UNIVERSITAT POLITÈCNICA DE VALÈNCIA

_ TELECOM ESCUELA TÉCNICA VLC SUPERIOR DE INGENIERÍA DE TELECOMUNICACIÓN

Diseño RTL de procesador RISC-V sobre tecnología XILINX y verificación física mediante plataforma PYNQ

Zomeño Tortajada, Alejandro

Tutor: Gadea Girones, Rafael

Trabajo Fin de Grado presentado en la Escuela Técnica Superior de Ingenieros de Telecomunicación de la Universitat Politècnica de València, para la obtención del Título de Graduado en Ingeniería de Tecnologías y Servicios de Telecomunicación

Curso 2019-20

Valencia, 23 de junio de 2020

Escuela Técnica Superior de Ingeniería de Telecomunicación
Universitat Politècnica de València
Edificio 4D. Camino de Vera, s/n, 46022 Valencia
Tel. +34 96 387 71 90, ext. 77190
www.etsit.upv.es

VLC/CAMPUS
VALENCIA, INTERNATIONAL CAMPUS OF EXCELLENCE

EMAS
Gestión ambiental verificada
REG.NO. ES-CV-000030

Resumen

En el presente trabajo se ha procedido a diseñar un procesador basado en RISC-V y elaborar un entorno de verificación hardware para este. La metodología para el diseño del procesador pasa por la utilización del programa Vivado de Xilinx y la programación en un lenguaje HDL como es Verilog, en cuanto al entorno de verificación hardware se ha utilizado una placa PYNQ, una FPGA de la familia ZYNQ de Xilinx, que es un entorno de desarrollo Python que se basa en cuadernos Jupyter. Como resultado tenemos una plataforma de verificación hardware rápida para la verificación de procesadores basados en arquitectura RISC-V.

En conclusión, se ha elaborado un entorno de verificación que ya no es solamente virtual, como podría ser la elaboración y ejecución de test bench mediante simuladores como ModelSim o Questasim, sino, un entorno de verificación integrado físicamente en una FPGA por lo que se podrá comprobar como un procesador funcionará en un entorno real.

Resum

En el present treball s'ha procedit a dissenyar un processador basat en RISC-V i elaborar un entorn de verificació hardware per a aquest. La metodologia per al disseny del processador passa per la utilització de el programa Vivado de Xilinx i la programació en un llenguatge HDL com és Verilog, pel que fa a l'entorn de verificació hardware s'ha utilitzat una placa PYNQ, un FPGA de la família ZYNQ de Xilinx, que és un entorn de desenvolupament Python que es basa en quaderns Jupyter. Com a resultat tenim una plataforma de verificació maquinari ràpida per a la verificació de processadors basats en arquitectura RISC-V.

En conclusió, s'ha elaborat un entorn de verificació que ja no és només virtual, com podria ser l'elaboració i execució de test bench mitjançant simuladors com ModelSim o Questasim, sino, un entorn de verificació integrat físicament en una FPGA per la qual cosa es podrà comprovar com un processador funcionarà en un entorn real.

Summary

In the present work we have proceeded to design a processor based on RISC-V and develop a hardware verification environment for this. The methodology for the processor design involves the use of the Xilinx Vivado program and programming in an HDL language such as Verilog, as for the hardware verification environment a PYNQ board, an FPGA of the ZYNQ family of Xilinx has been used, which is a Python development environment that is based on Jupyter notebooks. As a result we have a fast hardware verification platform for the verification of processors based on RISC-V architecture.

In conclusion, a verification environment has been developed that is no longer only virtual, such as the preparation and execution of test bench through simulators such as ModelSim or Questaim, if not, a verification environment physically integrated into an FPGA so it will be able to check how a processor will work in a real environment.

UNIVERSITAT POLITÈCNICA DE VALÈNCIA

_ TELECOM ESCUELA TÉCNICA VLC SUPERIOR DE INGENIERÍA DE TELECOMUNICACIÓN

Índice

Capítulo 1. Introduction 2

1.1 RISC-V 2

1.1.1 Historia [1] 2

1.1.2 ¿Por que RISC-V? 2

1.1.3 Conjunto de instrucciones (ISA) 3

1.2 Creando un entorno de verificación física para el RISC-V 7

Capítulo 2. Objetivos del trabajo 9

Capítulo 3. Metodología de trabajo 10

3.1 Primeros pasos 10

3.1.1 Poner en marcha la placa PYNQ [6] 10

3.2 Creando el entorno de verificación en la placa PYNQ [7] 12

3.2.1 Consideraciones previas 13

3.2.2 Primera notebook: “Downloading And Configuring” 13

3.2.3 Segunda notebook: “Creating A Bitstream” 14

3.2.4 Tercera notebook “Compiling RISC-V GCC Toolchain” [8] 16

3.2.5 Cuarta notebook: “Packaging An Overlay” 17

Capítulo 4. Desarrollo y resultados del trabajo 21

4.1 Diseño del procesador Risc V [9] 21

4.1.1 Desarrollo del procesador “Single Cycle” 21

4.1.2 Desarrollo del procesador segmentado 31

4.2 Modificaciones necesarias en el entorno de verificación de la placa PYNQ 35

4.3 Verificación del procesador mediante Jupyter Notebook 37

4.4 Mejoras del entorno de verificación 39

4.4.1 Visualización de la memoria de registros mediante Jupyter Notebook 40

4.4.2 Creación de un debug para ejecutar las instrucciones del procesador paso a paso 41

Capítulo 5. Conclusiones y propuesta de trabajo futuro 43

5.1 Mejoras en el procesador 44

5.2 Mejoras en el entorno de verificación. 44

Capítulo 6. Bibliografía 45

Capítulo 1. Introduction

1.1 RISC-V

1.1.1 Historia [1]

En mayo de 2010 como parte del Laboratorio de Computación Paralela (Par Lab) en la universidad de California en Berkeley el profesor Krste Asanović y los estudiantes Yunsup Lee y Andrew Waterman comenzaron el conjunto de instrucciones RISC-V.

El Par Lab fue un proyecto que duro cinco años con la finalidad de mejorar la computación en paralelo, fue financiado por Intel y Microsoft además de otras compañías. Todos los proyectos en Par Lab fueron de código abierto utilizando la licencia Berkeley Software Distribution (BSD), incluido por supuesto el RISC-V. Posteriormente Andrew Waterman, Yunsup Lee, David A. Patterson y Krste Asanović publicaron el siguiente informe de Par Lab describiendo el conjunto de instrucciones RISC-V: The RISC-V Instruction Set Manual, Volume I: Base User-Level ISA (EECS-2011-62) o manual de conjunto de instrucciones RISC-V, Volumen I: ISA de nivel de usuario base en español.

Para RISC-V, los patrocinadores industriales de UC Berkeley ParLab proporcionaron la financiación inicial que se utilizó para desarrollar RISC-V. No pidieron explícitamente RISC-V, sino que su interés estaba en los sistemas de procesamiento en paralelo.

Mas allá de la publicación del informe anteriormente citado, los principales hitos del RISC-V fueron la creación de un chip RISC-V en FDSOI de 28nm en 2011, la publicación de un documento sobre los beneficios de una ISA abierta, el primer taller de RISC-V celebrado en enero de 2015, y el lanzamiento de la fundación RISC-V ese mismo año.

La especificación ISA en sí misma se dispuso a dominio público, aunque el informe técnico se puso bajo una licencia Creative Commons, permitiendo de esta forma que colaboradores externos la mejoraran, incluyendo claramente la fundación RISC-V.

No se presentaron patentes respecto a ninguno de los proyectos relacionados con RISC-V, ya que su ISA no representa ninguna tecnología nueva, debido a que se basa en ideas de arquitectura de computadores que datan de al menos 40 años.

1.1.2 ¿Por que RISC-V?

RISC-V es un conjunto de instrucciones, su principal atractivo es que esta estandarizado globalmente y es libre. Tiene detrás la fundación RISC-V, una fundación sin ánimo de lucro la cual promueve que cada persona que quiera pueda innovar y desarrollar nuevas mejoras, de esta forma mejorando el estándar. Sin esta filosofía de acceso abierto a RISC-V ISA y sus extensiones, la comunidad corre el riesgo de segmentarse y que aparezcan nuevos estándares. Todas las características citadas anteriormente sumado con que la arquitectura de este tipo de procesadores no es excesivamente compleja y que su conjunto de instrucciones sea reducido y sencillo, hacen del RISC-V una arquitectura de procesadores ideal para la docencia y el comienzo del aprendizaje en esta materia.

El conjunto de instrucciones o ISA del RISC-V es modular, diferenciándose de la mayoría de las otras arquitecturas de procesadores, las cuales suelen utilizar ISAs incrementales. Esto quiere decir que las otras arquitecturas van añadiendo y añadiendo nuevas instrucciones y de esta forma aumentando la complejidad de sus ISAs, el RISC-V, contrariamente, dispone de un único núcleo fundamental del ISA que es llamado RV32I el cual es invariable. La modularidad del RISC-V viene dada debido a que podemos añadir extensiones del ISA opcionales estandarizadas dependiendo de la aplicación que el hardware vaya a realizar. Esta ventaja permite implementaciones muy pequeñas y de bajo consumo, lo que podría ser crítico en aplicaciones embebidas. Un ejemplo de esta modularidad del ISA sería el RV32IMF que quiere decir que a un

determinado procesador le estamos añadiendo además del ISA original del RISC-V (RV32I) las extensiones que agrega la multiplicación (RV32M) y punto flotante de precisión simple (RV32F).

1.1.3 Conjunto de instrucciones (ISA)

La siguiente imagen establece todas las instrucciones del RV32I/64I/128I es decir, el corazón del conjunto de instrucciones del RISC-V.

Category	Name	Format	RV32I Base	+RV64	+RV128
Loads	Load Byte	I	LB rd,rs1,imm		
	Load Halfword	I	LH rd,rs1,imm		
	Load Word	I,Cx	LW rd,rs1,imm	LD rd,rs1,imm	LQ rd,rs2,imm
	Load Byte Unsigned	I	LBU rd,rs1,imm		
	Load Half Unsigned	I	LHU rd,rs1,imm	LWU rd,rs1,imm	LDU rd,rs1,imm
Stores	Store Byte	S	SB rs1,rs2,imm		
	Store Halfword	S	SH rs1,rs2,imm		
	Store Word	S,Cx	SW rs1,rs2,imm	SD rs1,rs2,imm	SQ rs1,rs2,imm
Arithmetic	ADD	R,Cx	ADD rd,rs1,rs2	ADDW rd,rs1,rs2	ADDD rd,rs1,rs2
	ADD Immediate	I,Cx	ADDI rd,rs1,imm	ADDIW rd,rs1,imm	ADDID rd,rs1,imm
	SUBtract	R,Cx	SUB rd,rs1,rs2	SUBW rd,rs1,rs2	SUBD rd,rs1,rs2
	Load Upper Imm	U	LUI rd,imm		
	Add Upper Imm to PC	U	AUIPC rd,imm		
Logical	XOR	R	XOR rd,rs1,rs2		
	XOR Immediate	I	XORI rd,rs1,imm		
	OR	R,Cx	OR rd,rs1,rs2		
	OR Immediate	I	ORI rd,rs1,imm		
	AND	R,Cx	AND rd,rs1,rs2		
	AND Immediate	I	ANDI rd,rs1,imm		
Shifts	Shift Left	R	SLL rd,rs1,rs2	SLLW rd,rs1,rs2	SLLD rd,rs1,rs2
	Shift Left Immediate	I,Cx	SLLI rd,rs1,shamt	SLLIW rd,rs1,shamt	SLLID rd,rs1,shamt
	Shift Right	R	SRL rd,rs1,rs2	SRLW rd,rs1,rs2	SRLD rd,rs1,rs2
	Shift Right Immediate	I	SRLI rd,rs1,shamt	SRLIW rd,rs1,shamt	SRLID rd,rs1,shamt
	Shift Right Arithmetic	R	SRA rd,rs1,rs2	SRAW rd,rs1,rs2	SRAD rd,rs1,rs2
	Shift Right Arith Imm	I	SRAI rd,rs1,shamt	SRAIW rd,rs1,shamt	SRAID rd,rs1,shamt
Compare	Set <	R	SLT rd,rs1,rs2		
	Set < Immediate	I	SLTI rd,rs1,imm		
	Set < Unsigned	R	SLTU rd,rs1,rs2		
	Set < Unsigned Imm	I	SLTIU rd,rs1,imm		
Branches	Branch =	SB,Cx	BEQ rs1,rs2,imm		
	Branch ≠	SB,Cx	BNE rs1,rs2,imm		
	Branch <	SB	BLT rs1,rs2,imm		
	Branch ≥	SB	BGE rs1,rs2,imm		
	Branch < Unsigned	SB	BLTU rs1,rs2,imm		
	Branch ≥ Unsigned	SB	BGEU rs1,rs2,imm		
Jump & Link	J&L	UJ,Cx	JAL rd,imm		
	Jump & Link Register	UJ,Cx	JALR rd,rs1,imm		
Synch	Synch threads	I	FENCE		
	Synch Instr & Data	I	FENCE.I		
System	System CALL	I	SCALL		
	System BREAK	I	SBREAK		
Counters	ReaD CYCLE	I	RDCYCLE rd		
	ReaD CYCLE upper Half	I	RDCYCLEH rd		
	ReaD TIME	I	RDTIME rd		
	ReaD TIME upper Half	I	RDTIMEH rd		
	ReaD INSTR RETired	I	RDINSTRET rd		
	ReaD INSTR upper Half	I	RDINSTRETH rd		

Figura 1. Conjunto de instrucciones de base entera (RV32I/64I/128I).

Todas las aplicaciones basadas en RISC-V deberán tener el conjunto de instrucciones que en la figura de arriba se describe. El ISA del RISC-V no solamente depende del conjunto de instrucciones anterior, también tiene más extensiones que se describen más adelante, pero el conjunto fundamental es el I. Dependiendo del hardware y de la aplicación a realizar se utilizarán unas u otras extensiones, pero siempre contando con la I. Todas las extensiones del RISC-V actualmente son:

Base	*Version*	*Draft Frozen?*
RV32I	2.0	Y
RV32E	1.9	N
RV64I	2.0	Y
RV128I	1.7	N
Extension	Version	Frozen?
M	2.0	Y
A	2.0	Y
F	2.0	Y
D	2.0	Y
Q	2.0	Y
L	0.0	N
C	2.0	Y
B	0.0	N
J	0.0	N
T	0.0	N
P	0.1	N
V	0.7	N
N	1.1	N

Figura 2. Extensiones del ISA RISC-V y su estado.

En la imagen superior podemos ver todas las extensiones que existen del ISA RISC-V actualmente, además de su estado, si su estado está en “Frozen” (Y de Yes en inglés) quiere decir que esa extensión ya no puede ser modificada. A continuación, se describirá de forma breve cada una de sus extensiones:

Extensiones base:

- RV32I: Conjunto de instrucciones de base entera 32-bits.
- RV32E: Conjunto de instrucciones de base entera 32-bits reducida para sistemas embebidos con 16 registros.
- RV64I: Conjunto de instrucciones de base entera 64-bits.
- RV128I: Conjunto de instrucciones de base entera 128-bits.

Extensiones estándar:

- M: Extensión estándar para multiplicación y división de enteros.
- A: Extensión estándar para instrucciones atómicas.
- F: Extensión estándar para punto flotante de precisión simple.
- D: Extensión estándar para punto flotante de precisión doble.
- Q: Extensión estándar para punto flotante de precisión cuádruple.
- L: Extensión estándar para punto flotante decimal.
- C: Extensión estándar para instrucciones comprimidas.
- B: Extensión estándar para manipulación de bits.
- J: Extensión estándar para lenguajes traducidos dinámicamente.
- T: Extensión estándar para memoria transaccional.
- P: Extensión estándar para instrucciones empaquetadas-SIMD.
- V: Extensión estándar para operaciones de vector.
- N: Extensión estándar para interrupciones a nivel de usuario.

Ahora veremos cómo se realiza la codificación de las instrucciones. En la siguiente imagen podemos ver que representa cada uno de los 32 bits dependiendo del formato a codificar.

CORE INSTRUCTION FORMATS

	31 27 26 25	24 20	19 15	14 12	11 7	6 0
R	funct7	rs2	rs1	funct3	rd	Opcode
I	imm[11:0]		rs1	funct3	rd	Opcode
S	imm[11:5]	rs2	rs1	funct3	imm[4:0]	opcode
SB	imm[12|10:5]	rs2	rs1	funct3	imm[4:1|11]	opcode
U	imm[31:12]				rd	opcode
UJ	imm[20|10:1|11|19:12]				rd	opcode

Figura 3. Codificación de las instrucciones según el formato en RISC-V.

Para poder comprender mejor como se produce la codificación de las instrucciones se explicará que significa cada formato, y que es cada campo en la codificación.

Formatos:

- R: Formato registro a registro.
- I: Formato con inmediatos y "load" (cargas de datos de memoria a registros).
- S: Formato "storage" (carga de datos de registros a memoria).
- SB: Formato "Branch" (saltos condicionales).
- U: Formato inmediato superior.
- UJ: Formato de saltos no condicionales.

Campos:

- Opcode: Especifica parcialmente cuál de los 6 formatos es.
- funct7+funct3: Estos dos campos combinados con el opcode establecen la operación a realizar.
- rs1: Especifica que registro contiene el primer operando.
- rs2: Especifica que registro contiene el segundo operando.
- rd: Registro destino, especifica que registro contendrá el resultado de la operación.
- imm: Valor inmediato para las instrucciones que lo requieran.

En la siguiente imagen contiene la información de la codificación de los opcode más el funct7 más el funct3 para codificar todas las instrucciones del RV32I.

OPCODES IN NUMERICAL ORDER BY OPCODE

MNEMONIC	FMT	OPCODE	FUNCT3	FUNCT7 OR IMM	HEXADECIMA
lb	I	0000011	000		03/0
lh	I	0000011	001		03/1
lw	I	0000011	010		03/2
ld	I	0000011	011		03/3
lbu	I	0000011	100		03/4
lhu	I	0000011	101		03/5
lwu	I	0000011	110		03/6
addi	I	0010011	000		13/0
slli	I	0010011	001	0000000	13/1/00
slti	I	0010011	010		13/2
sltiu	I	0010011	011		13/3
xori	I	0010011	100		13/4
srli	I	0010011	101	0000000	13/5/00
srai	I	0010011	101	0100000	13/5/20
ori	I	0010011	110		13/6
andi	I	0010011	111		13/7
auipc	U	0010111			17
addiw	I	0011011	000		1B/0
slliw	I	0011011	001	0000000	1B/1/00
srliw	I	0011011	101	0000000	1B/5/00
sraiw	I	0011011	101	0100000	1B/5/20
sb	S	0100011	000		23/0
sh	S	0100011	001		23/1
sw	S	0100011	010		23/2
sd	S	0100011	011		23/3
add	R	0110011	000	0000000	33/0/00
sub	R	0110011	000	0100000	33/0/20
sll	R	0110011	001	0000000	33/1/00
slt	R	0110011	010	0000000	33/2/00
sltu	R	0110011	011	0000000	33/3/00
xor	R	0110011	100	0000000	33/4/00
srl	R	0110011	101	0000000	33/5/00
sra	R	0110011	101	0100000	33/5/20
or	R	0110011	110	0000000	33/6/00
and	R	0110011	111	0000000	33/7/00
lui	U	0110111			37
addw	R	0111011	000	0000000	3B/0/00
subw	R	0111011	000	0100000	3B/0/20
sllw	R	0111011	001	0000000	3B/1/00
srlw	R	0111011	101	0000000	3B/5/00
sraw	R	0111011	101	0100000	3B/5/20
beq	SB	1100011	000		63/0
bne	SB	1100011	001		63/1
blt	SB	1100011	100		63/4
bge	SB	1100011	101		63/5
bltu	SB	1100011	110		63/6
bgeu	SB	1100011	111		63/7
jalr	I	1100111	000		67/0
jal	UJ	1101111			6F
ecall	I	1110011	000	000000000000	73/0/000
ebreak	I	1110011	000	000000000001	73/0/001

Figura 4. Codificación de las instrucciones dependiendo del opcode, funct7 y funct3.

También cabe destacar que la mayoría de los formatos del RISC-V disponen de 32 registros (exceptuando el RV32E que dispone de 16) teniendo un registro que no se pude modificar y que siempre estará a 0, este es el registro x0. En la imagen siguiente podemos ver el uso más generalizado que tienen los diferentes registros, así como sus nombres:

REGISTER NAME, USE, CALLING CONVENTION

REGISTER	NAME	USE	SAVER
x0	zero	The constant value 0	N.A.
x1	ra	Return address	Caller
x2	sp	Stack pointer	Callee
x3	gp	Global pointer	--
x4	tp	Thread pointer	--
x5-x7	t0-t2	Temporaries	Caller
x8	s0/fp	Saved register/Frame pointer	Callee
x9	s1	Saved register	Callee
x10-x11	a0-a1	Function arguments/Return values	Caller
x12-x17	a2-a7	Function arguments	Caller
x18-x27	s2-s11	Saved registers	Callee
x28-x31	t3-t6	Temporaries	Caller
f0-f7	ft0-ft7	FP Temporaries	Caller
f8-f9	fs0-fs1	FP Saved registers	Callee
f10-f11	fa0-fa1	FP Function arguments/Return values	Caller
f12-f17	fa2-fa7	FP Function arguments	Caller
f18-f27	fs2-fs11	FP Saved registers	Callee
f28-f31	ft8-ft11	R[rd] = R[rs1] + R[rs2]	Caller

Figura 5. Uso generalizado de los registros en RISC-V.

1.2 Creando un entorno de verificación física para el RISC-V

La necesidad de un entorno dinámico en el que podamos introducir a nivel hardware diferentes tipos de procesadores de forma reiterada hace que la mejor opción sea la utilización de FPGAs o incluso la única. Por ello, para la elaboración de este entorno de verificación se ha decidido utilizar una PYNQ, una FPGA de la familia ZYNQ del fabricante estadounidense Xilinx.

PYNQ no es una FPGA cualquiera, PYNQ permite diseñar circuitos lógicos programables sin utilizar directamente herramientas de diseño de estilo ASIC como podrían ser lenguajes de diseño hardware HDL como Verilog, sino, que utiliza de forma nativa el lenguaje de alto nivel Python. Que utilice de forma nativa este lenguaje no quiere decir que no podamos diseñar un proyecto con un lenguaje HDL y posteriormente introducírselo. Para programar la PYNQ mediante Python de forma estándar, utilizaremos unos ficheros llamados Jupyter notebooks los cuales residen en la placa y son accesibles desde la mayoría de los navegadores web como serian Google Chrome o Firefox. La ventaja de estos notebooks radica en que son documentos muy dinámicos, se pueden realizar multitud de programas, agregar imágenes, videos, etc, haciendo que este tipo de documentos sean ideales para la docencia. Un ejemplo de la ejecución de uno de estos notebooks podemos verlo en la figura 5, donde la PYNQ está ejecutando un sencillo programa escrito en Python donde tenemos que adivinar un número del 0 al 10 pudiendo ver las salidas e introducir los datos que nos pida el programa desde nuestro navegador.

jupyter 3_jupyter_notebooks_advanced_features Last Checkpoint: 17 minutes ago (autosaved)

File Edit View Insert Cell Kernel Widgets Help

Guess that number game

Run the cell to play
Cell can be run by selecting the cell and pressing *Shift+Enter*

```
In [*]: import random

the_number = random.randint(0, 10)
guess = -1

name = input('Player what is your name? ')

while guess != the_number:
    guess_text = input('Guess a number between 0 and 10: ')
    guess = int(guess_text)

    if guess < the_number:
        print(f'Sorry {name}, your guess of {guess} was too LOW.\n')
    elif guess > the_number:
        print(f'Sorry {name}, your guess of {guess} was too HIGH.\n')
    else:
        print(f'Excellent work {name}, you won, it was {guess}!\n')

print('Done')
```

```
Player what is your name? Alex
Guess a number between 0 and 10: 1
Sorry Alex, your guess of 1 was too LOW.

Guess a number between 0 and 10: 4
Sorry Alex, your guess of 4 was too HIGH.

Guess a number between 0 and 10:
```

Figura 6. Ejemplo de ejecución de un documento Jupyter notebook.

La utilización de esta placa no es para nada arbitrario, utilizar esta forma de verificación física tiene numerosas ventajas frente a las verificaciones mediante simuladores u otras verificaciones físicas.

La ventaja principal que intenta implementar este trabajo frente a verificaciones mediante simuladores como podría ser una verificación del tipo RTL o Gate level, es la que ya se ha hablado con anterioridad, es decir, contar con un procesador real y comprobar como este funciona tras ejecuciones de programas, está claro que una verificación del tipo Gate level se aproxima más a lo que sería una verificación real, pero sigue siendo una simulación software. La simulación mediante la placa PYNQ se basará fundamentalmente en comprobar los bloques de memoria del procesador después de que este ejecute un programa, la desventaja de esta verificación frente a los simuladores es que no permitiría (al menos no de forma tan directa) observar todas las señales internas del procesador por lo que la verificación no podría ser tan completa o por así decirlo tan precisa.

La utilidad de este trabajo sale a relucir al comparar esta forma de verificación física con otras del mismo estilo. Se pondrá como ejemplo la verificación física mediante la placa DE2 de Altera. Las verificaciones mediante este tipo de placa consisten principalmente en introducir el diseño de Quartus en esta placa y asignar mediante una herramienta llamada "pin planner" las señales de las que dispone el diseño del procesador a los pines de la FPGA. En comparación con este trabajo, el usuario solamente tendrá que introducir el proyecto del procesador en el proyecto de verificación y posteriormente elaborar el bitstream e introducirlo a la placa, pueden parecer bastantes acciones, pero con toda seguridad se realizan con más rapidez que la asignación de pines uno a uno. La rapidez no es solamente la única ventaja, este trabajo además ofrece muchísima más flexibilidad al realizar en si la verificación. A continuación, se enumerará las diferentes ventajas que ofrece frente a una verificación física convencional:

Una vez introducido el proyecto de verificación en la placa, mediante las notebooks se puede verificar el proyecto observando el contenido de las memorias del procesador, esto en la placa DE2 no se podría hacer de forma tan sencilla ni de lejos, además de que se pueden realizar diferentes notebooks para automatizar la verificación completamente mientras que el usuario de la placa DE2 tendrá que verificar el mismo si las señales que está observando son las correctas. Con la placa PYNQ se pude cambiar el programa que vaya a ejecutar el procesador en cuestión de segundos a diferencia de la placa DE2.

En resumen, la verificación física que se ha desarrollado en este trabajo presenta un gran dinamismo a la hora de la verificación, haciendo que esta sea muchísimo más rápida e intuitiva para el usuario, pudiendo rivalizar incluso con las verificaciones mediante simuladores en cuestión de rapidez, teniendo solamente como ventaja los simuladores el poder acceder de forma más directa a cualquiera de las señales del procesador.

Capítulo 2. Objetivos del trabajo

Este trabajo consta de dos objetivos bien marcados:

- **Diseño de un procesador basado en RISC-V:** Como se ha dicho anteriormente, este objetivo del trabajo se basará en la elaboración de un procesador basado en arquitectura RISC-V mediante lenguajes de diseño hardware, en el caso de este trabajo, utilizando SystemVerilog, lenguaje basado en Verilog.

- **Elaboración de un entorno de verificación hardware:** En cuanto al segundo objetivo del trabajo, este trata de diseñar una forma de verificación a nivel hardware dinámica y que se pueda emplear en la mayoría o totalidad de las estructuras que ofrece el RISC-V. La razón de esto es para hacernos una idea más cercana de cómo se comportará un procesador de esta arquitectura cuando se implemente físicamente, de esta forma, no tendremos solamente que confiar en las verificaciones a nivel software ya que al fin y al cabo solamente son simulaciones y no señales reales.

Así pues, la filosofía del presente trabajo es la elaboración de un procesador RISC-V mediante un lenguaje HDL, más concretamente SystemVerilog, introducir dicho procesador en un entorno de verificación hardware elaborado mediante la placa PYNQ y por último diseñar diferentes documentos Jupyter notebook para automatizar el proceso de verificación del correcto funcionamiento del procesador sometido al test.

En algunos puntos de esta memoria, sobre todo en el capítulo de la metodología de trabajo, se ha decidido ser muy concreto a la hora de como se ha ido trabajando para elaborar el entorno de verificación, esto es debido a que la presente memoria también tiene como objetivo poder ser utilizada a modo de guía para la realización de dichos entornos.

Capítulo 3. Metodología de trabajo

3.1 Primeros pasos

Para la elaboración del segundo objetivo del presente trabajo, es decir la elaboración de un entorno de verificación hardware, se ha partido de un trabajo realizado por Dustin Richmond y colgado bajo el nombre de usuario drichmond en la página web de GitHub[5]. En los siguientes puntos se explicará los pasos que se realizaron para la creación de dicho entorno de verificación.

Las aportaciones fundamentales que se han añadido al trabajo de dicho usuario y de forma resumida son: adaptación del trabajo para pasar de la utilización de una arquitectura del tipo Von Neumann con una única memoria de datos e instrucciones a una arquitectura del tipo Harvard con dos memorias, una para datos y otra para instrucciones. Añadir la posibilidad de observar el banco de registros desde Jupyter Notebooks y agregar un debug paso a paso.

3.1.1 Poner en marcha la placa PYNQ [6]

El primer punto que realizar será la puesta en marcha de la placa PYNQ, para ello necesitaremos realizar los siguientes pasos:

3.1.1.1 Prerrequisitos:

- Una placa PYNQ-Z1
- Un ordenador con uno de los siguientes navegadores: Chrome, Safari o Firefox
- Un cable Ethernet
- Un cable micro USB
- Una tarjeta MicroSD de al menos 8GB para soportar la imagen de la PYNQ, pero en nuestro caso necesitaremos una de al menos 16GB para instalar todas las herramientas necesarias en la placa para construir el entorno de verificación.

3.1.1.2 Configuración de la tarjeta MicroSD

Los pasos que realizar para la preparación de la tarjeta MicroSD son los siguientes:

1. Descargar la imagen de la PYNQ-Z1. Esta imagen se puede descargar en el enlace [6] de la bibliografía en el primer punto del apartado “MicroSD Card Setup”.
2. Descargar el programa Win32DiskImager.
3. Conectar la tarjeta MicroSD al ordenador.
4. Ejecutar el programa Win32DiskImager.
5. Seleccionar la imagen de la PYNQ anteriormente descargada y la MicroSD a escribir.

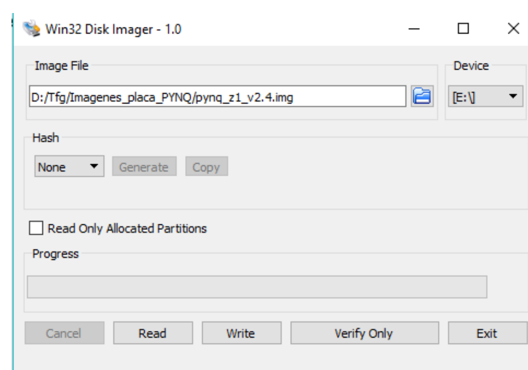

Figura 7. Ejemplo de escritura de la imagen PYNQ sobre una tarjeta MicroSD.

3.1.1.3 Configuración Ethernet

Para poder acceder a la placa PYNQ desde un navegador tendremos dos opciones:

1. **Conectar la placa a un router o a una red con un servidor DHCP**: Si se utiliza esta opción la placa adquirirá una dirección IP de forma automática. Los pasos que seguir son los siguientes:

    1) Conectar la placa mediante un cable Ethernet a un puerto del router o switch.

    2) Con el ordenador, conectarte vía cable Ethernet o WiFi al router o switch en el que está conectado la placa.

    3) Buscar en cualquiera de los anteriores navegadores mencionados la dirección http://pynq:9090 o directamente la dirección IP que haya adquirido la placa PYNQ. En el siguiente punto se explica cómo se puede saber la dirección IP que la placa ha adquirido.

2. **Conectar la placa a un ordenador de forma directa:** La opción recomendada si se disponen de los medios necesarios, es la explicada en el punto uno, ya que si no, de la forma siguiente, tanto la placa como el ordenador a la que se conecte no dispondrán de acceso a internet. Algunos pasos para la creación del entorno de verificación la placa necesita de conexión a internet, así que para el cumplimiento de estos solo se podrá emplear el tipo de conexión del punto anterior. Los pasos para conectar la placa de forma directa al ordenador son los siguientes:

    1) Asignar al ordenador una IP estática: Para poder cambiarla podemos seguir los siguientes pasos (en Windows):
        - Ir a: panel de control/ Redes e Internet/ Centro de redes y recursos compartidos.
        - En la pestaña de “Ver las redes activas” hacer clic con el ratón en “Conexiones”.
        - Hacer clic sobre propiedades.
        - Buscar “Protocolo de Internet versión 4(TCP/IPv4)” y clicar sobre este dos veces.
        - Cambiar la opción de “Obtener una dirección IP automáticamente” a “Usar la siguiente dirección IP”
        - Poner en la máscara de subred 255.255.255.0.
        - Poner en la dirección IP 192.168.2.1 o cualquiera que esté dentro del alcance de la placa, siendo la dirección IP por defecto de la placa 192.168.2.99 y siendo la máscara de subred la del anterior punto podremos poner las siguientes IPs: 192.168.2.x donde x va de 1-254 sin el numero 99 ya que de lo contrario estaríamos poniendo la misma IP de la placa al ordenador.
        - Por último, darle a aceptar.

    2) Conectar la placa PYNQ a uno de los puertos Ethernet del ordenador.

3) Buscar en un navegador la dirección IP por defecto de la placa: http://192.168.2.99:9090.

Al completar uno de estos dos pasos se accederá a jupyter notebooks, entorno en el que se crearan los ficheros con extensión .ipynb, ficheros que ejecuta la placa PYNQ.

Si por algún caso se necesita saber con certeza que dirección IP ha adquirido la placa PYNQ se puede hacer lo siguiente:

- Descargar el programa PuTTY.
- Conectar la placa al ordenador con el cable MicroUSB.
- Abrir el administrador de dispositivos y averiguar en qué puerto COM se encuentra la placa PYNQ.
- Introducir los datos en el programa tal y como se muestra en la siguiente imagen. Tener cuidado con poner el puerto COM adecuado.

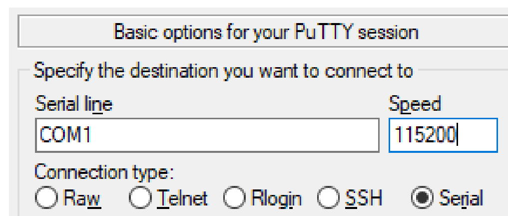

Figura 8. Configuración de PuTTY para abrir un terminal serie hacia la placa PYNQ.

- Verificar que las siguientes opciones en la pestaña de "Serial" coinciden:
  1) 115200 baud
  2) 8 data bits
  3) 1 stop bit
  4) No Parity
  5) No Flow Control

- Por último, volver a la pestaña "Session" y darle al botón "Open".

De esta forma se ha abierto un terminal serie para poder acceder a la consola de comandos de la placa PYNQ. En caso de querer saber la dirección IP de la placa, introducir en la consola el comando "ifconfig"

Una vez completados todos los pasos del punto "3.1.1 Poner en marcha la placa PYNQ" ya se puede encender la placa PYNQ, para ello mover a la posición de ON el switch de encendido, si todo se ha configurado de forma adecuada el LED LD12 se encenderá para indicar que la placa está operativa. Después de alrededor de un minuto los LEDs LD4 y LD5 deberán parpadear con un color azul, indicando que la placa ya se puede utilizar.

3.2 Creando el entorno de verificación en la placa PYNQ [7]

Este apartado se basa en seguir las cuatro notebooks que conforman el tutorial para crear el entorno básico de verificación de la placa y los archivos necesarios los cuales se elaboran a partir de la unión de los proyectos en vivado de verificación y del procesador RISC-V. Estas cuatro notebooks se pueden encontrar en el enlace de la bibliografía [7].

3.2.1 Consideraciones previas

Antes de seguir las instrucciones indicadas en las notebooks es recomendable descargarlas antes e introducirlas en la placa PYNQ, de esta forma, los comandos que aparezcan en las notebooks se pueden ejecutar directamente.

Para descargar las notebooks se puede ir al enlace [7] hacer clic derecho en la notebook, clicar sobre "Guardar enlace como…" y descargarlas con formato ipynb. Para subirlas a la placa se puede acceder a jupyter notebooks y clicar sobre el botón "Upload".

3.2.2 Primera notebook: "Downloading And Configuring"

Esta primera notebook se centra en descargar todas las herramientas necesarias para la creación del entorno de verificación que estamos buscando. Para esta notebook la placa PYNQ necesitará una conexión a internet, por lo que si hemos conectado la placa tal y como se explica en el apartado 2 del punto 3.1.1.3 Configuración Ethernet no se podrán descargar dichas herramientas.

Para comprobar que la placa PYNQ dispone de una conexión a internet podemos ejecutar el comando siguiente en la placa:

```
!ping google.com -c 10
```

De esta forma la placa enviará 10 paquetes a la dirección de google.com y deberá de recibir el mismo número de respuestas de este.

Para asegurarnos que la versión de la imagen que se ha introducido en la placa esta actualizada, se puede ejecutar el siguiente código, el cual informa si la imagen esta actualizada o en el caso contrario se encarga de renovarla.

```
REVISION = !cat /home/xilinx/REVISION
if('Release 2017_08_17 8123713' == REVISION[0]):

    !sed -i 's/ubuntu-ports/ubuntu/' /etc/apt/sources.list.d/multis
trap-wily.list
    !sed -i 's/ports/old-releases/' /etc/apt/sources.list.d/multist
rap-wily.list

    !apt update
else:
    print("PYNQ is Up-To-Date!")
```

Con la siguiente celda del notebook podremos instalar en la placa todas las dependencias que necesitaremos, como por ejemplo la de git para poder clonar el repositorio del usuario drichmond en nuestra placa.

```
!apt -y install autoconf automake autotools-dev curl libmpc-dev lib
mpfr-dev libgmp-dev gawk build-essential bison flex texinfo gperf l
ibtool patchutils bc zlib1g-dev git
```

Descargaremos los datos del GNU Toolchain, con la ayuda de las herramientas descargadas con comando anteriormente citado:

```
!git clone --recursive https://github.com/riscv/riscv-gnu-toolchain
/home/xilinx/riscv-gnu-toolchain
```

Y por último en esta notebook se descargará todo el repositorio de "RISC-V-On-PYNQ" del usuario en la placa, ya que, posteriormente será necesario.

```
!git clone --recursive https://github.com/drichmond/RISC-V-On-PYNQ
/home/xilinx/RISC-V-On-PYNQ
```

3.2.3 Segunda notebook: "Creating A Bitstream"

Esta notebook explica cómo se crean los Bitstream a partir de un proyecto de vivado para después introducirlos en la placa PYNQ y así poder verificarlos.

En esta notebook utilizaremos el proyecto en vivado de un procesador RISC-V que nos proporciona el repositorio del usuario drichmond. Posteriormente se darán todas las especificaciones de que características son necesarias para los procesadores propios para que el proyecto en vivado de verificación y estos se puedan comunicar entre sí.

Como paso previo para poder continuar tendremos que descargar los archivos en el repositorio de GitHub de drichmond en nuestro ordenador.

3.2.3.1 Creación de la IP asociada al proyecto vivado del procesador

Una vez descargados, el primer paso será crear un proyecto en vivado con cualquier nombre, para ello se iniciará vivado y se clicará en la opción del inicio rápido de "Create Project". Se nombrará y se elegirá el destino donde se guardará el proyecto del procesador. En la siguiente ventana al dar al botón "next" pondremos como opción "RTL Project". La siguiente ventana nos da la opción de añadir archivos al proyecto, herramienta que tendremos que utilizar para añadir el procesador de prueba que nos proporcionan, para esto seleccionaremos la opción de "Add Files" e iremos a la dirección donde tengamos descargado el repositorio y entraremos en la carpeta de "picorv32" seleccionando el archivo "picorv32.v". Posteriormente clicaremos sobre "next" hasta llegar a la ventana de selección de chip ("Default Part") y seleccionaremos el chip asociado a la placa PYNQ, es decir, xc7z020clg400-1, después de esto ya se habrá creado el proyecto.

Ahora crearemos una nueva IP a partir de este procesador. Para ello iremos a la pestaña de "Tools" y seleccionaremos la opción de "Create and Package New IP". En la ventana emergente podremos clicar en el botón de "next" con las opciones por defecto hasta cerrarla teniendo precaución de saber dónde se guarda la IP, ya que la tendremos que unir posteriormente a otro proyecto vivado. Una vez cerrada la ventana, se nos aparecerá las opciones de customización de la IP que vamos a crear, en ella podremos introducir el nombre con la que aparecerá en el "Block design" y quitado la posible personalización de esto, las opciones por defecto son las correctas y ya podremos clicar en "Review and Package" y dar en la opción de "Package IP" creando así la IP.

3.2.3.2 Puesta en conjunto del proyecto vivado asociado a la placa PYNQ con la IP del procesador

Este apartado consiste en la unión de la IP del procesador creada en el apartado anterior con el proyecto vivado de la placa, en el que se incluyen todas las entradas y salidas necesarias de esta además de todas las configuraciones adicionales para la buena conexión entre estos dos elementos, todo mediante la opción de "Block design" de vivado.

El usuario drichmond nos proporciona el citado proyecto asociado a la placa, para acceder a él sin embargo necesitaremos de una máquina virtual basada en Linux además de disponer de la versión de vivado 2017.4 para poder reconstruirlo ya que el usuario lo proporciona en un fichero tipo .tcl. Deberemos tener descargado el repositorio en la máquina virtual y ejecutar la siguiente instrucción en el terminal:

```
make -C <dirección al repositorio>/RISC-V-On-
PYNQ/riscvonpynq/picorv32/tut synth
```

Se habrá creado una carpeta llamada tutorial dentro de la carpeta tut donde podremos encontrar el proyecto deseado con el nombre de tutorial.xpr.

El siguiente paso será abrir el proyecto de tutorial. Una vez abierto, clicaremos sobre la flecha que se encuentra a la izquierda del fichero verilog “tutorial_wrapper” mostrando así el fichero “tutorial_i:tutorial” del “Block deisgn” abriéremos este y deberá tener un aspecto como la siguiente imagen.

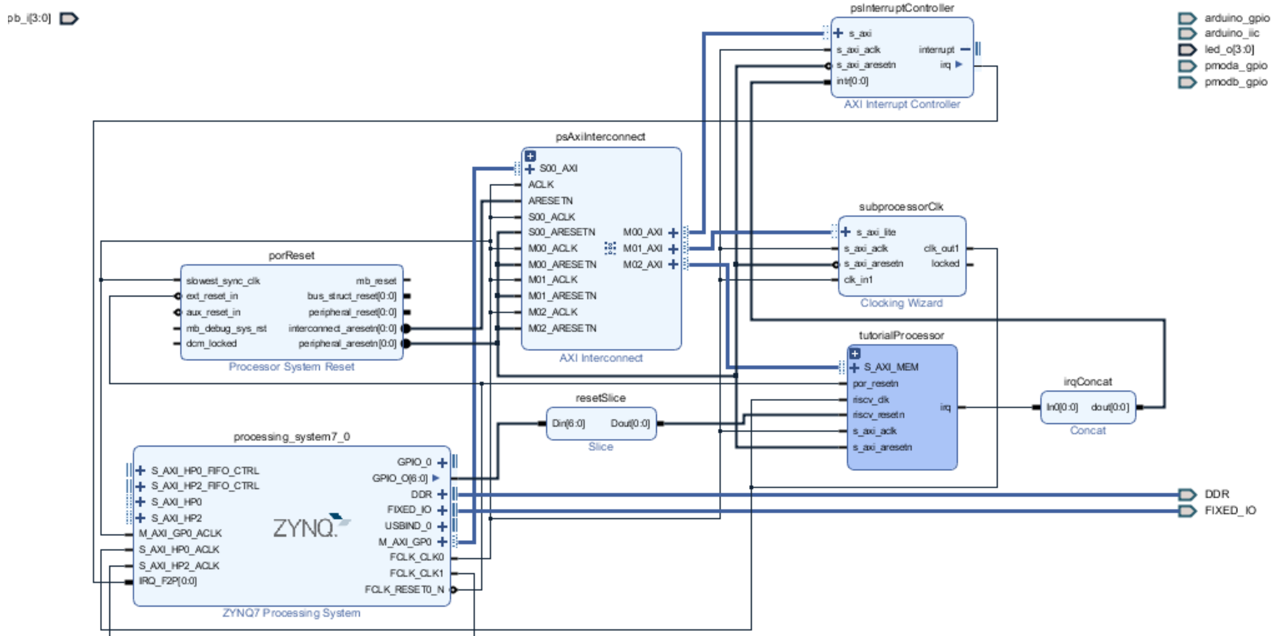

Figura 9. Proyecto que encapsulara al procesador sujeto a testeo que funciona como conexión entre la placa y el dicho procesador.

Ya podremos añadir la IP del procesador que nos han proporcionado, para ello, en el “Flow Navigator”, situado a la izquierda, pulsaremos sobre la opción de “IP Catalog” que se encuentra dentro de la sección de “PROJECT MANAGER” y haremos clic derecho sobre “User Repository” seleccionaremos “Add Repository” y seleccionaremos la carpeta donde guardamos la IP del procesador, de esta forma ya se podrá utilizar. Ahora simplemente tendremos que clicar sobre el recuadro en el que pone un “+” en el bloque de “tutorialProcessor” (el único en azul más oscuro), clicaremos dos veces sobre él, de esta forma nos aparecerá solamente en el visor este bloque, haremos clic derecho y daremos en la opción de “Add IP” aparecerá un buscador y tendremos que introducir el nombre con el que hemos guardado nuestra IP del procesador.

Para finalizar la unión de la IP con el proyecto de la placa tendremos que hacer unas cuantas conexiones que se pueden apreciar en la siguiente imagen:

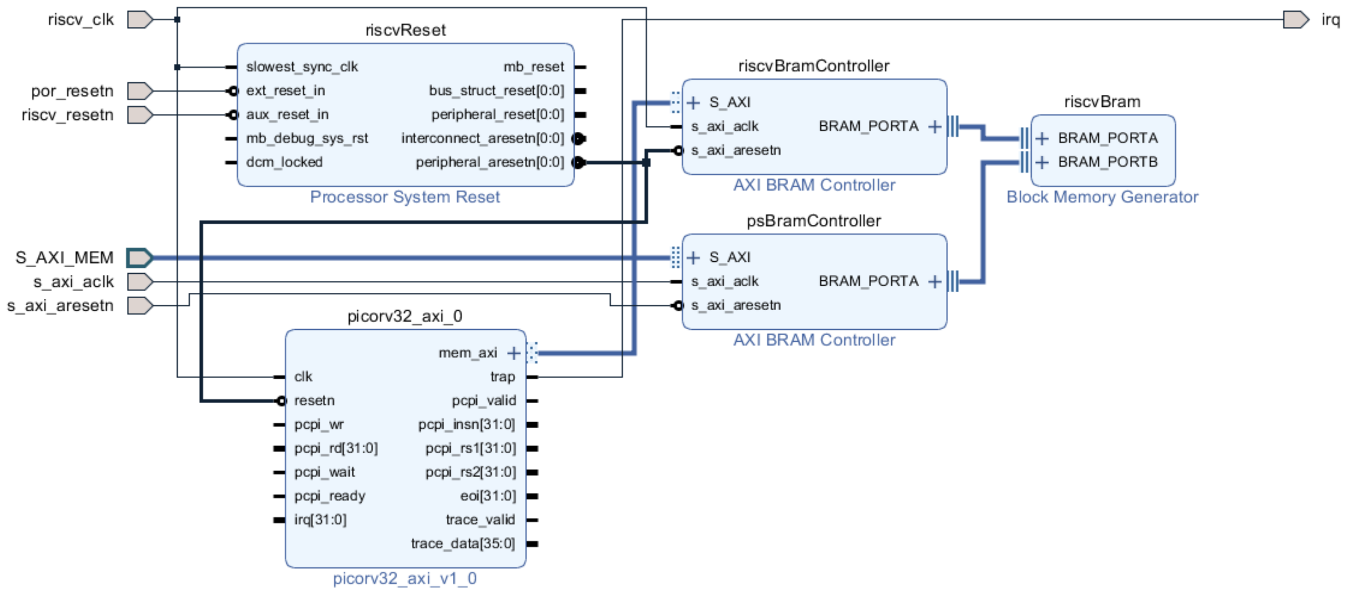

Figura 10. Proyecto de la placa conectado a la IP del procesador.

Como podemos ver en el esquema, solo hacen falta hacer las conexiones de la señal del procesador “clk” con la entrada al bloque de “riscv_clk”, la señal de “resetn” con “peripheral_aresetn”, el bus “mem_axi” con “S_AXI” y por último la señal “trap” con “irq”.

Ahora tendremos que asignar un rango de memorias para el bus de tipo AXI que tiene la IP del procesador. Para esto podemos introducir “Address Editor” en el buscador del programa situado en la parte superior central, desplegaremos la pestaña de “tutorialProcessor”, haremos clic derecho sobre “mem_axi” y seleccionaremos la opción de “Auto Assign Address” y la dirección de Offset la pondremos a 0.

Por último, ya solo nos quedará crear los ficheros que introduciremos a la placa, estos son un .bit y un .tcl. Para generarlos seguiremos los siguientes pasos:

Primero validaremos el diseño, para ello iremos a la pestaña de “Tools” y clicaremos sobre la primera opción “Validate Design”. Una vez verificado el diseño crearemos el fichero .bit, en el “Flow Navigator”, clicaremos sobre la opción de “Generate Bitstream” del apartado “PROGRAM AND DEBUG” y aceptaremos las opciones que aparezcan. El fichero .tcl podremos crearlo desde la pestaña “File” opción de “Export” y “Export Block Design” siempre teniendo el diagrama del “Block Design” abierto o esta opción no estará disponible.

3.2.4 Tercera notebook “Compiling RISC-V GCC Toolchain” [8]

Esta notebook se encarga de compilar las herramientas necesarias para el entorno de verificación de la placa que se han descargado durante el punto ***3.2.2*** .

La herramienta en cuestión es lo que se conoce en inglés como “toolchain”. Un “toolchain” es un conjunto de herramientas de desarrollo software que están conectadas entre sí por diferentes etapas.

Los componentes del “toolchain” que vamos a compilar en este notebook son los siguientes:

- **Binutils:** El "GNU Binutils" es el primer componente de un "toolchain", estos están compuestos de dos herramientas muy importantes, a saber:
    - El ensamblador, el cual se encarga de transformar el código en ensamblador a código binario.
    - El enlazador, el cual enlaza varios código objeto en una librería o ejecutable.

- **El compilador:** Este compilador se trata del GCC el cual admite los siguientes tipos de lenguaje: C, C++, Java, Fortran, Objective-C y Ada. En cuanto a la salida, esta puede dar soporte a una gran variedad de arquitecturas.

- **Una librería de C:** En nuestro caso se trata de "glibc" , librería de C usada virtualmente por todos los sistemas de escritorios y servidores GNU/Linux.

- **Debug:** El empleado será el "GDB" y será utilizado para depurar procesos en ejecución en nuestra placa.

Resumiendo, en esta notebook simplemente se tendrá que ejecutar los comandos escritos en esta mediante la placa PYNQ, los comandos se encargaran de hacer un listado del repositorio del "GNU Toolchain" que nos descargamos en la primera notebook, configurarlo, compilarlo, modificar el directorio para que Jupyter Notebooks pueda acceder a él y por último verificar la instalación para comprobar que todo se ha instalado y configurado de forma correcta.

3.2.5 Cuarta notebook: "Packaging An Overlay"

En la cuarta y última de las notebooks proporcionadas como tutorial, introduciremos el "Overlay" del procesador dentro de la placa PYNQ, el cual se generará mediante los ficheros creados en el punto ***3.2.3*** , además de realizar todos los ficheros necesarios para poder ejecutar ya programas en el procesador creado y verificar su correcto funcionamiento.

Ahora habrá que crear todo el sistema de ficheros dentro de la placa PYNQ para poder cargar el "Overlay". Necesitaremos, primeramente, de un programa para acceder a los archivos y poder modificarlos en la placa, WinSCP permite hacer conexiones SSH y tiene un entorno gráfico por lo que se ha decidido utilizar este. En el repositorio de "RISC-V-On-PYNQ" dentro de la placa crearemos una carpeta dentro de la dirección RISC-V-On-PYNQ/riscvonpynq/picorv32 donde guardaremos los ficheros .bit y .tcl creados anteriormente. El sistema de ficheros deberá tener la forma y archivos que aparece en la siguiente imagen:

/home/xilinx/RISC-V-On-PYNQ/riscvonpynq/picorv32/tutorial/

Nombre	Tamaño	Modificado	Permisos	Propiet...
..		01/08/2019 3:12:53	rwxr-xr-x	xilinx
__pycache__		01/08/2019 3:12:47	rwxr-xr-x	root
build		27/11/2019 1:32:44	rwxr-xr-x	xilinx
__init__.py	1 KB	16/07/2019 1:50:43	rw-rw-r--	xilinx
tutorial.bit	3.951 KB	13/02/2020 13:48:02	rw-rw-r--	xilinx
tutorial.py	3 KB	23/07/2019 1:29:03	rw-rw-r--	xilinx
tutorial.tcl	62 KB	13/02/2020 13:43:46	rw-rw-r--	xilinx

Figura 11. Forma del sistema de archivos que tenemos que crear para poder generar el "Overlay".

En mi caso, la carpeta se ha nombrado tutorial como se puede apreciar en la imagen, ciertos ficheros que se deberán crear después es recomendable nombrarlos igual que la carpeta para que no haya confusiones. Ahora se explicará que ficheros se deben crear, que deben contener, donde se deben guardar y con qué nombre:

- **Carpeta build:** Esta carpeta la podremos copiar desde el directorio RISC-V-On-PYNQ/riscvonpynq/picorv32/bram/build/ a la carpeta que hemos creado.

- **__init__.py(1):** En la dirección RISC-V-On-PYNQ/riscvonpynq/picorv32/ nos encontraremos con un archivo llamado __init__.py, tendremos que modificarlo con un editor de texto y añadirle una línea con el siguiente código "from . import <Nombre de la carpeta>" .

- **__init__.py(2):** Dentro de la carpeta que se ha creado se tendrá que crear un fichero con un editor de texto y nombrarlo __init__ con extensión .py. Dentro de él se tendrá que poner las dos siguientes líneas:

```
from . import <Nombre de la carpeta>
from . import build
```

- **"tutorial.py":** Volveremos a utilizar un editor de texto para crear un fichero con el nombre que le habremos dado a la carpeta y lo guardaremos en esta con una extensión .py. El contenido del fichero será el código que se encuentra a continuación:

```
from pynq import Overlay, GPIO, Register
import os
```

```
import inspect

class TutorialOverlay(Overlay):
    """Overlay driver for the PicoRV32 bram Overlay

    Note
    ----
    This class definition must be co-located with the .tcl and .bit
    file for the overlay for the search path modifications in
    riscvonpynq.Overlay to work. __init__ in riscvonpynq.Overlay uses
    the path of this file to search for the .bit file using the
    inspect package.

    """
    pass

class TutorialProcessor(BramProcessor):
    """Hierarchy driver for the PicoRV32 BRAM Processor

    Note
    ----
    In order to be recognized as a RISC-V Processor hierarchy, three
    conditions must be met: First, there must be a PS-Memory-Mapped
    Block RAM Controller where the name matches the variable
    _bram. Second, the hierarchy name (fullpath) must equal the
    variable _name. Finally, there must be a GPIO port with the name
    _reset_name.

    Subclasses of this module are responsible for setting _name (The
    name of the Hierarchy), _bits (Processor bit-width), _proc
    (Processor Type Name)

    This class must be placed in a known location relative to the
    build files for this processor. The relative path can be modified
    in __get_path.

    """
    _name = 'tutorialProcessor'
    _proc = 'picorv32'
    _bits = 32

    @classmethod
    def checkhierarchy(cls, description):
        return super().checkhierarchy(description)

    def __get_path(self):
        """Get the directory path of this file, or the directory path of the
        class that inherits from this class.

        """
        # Get file path of the current class (i.e. /opt/python3.6/<...>/stream.py)
        file_path = os.path.abspath(inspect.getfile(inspect.getmodule(self)))
        # Get directory path of the current class (i.e. /opt/python3.6/<...>/stream/)
        return os.path.dirname(file_path)

    def __init__(self, description, *args):
        """Return a new Processor object.

        Parameters
        ----------
        description : dict
            Dictionary describing this processor.

        """
        build_path = os.path.join(self.__get_path(), "build")
        reset_value = 0
        super().__init__(build_path, reset_value, description, *args)
```

- **"tutorial.bit y tutorial.tcl":** Estos son los ficheros que se han generado en el punto ***3.2.3*** y tendrán el nombre del proyecto de vivado, se cambiará el nombre al de la carpeta que

hemos creado con anterioridad y se guardará en esta. Cada vez que se quiera testear un nuevo procesador se tendrá que crear de nuevo los ficheros .bit y .tcl, renombrarlos y sustituir los viejos por estos en la carpeta que se ha creado. También se puede crear otro sistema de archivos con los .bit y .tcl nuevos para no borrar los anteriores.

Una vez introducidos todos estos archivos podremos poner a prueba el procesador. Para ello ejecutaremos los siguientes códigos:

```
import sys
sys.path.insert(0, '/home/xilinx/RISC-V-On-PYNQ/riscvonpynq/picorv3
2/')
sys.path.append('/home/xilinx/RISC-V-On-PYNQ/')


from <Nombre de la carpeta>.<Nombre de la carpeta> import TutorialO
verlay

overlay = TutorialOverlay("/home/xilinx/RISC-V-On-PYNQ/riscvonpynq/
picorv32/<Nombre de la carpeta>/<Nombre de la carpeta>.bit")
```

Este código generará el “Overlay” del procesador y lo introducirá dentro de la placa PYNQ.

```
%%riscvc test overlay.tutorialProcessor

int main(int argc, char ** argv){
    unsigned int * arr = (unsigned int *)argv[1];
    return arr[2];
}
```

Este código compilará un programa escrito en C++ y lo introducirá dentro del procesador sujeto a pruebas para que este lo ejecute. Se trata de un simple programa que recibirá como argumento un vector y el procesador devolverá el contenido de la posición 2.

```
import numpy as np
arg1 = np.array([4,2,3], np.uint32)

retval = overlay.tutorialProcessor.run(test, arg1)

if(retval != arg1[2]):
    print("Test failed!")
else:
    print("Test passed!")
```

Este último código se encargará de mandar como argumento el vector [4,2,3] y comprobar si la respuesta del procesador (retval) es ciertamente el numero guardado en la segunda posición del vector (en nuestro caso un 2). Si el valor enviado por el procesador es el correcto se habrá pasado el test, si no es así, alguno de los pasos se ha seguido mal.

Capítulo 4. Desarrollo y resultados del trabajo

4.1 Diseño del procesador Risc V [9]

Una vez probado que el entorno de verificación en la placa PYNQ funciona correctamente con el procesador de prueba, ya se puede pasar a la elaboración del procesador propio.

En este punto se explicará el esquema del procesador y como se desarrolló, así como que características deberá tener para poder introducirlo en nuestro sistema de verificación y que modificaciones de este se deberán hacer.

A la hora del desarrollo del procesador final se han pasado por tres fases:

1) Desarrollo del procesador "Single Cycle": La primera versión que se desarrolló del procesador es la conocida como versión "Single Cycle". Esta versión del procesador simplemente necesitaba de un ciclo de reloj para poder ejecutar una instrucción.

2) Segmentación del procesador "Single Cycle": A partir de la versión anterior del procesador se modificará de tal manera que ahora las instrucciones no se ejecutarán ciclo a ciclo, las instrucciones ahora irán pasando de etapa a etapa de ejecución con cada golpe de reloj mejorando la eficiencia de estas etapas del procesador y pudiendo aumentar la frecuencia de operación.

3) Fase de mejoras: En esta fase se añadirán módulos para evitar algunos de los posibles riesgos que contrae la utilización de un procesador del tipo segmentado o "pipeline".

Ahora se procederá a explicar con más detalle cada una de estas fases.

4.1.1 Desarrollo del procesador "Single Cycle"

4.1.1.1 introducción

Tanto la versión "Single Cycle" como la versión segmentada de este, sus estructuras vendrán marcadas por:

- **Un Datapath:** Serán todos los componentes que se encargarán de realizar todas las operaciones necesarias para la ejecución de las instrucciones.

- **Un Controlpath:** Los integrantes del controlpath serán los encargados de manejar los dispositivos que conforman el datapath. Controlarán que dispositivos y de qué forma operan, dependiendo de la instrucción a ejecutar.

En la siguiente imagen, podremos ver el esquema general del procesador y las partes del datapath (en negro) y del controlpath (en azul):

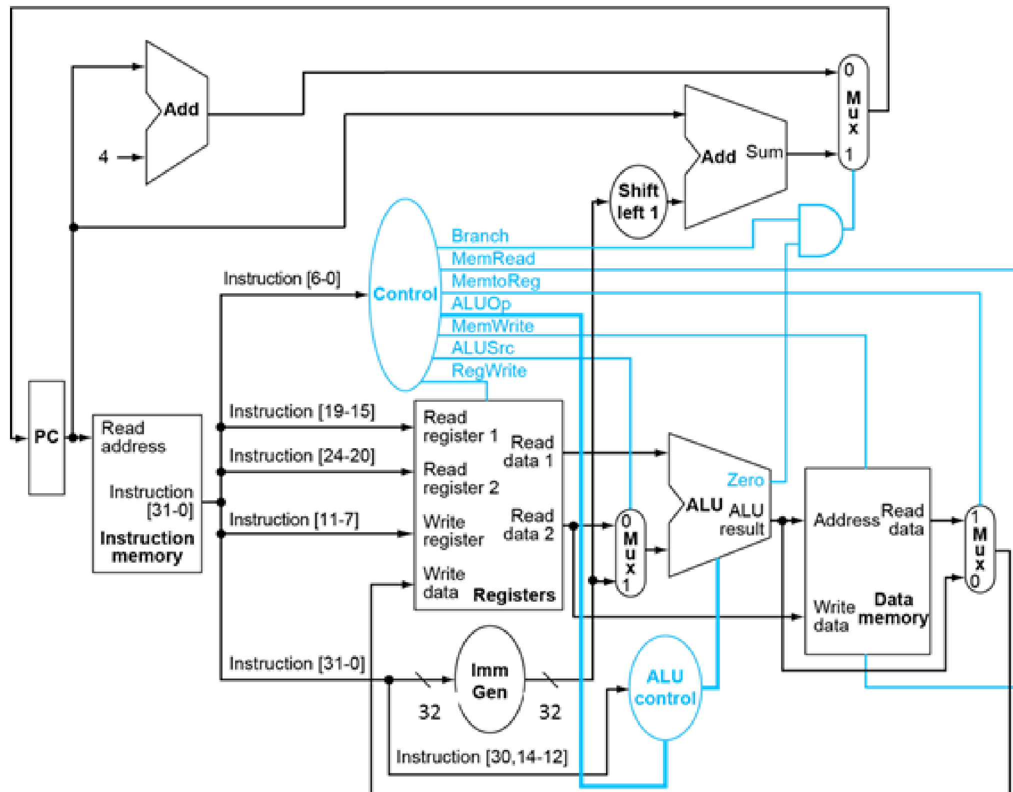

Figura 12. Esquema del procesador RISC-V monociclo.

A la hora de ejecutar una instrucción el procesador pasa por una serie de etapas, son las siguientes y las recorre en este orden:

1) **Búsqueda de instrucción (Instruction Fetch-IF):** El contador de programa se actualiza y señala a la dirección de la memoria de instrucciones donde se encuentra la siguiente instrucción a ejecutar.

2) **Decodificación de la instrucción (Instruction Decode-ID):** Los datos de la instrucción llegan al banco de registros, leyendo este los registros que se utilizarán y se generarán las señales del controlpath dependiendo del tipo de instrucción que se esté ejecutando en ese momento.

3) **Ejecución (Execution-EX):** Todas las operaciones necesarias para ejecutar la instrucción serán realizadas por una unidad aritmético-lógica ALU, o en el caso de un salto efectivo, un sumador. En esta etapa también se calculará la dirección de la siguiente instrucción si la que se está ejecutando en este momento se trata de una instrucción de salto y es efectiva.

4) **Acceso a memoria (Memory Access-MEM):** En esta etapa se accederá a la memoria de datos, ya sea para leer o escribir (puede que no se lea ni escriba ningún dato, esto dependerá de la instrucción ejecutada).

5) **Post-escritura en registro (Write Back to Register-WB):** En la etapa de post-escritura se actualizarán los valores de los registros en su memoria que hayan sido calculados por la ALU o que hayan sido cargados por la memoria de datos, si la instrucción es de ese tipo, si no, no se modificará nada.

4.1.1.2 Componentes del procesador RISC-V

Ahora se enumerarán los diferentes componentes de esta versión del procesador y se explicará su función.

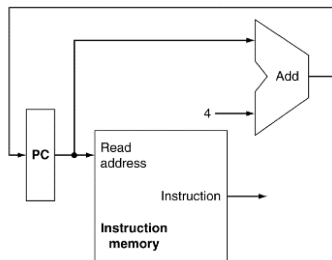

Figura 13. Componentes del datapath (Program counter y un sumador) y la memoria de instrucciones.

- **Program Counter (PC):** Se trata de un registro de almacenamiento de 32 bits con reset asíncrono, se encargará de almacenar la dirección de memoria donde se hallará la instrucción a ejecutar.

- **Memoria de instrucciones:** Es la memoria que se encargará de guardar las instrucciones que ejecutará el procesador. Tiene un tamaño de 1024x32 bits y su lectura es asíncrona (en esta versión). No se podrá escribir en ella desde el procesador. El PC se encargará de proporcionarle la dirección de memoria donde se encuentre la instrucción a ejecutar y la salida de esta será dicha instrucción.

- El último componente de la imagen se trata de un simple sumador que se encargará de sumar a la dirección de memoria que tenga el PC guardado el numero necesario para que este apunte a la siguiente dirección de memoria de instrucciones en el siguiente ciclo. Este es el modo normal de funcionamiento, pero en el caso de los saltos, la dirección de memoria de instrucciones del siguiente ciclo se tendrá que calcular mediante otros dispositivos que se explicarán más adelante.

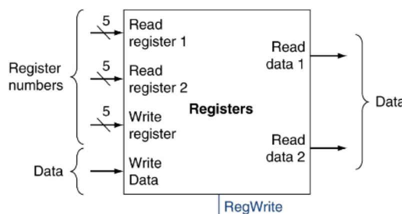

Figura 14. Memoria de registros del datapath.

- **Memoria de registros** que dispone de:

  - Read register 1 y 2: Dos entradas de 5 bits cada una que serán las direcciones de los dos registros que se vayan a leer.

  -Read data 1 y 2: Dos señales de salida de 32 bits, las cuales serán los datos leídos que se encuentren en las direcciones señaladas por las dos entradas anteriormente explicadas.

  -Write register: Señal de entrada de 5 bits que marcará la dirección del registro que se quiera escribir.

-Write Data: Señal de entrada de 32 bits que será el dato que se guardará en la dirección que marque la entrada anteriormente explicada.

-RegWrite: Señal de entrada de un solo bit y que sirve como habilitación para poder escribir en la memoria.

La escritura en esta memoria será síncrona y la lectura asíncrona.

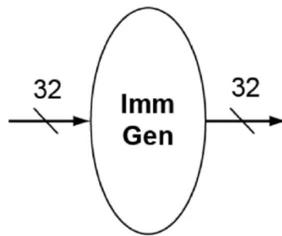

Figura 15. Generador de inmediatos.

- **Generador de inmediatos:** Como podemos ver en la **Figura 3. Codificación de las instrucciones según el formato en RISC-V**, dependiendo del formato de instrucción, el dato inmediato (imm) se encuentra en diferentes posiciones, por lo tanto, este componente del procesador tiene como entrada todos los bits de la instrucción y su función es la de organizar adecuadamente los datos de los inmediatos de esta, para los formatos I, S, SB, U y UJ.

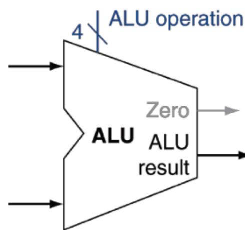

Figura 16. Unidad aritmético-lógica (ALU).

- **Unidad aritmético-lógica (ALU):** La ALU será la encargada de realizar todas las operaciones a los registros y a los inmediatos. Tiene dos entradas para estos operandos de 32 bits cada uno y una entrada de 4 bits de control que indica que operación debe realizar la unidad (ALU operation). Tiene una salida de 32 bits que indica el resultado de la operación (ALU result) y una salida de un solo bit la cual se trata de una bandera para poder realizar saltos condicionales (Zero).

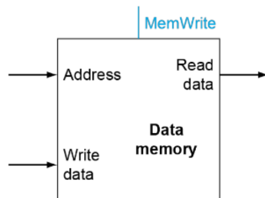

Figura 17. Memoria de datos.

- **Memoria de datos:** Como indica su nombre esta memoria será la encargada de guardar los datos que se encuentren en la memoria de registros (instrucciones de "storage") o bien de cargar ciertos valores que se encuentren en esta memoria de datos en los registros de la memoria de registros (instrucciones de "load"). La memoria dispone de:

  - Address :Entrada que será la dirección que se vaya a leer de la memoria o bien que se vaya a escribir.

-Write data: Entrada de 32 bits que será el dato a guardar en esta memoria (en el caso de escritura).

-Read data: Salida de 32 bits que será la señal leída de la memoria.

-MemWrite: Señal de control de un solo bit que indicará a la memoria si se puede escribir el dato de la señal "Write data" en ese ciclo.

Esta memoria como la de registros, tiene lectura asíncrona y escritura síncrona (en esta versión del procesador).

- **Add Sum:** El último modulo remarcable a citar del datapath es el sumador encargado de realizar los calculos de las dirrecciones efectivas de los saltos, este se puede apreciar en la **Figura 18,** arriba a la derecha. Una de sus entradas será el valor del registro PC y la otra será la salida del generador de inmediatos que indicará a que direccion debe saltar, con estos dos operandos el sumador ya puede calcular que valor debe sumar al pc en el siguiente ciclo para que se ejecute la instrucción a la que se quiere saltar.

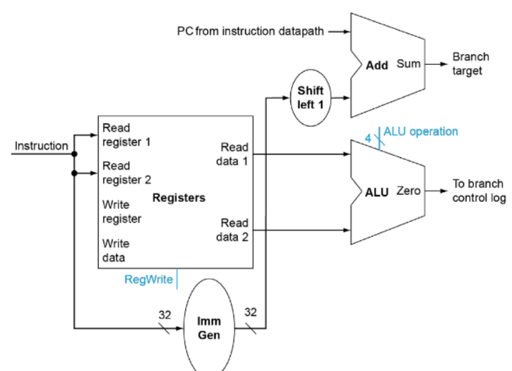

Figura 18. Sumador para el cálculo de la dirección efectiva de salto.

Con esto se da por explicado los módulos más importantes del datapath así como una imagen general de cómo funciona este, ahora explicaremos cómo funciona el controlpath que será nuestro director de orquesta de todos los elementos del datapath, se encargará de organizarlos para que estos trabajen en conjunto, de forma ordenada y correcta. El esquema de este se podrá ver en la figura 19.

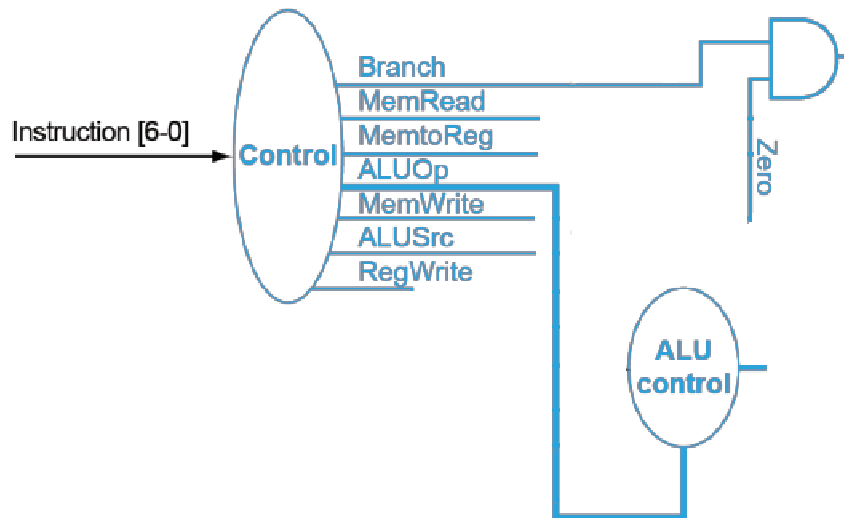

Figura 19. Controlpath del procesador.

Como podemos ver en el esquema del controlpath, este solamente tiene una entrada, se trata de los bits del 0 al 6 de la instrucción que se va a ejecutar. Si volvemos al capítulo donde se explica cómo se codifican las instrucciones y más concretamente a la **Figura 3. Codificación de las instrucciones según el formato en RISC-V**. veremos que en estas posiciones de la instrucción se halla el campo denominado "Opcode" que indica al controlpath de que formato es la instrucción y que señales de salida deberá activar este dependiendo del formato. Ahora se explicará cada una de las señales de salida del controlpath así como a que dispositivos del datapath están conectados y como los controla.

- **Branch y Zero:** Estas dos señales se utilizan para realizar, o no, los saltos condicionales. Si el formato de la instrucción que se va a ejecutar es del tipo B, es decir, de salto condicional, la señal Branch estará activa. En cuanto a la señal de Zero, esta es la procedente de una de las salidas de la ALU y se encargará de comprobar si el salto es efectivo o no (por ejemplo, en el caso de la instrucción BNE (el salto se debe realizar si los dos operandos son diferentes) hará la comprobación de si estos dos números son diferentes o no) en caso afirmativo, la señal de Zero se activará, en caso contrario no. Las dos señales de Branch y Zero deberán de estar activas para que el salto condicional sea efectivo, esto se consigue mediante una simple puerta AND de dos entradas (que se puede apreciar en el esquema, arriba a la derecha) a las que irán las señales explicadas. La salida de la AND irá al selector de un multiplexor que decidirá cuál de los dos sumadores deberá hacer la suma al registro PC, bien el sumador que añade el valor para apuntar a la dirección donde se encuentra la instrucción inmediatamente siguiente, o bien, el sumador que se encarga de calcular el valor a añadir cuando un salto a una instrucción cualquiera se realiza.

- **MemRead:** Esta señal se trata de una habilitación de lectura para la memoria de datos, en el caso de este procesador no se ha utilizado, ya que, en los únicos casos en la que los datos leídos de la memoria de datos son necesarios son en las de operaciones de carga a la memoria de registros. Para decidir si la memoria de registros guarda los datos que le llegan de la memoria de datos simplemente podemos utilizar la señal de habilitación de escritura de la memoria de registros.

- **MemtoReg:** La señal decide lo que se va a escribir en la memoria de registros mediante un multiplexor, esta elegirá si se escribe los datos procedentes de la ALU (cualquier operación de registro a registro o de registro con inmediatos, por ejemplo) o los datos procedentes de la memoria de datos (en este caso serían las operaciones de carga como lw).

- **ALUOp:** Se trata de una señal de control que indica a la "ALU control" de que formato es la instrucción que se va a ejecutar.

- **MemWrite:** Señal de habilitación de escritura para la memoria de datos, esta se activará solamente para las instrucciones que sea necesario guardar un dato en la memoria de datos (como por ejemplo las instrucciones del tipo "storage").

- **ALUSrc:** Señal que irá a un multiplexor y decidirá cuál es el segundo operando de la ALU, si bien el segundo registro leído de la memoria de registros (para instrucciones del tipo registro a registro, por ejemplo (formato R)) o bien de la salida del generador de inmediatos (para instrucciones del formato I o para instrucciones de salto, por ejemplo).

- **RegWrite:** Señal de habilitación de escritura para la memoria de registros. Habilitará la escritura en esta memoria si la instrucción así lo precisa.

- **ALU control:** Este módulo, mediante la señal ALUOp (para saber que formato de instrucción se está ejecutando) y de la propia instrucción, se encargará de indicar que operación debe realizar la ALU.

4.1.1.3 Ejemplos de ejecución de instrucciones

Ahora que se ha explicado todos los componentes tanto de la parte del datapath como la del controlpath, a continuación, se explicarán varios ejemplos de ejecución de instrucciones, dependiendo del formato de instrucción, donde se puede apreciar como estos dos componentes del procesador RISC-V trabajan en conjunto. Esto tiene como objetivo explicar de una forma más general cómo funciona el procesador, ya que las explicaciones anteriores estaban más enfocadas a explicar solamente los componentes de este.

Para que sea más fácil la explicación, se utilizarán esquemas de los componentes del procesador que son utilizados dependiendo de la instrucción y las señales que actúan sobre estos.

- **Instrucciones del formato R:**

Para explicar la ejecución de este tipo de instrucciones en el procesador, se pondrá como ejemplo la instrucción de este formato “add”, la cual es una simple suma entre dos registros y se guarda en el registro destino.

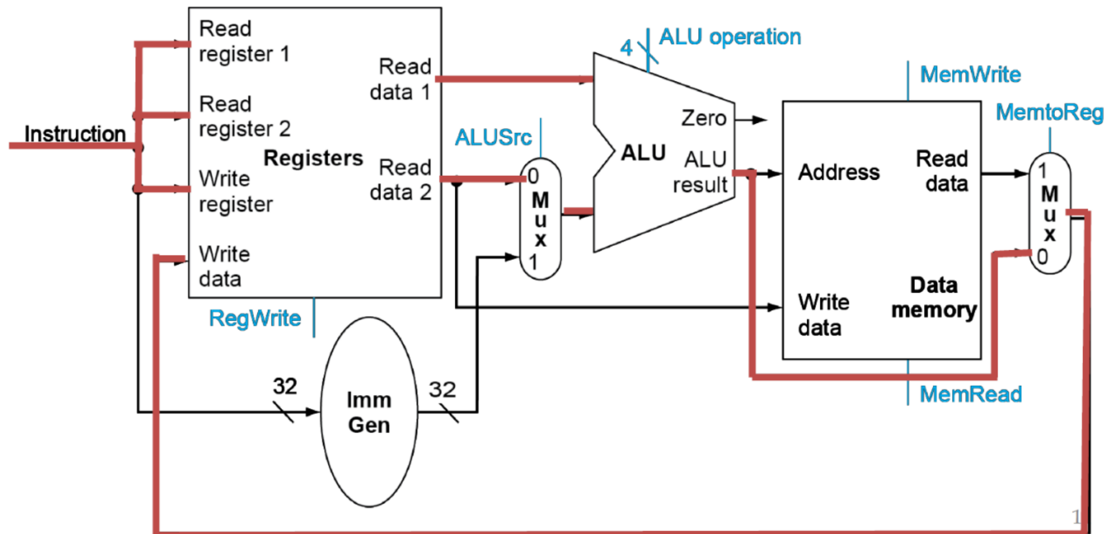

Figura 20. Señales y componentes utilizados en las instrucciones del formato R

Fijándonos en la figura 20, vemos que la señal de la instrucción se divide para llegar a tres de las entradas de la memoria de registros. En el “Read register 1” la instrucción introduce la dirección del primer registro de la suma, en “Read register 2” lo mismo para el segundo registro de la suma y en “Write register” será el registro donde se guardará el resultado de la operación (registro destino). “Read data 1” será el dato contenido en la dirección que apunte “Read register 1” e irá directamente a la primera entrada de la ALU, “Read data 2” será lo mismo pero en la dirección de “Read register 2” y esta a diferencia de la señal anterior no irá directa a la segunda entrada de la ALU, sino que, la señal procedente del controlpath, “ALUSrc”, se encargará de elegir esta señal o la de la salida de inmediatos. En nuestro caso ALUSrc tendrá un valor de “0” para elegir como segunda entrada a la ALU la señal de “Read data 2”, ya que el formato R es un formato que opera con registros. Teniendo en las dos entradas los dos registros, la ALU, con la señal de “ALU operation” realizará la operación que esta le indique, en este ejemplo, la operación de suma de los registros. El resultado de la suma saldrá por la salida de la ALU “ALU result” e irá a un multiplexor, este elegirá esta salida de “ALU result”, o bien, la procedente de la memoria de datos. Puesto que estamos realizando una instrucción de suma entre registros y no una de carga de datos “load” la señal de “MemtoReg”, selector de dicho multiplexor, seleccionará la señal de “ALU result”. Por último, la señal de “RegWrite” de la memoria de registros estará activa para permitir la escritura del resultado de la operación en el registro destino.
Todas las instrucciones del formato R se ejecutan igual en el procesador, la única señal que cambia es la ALUOp.

- **Instrucciones del formato I:**

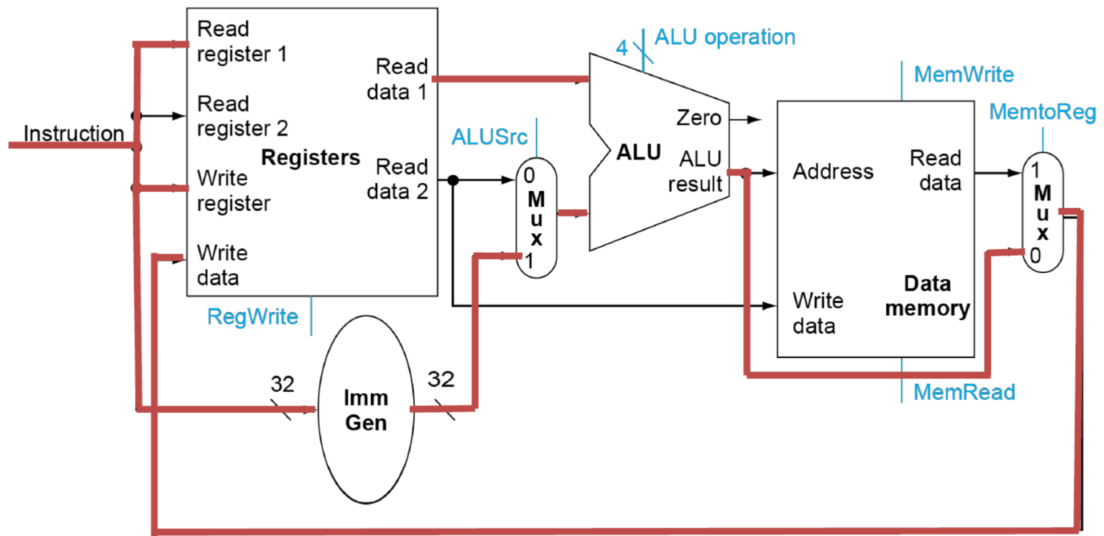

Figura 21. Señales y componentes utilizados en las instrucciones del formato I

Las instrucciones pertenecientes al formato de inmediatos se ejecutan de una forma muy parecida al formato explicado anteriormente. La única diferencia es que la señal de ALUSrc en este caso hará que seleccione la salida procedente del generador de inmediatos, por lo que el segundo operando de la ALU en vez de ser un registro será un inmediato procedente de este módulo.
Al igual que las instrucciones del formato R, las instrucciones del formato I todas se ejecutan de la misma forma en el procesador, únicamente cambia la señal de ALUOp.

- **Instrucciones de carga (lw):**

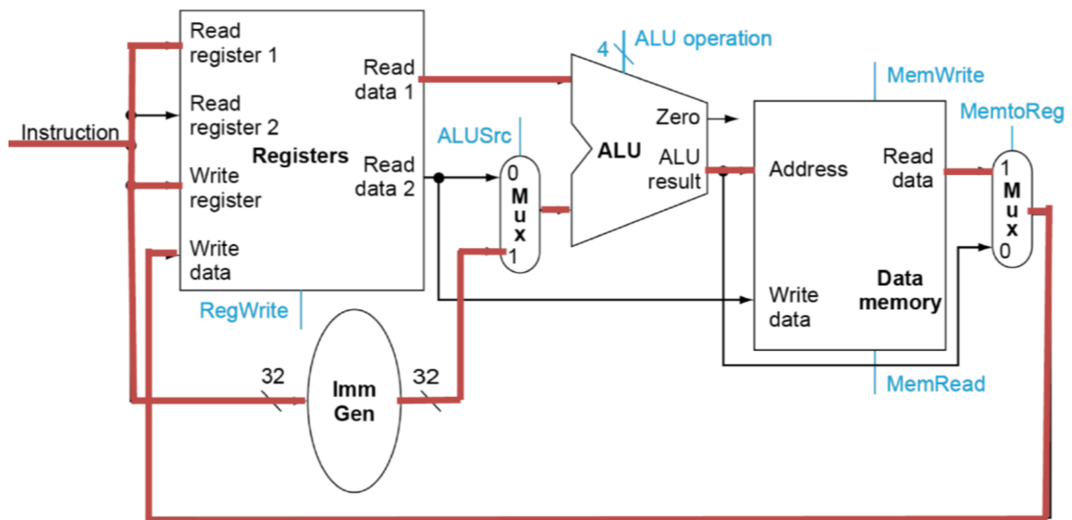

Figura 22. Señales y componentes utilizados en las instrucciones de carga

Como podemos ver en la figura 22, el procesador hará uso de las siguientes entradas y salidas. Empezando por la memoria de registros, “Read register 1” es el registro que contiene la dirección de la memoria de datos de la que se quiere hacer la carga. “Write

register" es el registro donde se hará la carga. "Read data 1" será el valor contenido en "Read register1", es decir, la dirección de la memoria de datos en la que se encuentra el valor que vamos a realizar la carga. Esta señal irá a la primera entrada de la ALU, en cuanto a la segunda entrada de esta, no utilizaremos otro registro, sino, la salida de inmediatos, por lo que la señal de ALUSrc seleccionará esta salida. El procesador escoge como segundo operando un inmediato y este será sumado a "Read data 1" (por ejemplo: en la instrucción de carga lw x9, 4(gp) , x9 será el registro donde se guardará el valor contenido en la dirección de la memoria de datos al que apunte el registro gp (este es el registro que irá a la ALU)+4 (este 4 es el valor del inmediato que irá al segundo operando de la ALU sumando 4 a la dirección contenida en gp, por lo que la memoria de datos devolverá el valor contenido en la dirección que apunte gp más 4 posiciones). El resultado de la ALU (que como se ha dicho en el ejemplo, será la dirección que apunte a la memoria de datos contenida en un registro, más el inmediato, el cual se trata de un offset) irá a la dirección de la memoria de datos, esta leerá el contenido de la dirección y el contenido se guardará en la memoria de registros en el registro que indique "Write data", por lo que la señal del controlpath de "RegWrite" estará activa para habilitar la escritura en la memoria de registros y la señal de "MemtoReg" seleccionará la salida de la memoria de datos en vez de la salida de la ALU directa.

- **Instrucciones "storage" (sw):**

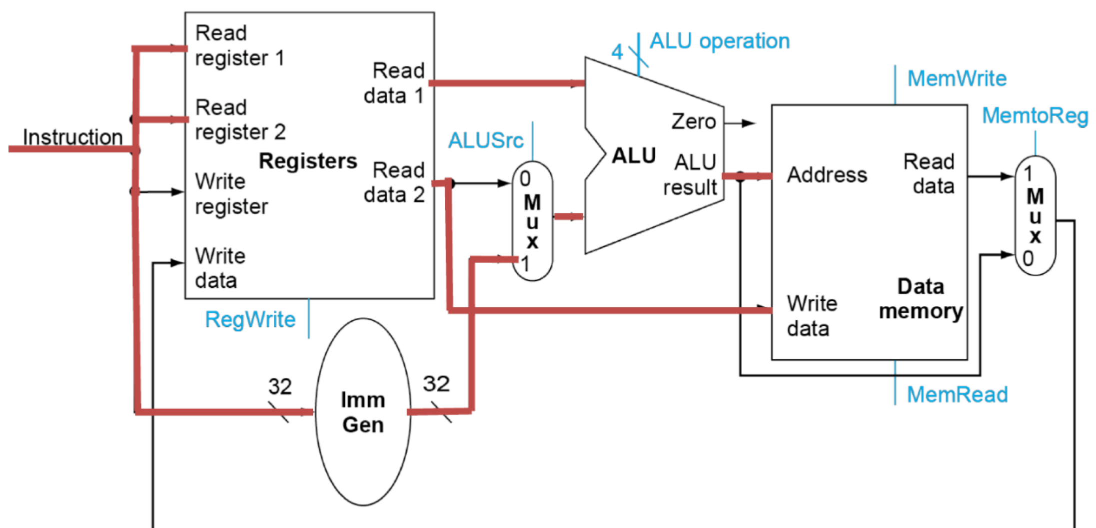

Figura 23. Señales y componentes utilizados en las instrucciones de "storage"

Ahora vamos a explicar la instrucción análoga a la explicada anteriormente, es decir, en vez de guardar un dato de la memoria de datos a la memoria de registros, el procesador hará la operación contraria, guardar un dato de la memoria de registros a la de datos. Para ello, como podemos ver en la figura 23, el procesador hará uso de los dos registros de la memoria de registros mediante las señales "Read register" 1 y 2, esta memoria los sacará por sus señales de salida "Read data", la primera irá al primer operando de la ALU ya que este registro es el que contiene la dirección de la memoria de datos donde el valor del registro se va a guardar, más un offset (que puede ser cero), que vendrá dado por la salida del generador de inmediatos. La salida del generador de inmediatos irá al segundo operando de la ALU (por lo que la señal de ALUSrc del multiplexor deberá seleccionarlo) (esta parte es idéntica a la de las instrucciones de carga), la salida de la ALU irá a la entrada de "Address" de la memoria de datos. En cuanto a la segunda señal de salida de

"Read data" será el valor que la instrucción quiere guardar en la memoria de datos mediante la entrada de "Write data". Por lo tanto, la señal de control de "MemWrite" de la memoria de datos deberá estar activa para permitir la escritura.

- **Instrucciones de salto condicional (formato SB)**:

Por último, se explicará cómo funcionan las instrucciones de salto condicional. Estas constan de dos partes importantes. La primera es comprobar si se ejecuta la instrucción de salto o no, para ello la instrucción utilizará los dos registros de la memoria de registros mediante las señales de "Read register" 1 y 2, los valores contenidos en estos dos registros saldrán por "Read data" 1 y 2 hacia los operandos de la ALU (ALUSrc deberá seleccionar la salida de la memoria de datos en vez de la del generador de inmediatos) la ALU pues será la encargada de comprobar si se cumple la condición del salto (por ejemplo en beq que los dos operandos sean iguales, en bne que sean diferentes, etc), si esta se cumple, la señal de salida "Zero" de la ALU se activará y esta junto a la señal de "Branch" del controlpath (esta señal siempre estará activa para las instrucciones de salto) irán a una puerta AND y de esta a un multiplexor para seleccionar el sumador de cálculo de la dirección de salto en vez del sumador normal que solamente suma una posición al PC. La segunda parte importante de este tipo de instrucciones es el cálculo de la dirección de salto, para ello simplemente se calculará mediante el generador de inmediatos y el sumador del esquema que se sitúa arriba a la derecha, en el que una de las entradas será la citada del generador de inmediatos y la segunda será el valor actual del registro PC.

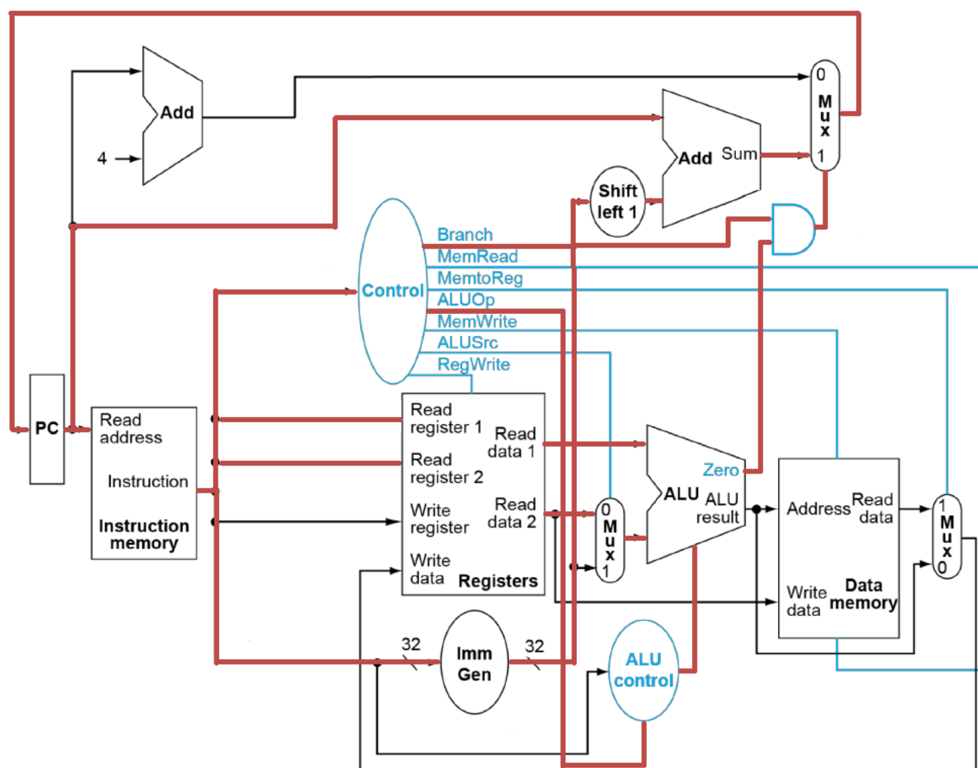

Figura 24. Señales y componentes utilizados en las instrucciones del formato B

4.1.1.4 Ventajas y desventajas de la versión "Single Cycle"

- Esta versión del procesador ejecuta una instrucción por ciclo, esto a priori podría parecer una ventaja, pero en realidad limita en gran medida la frecuencia de reloj a la que pueda trabajar el procesador. Esta se tendrá que ajustar acorde al retardo combinacional mayor del procesador, que es cuando se ejecuta las instrucciones de carga, ya que, utilizan todos los componentes del datapath en serie.

- Una ventaja que presenta esta versión en comparación a la que se va a explicar más adelante, es que esta versión del procesador no tiene ningún tipo de riesgos.

4.1.2 Desarrollo del procesador segmentado

4.1.2.1 Introducción

Una vez expuesto el procesador "Single Cycle" ya se puede explicar como a partir de esta versión se desarrolló la versión segmentada. Esta será la versión que se introducirá en el entorno de verificación anteriormente creado.

Con el objetivo de facilitar la explicación lo primero que introduciremos será el esquema de la nueva versión.

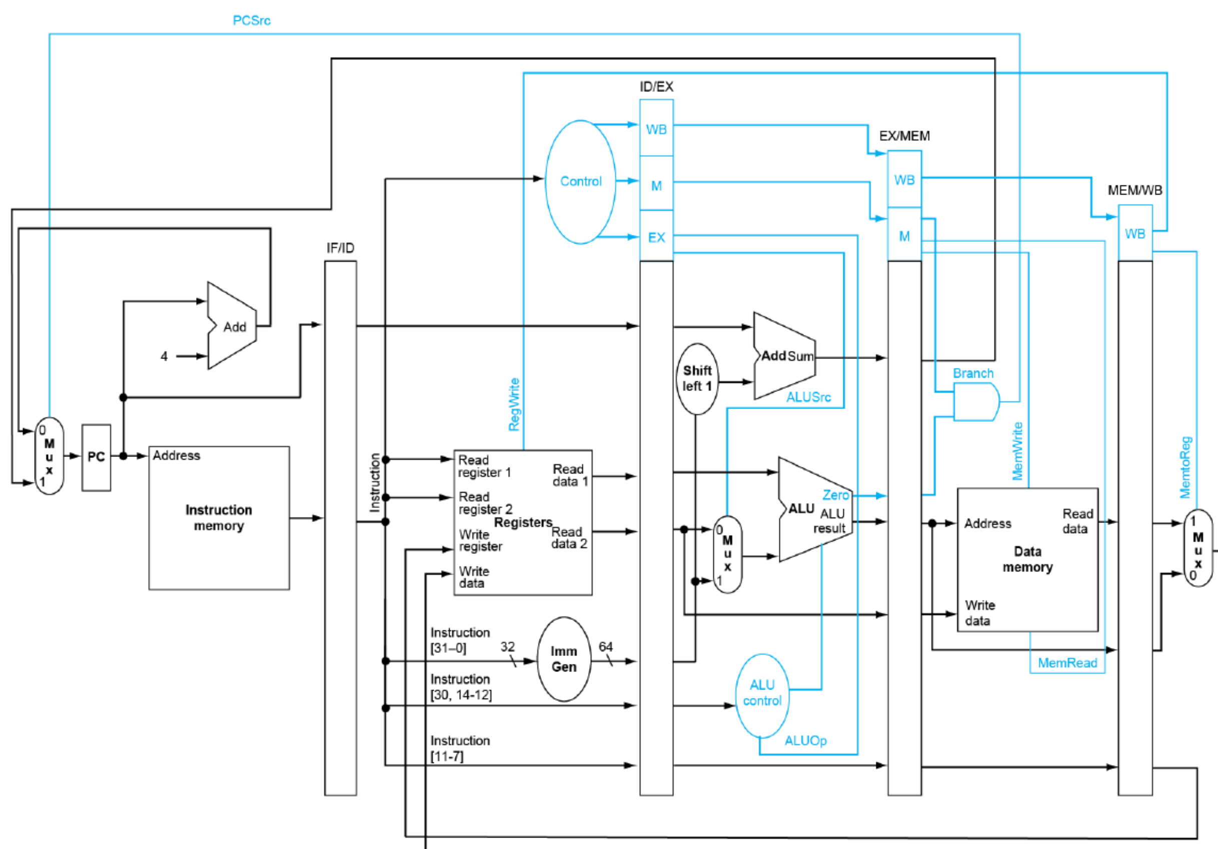

Figura 25. Esquema del procesador RISC-V segmentado

La gran diferencia que trae esta versión del procesador con la versión monociclo, es que se han introducido registros entre todas las etapas, registrando todas las señales del nuevo procesador entre una etapa y la siguiente.

La filosofía de esta nueva versión es de aumentar la frecuencia de reloj ajustándola para poder ejecutar la fase de ejecución con más retardo y que cada uno de los componentes del procesador este trabajando constantemente. Una vez ejecutado todas las operaciones necesarias en una fase, al siguiente golpe de reloj esta fase empezara trabajar en la siguiente instrucción, la anterior instrucción pasará a la siguiente fase, pero esto no quiere decir que se haya ejecutado en su totalidad esta. En contraposición, el procesador monociclo se ajusta la frecuencia de reloj para cubrir el retardo combinacional máximo de todos los componentes del datapath, no de una única fase de la instrucción.

Para explicar con más facilidad esto, podemos ver en la siguiente imagen una comparativa de como ejecutan las instrucciones las diferentes versiones del procesador, así como la diferencia de tiempos de ejecución.

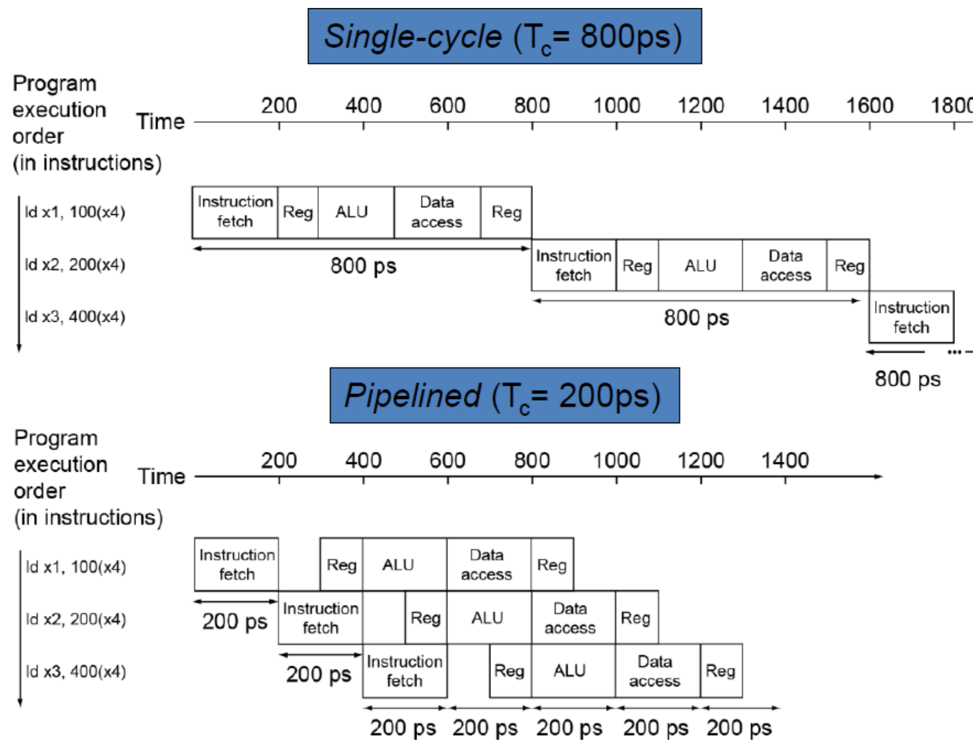

Figura 26. Comparativa de ejecución de instrucciones entre la versión del procesador monociclo y la segmentada respectivamente.

Resumiendo, en la parte superior de la imagen podemos ver como ejecuta las instrucciones la versión monociclo, esta ejecuta todas las fases de una instrucción (búsqueda de la instrucción, decodificación de la instrucción, …) en un solo ciclo de reloj, por lo que este se tendrá que ajustar al mayor retardo combinacional, el cual se trata de la suma de todos los componentes del datapath. La versión segmentada en vez de ejecutar todas las fases de una instrucción por cada ciclo de reloj ejecuta solamente una, por lo que la frecuencia máxima se podrá aumentar y solamente tendrá que cubrir el retardo de la fase de instrucción más lenta.

4.1.2.2 Ventajas y desventajas de la versión segmentada

- La principal ventaja de esta versión es la que se ha explicado con anterioridad, es decir, que esta versión no ejecuta todas las fases de una instrucción en el mismo ciclo sino, ejecuta una fase por ciclo, por lo que podemos aumentar mucho la frecuencia del reloj aumentando así la eficiencia del procesador.

- Esta segmentación trae consigo una serie de riesgos para el procesador, son los siguientes:

    - **Riesgos de datos:** Los riesgos de datos se producen cuando en una instrucción no se ha terminado aun de hacer un cálculo y/o actualizar el correspondiente en la memoria de registros, una instrucción que ya está en ejecución se encontrará en una de las fases en las que requiere este dato actualizado, no lo estará y la instrucción obtendrá un dato desfasado. Como se puede observar, estos riesgos se producen si existe una dependencia entre instrucciones que se vayan a ejecutar próximas entre sí.

    - **Riesgo de datos por carga:** Este tipo de riesgo es muy parecido al anterior, el resultado es el mismo (se operará con un dato desfasado) pero el origen es distinto. Este se produce cuando se ejecutan operaciones de carga "load" y las instrucciones inmediatamente siguientes quieren utilizar el dato cargado, el procesador no habrá tenido tiempo de completar la operación de carga, y las instrucciones que deseaban utilizar este dato, utilizan uno desfasado.
    Tanto este tipo de riesgos, como los explicados en el anterior punto, se pueden solventar mediante una buena programación con el objetivo de evitar estos tipos de riesgos o bien introduciendo burbujas (una burbuja es una instrucción que no hace nada pero que el procesador aun así ejecuta) entre las instrucciones dependientes.

    - **Riesgo de control:** Cuando se quiere ejecutar una instrucción de salto, hasta que el procesador realiza la comprobación de si se ha de saltar o no, tres instrucciones nuevas ya se han empezado a ejecutar. Si el salto es efectivo, estas tres instrucciones se estarán ejecutando cuando no deberían. En este caso se tienen que introducir burbujas para que el procesador no ejecute instrucciones innecesarias.

4.1.2.3 Mejoras del procesador segmentado

Las mejoras que se han aplicado al procesador básico segmentado en este trabajo han sido la elaboración de un dispositivo denominado "Data Forwarding" con el fin de eliminar la primera clase de los riesgos explicados en el apartado anterior, es decir, eliminar los riesgos de datos.

Este módulo se encarga de detectar si un dato de una instrucción es dependiente de una anterior que aún se esté ejecutando, si es así, el módulo lo detecta y aunque no se haya terminado de realizar la actualización del dato en el registro, este lo adelanta.

Este dispositivo se añade al procesador y el esquema con este quedaría de la siguiente forma:

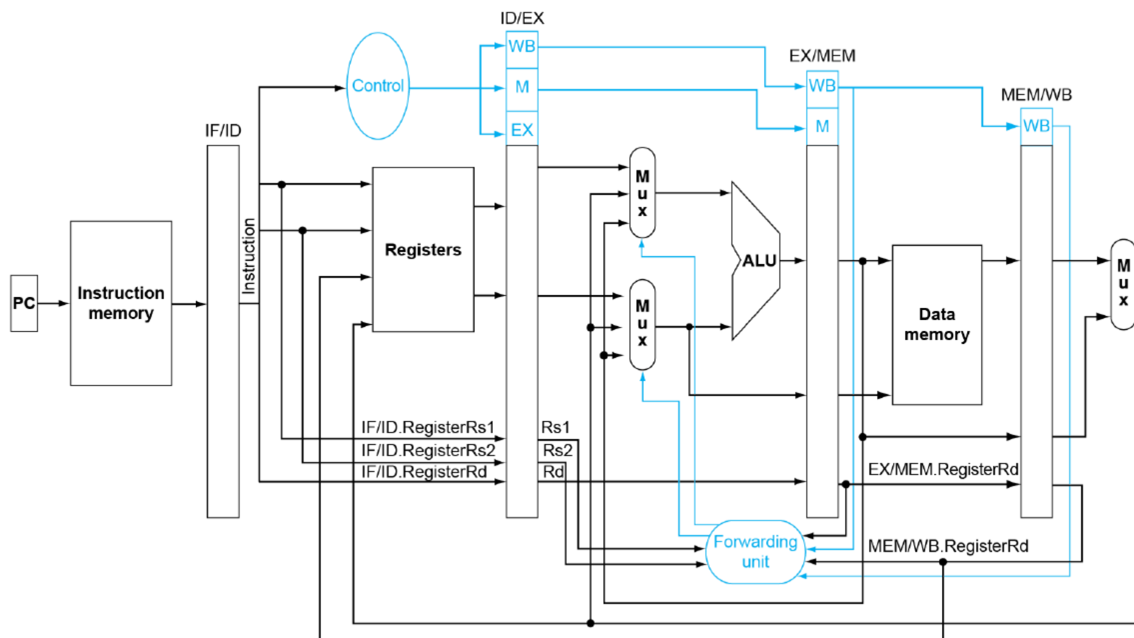

Figura 27. Esquema simplificado del procesador segmentado junto con la unidad "Data Forwarding"

La unidad de "Forwarding", como podemos ver en el esquema, necesitará como entrada ciertas señales procedentes del controlpath, más concretamente la señal registrada de habilitación de escritura de la memoria de registros de las fases MEM y WB. El dispositivo necesitará de estas señales para saber si se quiere hacer una escritura en la memoria de registros, ya que, si no fuera así, automáticamente no se necesitaría un adelantamiento de los datos. Las otras señales que necesitará el "Data Forwarding" serán las señales registradas de las direcciones de los registros 1 y 2 en la etapa EX y la señal registrada de la dirección del registro destino de las etapas MEM y WB. El dispositivo comprobará si los registros destino registrados de cualquiera de las dos etapas coinciden con alguno de los valores de las direcciones de los registros 1 y/o 2 registrados, si coinciden alguno de estos y además la señal registrada de habilitación de escritura de la memoria de registros esta activa para la supuesta fase, esto significará que hay dependencia de datos y la unidad de "Forwarding" deberá adelantar los datos a la fase EX desde la fase MEM o WB según convenga. Puede suceder que el dispositivo detecte que existe una dependencia de datos en las etapas MEM y WB simultáneamente (la señal registrada del registro destino en la fase MEM coincide con los dos o con alguno de los registros fuente en la etapa EX y la señal de habilitación de escritura de la memoria de registros de esta etapa esta activa y además están todas las mismas condiciones análogas para la etapa de WB), si se da el caso, el "Data Forwarding" solo adelantará el dato más reciente, es decir, el que se encuentre en la etapa MEM.

La salida de la unidad de "Forwarding" serán dos señales que irán a las entradas de selección de dos multiplexores diferentes. Las entradas de estos multiplexores serán el valor del registro fuente leído de la memoria de registros (registro fuente uno hacia un multiplexor y registro fuente dos hacia el otro), la señal del registro destino de la etapa MEM y esta misma señal, pero de la etapa WB. Mediante estas señales de selección, la unidad elegirá la primera entrada del multiplexor explicada cuando no se necesite adelantamiento de los datos, la segunda explicada si existe dependencia de datos entre la etapa MEM y la EX (también elegirá esta si hay dependencia simultanea entre MEM y EX, y, WB y EX) y la tercera si existe dependencia entre WB y EX.

4.2 Modificaciones necesarias en el entorno de verificación de la placa PYNQ

Una vez configurado el entorno de verificación en la placa y diseñado el propio procesador, será necesario realizar algunas modificaciones en estas dos partes del trabajo para que puedan funcionar en conjunto.

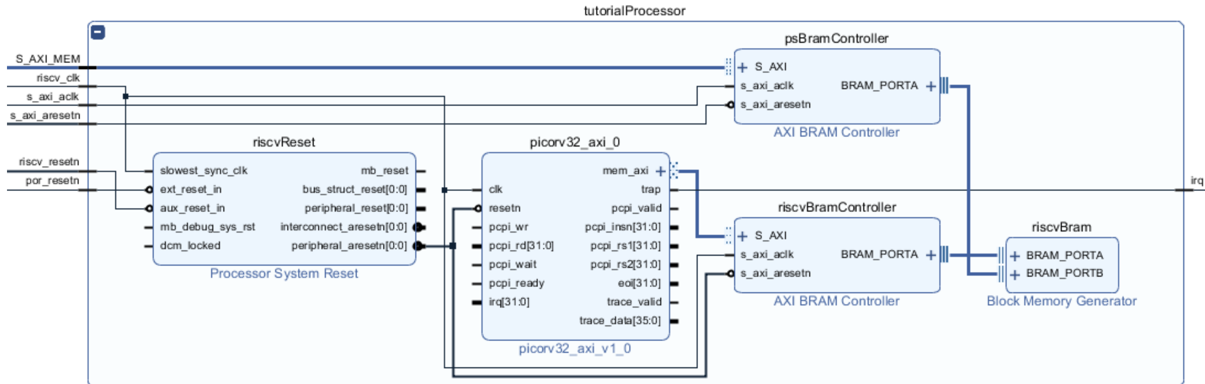

Figura 28. Esquema donde se encapsula el procesador

En la figura 28 podemos apreciar el esquema de uno de los módulos que componen el proyecto de vivado para la creación del entorno de verificación dentro de la placa. Este módulo es donde se introduce la ip del procesador. En el extremo de la derecha podemos apreciar un dispositivo que se llama "riscvBram" este módulo es una memoria de doble puerto, en el puerto A accederá el procesador diseñado y desde el puerto B se podrá acceder mediante las notebooks para realizar las comprobaciones del buen funcionamiento del procesador. Como se puede ver, el proyecto que nos facilita el usuario de GitHub, por defecto solo utiliza una memoria para el procesador (arquitectura Von Neumann), al contrario que el que se ha diseñado en este trabajo, que utiliza dos memorias (memoria de instrucciones y memoria de datos (arquitectura Harvard)). Así pues, el primer paso para poder cohesionar estos dos elementos será añadir una memoria adicional.

Los bloques de memoria del proyecto tienen las siguientes entradas y salidas:

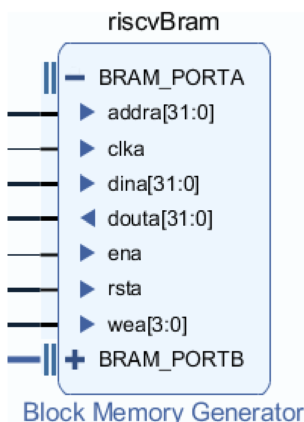

Figura 29. Bloque de memoria en vivado

Esta memoria generada es muy parecida a las vistas en los esquemas de los procesadores anteriormente explicados. La diferencia más remarcable en comparación con las otras memorias son que este tipo es de doble puerto, es decir, se puede acceder a ella desde dos dispositivos a la vez. En cuanto a las entradas y salidas no difieren mucho, pero algunas no funcionan de forma totalmente igual. La entrada "addra" es simplemente la dirección de las anteriores memorias y funciona igual, "clk" es la entrada de reloj. La señal "dina" es el dato de entrada y "douta" el de salida (funcionan igual que las otras memorias). La entrada de habilitación "ena" sirve tanto de habilitación de escritura como de lectura. Esta memoria dispone de una entrada de reset "rsta" activa a nivel alto. Por último, la entrada de "wea", se trata de una habilitación de escritura para la memoria, pero la diferencia es que esta señal es de 4 bits y no de un bit, porque esta señal permite que se escriba un número determinado de bits en vez de todo el dato de entrada. Por ejemplo, si wea=0000 no escribirá ninguno de los bits del dato de entrada, si wea=0001 escribirá los 8 bits menos significativos, si wea=0011 escribirá los 16

bits menos significativos y así. Puesto que la señal de habilitación de escritura que se utiliza en el procesador es de un solo bit, esto se tendrá que modificar simplemente haciendo que la señal de salida de habilitación de escritura que sale del procesador valga 1111 (el procesador diseñado solo tiene direccionamiento de 32 bits) cuando se quiera escribir y 0000 cuando no.

En el block design se añadirá un bloque de memoria adicional, teniendo ya dos de estos bloques, uno hará el rol de memoria de instrucciones mientras que el otro de datos. Al introducir un nuevo bloque de memoria también tendremos que crear un controlador AXI para el puerto b de esta, esto es necesario para comunicarse con el resto del proyecto ya que este utiliza una interfaz de comunicación de tipo AXI.

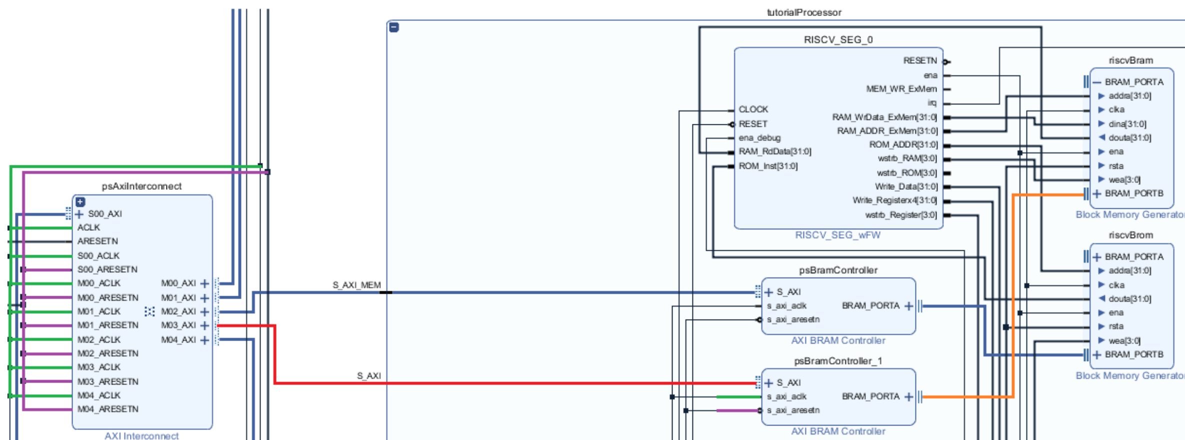

Figura 30. Conexiones que realizar cuando se introduce el nuevo bloque de memoria

Una vez introducido el nuevo controlador AXI (psBramController_1) el bus de salida se tendrá que conectar al puerto b de la memoria (en la figura 30 esta conexión es la naranja). El bus de entrada AXI irá al interconector AXI que hay en el proyecto, pero para esto se deberá crear una nueva interfaz maestra dentro de este, para ello se hará doble clic en el módulo psAxiInterconnect y añadiremos uno más en la opción de "Number of Master Interfaces". El nuevo bus de salida creado irá al bus de entrada del controlador AXI de la memoria (conexión roja) y las nuevas entradas del interfaz 3, en nuestro caso, serán de reloj y de reset, cada una de estas irá conectada a las señales de reloj (conexiones verdes) y de reset (conexiones moradas) de las otras interfaces respectivamente, también se tendrán que conectar a estas las señales de reloj y de reset del controlador AXI como corresponde.

Ahora se introducirá la ip del procesador diseñado siguiendo los pasos de ***Segunda notebook: "Creating A Bitstream"***. Lo primero será conectar al procesador las señales de reloj y reset, estas serán la entrada al bloque de "tutorialProcessor" "riscv_clk" para el reloj y la salida del bloque "riscvReset" "peripheral_aresetn" para el reset. Se conectarán todas las señales del procesador con las señales de las nuevas memorias de datos e instrucción como corresponda. Se crearán constantes para las siguientes señales de las memorias que se habrán quedado sin conectar:

- ena: Tanto para la memoria de datos como de instrucciones se creará un valor constante de "1" ya que siempre querremos que este activa la lectura y escritura. La escritura como se ha explicado anteriormente se habilitará o no mediante la señal wea.

- rsta: Este reset es activo a nivel alto, por lo que se pondrá un valor constante de “0” a estas entradas de la memoria de datos y de instrucciones. Desde el procesador no querremos realizar resets a las memorias, de ser necesario se harán mediante los puertos b de las memorias.

- wea: Puesto que no querremos escribir nunca en la memoria de instrucciones desde el procesador, se creará una constante de 4 bits con valor de 0 para esta entrada, solo de la memoria de instrucciones del puerto a.

Por último, mediante el “address editor” asignaremos direcciones a los controladores AXI de la memoria de instrucciones y de datos. Para el “Program counter” del procesador se tendrá que haber creado un “terminal count”, es decir, cuando el PC llegue a cierto valor este no deberá aumentar, de lo contrario, se producirían desbordamientos constantemente y el procesador ejecutaría una y otra vez el contenido de la memoria de instrucciones. También el rango que le asignemos al controlador AXI de la memoria de instrucciones deberá ser menor o igual al PC máximo que se haya configurado, de lo contrario, la diferencia de estos dos valores será la porción que se volverá a ejecutar de la memoria de instrucciones.

Una vez añadida la segunda memoria, realizado las conexiones necesarias y haber mapeado las direcciones correctamente de los controladores AXI de las memorias, ya se podrá generar el “bitstream” del proyecto e introducirlo en la placa.

4.3 Verificación del procesador mediante Jupyter Notebook

Para las verificaciones del procesador se han utilizado Jupyter Notebooks, la estructura que se ha seguido en ellas es la siguiente:

1. Creación del Overlay mediante el archivo .bit.
2. Compilación de un programa que será ejecutado por el procesador.
3. Introducción del programa compilado y verificación de su correcta ejecución por parte del procesador.

Para el primer punto utilizaremos el siguiente código que ya vimos en el apartado ***3.2.5***.

```python
import sys
sys.path.insert(0, '/home/xilinx/RISC-V-On-PYNQ/riscvonpynq/picorv3
2/')
sys.path.append('/home/xilinx/RISC-V-On-PYNQ/')


from tutorial.tutorial import TutorialOverlay

overlay = TutorialOverlay("/home/xilinx/RISC-V-On-PYNQ/riscvonpynq/
picorv32/tutorial/tutorial.bit")
```

Mediante la clase sys y los métodos sys.path.insert y sys.path.append, añadiremos los path para acceder a los métodos creados por el usuario de GitHub. Mas concretamente, se quiere acceder al método de TutorialOverlay que se encuentra en el fichero tutorial.py de la carpeta tutorial (en este caso). Este método se encargará de generar el "overlay" mediante el fichero tutorial.bit pasándole como argumento la dirección donde se encuentra este.

Para el segundo punto, uno de los programas que se han elaborado es el de la serie de Fibonacci. Los programas realizados se han escrito mediante las instrucciones del ISA del RISC-V directamente, ya que no se ha podido introducir programas escritos en lenguajes en alto nivel como C (se pueden compilar programas en C, C++ y ensamblador). Debido a que no se sabe cómo se realiza la compilación de los lenguajes de alto nivel y su posterior transformación a lenguaje ensamblador, estos programas escritos en lenguaje de alto nivel pueden presentar instrucciones que un procesador no tan avanzado pueda no ejecutar, o que no respete ciertos riesgos que este mismo podría tener. Por eso se ha decidido utilizar el lenguaje ensamblador de forma predeterminada para la elaboración de los programas de verificación.

El siguiente programa es la serie de Fibonacci elaborado en lenguaje ensamblador:

```
%%riscvasm test_asm overlay.tutorialProcessor

.global main

main:
addi x9, x9, 0 #iniciacion de la serie de fibonacci
addi x8, x8, 1 #iniciacion de la serie de fibonacci
nop
nop
sw x9, 0(gp) #guardamos x9 para su uso
sw x8, 4(gp) #guardamos x8 para su uso
nop
addi x11, x11, 10 #ponemos los numeros de la serie de fibonacci que queremos sacar
addi x12, x12, 2
addi gp, gp, 8
Loop: addi gp, gp, -8
nop
nop
nop
lw x9, 0(gp)
nop
nop
addi gp, gp, 4
nop
nop
nop
lw x8, 0(gp)
nop
nop
add x10, x8, x9
addi gp, gp, 4
nop
add x9,x8,x0 #copiamos lo que tenemos en el reg 8 al 9
add x8,x10,x0 #copiamos lo que tenemos en el reg 10 al 8
nop
nop
nop
sw x8, 0(gp)
nop
addi x12, x12, 1 #aumentamos el contador del bucle condicional
add x10,x0,x0 #reseteamos el registro auxiliar 10
addi gp, gp, 4
nop
beq x11,x12,Exit #si el registro 12 es igual al 11 el programa finaliza
nop
nop
nop
beq x0,x0, Loop
nop
nop
nop
Exit:
```

Figura 31. Celda para compilar el programa de Fibonacci

En la primera línea de la celda indicamos en que lenguaje se va a programar el programa (riscvasm para ensamblador, riscvc para C y riscvcpp para C++), que nombre tendrá (en este caso test_asm) y por último crear una nueva clase Processor.

Como podemos ver, muchas de las instrucciones son instrucciones nop, estas no realizan nada en el procesador, pero están ahí para crear burbujas y solventar los riesgos de datos por carga y por estructura explicados en el punto Ventajas y desventajas de la versión segmentada.

El programa guardará en la memoria de datos el número de valores que queramos de la serie de Fibonacci.

En el tercer punto introduciremos el programa compilado y se comprobará si se ha ejecutado de forma correcta. Ejecutaremos la siguiente celda para realizar todo esto:

```
import numpy as np
num_fibo=10;
overlay.tutorialProcessor.run(test_asm)
vector_proc=np.zeros((1,num_fibo))
vector_res=np.zeros((1,num_fibo))
vector_res[0,1]=1

for i in range(num_fibo):
    vector_proc[0,i]=overlay.tutorialProcessor.psBramController_1.mmio.read(i*4,4)

for i in range(num_fibo-2):
    vector_res[0,i+2]=vector_res[0,i]+vector_res[0,i+1]

resul=(vector_res==vector_proc) .all()

if resul==True:
    print('¡El test ha pasado correctamente!')
else:
    print('¡El test ha fallado!')

¡El test ha pasado correctamente!
```

Figura 32. Inserción del programa en el procesador y comprobación de la buena ejecución por parte de este.

El programa compilado anteriormente se ejecutará en el procesador mediante la ejecución del método Processor.run y le pasaremos como argumento el programa (overlay.tutorialProcessor.run(test_asm)). El resto de la celda será para comprobar que el programa se ha ejecutado en el procesador correctamente. Para ello mediante el primer bucle for leeremos y guardaremos el contenido de la memoria de datos del procesador en un vector hasta el valor que marque la variable llamada num_fibo, podremos realizar esto mediante el controlador AXI de la memoria (psBramController_1) y la clase mmio de las librerías PYNQ que permite acceder a un objeto Python a las direcciones de la memoria mapeada del sistema. La filosofía para realizar los test a los procesadores será esta, se elaborará un programa y se comprobará las memorias del procesador para ver si los valores guardados en esta son los correctos después de la ejecución. El segundo bucle calculará los valores de la serie de Fibonacci por sí mismo y lo guardará en otro vector. Se comparará estos dos vectores, si coinciden, significará que el procesador ha ejecutado bien el programa.

4.4 Mejoras del entorno de verificación

Por último, en este trabajo se ha decidido realizar ciertas mejoras en el proyecto de vivado asociado al entorno de verificación con el fin de disponer de alguna herramienta más para ayudar a realizar la verificación del procesador. Estas son las mejoras que se han hecho:

4.4.1 Visualización de la memoria de registros mediante Jupyter Notebook

Puesto que para hacer las comprobaciones de si el procesador funciona correctamente solo se disponía de la memoria de datos para comprobar su contenido, se consideró oportuno también el poder visualizar el contenido de la memoria de registros.

Para integrar esta funcionalidad, el primer paso será añadir otro bloque de memoria como se explicó en el punto ***Modificaciones necesarias en el entorno de verificación de la placa PYNQ*** y las correspondientes conexiones explicadas en este mismo punto.

Este nuevo bloque de memoria servirá como copia de la memoria de registros que se encuentra en el interior de nuestro procesador. Para esto habrá que poner como salidas del bloque del procesador las siguientes señales que van a la memoria de registros interna.

- Write register: Señal que indica que registro se va a escribir. Esta señal irá a la entrada del nuevo bloque de memoria addra.
- Write data: Señal que indica que valor se escribirá en el registro deseado. Write data irá a la entrada dina.
- RegWrite: Habilitación de escritura de la memoria de registros. Esta señal de un bit se tendrá que duplicar y crear una nueva de 4 bits, cuando la señal de un bit este activa la de 4 tendrá que valer “1111” en caso contrario tendrá que valer “0”. La habilitación de la memoria de registros interna deberá ir a la entrada del bloque de memoria wea.

Con estas conexiones, todo dato que se escriba en cierto registro de la memoria de registros interna se guardará en la nueva copia que se ha creado. El esquema quedaría de la siguiente forma:

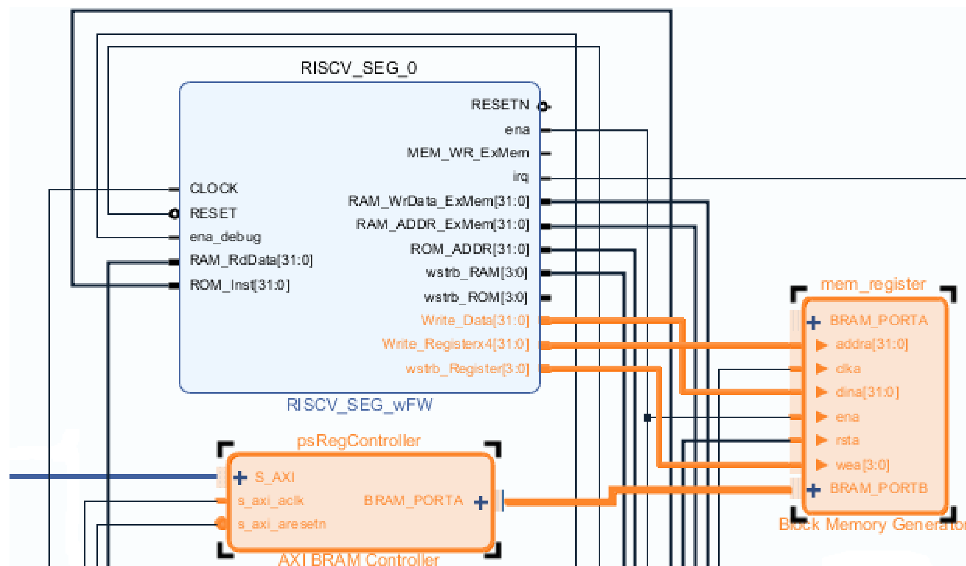

Figura 33. Conexiones necesarias para realizar la copia de la memoria de registros

A partir de ahora ya se podría ver el contenido de la memoria de registros mediante una notebook.

4.4.2 *Creación de un debug para ejecutar las instrucciones del procesador paso a paso*

Otra mejora que se consideró oportuna fue la elaboración de un tipo de debug que obligará al procesador a ejecutar las instrucciones paso a paso para poder así comprobar con más facilidad si este está funcionando correctamente, o en caso contrario, detectar donde se están produciendo los errores.

Se podrá alternar entre la ejecución normal y la ejecución paso a paso mediante el switch 0 de la placa PYNQ, para ejecutar las instrucciones paso a paso se utilizará el botón 0 de esta.

Lo primero pues será añadir los puertos de los botones, los interruptores y los leds que vamos a utilizar en este apartado en el proyecto de vivado. Para ello se tendrá que descargar el archivo .xdc maestro de la placa PYNQ que se podrá encontrar en la página https://reference.digilentinc.com/reference/programmable-logic/pynq-z1/start. Por defecto el proyecto incluye los botones y los leds, pero no los interruptores. Para añadir estos últimos se tendrá que añadir las siguientes líneas (del fichero descargado) al fichero .xdc que se encuentra en el apartado de "constraints" en vivado.

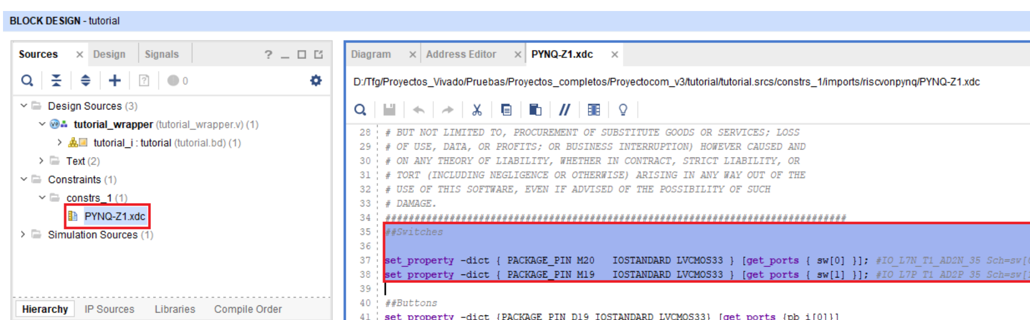

Figura 34. Como añadir los interruptores de la placa PYNQ al proyecto en vivado.

Ahora ya se podrá hacer uso de los interruptores en el proyecto de vivado, para ello se tendrá que crear un puerto de entrada de dos bits con el nombre de sw.

Para poder seleccionar entre el modo de ejecución normal y el paso a paso se tendrá que diseñar un módulo de selección de reloj como el que representa la siguiente imagen:

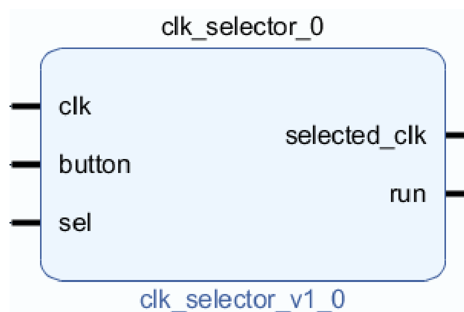

Figura 35. Esquema del selector de reloj

En un principio, este esquema simplemente funcionaba como un multiplexor, es decir como entrada este tendría la señal de reloj normal, la señal procedente del botón 0 de la placa PYNQ y la señal de selección que elegiría una de estas dos entradas mediante el interruptor 0. La idea era que en el modo paso a paso el selector sustituyera el reloj normal por una entrada de los botones de la placa, así pues, cada vez que el usuario pulsase dicho botón, en teoría, solamente estaría introduciendo un ciclo de reloj en el procesador sujeto a prueba. Los botones de la placa PYNQ

no tienen protección contra rebotes, por lo que la señal cuando son pulsados sería más o menos la siguiente:

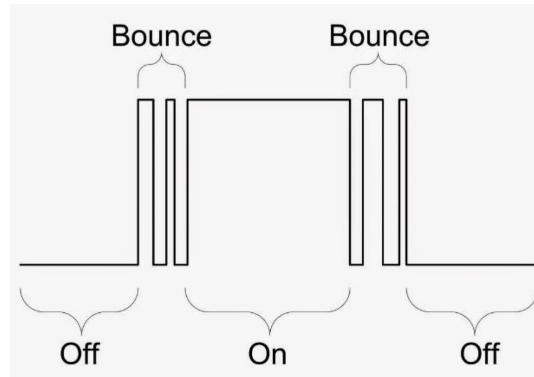

Figura 36. Señal aproximada generada por los botones de la placa

Al existir rebotes, en una sola pulsación del botón, el procesador puede entender que han sucedido varios flancos de reloj en una misma pulsación, por lo que este ejecutará más de una instrucción a la vez perdiendo toda su utilidad.

Para solventar este problema se decidió añadir un contador a este esquema para que una vez se pulsase el botón 0, el contador se iniciaría y periódicamente generase pulsos cuando llegase al final de cuenta, si se vuelve a pulsar este botón, el contador se pararía y no volvería a funcionar hasta que este se volviera a pulsar. La frecuencia del generador de reloj del proyecto es de 50MHz, se decidió que el contador tuviera 150 millones de cuentas para que así este generase un pulso de reloj cada 3 segundos más o menos.

La señal de salida run que se puede apreciar en el esquema del selector de reloj, va conectada al led 0 de la placa, e indica al usuario si el contador que genera los pulsos cada 3 segundos esta activo o no, ya que, al pulsar, debido a los rebotes del botón se podría pulsar este, pero no iniciar al contador (cuando el selector de reloj está en modo de ejecución paso a paso).

Un ejemplo de utilidad de este debug podemos verlo en la siguiente celda de una notebook:

```
import time
import os
from IPython.display import clear_output
interval=3;
#periodic_work(interval)
def periodic_work(interval):

    while True:

        #change this to the function you want to call, or paste in the code you want to run
        arr = overlay.tutorialProcessor.psRegController.mmio.array
        for i in range(32):  #128
            print(f'Memory Index {i:3}: {arr[i]:#0{10}x}')

        #interval should be an integer, the number of seconds to wait
        time.sleep(interval)
        clear_output()

periodic_work(interval)
```

Figura 37. Celda de una notebook para utilizar el debug paso a paso

Esta celda mostrará el contenido de la memoria de registros del procesador, con la peculiaridad de que se ejecutará periódicamente cada 3 segundos, por lo que se podrá observar cómo cambia el contenido de esta memoria con cada instrucción ejecutada en el modo debug.

Capítulo 5. Conclusiones y propuesta de trabajo futuro

Como se ha ido explicando en la presente memoria, el trabajo consta de dos partes bien diferenciadas, la elaboración de un procesador basado en arquitectura RISC-V y un entorno de verificación basado en una placa PYNQ.

En cuanto al diseño del procesador se partió desde una versión sencilla monociclo, se intentó hacer las primeras pruebas de verificación con esta versión, pero al tener que utilizar bloques de memoria de escritura y lectura síncrona no pudo ser así ya que esa versión necesita de memorias con lectura asíncrona. Por esta razón y debido a que la versión segmentada del procesador presentaba mayores ventajas, entre ellas una mejora de la eficiencia y por lo tanto una reducción considerable del tiempo de ejecución de instrucciones, se modificó el procesador registrando las señales entre etapa y etapa para así convertir esta versión en la versión mejorada segmentada. Una vez completada esta variante, se añadió la unidad de "forwarding" para corregir los riesgos de datos que la versión segmentada presenta, para que en caso de que el programa que vaya a ejecutar el procesador no se pueda ordenar las instrucciones para evitar estos riesgos. De esta forma no solo se evita la posible mal ejecución del programa (debido a riesgos de datos) sino que se evita tener que introducir burbujas en el procesador, por lo que no es necesario ejecutar instrucciones adicionales acelerando así la ejecución del programa. Por tanto, el procesador no solo se diseñó teniendo como objetivo la comprobación del entorno de verificación, sino que se diseñó como un elemento fundamental del trabajo y que este presentase cierta polivalencia.

Para el desarrollo del entorno de verificación, como se ha ido explicando durante la memoria, se ha partido de un trabajo ya existente en GitHub. El trabajo realizado para la elaboración de esta parte del proyecto no ha sido meramente la utilización de este entorno de verificación aportado directamente, sino que, el estudiante ha tenido que analizar el proyecto para entender en rasgos generales como funciona para poder modificarlo, debido a que este estaba orientado a procesadores con una arquitectura Von Neumann, es decir, con una única memoria de datos e instrucciones mientras que el procesador diseñado tiene una arquitectura tipo Harvard con una memoria exclusiva para datos y otra para instrucciones. El trabajo no solo ha consistido en dicha adaptación, sino que se han añadido bloques para facilitar la verificación del procesador, entre ellos el que se pueda observar una copia de la memoria de registros para detectar posibles errores en procesadores en desarrollo o en primeras fases de verificación o un debug hardware paso a paso que permite observar los cambios que se producen en las memorias tras la ejecución de cada instrucción. Por último, para completar este bloque del trabajo, se han elaborado varias notebooks con el propósito de acelerar la verificación. La idea pues, es la ejecución de una de las notebooks que hace ejecutar dos programas al procesador y comprueba que el contenido de la memoria de datos es acorde con lo ejecutado, si es así, la notebook indicará que el test ha sido pasado correctamente y por tanto es presumible que el procesador sujeto a verificación funciona correctamente. Si no pasa el test, se han programado otras notebooks que permiten al usuario comprobar el contenido de las memorias para poder así indagar y tratar de encontrar el problema. Por tanto, no solo se ha conseguido elaborar un entorno de verificación físico, sino que este es increíblemente dinámico y puede realizar verificaciones realmente rápidas, permitiendo también al usuario crear notebooks por su cuenta para añadir prestaciones a la verificación, realizar verificaciones concretas o de otro modo al que se proporciona en este trabajo.

Por lo tanto, en el presente trabajo se ha conseguido crear un entorno de verificación avanzado para procesadores del tipo RISC-V y un procesador de este mismo tipo para comprobar su buen funcionamiento, pero para estas dos partes del trabajo aún se pueden realizar algunas mejoras:

5.1 Mejoras en el procesador.

1. El procesador diseñado aun presenta ciertos riesgos como se ha explicado anteriormente, este se podría mejorar para que no tuviera riesgos de datos por carga y que realice predicciones de salto dinámicas para aumentar la eficiencia de la ejecución de instrucciones de salto.

2. Modificar el procesador para que permita el direccionamiento de 8 y 16 bits, ya que el actual solo puede realizar direccionamientos de 32 bits.

3. Aumentar el número de instrucciones que pueda ejecutar el procesador.

5.2 Mejoras en el entorno de verificación.

1. Poder crear los programas que se vayan a introducir en el procesador en lenguaje de alto nivel directamente y no necesariamente en ensamblador.

2. Creación de Notebooks avanzadas de verificación que puedan llevar al límite al procesador, otras para comprobar cuantas instrucciones pueden ejecutar el procesador, etc. Es decir, elaborar más notebooks para automatizar aún más el proceso de verificación.

Capítulo 6. Bibliografía

[1] RISC-V Foundation | Instruction Set Architecture (ISA), "RISC-V History" https://riscv.org/risc-v-history/ [Online].

[2] Patterson y Waterman "Guia practica de RISC-V. El Atlas de una Arquitectura Abierta".

[3] Krste Asanović and David A. Patterson "Instruction Sets Should Be Free: The Case For RISC-V" August 6, 2014

[4] Andrew Waterman1, Krste Asanović "The RISC-V Instruction Set Manual Volume I: Unprivileged ISA" CS Division, EECS Department, University of California, Berkeley, June 8, 2019

[5] Dustin Richmond "RISC-V-On-PYNQ" GitHub https://github.com/drichmond/RISC-V-On-PYNQ

[6] Python productivity for Zynq "Getting Started" https://pynq.readthedocs.io/en/v2.0/getting_started.html

[7] Dustin Richmond "RISC-V-On-PYNQ notebooks tutorial" GitHub https://github.com/drichmond/RISC-V-On-PYNQ/tree/master/notebooks/tutorial

[8] elinux.org | Toolchains https://elinux.org/Toolchains

[9] Información proporcionada en la asignatura de Integración de sistemas digitales (ISDIGI) de la ETSIT de la Universidad Politécnica de Valencia.