Código	FPI-009
Objeto	Guía de elaboración de Informe final de proyecto
Usuario	Director de proyecto de investigación
Autor	Secretaría de Ciencia y Tecnología de la UNLaM
Versión	5
Vigencia	03/9/2019

Departamento:
Ingeniería e Investigaciones Tecnológicas

Programa de acreditación:
PROINCE

Programa de Investigación[1]:

Código del Proyecto:
C-219

Título del proyecto
Desarrollo e implementación de una arquitectura basad en el conjunto de instrucciones RISC-V

Director:
Ing. Szklanny, Fernando Ignacio

Codirector:
Lic. Maidana, Carlos Eduardo

Integrantes:
Ing. Gho, Edgardo Alberto
Ing.Ferreyra Birón, Martín

Resolución Rectoral de acreditación: N.º 355/2019
Fecha de inicio:
01.01.2019
Fecha de finalización:
30.04.2021

1 Los Programas de Investigación de la UNLaM están acreditados con resolución rectoral, según lo indica la Resolución HCS Nº 014/15 sobre **Lineamientos generales para el establecimiento, desarrollo y gestión de Programas de Investigación a desarrollarse en la Universidad Nacional de La Matanza**. Consultar en el departamento académico correspondiente la inscripción del proyecto en un Programa acreditado.

● Desarrollo del proyecto

A.1. Grado de ejecución de los objetivos inicialmente planteados, modificaciones o ampliaciones u obstáculos encontrados para su realización (desarrolle en no más de dos (2) páginas)

Análisis de tareas realizadas y grado de cumplimiento

Se debe tener en cuenta que el año 2020 fue un año particularmente especial, debido a la implantación del ASPO (Aislamiento Social, Preventivo y Obligatorio), producto de la pandemia COVID-19. Esto impidió a los investigadores asistir a la universidad a partir del mes de marzo 2020. No obstante, el grupo de investigadores pudo cumplir en la totalidad las tareas previstas de acuerdo a lo planteado en la programación de actividades del punto 2.14 del protocolo de presentación. Se detalla a continuación cada uno de los puntos:

ETAPA I

1. **Análisis detallado del juego de instrucciones RISC-V.**
2. **Análisis de arquitecturas ya implementadas. Análisis de sus ventajas, desventajas y limitaciones.**
3. **Definición y análisis básico del hardware a implementar.**
4. **Diseño e implementación de los bloques funcionales a implementar.**
5. **Desarrollo de cada uno de los bloques funcionales.**
6. **Selección del elemento de lógica programable a utilizar.**
7. **Simulación y puesta en marcha del sistema desarrollado sobre el elemento de lógica programable seleccionado.**

Los puntos que se encontraban en un grado de avance importante al finalizar el año 2019 (puntos 3,4 y 5) se completaron a principios de 2020 y se comenzó a avanzar sobre los puntos de la ETAPA II. A partir de marzo de 2020, el grupo no pudo acceder a la computadora que se adquirió específicamente para la tarea de desarrollo, como tampoco se pudo acceder al hardware específico requerido (FPGA Xilinx Artix-7 XC7A35), lo que implicó que todas las tareas se simularan en la plataforma Vivado. Cabe destacar que esta plataforma está diseñada específicamente para poder realizar estas simulaciones con un grado de precisión muy alto.

ETAPA II

1. **Puesta en marcha del prototipo de laboratorio.**
2. **Implementación de otras alternativas de softcore sobre el hardware elegido.**

3. **Ejecución de pruebas (benchmarks) sobre el prototipo desarrollado y los otros softcores respecto de los cuales se desea comparar el rendimiento con el desarrollo del presente proyecto:**
4. **Correcciones sobre diseño en base a resultados de las pruebas realizadas.**
5. **Selección del elemento a utilizar para el prototipo definitivo.**
6. **Pruebas finales.**
7. **Documentación del desarrollo, tal como se requiere para formar parte de la comunidad RISC-V.**

La fase 1 de la ETAPA II se pudo llevar a cabo durante el mes de febrero y parte de marzo de 2020. Como se mencionó antes, el grupo tuvo que dejar de asistir al laboratorio a fines del mes de marzo de 2020.

Las pruebas de eficiencia se llevaron a cabo y se implementaron en la plataforma Vivado. El producto resultante de estas pruebas se documentó en el ANEXO I de este informe final en el que se pueden apreciar los diagramas de tiempo del procesador.

Cabe destacar que se implementó una mejora en el diseño vinculado con la alineación de datos. Uno de los puntos donde RISC-Vp tiene una notable mejora es en el acceso de datos no alineados. El procesador FE310 (que fuera elegido como referencia de un chip comercial, para poder realizar las comparaciones con RISC-Vp) no soporta desde el hardware acceso a datos de forma desalineada.

El mismo genera una falla (trap) cuando detecta este tipo de accesos a memoria, lo que implica una importante penalidad de tiempo requerida para resolver la situación al capturar el trap, o, por el contrario, se requiere pensar el código de forma tal que se eviten los accesos desalineados, lo que complica el desarrollo del programa. El procesador RISC-Vp soporta nativamente accesos desalineados con penalidades de tiempo muy menores a las que se podrían obtener con FE310. Inclusive permite utilizar accesos desalineados sin que el programador se vea involucrado, lo que simplifica mucho este tipo de accesos ya que no hay que tomar precauciones en el código para evitar los mismos.

Para poder realizar los programas de prueba se utilizó la herramienta RIPES, que es un simulador/ensamblador que brinda como salida un archivo del tipo .BIN· Para que ese programa pueda ser cargado en RISC-Vp debe alojarse en un BLOCKRAM del FPGA, como habitualmente se realiza en este tipo de diseños (SoC). Los formatos que se requieren para poder definir los valores cargados en el BLOCKRAM contra el formato que emite RIPES son muy diferentes. Para poder resolver este problema de conversión de formatos, se desarrolló un pequeño programa en lenguaje C que toma como entrada el archivo binario (.bin de RIPES) y lo traduce a un archivo de secuencia de inicialización de BLOCKRAM. Todos los aspectos técnicos se encuentran detallados minuciosamente en el ANEXO I.

El código del prototipo ha sido publicado a la comunidad en el sitio ***github*** bajo una licencia de uso abierto respetando la propiedad intelectual de los autores (GPL). El procesador RISC-Vp puede ser descargado y simulado dentro del software Xilinx Vivado 2019-2. Todos los programas y herramientas desarrollados se encuentran también disponibles.

Sitio de descarga del código: https://github.com/edgardogho/RiscVP

- **Principales resultados de la investigación**

B.1. Publicaciones en revistas (informar cada producción por separado)

Artículo 1:	
Autores	
Título del artículo	
N° de fascículo	
N° de Volumen	
Revista	
Año	
Institución editora de la revista	
País de procedencia de institución editora	
Arbitraje	Elija un elemento.
ISSN:	
URL de descarga del artículo	
N° DOI	

B.2. Libros

Libro 1	
Autores	
Título del Libro	
Año	
Editorial	
Lugar de impresión	
Arbitraje	Elija un elemento.
ISBN:	
URL de descarga del libro	
N° DOI	

B.3. Capítulos de libros

Autores	
Título del Capitulo	
Título del Libro	
Año	
Editores del libro/Compiladores	
Lugar de impresión	
Arbitraje	Elija un elemento.
ISBN:	
URL de descarga del capítulo	
N° DOI	

B.4. Trabajos presentados a congresos y/o seminarios

Autores	
Título	
Año	
Evento	
Lugar de realización	
Fecha de presentación de la ponencia	
Entidad que organiza	
URL de descarga del trabajo (especificar solo si es la descarga del trabajo; formatos pdf, e-pub, etc.)	

B.5. Otras publicaciones

Autores	Edgardo Gho
Año	2020
Título	VHDL RISC-Vp Core
Medio de Publicación	https://github.com/edgardogho/RiscVP

- **Otros resultados. Indicar aquellos resultados pasibles de ser protegidos a través de instrumentos de propiedad intelectual, como patentes, derechos de autor, derechos de obtentor, etc. y desarrollos que no pueden ser protegidos por instrumentos de propiedad intelectual, como las tecnologías organizacionales y otros. Complete un cuadro por cada uno de estos dos tipos de productos.**

C.1. Títulos de propiedad intelectual. Indicar: Tipo (marcas, patentes, modelos y diseños, la transferencia tecnológica) de desarrollo o producto, Titular, Fecha de solicitud, Fecha de otorgamiento

Tipo	Titular	Fecha de Solicitud	Fecha de Emisión

C.2. Otros desarrollos no pasibles de ser protegidos por títulos de propiedad intelectual. Indicar: Producto y Descripción.

Producto	Descripción

D. Formación de recursos humanos. Trabajos finales de graduación, tesis de grado y posgrado. Completar un cuadro por cada uno de los trabajos generados en el marco del proyecto.

D.1. Tesis de grado

Director (apellido y nombre)	Autor (apellido y nombre)	Institución	Calificación	Fecha /En curso	Título de la tesis

D.2 Trabajo Final de Especialización

Director (apellido y nombre)	Autor (apellido y nombre)	Institución	Calificación	Fecha /En curso	Título del Trabajo Final

D.2. Tesis de posgrado: Maestría

Director (apellido y nombre)	Tesista (apellido y nombre)	Institución	Calificación	Fecha /En curso	Título de la tesis

D.3. Tesis de posgrado: Doctorado

Director (apellido y nombre)	Tesista (apellido y nombre)	Institución	Calificación	Fecha /En curso	Título de la tesis

D.4. Trabajos de Posdoctorado

Director (apellido y nombre)	Posdoctorando (apellido y nombre)	Institución	Calificación	Fecha /En curso	Título del trabajo	Publicación

E. Otros recursos humanos en formación: estudiantes/ investigadores (grado/posgrado/ posdoctorado)

Apellido y nombre del Recurso Humano	Tipo	Institución	Período (desde/hasta)	Actividad asignada

F. Vinculación: Indicar conformación de redes, intercambio científico, etc. con otros grupos de investigación; con el ámbito productivo o con entidades públicas. Desarrolle en no más de dos (2) páginas.

El objetivo principal de este proyecto fue adquirir conocimientos y experiencia en la arquitectura RISC-V.

La filosofía de este tipo de implementaciones se está haciendo cada vez más popular. La posibilidad de desarrollar un núcleo de una unidad central de procesos y al mismo tiempo tener personas en el mundo que desarrollan software de base para poder implementar y probar el diseño hace que esta actividad se potencia día a día.

En muchos casos ya hay empresas que ven a RISC-V como una Unidad Central de Procesos de alternativa gratuita para el diseño de CPUs de teléfonos celulares, microcontroladores de aplicaciones industriales y todas las aplicaciones en las que la familia ARM hoy tiene supremacía.

Al existir una plataforma que no requiere la erogación de ningún royalty, la propuesta es sumamente tentadora.

G. Otra información. Incluir toda otra información que se considere pertinente.

En este caso particular, la vinculación se fundamenta en obtener una plataforma y las herramientas necesarias gratuitas para el dictado de la asignatura Arquitectura de Computadoras (1109) de la carrera de Ingeniería en Informática del Departamento de Ingeniería e Investigaciones Tecnológicas de la Universidad Nacional de La Matanza. Actualmente en la materia Arquitectura de Computadoras se instruye a los alumnos utilizando una Unidad Central de Procesos comercial de arquitectura Von Neumann, en la que no pueden modificar nada y no se puede "ver" que sucede en el interior. La idea de comenzar a desarrollar esta CPU de arquitectura Harvard, motivó al grupo docente, debido a que el proyecto RISC-V provee a la comunidad de lo que se denomina una arquitectura de set de instrucciones, que incluye y define: un set de instrucciones, los modos de direccionamiento, la estructura de registros y el espacio de memoria direccionable. Esto permite tener la libertad de implementar el hardware interno de la manera que uno crea conveniente y es el principio de la propuesta pedagógica, en el que los alumnos podrán, no solo ver el funcionamiento interno, podrán ver los diagramas de tiempo (en tiempo real) y podrán entender el "Data Path" de los datos internos y poder modificarlo a su antojo, pudiendo mejorar si lo deseara, la perfomance de un procesador.

La comunidad RISC-V provee gratuitamente de herramientas de programación y simulación que permitirá al alumno interpretar el funcionamiento interno de una CPU de arquitectura moderna.

Es importante volver a remarcar que el proyecto RISC-V es público y cualquier fabricante de SoC (System On Chip) puede usarlo sin necesidad de pagar franquicias. Con lo que se espera que en un mediano plazo cualquier dispositivo de IOT o dispositivo móvil incorpore un chip de este tipo. Este aporte a los alumnos creemos que es significativo ya que para la producción de software de base es imprescindible tener estos conocimientos.

H. Cuerpo de anexos:

- Anexo I: Descripción del proyecto y código completo del Core RISC-Vp.
- Anexo II:
  - FPI-013: Evaluación de alumnos integrantes. (si corresponde)
  - FPI-014: Comprobante de liquidación y rendición de viáticos. (si corresponde)
  - FPI-015: Rendición de gastos del proyecto de investigación acompañado de las hojas foliadas con los comprobantes de gastos.
  - FPI-035: Formulario de reasignación de fondos en Presupuesto.
- Anexo III: Alta patrimonial de los bienes adquiridos con presupuesto del proyecto (FPI 017)
- Nota justificando baja de integrantes del equipo de investigación.

Lic. Carlos Eduardo Maidana
Codirector

Lugar y fecha : San Justo 23 de abril de 2021

ANEXO I

Descripción del proyecto

Se realizó el diseño e implementación en VHDL de un procesador que cumple con el set de instrucciones RV32I (Risc-V 32 bits Integer base). El mismo hace una implementación de todos los modos de direccionamiento (R,I,S,B,U y J). Se obviaron ciertas instrucciones como FENCE, que es útil en entornos multiprocesador (solo se implementó un único núcleo hardware thread), instrucciones como ECALL, EBREAK que llaman al sistema operativo (debido a que este diseño tiene fines de sistema embebido más parecido a un microcontrolador) y las de acceso a los registros CSR (que no fueron implementados ya que exceden los alcances de esta etapa del desarrollo). De todas formas, sus respectivos modos de direccionamiento están soportados, y por ende, agregar estas instrucciones faltantes es trivial.

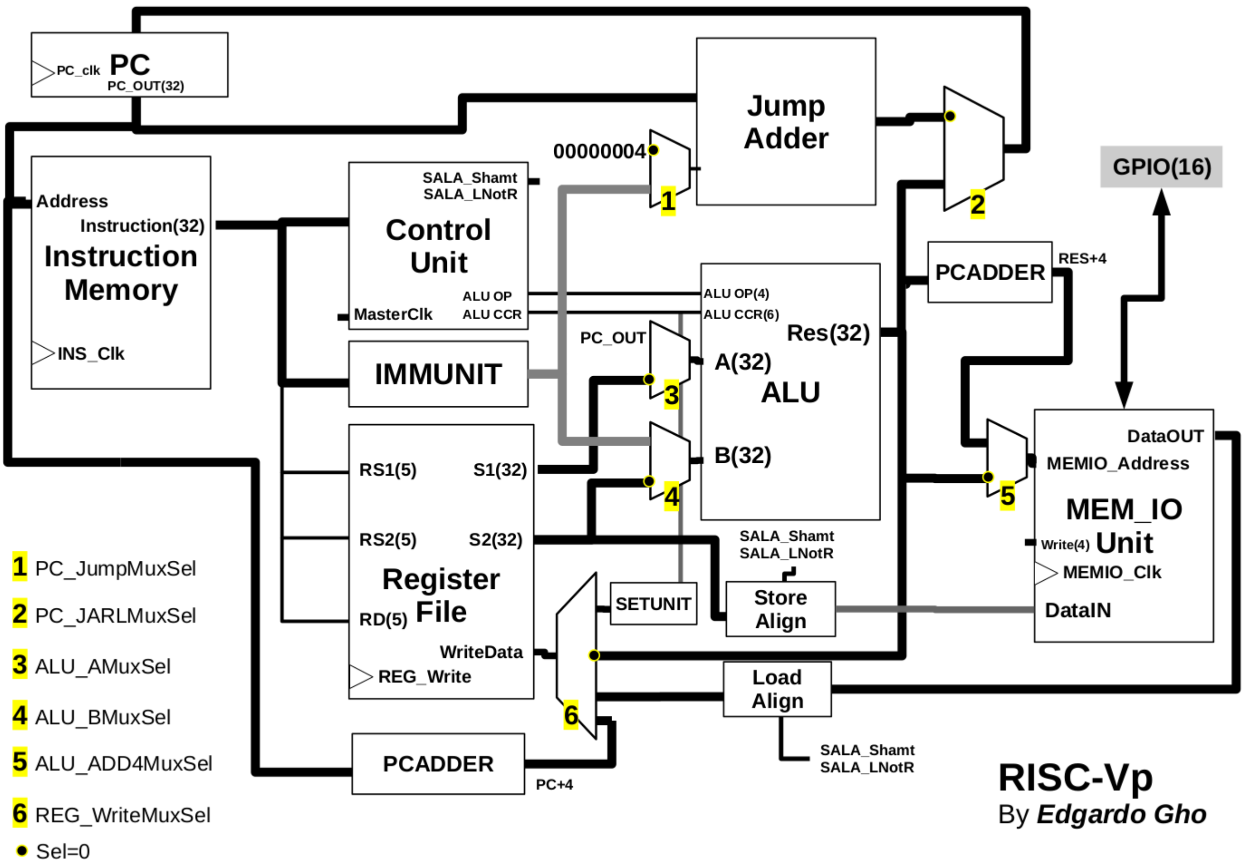


Figura 1 - Diagrama en bloques procesador RISC-Vp

En la figura 1 se ven los distintos bloques funcionales que componen el procesador diseñado. Cabe aclarar que la figura carece de algunas señales de control, las que han sido

obviadas para no recargar el diagrama con señales cuya función no brinda mayor claridad al mismo.
También es importante aclarar que se elige el idioma inglés para todo lo referente a la programación y el diseño, ya que el procesador se encuentra publicado y esto facilita futuros artículos y presentaciones en congresos que hagan referencia al mismo.
A continuación se realiza una descripción de cada uno de los bloques.

PC

El módulo PC se compone por un registro tipo Latch de 32 bits. No tiene major complejidad, solo almacena el valor del puntero a la memoria de instrucciones.

Entidad VHDL

```
entity PC is
   Port ( PC_IN : in STD_LOGIC_VECTOR(31 downto 0);
        PC_OUT :out STD_LOGIC_VECTOR(31 downto 0);
PC_Clock : in STD_LOGIC;
PC_Reset : in STD_LOGIC);
end PC;
```

InstructionMemory

Este módulo contiene una memoria BlockRAM utilizada como medio de almacenamiento de instrucciones, ya que si bien es una memoria RWM puede inicializarse (dentro de una FPGA) con valores determinados. La misma puede ser reemplazada por una memoria tipo ROM, pero a los fines del desarrollo y prototipado, la elección de la BlockRAM brinda más conveniencia.

Puede verse a continuación que se requiere una dirección de 32 bits (solo soporta accesos alineados, ya que las instrucciones soportadas son únicamente de 32 bits) y que cuando recibe un flanco ascendente en su línea de clock entrega la instrucción correspondiente a la posición de memoria solicitada. La capacidad mínima es de 1024 palabras (instrucciones), pero al estar implementada con BlockRAM puede expandirse en cascada hasta la capacidad soportada por el FPGA.

Entidad VHDL

```
entityInstructionMemis
   Port ( IM_Address : in STD_LOGIC_VECTOR (31 downto 0);
IM_Clock : in STD_LOGIC;
Instruction :out STD_LOGIC_VECTOR (31 downto 0));
endInstructionMem;
```

Control Unit

Sin lugar a dudas es el módulo más importante y complejo de todos. Su implementación consiste en una máquina de estados que lleva las distintas fases de la ejecución de la instrucción. Debido a su complejidad se brinda una tabla de transición de estados. La máquina de estados opera en cuatro ciclos de reloj para todas las instrucciones, excepto aquellas que realicen accesos a memoria (tanto para lectura como para escritura) que no se encuentren alineados al tamaño de palabra (4 bytes).
El reloj trabaja por flanco ascendente realizando una secuencia de 4 pulsos.

Tabla 1 - Ciclos para instrucciones alineadas

Primer Flanco	Segundo Flanco	Tercer Flanco	Cuarto Flanco
Fetch	**Decode/Execute**	**Mem**	**Write Back**
Se lee la instrucción apuntada por PC.	Se decodifica la instrucción, se selecciona la operación en la ALU y se avanza a un estado específico por instrucción.	Se realizan los accesos a memoria de datos tanto para escritura como lectura siempre y cuando estén alineados. Se seleccionan los MUX de saltos en caso de instrucciones tipo Branch.	Se hace el Write Back a un registro en el caso de una operación Aritmética/lógica o de lectura de memoria. En el caso de escritura de memoria o saltos condicionales se ignora el write back. En todos los casos se actualiza el PC para apuntar a la siguiente instrucción. No existen accesos a memoria de ningún tipo en este flanco.

Si bien existen formas de reducir la cantidad de pulsos para una gran cantidad de instrucciones, como por ejemplo las aritméticas, este procesador busca implementar una cantidad fija de ciclos de reloj (salvo el caso de accesos no alineados) y etapas marcadas con independencia entre memoria y write back.
Esto quiere decir que si bien se podría hacer el acceso a memoria (en una escritura) y evitar la fase de write back, se busca que todas las instrucciones se resuelvan de manera homogénea. El motivo fundamental de esta decisión de diseño es poder implementar a futuro un esquema de pipelining. Esto solo se puede realizar si las etapas tienen independencia entre sí.

Cabe destacar que en diseños que soportan pipelining como la arquitectura MIPS se utiliza esta misma técnica pero la fases de decodificación y ejecución se encuentran separadas. En el caso de RISC-Vp la ALU es enteramente combinacional, lo que permite juntar ambas etapas pero eso no quiere decir que no tengan que desdoblarse por cuestiones de implementación.

Tabla de transición de estados

RISC-Vp Control Unit FSM

	Inputs							Outputs																
Current State	Reset	FUNCT7	FUNC3	OPCODE	ALU_RES(2)	ALU_CCR(6)	Next State	ALU_AMuxSel	ALU_BMuxSel	ALU_OP(4)	ALU_ADD4MuxSel	INST_Clk	PC_JumpMuxSel	PC_JARLMuxSel	PC_Clk	REG_WriteMuxSel(2)	REG_WriteClk	MEMIO_Clk	MEMIO_Write(4)	LA_LatchWrite(4)	LA_LatchMuxSel	LA_SizeAdj(3)	SALA_LNotR	SALA_Shamt(2)
Hold	1						Hold	0	0	0000	0	0	0	0	0	0	0	0	0000	0000	0	000	0	00
Hold	0						Fetch	0	0	0000	0	0	0	0	0	0	0	0	0000	0000	0	000	0	00
Fetch	0						Decode					1												
Decode	0			0110111			LUI		1	1111		0												
Decode	0			0010111			AUIPC	1	1			0												
Decode	0			1101111			JAL					0	1			10								
Decode	0		000	1100111			JALR		1			0		1		10								
Decode	0		000	1100011			BEQ			1000		0												
Decode	0		001	1100011			BNE			1000		0												
Decode	0		100	1100011			BLT			1000		0												
Decode	0		101	1100011			BGE			1000		0												
Decode	0		110	1100011			BLTU			1000		0												
Decode	0		111	1100011			BGEU			1000		0												
Decode	0		000	0000011			LWAlignCheck		1			0				01								
Decode	0		001	0000011			LBAlignCheck		1			0				01								
Decode	0		010	0000011			LBUAlignCheck		1			0				01								
Decode	0		100	0000011			LHAlignCheck		1			0				01								
Decode	0		101	0000011			LHUAlignCheck		1			0				01								
Decode	0		000	0100011			SWAlignCheck		1			0							1111					
Decode	0		001	0100011			SBAlignCheck		1			0							0001					
Decode	0		010	0100011			SHAlignCheck		1			0							0011					

Only verified inputs are displayed on this FSM. Empty on an input means don't care for that particular state. Only forced outputs are displayed on each state. Empty on an output means it keeps its previous state (latched).

RISC-Vp Control Unit FSM

	Inputs							Outputs																
Current State	Reset	FUNCT7	FUNC3	OPCODE	ALU_RES(2)	ALU_CCR(6)	Next State	ALU_AMuxSel	ALU_BMuxSel	ALU_OP(4)	ALU_ADD4MuxSel	INST_Clk	PC_JumpMuxSel	PC_JARLMuxSel	PC_Clk	REG_WriteMuxSel(2)	REG_WriteClk	MEMIO_Clk	MEMIO_Write(4)	LA_LatchWrite(4)	LA_LatchMuxSel	LA_SizeAdj(3)	SALA_LNotR	SALA_Shamt(2)
Decode	0		000	0010011			ADDI		1			0												
Decode	0		010	0010011			SLTI		1	1000		0				11								
Decode	0		011	0010011			SLTIU		1	1000		0				11								
Decode	0		100	0010011			XORI		1	0100		0												
Decode	0		110	0010011			ORI		1	0110		0												
Decode	0		111	0010011			ANDI		1	0111		0												
Decode	0	0000000	001	0010011			SLLI		1	0001		0												
Decode	0	0000000	101	0010011			SRLI		1	0101		0												
Decode	0	0100000	101	0010011			SRAI		1	1101		0												
Decode	0	0000000	000	0110011			ADD					0												
Decode	0	0100000	000	0110011			SUB			1000		0												
Decode	0	0000000	001	0110011			SL_L			0001		0												
Decode	0	0000000	010	0110011			SLT			1000		0				11								
Decode	0	0000000	011	0110011			SLTU			1000		0				11								
Decode	0	0000000	100	0110011			XO_R			0100		0												
Decode	0	0000000	101	0110011			SR_L			0101		0												
Decode	0	0100000	101	0110011			SR_A			1101		0												
Decode	0	0000000	110	0110011			O_R			0110		0												
Decode	0	0000000	111	0110011			AN_D			0111		0												
LUI	0						WriteBack																	
AUIPC	0						WriteBack																	
JAL	0						WriteBack																	
JALR	0						WriteBack																	

RISC-Vp Control Unit FSM

	Inputs							Outputs																
Current State	Reset	FUNCT7	FUNC3	OPCODE	ALU_RES(2)	ALU_CCR(6)	Next State	ALU_AMuxSel	ALU_BMuxSel	ALU_OP(4)	ALU_ADD4MuxSel	INST_Clk	PC_JumpMuxSel	PC_JARLMuxSel	PC_Clk	REG_WriteMuxSel(2)	REG_WriteClk	MEMIO_Clk	MEMIO_Write(4)	LA_LatchWrite(4)	LA_LatchMuxSel	LA_SizeAdj(3)	SALA_LNotR	SALA_Shamt(2)
BEQ	0						Branch																	
BNE	0						Branch																	
BLT	0						Branch																	
BGE	0						Branch																	
BLTU	0						Branch																	
BGEU	0						Branch																	
Branch	0		000			100000	NopWriteBack						1											
Branch	0		001			010000	NopWriteBack						1											
Branch	0		100			001000	NopWriteBack						1											
Branch	0		101			000100	NopWriteBack						1											
Branch	0		110			000010	NopWriteBack						1											
Branch	0		111			000001	NopWriteBack						1											
Branch	0		---			000000	NopWriteBack						0											
ADDI	0						WriteBack																	
SLTI	0						WriteBack																	
SLTIU	0						WriteBack																	
XORI	0						WriteBack																	
ORI	0						WriteBack																	
ANDI	0						WriteBack																	
SLLI	0						WriteBack																	
SRLI	0						WriteBack																	
SRAI	0						WriteBack																	
ADD	0						WriteBack																	

RISC-Vp Control Unit FSM

	Inputs							Outputs																
Current State	Reset	FUNCT7	FUNC3	OPCODE	ALU_RES(2)	ALU_CCR(6)	Next State	ALU_AMuxSel	ALU_BMuxSel	ALU_OP(4)	ALU_ADD4MuxSel	INST_Clk	PC_JumpMuxSel	PC_JARLMuxSel	PC_Clk	REG_WriteMuxSel(2)	REG_WriteClk	MEMIO_Clk	MEMIO_Write(4)	LA_LatchWrite(4)	LA_LatchMuxSel	LA_SizeAdj(3)	SALA_LNotR	SALA_Shamt(2)
SUB	0						WriteBack																	
SL_L	0						WriteBack																	
SLT	0						WriteBack																	
SLTU	0						WriteBack																	
XO_R	0						WriteBack																	
SR_L	0						WriteBack																	
SR_A	0						WriteBack																	
O_R	0						WriteBack																	
AN_D	0						WriteBack																	
LBAlignCheck	0				00		WriteBack											1				001	0	00
LBAlignCheck					01		WriteBack											1				001	0	01
LBAlignCheck					10		WriteBack											1				001	0	10
LBAlignCheck					11		WriteBack											1				001	0	11
LBUAlignCheck					00		WriteBack											1				101	0	00
LBUAlignCheck					01		WriteBack											1				101	0	01
LBUAlignCheck					10		WriteBack											1				101	0	10
LBUAlignCheck					11		WriteBack											1				101	0	11
LHAlignCheck					00		WriteBack											1				010	0	00
LHAlignCheck					01		WriteBack											1				010	0	01
LHAlignCheck					10		WriteBack											1				010	0	10
LHAlignCheck					11		LHAlignFix											1				010	0	11
LHAlignFix							LHAlignPenality				1							0		0001	1	010	0	11
LHAlignPenality							LHAlignFinish											1		0000			1	01

RISC-Vp Control Unit FSM																								
	Inputs							Outputs																
Current State	Reset	FUNCT7	FUNC3	OPCODE	ALU_RES(2)	ALU_CCR(6)	Next State	ALU_AMuxSel	ALU_BMuxSel	ALU_OP(4)	ALU_ADD4MuxSel	INST_Clk	PC_JumpMuxSel	PC_JARLMuxSel	PC_Clk	REG_WriteMuxSel(2)	REG_WriteClk	MEMIO_Clk	MEMIO_Write(4)	LA_LatchWrite(4)	LA_LatchMuxSel	LA_SizeAdj(3)	SALA_LNotR	SALA_Shamt(2)
LHAlignFinish							WriteBack											0		0010			1	
LHUAlignCheck					00		WriteBack											1				110	0	00
LHUAlignCheck					01		WriteBack											1				110	0	01
LHUAlignCheck					10		WriteBack											1				110	0	10
LHUAlignCheck					11		LHAlignFix											1				110	0	11
LHUAlignFix							LHUAlignPenality				1							0		0001	1	110	0	11
LHUAlignPenality							LHUAlignFinish											1		0000			1	01
LHUAlignFinish							WriteBack											0		0010			1	
LWAlignCheck					00		WriteBack											1					0	00
LWAlignCheck					01		LWAlignFix											1					0	01
LWAlignCheck					10		LWAlignFix											1					0	10
LWAlignCheck					11		LWAlignFix											1					0	11
LWAlignFix					01		LWAlignPenalty				1							0		0111				01
LWAlignFix					10		LWAlignPenalty				1							0		0011				10
LWAlignFix					11		LWAlignPenalty				1							0		0001				11
LWAlignPenalty					01		LWAlignFinish											1		0000			1	11
LWAlignPenalty					10		LWAlignFinish											1		0000			1	10
LWAlignPenalty					11		LWAlignFinish											1		0000			1	01
LWAlignFinish					01		WriteBack											0		1000				
LWAlignFinish					10		WriteBack											0		1100				
LWAlignFinish					11		WriteBack											0		1110				
SBAlignCheck					00		NopWriteBack											1	0001				1	00
SBAlignCheck					01		SaveRam												0010				1	01

RISC-Vp Control Unit FSM																								
	Inputs							Outputs																
Current State	Reset	FUNCT7	FUNC3	OPCODE	ALU_RES(2)	ALU_CCR(6)	Next State	ALU_AMuxSel	ALU_BMuxSel	ALU_OP(4)	ALU_ADD4MuxSel	INST_Clk	PC_JumpMuxSel	PC_JARLMuxSel	PC_Clk	REG_WriteMuxSel(2)	REG_WriteClk	MEMIO_Clk	MEMIO_Write(4)	LA_LatchWrite(4)	LA_LatchMuxSel	LA_SizeAdj(3)	SALA_LNotR	SALA_Shamt(2)
SBAlignCheck					10		SaveRam												0100				1	10
SBAlignCheck					11		SaveRam												1000				1	11
SHAlignCheck					00		NopWriteBack											1	0011				1	00
SHAlignCheck					01		SaveRam												0110				1	01
SHAlignCheck					10		SaveRam												1110				1	10
SHAlignCheck					11		SHAlignFix												1000				1	11
SHAlignFix							SHAlignPenalty											1						
SHAlignPenalty							SHAlignFinish				1													
SHAlignFinish							SaveRam				1							0	0001				0	01
SWAlignCheck					00		NopWriteBack												1111					
SWAlignCheck					01		SWAlignFix												1110				1	01
SWAlignCheck					10		SWAlignFix												1100				1	10
SWAlignCheck					11		SWAlignFix												1000				1	11
SWAlignFix							SWAlignPenalty											1						
SWAlignPenalty							SWAlignFinish				1													
SWAlignFinish					01		SaveRam				1							0	0001				0	11
SWAlignFinish					10		SaveRam				1							0	0011				0	10
SWAlignFinish					11		SaveRam				1							0	0111				0	01
WriteBack							Fetch								1	1								
SaveRam							NopWriteBack											1						
NopWriteBack							Fetch								1									

Entidad VHDL

```
entityControlUnitis
   Port ( Instruction : in STD_LOGIC_VECTOR (31 downto 0);
Reset : in std_logic;
MasterClock : in std_logic;
        ALU_CCR: in STD_LOGIC_VECTOR(5 downto 0);
        ALU_RES: in STD_LOGIC_VECTOR(1 downto 0);
ALU_AMuxSel :outstd_logic;
ALU_BMuxSel :outstd_logic;
        ALU_OP: out STD_LOGIC_VECTOR (3 downto 0);
        ALU_ADD4MuxSel :out STD_LOGIC;
INST_Clk :outstd_logic;
PC_JumpMuxSel :out STD_LOGIC;
PC_JARLMuxSel :out STD_LOGIC;
PC_Clk :out STD_LOGIC;
REG_WriteMuxSel :out STD_LOGIC_VECTOR(1 downto 0);
REG_WriteClk: outstd_logic;
MEMIO_Clk :outstd_logic;
MEMIO_Write :outstd_logic_vector(3 downto 0);
LA_LatchWrite :outstd_logic_Vector(3 downto 0);
LA_LatchMuxSel :outstd_logic;
LA_SizeAdj :outstd_logic_vector(2 downto 0);
SALA_LNotR: outstd_logic;
SALA_Shamt :outstd_logic_vector(1 downto 0)
);
endControlUnit;
```

IMMUNIT

Este módulo es completamente combinacional. Se encarga de generar el valor inmediato que acompaña a las distintas instrucciones. Dado que cada tipo de instrucción almacena los valores inmediatos de diversa forma, el módulo recibe toda la instrucción y luego de decodificar el tipo de la misma genera la secuencia correcta en forma combinatoria construyendo el número de 32 bits correspondiente.

Entidad VHDL

```
entity IMMUNIT is
   Port ( INSTRUCTION : in STD_LOGIC_VECTOR (31 downto 0);
```

```
IMMEDIATE :out STD_LOGIC_VECTOR (31 downto 0));
end IMMUNIT;
```

Register File

Este módulo es equivalente a casi todos los modelos homónimos en otras arquitecturas RISC. Posee 32 registros de 32 bits, los cuales pueden leerse de a dos (direccionados por dos puertos de 5 bits) y escritos de a uno (también con un puerto de 5 bits). El registro 0 tiene la particularidad de leerse siempre en 0 y no generar ningún cambio en la escritura.

Entidad VHDL

```
entityRegisterFileis
   Port ( REG_RS1 : in STD_LOGIC_VECTOR (4 downto 0);
          REG_RS2 : in STD_LOGIC_VECTOR (4 downto 0);
          REG_RD : in STD_LOGIC_VECTOR (4 downto 0);
          REG_S1 :out STD_LOGIC_VECTOR (31 downto 0);
          REG_S2 :out STD_LOGIC_VECTOR (31 downto 0);
REG_Data : in STD_LOGIC_VECTOR (31 downto 0);
reset : in std_logic;
REG_WriteClk : in STD_LOGIC);
endRegisterFile;
```

PCAdder

Este módulo es un sumador que suma el valor fijo 4 a su entrada de 32 bits. Dado que en RISC-V el direccionamiento se realiza al byte pero las palabras son de 32 bits, es necesario incrementar en 4 el PC con cada instrucción. Es un simple sumador combinacional. Debido a que se diseñó teniendo en cuenta el uso de FPGAs con soporte de slices DSP, se implementó este sumador como parte del IP Adder/Sub con suma fija. Esto permite aprovechar los slices DSP dentro del FPGA.

Entidad VHDL

```
ENTITY PCADDER IS
  PORT (
A : IN STD_LOGIC_VECTOR(31 DOWNTO 0);
S : OUT STD_LOGIC_VECTOR(31 DOWNTO 0)
);
END PCADDER;
```

JumpAdder

Es un simple sumador de 32 bits. Se incorpora este sumador adicional a la ALU con el fin de resolver valores del PC en saltos. No reviste mayor complejidad.

Entidad VHDL

```
ENTITY JumpAdder IS
  PORT (
A : IN STD_LOGIC_VECTOR(31 DOWNTO 0);
B : IN STD_LOGIC_VECTOR(31 DOWNTO 0);
S : OUT STD_LOGIC_VECTOR(31 DOWNTO 0)
);
END JumpAdder;
```

ALU

La unidad aritmético lógica es enteramente combinacional. Como entrada tiene los operandos A y B junto a 4 líneas de selección de operación. Soporta sumas, restas, desplazamientos a derecha (lógicos y aritméticos), desplazamientos a izquierda y operaciones lógicas AND, OR y XOR. Ofrece como salida el resultado de la operación junto a 6 líneas de condición de estado (funcionales en la resta) indicando EQ, NE, LT, GE, LTU y GEU.

La implementación del sumador/restador se hizo utilizando el núcleo IP de Xilinx correspondiente implementado en el DSP48 incluido en la familia Xilinx Artix-7. Esto quiere decir que no consume recursos de lógica dentro del FPGA, lo que lo hace muy práctico.

Los desplazamientos se realizan con BarrelShifters, por lo que son totalmente combinacionales y no consumen ciclos de reloj.

Entidad VHDL

```
ComponentAdderSub
port( A : IN STD_LOGIC_VECTOR(31 DOWNTO 0);
B : IN STD_LOGIC_VECTOR(31 DOWNTO 0);
ADD : IN STD_LOGIC;
   C_IN : IN STD_LOGIC;
   C_OUT : OUT STD_LOGIC;
S : OUT STD_LOGIC_VECTOR(31 DOWNTO 0) );
endcomponent;
```

SetUnIt

Este módulo evalúa el resultado de la ALU y genera un 1 o 0 en 32 bits dependiendo del resultado de LT y LTU. Sirve como auxiliar a las instrucciones que cargan un registro dependiendo de las comparaciones LT y LTU.

Entidad VHDL

```
entitySetUnitis
   Port ( SU_lt : in STD_LOGIC;
SU_ltu : in STD_LOGIC;
SU_unsig : in stD_logic; --ConnectedtoInstruction(12) sinceltu has Instruction(12)=1
SU_Data :out STD_LOGIC_VECTOR (31 downto 0));
endSetUnit;
```

StoreAlign

Este módulo se encarga de alinear los datos tomados desde R2 para accesos a memoria o entrada salida desalineados. A tal fin utiliza BarrelShifters para desplazar el valor de R2, de forma que quede alineado a nivel byte con aquellos bytes que vayan a ser escritos. En resumen, es un desplazador configurable.

Su valor por defecto lo hace transparente en el caso que el acceso a memoria se realice de forma alineada.

Entidad VHDL

```
entityStoreAlignmentUnitis
   Port ( SA_UnalignedData : in STD_LOGIC_VECTOR (31 downto 0);
SA_AlignedData :out STD_LOGIC_VECTOR (31 downto 0);
SA_LNotR : in STD_LOGIC;
SA_Shamt : in STD_LOGIC_VECTOR (1 downto 0));
endStoreAlignmentUnit;
```

LoadAlign

El módulo de load alignment es más complejo que su contraparte para los stores. Se encarga de alinear los datos provenientes de accesos desalineados en el caso que estos existan. A tal fin utiliza no solo desplazadores sino un registro Latch auxiliar que almacena los bytes desalineados temporalmente mientras se accede a la palabra contigua en busca de los bytes faltantes. En el caso de lecturas alineadas, este módulo soporta un paso transparente del valor de entrada, evitando así la generación de ciclos adicionales.

Entidad VHDL

```
entityLoadAlignmentUnitis
   Port ( LA_AlignedData : out STD_LOGIC_VECTOR (31 downto 0);
LA_UnalignedData : in STD_LOGIC_VECTOR (31 downto 0);
LA_Shamt : in STD_LOGIC_VECTOR (1 downto 0);
LA_LNotR : in STD_LOGIC;
LA_LatchWrite : in STD_LOGIC_VECTOR (3 downto 0);
LA_LatchMuxSel : in STD_LOGIC;
LA_SizeAdj : in STD_LOGIC_VECTOR (2 downto 0);
reset : in STD_LOGIC);
endLoadAlignmentUnit;
```

MEMIOUnit

Este módulo concentra la memoria de datos (un BLOCKRAM que ofrece 1024 palabras ubicadas a partir de la posición 0x1000000) y los registros de entrada y salida implementando un puerto GPIO de 8 bits independientes, o sea 16 bits en total (cada uno con una dirección fija asignada). Si bien es un periférico muy básico, cumple con el alcance de tener comunicación con el mundo exterior.

Es importante aclarar que se dispone de una línea independiente para la escritura de cada byte direccionado por las líneas de dirección.

Gracias a esto, en conjunto con la unidad de StoreAlignment se pueden realizar con facilidad escrituras desalineadas.

Entidad VHDL

```
entity MEMIOUNIT is
   Port ( MEMIO_DataIn : in STD_LOGIC_VECTOR (31 downto 0);
MEMIO_DataOut :out STD_LOGIC_VECTOR (31 downto 0);
MEMIO_Clk : in STD_LOGIC;
MEMIO_Reset : in STD_LOGIC;
MEMIO_Address : in STD_LOGIC_VECTOR (31 downto 0);
        MEMIO_GPI : in STD_LOGIC_VECTOR (7 downto 0);
        MEMIO_GPO :out STD_LOGIC_VECTOR (7 downto 0);
MEMIO_Write : in STD_LOGIC_VECTOR (3 downto 0));
end MEMIOUNIT;
```

Casos de prueba

Con el fin de realizar pruebas sobre el procesador se escribieron programas diversos que intentan utilizar los diversos módulos con el fin de tener una gran cobertura.

Programa Ejemplo

```
# Programa de prueba RISC-Vp
#Seccion Datos, almacenada en 0x1000000
.data
VarA:   .word
VarB:   .word
VarC:   .word

#Seccion Programa, almacenado en 0x00000000
.text
        la      x2,VarA      #X2 guarda inicio de RAM
        li      x3,0x00000000  #X3 lleva los flags de OK
        li      x4,0x12345678  #X4 numero fijo 0x12345678
        add     x5,x4,x0     #X5 = X4
        #Prueba de branchifequal
        beq     x5,x4,BEQOK


ERROR:
        beq     x0,x0,ERROR

FIN:
        jal     x1,FIN

BEQOK:
        #Aqui el salto BEQ funciono bien...
        #Guardamos 1 en x3
        slli    x3,x3,1
        ori     x3,x3,1

        #Probamos salto por distinto
        bne     x3,x4,BNEOK
```

```
        beq     x0,x0,ERROR


BNEOK:
        #Aqui el salto a BNE funciono bien...
        #Otro flag
        slli    x3,x3,1
        ori     x3,x3,1

        #Probamos el salto BLT
        blt     x3,x4,BLTOK
        beq     x0,x0, ERROR


BLTOK:
        #Aqui el salto a BNE funciono bien...
        #Otro flag
        slli    x3,x3,1
        ori     x3,x3,1

        bge     x4,x3,BGEOK
        beq     x0,x0,ERROR

BGEOK:
        #Aqui el salto a BNE funciono bien...
        #Otro flag
        slli    x3,x3,1
        ori     x3,x3,1

        #Oparitméticas varias
        add     x6,x4,x3
        sub     x7,x4,x6
        xori    x8,x3,0x7FF
        andi    x9,x8,0x7FF
        lui     x10,0x80000
        addi    x10,x10,0
        srai    x10,x10,1
        srai    x10,x10,17
        srli    x10,x10,8

        jal     x1,SWTEST
        jal     x1,LWTEST

        jal     x1,FIN
```

```
SWTEST:
        #Store alineado
        lui     x11,0x11223
        addi    x11,x11,0x344
        lui     x12,0x55667
        addi    x12,x12,0x788

        sw      x11,0(x2)
        sw      x12,4(x2)

        #Store desalineado
        sw      x11,9(x2) #deberia tener penalidad
        srli    x13,x12,16
        sh      x13,13(x2)
        srli    x14,x12,8
        sb      x14,15(x2)
        sb      x12,16(x2) #Alineado de nuevo
        jalr    x1



LWTEST:
        lw      x15,0(x2) #Alineado
        lw      x16,1(x2) #No Alineado
        lh      x17,1(x2) #no alineado 1 penalidad
        lh      x18,3(x2) #no alineado 2 penalidad
        lh      x19,0(x2) #Alineado
        lb      x20,0(x2) #Alineado
        lb      x21,1(x2) #No alineado

        jalr    x1

#######################
```

Los programas escritos en ASM para RISC-V requieren ser ensamblados, para obtener como resultado un archivo binario. A los fines de este proyecto se utilizó el Simulador/Ensamblador Ripes. Éste genera un archivo binario (.bin) durante el ensamblado. Este archivo es simplemente una secuencia de bytes que forman palabras (little-endian) representando las instrucciones. Para poder ejecutar estos programas esta secuencia debe transformarse a un formato compatible con la inicialización de la BLOCKRAM. A tal fin se llevó a cabo el desarrollo de un pequeño y sencillo programa en C que toma como entrada un archivo binario e imprime en pantalla la secuencia de inicialización de la BLOCKRAM.

ArchivoBlockRamConv.c

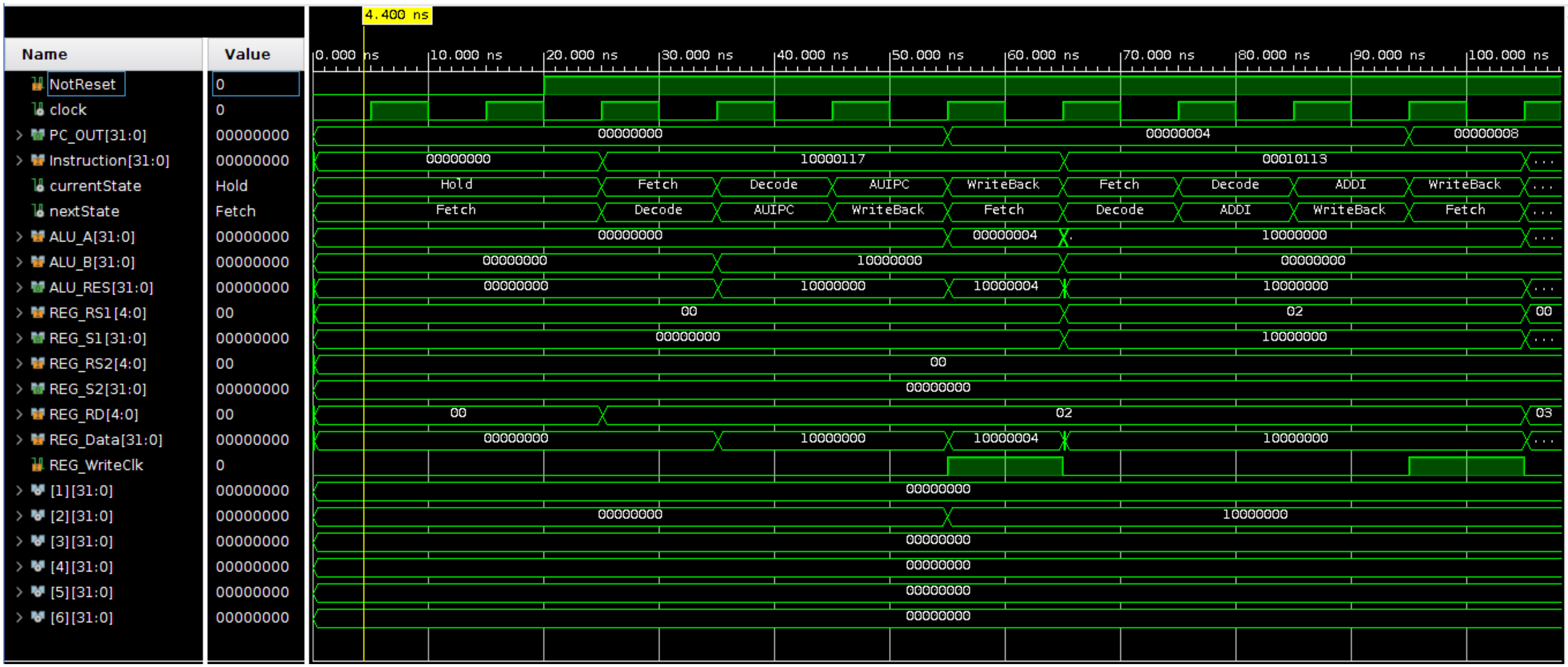
```
#include<stdio.h>
#include<stdlib.h>
#include<stdint.h>

//BlockRaminitializeslines of 8wordsx8bytes each.
#define LINESIZE 8*8

//Thisprogramreads a binary files and generatesthe Xilinx BlockRAM
//expected INIT_XX => X".....", line structure.
intmain(intargc, char *argv[]){
	FILE *readFile;
	uint8_t lineCounter=0; //Keeptrack of line
	uint8_t hexCounter=0; //Keeptrack of hexinline
	uint8_t line[LINESIZE+1]; //Buffer for line, add 1 for \n
	uint8_t hex[2+1]; //Buffer toconvert byte tohex, add1 for \n
	uint8_t aux; //Buffer toreadfrom file (single byte)

	//Checkforproperarguments (basically a file name)
	if (argc!=2){
		printf("Error: use %s PATH_TO_FILE\n",argv[0]);
		exit(1);
	}
	readFile = fopen(argv[1],"r");
	if (readFile==NULL){
		printf("Error opening %s\n",argv[1]);
		exit(1);
	}

	//Init line withall 0s
	sprintf((char*)line,"0000000000000000000000000000000000000000000000000000000
000000000");

	//Loopreading byte after byte from file
	while(!feof(readFile)){
		//Read a byte
		fread(&aux,sizeof(uint8_t),1,readFile);
		sprintf((char*)hex,"%02X",aux); //Quick and dirty byte toHexusingprintf
```

```
			//Nowsave 2 hexon line
			line[LINESIZE-1-hexCounter]=hex[1];
			line[LINESIZE-2-hexCounter]=hex[0];
			hexCounter+=2;

			//Ifitreachesend of line...
			if (hexCounter>(LINESIZE-1)){
				//Convert line counteralsotohex
				sprintf(hex,"%02X",lineCounter);
				//Nowprintthe line in Xilinx BLOCKRAM format
				printf("\tINIT_%c%c => X\"%s\",\n",hex[0],hex[1],line);
				//Increase line counter and resethexcounter and line
				lineCounter++;
				hexCounter=0;

	sprintf((char*)line,"0000000000000000000000000000000000000000000000000000
000000000");
			}
	}
	//Thereisalwaysonelast line toprint, evenifitsalljust 0s
	sprintf(hex,"%02X",lineCounter);
	printf("\tINIT_%c%c => X\"%s\",\n",hex[0],hex[1],line);

	//So long
	return 0;
}
```

Análisis de señales

Se proveen a continuación imágenes representando diversos estados de la ejecución del programa de prueba con un detalle de cómo cambian las señales y una breve descripción.

Operaciones Inmediatas

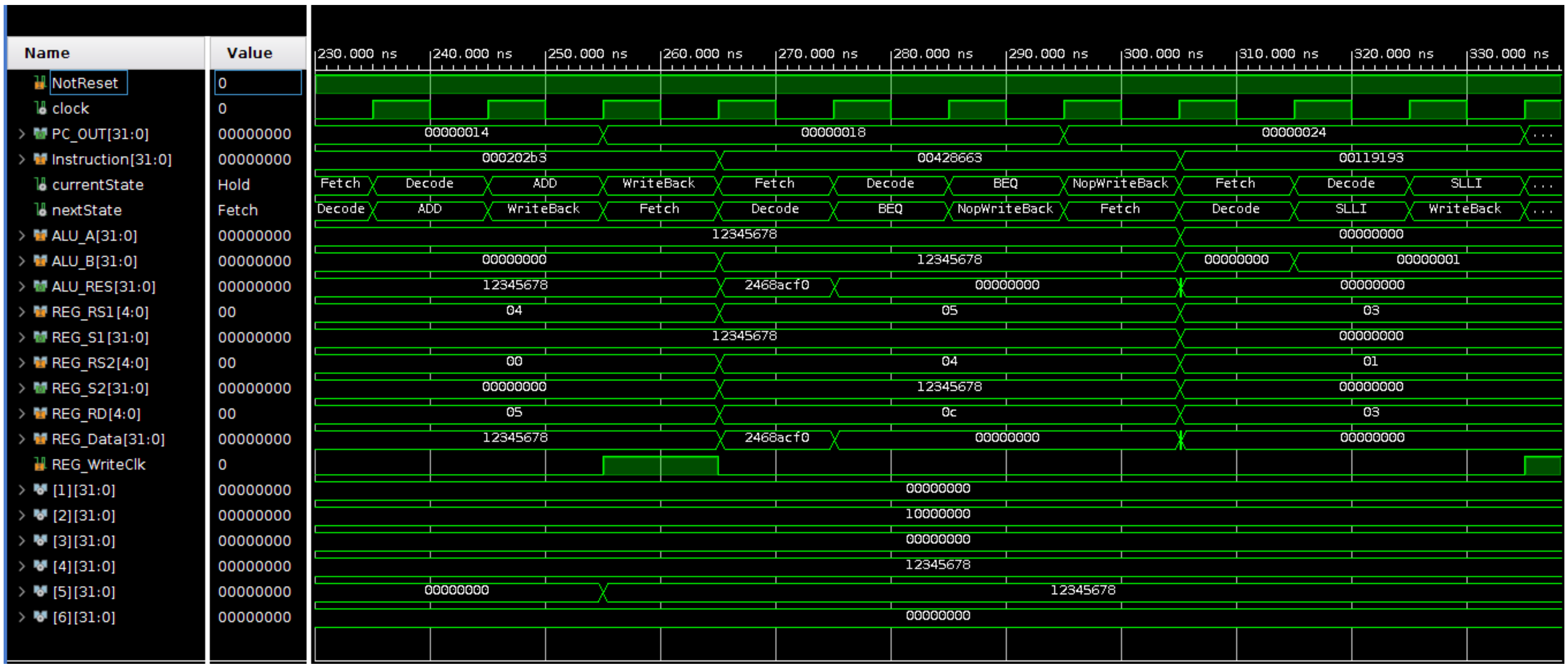


El diagrama de tiempos muestra la ejecución de:

0:	10000117	auipc x2 0x10000
4:	00010113	addi x2 x2 0

Se puede ver en el flanco de 20ns (luego de 2 pulsos de clock operando a 100Mhz) que la señal de NotReset pasa de 0 a , quitando al procesador de su estado “Hold” y pasando a “Fetch”. Nótese que en este cambio el valor de instrucción pasa a ser 0x10000117. Se ve claramente cómo esta instrucción se resuelve en 4 fases (Fetch, Decode, AUIPC, WriteBack), incrementando el PC_OUT con el pulso de WriteBack para que el mismo apunte a la siguiente instrucción. Esto se respetará en todas las instrucciones, excepto en aquellas que accedan a memoria de forma desalineada. Se ve que a continuación ejecuta addi también en 4 pulsos de reloj. Este conjunto de instrucciones (auipc+addi) sobre el mismo registro corresponden a la pseudo-instrucción la x2,0x10000000 la cual utiliza las dos instrucciones mencionadas para almacenar una dirección de memoria de datos completa de 32 bits en un registro (x2).

Saltos condicionales

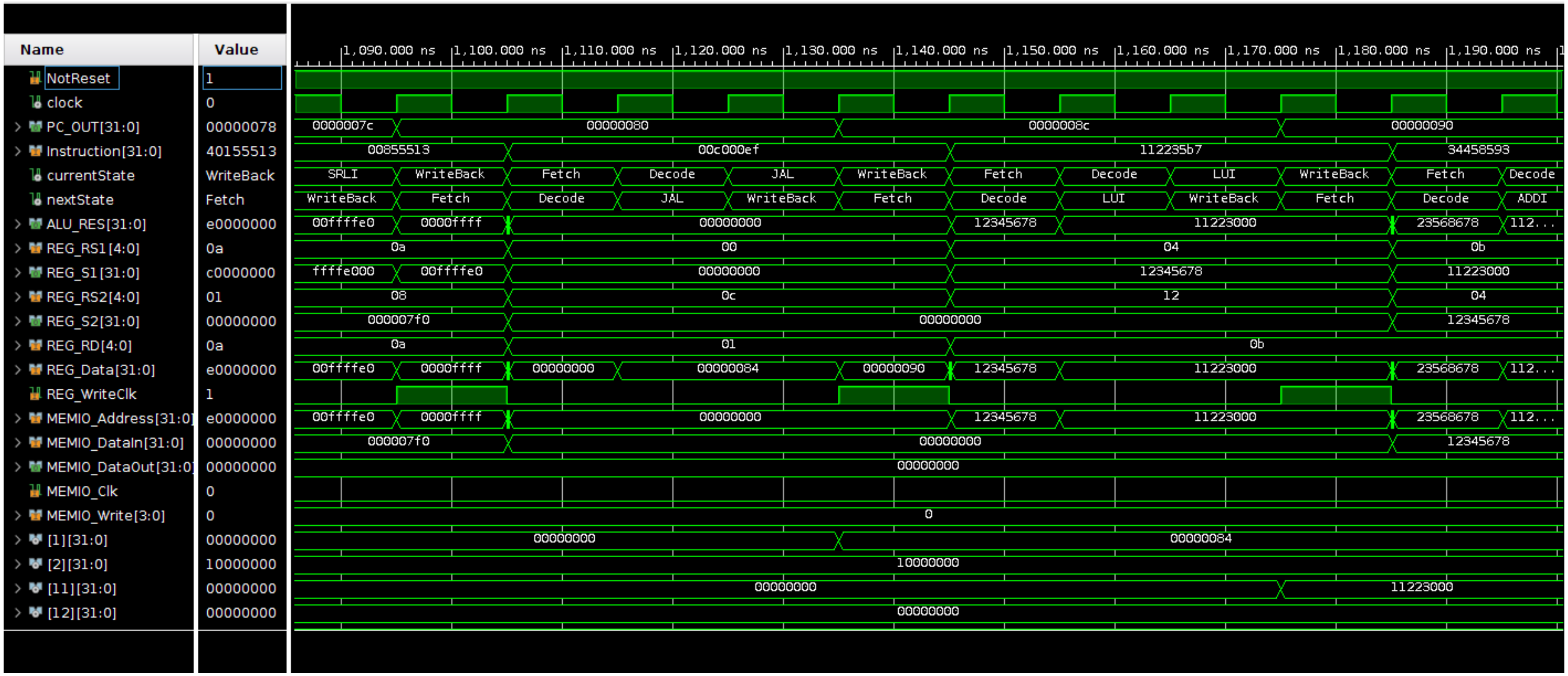


El diagrama de tiempos muestra la ejecución de:

	14:	000202b3	add x5 x4 x0
	18:	00428663	beq x5 x4 12 <BEQOK>
ERROR	1c:	00000063	beq x0 x0 0 <ERROR>
FIN	20:	000000ef	jal x1 0x20 <FIN>
BEQOK	24:	00119193	slli x3 x3 1

La instrucción BEQ x5,x4,12 compara el contenido de x5 y x4 (restando en la ALU y mirando el flag EQ del CCR), y en caso de ser iguales adiciona un valor inmediato (en este caso 12) al PC. Vemos que efectivamente x5 y x4 tienen el mismo valor (0x12345678) y por ende se cambia el PC a 0x18+0x0C=0x24, con lo que se realiza efectivamente el salto condicional. Nótese que el último estado de la instrucción es NopWriteBack, ya que no se modifica ningún registro (salvo el PC). Toda la instrucción consume, como es de esperar, cuatro ciclos de reloj.

Saltos incondicionales

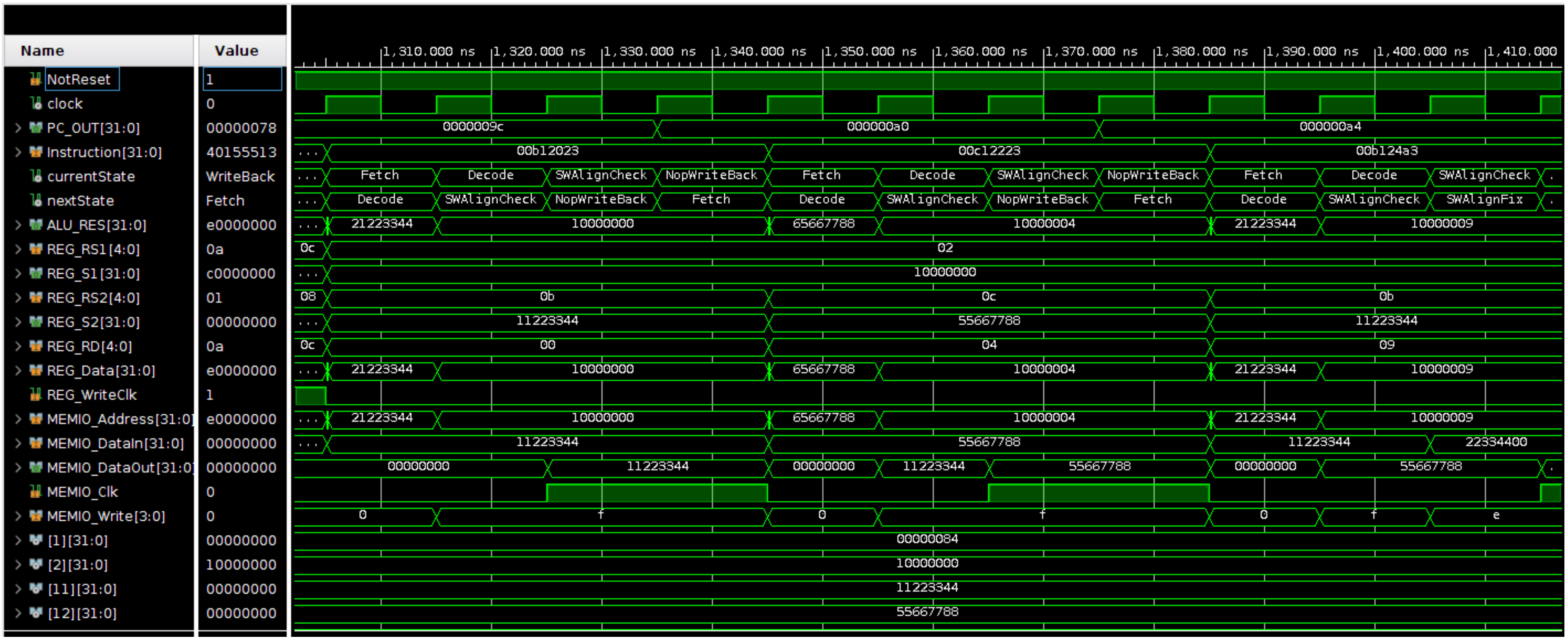


El diagrama de tiempos muestra la ejecución de:

	80:	00c000ef	jal x1 0x8c <SWTEST>
	84:	03c000ef	jal x1 0xc0 <LWTEST>
	88:	f99ff0ef	jal x1 0x20 <FIN>
SWTEST	8c:	112235b7	lui x11 0x11223
	90:	34458593	addi x11 x11 836

El salto JAL realiza un salto incondicional sumando un valor inmediato al PC. En este caso se le suma 0x0C codificado en la instrucción. Eso brinda como resultado que el PC apunte a 0x8C. Nótese que la dirección PC+4 (original 0x84) queda almacenada en x1. Esto cumple con la parte de link de la instrucción Jump And Link. Esto asegura que pueda retornar a 0x84 tomando la dirección de x1 en un futuro salto JALR.

Escrituras de memoria



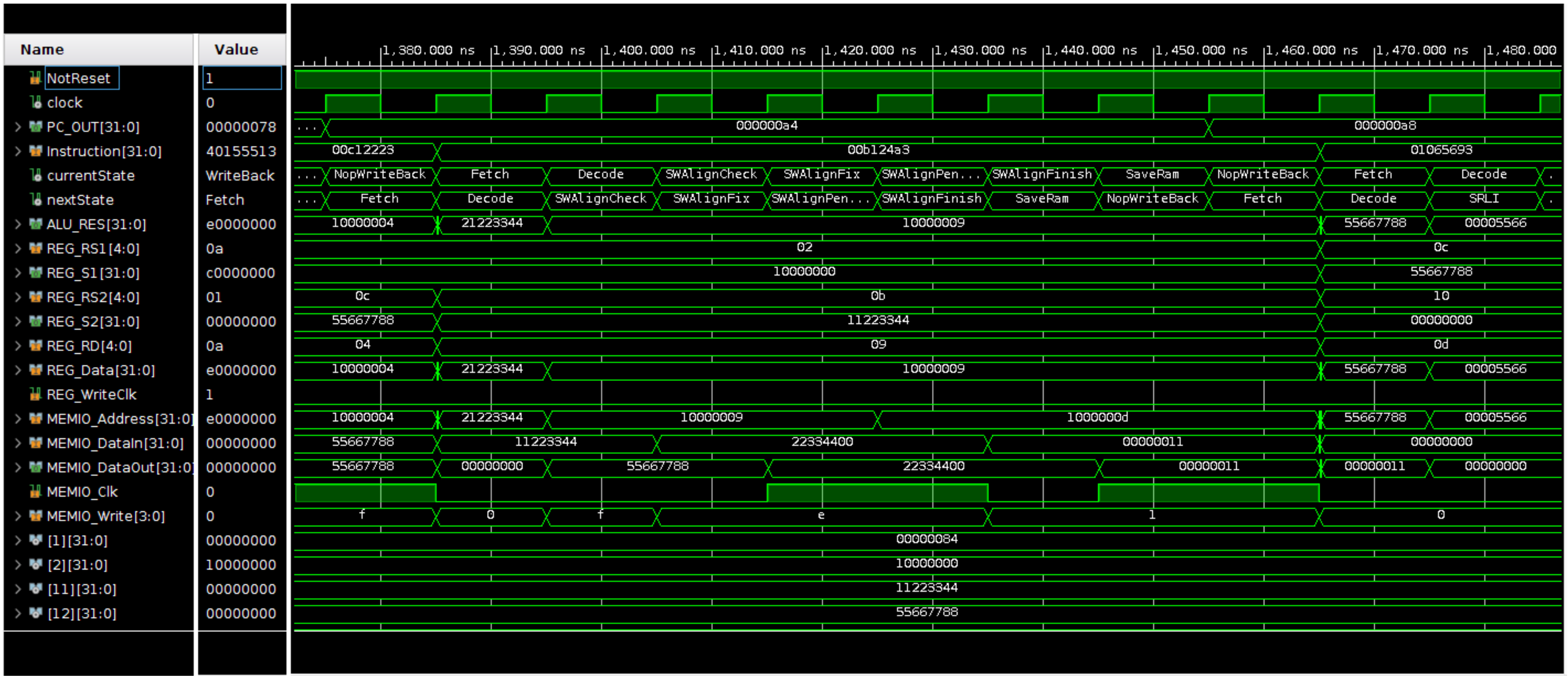
El diagrama de tiempos muestra la ejecución de:

9c: 00b12023 sw x11 0(x2)

a0: 00c12223 sw x12 4(x2)

Se ve que ambas escrituras se encuentran alineadas a la palabra de 4 bytes. En ninguno de los dos casos existe penalidad de tiempos, ya que al encontrarse alineadas las escrituras el módulo de StoreAlignment no debe hacer ningún desplazamiento. Se ven las lineas de escrituras MEMIO_Write con sus 4 lineas en 1 (valor F hexadecimal) al momento de generar el pulso de escritura (MEMIO_Clk).

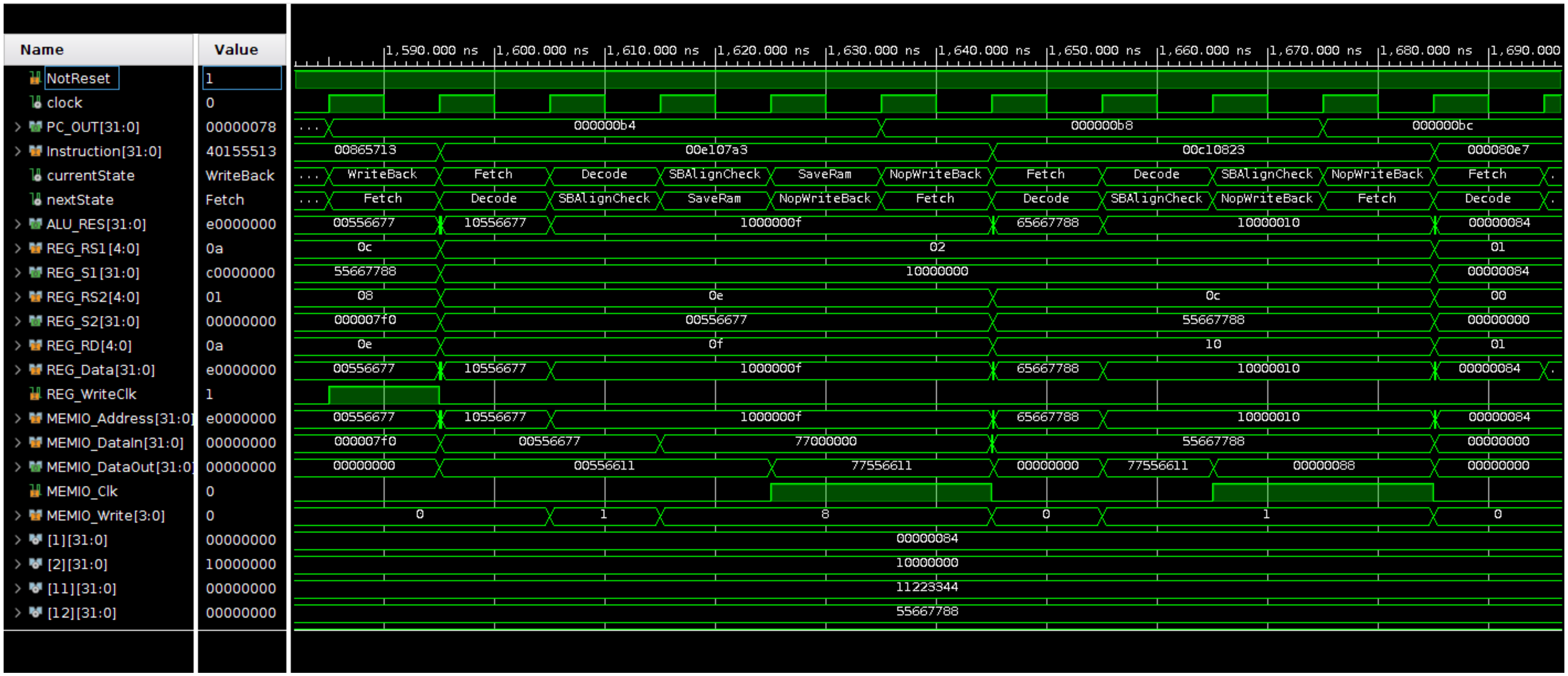
Escritura penalizada



En el diagrama de tiempos se ejecuta:

a4: 00b124a3 sw x11 9(x2)

Se ve claramente que esta escritura es desalineada, ya que el offset 9 sobre 0x1000000 apunta a la dirección 0x10000009. Los últimos 2 bits de la misma corresponden a 01, por ende se deben grabar los 3 bytes menos significativos del registro x11 a partir del segundo byte de la posición de memoria 0x10000008. Luego el byte más significativo de x11 debe grabarse en el byte menos significativo de la posición de memoria 0x1000000D. Nótese que la cantidad de pulsos de reloj es de 8. Esto se debe a que es necesario hacer dos accesos a la memoria de datos (0x10000008 y 0x1000000D). En el primer acceso es importante destacar el valor de MEMIO_Write=e (1110) grabando los 3 bytes más significativos de 0x10000008 y luego MEMIO_Write=1 (0001) grabando el byte faltante en 0x1000000D.

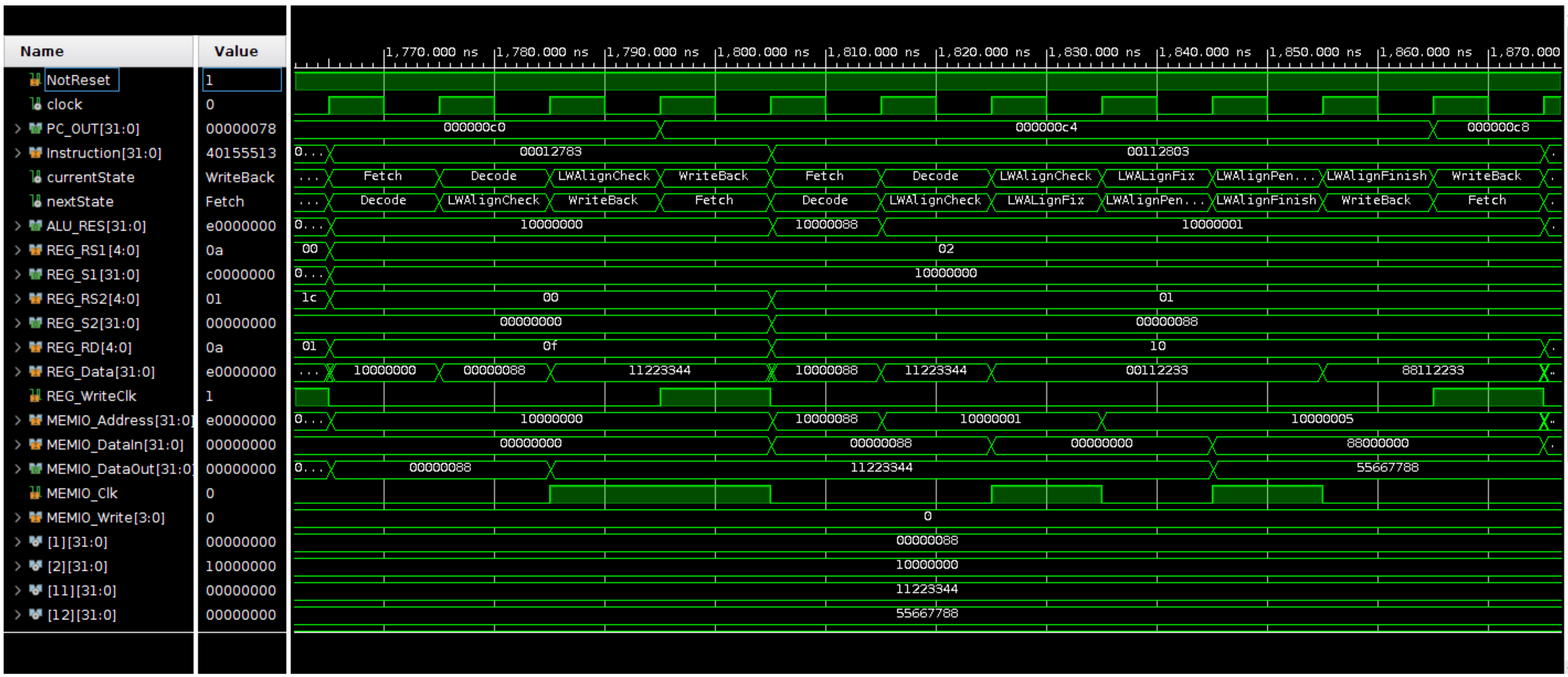


En la imagen se muestra la ejecución de:

b4:	00e107a3	sb x14 15(x2)
b8:	00c10823	sb x12 16(x2)

Ambas escrituras se encuentran desalineadas. Debido a esto, se requiere un pulso extra de reloj para esperar que el StoreAlignment pueda acomodar (desplazando) el byte de la forma necesaria para ser escrito en la memoria. Cabe aclarar que este caso en particular podría salvarse utilizando una lógica combinacional en el StoreAlignment que pueda calcular qué tan desalineado se encuentra el acceso a memoria. En el caso de RISC-Vp esto se resuelve secuencialmente en la máquina de estados de la unidad de control, motivo por el cual se paga una penalidad de un ciclo de reloj en este tipo de accesos desalineados. Queda claro que se desaconsejan siempre los accesos desalineados a memoria, en el caso de ser posible evitarlos.

Lecturas penalizadas

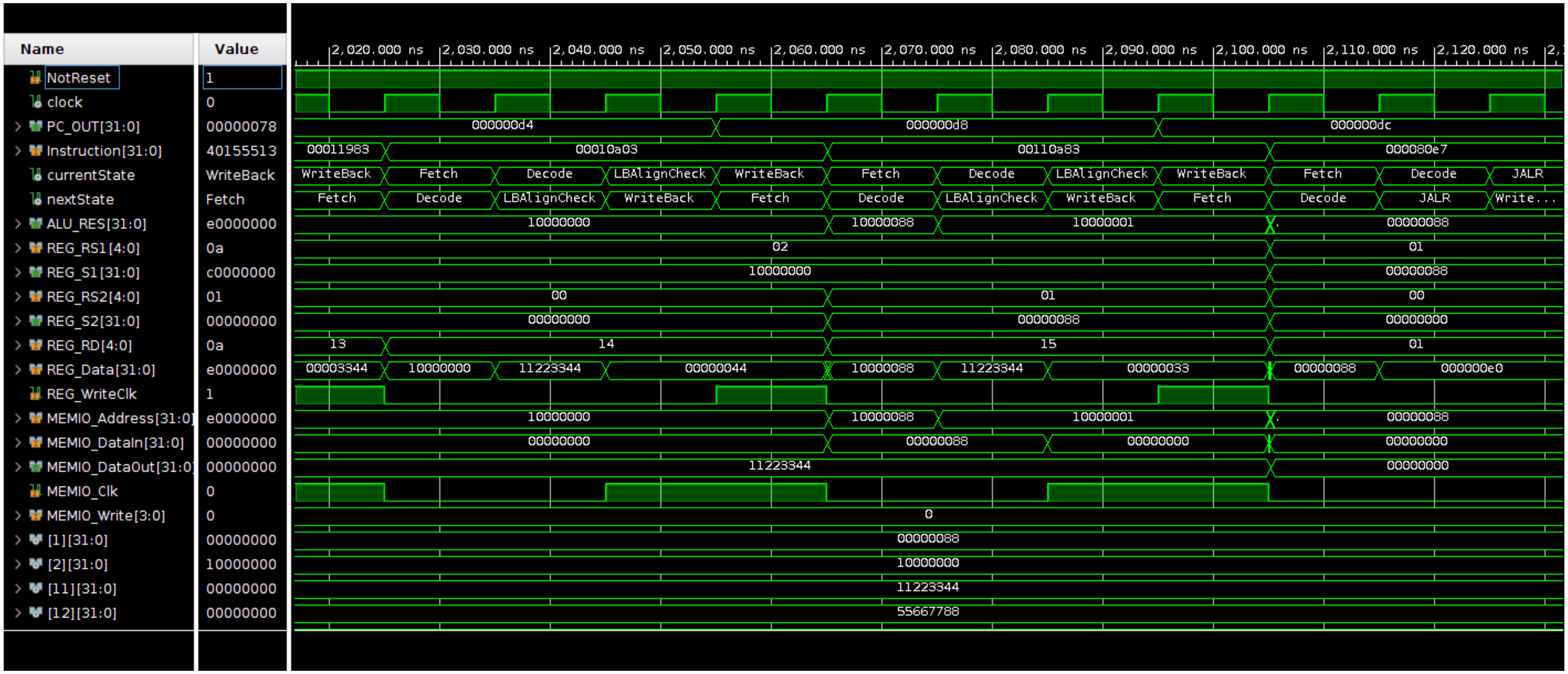


En la imagen se ejecuta:

c0: 00012783 lw x15 0(x2)

c4: 00112803 lw x16 1(x2)

Puede verse que en el primer caso, lw x15 0(x2) hace un acceso alineado. Es por esto que la unidad de alineamiento es transparente y la ejecución se realiza en cuatro ciclos de reloj. Sin embargo en el segundo caso, lw x16 1(x2) hace un acceso desalineado a memoria, y por ende paga una penalidad ya que debe acceder a la dirección de memoria siguiente para completar el dato, utilizando un registro latch auxiliar dentro de la unidad de alineamiento para almacenar temporalmente la primer lectura. Vemos que todo esto requiere siete ciclos de reloj para completarse.

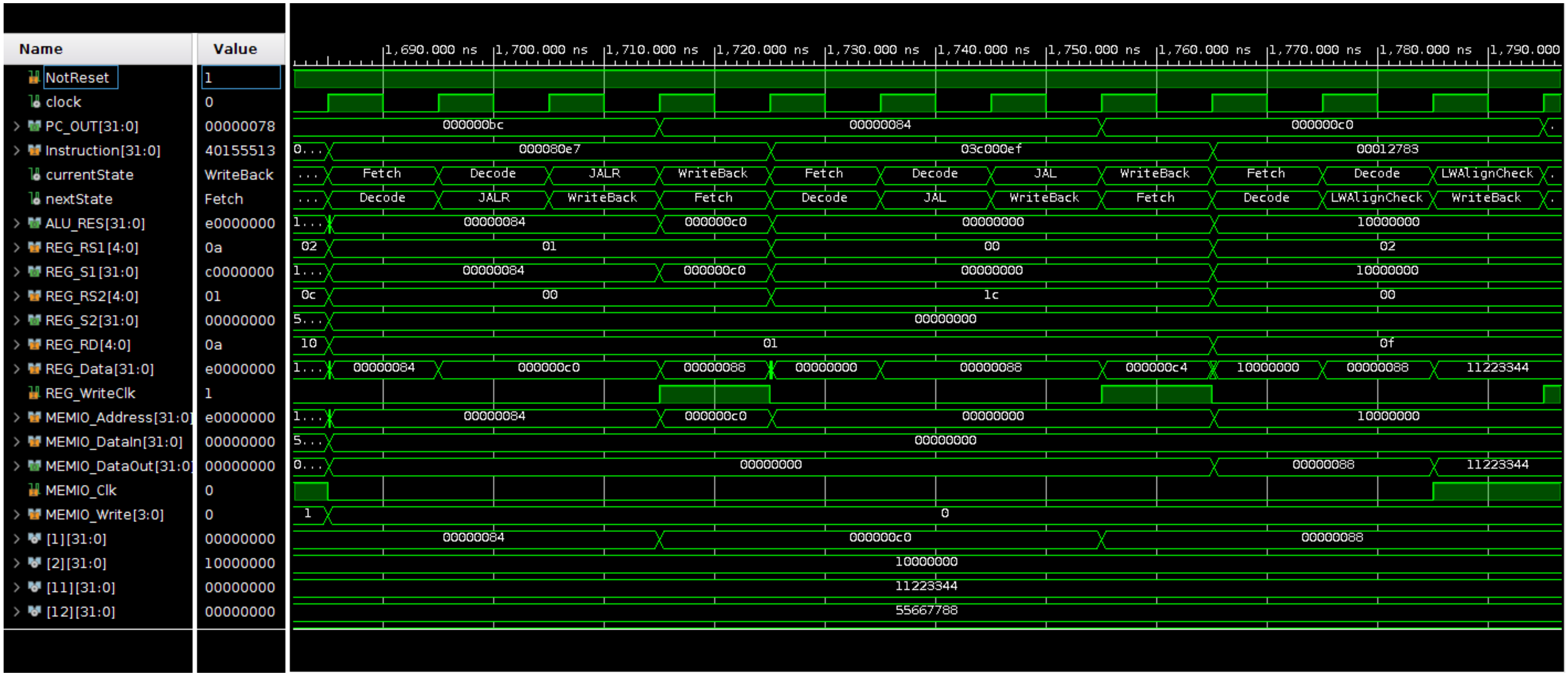


En la imagen se ejecuta:

d4:	00010a03	lb x20 0(x2)
d8:	00110a83	lb x21 1(x2)

En el caso anterior lw desalineado requiere acceder dos veces a memoria para completar una palabra de cuatro bytes. En este caso de lb, si bien el acceso puede estar desalineado, en el momento de acceder a la memoria se detecta esta condición. Debido a que la unidad de alineamiento puede desplazar en forma combinacional valores sin penalidad de ciclos de reloj, se puede ver que el segundo acceso lb x21, 1(x2) no tiene penalidad alguna en ciclos de reloj. Las instrucciones de tipo LB/LBU no pagan penalidad en ningún caso, como si la paga su contraparte SB. Algo similar ocurre con LH en el caso que la desalineación sea del tipo 01 o 10. Dado que estos casos solo requieren un acceso a memoria la unidad de alineamiento puede solucionar el problema. Sin embargo, en desalineaciones del tipo 11 es imposible evitar la penalidad ya que se requieren dos accesos a memoria.

Retornos



En el diagrama de tiempos se ejecuta:

bc: 000080e7 jalr x1 x1 0

En este caso se toma el valor almacenado en x1 y se carga en el PC, con un offset 0, y se guarda el valor actual PC+4 en el registro x1 nuevamente. Esto sirve como retorno de llamada a función, donde la dirección de retorno quedó previamente almacenada en x1. Puede verse que luego de ejecutar la instrucción, el PC pasa a tener 0x84 (valor almacenado originalmente en x1) y que x1 pasa a guardar 0xC0 (valor de PC+4).

Resultados obtenidos

A continuación se detalla un listado de componentes utilizados simulando la implementación dentro del FPGA Xilinx Artix-7 XC7A35. Estos números son de referencia ya que el objetivo es medir en promedio cuánto ocupa el diseño en un FPGA comercialmente accesible.

SiteType	Used	Fixed	Available	Util%
SliceLUTs*	1514	0	20800	7.28
LUT as Logic	1514	0	20800	7.28
LUT as Memory	0	0	9600	0.00
SliceRegisters	1183	0	41600	2.84
Register as FlipFlop	1122	0	41600	2.70
Register as Latch	61	0	41600	0.15
F7 Muxes	256	0	16300	1.57
F8 Muxes	64	0	8150	0.79

SiteType	Used	Fixed	Available	Util%
Block RAM Tile	2	0	50	4.00
RAMB36/FIFO*	2	0	50	4.00
RAMB36E1 only	2			
RAMB18	0	0	100	0.00

SiteType	Used	Fixed	Available	Util%
DSPs	4	0	90	4.44
DSP48E1 only	4			

RefName	Used	FunctionalCategory
FDCE	1121	Flop&Latch
LUT6	1075	LUT
LUT5	299	LUT
MUXF7	256	MuxFx
LUT3	148	LUT

```
| LUT4      |  109 |                 LUT |
| MUXF8     |   64 |               MuxFx |
| LDCE      |   61 |        Flop&Latch |
| LUT2      |   50 |                 LUT |
| BUFG      |   12 |               Clock |
| OBUF      |    8 |                  IO |
| DSP48E1   |    4 |    Block Arithmetic |
| LUT1      |    3 |                 LUT |
| RAMB36E1  |    2 |        Block Memory |
| IBUF      |    2 |                  IO |
| FDPE      |    1 |        Flop&Latch |
+----------+------+--------------------+
```

Según el reporte de implementación, se utiliza menos del 8% de los recursos de la FPGA.

```
Max DelayPaths
--------------------------------------------------------------------------------------
Slack (MET) :           6.549ns  (required time - arrival time)
Source:                 U0/FSM_onehot_currentState_reg[2]/C
                        (risingedge-triggeredcell FDCE clockedbyclock  {rise@0.000ns fall@5.000ns
period=10.000ns})
Destination:            U0/FSM_onehot_currentState_reg[7]/D
                        (risingedge-triggeredcell FDCE clockedbyclock  {rise@0.000ns fall@5.000ns
period=10.000ns})
PathGroup:              clock
PathType:               Setup (Max at SlowProcessCorner)
Requirement:            10.000ns  (clock rise@10.000ns - clock rise@0.000ns)
  Data PathDelay:       3.115ns  (logic 0.823ns (26.419%)  route 2.292ns (73.581%))
LogicLevels:            3  (LUT2=1 LUT4=1 LUT6=1)
ClockPathSkew:          -0.025ns (DCD - SCD + CPR)
DestinationClockDelay (DCD):    4.817ns = ( 14.817 - 10.000 )
SourceClockDelay   (SCD):    5.115ns
ClockPessimismRemoval (CPR):    0.273ns
ClockUncertainty:       0.035ns  ((TSJ^2 + TIJ^2)^1/2 + DJ) / 2 + PE
   Total SystemJitter   (TSJ):    0.071ns
   Total Input Jitter      (TIJ):    0.000ns
DiscreteJitter   (DJ):    0.000ns
Phase Error             (PE):    0.000ns
```

Según el reporte de tiempo, el mayor retardo de reloj es de 3,115ns. Esto quiere decir que la frecuencia máxima teórica a la que podría operar es 321MHz. La frecuencia utilizada en el kit de desarrollo es de 100MHz; por ende, las pruebas realizadas se encuentran limitadas a este número, pero existe la posibilidad de utilizar el diseño a mayor frecuencia.

Cabe aclarar que estos números son estimados en el proceso de síntesis, la implementación real en un FPGA suele distar de estas estimaciones. Esto se debe a que en particular el FPGA debe respetar líneas internas con retardos de reloj, adaptar los flipflop tipo D del FPGA a su modelo VHDL y conectarse con memorias (Flash, DDR) reales y no simuladas.

Comparación con SiFive FE310-G002

A nivel comercial puede adquirirse el procesador de SiFive FE310-G002. Este implementa el set de instrucciones RV32i al igual que RISC-Vp, pero también dispone de las extensiones M (para multiplicar y dividir) que no son incluidas en RISC-Vp. La frecuencia máxima a la que puede operar el FE310 es 320MHz; por ende, este valor es equivalente en ambos procesadores (tomando como valor para Risc-Vp el valor teórico reportado en la síntesis).

El procesador FE310 posee una memoria caché de instrucciones de 16KB, o sea 4096 instrucciones (32 bits por instrucción). El procesador RISC-Vp utiliza 1024 instrucciones en su configuración básica, pero dispone de 48 bloques extras de memoria de 1024 palabras, por ende, puede superar ampliamente al procesador FE310. Algo similar ocurre con la memoria RAM scratchpad. En FE310 existen 16KB de memoria RAM estática. Esto quiere decir 4096 palabras. El procesador RISC-Vp utiliza 1024 palabras en configuración básica, pero tiene hasta 48 bloques extra (compartidos con la memoria de instrucciones).

El procesador FE310 posee periféricos como UART, QSPI, PWM y Timers. Estos han sido obviados en esta etapa de desarrollo de RISC-Vp, pero al disponer de más del 90% de recursos dentro de la FPGA los mismos pueden ser implementados sin mayores complicaciones en una etapa futura de desarrollo.

El procesador FE310 implementa un esquema de pipelining mientras que RISC-Vp no utiliza pipelining aunque respeta una cantidad fija de ciclos de reloj por instrucción lo que permitirá implementar pipelining en una etapa futura de desarrollo.

Uno de los puntos donde RISC-Vp tiene una notable mejora es en el acceso de datos no alineados. El procesador FE310 no soporta desde el hardware acceso a datos de forma desalineada. El mismo genera un trap cuando detecta este tipo de accesos a memoria, lo que implica una penalidad de tiempo grande ya que debe resolverse la situación al capturar el trap o pensar el código de forma tal que se eviten los accesos desalineados complicando el desarrollo del programa. El procesador RISC-Vp soporta nativamente accesos desalineados con penalidades de tiempo muy menores a las que se podrían obtener con FE310. Inclusive permite utilizar accesos desalineados sin que el programador se vea involucrado, lo que simplifica mucho este tipo de accesos ya que no hay que tomar precauciones en el código para evitar los mismos.

Conclusiones

Los objetivos planteados en esta investigación han sido cumplidos satisfactoriamente.
En la sección 2.9 del protocolo de investigación se describen:

Objetivos específicos:

- Adquirir conocimientos sobre diversas arquitecturas abiertas de programación.
- Analizar en profundidad las arquitecturas RISC V y MIPS, mediante un análisis comparativo que permita conocer las ventajas y desventajas de cada una de ellas.
- Implementar un sistema prototipo basado en alguna de las arquitecturas mencionadas, utilizando lógica programable.
- Comparar el prototipo desarrollado con otras implementaciones de la misma arquitectura más otras arquitecturas RISC. De esta manera se busca encontrar puntos de mejora del desarrollo.
- Abrir el desarrollo encarado a la comunidad para retroalimentar el mismo con la visión de usuarios finales y enlistar oportunidades de mejora en futuros proyectos.

Como ha quedado demostrado, se ha cumplido con cada uno de los puntos arriba mencionados.

El código del prototipo ha sido publicado a la comunidad en el sitio ***github***bajo una licencia de uso abierto respetando la propiedad intelectual de los autores (GPL). El procesador RISC-Vp puede ser descargado y simulado dentro del software Xilinx Vivado 2019-2. Todos los programas y herramientas desarrollados se encuentran también disponibles.

Futuros trabajos

El objetivo de esta investigación consistió en desarrollar un procesador compatible con el set de instrucciones RISC-V e implementar el mismo en lógica programable. Al completar el diseño y la implementación en VHDL este objetivo fue cumplido.

Una mejora sobre el mismo es utilizar el diseño en un circuito del tipo FPGA. Si bien esto debería ser sencillo teniendo ya el diseño en VHDL esto no siempre es fácil. Los circuitos FPGA requieren consideraciones en el diseño que exceden lo alcanzado por VHDL. Un ejemplo de esto es el manejo de las líneas de tiempo. En un procesador como RISC-Vp no se intenta optimizar por donde se direccionan las líneas de reloj para minimizar los retardos, sino que se prioriza cumplir con el diseño de un procesador sencillo de entender.

Se propone, como siguiente paso, la creación de una nueva versión del mismo, optimizado para ejecutar dentro de un FPGA en silicio. Esto permitirá conectarlo a distintos dispositivos como sensores, actuadores, etc.

Como posibles mejoras se plantean también la implementación de un controlador de memorias SDRAM DDR3 (compatible con la memoria del mismo tipo incluido en la placa prototipo DigilentArty-7), un controlador de memoria QuadSPI (también incluida en la placa prototipo) y un controlador de memoria (MMU) que implementa una memoria caché con fines de expandir la capacidad de RISC-Vp más allá de los límites de las memorias BLOCKRAM. Esto permitiría como herramienta didáctica ser un apoyo en materias donde se vean arquitecturas de computadora, ya que permite hacer prácticas con diversos esquemas de memoria caché comparando resultados entre las diversas opciones de configuración.

La expansión del set de instrucciones a RV32IM (con el agregado de multiplicadores y divisores de hardware) sería también una buena mejora al prototipo, considerando que existe todavía mucho recurso de la FPGA que no se está utilizando.