UNIVERSIDAD NACIONAL DE CÓRDOBA

FACULTAD DE CIENCIAS EXACTAS, FÍSICAS Y NATURALES

ARQUITECTURA DE COMPUTADORAS

Trabajo Practico Final

PROCESADOR MIPS SIMPLIFICADO

- **Integrantes:** Aguilar Mauricio 34496071
  Tarazi Pedro Esequiel 35035736
- **Docente:** Ing. Santiago Rodríguez

- **Fecha de Entrega:** 19/09/2018
- **Carrera:** Ingeniería en Computación

2018

INTRODUCCIÓN

El objetivo del presente trabajo es la implementación del pipeline del procesador MIPS. Se debe realizar el MIPS segmentado en las siguientes etapas:

- IF (Instruction Fetch): Busqueda de la instrucción en la memoria de programa.
- ID (Instruction Decode): Decodificacion de la instrucción y lectura de registros.
- EX (Execute): Ejecucion de la instrucción propiamente dicha.
- MEM (Memory Access): Lectura o escritura desde/hacia la memoria de datos.
- WB (Write-Back): Escritura de los resultados en los registros.

Ademas, se deben implementar las siguientes instrucciones, las cuales están detalladas en el MIPS IV Instruction Set:

- R-Type: SLL, SRL, SRA, SLLV, SRLV, SRAV, ADDU, SUBU, AND, OR, XOR, NOR, SLT.
- I-Type: LB, LH, LW, LWU, LBU, LHU, SB, SH, SW, ADDI, ANDI, ORI, XORI, LUI, SLTI, BEQ, BNE, J, JAL
- J-Type: JR, JALR

El procesador debe tener soporte para riesgos:

- Estructurales: Se producen cuando dos instrucciones tratan de utilizar el mismo recurso en el mismo ciclo.
- De datos: Se intenta utilizar un dato antes de que este preparado. Mantenimiento del orden estricto de lecturas y escrituras.
- De control: Intentar tomar una decisión sobre una condición todavía no evaluada.

Debe contar con Unidad de cortocircuitos y unidad de detección de riesgos.
El programa a ejecutar debe cargarse desde un archivo ensamblado. Debe implementarse un programa ensamblador, y debe transmitirse ese programa por UART antes de comenzar a ejecutar. Ademas, se debe incluir una unidad de Debug que envie información desde y hacia la PC.
La unidad de Debug debe enviar a través de UART los siguientes datos:

- Contenido de los latches intermedios
- Contenido de los 32 registros
- PC
- Contenido de la memoria de datos usada
- Cantidad de ciclos de clock desde el inicio

Modos de operación:

- Antes de estar disponible para ejecutar, el procesador esta a la espera para recibir un programa mediante la Debug Unit
- Una vez cargado el programa, debe permitir dos modos de operación:

- Continuo: se envía un comando a la FPGA por la UART y este inicia la ejecución del programa hasta llegar al final del mismo (HALT). Llegado a ese punto, se muestran todos los valores indicados en pantalla.
- Paso a paso: Enviando un comando por UART, se ejecuta un ciclo de clock. Se debe mostrar en cada paso los valores indicados.

DESARROLLO

El proceso de desarrollo del procesador fue realizado mediante divisiones de cada etapa con módulos. Es decir, cada etapa es un módulo verilog, el cual contiene un conjunto de submodulos que realizar una sola tarea cada uno. La idea principal fue desglozar el procesador en tantos módulos como tareas deba realizar, de manera de que cada uno sea lo mas sencillo posible.

Set de instrucciones implementadas

Tipo		Operación	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
			OPCODE						RS					RT					RD					SHAMT					FUNCT					
R	REGISTER	SLL	0	0	0	0	0	0	0	0	0	0	0						rd <- rt << shamt										0	0	0	0	0	0
		SRL	0	0	0	0	0	0	0	0	0	0	0						rd <- rt >> shamt										0	0	0	0	1	1
		SRA	0	0	0	0	0	0	0	0	0	0	0						rd <- rt >> shamt										0	0	0	0	1	0
		SLLV	0	0	0	0	0	0						rd <- rt << rs										0	0	0	0	0	0	0	0	1	0	0
		SRLV	0	0	0	0	0	0						rd <- rt >> rs										0	0	0	0	0	0	0	0	1	1	0
		SRAV	0	0	0	0	0	0						rd <- rt >> rs										0	0	0	0	0	0	0	0	1	1	1
		ADDU	0	0	0	0	0	0						rd <- rs + rt										0	0	0	0	0	1	0	0	0	0	1
		SUBU	0	0	0	0	0	0						rd <- rs - rt										0	0	0	0	0	1	0	0	0	1	1
		AND	0	0	0	0	0	0						rd <- rs AND rt										0	0	0	0	0	1	0	0	1	0	0
		OR	0	0	0	0	0	0						rd <- rs OR rt										0	0	0	0	0	1	0	0	1	0	1
		XOR	0	0	0	0	0	0						rd <- rs XOR rt										0	0	0	0	0	1	0	0	1	1	0
		NOR	0	0	0	0	0	0						rd <- rs NOR rt										0	0	0	0	0	1	0	0	1	1	1
		SLT	0	0	0	0	0	0						rd <- (rs < rt)										0	0	0	0	0	1	0	1	0	1	0
		JR	0	0	0	0	0	0	PC <- rs					0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0
		JALR	0	0	0	0	0	0	rd <- return_addr					0	0	0	0	0	PC <- rs					0	0	0	0	0	0	0	1	0	0	1
			OPCODE						BASE / RS					RT					OFFSET / INMEDIATE															
I	LOAD	LB	1	0	0	0	0	0											rt <- memory[base + offset]															
		LH	1	0	0	0	0	1											rt <- memory[base + offset]															
		LW	1	0	0	0	1	1											rt <- memory[base + offset]															
		LWU	1	0	0	1	1	1											rt <- memory[base + offset]															
		LBU	1	0	0	1	0	0											rt <- memory[base + offset]															
		LHU	1	0	0	1	0	1											rt <- memory[base + offset]															
	STORE	SB	1	0	1	0	0	0											memory[base + offset] <- rt															
		SH	1	0	1	0	0	1											memory[base + offset] <- rt															
		SW	1	0	1	0	1	1											memory[base + offset] <- rt															
	INMEDIATE	ADDI	0	0	1	0	0	0											rt <- rs + inmediate															
		ANDI	0	0	1	1	0	0											rt <- rs AND inmediate															
		ORI	0	0	1	1	0	1											rt <- rs OR inmediate															
		XORI	0	0	1	1	1	0											rt <- rs XOR inmediate															
		LUI	0	0	1	1	1	1	0	0	0	0	0						rt <- inmediate || 0^16															
		SLTI	0	0	1	0	1	0											rt <- (rs < inmediate)															
	BRANCH	BEQ	0	0	0	1	0	0											if (rs = rt) then branch															
		BNE	0	0	0	1	0	1											if (rs != rt) then branch															
			OPCODE						DIRECCION																										
J	JUMP	J	0	0	0	0	1	0																										
		JAL	0	0	0	0	1	1																										

Interfaz de Usuario

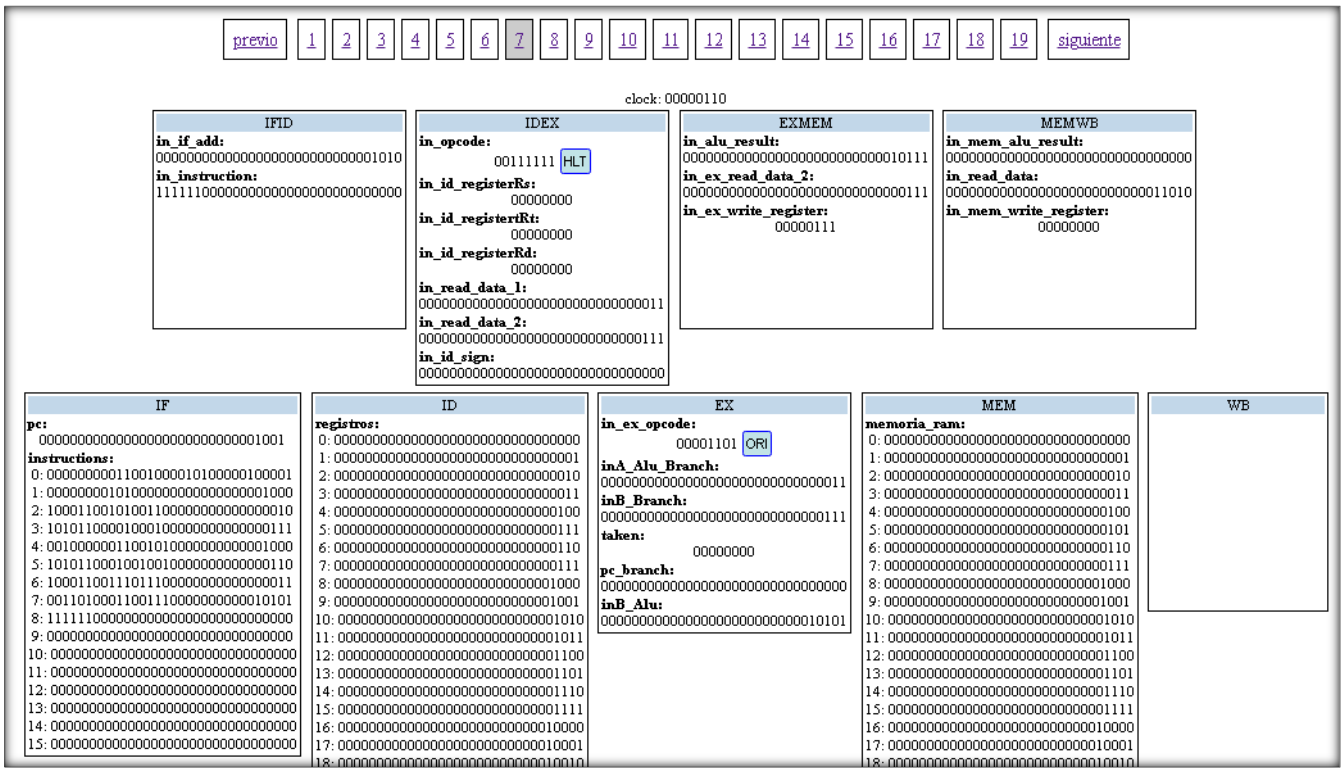
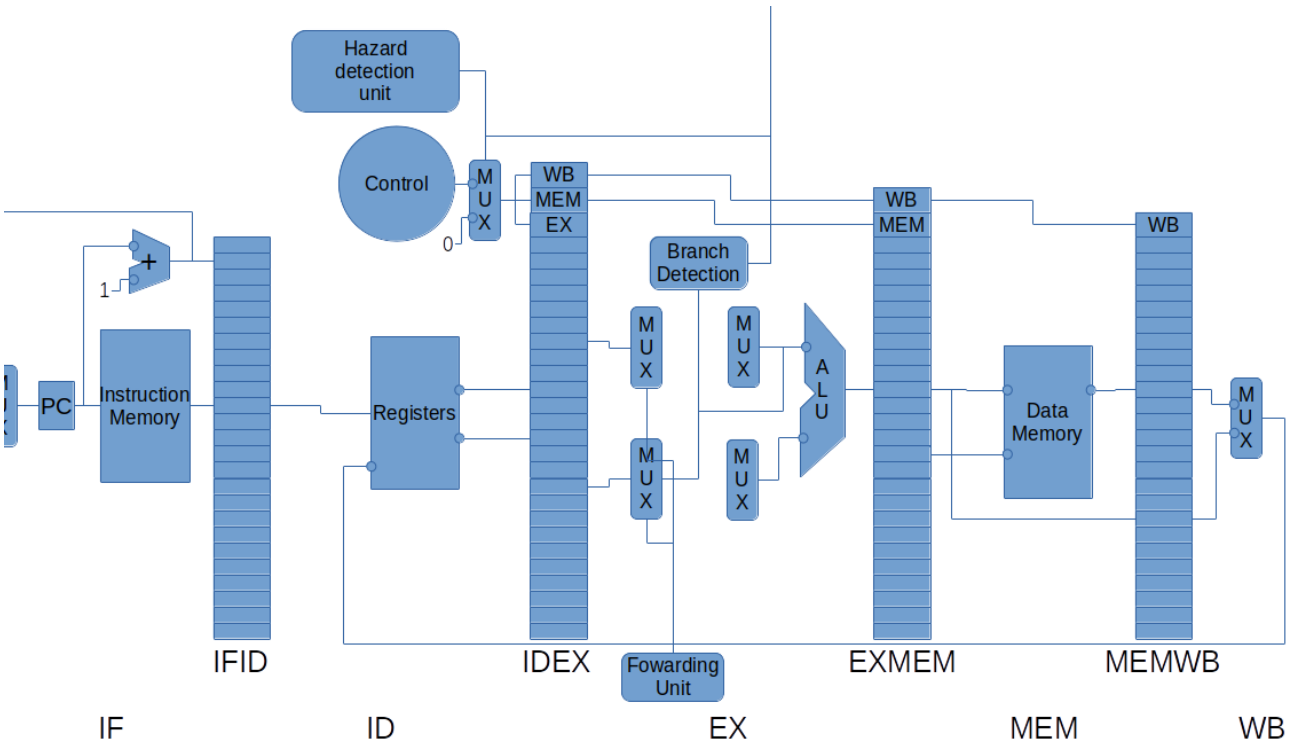


Diagrama general



Módulos

La estructura de módulos es la siguiente:

- TPF
  - Pipeline
    - IF (Instruction Fetch)
      - MUX2
      - PC
      - IM
      - ADD
    - IFID (Latch IF/ID)
    - ID (Instruction Decode)
      - REGS
      - CONTROL
      - SIGNAL EXTEND
      - LOAD FILTER
    - IDEX (Latch ID/EX)
    - EX (Execute)
      - MUX3 (para entrada 1 ALU)
      - MUX3 (para entrada 2 ALU)
      - MUX2 (para entrada 1 ALU)
      - MUX2 (para entrada 2 ALU)
      - ALU
      - ALU Control
      - MUX2 (para Write Register)
      - BRANCH (unidad de Branch y Jump)
      - STORE FILTER
    - EXMEM (Latch EX/MEM)
    - MEM (Memory Access)
      - DM
    - MEMWB (Latch MEM/WB)
    - WB (Write-Back)
      - MUX2
    - HU (Hazard Unit)
    - FU (Forwarding Unit)
    - CC (Clock Counter)
  - UART
    - BRG
    - INTERFACE
    - RX
    - TX

El modulo IF se encarga, básicamente, de seleccionar una instrucción y ponerla a la salida para su posterior decodificación. Luego hace que el PC sume uno, para que en el siguiente

clock se seleccione la siguiente instrucción. El nuevo PC puede ser el PC+1 o el PC_Branch, en caso de que la instrucción sea un salto.

El modulo ID se encarga de decodificar la instrucción para definir que registros leer, cual escribir, que operación realizar, entre otros. El modulo Control es el encargado de activar o desactivar el funcionamiento de ciertos módulos, de acuerdo a la instrucción que se va a ejecutar.

El modulo EX es el que contiene la ALU, la cual se encarga de realizar todas las operaciones de registros detalladas anteriormente. Además, en este modulo se encuentra el módulo de detección de saltos condicionales y no condicionales. Este último permite detectar si se toma o no el salto, y en caso de tomarse, se calcula el nuevo PC.

El modulo MEM es el encargado de realizar la lectura/escritura de la Memoria de Datos, de acuerdo a la instrucción que fue decodificada.

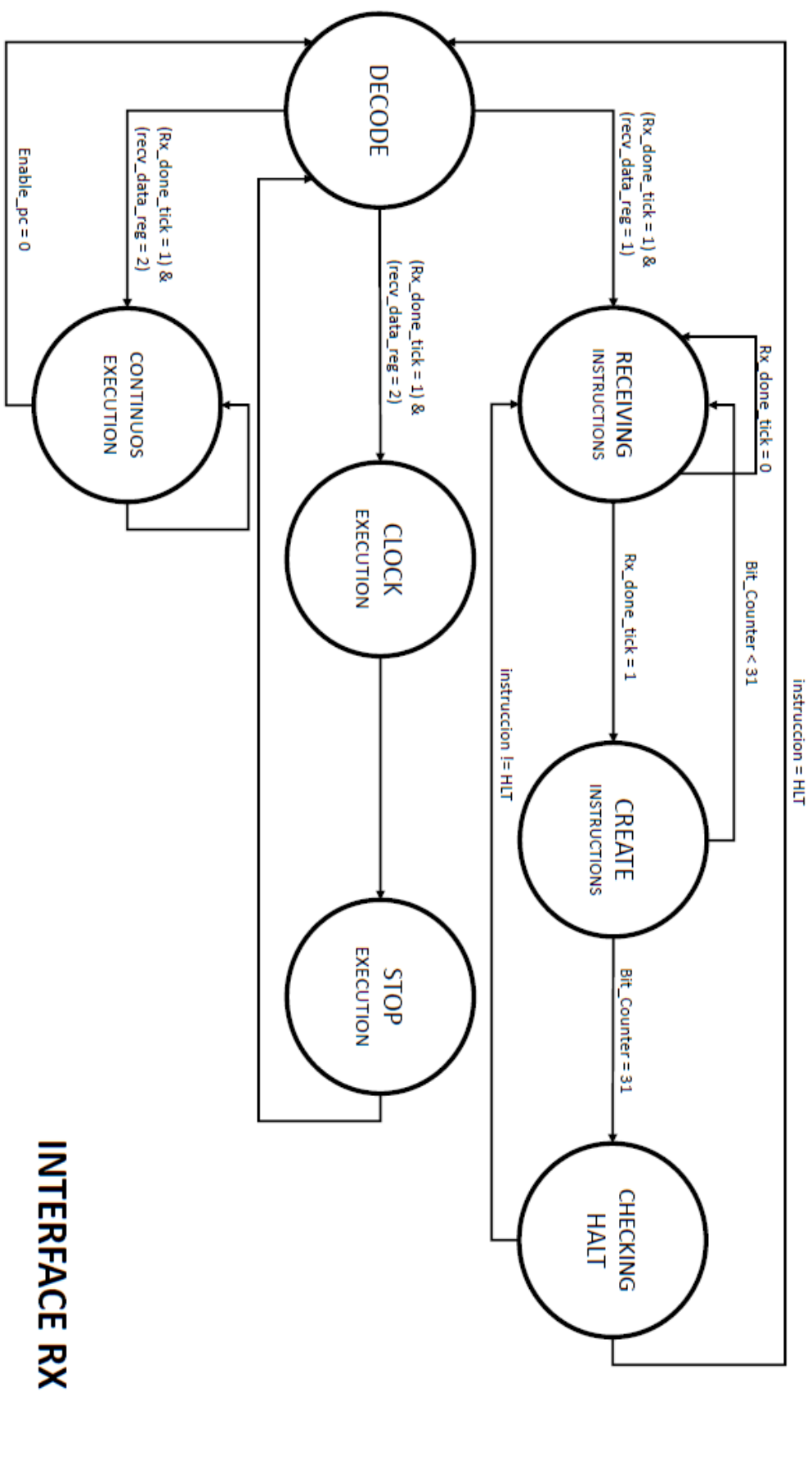
El modulo WB solo envía la información hacia el modulo REGS, para de esta manera guardar los datos en el registro correspondiente.

La FU permite, en caso de ser necesario, usar datos que todavía no están disponibles en los registros para realizar algún tipo de calculo en la ALU. Esto sirve para evitar riesgos de datos

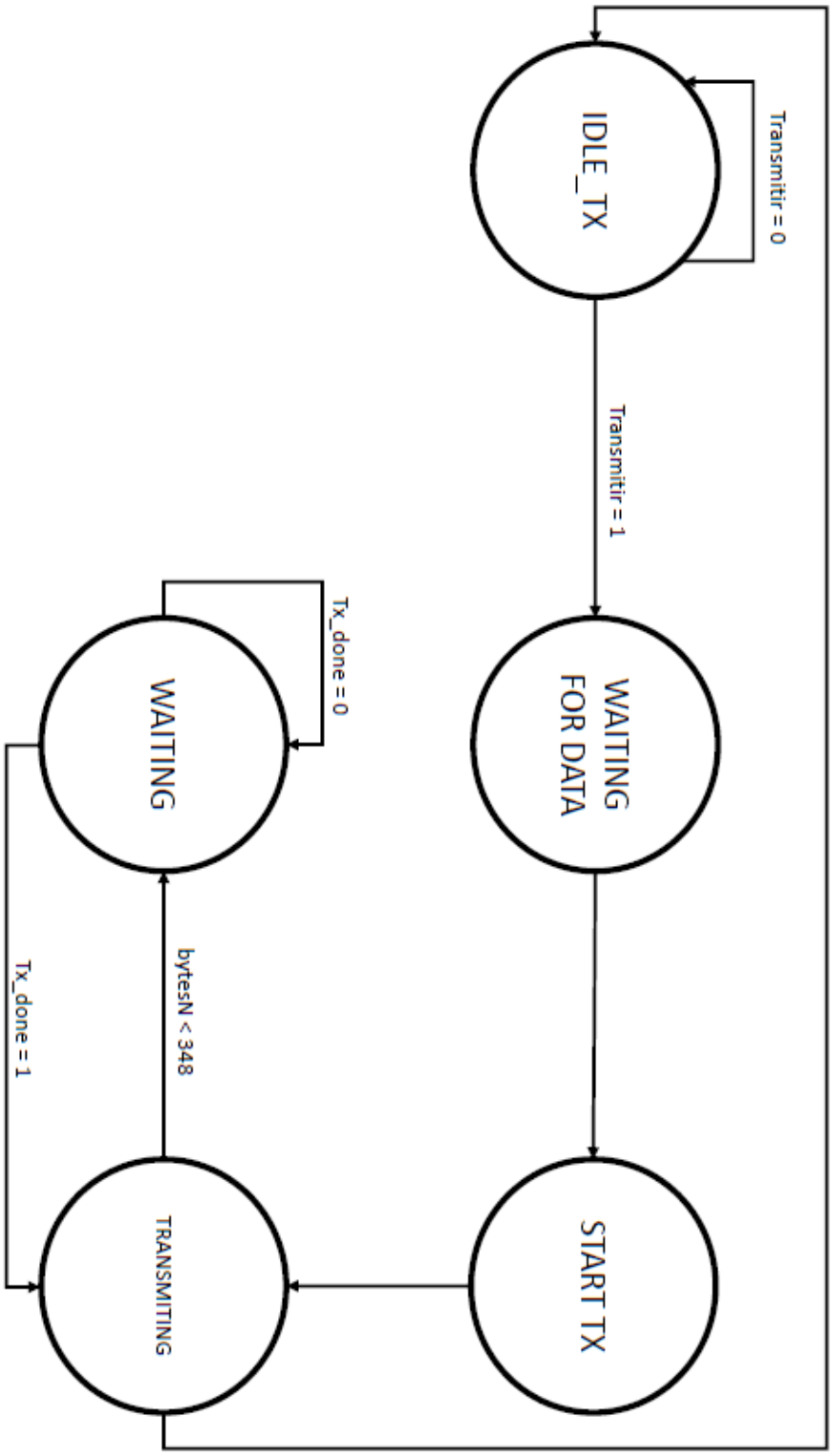
La HU permite detener el pipeline para casos particulares.

Diagramas de Estados

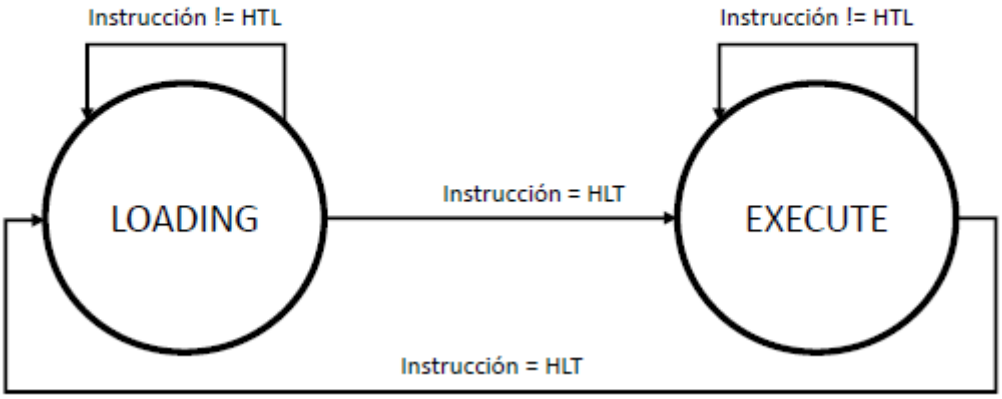
A continuación, se muestran los diagramas de estados realizados en algunos submódulos:


DECODE
(Rx_done_tick = 1) & (recv_data_reg = 1)
(Rx_done_tick = 1) & (recv_data_reg = 2)
(Rx_done_tick = 1) & (recv_data_reg = 2)
Enable_pc = 0
RECEIVING INSTRUCTIONS
Rx_done_tick = 0
Rx_done_tick = 1
Bit_Counter < 31
instruccion != HLT
instruccion = HLT
CREATE INSTRUCTIONS
Bit_Counter = 31
CHECKING HALT
CONTINUOS EXECUTION
CLOCK EXECUTION
STOP EXECUTION
INTERFACE RX

INTERFACE TX



IM


Instrucción != HTL
Instrucción != HTL
LOADING
Instrucción = HLT
EXECUTE
Instrucción = HLT

CONCLUSION

El diseño en placas FPGA permite diagramar la arquitectura de un componente de hardware por medio de software, y testear su funcionamiento antes de imprimirlo. Esto es una gran ventaja debido a que nos ahorra una gran cantidad de tiempo y dinero en impresión de placas de prueba, y además se puede debuggear en tiempo real si el comportamiento es el deseado.

En el presente trabajo se implementó un microprocesador MIPS en Verilog y se lo testeó en una placa Artix-7 Nexys 4, utilizando su IDE de desarrollo Vivado 17.4. Para esto se siguió una lógica de módulos para representar cada una de las partes del mismo, como ser Bancos de Registro, Memoria, Registros o Latches entre etapas, Multiplexores, entre otros. Se diseñó cada una de las etapas, junto a las unidades de prevención de riesgos. Además, se creó un debugger para el análisis del comportamiento del MIPS, cuyos datos son enviados por cada clock de ejecución a través de una UART. Las mayores dificultades se presentaron al intentar implementar soluciones para las unidades de riesgos, ya que eso requirió una fuerte comprensión de lo que sucede realmente en el procesador.

En resumen, hemos aprendido a describir hardware por medio de software, a utilizar placas de desarrollo FPGA’s y a crear un procesador MIPs con estas herramientas, las cuales presentan un gran potencial.