Proyecto Fin de Grado
Grado en Ingeniería Aeroespacial
Intensificación de Navegación Aérea

Desarrollo de un flujo de preparación de diseño para inyección de fallos sobre el microcontrolador openMSP430

Autor: José María Bascón Mallado
Tutor: Hipólito Guzmán Miranda

Dep. de Ingeniería electrónica
Escuela Técnica Superior de Ingeniería
Universidad de Sevilla

Sevilla, 2017

Proyecto Fin de Grado
Grado en Ingeniería Aeroespacial
Intensificación de Navegación Aérea

Desarrollo de un flujo de preparación de diseño para inyección de fallos sobre el microcontrolador openMSP430

Autor:
José María Bascón Mallado

Tutor:
Hipólito Guzmán Miranda
Profesor Contratado Doctor

Dep. de Ingeniería Electrónica
Escuela Técnica Superior de Ingeniería
Universidad de Sevilla
Sevilla, 2017

Proyecto Fin de Grado: Desarrollo de un flujo de preparación de diseño para inyección de fallos sobre el microcontrolador openMSP430

Autor: José María Bascón Mallado

Tutor: Hipólito Guzmán Miranda

El tribunal nombrado para juzgar el Proyecto arriba indicado, compuesto por los siguientes miembros:

Presidente:

Vocales:

Secretario:

Acuerdan otorgarle la calificación de:

Sevilla, 2017

El Secretario del Tribunal

A mis padres y hermana

A mi familia y amigos

A todos los que han contribuido a mi formación.

Agradecimientos

Son muchas las personas a las que debo de dedicar este esfuerzo, en el ámbito académico y el ámbito personal.

En primer lugar, debo destacar el papel de dos personas fundamentales en este proyecto, por su esfuerzo, trabajo, tiempo, paciencia y dedicación. Quiero dar las gracias al catedrático Miguel Ángel Aguirre por brindarme la oportunidad de aprender sobre una materia totalmente desconocida para mí, y en mi opinión de enorme importancia en el mundo aeroespacial, y por sus aportes durante todo el trabajo.

Este trabajo me ha permitido conocer a una persona y un profesional brillante, el doctor Hipólito Guzman Miranda, el tutor de este TFG. Debo darle mil gracias por su infinita paciencia conmigo. A pesar de estar desbordado en el ámbito laboral, no ha dudado en sacar cada semana un rato para atender mis problemas y transmitirme el conocimiento que he ido necesitando adquirir, así como contestar a cada uno de los emails que le he enviado. Gracias Hipólito, creo que has sido uno de los mejores profesores que he conocido durante esta etapa, gracias.

También estoy orgulloso de haber conocido y poder destacar el papel del investigador y estudiante de doctorado Javier Barrientos. Gracias Javi por ayudarme en lo que he necesitado de forma totalmente voluntaria, agradezco mucho el tiempo que me has dedicado y la información que me has propocionado.

Me gustaría nombrar a otra persona muy influyente en mi formación académica y personal, Teresa Moliz, mi profesora de electrotecnia durante el segundo curso de bachillerato. Gracias por transmitirme, además de conocimiento, las ganas por saber y la humildad de la que haces gala siempre. Secundaria y bachillerato necesita muchísima gente como tú.

Como no, debo destacar el papel durante todos estos años de mi madre Mercedes y mi hermana Marta. Gracias por confiar en mí desde el primer momento, y a pesar de los altibajos vividos durante esta etapa, siempre darme ese empujón que he necesitado para continuar hacia adelante y no rendirme.

A todos los que de una forma u otra han contribuido a mi formación, gracias a todos y cada uno de ellos.

Aquí, todos, tenéis un amigo.

José María Bascón Mallado

Julio de 2017

Resumen

En el proyecto desarrollado se ha diseñado un flujo de trabajo para la implementación del microcontrolador sintetizable openMSP430 de código abierto en el sistema de inyección de fallos FT-UNSHADES 2. Con este método se podrán implementar de forma automatizada cualquier microcontrolador de la familia que incluye el openMSP430 sin necesidad de llevar a cabo todo el proceso de síntesis cada vez que se desee implementar un programa en un mismo diseño, ahorrando tiempo y recursos.

El interés de uso de este microcontrolador concreto, y no otro, se debe a varias características, entre las que destacan su bajo consumo, lo cual es notablemente interesante para su aplicación en el ámbito espacial, así como su gran parecido a la familia de microncontroladores comerciales MSP430. Su uso en el ámbito espacial y sus buenas resultados, suponen un atractivo muy interesante para su implementación. Además, las características de configuración, prestaciones y respuesta ante ambientes radiactivos de las FPGAs, motivan a la implementación de circuitos de distinta índole en estos dispositivos.

Expuesto el flujo de trabajo desarrollado y explicadas las herramientas usadas durante el proceso, se expondráN los resultados obtenidos para un modelo concreto del openMSP430. En cuestión, se ha implementado un modelo que dispone de 48kB de memoria de programa, o memoria ROM, y 10kB de memoria de datos, también conocida como memoria RAM.

Abstract

The objective of this Project is to develop a sistematical way to implement any version of openMSP430 in FTU2.

If we provide a system which can emulate ionizing environments we are able to test our design and detect where is critical parts are located. This offers an huge potential because we can test each parts of our design before past the real test and strengthen weak parts. This means saving money and resources, because we only have to apply protection to the parts which are critical, and no to the complete design like it is frequently done, o we can design hardenes software and we not apply protection to the critical parts; this can be done, for example, using fault detection techniques.

Other interesting question in this Project is the selection of this microcontroller and the implementation in a Virtex 5FPGA. The commercial MSP430 models have a very interesting characteristic for space: its ultra low power. One of the main problems at space is power, and you can´t waste it. The other problem is radiation. Electronic devices are exposed to solar winds and cosmic radiation which affect the operation of those devices. FPGAs are a good response before radiation. The implementation allows to predict seu error rates in the commercial and openMPSP430 versions.

Índice de Ilustraciones

Ilustración 1. FPGA Virtex 5 [16] 5
Ilustración 2. Satélite Ceres. Planetary Resources [2] 6
Ilustración 3. Estructura del openMSP430 [4] 7
Ilustración 4. MSP430F1611 [5] 8
Ilustración 5. Arquitectura openMSP430 [4] 9
Ilustración 6. Relación primitiva-tipo de memoria. Fuente [9] 25
Ilustración 7. Contenido del Block RAM 0 para el caso de 32kB 30
Ilustración 8. Contenido del Block RAM 4 para el caso de 32kB 30
Ilustración 9. Contenido del Block RAM 6 para el caso de 32 kB 31
Ilustración 10. Contenido del Block RAM 2 para el caso de 32 kB 31
Ilustración 11. Contenido del Block RAM 7 para el caso de 32 kB 31
Ilustración 12. Contenido del Block RAM 3 para el caso de 32 kB 31
Ilustración 13. Ventana de diálogo para cargar archivo coe 38
Ilustración 14. Cambio de las opciones principales en FPGA Editor 39
Ilustración 15. Bloque de memoria físico 40
Ilustración 16. Contenido memoria según archivo coe 40
Ilustración 17. Contenido Block RAM 0 cargado mediante data2mem 41
Ilustración 18. Contenido Block RAM 6 cargado mediante data2mem 41
Ilustración 19. Inicio de programa para el openMSP430 48k10k 44
Ilustración 20. Direcciones de memoria según interrupción. Fuente [4] 45
Ilustración 21. Salida del primer número del programa tfg_ports 45
Ilustración 22. Evolución de las salidas según el programa tfg_ports 46
Ilustración 23. Etapas iniciales funcionando en FTU 46
Ilustración 24. Primera salida observada en p1_pout_ext en FTU2 46
Ilustración 25. Salidas de las señales p1_pout_ext y p2_pout_ext 47
Ilustración 26. Etapas iniciales en FTU con contenido cargado mediante data2mem 47

Ilustración 27. Primera salida de p1_pout_ext y p2_pout_ext en FTU con contenido cargado mediante data2mem 47
Ilustración 28. Salidas p1_pout_ext y p2_pout_ext en FTU con contenido cargado mediante data2mem 48
Ilustración 29. Primera salida para el puerto p1_pout_ext para el programa tfg_sqr 49
Ilustración 30. Primera salida para el puerto p2_pout_ext para el programa tfg_sqr 49
Ilustración 31. Salidas p1_pout_ext y p2_pout_ext para el programa tfg_sqr 49
Ilustración 32. Salidas p1_pout_ext y p2_pout_ext para instante aleatorio 50

Índice

Agradecimientos **ix**

Resumen **xi**

Abstract **xiii**

Índice de Ilustraciones **xv**

Índice **1**

1 Introducción **3**

2 openMSP430 **9**

2.1 Arquitectura del openMSP430 *9*

2.2 El archivo openMSP430_defines *10*

2.3 Generación de memorias *12*

2.4 Registros de interés del openMSP430 *13*

3 FT-UNSHADES **15**

3.1 Introducción a FT-UNSHADES 2 *15*

3.2 Modos de trabajo *16*

3.2.1 Modo Debug 16

3.2.2 Modo Campaña 16

3.3 Diagrama de flujo para uso de FTU *17*

4 Preparación del diseño y flujo de trabajo **19**

4.1 Introducción a la problemática presentada por el diseño y justificación del modelo implementado *19*

4.2 Generación de archivos para implementación *20*

4.3 Tratamiento de las Blocks RAM para su uso en el flujo de trabajo *22*

4.4 Generación del archivo bmm *24*

4.5 Instancia de memoria para deducción de relaciones entre Blocks RAM *27*

4.6 Generación de un archivo mem a partir de un programa descrito en lenguaje C *33*

4.7 Generación del archivo coe para el primer arranque *37*

4.8 Flujo de diseño para la implementación en FTU2 *41*

5 Resultados **43**

5.1 Resultados obtenidos para el ejemplo 1 *43*

5.2 Resultados obtenidos para el ejemplo 2 *48*

6 Conclusiones y Trabajos Futuros **51**

REFERENCIAS **53**

1 INTRODUCCIÓN

La ciencia es el alma de la prosperidad de las naciones y la fuente de vida de todo progreso.

Louis Pasteur

El interés del ser humano por conocer que hay más allá de nuestro planeta es inherente a nuestra naturaleza. La curiosidad y la necesidad de conocimiento nos ha llevado a construir satélites artificiales que nos permiten satisfacer estas inquietudes.

En cada instante hay millones de personas haciendo uso de la información enviada por los satélites. ¿Quién no usa de forma habitual las señales de los sistemas GNS? Si me encuentro en una zona aislada y necesito comunicarme a través de mi teléfono móvil puede hacerlo a través de un satélite. Conocer la previsión metereológica para los días posteriores implica estudiar el movimiento de la atmósfera desde un satélite geoestacionario y disponer de esa información para llevar a cabo la predicción. Las señales de televisión, monitorizar que ocurre en cada instante en cualquier parte del mundo, cartografiar la superficie terrestre, estudiar el movimiento de las placas tectónicas, etc, implica en algún punto de la comunicación el uso de información adquirida y/o proporcionada por un satélite.

Pero ¿sabemos realmente qué es un satélite? Pues básicamente un satélite es una caja formada por elementos que captan energía, ya que actualmente no pueden usarse reactores nucleares que la generen, lo que obliga a hacer uso de placas solares que recarguen las baterías de estes dispositivos, y elementos que consumen dicha energía para llevar a cabo las funciones para las que estén diseñados. Y ahora cabe preguntarse, ¿qué requerimientos tienen los elementos electrónicos que se montan en los satétiles? La respuesta a esta pregunta no es simple, veamos qué y por qué.

Argumentado con las consideraciones expuestas por Olav Thorheim[1] en el artículo *Electronics in space* [1], la industria espacial es una de las mayores fuerzas que hacen avanzar a la electrónica. Hay una brecha considerable entre las características de los componentes electrónicos espaciales y los componentes comerciales que usamos los consumidores habituales.

La electrónica espacial está sometida a unas duras condiciones en su entorno. Lo primero es que los componentes tienen que soportar las vibraciones cuando son lanzadas en el cohete. Lo segundo es que tienen que resistir

[1] Senior Development Engineer, Data Responses

variaciones de temperatura muy altas. Debido al vacío del espacio, no es posible que se den los fenómenos de conducción o convección térmica, por lo que el único mecanismo de transmisión de calor es la radiación. Un satélite orbitando alrededor de la Tierra experimenta variaciones de temperatura desde más de 120 ºC a menos de -150ºC.

La desgasificación es otra cuestión a tratar. Los componentes se localizan normalmente junto con instrumentos. No sería aceptable que estos componentes despidieran gases que interfirieran en las características de otros instrumentos, por ejemplo, depositando material en componentes ópticos. Los componentes espaciales calificados son fabricados usando materiales cerámicos -los materiales plásticos no se usan habibualmente.

En el espacio los dispositivos se encuentran con ambientes que no se dan de forma habitual en la Tierra. El fenómeno más destacable de estos es la radiación ionizante. Existen diferentes fuentes de radiación. Los vientos solares mueven electrones, protones e iones pesados. También hay protones e iones pesados provenientes de la radiación cósmica. Cuando estas partículas se aproximan a la Tierra son capturadas por su campo magnético, en concreto los electrones y protones son atrapados en los cinturones de Van Allen. Los iones pesados, debido a su alta energía, son deflectados por la magnetosfera, y solo una pequeña proporción son capturados por los cinturones de radiación o Van Allen. En los satélites en órbitas geosíncronas, los cinturones de radiación causan problemas significativos en la electrónica.

Hay dos formas de que la radiación produzca daños, la conocida como Efectos de un Único Evento (SEE -Single Event Effects), o a través de la llamada Dosis Ionizante Total (TID -Total Ionozing Dose). La primera, SEE, se refiere al efecto causado por una partícula. La segunda, TID, a la cantidad de radiación que los circuitos electrónicos han recibido durante su vida útil.

La razón por la que la radiación destruye un circuito electrónico debido a la Dosis Ionizante Total es la siguiente: Una partícula que llega a un transistor genera pares de huecos de electrones en el términal óxido de los transistores integrados. Los electrones generados tienen una alta movilidad y encuentran rápidamente el camino para salir del óxido. El resto del circuito irradiado pasa por el mismo proceso. Esto provoca que cambie el umbral de tensión del transistor hasta que este conduce, o no, de forma permanente.

Volviendo a las formas que la radiación daña un circuito, atendiendo a la primera forma, SEE, se distinguen tres tipos de efectos diferentes:

1. Single Events Upsets (SEU) se define por la NASA como “errores inducidos por radiación en circuitos microelectrónicos causados cuando partículas cargadas pierden energía por inonización del medio por el que pasan, dejando detrás una estela de pares de electrones y huecos”. Este fenómeno no destruye el circuito, pero corrompe la información en los resgistros y elementos de memoria. Los efectos producidos por este fenómeno pueden corregirse haciendo reset o reprogramando los elementos de memoria afectados.
2. Single Level Latchup es un fenómeno que ocurre cuando partículas cargadas penetran en el substrato de circuitos integrados[2]. Este fenómeno provoca que el circuito entre en realimentación positiva lo que hace que vaya aumentando la corriente que circula por el substrato. Si no hay mecanimos que corten esta realimentación, la corriente puede continuar aumentando hasta que el circuito se dañe térmicamente.
3. Single Event Burnout es un fenómeno en el que un ión pesado pasa a través de un transisistor depositando suficiente carga para provocar que conduzca. Si el transistor se mantiene en este estado, las altas corrientes a través del transistor pueden ser suficientes para destruir el circuito entero. Este efecto suele darse en transistores y diodos de potencia.

¿Cómo podemos solucionar estos problemas? La respuesta es el blindaje, la redundancia y en muchos casos, la robustez software. Pero ¿cómo se puede blindar un componente?, ¿lo hacemos todo redundante o una parte solo? ¿se opta por el desarrollo de software robusto? Esta es la principal motivación de este trabajo.

Se presentan tres opciones. La primera, blindar el circuito electrónico completo, o bien blindar las partes que sean más críticas del circuito integrado que hayamos diseñado. La tercera, y la opción más innovadora, es el desarrollo de software robusto, es decir, software que sea capaz de llevar a cabo la tarea programada a pesar de

[2] Estructura de pequeñas dimensiones de material semiconductor, normalmente silicio, sobre la que se fabrican circuitos electrónicos generalemente mediante fotolitrografía y que está protegida dentro de un encapsulado de plástico o cerámica.

fallos, o si detecta ciertas condiciones actúe en consecuencia de la forma que se ha programado.

El blindaje da buen resultado frente a los vientos solares, pero su efecto es pequeño contra la radiación cósmica y las partículas de alta energía. Es útil para reducir la cantidad de TID, pero contra los Efectos de Evento Único reduce su efectividad.

La Agencia Espacial Europea, ESA, exige requerimientos muy estrictos a los componentes que vayan a ser montados en un proyecto financiado por esta agencia. La producción de los componentes se hace de una forma particular, siendo estos testados de acuerdo a unos estrictos estándares que los califiquen para el espacio. Esto hace que el coste sea mucho más alto que los componentes comerciales.

Para el proceso de calificación la ESA ha definido sus propios estándares, llamado European Cooperation for Space Standarization, ECSS. ECSS tiene tres ramas: estándar de gestión de proyectos, estándares de construcción y estándares de garantía de calidad.

Los circuitos programables integrados reúnen tantas funciones como sean posible dentro del mismo chip. Estos circuitos se denominan FPGAs (Field Programmable Gate Arrays). Las FPGAs no tenían demasiado uso en el campo espacial en sus orígenes, ya que su tecnología de configuración SRAM no era lo suficientemente resistente a la radiación. El desarrollo de la FPGA que usaba tecnología antifusible para establecer las conexiones demostró su resistencia a la radiación, y aquí empieza su expansión en este ámbito. Estos circuitos tienen la desventaja de que solo pueden ser programados una vez, y por tanto la arquitectura de la FPGA queda cerrada para toda la misión

Para mantener las características ventajosas ofrecidas por las FPGAs y añadir la flexibilidad de la reprogramación tantas veces como se desee, la solución son las FPGAs con tecnología SRAM. Las FPGAs basadas en la tecnología SRAM no solo tienen la ventaja de ser reprogramables frente a la tecnología antifusible, sino que presentan mucha mayor densidad de lógica programable, lo que permite configurar circuitos digitales muchos más complejos y de mayor tamaño.

Para protegerse de los fallos causados por la radiación, la memoria de configuración de una FPGA basada en SRAM se reescribe constantemente, incluso durante la operación normal de la misma. Esta técnica se denomina scrubbing. La técnica puede usarse para llevar a cabo una reconfiguración total o parcial de la FPGA. Cuando se hace una reconfiguración parcial, los códigos CRC[3] detectan los fallos y las partes de las memorias de configuración que contienen errores se sobreescriben con los datos de configuración correctos. Estos datos están almacenados en una memoria Flash resistente a la radiación.

Xilinx se encuentra entre las empresas que producen FPGAs SRAM calificadas para misiones espaciales. En concreto la familia Virtex-4VQ y Virtex-5VQ. Estos modelos tienen la ventaja de que mientras una celda SRAM habitual está formada por seis transistores, estas usan doce interconectados entre sí. Esta construcción hace que sea mucho más difícil que una partícula cargada cambie la polaridad de una celda.

Ilustración 1. FPGA Virtex 5 [16]

Atendiendo ahora a los microcontroladores montados en los satélites, además de cumplir los requerimientos ya mencionados en el diseño que sean montados, la potencia consumida por estos dispositivos debe de ser muy

[3] La verificación por redundancia cíclica (CRC) es un código de detección de errores para detectar cambios accidentales en los datos.

pequeña, ya que la energía no es un recurso abundante durante una misión espacial. Es aquí donde aparece la familia de microprocesadores MSP430 de Texas Instruments.

El MSP430 es una familia de microcontroladores fabricados por Texas Instruments. Construido con una CPU de 16 bits, está diseñado para aplicaciones embebidas de bajo costo y muy bajo consumo de energía. Este dispositivo tiene una gran variedad de configuraciones que se agrupan en familias, con velocidades máximas de procesamiento y capacidades de direccionamiento diferentes, así como diferentes selecciones de periféricos.

La compañía *Planetary Resources* monta esta familia de microcontroladores en sus satélites. Según afirman en el artículo *How Planetary Resources uses TI MSP430 MCUs to discover Earth´s resources from space,* sus requerimientos de ultra bajo consumo lo hacen especialmente adecuado para misiones espaciales. En concreto usan el MSP430 en sus satélites Ceres, perteneciente a la familia Arkyd. La misión principal de estos satélites (forman una constelación de diez satétiles) es analizar la firma espectral de los cultivos proporcionando información personalizada a los agricultores, identificando fuentes de energía y minerales, así como tuberías e infraestructuras remotas. El sistema también es capaz de seguir floraciones de algas tóxicas, analizar la calidad del agua y permite la detección del fuego en sus etapas tempranas.

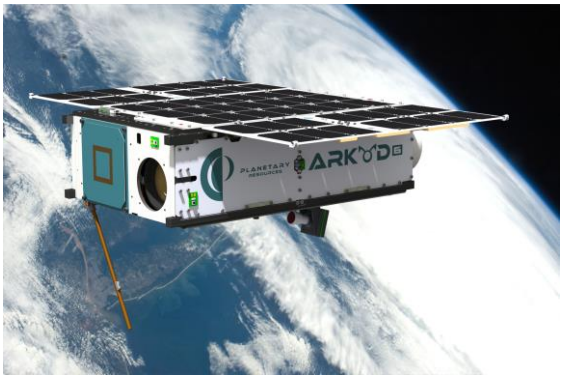


Ilustración 2. Satélite Ceres. Planetary Resources [2]

El proyecto *KickSat*[4] monta en más de cien satélites un modelo de la familisa MSP430. Cada Sprite -es así como denomina a cada satélite su creador, Zac Manchester -contiene un microcontrolador de la serie MSP430. Su elección se debe fundamentalmente a su ultra bajo consumo de energía. Cada satélite orbita a una altitud de entre 280 km y 360 km, transmitiendo y recibiendo datos a la Tierra. Toda la información sobre el proyecto [3]

¿Por qué es útil una herramienta de inyección de fallos? Veamos qué es la inyección y por qué es beneficioso disponer de esto.

Se han expuesto los problemas de radiación que se dan en el espacio. Al diseñar un dispositivo electrónico que se pretenda montar en un satélite hay que tener en cuenta todos esos factores que pueden hacer que el diseño deje de funcionar o no funcione de la forma adecuada. Aunque el diseño se haga pensando en esto, tras la construcción de los diversos sistemas que componen el circuito habrá que testarlos para verificar que funciona bajo las situaciones en las que trabajará. Crear un ambiente que simule las condiciones del espacio y probar el dispositivo tiene un coste de entre 6000 y 8000 euros diarios. Si cada parte del diseño necesita supongamos de media 5 o 6 días, el precio de poner en órbita un sistema empieza a dispararse. Sí, se pueden hacer redundantes en tres o cuatro veces las partes críticas del sistema, pero ya se ha expuesto que el precio de los dispositivos a usar es elevado, luego se continúa aumentado el coste de poner este sistema operativo. Pero, ¿y si tuviésemos un sistema en el que podamos simular nuestro circuito e inyectar fallos donde deseemos, y los resultados nos permitan identificar las partes críticas o procesos críticos del sistema? Si se tienen identificadas qué partes o procesos son las más débiles, se puede optar por algunas de las técnicas que hagan más fiable el diseño. Si esto es posible, aunque haya que simular el sistema en un ambiente de radición, ya se sabe con una alta probabilidad donde pueden ocurrir errores y por tanto esas partes o procesos irán reforzadas de alguna forma. Esto permite reducir costes de producción, reducir uso de recursos de silicio, asi como los tiempos de testeo del dispositivo, pues es más rápido el estudio mediante un sistema de simulación, que el uso de un acelerador de partículas y su

[4] Proyecto de satélite artificial inagurado en octubre de 2011 con el objetivo de poner en marcha un gran número de pequeños satélites de un tipo Cubesat.

posterior análisis de resultados.

FT-UNSHADES 2 (Fault Tolerant-University of Sevilla HArdware DEbugging System) es un sistema de emulación de ambientes ionizantes de radiación ionizante en las etapas tempranas del proceso de diseño de un circuito integrado o de FPGAs SRAM. Se han expuesto las ventajas que presentan las FPGAs SRAM en su aplicación espacial, y además disponemos de un sistema que permite diseñar y emular nuestro diseño en ambientes de radiación.

La herramienta, FT-UNSHADES ha sido desarrollada por profesores de la Escuela Técnica Superior de Ingeniería de la Universidad de Sevilla. Se ha desarrollado un modelo cliente servidor, mediante el cual el cliente se conecta con el servidor a través de un sistema de claves y cuentas de usuario proporcionadas por la Universidad de Sevilla, permitiéndole manejar el sistema. El sistema permite comparar el comportamiento de una FPGA golden, a la cual no se le inyectan fallos, con el comportamiento de una FPGA target que si se le pueden inyectar los fallos que se deseen[5]. Si no se le inyectan errores a esta segunda, se observa que el comportamiento de ambas es idéntico.

Ahora sí estamos en condiciones de presentar el objetivo y alcance de este proyecto.

En el proyecto presente se ha implementado una versión del MSP430 en una FPGA, en concreto en la XC5VFX70T. La descripción del diseño está hecha en Verilog por Olivier Girard[6]. El openMSP430, así es como se denomina este diseño, es un microcontrolador de 16 bits sintetizable compatible con la familia de microcontroladores de Texas Instruments MSP430. El core viene con algunos periféricos (16x16 Hardware Multiplier, Watchdog, GPIO, Timer A, plantillas genéricas), una interfaz DMA y dos salidas Serial Debug Interface. A continuación, se muestra un esquema básico de los componentes de este diseño.

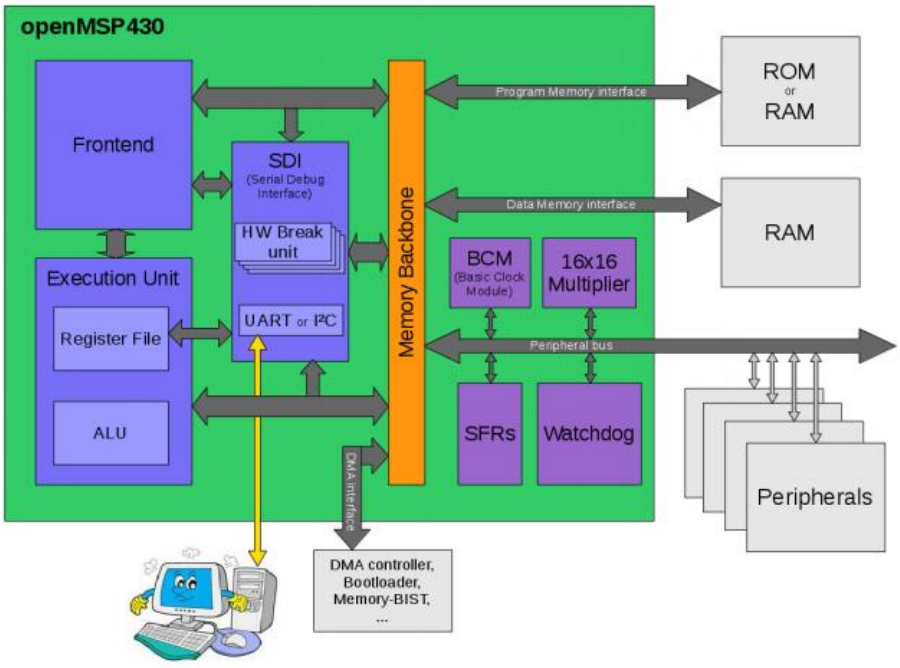


Ilustración 3. Estructura del openMSP430 [4]

Se parte de una modificación del código original que implementa una versión con 8 kB de memoria ROM, 1kB de memoria RAM y 512 B de memoria de periféricos. A partir de este, se pretende implementar un microcontrolador con mayores capacidades, en concreto se implementará un dispositivo con 48 kB de memoria ROM o de programa, 10 kB de memoria RAM o de datos y 512 B de memoria de periféricos. El criterio de elección de estas características se debe a que coinciden con las del microcontrolador comercial MSP430F1611. Por tanto, se tiene un diseño libre con un comportamiento y unas características similares a la del microcontrolador comercial, incorporando las características ventajosas de estar sintetizado en una FPGA

[5] El funcionamiento de FTU se describe en el Capítulo 2.
[6] Ingeniero de diseño de Apple,

SRAM.

Ilustración 4. MSP430F1611 [5]

El primer objetivo del proyecto es conseguir que el dispositivo funcione de foma adecuada. En primer lugar, comprobaremos que el sistema funciona en el simulador ISIM de Xilinx, y conseguido esto el siguiente objetivo será conseguir que el microcontrolador funciona en la FPGA real, para lo cual se usará el emulador FT-UNSHADES. Para que funcione en el dispositivo real, primeramente se hará mediante un archivo de configuración .bit generado directamente con la herramienta ISE que cargará el contenido en memoria. Con esto se estudiará la organización de las palabras en los bloques de memorias generados por el software ISE y con ello se podrá generar un diseño con la memoria ROM vacía donde se podrá cargar el programa que se desee mediante la herramienta **data2mem**. Con esto se acelerará el proceso de carga de programas en memoria, pues con un diseño hardware que no cambia podremos cargar programas en cuestión de segundos. Hay que matizar que para llevar a cabo esto hay que conocer cómo se comunican los bloques de memoria entre sí. Para ello se pueden generar una instancia de memoria y cargando palabras conocidas se irá descifrando la interacción entre las mismas[7]. Concluido esto habremos conseguido un flujo de trabajo para implementar el microcontrolador en FT-UNSHADES y poder llevar a cabo una campaña de inyección.

[7] Este proceso se detalla en el Capítulo 3.

2 OPENMSP430

Todo debe simplificarse lo máximo posible, pero no más.

Albert Einstein

2.1 Arquitectura del openMSP430

El openMSP430 es un microcontrolador de 16 bits descrito en Verilog. Es compatible con la familia de microprocesadores MSP430 de Texas Instruments y puede ejecutar el código generado por una herramienta del MSP430 de una forma muy parecida. La diferencia con este se encuentra en que no es 100% cycle-accurate, es decir que los resultados son los mismos pero tarda algunos ciclos más o menos.

El diseño incorpora algunos periféricos (16x16 Hardware Multiplier, Watchdog, GPIO, Timer A, plantillas genéricas), una interfaz DMA y dos salidas Serial Debug Interface. El compilador necesario para obtener los archivos que soporta el diseño es el MSP430-GCC [6] .

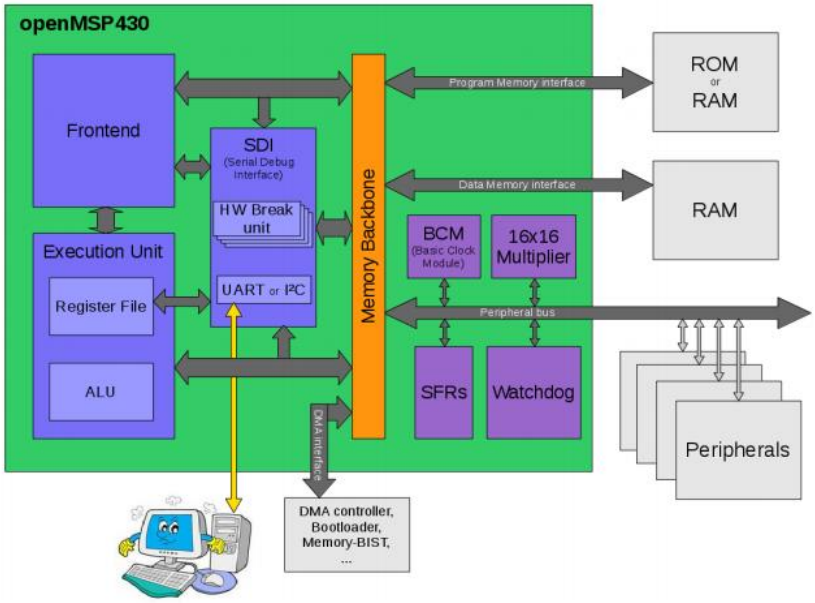


Ilustración 5. Arquitectura openMSP430 [4]

El openMSP430, al igual que el MSP430, se basa en una arquitectura de Von Neumann, con un solo espacio de direcciones para instrucciones y datos. La memoria se direcciona por bloques de 1 byte, es decir, las direcciones de memoria se expresan en bytes, y no en palabras de 16 bits, y los pares de byte se combinan para hacer palabras de 16 bits.

2.2 El archivo openMSP430_defines

El diseño incorpora un archivo, ***openMSP430_defines.v,*** localizado en el directorio ***rtl*** que permite configurar el openMSP430 según la memoria de programa, datos y periféricos que deseemos. Es muy importante tener en cuenta que la suma de la memoria de datos, programa y periféricos no debe de superar los 64 kB.

En este archivo, tres grupos de parámetros permiten personalizar el diseño. El orden de estos parámetros es el de tamaño de memoria de programa (***Program Memory Size),*** tamaño de memoria de datos (***Data Memory Size),*** y tamaño de memoria de periféricos (***Peripheral Memory Size)***. El archivo incorpora por defecto una serie de tamaños de memoria de datos, programa y periféricos predefinidos por el autor, pero realmente se pueden configurar los tamaños que deseemos de cada memoria siempre y cuando se cumpla con la condición expuesta más arriba y que se vuelve a insistir a continuación: **La suma de la memoria de programa, datos y periféricos no puede superar los 64 kB.**

El proyecto se ha montado en el software ISE 14.7 de Xilinx. Con el proyecto listo para editar en este software se pasa a explicar la generación de los elementos básicos de este diseño.

A continuación se muestran partes del archivo ***openMSP430_defines.v*** donde se deben de configurar estas opciones:

```
//============================================================================
=
//============================================================================
=
// BASIC SYSTEM CONFIGURATION
//============================================================================
=
//============================================================================
=
//
// Note: the sum of program, data and peripheral memory spaces must not
//       exceed 64 kB
//

// Program Memory Size:
//                      Uncomment the required memory size
//-----------------------------------------------------
//`define PMEM_SIZE_CUSTOM
//`define PMEM_SIZE_59_KB
//`define PMEM_SIZE_55_KB
//`define PMEM_SIZE_54_KB
//`define PMEM_SIZE_51_KB
`define PMEM_SIZE_48_KB
//`define PMEM_SIZE_41_KB
//`define PMEM_SIZE_32_KB
//`define PMEM_SIZE_24_KB
//`define PMEM_SIZE_16_KB
//`define PMEM_SIZE_12_KB
//`define PMEM_SIZE_8_KB
//`define PMEM_SIZE_4_KB
//`define PMEM_SIZE_2_KB
//`define PMEM_SIZE_1_KB
```

```
// Data Memory Size:
//                     Uncomment the required memory size
//-------------------------------------------------------
//`define DMEM_SIZE_CUSTOM
//`define DMEM_SIZE_32_KB
//`define DMEM_SIZE_24_KB
//`define DMEM_SIZE_16_KB
`define DMEM_SIZE_10_KB
//`define DMEM_SIZE_8_KB
//`define DMEM_SIZE_5_KB
//`define DMEM_SIZE_4_KB
//`define DMEM_SIZE_2p5_KB
//`define DMEM_SIZE_2_KB
//`define DMEM_SIZE_1_KB
//`define DMEM_SIZE_512_B
//`define DMEM_SIZE_256_B
//`define DMEM_SIZE_128_B


//-------------------------------------------------------
// Peripheral Memory Space:
//-------------------------------------------------------
// The original MSP430 architecture map the peripherals
// from 0x0000 to 0x01FF (i.e. 512B of the memory space).
// The following defines allow you to expand this space
// up to 32 kB (i.e. from 0x0000 to 0x7fff).
// As a consequence, the data memory mapping will be
// shifted up and a custom linker script will therefore
// be required by the GCC compiler.
//-------------------------------------------------------
//`define PER_SIZE_CUSTOM
//`define PER_SIZE_32_KB
//`define PER_SIZE_16_KB
//`define PER_SIZE_8_KB
//`define PER_SIZE_4_KB
//`define PER_SIZE_2_KB
//`define PER_SIZE_1_KB
`define PER_SIZE_512_B
```

Si quitamos los comentarios de algunos de los tamaños que incluye por defecto el microprocesador, ya definimos los tamaños de memoria de cada tipo. En el caso de que se deseara disponer de una cantidad de memoria distinta de las que se muestran, se eliminarían los comentarios, *`define PMEM_SIZE_CUSTOM*, *`define DMEN_SIZE_CUSTOM* , y *`define PER_SIZE_CUSTOM* , y los tamaños se definirían en la parte del archivo que se muestra a continuación:

```
/// Custom Program/Data and Peripheral Memory Spaces
//-------------------------------------------------------
// The following values are valid only if the
// corresponding *_SIZE_CUSTOM defines are uncommented:
//
//  - *_SIZE   : size of the section in bytes.
//  - *_AWIDTH : address port width, this value must allow
//               to address all WORDS of the section
//               (i.e. the *_SIZE divided by 2)
```

```
//------------------------------------------------------

// Custom Program memory (enabled with PMEM_SIZE_CUSTOM)
`define PMEM_CUSTOM_AWIDTH      15
`define PMEM_CUSTOM_SIZE      49152
//`define PMEM_CUSTOM_AWIDTH      14
//`define PMEM_CUSTOM_SIZE      32768
//`define PMEM_CUSTOM_AWIDTH      12
//`define PMEM_CUSTOM_SIZE      8192

// Custom Data memory    (enabled with DMEM_SIZE_CUSTOM)
`define DMEM_CUSTOM_AWIDTH       13
`define DMEM_CUSTOM_SIZE       10240
//`define DMEM_CUSTOM_AWIDTH       11
//`define DMEM_CUSTOM_SIZE       4096
//`define DMEM_CUSTOM_AWIDTH       9
//`define DMEM_CUSTOM_SIZE       1024

// Custom Peripheral memory  (enabled with PER_SIZE_CUSTOM)
`define PER_CUSTOM_AWIDTH        8
`define PER_CUSTOM_SIZE        512
```

Aunque se han definido tamaños incluidos por defecto, esta parte del código queda ignorada al no estar descomentadas las sentencias expuestas más arriba.

Toda la familia MSP430 tiene una memoria de periféricos de 512 B. En cambio, el openMSP430 también puede configurar la cantidad de memoria de periféricos que se desea establecer, está comprendida está entre 512 B y 32kB. Esto presenta ventajas frente al modelo comercial, ya que se pueden establecer tantos periféricos como se desee.

El openMSP430 soporta un vector de interrupciones de tamaño 16, 32 o 64 interrupciones. Para el microprocesador implementado en este proyecto se ha elegido la opción de 16, ya que con esta será suficiente para el objetivo que se pretende alcanzar.

2.3 Generación de memorias

Configurado el archivo ***opemMSP430_defines.v***, hay que generar o regenerar las memorias de programas y datos. En el caso tratado, al partir de un modelo de menores prestaciones, pero idéntica estructura en cuestión de diseño, solo se han regenerado las memorias. Si partimos de un proyecto en el que no existen estas memorias, se generarían mediante la utilidad "Core Generator" que se ubica en las herramientas ("Tools") de ISE.

Para la memoria de programa, llamada ***pmem.xco,*** las opciones que se deben seleccionar o editar son:

- Memory Type: Single Port RAM
- Seleccionar "Use Byte Write Enable". El tamaño de Byte debe ser de 8.
- Seleccionar "Minimum Area"
- La anchura de las palabras que se escribirán en la memoria de programa es de 16 bits, luego en la opción "Write Width" se escribirá 16.

- La profundidad de la memoria será de la mitad de la memoria que deseemos. Como cada bloque contiene palabras de 8 bits, para conseguir palabras de 16 hay que conexionar estos bloques, y por tanto estamos reduciendo la profundidad de la memoria a la mitad. Es decir, si quiero una memoria de 48kB, como es el caso presente, debemos escribir una profundidad de memoria de 24576 (49152/2).

- Operating Mode: Write First
- Para la operación de escritura hay que seleccionar que el pin de habilitación esté activo: Use ENA Pin

Para la memoria de datos, ***dmem.xco***, las opciones que debemos seleccionar son las mismas que las mostradas para la memoria de programa, modificando los tamaños que sean necesarios. A continuación, se presentan dichas opciones:

- Memory Type: Single Port RAM
- Seleccionar "Use Byte Write Enable". El tamaño de Byte debe ser de 8.
- Seleccionar "Minimum Area"
- La anchura de las palabras que se escribirán en la memoria de programa es de 16 bits, luego en la opción "Write Width" se escribirá 16.
- La profundidad de la memoria será de la mitad de la memoria que deseemos. Como cada bloque contiene palabras de 8 bits, para conseguir palabras de 16 hay que conexionar estos bloques, y por tanto estamos reduciendo la profundidad de la memoria a la mitad. Es decir, si quiero una memoria de 48kB, como es el caso presente, debemos escribir una profundidad de memoria de 24576 (49152/2).
- Operating Mode: Write First
- Para la operación de escritura hay que seleccionar que el pin de habilitación esté activo: Use ENA Pin.

2.4 Registros de interés del openMSP430

El openMSP430 dispone de varios registros que resultan de interés para estudios posteriores. Destacan los siguientes:

- Registro R0. Contador de programa. Este registro en FTU[8] se denomina pc (program counter) y recorre las direcciones de memoria del microcontrolador de forma global. Es decir, si nuestra memoria es de 48kB y localmente se ha definido que su espacio está comprendido entre [0x0000:0xBFFF] (notación hexadecimal), de forma global está memoria está comprendida entre la dirección [0x4000:0xFFFF]. A continuación se muestra un mapa de organización de la memoria para el MSP430F1611 que coincide con el mapa de memoria del openMPS430 para estas características.

memory organization, MSP430F161x

		MSP430F1610	MSP430F1611	MSP430F1612
Memory Main: interrupt vector Main: code memory	Size Flash Flash	32KB 0FFFFh – 0FFE0h 0FFFFh – 08000h	48KB 0FFFFh – 0FFE0h 0FFFFh – 04000h	55KB 0FFFFh – 0FFE0h 0FFFFh – 02500h
RAM (Total) Extended Mirrored	Size Size Size	5KB 024FFh – 01100h 3KB 024FFh – 01900h 2KB 018FFh – 01100h	10KB 038FFh – 01100h 8KB 038FFh – 01900h 2KB 018FFh – 01100h	5KB 024FFh – 01100h 3KB 024FFh – 01900h 2KB 018FFh – 01100h
Information memory	Size Flash	256 Byte 010FFh – 01000h	256 Byte 010FFh – 01000h	256 Byte 010FFh – 01000h
Boot memory	Size ROM	1KB 0FFFh – 0C00h	1KB 0FFFh – 0C00h	1KB 0FFFh – 0C00h
RAM (mirrored at 018FFh - 01100h)	Size	2KB 09FFh – 0200h	2KB 09FFh – 0200h	2KB 09FFh – 0200h
Peripherals	16-bit 8-bit 8-bit SFR	01FFh – 0100h 0FFh – 010h 0Fh – 00h	01FFh – 0100h 0FFh – 010h 0Fh – 00h	01FFh – 0100h 0FFh – 010h 0Fh – 00h

Figure 1. Memory organization MSP430F161x [7]

- R1. Puntero de pila

[8] Ver Capítulo 3 para una descripción de la herramienta.

- R2. Registro de estado.
- R3. Registro denominado generador de constantes. Provee el acceso a 6 valores constantes utilizados de forma habitual.
- Registros R4 hasta R15. Registros de propósito general.

3 FT-UNSHADES

La ciencia de hoy es la tecnología del mañana

Edward Teller

3.1 Introducción a FT-UNSHADES 2

Con el Sistema FT-UNSHADES (Fault Tolerance- University of Sevilla HArdware DEbugging System) es posible simular ambientes de radiación inonizantes en el proceso de diseño de un circuito integrado sintetizable en la FPGA SRAM Virtex 5.

Esto proporciona un elevado potencial de diseño, ya que desde el primer prototipo de circuito integrado sintetizable que se disponga se puede empezar a testar inyectándole, para ello, fallos en los registros o memorias que se considere que sean propicias a alterar el funcionamiento correcto del circuito. El potencial de todo esto se encuentra en que gracias a estas inyecciones se pueden conocer desde el primer momento los procesos más críticos o las partes más débiles del diseño realizado, pudiendo reconfigurar cuantas veces se considere necesario el diseño y estudiar de nuevo la respuesta de este rediseño.

Como se expuso en la introducción de este trabajo, disponer de un sistema así no solo supone una ventaja sobresaliente a la hora de diseñar, sino que también repercute de forma directa en los costes de desarrollo de cualquier circuito sintetizable que vaya a estar expuesto a condiciones ionizantes. Ciertas aplicaciones requieren ensayos de radiación. Estos ensayos hay que llevarlos a cabo aunque se disponga de este sistema, pero si que permite ahorrar tiempo de los mismos y por tanto costes. Además nos proporciona la capacidad de optimizar los recursos de los que disponga la FPGA en la que vaya a ser descrito el diseño, ya que conociendo las partes críticas del mismo habría que aplicar técnicas de refuerzo -ya sea redundancia hardware o robustez software-, quedando el resto de recursos disponibles para describir cualquier otro circuito integrado en el caso de que se optara por la redundancia hardware.

Existen dos guías, denominadas **UFF 3.5** User Guide y **UFF 3.5** Getting Started Guide, que explican los archivos necesarios para el funcionamiento de esta plataforma, así como la generación de estos mismos archivos usando el software ISE 14.7 de Xilinx y el simulador que incorpora, ISim. Aunque ahí se detallan las funciones de las que dispone el sistema y como usarlas, se introducirán algunos conceptos durante este capítulo.

Debido a la complejidad de transporte de este sistema, se ha desarrollado una version cliente-servidor, así solo es necesario un usario y una clave proporcionada por la Universidad de Sevilla para trabajar con el sistema. Actualmente, el cliente solo puede hacer uso de una unidad hardware, por lo que es recomendable que mientras no se esté usando se dejé dicha unidad libre a disposición del resto de usuarios.

Es importante indicar que las FPGAs que monta esta plataforma son las Virtex XC5VFX70T. Cuando se esté realizando el diseño en ISE 14.7 habrá que establecer esta FPGA en las opciones del proyecto.

3.2 Modos de trabajo

La plataforma FT-UNSHADES o FTU, es así como se le denomina de forma habitual, presenta dos modos de trabajo según el objetivo que pretendamos conseguir.

3.2.1 Modo Debug

El modo Debug resulta adecuado para las primeras etapas del diseño. Con este modo se pueden observar las salidas, entradas o registros del sistema que se deseen, permitiendo estudiar el comportamiento de los componentes indicados. Permite depurar el comportamiento del diseño.

Esto ofrece un gran potencial a la hora de estudiar nuestro diseño y verificar que el comportamiento es el que se desea.

Aunque existen simuladores, como ISim, que nos permiten verificar que la descripción del diseño funciona como se pretende, estas herramientas no nos permiten verificar si los archivos que se generan del proceso de síntesis sintetizan el diseño de forma adecuada. Disponer de un sistema como FTU permite comparar el funcionamiento entre la simulación y el funcionamiento real en la FPGA.

En este Proyecto, este modo ha sido de gran utilidad, ya que ha permitido, entre otras cosas, verificar qué herramientas estaban funcionando de forma correcta y cuáles no, ofreciendo resultados indicativos de dónde se encontraban los errores.

3.2.2 Modo Campaña

El modo campaña es útil en el momento que se puede verificar que el circuito diseñado funciona de forma correcta bajo condiciones de ausencia de radiación.

Concluido el primer prototipo del circuito que se pretende implementar y conocido que su comportamiento, bajo condiciones de ausencia de radiación, es el adecuado, se está en condiciones de pasar a testar el circuito emulando condiciones ionizantes de trabajo. Con el modo Campaña se pueden programar una serie de inyecciones de fallos en los registros que se desee y observar el comportamiento del circuito integrado tras los mismos.

El funcionamiento correcto o incorrecto del diseño se conoce porque, tanto en el modo Debug como en el modo Campaña, los estimulos están ejecutándose en dos FPGAs al mismo tiempo. En el modo Debug, al no inyectar fallos, el comportamiento de ambas unidades debe ser idéntico. Sin embargo, en el modo Campaña la inyección de fallos se lleva a cabo solo en una de las unidades - distinguimos entre una FPGA golden y una FPGA target, haciendo la inyección en la FPGA target-, en la FPGA target, y teniendo como referencia la unidad golden. Con ello se puede observar a partir de que instante difieren los comportamientos de ambas unidades, identificando por qué se ha producido el error y en qué instante. Esto proporciona un potencial de diseño muy grande.

Apuntando a este proyecto, para el openMSP430, es interesante la inyección de fallos en los registros R0 a R15. El significado de cada registro se expone en el aparatdo **1.4** del Capítulo 1.

Tras la finalizar el modo campaña se genera una carpeta de resultados, denominada ***results***, que contiene los siguientes archivos:

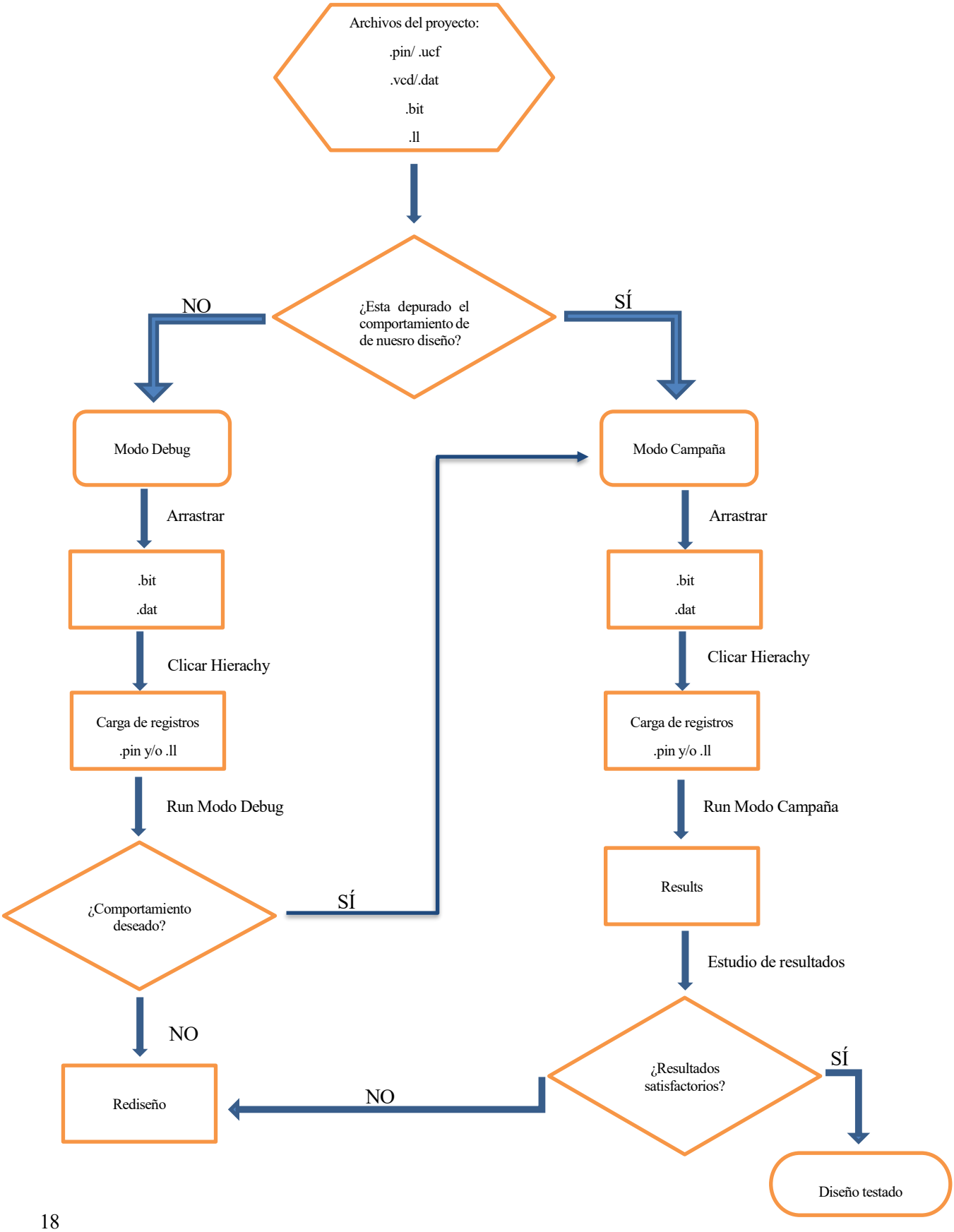
- damages.csv. Contiene la información que se configuró al inicio de la campaña para ser guardada.
- injections.csv. Guarda información relacionada con todas las inyecciones llevadas a cabo durante la campaña.

- reg_names.txt. Contiene una lista de los registros a los que se le han inyectado errores durante la campaña. Asigna una índice numérico a cada registro.
- run.tcl. Script que permite reproducir la campaña ejecutada.
- stats.txt. Muestra resultados estadísticos de la campaña.

Toda la información puede ampliarse en el manual **UFF 3.5** User Guide.

3.3 Diagrama de flujo para uso de FTU

En este apartado, se presenta de forma general el diagrama de flujo que se debe de seguir para trabajar con FTU según la etapa de diseño en la que se encuentre el circuito. No pretende ser un esquema que contradiga lo expuesto en los manuales, solo exponer de forma gráfica los pasos a seguir si queremos hacer uso de esta plataforma y se pretende implementar un modelo de microcontrolador openMSP430. La metodología expuesta es la que se ha seguido durante el poyecto.


Archivos del proyecto:
.pin/ .ucf
.vcd/.dat
.bit
.ll
¿Esta depurado el comportamiento de de nuesro diseño?
NO
SÍ
Modo Debug
Arrastrar
.bit
.dat
Clicar Hierachy
Carga de registros
.pin y/o .ll
Run Modo Debug
¿Comportamiento deseado?
SÍ
NO
Rediseño
Modo Campaña
Arrastrar
.bit
.dat
Clicar Hierachy
Carga de registros
.pin y/o .ll
Run Modo Campaña
Results
Estudio de resultados
¿Resultados satisfactorios?
NO
SÍ
Diseño testado

4 PREPARACIÓN DEL DISEÑO Y FLUJO DE TRABAJO

Un viaje de diez mil kilómetros empieza por un solo paso

Proverbio chino

4.1 Introducción a la problemática presentada por el diseño y justificación del modelo implementado

La implementación de un modelo concreto del openMSP430 introduce una problemática cuya solución no es inmediata, ya que implementarlo no implica solo llevar a cabo el proceso de síntesis para obtener un archivo de configuración .bit, una simulación en ISim para obtener el archivo .vcd, y con ellos el microcontrolador funciona en FTU.

Se plantean varias preguntas a las que se irá respodiendo a lo largo del capítulo, y que se exponen a continuación. Configurado el modelo concreto del openMSP430 que se desea implementar, sabemos que el número de block rams que se han generado son N, en nuestro caso 12, pero si mi objetivo es generar un modelo sin archivo que inicialice la memoria[9]:

- ¿Cómo genero un archivo que ordene las memorias y describa la relación entre ellas?
- ¿Cómo se llama este archivo?
- ¿Qúe herramienta uso?
- ¿Cuáles son las direcciones de memoria dentro del microcontrolador?
- ¿Qúe tipo de archivo hay que cargar en memoria y cómo se genera?

[9] Si se carga un archivo .coe en memoria de programa se puede observar el comportamiento en simulación, pero este no es el flujo de trabajo ideal ya que cambiar el programa requeriría una nueva implementación del diseño.

- Si no funciona, ¿se puede de alguna forma buscar dónde se encuentra el origen del problema?

Para el tratamiento de toda esta problemática se recomienda usar un buen editor de texto, ya que muchos resultados dependerán de él.

En cuanto a la justificación del modelo de openMSP430 implementado, se ha optado por la implementación de un diseño que disponga de 48kB de memoria de programa o memoria ROM, y 10kB de memoria de datos o memoria RAM. La memoria de periféricos es de 512B. La elección de estas características se debe a las siguientes razones:

- Con los tamaños de memoria de programa y memoria de datos seleccionada, se dispone de un diseño con una elevada versatilidad para ser testado con diversos programas.
- ¿Por qué 48kB y 10kB y no otros números?

  Se pueden elegir los tamaños que se deseen siempre y cuando se cumpla con la condición definida en el **Capítulo 1**, pero si se elige un diseño con estas características expuestas, se dispone de un circuito con las mismas capacidades que el modelo MSP430F1611 [8] de Texas Intruments.

4.2 Generación de archivos para implementación

Configurado el microcontrolador con las especificaciones requeridas, hay que generar los archivos necesarios para la implementación en FTU2. Para llevar a cabo esta implementación es necesario disponer de varios archivos, presentados a continuación, para los cuales hay que seguir los pasos expuestos en el manual de FTU2 **UFF 3.5** Getting Started Guide, salvo una excepción expuesta en este apartado.

Los archivos necesarios para que sea implementable el diseño en FTU2 son:

- Archivo name.pin. En este archivo se definen las señales de nuestro microcontrolador. En este caso estas señales se ubican en el archivo ***wr_openmsp430.vhd***. Para generar este archivo se necesita un editor de texto que nos permita salvar el mismo con la extesión .pin.

  Aunque la sintaxis del mismo se expone en la guía indicada, a continuación se muestra el archivo del openMSP430 sintetizado

```
--control

dco_clk

--input

reset_n

irq

nmi

uart_rxd

p1_in_ext 7 0


p2_in_ext 7 0

--bidir
```

```
--output

uart_txd

p1_pout_ext 7 0

p2_pout_ext 7 0
```

Con este archivo se debe generar el archivo .ucf tal y como se indica en la guía expuesta en esta sección. Este archivo .ucf hay que añadirlo a nuestro proyecto en ISE, ya que contiene las conexiones físicas de los pines de la FPGA con cada señal del microcontrolador.

- Añadido el archivo ucf al proyecto, se está en condiciones de generar los archivos .ll y .bit, entre otros generados al ejecutar el proceso de implementación, para cargarlos en FTU. La generación de estos archivos se lleva a cabo siguiendo las pautas propocionadas por ***UFF 3.5*** Getting Started Guide.
- Archivo name.vcd. Para generar el archivo vcd, hay que tener el cuenta la siguiente modificación.

  En primer lugar, cuando se intenta llevar a cabo la simulación en ISim, se generan una serie de errores referidos a que existe un índice, que recorre un vector, que apunta a la componente -1 del mismo. Realmente en los archivos que este error aparece, se quiere indicar que este índice debe recorrer 64 componentes, desde la 0 hasta la 63, pero en vez de escribir el número 63 se define una variable, cuyo valor es 64, y se le resta 1. La exprexión definida no es interpretada de forma correcta por ISE y genera un error. La solución al mismo es elimiar todas esas expresiones y sustituirlas directamente por 63.

  En segundo lugar, para generar el archivo vcd, hay que matizar una de las directrices expuestas por la guía, ya que si no se hace cuando se intente generar el archivo .dat, a partir del .pin y .vcd, se imprime por pantalla un mensaje de error indicando que existen elementos multibits en este archivo name.vcd. Veamos que indica la guía y como debe de hacerse[10].

  Instrucciones según **UFF 3.5** Getting Started Guide:

  1. In the simulation console type:

     restart

  2. In the simulation console type:

     vcd dumpfile <design>.vcd

  3. In the simulation console type:

     vcd dumpvars -m <inst> -l 2

  4. Run the simulation up to the desired time:
  5. Once the simulation has finished, type:

     vcd dumpflush

  6. In the simulation console type:

     quit

  Modificación para que funcione en FTU2:

  1. In the simulation console type:

     restart

  2. In the simulation console type:

[10] -l<N> hace referencia al número de señales internas del <inst> vuelca en el .vcd.

vcd dumpfile <design>.vcd

3. In the simulation console type:

   vcd dumpvars -m <inst> -l 0

4. Run the simulation up to the desired time:
5. Once the simulation has finished, type:

   vcd dumpflush

6. In the simulation console type:

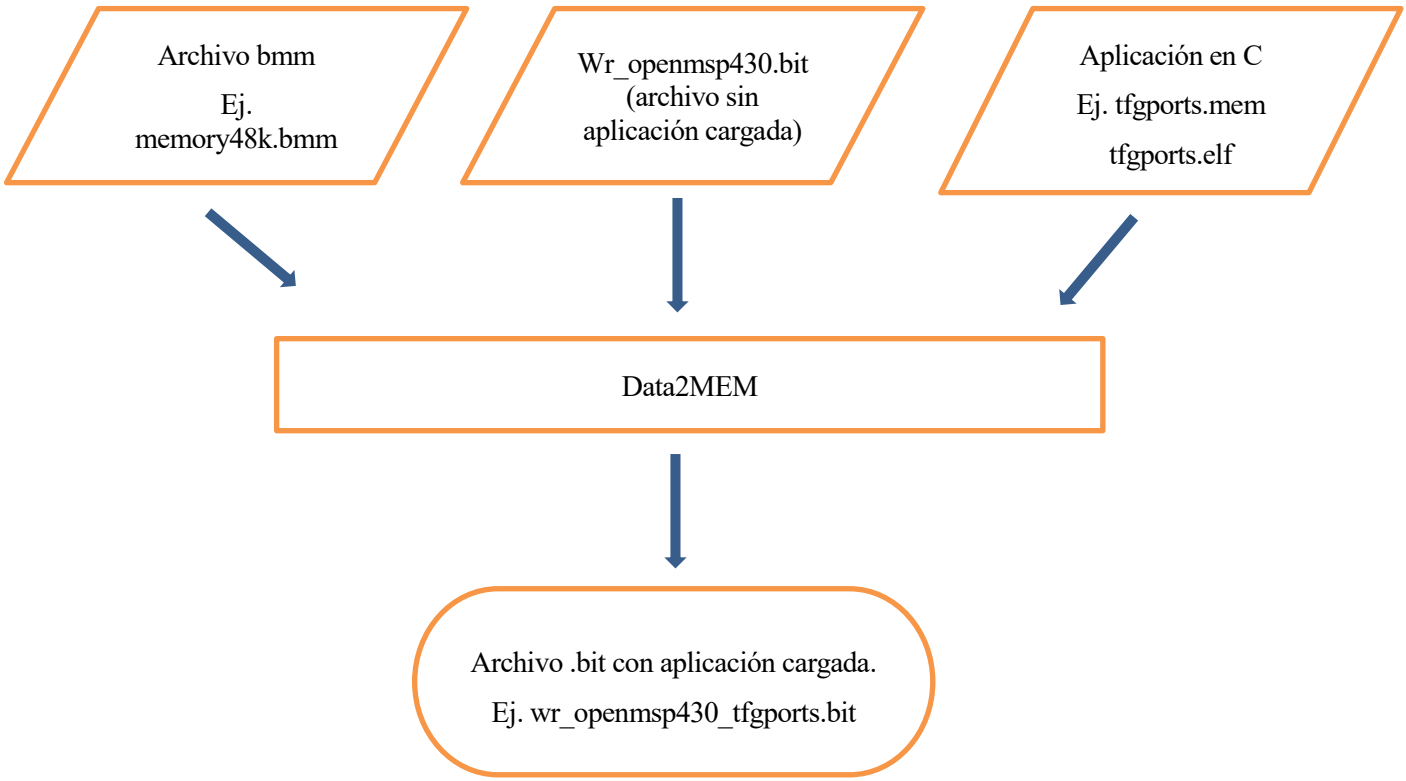
   quit

Generados todos estos archivos, nuestro proyecto en FTU2 está a la espera de modificar el archivo .bit para inicializar las BRAM con el programa que se desee cargar.

4.3 Tratamiento de las Blocks RAM para su uso en el flujo de trabajo

Para cargar en memoria un programa, se necesitan tres archivos:

- Archivo name.bit. Este archivo contiene el rutado de la FPGA para su configuración. Para el trabajo desarrollado, este archivo se denomica ***wr_openmsp430.bit.***
- Archivo name.elf o name.mem -es indiferente, a priori, una u otra extension para que el contenido se cargue en memoria-. Este archivo contiene la apliación que se desea ejecutar en el microcontrolador implementado en la FPGA.
- Archivo name.bmm. Este archivo contiene la relación y el orden de los bloques de memorias que contiene el modelo concreto del openMSP430. Cada bloque tiene un nombre y una ubicación dentro de la FPGA. Es este nombre y su tratamiento el objetivo a tratar en esta sección.



Data2mem es una aplicación command-line. Cómo generar el archivo de implementación con la aplicación cargada se expone en el manual de la herramienta, **Data2MEM User Guide** [9]**.** Para facilitar y agilizar la implementación a quién desee implementar un modelo del openMSP430, se expone el commando que debe ejecutar:

Ruta donde se ubican los 3 archivos anteriores>C:\Xilinx\14.7\ISE_DS\ISE\bin\nt64\data2mem -bm name.bmm -bd aplicacion.mem -bt wr_openmsp430.bit -o b new_name.bit

Si cargamos un archivo elf el comando a ejecutar sería:

Ruta donde se ubican los 3 archivos anteriores>C:\Xilinx\14.7\ISE_DS\ISE\bin\nt64\data2mem -bm name.bmm -bd aplicacion.elf -bt wr_openmsp430.bit -o b new_name.bit

Para el microncontrolador implementado este comando sería:

E:\MSP430F1611_testbmm\Prueba bm 1> C:\Xilinx\14.7\ISE_DS\ISE\bin\nt64\data2mem -bm memory48k.bmm -bd tfgports.mem -bt wr_openmsp430.bit -o b new_openmsp430f1611.bit

Para generar el archivo .bmm hay que obtener el nombre de cada bloque de memoria, o blocks ram. Para ello se abrirá la herramieta ***FPGA Editor***, y de las dos opciones propuestas por este ***Post-Place & Route***. Cuando se abre esta herramienta aparece una lista con todos los componentes que forman el diseño. En esa lista hay que localizar los bloques de memoria, que para este caso concreto son del tipo ***RAMB36.*** Si se clica sobre uno de ellos, a continuación se vuelve a clicar sobre *Zoom Selection* y se pincha sobre el bloque que aparece en color rojo -color por defecto, en la línea de commandos de la herramienta aparece el nombre completo.

Ejecutando esta operación, por ejemplo, para el primer bloque de mmoria que aparece en la lista, se obtiene lo siguiente:

```
comp
"uP_map/pmem/BU2/U0/blk_mem_generator/valid.cstr/ramloop[0].ram.r/v5_n
oinit.ram/SP.SINGLE_PRIM36.SP", site "RAMB36_X1Y24", type=RAMB36_EXP
(RPM grid X53Y240)
```

Si hacemos esto para los siguiente bloques obtenemos el nombre de todos.

¿Qué cambios hay que hacer?

1. Eliminar las comillas y la palabra *comp.*
2. Eliminar todo lo queda por detras de la expresión "`RAMB36_X1Y24"`.
3. Sustituir la expresión *RAMB36_ por PLACED=.*

Cada vez que regeneremos el diseño, la ubicación de las memorias cambiará, -esto se corresponde con el campo "RAMB36_X1Y24", por lo que resulta de interés generar solo una vez el modelo y no volver a llevar a cabo el proceso de síntesis. En el caso de que se volviera a hacer, de los nombres obtenidos (suponiendo que el modelo que se pretender implementar no ha cambiado y tiene la misma memoria de programa y datos) solo se debe cambiar la ubicación.

Para el modelo objetivo de este trabajo, llevando a cabo las instrucciones indicadas, se obtiene lo siguiente:

```
 uP_map/pmem/BU2/U0/blk_mem_generator/valid.cstr/ramloop[0].ram.r/v5_n
oinit.ram/SP.SINGLE_PRIM36.SP PLACED=X1Y24

 uP_map/pmem/BU2/U0/blk_mem_generator/valid.cstr/ramloop[1].ram.r/v5_n
oinit.ram/SP.SINGLE_PRIM36.SP PLACED=X2Y22

 uP_map/pmem/BU2/U0/blk_mem_generator/valid.cstr/ramloop[2].ram.r/v5_n
oinit.ram/SP.SINGLE_PRIM36.SP PLACED=X1Y23

 uP_map/pmem/BU2/U0/blk_mem_generator/valid.cstr/ramloop[3].ram.r/v5_n
oinit.ram/SP.SINGLE_PRIM36.SP PLACED=X0Y23
```

```
  uP_map/pmem/BU2/U0/blk_mem_generator/valid.cstr/ramloop[4].ram.r/v5_n
oinit.ram/SP.SINGLE_PRIM36.SP PLACED=X0Y24

  uP_map/pmem/BU2/U0/blk_mem_generator/valid.cstr/ramloop[5].ram.r/v5_n
oinit.ram/SP.SINGLE_PRIM36.SP PLACED=X0Y22

  uP_map/pmem/BU2/U0/blk_mem_generator/valid.cstr/ramloop[6].ram.r/v5_n
oinit.ram/SP.SINGLE_PRIM36.SP PLACED=X2Y23

  uP_map/pmem/BU2/U0/blk_mem_generator/valid.cstr/ramloop[7].ram.r/v5_n
oinit.ram/SP.SINGLE_PRIM36.SP PLACED=X1Y20

  uP_map/pmem/BU2/U0/blk_mem_generator/valid.cstr/ramloop[8].ram.r/v5_n
oinit.ram/SP.SINGLE_PRIM36.SP PLACED=X1Y21

  uP_map/pmem/BU2/U0/blk_mem_generator/valid.cstr/ramloop[9].ram.r/v5_n
oinit.ram/SP.SINGLE_PRIM36.SP PLACED=X2Y21

  uP_map/pmem/BU2/U0/blk_mem_generator/valid.cstr/ramloop[10].ram.r/v5_
noinit.ram/SP.SINGLE_PRIM36.SP PLACED=X0Y21

  uP_map/pmem/BU2/U0/blk_mem_generator/valid.cstr/ramloop[11].ram.r/v5_
noinit.ram/SP.SINGLE_PRIM36.SP PLACED=X1Y22
```

Las memorias están listas para generar el archivo bmm.

La forma de generar el archivo bmm se expone en el manual **Data2MEM User Guide** [9]**.** Aunque en este documento se explica como generar el archivo de memoria bmm, surgen cuestiones como con qué bloque de memoria se agrupa cada uno de los bloques o dónde direcciono si pretendo escribir algo en una dirección concreta de memoria. La segunda cuestión queda respondida en el apartado **3.5**. Sobre la primera de ellas, aunque también es contestada en ese apartado, su verificación se hará en el siguiente.

4.4 Generación del archivo bmm

Para generar el archivo de organización de memoria name.bmm, en este caso ***memory48k.bmm***, se debe de saber que este archivo, según el manual de la aplicación **data2mem** [9]**,** consta de las siguientes partes, y en el orden que se expone:

1. Definición del espacio de memoria -cantidad de memoria ROM del diseño, nombre dado al espacio (este nombre no influye cuando se usa la misma) y el tipo de memoria usado.

   La sintaxis para esta parte del archivo es como sigue:

   ADDRESS_SPACE program_LANL_rom RAMB32 [0x00000000:0x0000BFFF]

   El nombre asignado a la memoria es program_LANL_rom, el cual, como se ha indicado ya, puede ser el que el usuario desee.

   ADDRESS_SPACE define el espacio de memoria ROM del que dispone el diseño. Para el caso expuesto en este trabajo, se dispone de de 48kB, que en hexadecimal sería un espacio comprendido entre la dirección 0x0000 hasta la 0xBFFF. Hay que aclarar dos cuestiones de interés:

   - La primera de ellas se refiere a si el espacio de direcciones es local o global. La respuesta es que es indiferente, solo que hay que ser consecuente cuando se direccione en el archivo de contenido de memoria name.mem. La herramienta **data2mem** lo entiende como un

direccionamiento local, he aquí la razón por la que es indiferente de dónde a dónde se direccione. Sí hay que tener en cuenta que se deben escribir ocho "8" caracteres en hexadecimal.

Para el caso propuesto de 48kB de espacio de memoria, se podría también definir un espacio comprendido entre la dirección- en hexadecimal de nuevo- 0x4000 hasta la dirección 0xFFFF, tal como estaría globalmente definido en el microcontrolador commercial MSP430F1611. Si se hiciese de esta forma, la cuestión a tener en cuenta es que cuando se direccionen palabras de memoria que deben ser escritas en estos bloques de memoria o Block RAMs, el espacio donde se direcciona está comprendido entre las direcciones propuestas en este párrafo, es decir, entre la 0x4000 hasta la 0xFFFF.

Para evitar confusiones a la hora de direccionar o usar el archivo .mem, -del que se hablará en los apartados que siguen, se recomienda **usar un espacio de direccionamiento local**. Es decir, si el microcontrador elegido tiene 48kB de memoria de programa, como es el caso en cuestión, definir el espacio tal y como se ha hecho aquí, comprendido entre la dirección 0x0000 y 0xBFFF. Si el espacio es, por ejemplo, de 32 kB, definiríamos un espacio comprendido entre las direcciones 0x0000 y 0x7FFF.

- La segunda cuestión a tratar es que, como se ha estado haciendo hasta ahora, el espacio de memoria se define en hexadecimal, y no en otro sistema de numeración. Esta cuestión viene impuesta por la herramienta de la que se hace uso, **data2mem.**

La palabra RAMB32 se refiere al tipo de bloque de memoria Block RAM usado. En la guía de usuario **Data2MEM User Guide** [9] se pueden consultar unas tablas que según la primitiva usada justifica el uso de lo que ahí llama **Memory Type**. A continuación se muestra dicha tabla.

Block RAM Configurations for 36 Kbit Virtex-5 Devices

Primitive	Data Depth	Data Width	Memory Type
RAMB36	32768	1	RAMB32
RAMB36	16384	2	RAMB32
RAMB36	8192	4	RAMB32
RAMB36	4096	8	RAMB32
RAMB36	4096	9	RAMB36
RAMB36	2048	16	RAMB32
RAMB36	2048	18	RAMB36
RAMB36	1024	32	RAMB32
RAMB36SDP	512	64	RAMB32

Ilustración 6. Relación primitiva-tipo de memoria. Fuente [9]

2. Definido el espacio de direcciones, hay que asociar los bloques de memoria de dos en dos, ¿por qué? La respuesta viene dada porque cada bloque de memoria o Block RAM, -como se denomina en el software usado-, es de 8kB. Cada dirección de memoria puede contener 8 bits, o 1 Byte, pero se pretende cargar en memoria palabras de 16 bits, o 2 Bytes, por lo que hay que conexionar de alguna forma dos bloques de memoria para que se escriba la mitad de la palabra en un bloque y la mitad restante en el otro.

   En este punto surge una cuestión de vital importancia, y es cómo se unen los bloques, y qué parte de la palabra se escribe en cada uno. La respuesta no es trivial, ya que veamos lo siguiente:

   - Si dispongo de 2 bloques de memoria, tengo cuatro opciones para probar. Puedo probar cada una de ellas y verificar cual es la correcta.
   - Si dispongo de 4 bloques de memoria, tengo 16 opciones.

- Si aumento el número de bloque, el número de opciones se dispara, y no es viable solucionar el problema a ciegas y probando.

La opción que aquí se propone es la realización de una instancia de memoria- la misma del microcontrolador-, y mediante la herramienta **data2mem** cargar contenido conocido en memoria. Con esto puede deducirse cómo escribe en memoria y cual es el orden correcto de esta. Aún no se aprecia el potencial de esta idea, pero se apreciará en el apartado que continúa.

Queda por definir la sintaxis para asociar los bloques de memoria. Simplemente hay que usar la palabras *BUS_BLOCK Y END_BUS_BLOCK;* para abrir y cerrar un bloque de 2 Blocks RAM relacionadas. A continuación se muestra un ejemplo:

BUS_BLOCK

uP_map/pmem/BU2/U0/blk_mem_generator/valid.cstr/ramloop[6].ram.r/v5_noinit.ram/SP.SINGLE_PRIM36.SP [15:8] PLACED=X2Y23;

uP_map/pmem/BU2/U0/blk_mem_generator/valid.cstr/ramloop[0].ram.r/v5_noinit.ram/SP.SINGLE_PRIM36.SP [7:0] PLACED=X1Y24;

END_BUS_BLOCK;

¿Qué novedad se presenta en estos bloques respecto al apartado **3.3** ? Se ha añadido los bits de la palabra que contiene cada bloque. Aún no se ha explicado por qué se ha hecho así, se hará en el siguiente apartado, pero se pueden observar los corchetes introducidos delante de la palabra ***PLACED***.

El archivo de memoria que queda para el caso desarrollado es el que se expone más abajo.

```
ADDRESS_SPACE program_LANL_rom RAMB32 [0x00000000:0x0000BFFF]

BUS_BLOCK

uP_map/pmem/BU2/U0/blk_mem_generator/valid.cstr/ramloop[6].ram.r/v5_noinit.ra
m/SP.SINGLE_PRIM36.SP [15:8] PLACED=X2Y23;
uP_map/pmem/BU2/U0/blk_mem_generator/valid.cstr/ramloop[0].ram.r/v5_noinit.ra
m/SP.SINGLE_PRIM36.SP [7:0] PLACED=X1Y24;

END_BUS_BLOCK;

BUS_BLOCK

uP_map/pmem/BU2/U0/blk_mem_generator/valid.cstr/ramloop[7].ram.r/v5_noinit.ra
m/SP.SINGLE_PRIM36.SP [15:8] PLACED=X1Y20;
uP_map/pmem/BU2/U0/blk_mem_generator/valid.cstr/ramloop[1].ram.r/v5_noinit.ra
m/SP.SINGLE_PRIM36.SP [7:0] PLACED=X2Y22;

END_BUS_BLOCK;

BUS_BLOCK

uP_map/pmem/BU2/U0/blk_mem_generator/valid.cstr/ramloop[8].ram.r/v5_noinit.ra
m/SP.SINGLE_PRIM36.SP [15:8] PLACED=X1Y21;
uP_map/pmem/BU2/U0/blk_mem_generator/valid.cstr/ramloop[2].ram.r/v5_noinit.ra
m/SP.SINGLE_PRIM36.SP [7:0] PLACED=X1Y23;
```

```
END_BUS_BLOCK;

BUS_BLOCK

uP_map/pmem/BU2/U0/blk_mem_generator/valid.cstr/ramloop[9].ram.r/v5_noinit.ra
m/SP.SINGLE_PRIM36.SP [15:8] PLACED=X2Y21;
uP_map/pmem/BU2/U0/blk_mem_generator/valid.cstr/ramloop[3].ram.r/v5_noinit.ra
m/SP.SINGLE_PRIM36.SP [7:0] PLACED=X0Y23;

END_BUS_BLOCK;

BUS_BLOCK

uP_map/pmem/BU2/U0/blk_mem_generator/valid.cstr/ramloop[10].ram.r/v5_noinit.r
am/SP.SINGLE_PRIM36.SP [15:8] PLACED=X0Y21;
uP_map/pmem/BU2/U0/blk_mem_generator/valid.cstr/ramloop[4].ram.r/v5_noinit.ra
m/SP.SINGLE_PRIM36.SP [7:0] PLACED=X0Y24;

END_BUS_BLOCK;

BUS_BLOCK

uP_map/pmem/BU2/U0/blk_mem_generator/valid.cstr/ramloop[11].ram.r/v5_noinit.r
am/SP.SINGLE_PRIM36.SP [15:8] PLACED=X1Y22;
uP_map/pmem/BU2/U0/blk_mem_generator/valid.cstr/ramloop[5].ram.r/v5_noinit.ra
m/SP.SINGLE_PRIM36.SP [7:0] PLACED=X0Y22;

END_BUS_BLOCK;

END_ADDRESS_SPACE;
```

Se debe constatar, que cuando se abre un espacio de direcciones con la expression `ADDRESS_SPACE`, este también debe de ser cerrado con la expresión `END_ADDRESS_SPACE;`

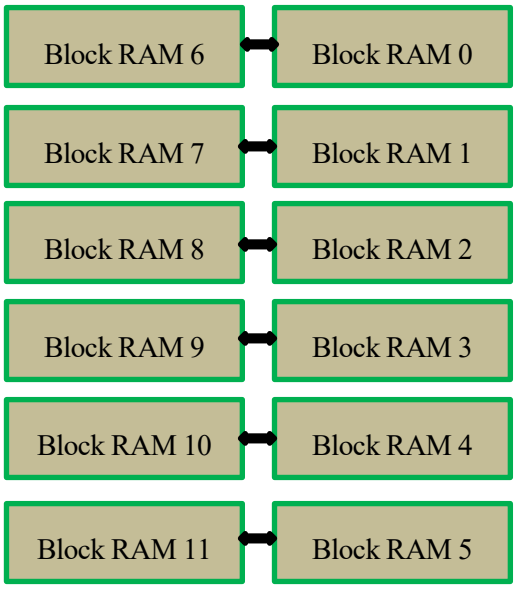
4.5 Instancia de memoria para deducción de relaciones entre Blocks RAM

Para deducir la relación entre las Block RAMs y que parte de las palabras de 16 bits guarda cada una de ellas, si la más significativa o la menos significativa, así como cual es el espacio de direcciones de cada bloque se ha hecho una instancia de la memoria. ¿Cómo se hace esto? Se necesitan los siguientes archivos del proyecto del microcontrolador completo:

- ram_wrapper.vhd. Archivo que instancia a la memoria de programa.
- tb_ram_wrapper.vhd. Es el test bench que define el comportamiento de la instancia de memoria.
- virtex5_pmem.xco. Bloques de memorias generados con **Core Generator**.

Con estos archivos se puede crear una instancia de la memoria de programa del openMSP430 elegido, y cuyo comportamiento es el mismo que en el microcontrolador.

Se dispone de 12 bloques de memoria pero no se sabe como ordenarlos. Hay que hacer una primera prueba para obtener los primeros resultados. La propuesta que aparecía en un foro de Xilinx -con la cual se hizo la primera prueba- era la que se presenta más abajo Además según el orden propuesto por el Catedrático Miguel Ángel Aguirre en la version sobre la que se construye este diseño, los dos bloques de memoria de los que él hace uso concuerda con la propuesta que se encontró. Su validación se hará en el apartado 7 de este capítulo.



¿Por qué tiene lógica este esquema?

Parece lógico que los bloques de memoria no se relacionan de cualquier forma entre sí, o de una forma aleatoria. Si disponemos de 12 bloques de memoria, ordenarlos de esa forma implicaría que una palabra de 16 bits escribiría sus bits más significativos en uno de los bloques de la izquierda, es decir, en el Block RAM 6, 7, 8 9, 10 u 11, y los bits menos significativos los incluiría en unos de los Blocks RAM de la derecha; el 0, 1, 2, 3, 4 o 5.

Este esquema justifica el archivo de memoria, ***memory48k.bmm*** propuesto en el apartado **3.4**.

Si el espacio de memoria está definido entre las direcciones 0x0000 y 0xBFFF, como es el caso propuesto, la primera dirección de memoria es la 0x0000, pero no se sabe cuál es la segunda, o la dirección del primer espacio de memoria del bloque compuesto por la Block RAM 10 y 4.

Para saber en qué espacio escribe y como lo hace, se va a generar un archivo de extensión .mem con un patron de palabras escritas en hexadecimal conocido. La sintaxis de este archivo .mem puede consultarse de nuevo en el manual **Data2MEM User Guide** [9]**.** Se facilita un breve resumen de la misma.

Para generar un archivo con extension mem de forma manual, lo único que hay que llevar a cabo es la escritura en hexadecimal, con la sintaxis que se describe a continuación, de las direcciones en las que se desea escribir.

- Si la dirección es la 0x4000, por ejemplo, la sintaxis es @4000

  Si deseamos escribir en esa dirección la palabra BEEF, por ejemplo, habría que hacer:

  @4000 BEEF

  Si se escribe otra palabra, de 16 bits, a continuación de BEEF, por ejemplo F00D, sería los mismo que escribir lo siguiente:

  @4000 BEEF F00D es equivalente a

  @4000 BEEF

  @4002 F00D

Para el caso en estudio, se hizo sobre una instancia de memoria de 32kB, para entre otras razones facilitar la generación del archive mem. Haciendo uso de lo expuesto, se puede generar un archivo mem, dándole por ejemplo una dirección inicial, y este rellenará de una forma desconocida aún la memoria ROM. El archivo presentado a continuación direcciona desde las 0x8000- ya que el espacio de direcciones para ese caso estaba

comprendido entre la dirección 0x8000 hasta la 0xFFFF- y se rellenan líneas que contienen 32 palabras de 16 bits cada una, pues el tamaño de la memoria es de 32kB. El archivo generado tiene el siguiente aspecto:

```
@8000 AA55 AA55 AA55 AA55 AA55 AA55 AA55 AA55 AA55 AA55 AA55 AA55 AA55 AA55
AA55 AA55 AA55 AA55 AA55 AA55 AA55 AA55 AA55 AA55 AA55 AA55 AA55 AA55 AA55
AA55 AA55 AA55
AA55 AA55 AA55 AA55 AA55 AA55 AA55 AA55 AA55 AA55 AA55 AA55 AA55 AA55 AA55
AA55 AA55 AA55 AA55 AA55 AA55 AA55 AA55 AA55 AA55 AA55 AA55 AA55 AA55 AA55
AA55 AA55
DEAD BEEF DEAD BEEF DEAD BEEF DEAD BEEF DEAD BEEF DEAD BEEF DEAD BEEF DEAD
BEEF DEAD BEEF DEAD BEEF DEAD BEEF DEAD BEEF DEAD BEEF DEAD BEEF DEAD BEEF
DEAD BEEF
DEAD BEEF DEAD BEEF DEAD BEEF DEAD BEEF DEAD BEEF DEAD BEEF DEAD BEEF DEAD
BEEF DEAD BEEF DEAD BEEF DEAD BEEF DEAD BEEF DEAD BEEF DEAD BEEF DEAD BEEF
DEAD BEEF
F00D BAAD F00D BAAD F00D BAAD F00D BAAD F00D BAAD F00D BAAD F00D BAAD F00D
BAAD F00D BAAD F00D BAAD F00D BAAD F00D BAAD F00D BAAD F00D BAAD F00D BAAD
F00D BAAD
F00D BAAD F00D BAAD F00D BAAD F00D BAAD F00D BAAD F00D BAAD F00D BAAD F00D
BAAD F00D BAAD F00D BAAD F00D BAAD F00D BAAD F00D BAAD F00D BAAD F00D BAAD
F00D BAAD
```

Si esto se repite el número de veces necesario hasta que desborde el primer bloque de memoria, se podrá observar donde se escriben las palabras que no ha sido posible escribir en ese primer bloque.

Para cargar el contenido en memoria se necesita un archivo .bit con las memorias de programa vacías, el archivo .mem y el archivo de organización de memoria .bmm, tal y como se expone en el esquema presentado en el apartado **3.3**. Para el caso propuesto, estos archivos reciben el nombre de ***ram_wrapper.bit, memory32.bmm y tm_newspace.mem***. Se insiste en lo ya mencionado; no se ha usado la memoria de 48kB para esta prueba, sino la de 32Kb por la razón expuesta en este apartado.

El commando que se debe de ejecutar, el cual se ha obtenido del manual **Data2MEM User Guide** [9], es:

Ruta donde se ubican los archivos>C:\Xilinx\14.7\ISE_DS\ISE\bin\nt64\data2mem -bm name.bmm -bd aplicacion.mem -bt wr_openmsp430.bit -o b new_name.bit

que para este caso concreto sería:

Ruta donde se ubican los archivos>C:\Xilinx\14.7\ISE_DS\ISE\bin\nt64\data2mem -bm memory32.bmm -bd tm_newspace.mem -bt ram_wapper.bit -o b new_ram_wrapper.bit

Para visualizar el contenido cargado en memoria se debe ejecutar lo siguiente:

Ruta donde se ubican los archivo>C:\Xilinx\14.7\ISE_DS\ISE\bin\nt64\data2mem -bm memory32.bmm –bt new_ram_wrapper.bit -d > memcont

El archivo **memcont** es un archivo de texto plano que contiene, entre otra información, el contenido de los bloques de memoria o Block RAMs. Si se hace esto, para el caso en cuestión, se presentan algunos resultados de interés.

```
BRAM data, Column 00, Row 29. Design instance "uP_map/pmem/BU2/U0/blk_mem_generator/valid.cstr/ramloop[0].ram

 00000000:   55 55 55 55 55 55 55 55 55 55 55 55 55 55 55 55 55 55 55 55 55 55 55 55 55 55 55 55 55 55 55 55
 00000020:   55 55 55 55 55 55 55 55 55 55 55 55 55 55 55 55 55 55 55 55 55 55 55 55 55 55 55 55 55 55 55 55
 00000040:   AD EF AD EF AD EF AD EF AD EF AD EF AD EF AD EF AD EF AD EF AD EF AD EF AD EF AD EF AD EF AD EF
 00000060:   AD EF AD EF AD EF AD EF AD EF AD EF AD EF AD EF AD EF AD EF AD EF AD EF AD EF AD EF AD EF AD EF
 00000080:   0D AD 0D AD 0D AD 0D AD 0D AD 0D AD 0D AD 0D AD 0D AD 0D AD 0D AD 0D AD 0D AD 0D AD 0D AD 0D AD
 000000A0:   0D AD 0D AD 0D AD 0D AD 0D AD 0D AD 0D AD 0D AD 0D AD 0D AD 0D AD 0D AD 0D AD 0D AD 0D AD 0D AD
 000000C0:   55 55 55 55 55 55 55 55 55 55 55 55 55 55 55 55 55 55 55 55 55 55 55 55 55 55 55 55 55 55 55 55
 000000E0:   55 55 55 55 55 55 55 55 55 55 55 55 55 55 55 55 55 55 55 55 55 55 55 55 55 55 55 55 55 55 55 55
 00000100:   AD EF AD EF AD EF AD EF AD EF AD EF AD EF AD EF AD EF AD EF AD EF AD EF AD EF AD EF AD EF AD EF
```

Ilustración 7. Contenido del Block RAM 0 para el caso de 32kB

```
BRAM data, Column 00, Row 31. Design instance "uP_map/pmem/BU2/U0/blk_mem_generator/valid.cstr/ramloop[4].ram

 00000000:   AA AA AA AA AA AA AA AA AA AA AA AA AA AA AA AA AA AA AA AA AA AA AA AA AA AA AA AA AA AA AA AA
 00000020:   AA AA AA AA AA AA AA AA AA AA AA AA AA AA AA AA AA AA AA AA AA AA AA AA AA AA AA AA AA AA AA AA
 00000040:   DE BE DE BE DE BE DE BE DE BE DE BE DE BE DE BE DE BE DE BE DE BE DE BE DE BE DE BE DE BE DE BE
 00000060:   DE BE DE BE DE BE DE BE DE BE DE BE DE BE DE BE DE BE DE BE DE BE DE BE DE BE DE BE DE BE DE BE
 00000080:   F0 BA F0 BA F0 BA F0 BA F0 BA F0 BA F0 BA F0 BA F0 BA F0 BA F0 BA F0 BA F0 BA F0 BA F0 BA F0 BA
 000000A0:   F0 BA F0 BA F0 BA F0 BA F0 BA F0 BA F0 BA F0 BA F0 BA F0 BA F0 BA F0 BA F0 BA F0 BA F0 BA F0 BA
```

Ilustración 8. Contenido del Block RAM 4 para el caso de 32kB

En las ilustraciones 7 y 8 puede observarse como según el patrón de contenido propuesto, se rellenan las memorias según se había pensado. A falta de una verficación que se expondrá en este capítulo, el archivo de memoria tiene un orden correcto.

Si queremos escribir en una dirección concreta, observando y haciendo pruebas de esta forma, se puede deducir la siguiente propiedad:

- Si el espacio de memoria empieza en la dirección 0x0000, las direcciones del primer bloque están contenidas entre las direcciones 0x0000y 0x1FFF. El bloque continuo, contiene el espacio de memoria comprendido entre 0x2000 y 0x3FFF.

  Si por ejemplo, se se decide direccionar de forma “global”, tal y como lo haría el modelo comercial, y el espacio empieza en 0x4000 hasta 0xFFFF, esto es para el caso de 48kB de memoria ROM, el bloque de memoria formado por las Block RAMs 0 y 6, se direccionaría entre 0x8000 y 0x9FFF. El bloque formado por las Block RAMs 1 y 7, se direccionaría entre 0xA000 y 0xBFFF.

Para verificar esta propiedad, haciendo uso de los archivos generados para el caso de 32kB, se escriben palabras en las direcciones 0xC000 y 0xE000, las cuales deberían de aparecer en los bloques 6 y 2, y 7 y 3. El resultado se presenta a continuación.

En el archivo mem se escribe:

@C000 7711

@E000 8822

En las memorias que componen la instancia el resultado es:

```
BRAM data, Column 02, Row 31. Design instance "uP_map/pmem/BU2/U0/blk_mem_generator/valid.cstr/ramloop[6].ram

00000000:   77 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00000020:   00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00000040:   00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
```

Ilustración 9. Contenido del Block RAM 6 para el caso de 32 kB

```
BRAM data, Column 00, Row 30. Design instance "uP_map/pmem/BU2/U0/blk_mem_generator/valid.cstr/ramloop[2].ram

00000000:   11 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00000020:   00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00000040:   00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00000060:   00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
```

Ilustración 10. Contenido del Block RAM 2 para el caso de 32 kB

```
BRAM data, Column 01, Row 31. Design instance "uP_map/pmem/BU2/U0/blk_mem_generator/valid.cstr/ramloop[7].ram.

00000000:   88 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00000020:   00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00000040:   00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00000060:   00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
```

Ilustración 11. Contenido del Block RAM 7 para el caso de 32 kB

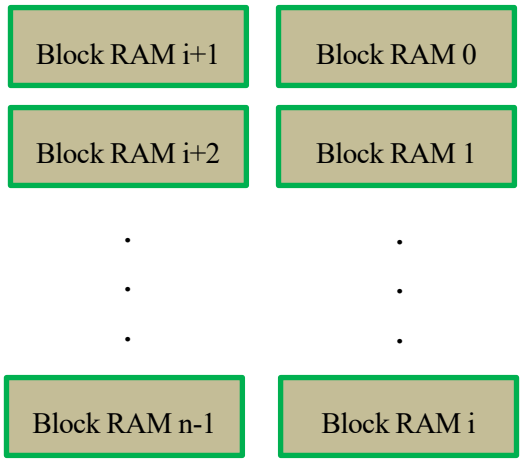
```
BRAM data, Column 01, Row 29. Design instance "uP_map/pmem/BU2/U0/blk_mem_generator/valid.cstr/ramloop[3].ram

00000000:   22 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00000020:   00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00000040:   00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00000060:   00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
--------    -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
```

Ilustración 12. Contenido del Block RAM 3 para el caso de 32 kB

Resumiendo, ya que este apartado es de vital importancia para lo que sigue, se han obtenido los siguientes resultados:

Si se dispone de n[11] bloques de memoria, estas se relacionan en el archivo de memoria -archivo con extension bmm- de la siguiente forma:

[11] n puede estar comprendido entre 2 y 14 por la naturaleza del microcontrolador en cuestión



Para el espacio de direcciones, aunque puede ser local o global, se recomienda el uso de un espacio local ya que simplificará el trabajo. Si se dispone de un espacio de direcciones local, las direcciones para el caso de 12 bloques de memoria, por ejemplo, quedarían de la siguiente forma:

0x0000
0x1FFF

Bloque de memoria 1

0x2000
0x3FFF

Bloque de memoria 2

0x4000
0x5FFF

Bloque de memoria 3

0x6000
0x7FFF

Bloque de memoria 4

0x8000
0x9FFF

Bloque de memoria 5

0xA000
0xBFFF

Bloque de memoria 6

4.6 Generación de un archivo mem a partir de un programa descrito en lenguaje C

Para cargar un programa en memoria, se presentan dos opciones:

- Hacerlo mediante la generación de un fichero binario .elf
- Hacerlo mediante la generación de un fichero .mem generado a partir del fichero .elf

En este proyecto se ha optado por la segunda opción. ¿Por qué? Es cierto que, en princio, debería ser indiferente, pero la herramienta que se usa para cargar el contenido en memoria es **data2mem**, y por razones internas de trabajo de la herramienta, al cargar este archivo corrompe la información, y por tanto el microcontrolador no hará nada. Sin embargo, si se genera un archivo mem a partir de un archivo elf, y se usa la misma herramienta, **data2mem**, se carga el contenido en memoria correctamente, con una puntualización que se comentará en este apartado.

Para generar el archivo mem, se necesita disponer de un sistema linux, en particular se ha usado Centos 7 [10]. En este sistema se instalarán dos herramientas, aunque básicamente se necesita solo una de ellas, la otra herramienta se ha usado con el primer programa para verificar que el programa funcionaba de forma correcta, aunque se puede seguir usando si se desea. Los programas que se necesitan son:

- El compilador libre de Texas Instruments [7] GCC. Este compilador permite generar el archivo elf
- El simulador Questasim [11], que permitirá visualizar los resultados.

Para generar el archivo mem que se pretende cargar en memoria, se hará uso de un repositorio git[12]. En este repositorio, entre otros archivos y directorios, se disponen de un carpeta denominada ***src-c***, en la cual se encuentran los programas que se han probado en el microcontrolador, aunque en este texto solo se presentan el resultado de dos ellas, ***tfg_ports*** y ***tfg_sqr***. Con las herramientas indicadas más arriba instaladas, se creará una copia del repositorio remoto en la máquina que estemos usando. Creada esta copia, los pasos a seguir para generar el archivo elf son:

Se accede a la carpeta donde se encuentran los programas, en C, que se deseen cargar en memoria. Para ello la ruta es:

- `~/openmspftu/openmsp430/core/sim/rtl_sim/src-c/`

Cada uno de las carpetas contiene los siguientes archivos:

- **main.c.** Esta fichero describe unas operaciones en C con el objetivo de que las ejecute el microcontrolador.
- **makefile**. Este fichero contiene una serie de acciones que se deben de ejecutar para generar el archivo .elf. Hace uso de la herramienta ***make*** para definir una serie de operaciones y no se tengan que ejecutar cada vez que se desee generar un nuevo programa.
- **name.v.** En este fichero se establece el tiempo de simulación que se desea, donde {name} es el nombre del programa.

[12] La dirección del repositorio es: /var/git/openmspftu y el servidor se denomina woden.us.es

- **Linker.msp430-elf.x**. Linca los scripts que se necesitan para generar el programa.
- **omsp_system.h**. Define una serie de aspectos hardware para que sean leídos desde el compilador.

Se entra en la carpeta del programa que se desee, es decir bajamos un nivel en la jerarquía de directorios establecida. Por ejemplo, si quiere accede a tfg_ports, se debe de ejecutar:

- ~/openmspftu/openmsp430/core/sim/rtl_sim/src-c/tfg_ports

Dentro de esta carpeta, se hace make:

- ~/openmspftu/openmsp430/core/sim/rtl_sim/src-c/tfg_ports/make

Tras esto, se genera el archivo .elf. ¿Cómo se genera el archivo mem a partir del mem? Para generar el archivo mem se hace uso del ejecutable **ihex2mem.tcl**, el cual tiene como argumento de entrado el archivo elf y genera el archivo mem, -el nombre será pmem.mem.

Para crear un programa nuevo hay que copiar estos archivos en una nueva carpeta dentro del directorio ***src-c***, y modificar lo siguiente:

1. El **main.c**, describiendo dentro lo que se desee.
2. Dentro del fichero **makefile** se debe cambiar el NAME del ejecutable. Por ejemplo, si el ejecutable se denomina tfg_ports, entonces quedará lo siguiente:

```
# makefile configuration
NAME            = tfg_ports
OBJECTS         = main.o
```

3. Modificar el nombre del archivo .v, con el nombre que se le haya dado en el **makefile**. Si se denomina tfg_ports, entonces este archivo se llamará tfg_ports.v

Todas estas funciones han sido copiadas del directorio del openMSP430 de Olvier Girard [12]. En el manual openMSP430 [4] se describe de forma detallada la jerarquía del repositorio y las funciones que se incluyen.

El archivo mem generado incorpora direcciones de memoria cada 16 direcciones, es decir, incrementa la dirección cada 16 palabras de 16 bits. Ya se comentó en el apartado **3.4** los problemas que se podían generar de un direccionamiento local o global de la memoria. Para evitar este hecho, además de problemas que puedan venir por un mal direccionamiento en el archivo mem, se ha modificado la función que genera el archivo mem, para que además de generar este, genera un idéntico pero sin ninguan dirección. ¿Y por qué esto es posible?

Esto puede hacerse por dos razones:

- El archivo mem generado contiene justamente la cantidad de palabras que es capaz de almacenar el microcontrolador que se haya elegido.
- La herramienta, al cargar el contenido en memoria, assume que la primera palabra se corresponde con la dirección 0x0000 y el resto con las que continúan.

En el caso de no existir contenido, rellena con ceros. De una forma más simple, si nuestro diseño tiene un memoria de programa de 48kB, se genera un archivo mem que rellene justamente esos 48kB, si es de otra capacidad ídem. Esto puede observarse en los ejemplos que muestran a continuación para el ejemplo 1 de este proyecto.

```
@0000  007E 0206 0043 0000 0000 0000 0000 0000 FFFF 41AA 0000 FFFF 0000 4031
2A00 403C
@0010  027E 430D 403E 0012 12B0 43D8 403C 0200 403D 4424 9C0D 2404 403E 007E
12B0 439C
@0020  12B0 4408 430C 12B0 4154 12B0 4334 4034 400C 4035 400C 4326 4030 407A
4034 400C
@0030  4035 400C 4326 4030 407A 4034 400C 4035 400C 4036 FFFE 4030 407A 9405
2405 4427
@0040  5604 12A7 4010 FFF4 4130 4130 4130 403C 4424 803C 4423 436D 9C0D 2C07
403D 0000
```

```
@0050  930D 2403 403C 4424 128D 4130 120A 403A 4424 803A 4424 110A 4A0C 12B0
41D2 5A0C
@0060  4C0D 110D 930D 2407 403E 0000 930E 2403 403C 4424 128E 413A 4130 120A
1209 93C2
@0070  027E 2017 403A 4018 803A 4016 110A 533A 4039 4016 421C 0280 9A0C 280D
12B0 408E
@0080  403D 0000 930D 2403 403C 4008 128D 43D2 027E 4030 41CC 531C 4C82 0280
5C0C 590C
@0090  4C2C 128C 4030 40F4 403E 0000 930E 2405 403D 0282 403C 4008 128E 403C
0200 938C
@00a0  0000 2405 403D 0000 930D 2401 128D 12B0 40AC 4130 120A 1209 1208 1207
1206 40B2
@00b0  5A80 0120 43F2 0022 43F2 002A 43F2 001A 434A 4036 42D8 4077 0021 4078
0029 4079
@00c0  0019 4AC7 0000 4A4C 5A4C 4CC8 0000 4A0D 4A0C 1286 4CC9 0000 4A4C 535C
4C4A 907C
@00d0  0064 23EF 434C 4030 41C6 403C 0200 903C 4424 2406 421E 4000 403D 4424
12B0 431E
@00e0  4130 4134 4135 4136 4137 4138 4139 413A 4130 C312 100C C312 100C C312
100C C312
@00f0  100C C312 100C C312 100C C312 100C C312 100C C312 100C C312 100C C312
100C C312
@0100  100C C312 100C C312 100C C312 100C 4130 533D C312 100C 930D 23FB 4130
C312 100D
@0110  100C C312 100D 100C C312 100D 100C C312 100D 100C C312 100D 100C C312
100D 100C
@0120  C312 100D 100C C312 100D 100C C312 100D 100C C312 100D 100C C312 100D
100C C312
@0130  100D 100C C312 100D 100C C312 100D 100C C312 100D 100C 4130 533E C312
100D 100C
.
.
.
@5ff0  0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000
0000 401A
```

Si ahora eliminamos todas las direcciones de memoria, quedará simplemente el contenido de la misma:

```
007E 0206 0043 0000 0000 0000 0000 0000 FFFF 41AA 0000 FFFF 0000 4031 2A00
403C
027E 430D 403E 0012 12B0 43D8 403C 0200 403D 4424 9C0D 2404 403E 007E 12B0
439C
12B0 4408 430C 12B0 4154 12B0 4334 4034 400C 4035 400C 4326 4030 407A 4034
400C
4035 400C 4326 4030 407A 4034 400C 4035 400C 4036 FFFE 4030 407A 9405 2405
4427
5604 12A7 4010 FFF4 4130 4130 4130 403C 4424 803C 4423 436D 9C0D 2C07 403D
0000
930D 2403 403C 4424 128D 4130 120A 403A 4424 803A 4424 110A 4A0C 12B0 41D2
5A0C
4C0D 110D 930D 2407 403E 0000 930E 2403 403C 4424 128E 413A 4130 120A 1209
93C2
027E 2017 403A 4018 803A 4016 110A 533A 4039 4016 421C 0280 9A0C 280D 12B0
408E
403D 0000 930D 2403 403C 4008 128D 43D2 027E 4030 41CC 531C 4C82 0280 5C0C
590C
4C2C 128C 4030 40F4 403E 0000 930E 2405 403D 0282 403C 4008 128E 403C 0200
938C
0000 2405 403D 0000 930D 2401 128D 12B0 40AC 4130 120A 1209 1208 1207 1206
40B2
5A80 0120 43F2 0022 43F2 002A 43F2 001A 434A 4036 42D8 4077 0021 4078 0029
4079
```

```
0019 4AC7 0000 4A4C 5A4C 4CC8 0000 4A0D 4A0C 1286 4CC9 0000 4A4C 535C 4C4A
907C
0064 23EF 434C 4030 41C6 403C 0200 903C 4424 2406 421E 4000 403D 4424 12B0
431E
4130 4134 4135 4136 4137 4138 4139 413A 4130 C312 100C C312 100C C312 100C
C312
100C C312 100C C312 100C C312 100C C312 100C C312 100C C312 100C C312 100C
C312
100C C312 100D 100C C312 100D 100C C312 100D 100C C312 100D 100C C312 100D
100C
C312 100D 100C C312 100D 100C C312 100D 100C C312 100D 100C C312 100D 100C
C312
100D 100C C312 100D 100C C312 100D 100C C312 100D 100C 4130 533E C312 100D
100C
.
.
.
0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000
401A
```

Llegados a este punto recopilemos:

¿De qué se dispone? Se dispone del archivo mem, que contiene el programa que se desea ejecutar en el micro, se dispone del archivo de memoria bmm, y se dispone de un archivo .bit con las memorias de programa en blanco, es decir, sin escribir.

¿Qué se desea? Se desea implementar una aplicación en esas memorias vacías acorde con un archivo de descripción de memorias.

Se dispone de todo lo necesario para generar un archivo .bit con una aplicación cargada, y así poder observar el comportamiento en FTU.

Para cargar este archivo se hará uso, una vez más, de **data2mem**. Para ello hay que ejecutar el siguiente comando:

Ruta donde se ubican los archivos>C:\Xilinx\14.7\ISE_DS\ISE\bin\nt64\data2mem -bm name.bmm -bd aplicacion.mem -bt wr_openmsp430.bit -o b new_name.bit

Para el caso en cuestión, el ejemplo 1, quedaría lo siguiente:

Ruta donde se ubican los archivos>C:\Xilinx\14.7\ISE_DS\ISE\bin\nt64\data2mem -bm memory48k.bmm -bd pmem_ftu.mem -bt wr_openmsp430.bit -o b new_wr_openmsp430.bit

Tras ejecutar estos comandos, se generaría el archivo que sintetiza el diseño **new_wr_openmsp430.bit,** que además de sintetizar el microcontrolador, incorpora la aplicación que se le haya cargado al mismo.

Si se desea ver el contenido que se ha cargado en el nuevo archivo de implementación .bit, denominado **new_wr_openmsp430.bit**, debemos buscar la palabra *ramloop*[ ], y dentro de los corchetes la memoria que se desee mirar. El comando que hay que ejecutar será:

Ruta donde se ubican los archivo>C:\Xilinx\14.7\ISE_DS\ISE\bin\nt64\data2mem -bm memory48k.bmm –bt new_wr_openmsp430.bit -d > new_wr_openmsp430

El archivo **new_wr_openmsp430** contiene, entre otra información, el contenido de las memorias. Por ejemplo, para el mem que se ha expuesto más arriba, se puede observar si la palabra 401A se incluye en las Blocks RAM 5 y 11 como cabe esperar. Si se hace, se observa que efectivamente ocurre eso mismo.

Para la Block RAM 5,

```
BRAM data, Column 00, Row 22. Design instance
"uP_map/pmem/BU2/U0/blk_mem_generator/valid.cstr/ramloop[5]
00000000:   00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00 00 00
.
```

```
.
.
00000FE0:   00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00 00 1A
```

Y para la 11,

```
BRAM data, Column 01, Row 22. Design instance
"uP_map/pmem/BU2/U0/blk_mem_generator/valid.cstr/ramloop[11]
00000000:   00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00 00 00
00000FE0:   00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00 00 40
```

4.7 Generación del archivo coe para el primer arranque

Aunque se dispone de un archivo mem el cual puede cargar el contenido que se desee en las memorias del microcontrolador, y se sabe como se organizan las memorias y en qué orden se rellenan, sería interesante asegurarnos de si este contenido se carga como se piensa en las memorias, ya que si hubiese errores se dispondría de más herramientas y recursos para encontrar el mismo.

Este paso del flujo de trabajo **no es necesario cada vez que se desee implementar un** programa. Se ha llevado a cabo durante la primera implementación debido a errores aparecidos en FTU2, los cuales no se podían depurar con dicha herramienta. Se insiste, **no hay que ejecutar este paso, se expone este apartado a título informativo,** por si fuese útil para posteriores trabajos.

¿Por qué es interesante el archivo .coe y qué nos permite?

Un archivo coe nos permite inicializar la memoria de programa y observar en una simulación, en este caso en ISim, el comportamiento del circuito para su posterior comparación con el comportamiento en FTU2.

De la sintaxis de un archivo coe, aunque se expone de forma detallada en el manual de usuario para la generación de memorias mediante el uso de CORE Generator [13], se expone aquí lo usado en este proyecto:

- En la primera línea se indica la raíz de inicialización de la memoria, es decir la base en la que se escriben los datos, que en este caso es hexadecimal.

```
memory_initialization_radix=16;
```

- En la segunda línea se indica el vector de inicialización de la memoria, es decir, el contenido de la misma. La expresión usada para ello es:

```
memory_initialization_vector=
```

La dificultad de su generación viene dada porque hay que pasar de un archivo mem, con la forma que se ha visto, a un archivo coe cuya forma se muestra a continuación.

```
memory_initialization_radix=16;
memory_initialization_vector=
007E
0206
0043
0000
0000
0000
0000
0000
FFFF
```

```
4188
0000
FFFF
0000
4031
2A00
403C
.
.
.
401A
;
```

El número de palabras del archivo coe debe ser igual al número de posiciones de memoria que contenga la misma. Si el contenido de las memorias es "pequeño" y el tamaño de las mismas también, es viable realizar el archico coe de forma manual. Si no es así, hay que hacer el proceso de forma automática con un editor de texto.

¿Cómo se carga este archivo en memoria?

Para cargar el contenido de un archivo coe en memoria y así inicializarla, se abre en ISE la memoria de programa, que es donde se desea cargar el contenido. Abierta esta, en la página 3 de 5, se selecciona la opción **Load Init File**, y se carga el archico coe. A continuación, se muestra el archivo cargado para este trabajo.

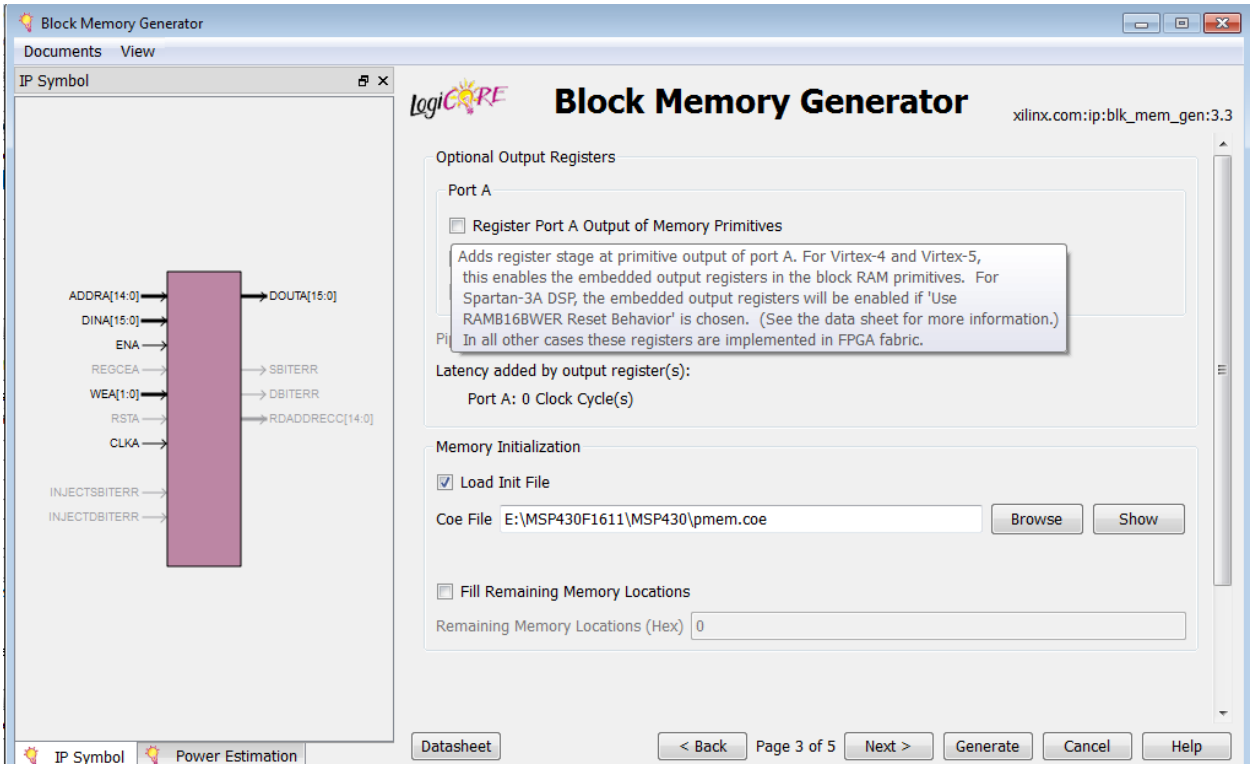


Ilustración 13. Ventana de diálogo para cargar archivo coe

Cargado el contenido en memoria, hay que llevar a cabo el proceso de síntesis y la generación de los archivos de configuración de la FPGA.

Podría surgir una cuestión importante de responder en este punto. Y es que si se dispone del archivo coe y se inicializan las memorias podría pensarse que siempre que deseemos cargar un programa en memoria podría hacerse así. Y la respuesta es afirmativa, y además esto no generará ningún problema para la posterior implementación en FTU2. Entonces, ¿por qué el intereés de cargar el contenido en memoria mediante la herramienta **data2mem** y generar un microcontrolador vacío? Hay varios argumentos que justifican y responden a esto:

1. Cada vez que se lleva a cabo el proceso de síntesis y la posterior generación de los archivos, la ocupación dentro de la arquitectura interna de la FPGA cambia. Es decir, aunque se implemente el mismo circuito, este se hace usando recursos diferentes de los que se usaron en la implementación anterior. Si cada vez que se carga contenido en memoria, se cambia la distribución de los elementos que forman el circuito, dejan de tener sentido las campañas de inyección de fallos, ya que el modelo tendrá sus elementos críticos en distinta ubicación en cada implementación.

   Se necesita un modelo de circuito que describa siempre las mismas conexiones, y así poder testar qué elementos o conexiones son críticas y donde se ubican para actuar en consecuencia.

2. Pasar por el proceso de síntesis y generación de archivos cada vez que se carga contenido en memoria implica un gran uso de un recurso fundamental, el tiempo. Llevar a cabo estos procesos implican gran cantidad de tiempo en comparación con el tiempo que se tarda con la herramienta **data2mem** en cargar contenido en memoria.

La siguiente cuestión que hay que responder para concluir este apartado es dónde y cómo miro el contenido de la memoria. De nuevo, la respuesta está en el uso de la herramienta **FPGA Editor**. Terminado el proceso de síntesis y generados los archivos, se abre **FPGA Editor**. Seleccionamos **File> Main Properties**, y dentro de esta la opción **Read Write** y se aplica dicho cambio.

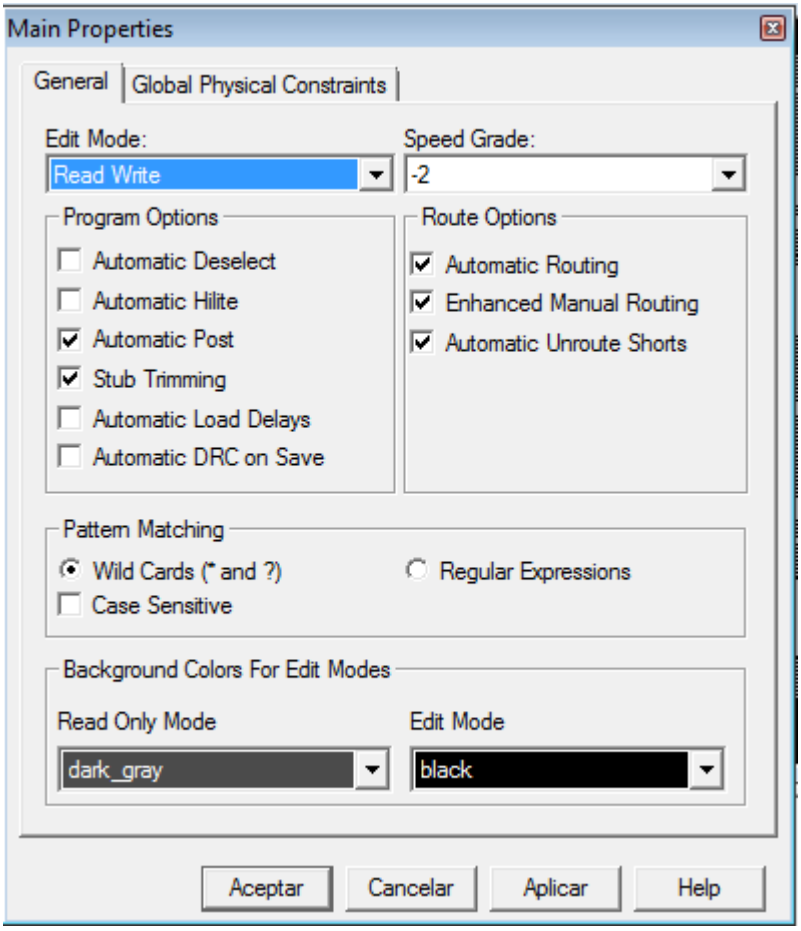


Ilustración 14. Cambio de las opciones principales en FPGA Editor

Ahora se buscan los bloques de memoria como se explica en el apartado **3.3**.

Clicando dos veces sobre el bloque de memoria deseado -hay que mirar el contenido de todos- se abre la siguiente ventana:

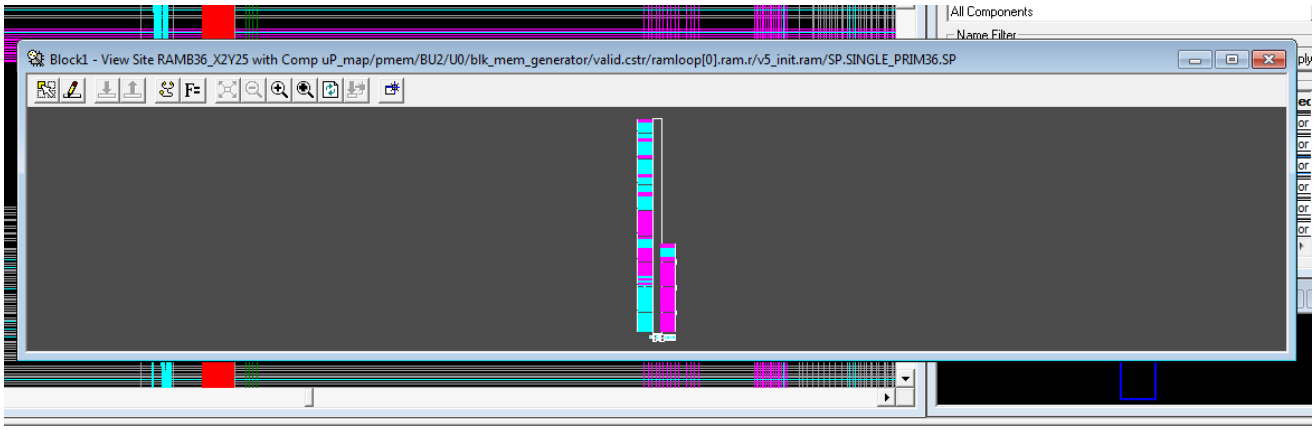


Ilustración 15. Bloque de memoria físico

Se clica sobre la opción **Begin Editing** y tras esta **Show/Hide Attributes**. Tras esto, aparece el contenido de la memoria.

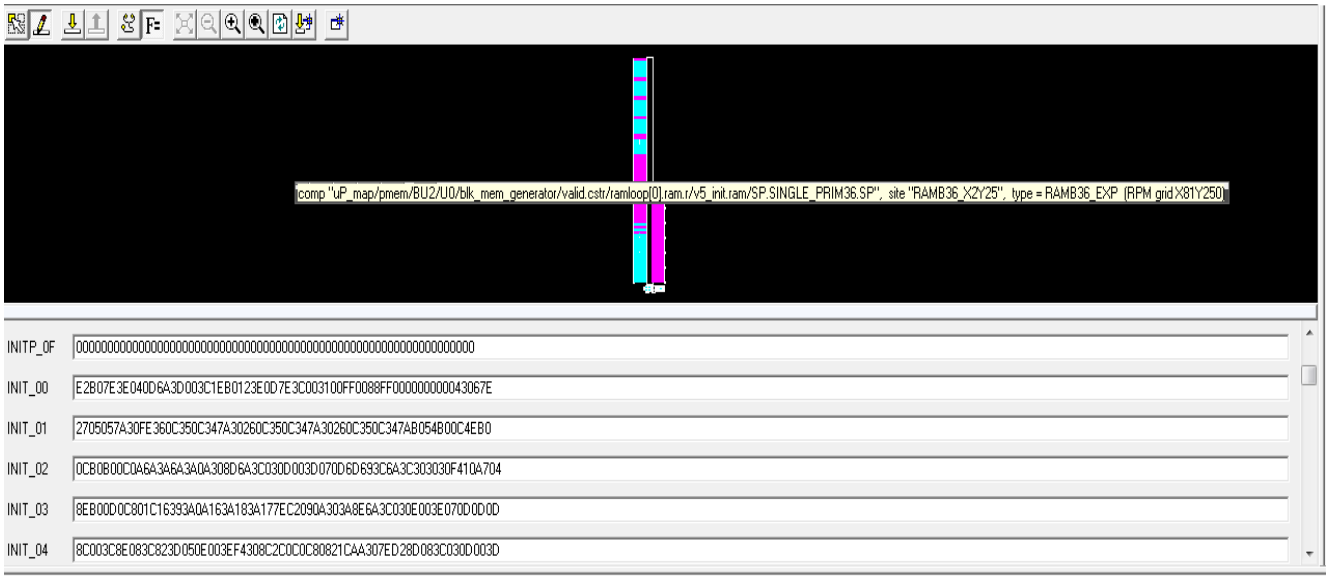


Ilustración 16. Contenido memoria según archivo coe

Para comparar el contenido de las memorias obtenido de esta forma, con el contenido que aparece cuando ejecutamos los comandos indicados en el apartado anterior, hay que hacer los siguientes cambios.

Para la dirección de memoria 00, o *INIT_00* que es como la llama **FPGA Editor:**

1. La dirección contiene 256 bits, es decir 32 palabras de 8 bits cada una. Se divide en 2 líneas de 128 bits cada una, -16 palabras de 8 bits cada línea, que es lo que cabe en cada dirección de memoria, ya que recordemos que aunque las palabras son de 16 bits, en una dirección de memoria uns Block RAM se puede escribir una palabra de 1 byte.
2. Separaramos en dos líneas de la misma longitud esa serie de caracteres, sin alterar el orden.

Tras estos pasos, para le dirección *INIT_00,* por ejemplo, queda lo siguiente:

```
ramloop[0]
INIT_00    3C 00 31 00 FF 00 88 FF 00 00 00 00 00 43 06 7E
     10    E2 B0 7E 3E 04 0D 6A 3D 00 3C 1E B0 12 3E 0D 7E
```

Si se hace esto mismo con el bloque de memoria 6 se obtiene:

```
ramloop[6]

INIT_00     40 2A 40 00 FF 00 41 FF 00 00 00 00 00 00 02 00
     10     42 12 00 40 24 9C 43 40 02 40 43 12 00 43 43 02
```

Comparando el contenido de estas direcciones con el contenido que se obtiene de ejecutar los commandos indicados en el apartado anterior, se puede observar que contienen las mismas palabras hexadecimales.

```
BRAM data, Column 01, Row 24. Design instance "uP_map/pmem/BU2/U0/blk_mem_generator/valid.cstr/ramloop[0].ram.

00000000:   7E 06 43 00 00 00 00 00 FF AA 00 FF 00 31 00 3C 7E 0D 3E 12 B0 D8 3C 00 3D 24 0D 04 3E 7E B0 9C
```

Ilustración 17. Contenido Block RAM 0 cargado mediante **data2mem**

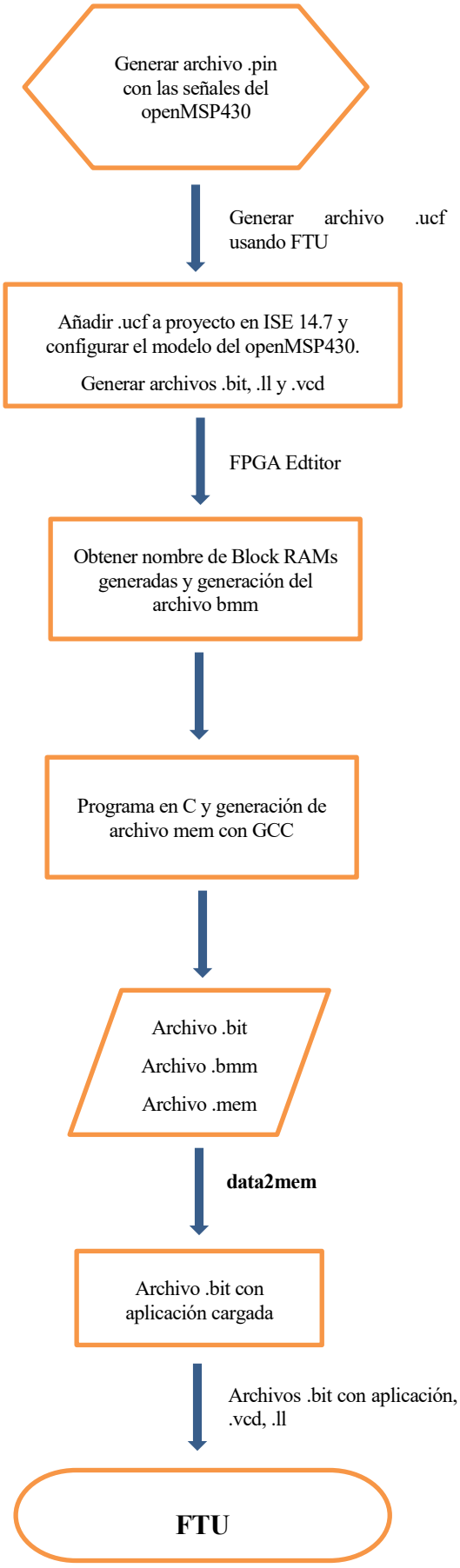
```
BRAM data, Column 02, Row 23. Design instance "uP_map/pmem/BU2/U0/blk_mem_generator/valid.cstr/ramloop[6].ram.

00000000:   00 02 00 00 00 00 00 00 FF 41 00 FF 00 40 2A 40 02 43 40 00 12 43 40 02 40 44 9C 24 40 00 12 43
```

Ilustración 18. Contenido Block RAM 6 cargado mediante **data2mem**

Esta prueba permite verificar que el orden establecido de las memorias es el correcto.

4.8 Flujo de diseño para la implementación en FTU2

En este apartado se muestra de forma gráfica el flujo de diseño para la implementación del openMSP430 en FTU.


Generar archivo .pin con las señales del openMSP430
Generar archivo .ucf usando FTU
Añadir .ucf a proyecto en ISE 14.7 y configurar el modelo del openMSP430.
Generar archivos .bit, .ll y .vcd
FPGA Edtitor
Obtener nombre de Block RAMs generadas y generación del archivo bmm
Programa en C y generación de archivo mem con GCC
Archivo .bit
Archivo .bmm
Archivo .mem
data2mem
Archivo .bit con aplicación cargada
Archivos .bit con aplicación, .vcd, .ll
FTU

5 RESULTADOS

El fracaso es una gran oportunidad para empezar de nuevo con más inteligencia.

Henry Ford

En este capítulo se expondrán los resultados obtenidos en ISim y en FTU2. En el primer ejemplo, se puede comparar el comportamiento mostrado en ISim y el obtenido en FTU2, y así verificar que el sistema funciona correctamente. En el segundo, ya que se sabe como generar el archivo mem y que la organización de la memoria es correcta, solo se ha optado por la simulación y verificación del comportamiento en FTU2.

5.1 Resultados obtenidos para el ejemplo 1

El programa **tfg_ports** descrito en C, o ejemplo 1 como se denomina en el título de este apartado, contiene un bucle que debe de repetir 100 veces el microcontrolador, y en cada iteración, saca por el puerto 1 el número de la interacción por la que va el bucle, y por el puerto 2, el doble de ese número. A continuación se presenta el programa.

```
#include "omsp_system.h"

int main(void) {

    WDTCTL = WDTPW | WDTHOLD;           // Disable watchdog timer

    P1DIR  = 0xff;                      // Port 1.0-1.7 = output
    P2DIR  = 0xff;                      // Port 2.0-2.7 = output
    //P3DIR  = 0xff;                      // Port 3.0-3.7 = output

    int i;

    for (i = 0; i < 100; i++) {
        P1OUT = i;
```

```
            P2OUT = 2*i;
            //P3OUT = i*i;
        }

        return 0;
}
```

Si observamos la simulación en ISim para verifcar que el sistema funciona correctamente, se obtiene el siguiente resultado. Las variables que se muestran son el contrador de programa,-**pc-**, el reloj, la señal de reset, y las señales de salida, **p1_pout_ext** y **p2_pout_ext**, que sacan los datos de P1OUT y P2OUT.

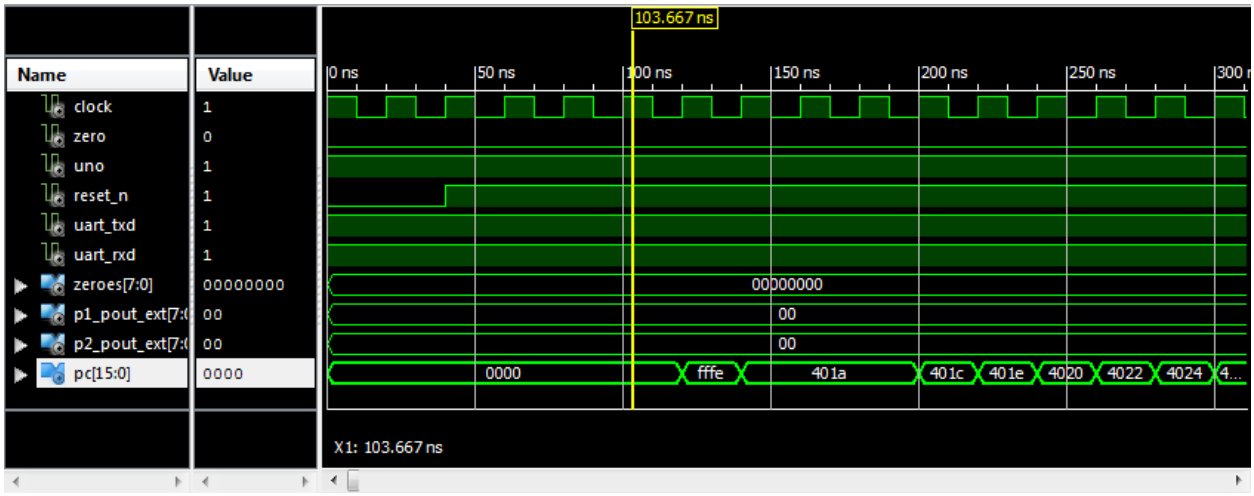


Ilustración 19. Inicio de programa para el openMSP430 48k10k

Se observa como el sistema tras la señal de reset, la cual tiene una duración de 40 ns como se indica en el test bench, el contador de programa se va a la dirección 0xFFFE, tal y como indica el manual del openMSP430.

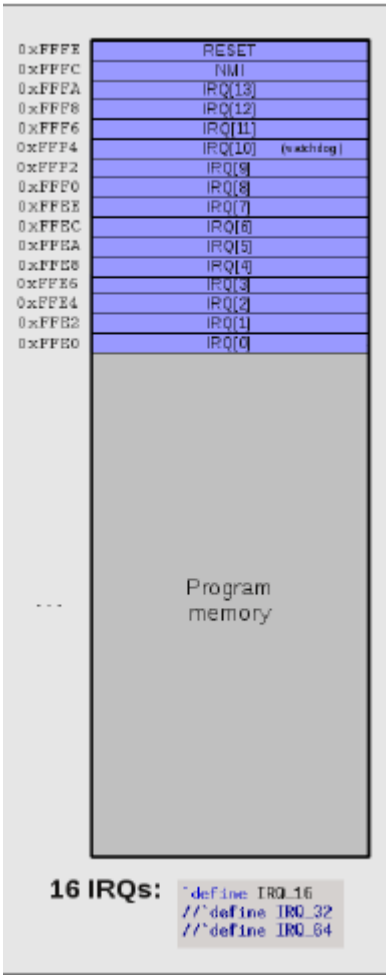


Ilustración 20. Direcciones de memoria según interrupción. Fuente [4]

Si se presta atención a la palabra escricta en la última dirección del archivo mem, esta resulta ser la palabra 0x401A, justo a donde apunta el contador de programa tras apuntar a la dirección 0xFFFE.

Ahora se muestra donde aparece la primera salida del programa que se ha creado. Se manifiesta que pasan bastantes ciclos de reloj hasta que aparece la primera salida, durante lo cuales el microcontrolador lleva a acabo operaciones internas de benificio propio.

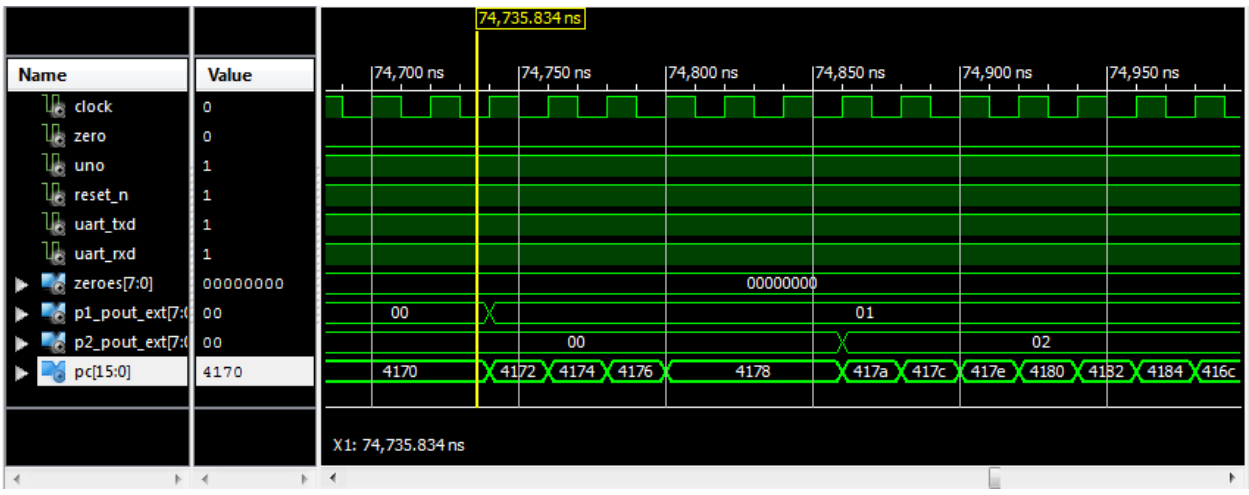


Ilustración 21. Salida del primer número del programa tfg_ports

Si se observan algunas etapas más para verifivar que el comportaminento es correcto, se obtienen resultados satisfactorios.

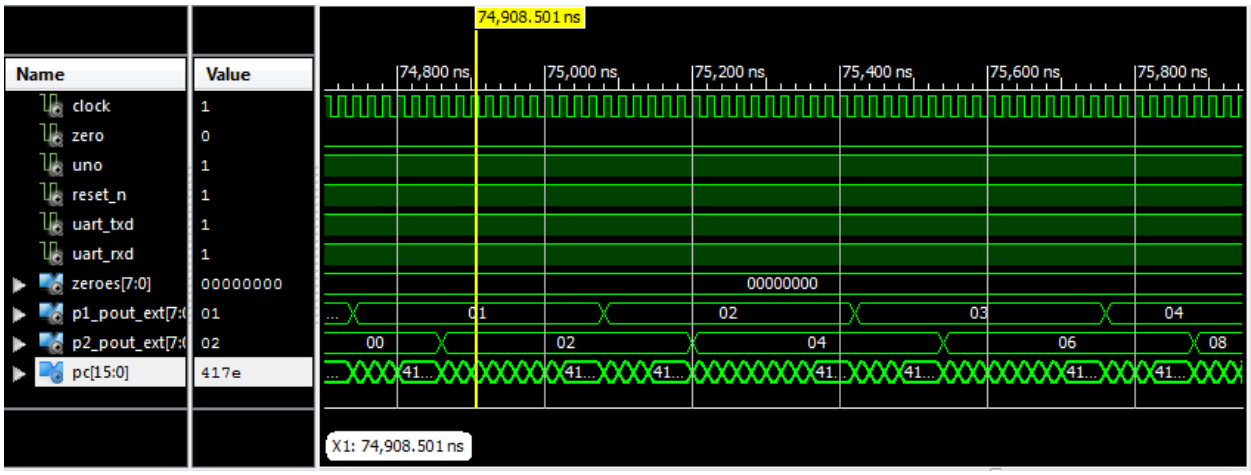


Ilustración 22. Evolución de las salidas según el programa tfg_ports

Veamos que ocurre en FTU si se crea un proyecto en el que se incluye el archivo .bit escrito mediante el proceso de síntesis y generación de archivos que lleva a cabo ISE. El comportamiento esperado debería ser idéntico.

Por utilidad, solo se representarán las señales **pc** (program counter), **p1_pout_ext**, **p2_pout_ext**.

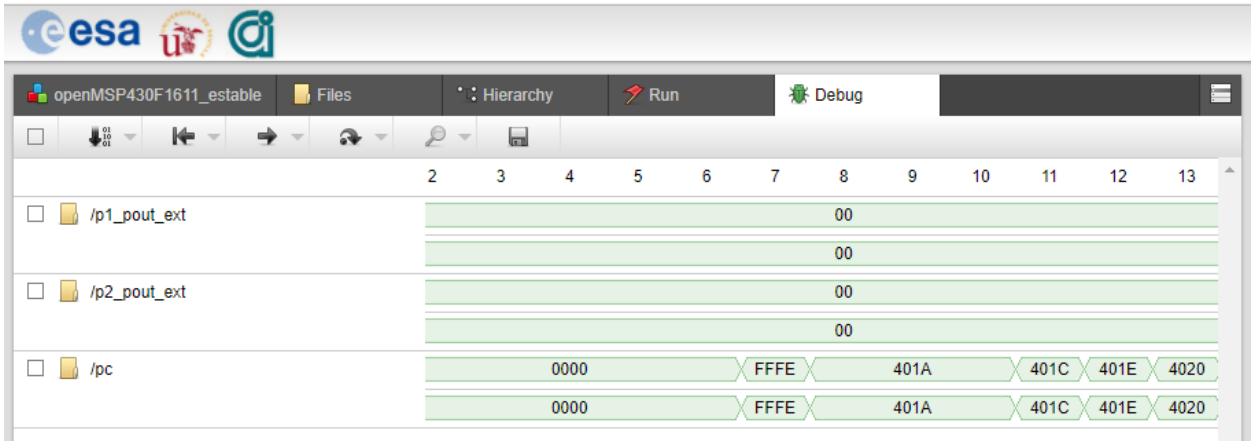


Ilustración 23. Etapas iniciales funcionando en FTU

Se ve como el comportamiento en las primeras etapas es idéntico al mostrado en simulación en ISim.

Si se busca dónde se observa la primera salida en las señales **p1_pout_ext** y **p2_pout_ext**, esto ocurre cuando pasan aproximadamente 3737 ciclos de reloj.

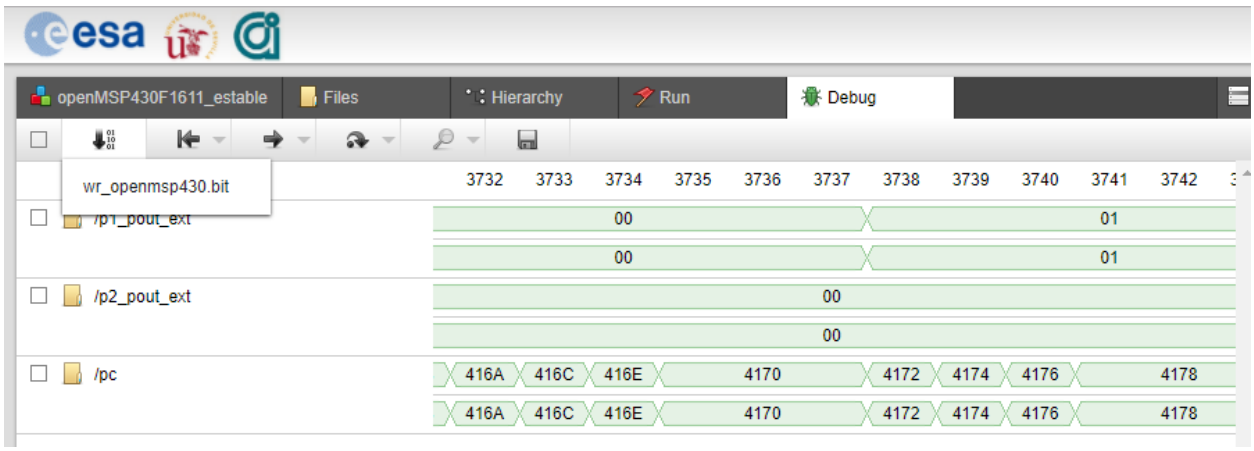


Ilustración 24. Primera salida observada en p1_pout_ext en FTU2

De nuevo, para verificar que el contenido es correcto, se observan algunas etapas más, obteniendo esta vez también resultados satisfactorios.

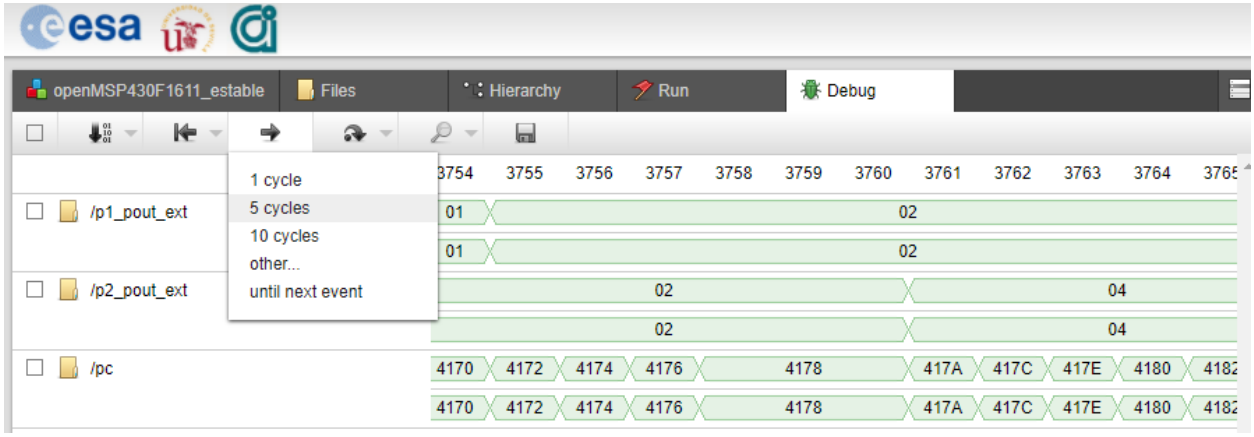


Ilustración 25. Salidas de las señales p1_pout_ext y p2_pout_ext

Ya queda que plasmar el resultado que se observa en FTU cuando el contenido se carga a través del archivo mem, con un archivo de configuración de la FPGA cuyas memorias de programa están vacías, y un archivo de distribución e interrelación de memorias.

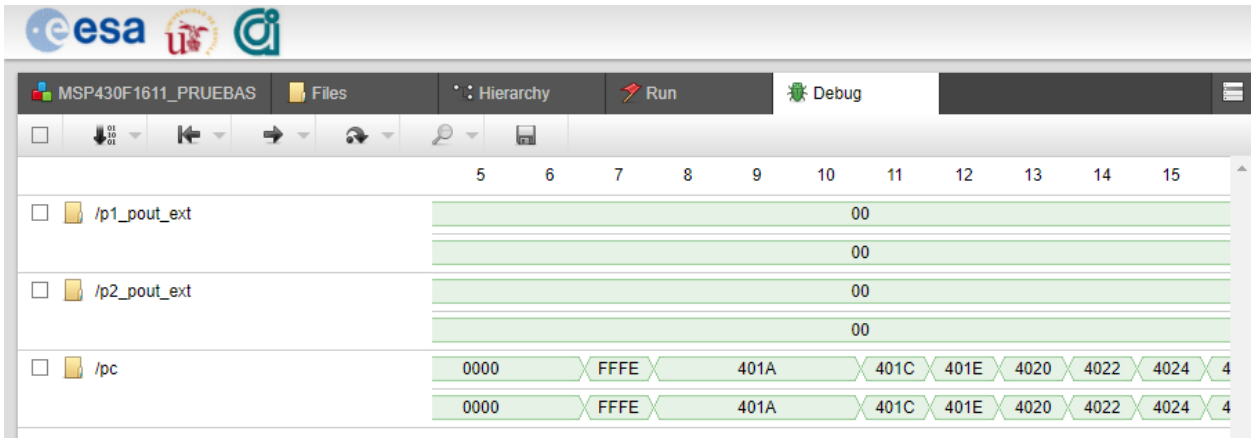


Ilustración 26. Etapas iniciales en FTU con contenido cargado mediante **data2mem**

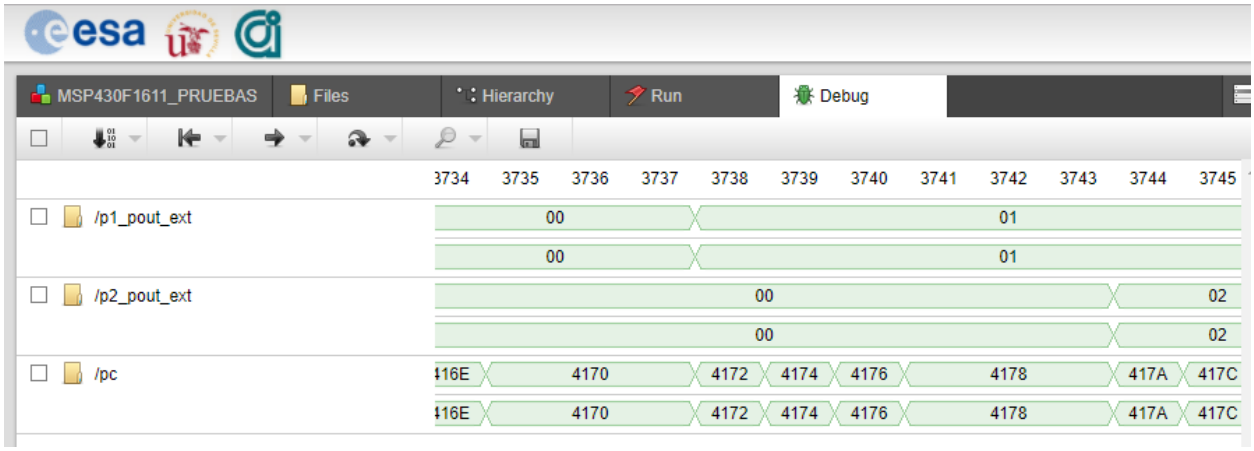


Ilustración 27. Primera salida de p1_pout_ext y p2_pout_ext en FTU con contenido cargado mediante **data2mem**

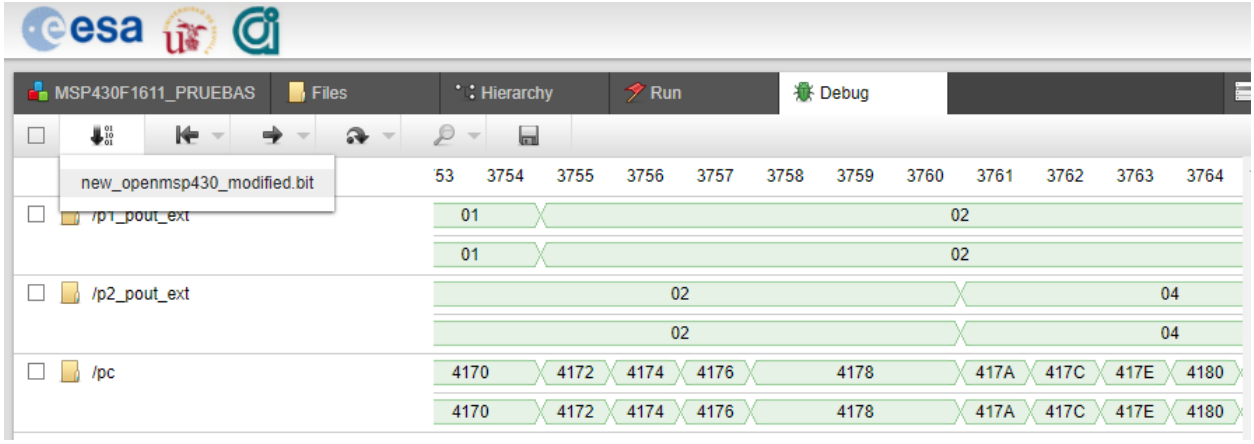


Ilustración 28. Salidas p1_pout_ext y p2_pout_ext en FTU con contenido cargado mediante **data2mem**

Se puede observar que el comportamiento es idéntico a los dos casos anteriores.

5.2 Resultados obtenidos para el ejemplo 2

El ejemplo 2, denominado en el repositorio **tfg_sqr**, está formado por un bucle de cien iteraciones, en el cual en cada iteración debe de sacar por el puerto 1 el número de la iteración que se está ejecutando en el bucle, y por el puerto 2 el cuadrado de esta iteración. Es decir, si el bucle está en la iteración 2, el puerto 1 debe mostrar el valor 2, y el puerto 2 el valor 4. El programa, descrito en C, se presenta más abajo.

```
#include "omsp_system.h"

int main(void) {

    WDTCTL = WDTPW | WDTHOLD;           // Disable watchdog timer

    P1DIR  = 0xff;                      // Port 1.0-1.7 = output
    P2DIR  = 0xff;                      // Port 2.0-2.7 = output


    int i;

    for (i = 1; i <= 100; i++) {
        P1OUT = i;
        P2OUT = i*i;

    }

    return 0;
}
```

Este ejemplo se ha probado directamente en FTU, ya que según y acorde a todo lo desarrollado, se dispone de todo lo necesario para que cualquier programa, siempre que no supere la capacidad de memoria del microcontrolador, funcione directamente en FTU.

A continuación, el funcionamiento del mismo en FTU.

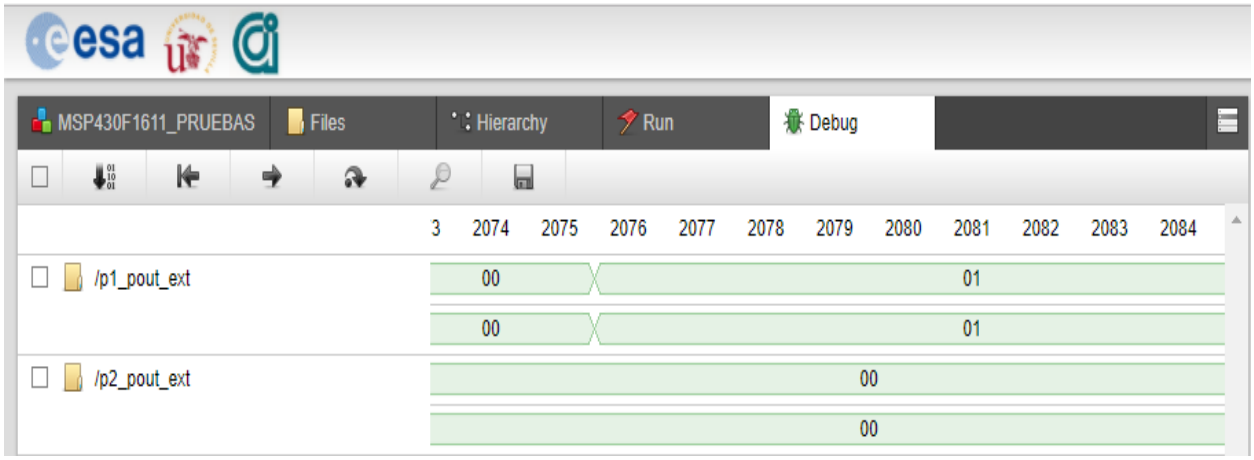


Ilustración 29. Primera salida para el puerto **p1_pout_ext** para el programa **tfg_sqr**

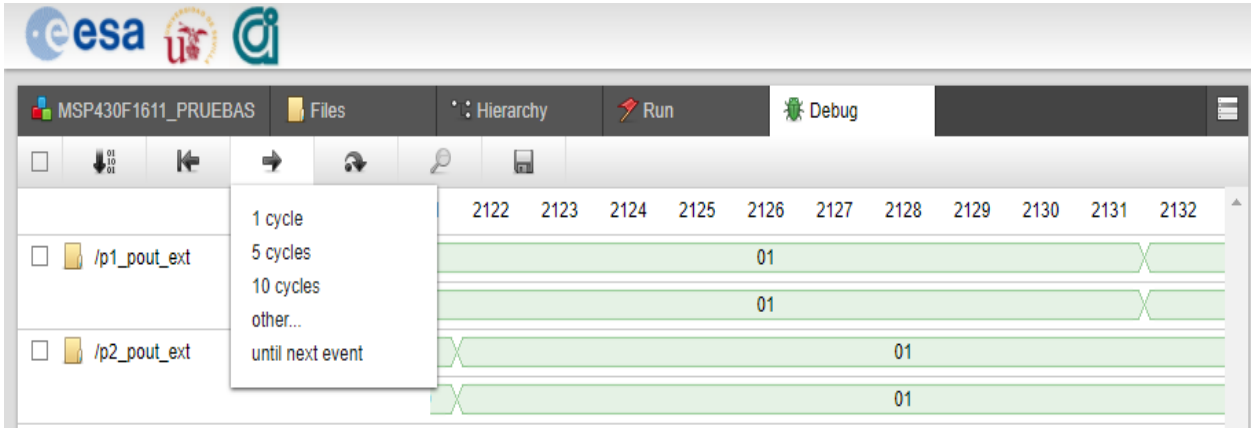


Ilustración 30. Primera salida para el puerto **p2_pout_ext** para el programa **tfg_sqr**

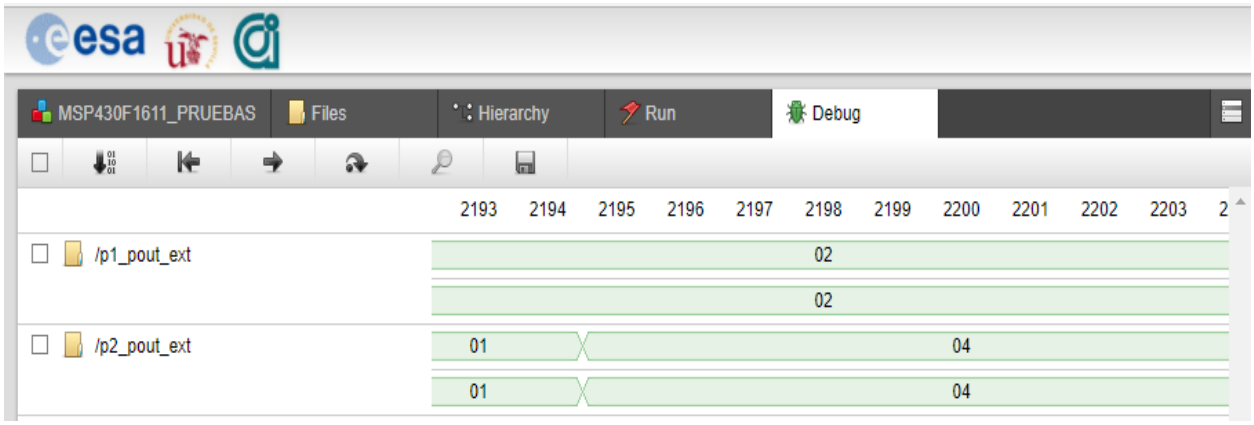


Ilustración 31. Salidas **p1_pout_ext** y **p2_pout_ext** para el programa **tfg_sqr**

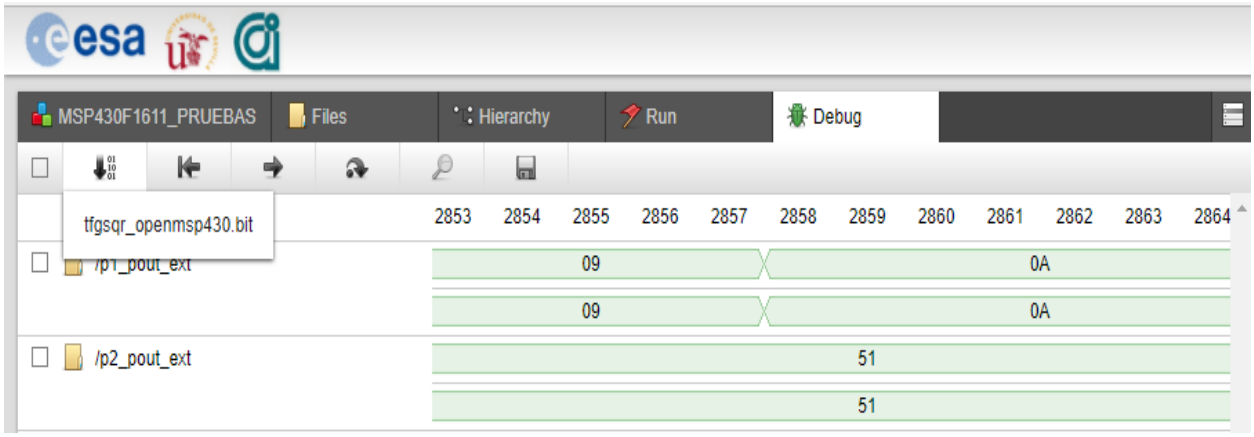


Ilustración 32. Salidas **p1_pout_ext** y **p2_pout_ext** para instante aleatorio

De las figuras expuestas, se observa como el programa se comporta de forma adecuada, sacando por el puerto 2, **p2_pout_ext**, el cuadrado de la salida del puerto 1, **p1_pout_ext**.

Se puede concluir que la metodología desarrollada permite la implementación de esta, y en general, cualquier versión de la familia de microporcesadores openMSP430 en el sistema FTU, lo que permite testar este dispositivo mediante campañas de inyección de fallos y observar su comportamiento.

6 CONCLUSIONES Y TRABAJOS FUTUROS

En verdad no puedes crecer y desarrollarte si sabes las respuestas antes que las preguntas.

Wayne Dyer

En el trabajo desarrollado, se ha establecido una metodología sistemática para implementar distintas versiones del microcontrolador sintetizable openMSP430 en la FPGA Virtex XC5VLX70T. El interés de usar ese modelo concreto de microcontrolador y que el mismo sea sintetizable viene justificado por varias razones.

La primera de estas razones es por la versatilidad, las prestaciones y la respuesta ante ambientes radiactivos de las FPGAs. En concreto, el interés de usar una FPGA SRAM viene motivado porque se puede configurar en cualquier momento. Sus respuestas ante fenómenos de radiación son mejores que dispositivos electrónicos constituidos de distinta forma, como puede ser un microcontrolador comercial. Pero como se ha dicho, aunque esta es una razón de peso, lo que realmente eleva el interés por estos dispositivos es la posibilidad de reconfigurarla incluso durante una misión. Esto es realmente importante para el ámbito espacial. Si se dispone de una FPGA en cuyo interior está descrito un circuito que desempeña una función determinada, y por distintos factores, ya sean vientos solares, radiación cósmica o un problema de índole distinta, se degradan las funciones porque el diseño ha sufrido daños o se ha desconfigurado, se dispone de la posibilidad de reconfigurar el dispositivo usando para ello los recursos que no estén dañados. Esto supone un enorme avance en las tareas de mantenimiento y control de los dispositivos espaciales.

La segunda de estas razones, se encuentra en el bajo consumo de este microcontrolador. Son bien conocidos los problemas energéticos en el espacio. La obtención de energía en el espacio es difícil, lo que exige un aprovechamiento muy alto de la misma. La version comercial está especialmente indicada para ello. El hecho de disponer de una versión sintetizable con un comportamiento muy parecido al de dicho microcontrolador, reúne las condiciones ventajosas de uno y otro sistema, lo cual evidencia aún más la utilidad de lo desarrollado.

El interés de testar un dispositivo en un sistema de inyección de fallos, como es FTU2, es una razón de peso para el desarrollo del trabajo realizada. El acceso a este sistema reduce los costos de desarrollo y el uso de recursos a la hora de diseñar un dispositivo que vaya a funcionar en ambientes radiactivos, ya sean espaciales u otro tipo. Como se ha constatado en el desarrollo de este trabajo, cualquier proyecto espacial financiado con dinero público en Europa o Estados Unidos, debe de pasar unas estrictas pruebas de radiación. Disponer de un sistema de emulación de fallos no implica que no se tengan que pasar dichas pruebas, pero sí nos permite testar nuestro diseño, y así detectar los puntos críticos o elementos más débiles del mismo. Esto ofrece la posibilidad del diseño

de distintas estrategias para reforzar el circuito en cuestión, pudiendo ser estas las tradicionales técnicas de redundancia, o, y es aquí donde aparece el enorme potencial de FTU2, aplicar técnicas que generen software robusto, con tolerancia a errores.

Con todos estos intereses expuestos, el desarrollo de este trabajo ha logrado tres objetivos a destacar:

1. Se dispone de una metodología probada para generar los archivos bmm que exponen la relación entre los bloques de memoria, que parte de las palabras guarda cada uno, dónde las guarda y cuál es la extensión de la memoria para cualquier versión del microcontrolador sintetizable openMSP430. Este objetivo es de extrema importancia para poder implementar un diseño "vacío" y escribir en memoria lo que se desee.
2. Se dispone de una forma de generar cualquier programa que quepa en memoria de programa, memoria ROM, y tener certeza de que este funciona.
3. Por último, se dispone de una herramienta que a partir del archivo de configuración bit "en blanco", del archivo de configuración de bloques de memorias y del archivo que contiene la aplicación que se desea ejecutar, genera un archivo que implementa ese comportamiento en la FPGA Virtex XC5VFX70T.

Como líneas futuras y de continuación de este proyecto se propone:

- Conocido el comportamiento del openMSP430, la programación de campañas de inyección de fallos en FTU, ya sea en memoria o regitros del microcontrolador, estudiar su comportamiento y detectar los elementos críticos de cada diseño.
- Desarrollo de estrategias software para la producción de código robusto tolerante a fallos. Conocido el comportamiento de esta familia de microcontroladores, se pueden desarrollar estrategias software que sean capaces de funcionar correctamente a pesar de fallos durante la operación.
- Generación de nuevos programas para probarlos en este modelo, y otros de la familia openMSP430, así como la modificación aplicando estrategias de tolerancia a errores.
- Implementación de diferentes versiones del openMSP430, y verificación de todo lo expuesto en este trabajo. Posterior campaña de inyección de errores y pruebas software.
- Estudio de la diferencia de comportamiento entre el MSP430F1611 y el openMSP430 desarrollado aquí. Esto permitirá conocer la diferencia de comportamiento entre el microcontrolador comercial y el sintetizable.

REFERENCIAS

[1] O. Thorheim. [En línea]. Available: http://www.datarespons.com/electronics-in-space/.

[2] P. Resources. [En línea]. Available: https://www.google.es/search?q=ceres+planetary+resources&source=lnms&tbm=isch&sa=X&ved=0ahUKEwiFiZqvjczUAhWLblAKHbX3BgkQ_AUIDCgD&biw=1366&bih=662#imgrc=xsPshU7WrkjL2M:.

[3] MarkWelsh. [En línea]. Available: http://www.ti.com/ep-mcu-msp-mspkick-mcufb-thinkinn-en.

[4] O. Girard. [En línea]. Available: http://opencores.org/websvn,filedetails?repname=openmsp430&path=%2Fopenmsp430%2Ftrunk%2Fdoc%2FopenMSP430.pdf.

[5] m. Google Imágenes. [En línea]. Available: https://www.google.es/search?q=msp430f1611&source=lnms&tbm=isch&sa=X&ved=0ahUKEwim6rCtpfrUAhURYlAKHdf4CV4Q_AUICygC&biw=1366&bih=613#imgrc=qhl7TTytlPN4CM:.

[6] E. d. d. d. compilador, «Texas Instruments,» [En línea]. Available: http://www.ti.com/tool/msp430-gcc-opensource.

[7] T. Instruments. [En línea]. Available: https://www.ti.com/lit/ds/symlink/msp430f1611.pdf.

[8] M. Texas Instruments. [En línea]. Available: http://www.ti.com/product/MSP430F1611.

[9] Xilinx. [En línea]. Available: https://www.google.es/search?q=virtex+5&source=lnms&tbm=isch&sa=X&ved=0ahUKEwijms75qM_UAhUPYlAKHeVPBKEQ_AUICigB&biw=1366&bih=662#imgrc=L4eLsvnpQ4kfVM:.

[10] C. Linux. [En línea]. Available: https://www.centos.org/download/.

[11] Mentor. [En línea]. Available: https://www.mentor.com/products/fv/questa/.

[12] O. P. b. O. Girard. [En línea]. Available: https://opencores.org/project,openmsp430,file%20and%20directory%20description.

[13] X. Core Generator. [En línea]. Available:

https://www.xilinx.com/itp/xilinx10/isehelp/cgn_r_coe_file_syntax.htm.

[14] Imagen. [En línea]. Available: https://www.google.es/search?q=virtex+5&source=lnms&tbm=isch&sa=X&ved=0ahUKEwiw2ITQ5c7UAhUINhoKHXQxAYAQ_AUICigB&biw=1366&bih=662#imgrc=L4eLsvnpQ4kfVM:.

[15] I. p. e. Google. [En línea]. Available: https://www.google.es/search?q=virtex+5&source=lnms&tbm=isch&sa=X&ved=0ahUKEwj0oO296M7UAhXJtRoKHQvgBoYQ_AUICigB&biw=1366&bih=662#imgrc=L4eLsvnpQ4kfVM:.

[16] I. d. Google. [En línea]. Available: https://www.google.es/search?q=fpga+virtex+5&source=lnms&tbm=isch&sa=X&ved=0ahUKEwiWtaqGpfrUAhXCa1AKHcuyBtkQ_AUICigB&biw=1366&bih=613#imgrc=L4eLsvnpQ4kfVM:.