentes hardware que hay en la plataforma, y algunos detalles importantes como son las direcciones. En la *Figura 5* se puede observar el resultado de ejecutar esa opción.

```
grmon2> info sys
  cpu0      Cobham Gaisler  LEON3 SPARC V8 Processor
            AHB Master 0
  ahbuart0  Cobham Gaisler  AHB Debug UART
            AHB Master 1
            APB: 80000700 - 80000800
            Baudrate 115200, AHB frequency 60.00 MHz
  ahbjtag0  Cobham Gaisler  JTAG Debug Link
            AHB Master 2
  svga0     Cobham Gaisler  SVGA frame buffer
            AHB Master 3
            APB: 80000600 - 80000700
            clk0: 25.00 MHz  clk1: 25.00 MHz  clk2: 40.00 MHz  clk3: 65.00 MHz
  greth0    Cobham Gaisler  GR Ethernet MAC
            AHB Master 4
            APB: 80000B00 - 80000C00
            IRQ: 7
            edcl ip 192.168.0.51, buffer 8 kbyte
  ddr2spa0  Cobham Gaisler  Single-port DDR2 controller
            AHB: 40000000 - 80000000
            AHB: FFF00100 - FFF00200
            64-bit DDR2 : 1 * 256 MB @ 0x40000000, 4 internal banks
            190 MHz, col 10, ref 7.8 us, trfc 131 ns
  apbmst0   Cobham Gaisler  AHB/APB Bridge
            AHB: 80000000 - 80100000
  dsu0      Cobham Gaisler  LEON3 Debug Support Unit
            AHB: 90000000 - A0000000
            AHB trace: 128 lines, 32-bit bus
            CPU0:  win 8, hwbp 2, itrace 128, V8 mul/div, srmmu, lddel 1, GRFPU-
lite
                   stack pointer 0x4ffffff0
                   icache 2 * 8 kB, 32 B/line
                   dcache 4 * 4 kB, 16 B/line rnd, snoop tags
  mctrl0    European Space Agency  LEON2 Memory Controller
            AHB: 00000000 - 20000000
            AHB: 20000000 - 40000000
            AHB: C0000000 - C2000000
            APB: 80000000 - 80000100
            16-bit prom @ 0x00000000
            32-bit static ram: 1 * 1024 kbyte @ 0xc0000000
  adev9     Cobham Gaisler  System ACE I/F Controller
            AHB: FFF00200 - FFF00300
            IRQ: 3
  apbmst1   Cobham Gaisler  AHB/APB Bridge
            AHB: 80100000 - 80200000
  uart0     Cobham Gaisler  Generic UART
            APB: 80000100 - 80000200
            IRQ: 2
            Baudrate 38461, FIFO debug mode
  irqmp0    Cobham Gaisler  Multi-processor Interrupt Ctrl.
            APB: 80000200 - 80000300
  gptimer0  Cobham Gaisler  Modular Timer Unit
            APB: 80000300 - 80000400
            IRQ: 8
            8-bit scalar, 2 * 32-bit timers, divisor 60
  ps2ifc0   Cobham Gaisler  PS2 interface
            APB: 80000400 - 80000500
            IRQ: 4
  ps2ifc1   Cobham Gaisler  PS2 interface
            APB: 80000500 - 80000600
            IRQ: 5
  gpio0     Cobham Gaisler  General Purpose I/O port
            APB: 80000800 - 80000900
  i2cmst0   Cobham Gaisler  AMBA Wrapper for OC I2C-master
            APB: 80000900 - 80000A00
            IRQ: 6
            Index for use in GRMON: 0
  i2cmst1   Cobham Gaisler  AMBA Wrapper for OC I2C-master
            APB: 80000C00 - 80000D00
            IRQ: 11
            Index for use in GRMON: 1
  ahbstat0  Cobham Gaisler  AHB Status Register
            APB: 80000F00 - 80001000
            IRQ: 1
  l3stat0   Cobham Gaisler  LEON3 Statistics Unit
            APB: 80100100 - 80100200
            cpus: 1, counters: 8, i/f index: 0
```

Figura 5. Resultado de 'info sys'

Cuando se ha obtenido el hardware, se ha pasado a adaptar el software. Se ha tenido que extender el código del CCSDS permitiendo utilizar como entrada un archivo de imagen o un array inicializado estáticamente con los datos de dicha imagen (La versión inicial solo permitía leer los datos desde un archivo). Con este cambio se pueden acceder a los datos de entrada y comprobar los datos de salida sin tener un sistema operativo.

También se han realizado una serie de pruebas para ajustar el tamaño de imagen con la que se opera, para que los datos entren en la memoria de Tsim, que al ser la versión gratuita, está limitada. Se ha obtenido como resultado que el tamaño de la imagen o del array tiene que ser de 200x200.

Una vez que se tiene el software y el hardware listos, ya se pueden probar distintas configuraciones en la plataforma.

4.2.2 – SEGUNDA ETAPA

En esta segunda etapa, se quiere conocer la diferencia a la hora de utilizar distintas configuraciones. Para obtener estas diferencias, se han realizado una serie de pruebas con diferentes parámetros sobre el código CCSDS. Para ello, se ha utilizado el simulador Tsim (solo en modo Bare C) e implementaciones en placa con uno y dos "cores".

En primer lugar, se ha querido comparar distintas unidades funcionales. Para ello se han probado alternativas con las configuraciones: sparc v7 con FPU, sparc v7 sin FPU ('-msoft-float'), sparc v8 con FPU ('-mcpu=v8') y sparc v8 sin FPU ('-msoft-float –mcpu=v8'). Para que el compilador utilice cada uno de esos parámetros, se han puesto esas 'flags' en la línea de compilación. Sin embargo, medir el impacto de las unidades funcionales que se activan con las 'flags', solo ha sido posible realizarlas con gcc, porque llvm ignora los parámetros.

En la *Tabla 2* se puede observar el tiempo que tarda el código en ejecutarse con el compilador gcc versión 4.4.2 en la placa, teniendo en cuenta que el código está compilado sin sistema operativo, es decir Bare C, y con un tamaño de imagen 200x200.

	V7 FPU	V7 NO FPU	V8 FPU	V8 NO FPU
DWT	0.36752	4.5485	0.35123	4.5221
Total (DWT+BPE)	0.85864	5.0406	0.84017	5

Tabla 2. Compilador gcc -O2. 200x200. Placa

Analizando la *Tabla 2*, se puede observar que la diferencia entre sparc v7 y sparc v8 es mínima.

Los datos de la *Tabla 2* son los que han sido utilizados a la hora de realizar las comparaciones con las demás medidas como referencia, calculando con ellos la diferencia.

En la *Tabla 3* se representa el tiempo que tarda el código en ejecutarse con el compilador llvm en la placa, compilado sin sistema operativo y con un tamaño de imagen de 200x200. Tal y como se ha dicho anteriormente, aquí solo se tiene el dato de sparc v7 FPU porque los demás parámetros los ignora a la hora de la compilación.

	V7 FPU (Ref)	V7 FPU	Diferencia
DWT	0.36752	0.39154	6.54 %
Total (DWT+BPE)	0.85864	0.95451	11.17 %

Tabla 3. Compilador llvm. 200x200. Placa

A continuación, en la *Tabla 4* y la *Tabla 5*, se pueden observar los mismos tiempos pero en lugar de ser ejecutados en placa son ejecutados en el simulador Tsim.

	V7 FPU		V7 NO FPU		V8 FPU		V8 NO FPU	
		Diferen.		Diferen.		Diferen.		Diferen.
DWT	0.30009	-18.35%	6.7517	48.44%	0.28362	-19.25%	6.3975	41.47%
Total	0.83842	-2.36 %	7.2964	44.75%	0.79011	-5.96%	6.9021	38.04%

Tabla 4. Compilador gcc. 200x200. Tsim. Como referencia la Tabla 2

	V7 FPU (Ref)	V7 FPU	Diferencia
DWT	0.36752	0.32051	-12.79%
Total	0.85864	0.94716	10.31%

Tabla 5. Compilador llvm. 200x200. Tsim

Como se puede observar en las *Tablas 4* y 5, los errores del Tsim son del 20% con FPU y del 50% sin ella. A primera vista, sin realizar ninguna prueba más, los números indican que Tsim supone que la FPU es más rápida de lo que en realidad es, y que la unidad de enteros es más lenta.

Seguidamente, se ha querido comparar las distintas opciones de compilación. Para ello el compilador de referencia gcc 4.4.2 con el 'flag' –O2 se ha comparado con gcc 4.4.2 –O3 y con gcc 3.4.4. Estas medidas se han realizado compilando el código sin sistema operativo y con una imagen de 200x200. Estos resultados se pueden observar en las *Tablas 6* y *7*.

	V7 FPU		V7 NO FPU		V8 FPU		V8 NO FPU	
		Diferen.		Diferen.		Diferen.		Diferen.
DWT	0.3651	-0.66%	4.5414	-0.16%	0.3487	-0.72%	4.5219	-0.01%
Total	0.8159	-4.98%	4.994	-0.92%	0.7932	-5.59%	4.9701	-0.59%

Tabla 6. Compilador gcc 4.4.2 -O3. 200x200. Placa. Como referencia la Tabla 2

En la *Tabla 6* se puede observar que la diferencia de compilar con –O2 o –O3 es mínima en DWT, pero no tanto en BPE.

	V7 FPU		V7 NO FPU		V8 FPU		V8 NO FPU	
		Diferen.		Diferen.		Diferen.		Diferen.
DWT	0.3699	0.65%	9.0695	99.39%	0.3474	-1.09%	9.0395	99.90%
Total	0.9028	5.14%	9.6092	90.64%	0.8787	4.59%	9.5755	91.51%

Tabla 7. Compilador gcc 3.4.4. 200x200. Placa. Como referencia la Tabla 2

En la *Tabla 7* se observa que la diferencia entre gcc 4.4.2 y gcc 3.4.4 es pequeña cuando se utiliza la FPU, pero muy grande si no se utiliza.

En tercer lugar, se ha querido comparar la diferencia que hay teniendo sistema operativo. Tal y como se comentó en el apartado anterior, utilizando un sistema operativo Linux se tienen dos opciones para leer los datos de entrada y salida, una de ellas utilizando un archivo y la otra utilizando un array. Por lo tanto, se han hecho dos comparaciones, una ejecutando el código con sistema operativo con arrays y la otra con sistema operativo con archivos.

A continuación en las *Tablas 8* y *9* se pueden observar estos resultados, tanto con compilador gcc como con llvm.

	V7 FPU		V7 NO FPU		V8 FPU		V8 NO FPU	
Gcc		Diferen.		Diferen.		Diferen.		Diferen.
DWT	0.42688	16.15%	5.0402	10.81%	0.41045	16.86%	3.5695	-21.07%
Total	0.90027	4.85%	5.5125	9.36%	0.88213	4.99%	4.0424	-19.15%
Llvm								
DWT	0.43737	19.01%						
Total	0.99618	16.02%						

Tabla 8. Compilación con SO (Arrays). 200x200. Placa. Como referencia la Tabla 2

	V7 FPU		V7 NO FPU		V8 FPU		V8 NO FPU	
Gcc		Diferen.		Diferen.		Diferen.		Diferen.
DWT	0.4278	16.40%	5.0421	10.85%	0.41241	17.42%	3.5621	-21.23%
Total	0.91913	7.04%	5.5347	9.80%	0.90397	7.59%	4.0546	-18.91%
Llvm								
DWT	0.43754	19.05%						
Total	1.0278	19.70%						

Tabla 9. Compilación con SO (Archivos). 200x200. Placa. Como referencia la Tabla 2

Estas medidas solo se han podido realizar con la placa, ya que el simulador Tsim no funciona con sistema operativo, solo funciona con BareC.

Observando ambas tablas se puede decir que la diferencia entre usar arrays o archivos es mínima.

En cuarto lugar, se quiere comparar las medidas del código normal con las medidas del código utilizando openmp, para ver si se obtiene mejora o no. Estas medidas solo se pueden hacer con el compilador gcc, ya que el soporte openmp en llvm no está disponible. Debido a que se añade openmp, se harán las comparaciones entre utilizar un core o dos.

	V7 FPU		V7 NO FPU		V8 FPU		V8 NO FPU	
		Diferen.		Diferen.		Diferen.		Diferen.
DWT	0.43413	18.12%	5.0459	10.94%	0.41329	17.67%	3.5752	-20.94%
Total	0.90789	5.74%	5.5064	9.24%	0.86158	2.55%	4.0237	-19.53%

Tabla 10. Un core. SO (Arrays). OpenMP. 200x200. Placa. Como referencia la Tabla 2

	V7 FPU		V7 NO FPU		V8 FPU		V8 NO FPU	
		Diferen.		Diferen.		Diferen.		Diferen.
DWT	0.43345	17.94%	5.0559	11.16%	0.41433	17.97%	3.5885	-20.65%
Total	0.92427	7.64%	5.5332	9.77%	0.88023	4.77%	4.0054	-19.89%

Tabla 11. Un core. SO (Archivos) OpenMP. 200x200- Placa. Como referencia la Tabla 2

	V7 FPU			V7 NO FPU		
		Difer.(gcc)	Difer.(gcc+oMP)		Difer.(gcc)	Difer.(gcc+oMP)
DWT	0.3046	-17.12%	-29.84%	2.6597	-41.53%	-47.29%
Total	0.8021	-6.58%	-11.65%	3.1606	-37.30%	-42.60%
	V8 FPU			V8 NO FPU		
		Difer.(gcc)	Difer.(gcc+oMP)		Difer.(gcc)	Difer.(gcc+oMP)
DWT	0.2854	-18.74%	-30.94%	1.953	-56.81%	-45.37%
Total	0.7829	-6.82%	-9.13%	2.4457	-51.09%	-39.22%

Tabla 12. Dos cores. SO(Arrays) OpenMP.200x200. Placa. Como referencia Tabla 2 y 9

	V7 FPU			V7 NO FPU		
		Difer.(gcc)	Difer.(gcc+oMP)		Difer.(gcc)	Difer.(gcc+oMP)
DWT	0.3052	-16.96%	-29.59%	2.6654	-41.40%	-47.28%
Total	0.8151	-5.07%	-11.81%	3.1759	-36.99%	-42.60%
	V8 FPU			V8 NO FPU		
		Difer.(gcc)	Difer.(gcc+oMP)		Difer.(gcc)	Difer.(gcc+oMP)
DWT	0.2845	-19%	-31.33%	1.9351	-57.21%	-46.07%
Total	0.7913	-5.82%	-10.10%	2.4529	-50.94%	-38.76%

Tabla 13. Dos cores. SO(Archivos) OpenMP. 200x200. Placa. Como referencia Tabla 2 y 10

Utilizando OpenMP en dos cores mejora algo en tiempo de DWT. Puede que con un uso más intensivo de OpenMP diese mejores resultados.

Por último, en quinto lugar, se hacen medidas con diferente tamaño de imagen, se cambia el tamaño de imagen a 1000x1000. Con este tamaño de imagen solo ha sido posible realizarlo en placa, ya que la gestión de memoria en Tsim no acepta ese tamaño debido a estar en la versión de evaluación.

En el Anexo 1 se encuentran las mismas tablas realizadas anteriormente, pero con un tamaño de 1000x1000.

Analizando las tablas del Anexo 1 y comparándolas con las anteriores, se puede destacar que utilizando dos "cores" se mejora el tiempo del DWT sobretodo en imágenes grandes.

Una vez que se tienen todos estos datos, en los que se ha obtenido los resultados con las diferentes opciones posibles. A partir de ahora, se trabajará con el código en Bare C con los dos compiladores, gcc y llvm.

4.2.3 – TERCERA ETAPA

En esta tercera etapa, se quiere obtener la información que necesita el simulador Vippe para modelar correctamente una plataforma basada en LEON-3, consiguiendo que el número de instrucciones y de ciclos en un código sea preciso. Para conseguir este objetivo se trabaja con el código en BareC, y con los compiladores gcc y llvm.

En primer lugar, Vippe es capaz de estimar el número de instrucciones ensamblador ejecutadas por un código de aplicación, pero no el número de ciclos. Para obtener los ciclos se precisa saber el número de ciclos que tarda cada instrucción ensamblador en ejecutarse. Estos ciclos serán medidos en la placa Virtex-5 ML506, sin embargo, este dato no lo proporciona la placa directamente, sino que se tiene que activar el periférico l3stat.

En este periférico se pueden elegir una serie de eventos a contar (mencionados en la *Tabla 1*), configurando sus registros de control a través de sus respectivas direcciones de memoria. A la hora de activar o desactivar los eventos que se necesiten conocer, hay que escribir una serie de bits del registro de control. Tanto los bits del registro de control como las direcciones se pueden observar en las *Figuras 6* y *7*.

31 – 28	27 – 23	22	21	20	19	18	17	16	15 – 14	13	12	11 – 4	3 – 0
NCPU	NCNT	MC	IA	DS	EE	AE	EL	CD	SU	CL	EN	EVENT ID	CPU/AHBM

Figura 6. Bits del registro de control. Captura del documento GRLIB IP Core User's Manual

APB address offset	Register
0x00	Counter 0 value register
0x04	Counter 1 value register
4 * *n*	Counter *n* value register
0x80	Counter 0 control register
0x84	Counter 1 control register
0x80 + (4 * *n*)	Counter *n* control register
0x100	Counter 0 max/latch register
0x104	Counter 1 max/latch register
0x100 + (4 * *n*)	Counter *n* max/latch register
0x180	Timestamp register

Figura 7. Direcciones del registro de control. Captura del documento GRLIB IP Core User's Manual

Para realizar la medida del número de ciclos en la placa Virtex-5 ML506, y conociendo tanto las direcciones como los bits, se ha realizado un código para utilizar este periférico. Este código, explicado parte a parte, se encuentra a continuación.

En primer lugar, se definen las direcciones de los registros del periférico.

```
#include <stdio.h>

#define L3STAT_ADDR 0x80100100
#define L3STAT_CFG_ADDR  (L3STAT_ADDR+0x80)


#define CLEAR_L3STAT_REG(reg) \
    l3stat_addr[reg] = 0x2000; \
    val = l3stat_value[reg];


static int *l3stat_addr = (int *)L3STAT_CFG_ADDR;
static int *l3stat_value = (int *)L3STAT_ADDR;
```

Figura 8. Código l3stat

En el código de la *Figura 8* se definen la dirección base en la que se encuentra el periférico, dato proporcionado por la herramienta grmon al utilizar la opción 'infosys' como se vio en la sección 4.2.1. Después se obtiene la dirección del registro de control, que tal y como se puede observar en la *Figura 7* es la dirección 0x80 dentro del periférico.

También, se define una función 'clear', donde se activa el bit de 'clear' (bit 13) que borra el contador cuando ha sido leído, para poner los contadores a cero al principio de cada medida.

```
void init_l3stat(){
    int reg = 0;
    int val;
    CLEAR_L3STAT_REG(reg);reg++;
    CLEAR_L3STAT_REG(reg);reg++;
    CLEAR_L3STAT_REG(reg);reg++;
    CLEAR_L3STAT_REG(reg);reg++;
    CLEAR_L3STAT_REG(reg);
    l3stat_addr[reg--] = 0x1080; //dcache miss
    l3stat_addr[reg--] = 0x1380; //dcache hit
    l3stat_addr[reg--] = 0x1000; //icache miss
    l3stat_addr[reg--] = 0x1150; //execution time
    l3stat_addr[reg] = 0x1110; //instructions
}
```

Figura 9. Código l3stat

En la *Figura 9* se muestra como borrar todos los registros que se van a utilizar. Una vez borrados, se desactiva el bit del 'clear' (bit 13) y se activa el bit del 'enable' (bit 12) para empezar a contar los eventos definidos en el 'event ID' (bit 11-4). Aquí se activan los datos que se quieren conocer, como instrucciones, tiempo de ejecución, fallos de caché de datos e instrucciones.

```
void stop_l3stat(){
    int reg = 0;
    l3stat_addr[reg++] = 0x0110; //instructions
    l3stat_addr[reg++] = 0x0150; //execution time
    l3stat_addr[reg++] = 0x0000; //icache miss
    l3stat_addr[reg++] = 0x0380; //dcache hit
    l3stat_addr[reg++] = 0x0080; //dcache miss
}
```

Figura 10. Código l3stat

En la *Figura 10* se muestra la función que para los contadores de los eventos que se están leyendo. Para eso hay que desactivar el bit 'enable' (bit 12).

```
void report_l3stat(){
    stop_l3stat();
    int reg = 0, val1, val2, val3, val4;
    float res1;
    char str[100];
    val1 = l3stat_value[reg++];
    sprintf(str,"Instructions: %d\n",val1); puts(str);
    sprintf(str,"Execution time: %d\n",l3stat_value[reg++]);puts(str);
    val2 = l3stat_value[reg++];
    sprintf(str,"ICache Miss: %d\n",val2);puts(str);
    sprintf(str,"Percentage ICache: %f\n",((float)val1-val2)/val1*100);puts(str);
    val3 = l3stat_value[reg++];
    sprintf(str,"Load&Store: %d\n",val3);puts(str);
    val4 = l3stat_value[reg++];
    sprintf(str,"DCache Miss: %d\n",val4);puts(str);
    sprintf(str,"Percentage DCache: %f\n",((float)val3-val4)/val3*100);puts(str);
    sprintf(str,"Percentage Caches: %f\n",((float)val1-val2+val3-val4)/(val1+val3)*100);puts(str);
}
```

Figura 11. Código l3stat

En la función de la *Figura 11*, se leen los valores contabilizados por los registros de los eventos y se imprimen.

```
int main(int argc, char ** argv){
    int ret;
    init_l3stat();
        ret=uc_main();
    report_l3stat();
    return ret;
}
```

Figura 12. Código l3stat

Por último, en la *Figura 12*, se tiene un pequeño código en el que se inician los contadores antes de llamar a la función en la que se quieren calcular esos datos, y una vez que se ha terminado la función se detienen los contadores y se muestra por pantalla los valores obtenidos.

A la hora de realizar la medida de ciclos por instrucción, se realiza un pequeño código en el que solo se realice esa instrucción muchas veces. Al ejecutar este código en placa tan solo una vez, el número que se obtenga no será fiable para sacar la cantidad de los ciclos debido a que en ese número están incluidos los fallos de caché.

Como solución para sacar dichos ciclos el proceso usado es ejecutar el código una vez, ejecutar el código 1001 veces y obtener la diferencia entre el número de instrucciones y de ciclos en los dos casos, pudiendo así despreciar los fallos de caché.

Para aplicar este procedimiento y obtener todos los datos para Vippe, se necesita saber que instrucciones ensamblador tiene el conjunto de instrucciones del procesador LEON-3. Al observar la cantidad de instrucciones que se tienen, se han definido diferentes grupos en los que las instrucciones tienen unas características similares. Por ejemplo, los saltos de mayor o igual o de menor o igual tardarán lo mismo en ejecutarse puesto que son instrucciones de salto, al igual que la suma y la resta, que son operaciones aritméticas.

En las tablas que vienen a continuación se pueden observar las instrucciones que forman cada grupo, con algún ejemplo de los ciclos que tardan alguna de esas instrucciones de ese grupo.

Instrucciones	Definición
Add	Acumulador
Addx	Acumulador con acarreo
Addcc	Acumulador y modificar 'icc'
Sub	Resta aritmética
Subx	Resta con acarreo
Subcc	Resta y modificar 'icc'
Or	Operación 'o' lógica
Orcc	Operación 'o' lógica y modificar 'icc'
And	Operación 'y' lógica
Andcc	Operación 'y' lógica y modificar 'icc'
Xor	Operación 'o exclusiva' lógica
Sll	Movimiento a la izquierda lógico
Srl	Movimiento a la derecha lógico
Sra	Movimiento a la derecha aritmético
Mov	Mover un dato
Cmp	Comparar

Tabla 14. Grupo 1: Instrucciones aritmético-lógicas

Como ejemplo de este grupo se tienen las operaciones suma y 'o' lógica.

Instrucción	Instr1	Ciclos1	Instr2	Ciclos2	InstrT	CiclosT	Ciclo/Ins
Add	88	328	77025	83307	76937	82979	1.08
Or	88	328	77025	83310	76937	82982	1.08

Tabla 15. Ejemplos de medida de ciclos en instrucciones aritmético-lógicas

Tal y como se muestra en la *Tabla 15*, las instrucciones aritmético lógicas tardan un ciclo en ejecutarse.

Instrucciones	Definición
Jmp	Salto a etiqueta
B	Saltar
Ba	Saltar siempre
Bl	Saltar si es menor
Be	Saltar si es igual
Bg	Saltar si es mayor
Ble	Saltar si es menor o igual
Bge	Saltar si es mayor o igual

Bne	Saltar si no son iguales
Bcc	Saltar sin acarreo (mayor o igual, sin signo)
Bcs	Saltar si el acarreo está activado
Bgu	Saltar si es mayor, sin signo
Bleu	Saltar si es menor o igual, sin signo

Tabla 16. Grupo 2: Instrucciones de salto

Instrucción	Instr1	Ciclos1	Instr2	Ciclos2	InstrT	CiclosT	Ciclo/Ins
Be	112	373	101126	108435	101014	108062	1.07
Be(real)	71	373	60085	108435	60014	108062	1.80

Tabla 17. Ejemplos ciclos instrucciones de salto

En el caso de las instrucciones de salto hay que quedarse con las instrucciones que hay cuando no se cumple la condición (datos que aparecen en 'be(real)'). Esto es debido a que cuando se cumple la condición el "l3stat" cuenta el doble de instrucciones de las que en realidad hay, porque las instrucciones las trae del bus pero después el procesador las descarta y no las ejecuta, aunque ya se han contado. Por lo tanto, en este grupo las instrucciones tardan dos ciclos en ejecutarse.

Instrucciones	Definición
Fstoi	Convertir flotante de simple precisión a entero
Fmuls	Multiplicación flotante con signo
Fitos	Convertir entero a flotante simple precisión
Fdivs	Dividir flotantes con signo
Fmovs	Mover flotantes
Fdtos	Convertir flotante doble precisión a simple precisión
Fadds	Suma flotante con signo
Fbne	Saltar si no son iguales
Fcmps	Comparar simple precisión
Fsubd	Restar doble precisión

Tabla 18. Grupo 3: Instrucciones de punto flotante

Instrucción	Instr1	Ciclos1	Instr2	Ciclos2	InstrT	CiclosT	Ciclo/Ins
Fmuls	72	621	61086	344669	61014	344048	5.64
Fdivs	72	1116	61086	891166	61014	890050	14.59
Fadds	92	745	81106	445788	81014	445043	5.49
Fitos	90	859	79104	569901	79014	569042	7.20

Tabla 19. Ejemplo ciclos instrucciones de punto flotante

Como se observa en la *Tabla 19*, la instrucción 'fmuls' y 'fadds' tardan cinco ciclos, 'fdivs' tarda catorce ciclos y 'fitos' tarda siete ciclos en ejecutarse.

Instrucciones	Definición
St	Guardar palabra
Stb	Guardar byte
Sth	Guardar media palabra
Std	Guardar doble palabra
Ld	Cargar palabra
Ldsb	Cargar byte con signo
Ldub	Cargar byte sin signo
Ldsh	Cargar media palabra con signo
Lduh	Cargar media palabra sin signo
Ldd	Cargar doble palabra

Tabla 20. Grupo 4: Instrucciones de guardar y cargar

Instrucción	Instr1	Ciclos1	Instr2	Ciclos2	InstrT	CiclosT	Ciclo/Ins
St	88	614	77025	317999	76937	317385	4.12
Sth	88	611	77025	317997	76937	317386	4.12
Stb	88	614	77025	317999	76937	31738	4.12
Ld	88	344	77025	83323	76937	82979	1
Ldub	88	1126	77025	151045	76937	149919	1.95

Tabla 21. Ejemplo ciclos de instrucciones de guardar y cargar

Como se observa en la *Tabla 21*, la instrucción 'st', 'sth' y 'stb' tardan cuatro ciclos, 'ld' tarda un ciclo y 'ldub' tarda dos ciclos en ejecutarse.

Las instrucciones 'smul' (multiplicación con signo), 'sdiv' (dividir con signo) y 'udiv' (dividir sin signo), han sido analizadas cada una por separado.

Instrucción	Instr1	Ciclos1	Instr2	Ciclos2	InstrT	CiclosT	Ciclo/Ins
Smul	60	272	49074	55336	49014	55064	1.12
Smul (dep)	60	324	49074	85379	49014	85055	1.74
Sdiv	61	1344	50075	1106395	50014	1102051	22.09
Udiv	81	2115	70095	1827158	70014	1825043	26.07

Tabla 22. Ciclos de instrucciones analizadas por separado

En el caso de las instrucciones de multiplicación 'smul', se observa que la multiplicación que tiene dependencia de datos 'smul(dep)' tarda dos ciclos en ejecutarla porque tiene que parar la instrucción hasta que ese dato esté disponible en el registro. Por lo tanto, la instrucción 'smul' tarda un ciclo en ser ejecutada.

La instrucción de división 'sdiv' tarda veintidós ciclos y 'udiv' tarda veintiséis ciclos en ejecutarse.

Observando los resultados del número de ciclos que tarda cada instrucción en ser ejecutada, se ve que el número de ciclos no es un número exacto. Esto se debe a que en ese código no sólo hay las instrucciones que queremos contar, sino que hay alguna más. Conseguir un número exacto sería realizar un código muy complicado, y con este, que es más sencillo, se consiguen los ciclos con un pequeño error.

En segundo lugar, una vez que se tienen los ciclos que tarda cada instrucción en ser ejecutada, se pasan a medir el número de ciclos que lleva un fallo de la cache de datos. Para poder hacer esta medida se realizan dos códigos con varias instrucciones load, la diferencia es que uno de ellos siempre leerá el mismo dato pero el otro leerá uno cada 128. Así, sumándole 128 cada vez se consigue que provoque un miss de datos, mientras que en el otro no se provoca. La diferencia entre ambos serán los ciclos que supone una miss de caché de datos.

	Suma 0	Suma 128
Instrucciones	149	149
Ciclos	487	1241
ICache Miss	26	26
DCache Miss	1	60

Tabla 23. Datos por miss de caché

$$\text{Ciclos Miss} = \frac{\text{Ciclos}(128) - \text{Ciclos}(0)}{\text{DCacheMiss}(128) - \text{DCacheMiss}(0)} = \frac{1241 - 487}{60 - 1} = 12.78$$

El número de ciclos por fallos de caché de datos son doce ciclos, suponiendo igual el número de fallos de caché de instrucciones.

Conocidos estos datos, se puede proceder a realizar las medidas necesarias para obtener unos resultados.

4.2.4 – CUARTA ETAPA

En esta última etapa se han aplicado los valores obtenidos en la etapa anterior a Vippe, obteniendo medidas en varias aplicaciones. A continuación se explica el procedimiento seguido para obtener los resultados.

Inicialmente se tiene que modificar el archivo 'opcode_costs' de Vippe, archivo en el que se indican los ciclos que tarda cada instrucción en ejecutarse en diferentes procesadores, modificando los datos del procesador sparc por los obtenidos en la etapa anterior, puesto que es lo que utiliza LEON-3.

A continuación, se compila el código con la 'flag' de sparc-v8 porque el LEON, tal y como se dijo anteriormente, es un sparc-v8. Esta compilación utilizará el archivo 'opcode_cost' en el que están los nuevos ciclos por instrucción, tanto para la placa como para el simulador, obteniendo de esta manera un ejecutable.

En caso de querer ejecutar ese archivo en la placa, se tiene que cargar en ella mediante la herramienta grmon (estando previamente la placa programada con la herramienta IMPACT). Por otra parte, en caso de querer ejecutarlo en el simulador solo será necesario un terminal donde poder ejecutar. En ambos casos, una vez ejecutado correctamente, se obtendrá en pantalla los datos del número de instrucciones, de ciclos y fallos de caché de datos y de instrucciones.

Una vez Vippe ha estado listo, se ha comenzado realizando estos pasos para códigos pequeños, para lo que se han utilizado algoritmos comúnmente conocidos. Estos códigos son hanoi, queens y factorial, ejemplos que se encontraban en la carpeta del simulador Vippe.

Como en Vippe se han utilizado tanto la técnica de anotación en código fuente como en código intermedio es necesario hacer varias pruebas. La de código fuente funciona obteniendo información del gcc, así que hay que compararlo compilando para LEON con gcc. La otra trabaja con código intermedio de llvm, y por eso hay que compararla compilando con clang.

Por ello, usando estos ejemplos se ha comparado los datos de la placa ML506 con los que da el simulador Vippe, tanto compilándolo con gcc como con llvm, como se muestra en la sección siguiente, de resultados.

Una vez obtenidos los datos de los códigos pequeños, se ha ejecutado el código del algoritmo CCSDS. Al igual que con los ejemplos pequeños se ha ejecutado en placa y en el simulador con ambos compiladores.

Finalmente en todos los códigos ejecutados se ha obtenido el error de estimación, utilizando como referencia los datos obtenidos con la placa ML506.

5 – RESULTADOS

Tal y cómo se ha dicho en la etapa anterior, se analizan primero los tres códigos pequeños.

- Hanoi

	Placa(gcc)	**Vippe(gcc)**	**|Error gcc (%)|**
Instrucciones	9928872	9568351	3.63
Ciclos	13687852	14287116	4.39
ICache miss	88	-	
DCache miss	26	-	

Tabla 24. Resultados Hanoi con gcc

	Placa(llvm)	**Vippe(llvm)**	**|Error llvm (%)|**
Instrucciones	7520501	7340144	2.40
Ciclos	10254106	11272483	9.94
ICache miss	31	11	
DCache miss	82	15	

Tabla 25. Resultados Hanoi con llvm

Como se ve, el error en número de ciclos es pequeño utilizando anotación en código fuente, y un poco mayor en código intermedio.

- Queens

	Placa(gcc)	Vippe (gcc)	|Error gcc (%)|
Instrucciones	18594804	18538018	0.31
Ciclos	22830989	25273945	10.70
ICache miss	10	-	
DCache miss	21	-	

Tabla 26. Resultados Queens con gcc

	Placa(llvm)	Vippe (llvm)	|Error llvm (%)|
Instrucciones	20642977	21499146	4.15
Ciclos	27358118	29940351	9.44
ICache miss	25	12	
DCache miss	9	17	

Tabla 27. Resultados Queens con llvm

En este caso los resultados muestran errores similares en ambos casos.

- Factorial

	Placa(gcc)	Vippe (gcc)	|Error gcc (%)|
Instrucciones	501027	875762	74.79
Ciclos	503222	1001017	98.95
ICache miss	13	-	
DCache miss	1	-	

Tabla 28. Resultados Factorial con gcc

	Placa(llvm)	Vippe (llvm)	|Error llvm (%)|
Instrucciones	625280	625260	0.003
Ciclos	627460	750515	19.64
ICache miss	13	2	
DCache miss	1	0	

Tabla 29. Resultados Factorial con llvm

En este caso el error con gcc es muy grande. A la vista del gran error en la *Tabla 28*, se procede a comparar el código ensamblador al compilar con gcc y el código fuente del ejemplo factorial.

```c
#include "stdio.h"

/**
 * \file factorial.cpp
 * \brief Recursive factorial algorithm
 */

int factorial(int n) {
        int var;
        if (n == 1 || n == 0) {
                return 1;
        }
        var = factorial(n-1);
        return (n * var);
}

int uc_main(void) {
        int i;
        int f=0;
        for (i=0 ; i<500 ; i++) {
                f += factorial(i);
        }
        return f;
}
```

Figura 13. Código fuente del Factorial

Tal y como se puede observar en la *Figura 13*, este ejemplo pequeño es una función recursiva

```
factorial:
        save    %sp, -96, %sp
        mov     1, %o0
        cmp     %i0, 1
        bleu    .LL3
         mov    %i0, %l0
.LL4:
        call    .umul, 0
         mov    %l0, %o1
        add     %l0, -1, %l0
        cmp     %l0, 1
        bgu     .LL4
         nop
.LL3:
        jmp     %i7+8
         restore %g0, %o0, %o0
        .size   factorial, .-factorial
        .section        .rodata.str1.8,"aMS",@progbits,1
        .align 8
```

Figura 14. Código ensamblador del Factorial

Sin embargo, en la *Figura 14* no se encuentra una función recursiva, si no que hace un lazo. Para que el simulador ofrezca unas medidas precisas se tiene que mantener la misma estructura que el código fuente, porque si no, el simulador no sabe como anotar el tiempo que tarda cada bloque básico del código fuente, por lo que los datos son totalmente erróneos. Esto muestra una fuente de error la técnica de anotación en código fuente.

Una vez analizados los ejemplos pequeños, se pasa a compilar y ejecutar el código CCSDS, un algoritmo para la comprensión de imágenes en escala de grises. Este algoritmo se pretende que sea adecuado para su uso a bordo de una nave espacial, por lo que la complejidad del algoritmo está diseñada para ser lo suficientemente baja para hacer la implementación de hardware de alta velocidad factible [7].

	Placa(gcc)	**Vippe (gcc)**	**|Error gcc (%)|**
Instrucciones	72171184	77334905	7.15
Ciclos	132567682	108369543	11.41
ICache miss	4154	-	
DCache miss	751370	-	

Tabla 30. Resultados CCSDS con gcc

	Placa(llvm)	**Vippe (llvm)**	**|Error llvm (%)|**
Instrucciones	88358593	74574141	15.60
Ciclos	156880140	111349363	23.14
ICache miss	13098	352957	
DCache miss	759114	898914	

Tabla 31. Resultados CCSDS con llvm

En primer lugar, como se puede observar en las tablas de resultados de los códigos probados, los fallos de caché de instrucciones y de datos no dan igual en placa que en Vippe compilado con llvm. Por otra parte, no se tienen datos de fallos de caché de instrucciones y de datos de Vippe compilado con gcc ya que el contador está desactivado, porque en la versión gcc con la que se ha trabajado daba un error de compilación con dicho contador activado.

Como solución a esto para calcular los errores entre la placa y el simulador, se han sumado a los ciclos que da el simulador los fallos de caché de la placa multiplicados por los ciclos que tarda cada miss.

Analizando la *Tabla 31*, utilizando vippe compilando con llvm se tiene un error en instrucciones del 15.60%, por lo tanto el número de ciclos también tiene un error elevado. En el momento que el número de instrucciones sea más preciso, el error en los ciclos disminuirá.

Por otra parte, analizando las *Tablas 30* y *31*, se puede concluir que Vippe con el compilador gcc es más preciso que utilizando el compilador llvm. Por eso, para obtener una medida lo más precisa posible, se ha seguido trabajando con el compilador gcc debido a que el número de instrucciones que estima Vippe ya tiene mucho error, y conseguir el objetivo de error (sobre un 10 %) en la estimación de los ciclos no es posible. Así, se ha realizado un análisis de las casos de error de la estimación en código fuente, para su resolución como trabajo futuro.

El primer error detectado es debido a que hay varias funciones que están en el código compilado sin estar en el código fuente. Por este motivo, el simulador Vippe no puede medir ni las instrucciones ni los ciclos de dichas funciones. Para corregir esto, en líneas futuras se deben de introducir esas funciones para que el simulador pueda medirlas y mejore la estimación en los datos, disminuyendo el porcentaje de error.

Para conocer cuáles son esas funciones que no puede estimar el simulador se ha utilizado la opción 'pro1' del simulador Tsim. Esta opción nos proporciona los datos de qué función tarda más tiempo en ejecutarse, en tanto por ciento. Las funciones de esa lista que estén iniciadas por '_' implican que el compilador no las puede estimar. Estas funciones con su respectivo porcentaje se encuentran en la *Tabla 32*.

Funciones	**Ratio (%)**
_calloc_r	5.87
_malloc_r	3.50
_free_r	2.21
__st_pthread_mutex_unlock	0.77
__st_pthread_mutex_lock	0.41
__malloc_lock	0.29
__malloc_unlock	0.26
__st_pthread_mutex_trylock	0.03
__st_pthread_mutex_deslock	0.03
_malloc_trim_r	0.01

Tabla 32. Funciones que el compilador no puede estimar

La suma de estas funciones es de un 13%, por lo que al sumarlas se reduciría mucho el error actual, que es del 11.41%.

El siguiente paso para localizar las fuentes de error, es analizar el código función a función, realizando en primer lugar el análisis de la función que más tiempo tarde en ejecutarse.

Para obtener qué función tarda más en ejecutarse, de la misma manera que en el caso anterior, se utiliza la opción 'pro1' del simulador Tsim. Utilizando esa opción, la función que tiene mayor porcentaje es 'ACBpeEncoding' (50.96%), dentro de ella la función que más ocupa es 'BlockScanEncode' (26.17%).

Una vez que se obtiene qué función es la que más tiempo necesita, se calcula el error que tienen los valores obtenidos con Vippe con los de la placa. Este error se puede ver en la *Tabla 33*.

	Placa(gcc)	**Vippe (gcc)**	**|Error gcc (%)|**
Instrucciones	18942886	23095545	21.92
Ciclos	25889846	25651438	6.70
ICache miss	163314	-	
DCache miss	917	-	

Tabla 33. Resultados función 'BlockScanEncode'

Como se observa en la *Tabla 33* el simulador cuenta instrucciones de más, lo que conlleva a contar ciclos de más (ciclos + fallos de caché). Por eso, se analiza el código poco a poco para encontrar las partes en las que se cuentan instrucciones de más.

En primer lugar, se realiza un cambio en los 'if' del código que están definidos de la siguiente manera:

```
temp_x = (i >= 1 ? 1 : 0);
```

Figura 15. Formato de if en la función

Sustituyendo cada uno por el 'if' desarrollado correspondiente, en el caso de la *Figura 15* es el siguiente:

```
if (i>=1){
        temp_x = 1;
}
else {
        temp_x = 0;
}
```

Figura 16. Código if desarrollado

Una vez desarrollado ese tipo de 'if', se realiza un código con solo esos 'if' para ver que hace exactamente el compilador, analizándolo en el código ensamblador. A continuación, en las *Figuras 17* y *18* se puede observar tanto el código fuente como el código ensamblador.

```
int uc_main(){
        register int temp_x = 0, temp =0;
        register int temp_y, k;
        int block[4][2]={{0, 1}, {2, 3}, {3, 2}, {1, 0}};

        int Bit_Set_Plane;

        for (k = 0; k < 20; k++) {
                Bit_Set_Plane = (1 << (k - 1)) ;
                if (k >= 1) {
                        temp_x = 1;
                }

                if (k != 1) {
                        temp_y = 1 + temp;
                }
                else {
                        temp_y =  temp;
                }
```

Figura 17. Código fuente

```
int uc_main(){
   0:   9d e3 bf 80     save  %sp, -128, %sp
        register int temp_x = 0, temp =0;
        register int temp_y, k;
        int block[4][2]={{0, 1}, {2, 3}, {3, 2}, {1, 0}};
   4:   13 00 00 00     sethi  %hi(0), %o1
   8:   90 07 bf e0     add  %fp, -32, %o0
   c:   92 12 60 00     mov  %o1, %o1
  10:   40 00 00 00     call  10 <uc_main+0x10>
  14:   94 10 20 20     mov  0x20, %o2
  18:   b0 10 20 00     clr  %i0
                }
                else {
                        temp_y =  temp;
                }
```

Figura 18. Código ensamblador

Analizando el código ensamblador de la *Figura 18*, se puede observar que el 'else' lo cuenta como instrucción aunque no haga operaciones dentro de él, es decir, no entra en el 'else'. Este es uno de los motivos por los que el simulador cuenta instrucciones de más.

Otro de los posibles errores, al igual que el anterior, tiene relación con los 'if'. Los 'if' de la función formados por varias operaciones 'and' (&&) y 'or' (||) comprueban la condición que se encuentra antes de la primera operación, y si es falso no continúa. Sin embargo, si esa primera condición es verdadera se comprueba la segunda condición de ese 'if', lo que hace que se introduzcan instrucciones y ciclos de más.

Para verificar que esto es verdad, se realiza un código 'if' cuya condición esté formada por varias operaciones 'and' y 'or'. Una vez realizado el código se saca el código ensamblador para verificar que el compilador hace varios saltos en lugar de uno.

```
#include <stdlib.h>
#include <math.h>

int funcionP(int i, int var1 , int var2 ){
        int k, var3 = 0;

        for (k=0; k=3; k++){
                if( ((i == 0) && (var1 < var2 )) || ((i == 1) && (var2 < var1 )) || ((i == 2) && (var3 < var1 )))
                {
                        var1 = var2 - 1;
                        var2 = var1 + 1;
                        var3 = var2 + 2;
                }
                return var3;
        }
}
```

Figura 19. Código de if de 'and' y 'or'

```
funcionP:
        cmp     %o1, %o2
        bge     .LL14
         cmp    %o2, %o1
        cmp     %o0, 0
        bne     .LL8
         cmp    %o2, %o1
.LL2:
        jmp     %o7+8
         add    %o2, 2, %o0
.LL8:
.LL14:
        bl      .LL12
         cmp    %o0, 1
        cmp     %o1, 0
        ble     .LL10
         cmp    %o0, 2
.LL15:
        be      .LL2
         nop
.LL10:
        jmp     %o7+8
         mov    0, %o0
.LL12:
        be      .LL2
         cmp    %o1, 0
        bg      .LL15
         cmp    %o0, 2
        b,a     .LL10
        .size   funcionP, .-funcionP
```

Figura 20. Código ensamblador del código if anterior

Como se observa en la *Figura 20,* se verifica que al realizar este tipo de 'if' el compilar realiza varios saltos. Siendo esta otra de las razones por la que se introducen instrucciones y ciclos de más.

Para solucionar un error en un futuro, estos 'if' podrían ser cambiados automáticamente por unos 'if' encadenados.

6 – CONCLUSIONES

En este proyecto se han desarrollado una serie de etapas para obtener un simulador que utilizando la técnica de simulación nativa sea capaz de estimar el número de ciclos que necesita una plataforma basada en el procesador LEON-3 para ejecutar un código, alcanzando el objetivo propuesto desde un primer momento. Para ello se ha extendido el simulador Vippe.

La primera etapa realizada es la familiarización con el entorno y las herramientas, así como disponer tanto el software como el hardware para poder realizar las primeras medidas sobre la plataforma. En la segunda etapa se realiza la evaluación de distintas configuraciones para analizar la diferencia entre unas y otras. La tercera etapa, es la que se centra en modelar correctamente la plataforma basada en el procesador LEON-3 para introducir los detalles en el simulador Vippe. Por último se ha comparado los resultados de Vippe con la plataforma real.

Durante la realización de la primera parte de este trabajo, se ha llegado a una serie de conclusiones, entre las que destacan que:

- La diferencia entra la arquitectura sparc v7 y sparc v8 es mínima.
- La diferencia entre utilizar la versión 4.4.2 o 3.4.4 del compilador gcc es mínima cuando se utiliza la FPU, pero muy grande si no se utiliza.

- El simulador Tsim supone que la FPU es más rápida de lo que es realmente, y que la unidad de enteros es más lenta que en la realidad.

Tras la realización del trabajo se ha alcanzado el objetivo de obtener un simulador que tenga precisión en el número de ciclos, obteniéndose un error de casi un 10 % con el ejemplo grande: un codificador CCSDS. Además, se han identificado fuentes de errores, de forma que introduciendo las modificaciones oportunas en Vippe este error podría ser mucho menor.

Por otra parte, se ha visto que actualmente, Vippe con el compilador de gcc (anotación en código fuente) es más preciso que con el compilador llvm (anotación en código intermedio).

REFERENCIAS

Las fuentes que se han consultado para elaborar el proyecto son las siguientes:

[1] "VIPPE: Native simulation and performance analysis framework for multi-processing embedded systems"
Díaz L., González E., Villar E., Sánchez P. (2014).

[2] Tesis "Estimación prestaciones para Exploración de Diseño en Sistemas Embebidos complejos HW/SW"
Héctor Posadas Cobo

[3] "LEON3/GRLIB. SOC IP Library"
Gaisler

[4] "GRLIB IB Core User's Manual. Versión 1.4.1 – b4156, May 2015"
Gaisler

[5] "ML505/ML506/ML507 Evaluation Platform. User Guide"
Xilinx

[6] "TSIM. LEON/ERC32 System Simulator"
Gaisler

[7] "Report Concerning Space Data System Standars. Image Data Compression"
CCSDS

ANEXO 1

A continuación se muestran las tablas realizadas para un tamaño de 1000x1000.

	V7 FPU	V7 NO FPU	V8 FPU	V8 NO FPU
DWT	8.9668	109.18	8.568	108.6
Total (DWT+BPE)	17.732	117.93	17.132	117.08

Tabla 1. Compilador gcc 1000x1000. –O2. Placa

Los datos de esta tabla son los que han sido utilizados a la hora de realizar las comparaciones con las demás medidas como referencia, calculando con ellos el error.

	V7 FPU (ref)	V7 FPU	Diferencia
DWT	8.9668	9.5331	6.32%
Total (DWT+BPE)	17.732	19.542	10.21%

Tabla 2. Compilador llvm 1000x1000. Placa

	V7 FPU		V7 NO FPU		V8 FPU		V8 NO FPU	
		Difer.		Difer.		Difer.		Difer.
DWT	8.9062	-0.68%	109.09	-0.08%	8.5073	-0.71%	108.59	-0.01%
Total	16.8793	-4.81%	117.06	-0.74%	16.3733	-4.43%	116.46	-0.53%

Tabla 3. Compilador gcc 4.4.2 -O3. 1000x1000. Placa

	V7 FPU		V7 NO FPU		V8 FPU		V8 NO FPU	
		Difer.		Difer.		Difer.		Difer.
DWT	9.02	0.59%	213.16	95.24%	8.4733	-1.11%	212.44	95.62%
Total	18.6678	5.28%	222.83	88.95%	18.1233	5.79%	222.10	89.70%

Tabla 4. Compilador gcc 3.4.4. 1000x1000. Placa

	V7 FPU		V7 NO FPU		V8 FPU		V8 NO FPU	
Gcc		Difer.		Difer.		Difer.		Difer.
DWT	11.425	27.41%	121.24	11.05%	10.954	27.85%	86.036	-20.78%
Total	21.826	23.09%	131.64	11.63%	21.333	24.52%	96.427	-17.64%
Llvm								
DWT	11.622	29.61%						
Total	23.561	32.87%						

Tabla 5. Compilación con SO (Arrays). 1000x1000. Placa

	V7 FPU		V7 NO FPU		V8 FPU		V8 NO FPU	
Gcc		Difer.		Difer.		Difer.		Difer.
DWT	11.424	27.40%	121.24	11.05%	10.954	27.85%	86.063	-20.75%
Total	22.171	25.03%	132.01	11.94%	21.709	26.72%	96.827	-17.30%
Llvm								
DWT	11.618	29.57%						
Total	24.262	36.83%						

Tabla 6. Compilación con SO (Archivos). 1000x1000. Placa

	V7 FPU		V7 NO FPU		V8 FPU		V8 NO FPU	
		Difer.		Difer.		Difer.		Difer.
DWT	11.5	28.25%	121.33	11.13%	10.995	28.33%	86.127	-20.69%
Total	21.908	23.55%	131.43	11.45%	20.835	21.61%	95.961	-18.04%

Tabla 7. Un core. SO (Arrays) OpenMP. 1000x1000. Placa

	V7 FPU		V7 NO FPU		V8 FPU		V8 NO FPU	
		Difer.		Difer.		Difer.		Difer.
DWT	11.5	28.25%	121.38	11.17%	10.994	28.31%	86.166	-20.66%
Total	22.268	25.58%	131.85	11.80%	21.204	23.77%	96.455	-17.62%

Tabla 8. Un core. SO (Archivos) OpenMP. 1000x1000. Placa

	V7 FPU			V7 NO FPU		
		Difer.(gcc)	Difer.(gcc+oMP)		Difer.(gcc)	Difer.(gcc+oMP)
DWT	8.1052	-9.61%	-29.52%	64.2629	-41.14%	-47.03%
Total	19.0127	7.22%	-13.22%	75.1977	-36.24%	-42.78%
	V8 FPU			V8 NO FPU		
		Difer.(gcc)	Error(gcc+oMP)		Difer.(gcc)	Difer.(gcc+oMP)
DWT	7.5907	-11.41%	-30.96%	46.7884	-56.92%	-45.68%
Total	18.4731	7.83%	-11.34%	57.644	-50.77%	-39.93%

Tabla 9. Dos cores. SO (Arrays) OpenMP. 1000x1000. Placa

	V7 FPU			V7 NO FPU		
		Difer.(gcc)	Difer.(gcc+oMP)		Difer.(gcc)	Difer.(gcc+oMP)
DWT	8.1369	-9.26%	-29.24%	64.3042	-41.10%	-47.02%
Total	19.4001	9.41%	-12.88%	75.6022	-35.89%	-42.66%
	V8 FPU			V8 NO FPU		
		Difer.(gcc)	Difer.(gcc+oMP)		Difer.(gcc)	Difer.(gcc+oMP)
DWT	7.5968	-11.34%	-30.90%	46.8157	-56.89%	-45.67%
Total	18.7771	9.60%	-11.45%	58.031	-50.43%	-39.84%

Tabla 10. Dos cores. SO (Archivos) OpenMP. 1000x1000. Placa