

INSTITUTO TECNOLÓGICO DE BUENOS AIRES

22.15 - ELECTRÓNICA V

TRABAJO PRÁCTICO 2

Microprocesador EV19 RISC-V

Alumnos:

Maximiliano ZITELLI 57213

Diego JUAREZ 57295

Tobías LIFSCHITZ 57330

Santiago IVULICH 57421

Sebastian MILHAS 55198

Marcos BRITO 56389

Ignacio DOMINGUEZ 56442

Profesores:

Andrés RODRIGUEZ

Pablo WUNDES

2 de julio de 2019

Resumen

Se diseñó e implementó la arquitectura de un microprocesador compatible con el set de instrucciones de código abierto RISC-V, incluyendo su versión estándar y la extensión de multiplicación y división entera. La microarquitectura esta dividida en un pipeline clásico de cinco etapas y cuenta con un predictor de saltos dinámico de dos bits, el cual brindó una notoria mejora en el tiempo de ejecución. Además se configuró el entorno de desarrollo para poder compilar código de C y C++, y se desarrolló una capa de abstracción de hardware para facilitar la escritura de código de prueba. Luego de depurar los errores de diseño por medio de simulaciones y pruebas reales, el resultado final ejecuta código de alto nivel de manera confiable y transparente al usuario, desde un entorno de desarrollo gráfico.

Índice

1. Introducción	2
2. Diseño	2
2.1. Micro-arquitectura	2
2.1.1. Fetch	3
2.1.2. Decode	3
2.1.3. Execute	4
2.1.4. Memory	5
2.1.5. Write-Back	5
2.1.6. Hazard and Forwarding Unit	5
2.1.7. Predictor de saltos	6
2.2. Periféricos	6
2.2.1. Salida de video VGA	7
2.2.2. Controlador SDRAM	8
2.2.3. Otros periféricos	8
2.3. Software	9
2.3.1. Simulación	9
2.3.2. Entorno de desarrollo	9
2.3.3. Capa de abstracción de hardware	9
3. Resultados	9
4. Trabajo futuro	10
5. Conclusión	11

1. Introducción

En el presente informe se plasma el diseño y desarrollo de un microprocesador *softcore* implementado en FPGA para la asignatura de Electrónica V del Instituto Tecnológico de Buenos Aires. Como punto de partida se decidió no implementar un set de instrucciones (ISA) no estándar dadas las limitaciones y complicaciones que esto supone, tanto en el nivel de micro-arquitectura (su implementación depende en gran parte de la ISA) como en el de lenguaje ensamblador y alto nivel (no se dispone de herramientas de desarrollo estándar). Por estas razones se investigó sobre ISAs de código abierto, y de tipo RISC (Reduced Instruction Set Computer). De las opciones disponibles se decidió por RISC-V por diversas ventajas: está detalladamente documentado y en constante revisión por parte de la comunidad, es modular por lo que se puede decidir que subconjunto de instrucciones implementar, está diseñado para optimizar diseños con pipeline, entre otros.

RISC-V es una ISA abierta y libre de tipo RISC que comenzó en el año 2010 como un proyecto de la Universidad de California en Berkeley cuyo objetivo era proveer una ISA abierta para incentivar el desarrollo de hardware dando una gran libertad en la arquitectura por parte de los diseñadores.

Al ser abierta a la comunidad, luego de ser liberada la ISA recibió una gran cantidad de colaboradores que continúan aportando a la idea que comenzó en California, logrando así una amplia gama de herramientas para el desarrollo de hardware y software sobre el. Además del set de instrucciones completamente documentado, se provee el tool-chain del compilador, simuladores de software completos, material de enseñanza, entre otros. Se puede encontrar toda la documentación y más en la página oficial del proyecto (*riscv.org*).

2. Diseño

El diseño del proyecto se puede dividir principalmente en tres áreas separadas: por un lado la micro-arquitectura del procesador; por otro lado sus periféricos, memorias, dispositivos de entrada, salida e interconexiones; y por último el software y entorno de desarrollo necesario para obtener código ejecutable a partir de un lenguaje de alto nivel.

2.1. Micro-arquitectura

Como se mencionó anteriormente, se comenzó el diseño de la micro-arquitectura dividiendo el procesamiento en cinco etapas distintas que conforman un pipeline clásico tipo RISC. En la Figura 1 se puede observar el diagrama en bloques que se implementó diseño e implementó.

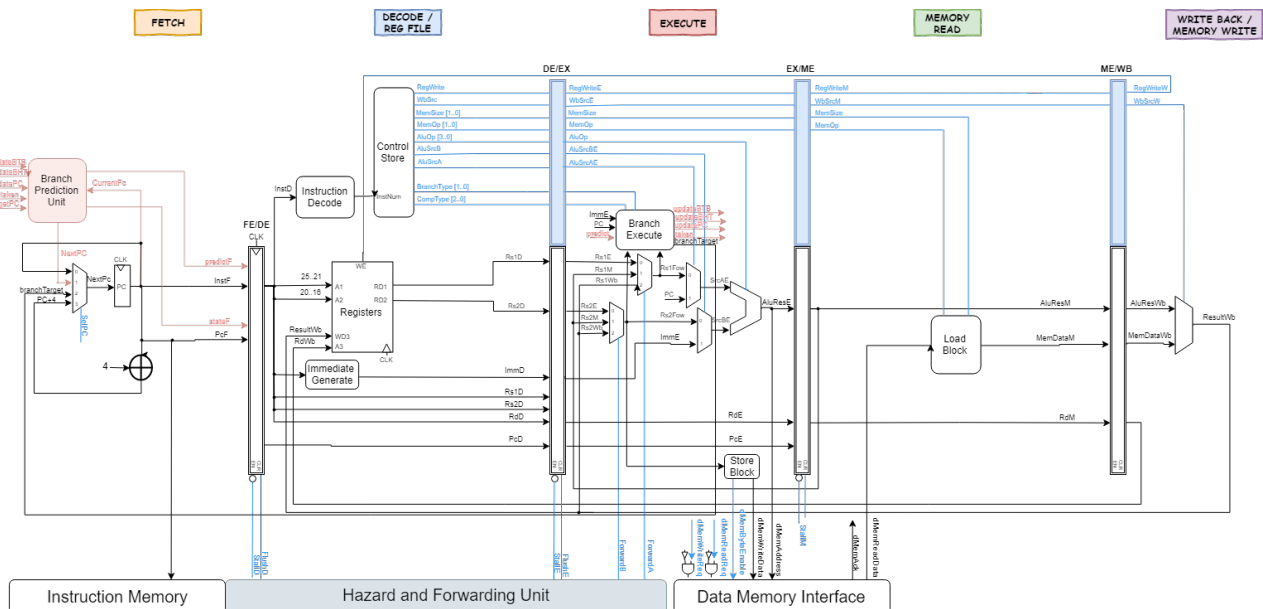


Figura 1: Diagrama en bloques de la micro-arquitectura.

Cada etapa cumple una función específica en el procesamiento de las instrucciones, y la interfaz entre cada etapa se implementa con un gran registro denominado *pipeline register*, el cual transfiere los datos entre cada etapa sincrónicamente con el clock. De esta forma se propaga a través del pipeline toda la información necesaria para que cada instrucción se ejecute correctamente. A través de estos registros también es posible detener el funcionamiento del pipeline en alguna etapa en particular, o eliminar información errónea u obsoleta como se detallará en siguientes secciones.

2.1.1. Fetch

Al compilar un programa en C para RISC-V, cada instrucción en C se convierte en una o varias palabras de 32 bits codificadas según la ISA. La primera operación que cualquier microprocesador debe realizar es leer estas instrucciones de alguna memoria, y esta lectura recibe el nombre de *fetch*. Esta etapa debe ser capaz de obtener las instrucciones de memoria para su posterior procesamiento en las siguientes etapas, como así también de redireccionar el flujo del programa ante saltos. En el caso de un programa completamente secuencial, esta etapa puede pensarse como un contador que se incrementa en saltos de a cuatro. Sin embargo la presencia de saltos, al igual que ciertos *hazards* complejizan su estructura. Debido a las características de la ISA todos los tipos de saltos requieren de un cálculo para la posición y en el caso condicional para definir si este ha de realizarse. Entonces, en principio es necesario obtener dichos resultados desde la etapa de Execute para poder definir el próximo PC al encontrarse una instrucción de salto en el flujo de un programa.

Para resolver la presencia de saltos, esta etapa del pipeline genera las distintas posibilidades de PC para tomar de la memoria, seleccionando la adecuada posteriormente. Debido a las características de la ISA, todos los tipos de saltos requieren de un cálculo para la posición, y en el caso condicional para definir si este ha de realizarse. Teniendo esto en cuenta la estructura principal de esta etapa se basa en la selección entre el PC actual incrementado (para avanzar secuencialmente), una dirección de salto proveniente de Execute, la permanencia del PC actual para detener el funcionamiento y un PC destinado a una dirección de reset.

Teniendo seleccionado el PC adecuado se lee la memoria para obtener una instrucción. Para esta operación se puede requerir usar el PC actual o el anterior si se da el caso en que se debe cancelar un fetch. Esto resulta relevante cuando se contempla el caso en que la memoria de instrucciones no responda en forma instantánea. Si esto ocurre, y al mismo tiempo la etapa de Execute indica que se debe tomar un branch, es necesario que el request a ese branch no se realice aún, por lo que se debe seleccionar el PC actual y se cancela el fetch (de aquí el nombre cancel fetch). Este cancel fetch es simplemente una máquina de estados (con dos estados, cancelar fetch o no) cuyas entradas son taken (si se tomó un branch, proviene de execute) y el ACK de la memoria. Además, esta etapa selecciona pasar como instrucción a la etapa de Decode lo leído de la memoria de instrucciones o forzar una instrucción NOP (no operation) para propagarla a través del pipeline. Esto resulta necesario para los casos de reset, flush (para eliminar la instrucción que iba a enviarse) o para detener únicamente a esta etapa del pipeline en caso de estar esperando la respuesta de la memoria. También se implementó un predictor dinámico de saltos en esta etapa para evitar tener que detener el funcionamiento del pipeline en cada operación de salto. El funcionamiento de dicha optimización se explicará más adelante.

2.1.2. Decode

La segunda etapa del pipeline se encarga de decodificar las instrucciones del programa y generar las señales que controlan la operación de las siguientes etapas del microprocesador. La instrucción obtenida por la etapa *Fetch* se encuentra codificada según la ISA. La instrucción llega en este formato a la etapa de *Decode* y es necesario interpretar el tipo de instrucción y desglosar la palabra de 32 bits de la ISA correctamente, obteniendo sus respectivos campos. Dependiendo del tipo de instrucción, se puede descomponer en los siguientes elementos: código de operación (OpCode), registros de origen de datos, registro de destino, valores inmediatos, etc.

Además, el bloque de 32 registros del microprocesador (Register File) se encuentra en la etapa Decode y es aquí donde se almacenan los 32 registros definidos en la especificación de RISC-V: 31 registros de propósito general x1-x31 y un registro x0 conectado a la constante 0.

Generación del OpCode y señales de control

Al extraer el OpCode, Func3 y Func7 de la instrucción en formato ISA, se decodifican en un número unívoco desde

0 a 44 que representa la operación a llevar a cabo, el cual es utilizado para desreferenciar una memoria de control (Control Store). Esta memoria de 45 palabras de 18 bits contiene el micro-código que comanda el procesador. Los 18 bits obtenidos del Control Store son conducidos a la siguiente etapa, donde algunos bits se utilizarán para comandar la misma. A su vez, los bits no utilizados son pasados a la siguiente con el mismo objetivo. Así, interpretando la instrucción una sola vez se logra que todas las etapas del microprocesador operen correctamente.

Registros de origen y destino

La etapa Decode se encarga del manejo del Register File (RF). Por esta razón, para operaciones que trabajan sobre los registros, se decodifica la instrucción y se accede al RF para que él o los registros de trabajo sean transferidos a la siguiente etapa Execute para que se operen sobre sus datos. También se hace la decodificación del registro donde se desea guardar los datos si es necesario.

Valores inmediatos

Para instrucciones que trabajan con números inmediatos, la codificación de estos dentro de la instrucción ISA depende del tipo, por lo que es necesario extraer los bits que componen el entero de 32 bits con el que puede trabajar el microprocesador. La extracción no es sencilla y depende del tipo de instrucción, por lo que un bloque dentro de la etapa se encarga exclusivamente de esta operación incluso aunque no se opere sobre el valor inmediato.

2.1.3. Execute

Esta etapa del pipeline es la encargada de la ejecución propiamente dicha de las instrucciones. Se compone de tres secciones que realizan las distintas funcionalidades de las instrucciones: la unidad de lógica aritmética (ALU), un módulo que procesa las instrucciones de salto, y otro que genera y envía solicitudes de lectura y escritura al bus de memoria.

Es importante destacar que los valores que ingresan a todos los bloques no son siempre el contenido de los registros que provienen de la etapa anterior. En el caso de un hazard del tipo read after write, los valores de los registros no estarían actualizados aún por lo que se realiza forwarding de las etapas siguientes. El proceso del forwarding así como también el análisis de los distintos hazards se realiza en el módulo de Hazard and Forwarding Unit que será explicado más adelante.

ALU

La ALU consiste de una serie de bloques que realizan las siguientes operaciones:

- Suma y resta
- Shifteo lógico y aritmético en ambas direcciones
- Operaciones lógicas de AND, OR y XOR
- Sumar 4 al PC
- Multiplicación (quedándose con la parte alta o la baja)
- División (quedándose con el cociente o el resto)
- Setear registro ante una comparación

Todos estos bloques se encuentran en paralelo, por lo que en cada instrucción se realizan todas las operaciones a la vez, en una suerte de ejecución especulativa. Finalmente, el resultado correcto es seleccionado al final con un gran Mux cuyos bits de selección provienen de la palabra de control, que indica que operación debe realizar la ALU. Resulta importante destacar que en el caso de las operaciones de división y resto, se implementaron en un primer momento de forma combinacional, introduciendo un retardo de propagación muy grande que limita la frecuencia máxima de clock a aproximadamente $10MHz$. Para solucionar este problema, se implementó esta instrucción en forma pipelineada, usando una cantidad de ciclos definida. Utilizando entonces un contador es posible saber cuando la operación larga esta en ejecución, y por lo tanto se frenan las etapas anteriores del pipeline.

Execute Branch

El bloque que controla los saltos tiene comparadores propios para verificar si se cumple la condición del branch. Al igual que en la ALU, todas las comparaciones se realizan en paralelo y se selecciona la correcta con el tipo de branch, el cual proviene de la palabra de control. Se cuenta además con un sumador propio para calcular el target del branch. Un detalle a destacar es que en el caso de que el target no esté alineado con la memoria (es decir que no es múltiplo de 4), la instrucción se descarta y no genera problemas en la memoria.

Bloque de Store

Para el caso en que la instrucción es de tipo *Store*, se puede realizar el request a memoria antes de que la instrucción llegue a la etapa de Memory. En este caso, a partir del tipo de operación a realizar (indicado por la palabra de control), se obtiene la palabra a escribir en memoria, junto con el paquete *ByteEnable* para escribir en memoria de forma adecuada. Además, en el caso de que la instrucción sea de tipo *Load* se realiza el request, para que en la etapa de Memory se obtenga el dato deseado y esta pueda ser procesado de forma correcta y almacenado en los registros de operación. El hecho de realizar todos los request a la memoria de datos dentro de esta etapa permite aprovechar los resultados de la ALU, y de esta forma no tener que esperar un ciclo de clock a que la instrucción avance a la siguiente etapa, ahorrándose entonces un ciclo.

2.1.4. Memory

La función de esta etapa es procesar los datos que se reciben de la memoria. Dado que hay tres tipos de lectura o escritura (byte, half o word) y que para el caso de la lectura de bytes y halves esta puede ser signada o no signada, es necesario procesar los datos recibidos de memoria ante un request o los que serán escritos a registros o memoria. Los datos son procesados a partir de la información que indica la palabra de control correspondiente a la instrucción de lectura o escritura y a la dirección de memoria correspondiente a la instrucción. Es importante destacar en este punto que hay algunas combinaciones de operación que son invalidas, como puede ser el load de un word cuando la dirección del mismo no es múltiplo de cuatro.

2.1.5. Write-Back

La última etapa de pipeline procura direccionar correctamente los resultados obtenidos a partir del procesamiento a lo largo del pipeline. A partir de la palabra de control, selecciona como resultado final lo obtenido a la salida de la ALU o el dato leído de memoria. Básicamente esta etapa opera como un mux controlado por la palabra de control.

2.1.6. Hazard and Forwarding Unit

El microprocesador cuenta con un bloque de detección de dependencias y hazards. Es importante mencionar que este unidad proporciona dos salidas de control diferentes para los demás bloques del pipeline: Stall y Flush. La primera se utiliza cuando es necesario frenar el avance de las instrucciones, mientras que la segunda se debe utilizar cuando hay datos erróneos en algunas etapa y por lo tanto deben eliminarse. A continuación se listan todos los casos en los que actúa la unidad de hazard y forwarding.

- Read After Write: se diferencian dos tipos, cuando el write hace un cálculo sobre los registros y cuando se hace un load. El primer caso se resuelve mediante Forwarding mientras que en el segundo con un Stall.
- Branch Misprediction: cuando el predictor de saltos realiza una predicción errónea. La solución es hacer un Flush de las etapas de Decode y Fetch.
- Latencia de la memoria de datos: es posible que la memoria de datos tarde varios ciclos en devolver el dato. Por lo tanto, se debe hacer Stall de las etapas previas a Memory.
- Instrucción larga: cuando la instrucción que debe ejecutar la ALU se extenderá por más de un ciclo de clock, envía una señal a esta unidad, y se hace un Stall y Flush de las etapas anteriores a Execute.

Además este bloque le proporciona a la etapa de Execute las señales de control para el forwarding. Se activa cuando los registros sobre los que trabaja la etapa de Execute son los mismo sobre los que una instrucción anterior va a volcar su resultado y que por lo tanto aún no están actualizados dichos registros. En lugar de frenar el pipeline en estos casos, se aprovecha el hecho de que estos resultados ya fueron calculados y sí están en una etapa posterior del

pipeline, esperando a ser guardados en los registros. Por lo tanto, dependiendo si esa instrucción ocurre justo antes, o dos antes, se toma el dato de la etapa de Memory o de la etapa de Write-Back.

2.1.7. Predictor de saltos

Como se explicó anteriormente, cada instrucción de salto se ejecuta en la etapa de Execute. Por lo tanto, en ese punto se encuentran cargadas en el pipeline las dos siguientes instrucciones, que pueden ser erróneas en el caso que el se deba tomar el branch, desperdiciándose en ese caso dos ciclos. Para evitar este problema se propuso implementar un predictor de saltos dinámico. El predictor consiste básicamente en dos bloques que se analizan a continuación.

El bloque de *branch history table* (**BHT**) es el encargado de realizar la predicción propiamente dicha. Funciona como una máquina de estados para cada instrucción de salto condicional (no para los saltos no condicionales), y almacena en una tabla el estado para cada salto. La máquina de estados tiene cuatro estados: fuertemente tomado, débilmente tomado, fuertemente no tomado y débilmente no tomado. El estado siguiente se calcula en base a si el salto fue tomado o no. En cuanto a la tabla que contiene los estados funciona con un sistema similar a una caché (pero sin tag), debido a la gran cantidad de PCs existentes. En cada línea de la memoria se guardan simultáneamente los estados para cuatro PCs.

Este bloque en principio no soluciona el problema de desperdiciar dos instrucciones en cada salto, dado que la información de si un salto es tomado o no no es suficiente para redireccionar el flujo del programa ya que el destino del salto se calcula en la etapa de Execute.

Esto se soluciona con el bloque denominado *branch target buffer* (**BTB**). El mismo se encarga de almacenar la posición a la que se debe saltar para cada instrucción de salto en el programa. Funciona como una memoria caché la cual tiene como entrada el PC actual, y como salida una señal de hit y el destino del salto.

Dado que no sería posible tener una tabla del tamaño de la memoria de instrucciones, la misma se implementa con un tamaño menor y se agrega un campo de *tag* para identificar cada PC.

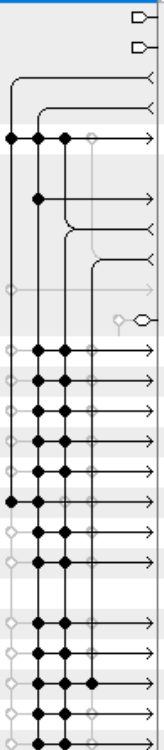
Todas las tablas del predictor se leen desde la etapa de *fetch* mientras que se actualizan desde la etapa de *execute*. De esta forma, los campos comienzan en estado inválido y se van completando a medida que los saltos se van ejecutando en *execute*. Los saltos condicionales actualizan tanto el **BTB** como la **BHT**, mientras que los saltos no condicionales solo actualizan el **BTB**. La etapa de *execute* también es la encargada de corregir los errores del predictor. Cuando se detecta que la predicción difiere del resultado real del salto, se corrige el PC en la etapa de *fetch*. Si el salto debía ser tomado se toma el destino calculado, y si el salto no debía ser tomado se toma la siguiente instrucción a la del salto. En ambos casos es necesario eliminar las instrucciones de *fetch* y *decode*.

2.2. Periféricos

Dado que el diseño de dispositivos periféricos, buses e interconexión no forman parte de los objetivos del trabajo, se decidió hacer uso de *Platform Designer* para generar periféricos mapeados en el mapa de memoria del microprocesador. Este software es parte de *Intel® Quartus® Prime* y se encarga de generar las conexiones entre distintos bloques IP y subsistemas. En la interfaz gráfica se generan los bloques que se desean utilizar configurando sus respectivos parámetros y luego se conectan sus distintas señales. La comunicación se realiza por medio de un bus desarrollado por Intel denominado *Avalon*, del cual se utilizaron tres interfaces:

- Avalon Memory Mapped (MM): Interfaz basada en solicitudes de lectura y escritura, típica de conexiones maestro-esclavo. Permite granularidad de bytes en los accesos del bus de 32 bits.
- Avalon Streaming (ST): Interfaz que soporta un flujo unidireccional de datos, streaming de paquetes, DSP, etc. En este proyecto se utiliza para las señales de video.
- Avalon Conduit: Señales que permiten el conexionado de componentes fuera del chip.
- Avalon Reset: Señales que proveen conexionado de resets.
- Avalon Clock: Interfaz que provee conexionado de clocks.

Con los archivos de diseño del proyecto se creó un componente en Platform Designer con el fin de conectarlo de forma integrada con los periféricos. Hace falta aclarar que para esta operación se debió convertir todos los archivos de diagrama en bloques, a archivos de VHDL que son los únicos que permite utilizar el programa. Una vez generado el bloque del procesador, se conectan sus respectivas señales de reset, clock, master de instrucciones y datos, como se observa en la siguiente figura.

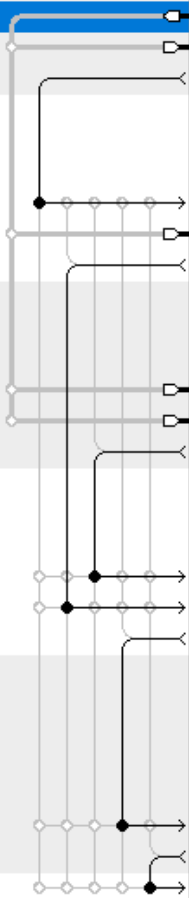


Connections	Name	Description	Base	End
	Clock	Clock Source		
	clk_in	Clock Input		
	clk_in_reset	Reset Input		
	clk	Clock Output		
	clk_reset	Reset Output		
	PLL	ALTPLL Intel FPGA IP	0x0100_1300	0x0100_130f
	EV19_Core_0	EV19_Core		
	reset	Reset Input		
	Data_Master	Avalon Memory Mapped Master		
	Instruction_Master	Avalon Memory Mapped Master		
	clock	Clock Input		
	enablePredictor	Conduit		
	ID	System ID Peripheral Intel FPGA IP	0x0100_0000	0x0100_0007
	LEDs	PIO (Parallel I/O) Intel FPGA IP	0x0100_1000	0x0100_101f
	Dip_Switch	PIO (Parallel I/O) Intel FPGA IP	0x0100_1100	0x0100_110f
	Push_Button	PIO (Parallel I/O) Intel FPGA IP	0x0100_1200	0x0100_120f
	ADC	ADC Controller for DE-series Boards	0x0100_2000	0x0100_201f
	Accelerometer	Accelerometer SPI Mode		
	Timer	Interval Timer Intel FPGA IP	0x0100_4000	0x0100_401f
	Performance_Counter	Performance Counter Unit Intel FPGA IP	0x0100_5000	0x0100_503f
	Mouse	PS/2 Controller		
	Keyboard	PS/2 Controller	0x0100_6100	0x0100_6107
	VGA	VGASubsystem	multiple	multiple
	ROM	On-Chip Memory (RAM or ROM) Intel ...	0x0000_0000	0x0000_0fff
	RAM	On-Chip Memory (RAM or ROM) Intel ...	0x0200_0000	0x0200_03ff
	SDRAM	SDRAM Controller Intel FPGA IP	0x0400_0000	0x05ff_ffff

Figura 2: Pantalla principal de Platform Designer

2.2.1. Salida de video VGA

A partir de la herramienta de *Platform Designer* se generó un módulo encargado de realizar la interfaz de VGA. El módulo recibe un clock de 25 (MHz) necesario para refrescar una imagen de 640x480 a 60pfs y una señal de video por un bus Avalon-ST. La misma se genera internamente con dos fuentes: un buffer de video y un buffer de caracteres. El buffer de video se encuentra en la SDRAM externa y un periférico específico se encarga de leerlo en forma secuencial y generar el stream de valores RGB. El buffer de caracteres se implementa con otro periférico al cual se le escriben caracteres ASCII y este los renderiza para generar la representación gráfica de los mismos. Ambas señales se unen con un bloque denominado *Alpha Blender*, obteniendo así la señal de video que toma el controlador VGA para generar las señales para cada color y las de sincronización vertical y horizontal. Cabe destacar que las señales RGB que genera el módulo son digitales. Por esta razón fue necesario diseñar una placa que cumpla la función de DAC y así obtener las señales analógicas para transmitir por el cable hasta el monitor y así obtener una salida de video.



Connections	Name	Description	Export	Clock
	⊕ Clock_System	Clock Source		exported
	⊕ Clock_VGA	Clock Source		exported
	⊖ Pixel_Buffer	Pixel Buffer DMA Controller		
	clk	Clock Input	<i>Double-click to export</i>	Clock_System
	reset	Reset Input	<i>Double-click to export</i>	[clk]
	avalon_pixel_dma_master	Avalon Memory Mapped Master	pixel_buffer_master	[clk]
	avalon_control_slave	Avalon Memory Mapped Slave	pixel_buffer_slave	[clk]
	avalon_pixel_source	Avalon Streaming Source	<i>Double-click to export</i>	[clk]
	⊖ Resampler	RGB Resampler		
	clk	Clock Input	<i>Double-click to export</i>	Clock_System
	reset	Reset Input	<i>Double-click to export</i>	[clk]
	avalon_rgb_sink	Avalon Streaming Sink	<i>Double-click to export</i>	[clk]
	avalon_rgb_slave	Avalon Memory Mapped Slave	rgb_resampler_slave	[clk]
	avalon_rgb_source	Avalon Streaming Source	<i>Double-click to export</i>	[clk]
	⊖ Character_Buffer	Character Buffer for VGA Display		
	clk	Clock Input	<i>Double-click to export</i>	Clock_System
	reset	Reset Input	<i>Double-click to export</i>	[clk]
	avalon_char_control_slave	Avalon Memory Mapped Slave	char_buffer_control_sla...	[clk]
	avalon_char_buffer_slave	Avalon Memory Mapped Slave	char_buffersource_slave	[clk]
	avalon_char_source	Avalon Streaming Source	<i>Double-click to export</i>	[clk]
	⊖ Alpha_Blender	Alpha Blender		
	clk	Clock Input	<i>Double-click to export</i>	Clock_System
	reset	Reset Input	<i>Double-click to export</i>	[clk]
	avalon_foreground_sink	Avalon Streaming Sink	<i>Double-click to export</i>	[clk]
	avalon_background_sink	Avalon Streaming Sink	<i>Double-click to export</i>	[clk]
	avalon_blended_source	Avalon Streaming Source	<i>Double-click to export</i>	[clk]
	⊖ Dual_Clock_FIFO	Dual-Clock FIFO		
	clock_stream_in	Clock Input	<i>Double-click to export</i>	Clock_System
	reset_stream_in	Reset Input	<i>Double-click to export</i>	[clock_stream_in]
	clock_stream_out	Clock Input	<i>Double-click to export</i>	Clock_VGA
	reset_stream_out	Reset Input	<i>Double-click to export</i>	[clock_stream_out]
	avalon_dc_buffer_sink	Avalon Streaming Sink	<i>Double-click to export</i>	[clock_stream_in]
	avalon_dc_buffer_source	Avalon Streaming Source	<i>Double-click to export</i>	[clock_stream_out]
	⊕ VGA_Controller	VGA Controller Intel FPGA IP		Clock_VGA

Figura 3: Sistema de video con salida VGA

2.2.2. Controlador SDRAM

Con el fin de aprovechar al máximo las características de la placa de desarrollo se configuró el bloque controlador de memoria SDRAM. La misma es externa a la FPGA y permite ampliar enormemente la cantidad de memoria disponible. Si bien es mas lenta que la memoria on-chip (las lecturas y escrituras toman entre 5 y 6 ciclos a corriendo tanto el core como el controlador a 50MHz).

2.2.3. Otros periféricos

Además de los módulos de VGA y SDRAM, se agregaron una serie de periféricos aprovechando los bloques IP que el software utilizado brinda por defecto. Entre ellos se encuentran un controlador PS2, controlador del ADC y acelerómetro integrados en la placa de desarrollo, timers por hardware, pines de entrada y salida, y bloques de protocolos de comunicación (UART, I2c y SPI). El objetivo de estos es el de probar la arquitectura en una diversa serie de aplicaciones.

2.3. Software

2.3.1. Simulación

Se implementó un simulador para verificar la factibilidad del diseño básico mientras se desarrollaba el hardware en Quartus. El simulador cuenta con un pipeline de 5 etapas, que es detenido cuando existen dependencias entre los datos. El simulador fue implementado en Python 3.7 y permite la programación del microcontrolador utilizando lenguaje assembly. Permite también el ingreso de breakpoints definidos por usuario y la ejecución paso a paso.

2.3.2. Entorno de desarrollo

Entorno en consola

Se compiló bajo un entorno Linux las herramientas de desarrollo GNU-GCC para que utilice la arquitectura RISC-V implementada. Esto permite el acceso a todas las herramientas de GNU para el desarrollo de software permitiendo el armado de mapas de memoria para cargar al microprocesador. Para realizar esto se clonó el repositorio mantenido por SiFive y se utilizó make para configurar los parámetros necesarios como la arquitectura del microprocesador.

IDE

Se configuró el entorno Eclipse para desarrollo RISC-V. Para esto se instalaron las dependencias utilizando el administrador de paquetes de node.js, para esto se utilizó el siguiente comando:

```
$ npm install @gnu-mcu-eclipse/riscv-none-gcc
```

Luego de la instalación se creó un proyecto en Eclipse y se configuró la arquitectura de RISC-V implementada. Configurando las herramientas necesarias para la generación de la imagen binaria que posteriormente será cargada al microprocesador.

Se escribió la rutina de start en assembler para inicializar el microprocesador y poder ejecutar código C, durante esta rutina se inicializan las variables globales, se limpia la memoria a usarse e inicializa el stack pointer en la posición de memoria necesaria.

Se configuró el linker script que permite posicionar las secciones de código necesarias en las distintas memorias iMem, Ram y Sram, indicando su posición lógica en el bus avalon y sus tamaños. También se incluye el código de inicialización y se indica la posición donde el script de start debe posicionar el stack.

Se desarrollaron las herramientas necesarias para cargar el programa generado al microprocesador, integrando esta a la secuencia de compilación. En primer lugar se convierte el iHex generado utilizando el programa objdump de gcc a formato mif, luego se invoca el comando para cargar el programa a la fpga, permitiendo el ágil desarrollo de software.

2.3.3. Capa de abstracción de hardware

Se implementó una capa de abstracción de hardware, HAL por sus siglas en inglés, que permite al programador utilizar los periféricos del microprocesador sin necesidad de acceder directamente a los registros y realizar las tareas repetitivas necesarias para la utilización de algunos protocolos.

3. Resultados

Finalmente, se logró llevar a cabo el diseño, desarrollo e implementación de un microprocesador basado en la ISA RISC-V exitosamente. El mismo funciona sobre una FPGA y cumple con todas las instrucciones establecidas en la ISA estándar más las extensiones para multiplicación y división entera. El clock del microprocesador puede lograr frecuencias de hasta 50 MHz habiendo optimizado los cuellos de botella que generaban, por ejemplo, la operación de división. Se logró exitosamente instalar y trabajar con las herramientas disponibles en internet para compilar diversos programas de prueba escritos en código C en RISC-V. Este código assembly generado se cargó en la FPGA y el procesador lo ejecuta correctamente.

Entre los diversos programas de prueba utilizados se encuentran cálculo series de Fibonacci (de forma iterativa y recursiva), fractales de Mandelbrot, lecturas de señales analógicas con el ADC representadas en los LEDs integrados, entre otros.

4. Trabajo futuro

Debido al tiempo disponible para realizar el proyecto, se prescindió de ciertas implementaciones y propuestas sobre las cuales sería interesante indagar con el fin de agregar funcionalidad al diseño.

- **CSR:** Los registros de control y estado (CSR) son necesarios para la detección de algunas excepciones y además distinguen el modo de usuario privilegiado. Esto permite la división entre sistema operativo y aplicación de usuario. Las instrucciones sobre los CSR se encuentran especificadas por RISC-V.
- **Interrupciones y excepciones:** Es posible implementar un sistema de interrupciones en el microprocesador para poder hacer saltos asincrónicos a subrutinas. Esto es de gran valor para todas las aplicaciones en tiempo real donde se pueda utilizar el microprocesador. Por otro lado, un sistema de excepciones para detectar condiciones inusuales que ocurren durante ejecución es de vital importancia. Esto asegura que no se ejecuten operaciones inválidas como pueden ser: *Address Misaligned Exception*, *No Instruction Fetch Misaligned Exception*, *Invalid Operation Exception*, entre otras.
- **Instrucciones específicas RISC-V** contempla el caso de implementar instrucciones que no están en el estándar como por ejemplo MAC (multiply and accumulate) para la implementación eficiente de un DSP. De esta forma se podrían implementar instrucciones específicas para optimizar una aplicación en particular con el uso de profiling y un análisis estadístico,
- **Caché** Se podría implementar hardware para agregar un sistema de caché, tanto de datos como de instrucciones. El objetivo de este sería obtener una mejora en el rendimiento reduciendo la latencia promedio en el acceso a datos. Sin embargo en este hardware en particular la mejora no sería apreciable, dado que el acceso a la memoria on-chip no tiene latencia alguna. Incluso se podría llegar a ver afectado el rendimiento, en el caso de que se tenga que hacer un stall en los casos de un miss.
- **FPU** El software de Quartus IP provee bloques para manejo aritmético en punto flotante, por lo que sería relativamente simple añadir esta extensión del estándar de RISC-V. Es necesario agregar un set completo de registros de punto flotante, la decodificación de las nuevas instrucciones junto con sus palabras de control en el control store, la extensión de la palabra de control para soportar las nuevas operaciones, entre otras cosas.
- **Optimización de recursos** La utilización de archivos de diagrama en bloques en Quartus puede provocar que el uso de recursos sea menos eficiente, dado que no se realizan optimizaciones a través de la frontera de los bloques. Usando íntegramente en el proyecto un lenguaje de descripción de hardware como VHDL y poniendo foco en algunas secciones del diseño sería posible reducir el área utilizada en la FPGA.

Por otro lado, como se mencionó anteriormente existe una latencia de algunos ciclos en los accesos a la RAM externa. Dado que se encontró que la frecuencia del procesador no se puede incrementar mucho más, una posible mejora sería correr la memoria externa a una frecuencia mayor.

- **Multiprocesamiento** Siguiendo con la idea del punto anterior, si se logra reducir el uso de recursos de la FPGA, el diseño de la arquitectura permitiría implementar con cierta facilidad un procesador con varios núcleos dado que es posible instanciar cada core y conectarlos a través del *Platform Designer*. De esta manera sería posible que todos los núcleos corran el mismo código (accediendo a la memoria de forma arbitrada) dividiendo sus tareas usando un número de identificación único para cada uno y que se comuniquen a través de un espacio de memoria común. Otra topología posible sería que cada núcleo tenga una memoria de programa, como así también una memoria de datos privada.

Una aplicación entre las ya mencionadas a lo largo de este trabajo sería la posibilidad de que el programa que calcula un fractal, divida el plano en por ejemplo, cuatro regiones, las cuales al ser completamente independientes se pueden ejecutar en paralelo en cada núcleo.

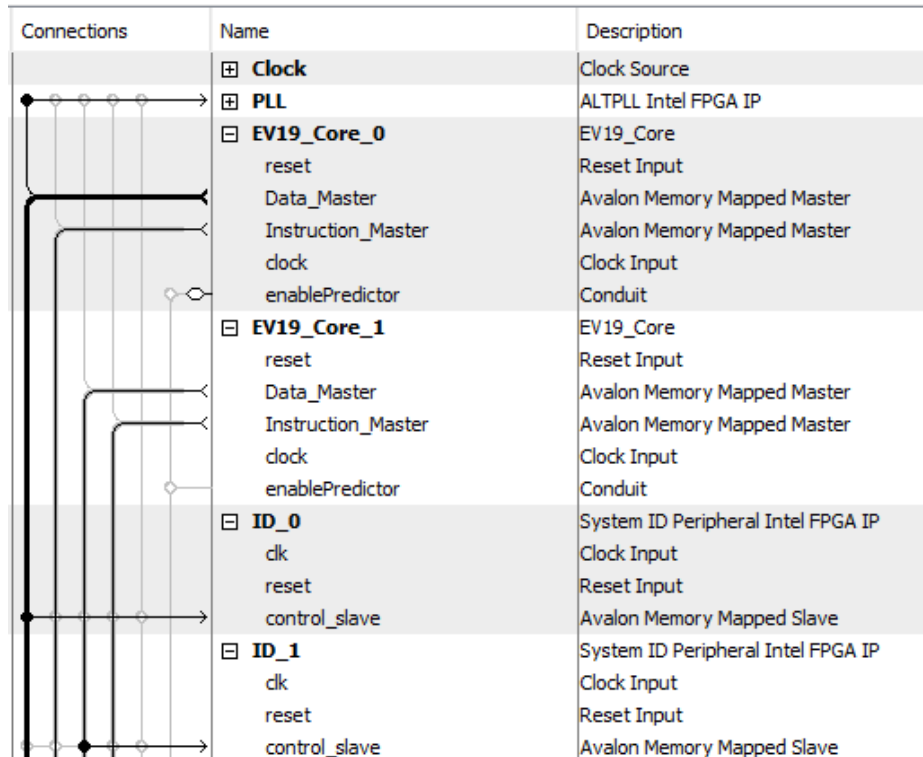


Figura 4: Proyecto con dos núcleos instanciados

Estas son solo algunas ideas que se podrían implementar. Sin embargo, debido al diseño particular de la ISA RISC-V, se da mucho lugar a extender las funcionalidades del microprocesador sin romper las extensiones existentes provistas por RISC-V y sin incurrir en fragmentación de software. Con un diseño correcto, un microprocesador de aplicación específica puede desarrollarse con tranquilidad sobre la base de una ISA robusta y utilizada internacionalmente, que además cuenta con el soporte de herramientas que permiten trabajar con lenguajes de más alto nivel.

5. Conclusión

Como se mencionó anteriormente, la implementación del microprocesador fue exitosa. El mismo es capaz de ejecutar código desarrollado en lenguaje C y compilado con el tool-chain de RISC-V. Durante toda la etapa del diseño e implementación se puede afirmar que el equipo se ha interiorizado ampliamente en el funcionamiento de un microprocesador. Además, las capacidades del microprocesador implementado son elevadas y tiene el potencial para ser utilizado en aplicaciones reales.

Entre las cosas más positivas, el hecho de utilizar una implementación estándar permitió no perder tiempo desarrollando las instrucciones ni el compilador. Todo el tool-chain se puede encontrar en internet y permite concentrarse en el diseño de la micro-arquitectura.

Referencias

- [1] A. Waterman, Y. Lee, D. A. Patterson, and K. Asanovi, *The RISC-V Instruction Set Manual. Volume 1: User-Level ISA, Version 2.0*, 2014.
- [2] D. K. Dennis, A. Priyam, S. S. Virk, S. Agrawal, T. Sharma, A. Mondal, and K. C. Ray, *Single cycle RISC-V micro architecture processor and its FPGA prototype*, 2017 7th International Symposium on Embedded Computing and System Design (ISED), 2017.
- [3] Yonghong Yan, *Lecture 09: RISC-V Pipeline Implementation*, CSE 564 Computer Architecture Summer 2017, Department of Computer Science and Engineering.
- [4] A. Raveendran, V. B. Patil, D. Selvakumar, and V. Desalphine, *A RISC-V instruction set processor-micro-architecture design and analysis*, 2016 International Conference on VLSI Systems, Architectures, Technology and Applications (VLSI-SATA), 2016.
- [5] D. A. Patterson and J. L. Hennessy, *Computer Organization and Design: The Hardware Software Interface: RISC-V Edition*. Cambridge, MA: Morgan Kaufmann Publishers, 2018.
- [6] S. L. Harris and D. M. Harris, *Digital design and computer architecture, 2nd ed.* Amsterdam: Elsevier / Morgan Kaufmann Publishers, 2018.