

Trabajo Práctico Final

Procesador MIPS Simplificado

Materia: Arquitectura de Computadoras

Alumnos: Coutinho, Juan Agustín **Mat.:**36548307
Malatini, Hernán **Mat.:**37196715

Profesor: Santiago Rodriguez

Año Lectivo: 2017

Consigna	4
Desarrollo	5
Ensamblador	5
Análisis de Instrucciones	6
Prueba de Implementación	10
Unidad de Debug	11
Diagrama de Estado	11
Pipeline	14
Diagrama esquemático	14
Unidad de detección de riesgos (Hazards)	15
Análisis del reporte de tiempos	15
Test Bench	17
Prueba de Implementación	21
Código assembler cargado en la placa de desarrollo	21
Funcionamiento en modo “continuo”:	22
Prueba de Implementación 2	23
Código assembler cargado en la placa de desarrollo	23
Conclusión	26

Introducción

En el presente trabajo práctico se implementará un procesador MIPS (Microprocessor without Interlocked Pipeline Stages) sobre la placa de desarrollo “Nexys 4 DDR”, la cual contiene una FPGA “Artix-7”.

El lenguaje de programación utilizado será Verilog y el entorno de desarrollo será el proporcionado por Xilinx (ISE Design Suite 14.9).

Consigna

Implementar el procesador MIPS, segmentado en las siguientes etapas:

- IF (Instruction Fetch): Búsqueda de la instrucción en la memoria del programa.
- ID (Instruction Decode): Decodificación de la instrucción y lectura de registros.
- EX (Execute): Ejecución 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.

El procesador, debe tener soporte para los siguientes tipos de 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 esté preparado. Mantenimiento del orden estricto de lecturas y escrituras.
- ☐ De control: Intentar tomar una decisión sobre una condición todavía no evaluada.

Por lo que deberá contar con una unidad de detección de riesgos y una unidad de cortocircuitos.

Los requerimientos son los siguientes:

- La memoria de datos debe estar separada de la memoria de instrucciones.
- El programa a ejecutar debe ser cargado en la memoria de programa mediante un archivo ensamblado.
 - ◆ Debe implementarse un programa ensamblador.
- Se debe incluir una unidad de debug que envíe información a la pc mediante la uart.

En cuanto a la unidad de debug:

- Se deben enviar a la PC a través de la uart:
 - Contenido de los 32 registros.
 - Contenidos de los latches intermedios.
 - PC.
 - Contenido de la memoria de datos utilizada.
- Debe permitir dos modos de operación:
 - Continuo: Se envía un comando a la fpga por la uart y esta inicia la ejecución del programa hasta llegar al final del mismo. Llegado a ese punto, se muestran todos los valores indicados en pantalla.
 - Paso a paso: Enviando un comando por la uart se ejecuta un ciclo de clock. Se debe mostrar a cada paso los valores indicados.

Desarrollo

Ensamblador

El ensamblador es el programa encargado de traducir código en lenguaje *assembler* a código binario (lenguaje máquina), el cual está compuesto únicamente por '1's y '0's, y en el caso del programa ensamblador utilizado, el código binario resultante se representa en hexadecimal.

El programa ensamblador fue desarrollado en lenguaje Python, el cual recibe un archivo de entrada ".asm" con las instrucciones y lo transforma en un archivo ".coe" listo para ser cargado en la memoria del programa. Las instrucciones soportadas por el mismo son:

- 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

Análisis de Instrucciones

SLL -- Shift left logical

Description:	Shifts a register value left by the shift amount listed in the instruction and places the result in a third register. Zeroes are shifted in.
Operation:	\$d = \$t << h; advance_pc (4);
Syntax:	sll \$d, \$t, h
Encoding:	0000 00ss ssst tttt dddd dhhh hh00 0000

SRL -- Shift right logical

Description:	Shifts a register value right by the shift amount (shamt) and places the value in the destination register. Zeroes are shifted in.
Operation:	\$d = \$t >> h; advance_pc (4);
Syntax:	srl \$d, \$t, h
Encoding:	0000 00-- ---t tttt dddd dhhh hh00 0010

SRA -- Shift right arithmetic

Description:	Shifts a register value right by the shift amount (shamt) and places the value in the destination register. The sign bit is shifted in.
Operation:	\$d = \$t >> h; advance_pc (4);
Syntax:	sra \$d, \$t, h
Encoding:	0000 00-- ---t tttt dddd dhhh hh00 0011

SLLV -- Shift left logical variable

Description:	Shifts a register value left by the value in a second register and places the result in a third register. Zeroes are shifted in.
Operation:	\$d = \$t << \$s; advance_pc (4);
Syntax:	sllv \$d, \$t, \$s
Encoding:	0000 00ss ssst tttt dddd d--- --00 0100

SRLV -- Shift right logical variable

Description:	Shifts a register value right by the amount specified in \$s and places the value in the destination register. Zeroes are shifted in.
Operation:	\$d = \$t >> \$s; advance_pc (4);
Syntax:	srlv \$d, \$t, \$s
Encoding:	0000 00ss ssst tttt dddd d000 0000 0110

ADDU -- Add unsigned (no overflow)

Description:	Adds two registers and stores the result in a register
Operation:	\$d = \$s + \$t; advance_pc (4);
Syntax:	addu \$d, \$s, \$t

Encoding:	0000 00ss ssst tttt dddd d000 0010 0001
-----------	---

SUBU -- Subtract unsigned

Description:	Subtracts two registers and stores the result in a register
Operation:	\$d = \$s - \$t; advance_pc (4);
Syntax:	subu \$d, \$s, \$t
Encoding:	0000 00ss ssst tttt dddd d000 0010 0011

AND -- Bitwise and

Description:	Bitwise ands two registers and stores the result in a register
Operation:	\$d = \$s & \$t; advance_pc (4);
Syntax:	and \$d, \$s, \$t
Encoding:	0000 00ss ssst tttt dddd d000 0010 0100

OR -- Bitwise or

Description:	Bitwise logical ors two registers and stores the result in a register
Operation:	\$d = \$s \$t; advance_pc (4);
Syntax:	or \$d, \$s, \$t
Encoding:	0000 00ss ssst tttt dddd d000 0010 0101

XOR -- Bitwise exclusive or

Description:	Exclusive ors two registers and stores the result in a register
Operation:	\$d = \$s ^ \$t; advance_pc (4);
Syntax:	xor \$d, \$s, \$t
Encoding:	0000 00ss ssst tttt dddd d--- --10 0110

SLT -- Set on less than (signed)

Description:	If \$s is less than \$t, \$d is set to one. It gets zero otherwise.
Operation:	if \$s < \$t \$d = 1; advance_pc (4); else \$d = 0; advance_pc (4);
Syntax:	slt \$d, \$s, \$t
Encoding:	0000 00ss ssst tttt dddd d000 0010 1010

LB -- Load byte

Description:	A byte is loaded into a register from the specified address.
Operation:	\$t = MEM[\$s + offset]; advance_pc (4);
Syntax:	lb \$t, offset(\$s)
Encoding:	1000 00ss ssst tttt iiiiiiii iiiiiiii

LW -- Load word

Description:	A word is loaded into a register from the specified address.
Operation:	\$t = MEM[\$s + offset]; advance_pc (4);
Syntax:	lw \$t, offset(\$s)
Encoding:	1000 11ss ssst tttt iiii iiii iiii

SB -- Store byte

Description:	The least significant byte of \$t is stored at the specified address.
Operation:	MEM[\$s + offset] = (0xff & \$t); advance_pc (4);
Syntax:	sb \$t, offset(\$s)
Encoding:	1010 00ss ssst tttt iiii iiii iiii

SW -- Store word

Description:	The contents of \$t is stored at the specified address.
Operation:	MEM[\$s + offset] = \$t; advance_pc (4);
Syntax:	sw \$t, offset(\$s)
Encoding:	1010 11ss ssst tttt iiii iiii iiii

ADDI -- Add immediate (with overflow)

Description:	Adds a register and a sign-extended immediate value and stores the result in a register
Operation:	\$t = \$s + imm; advance_pc (4);
Syntax:	addi \$t, \$s, imm
Encoding:	0010 00ss ssst tttt iiii iiii iiii

ANDI -- Bitwise and immediate

Description:	Bitwise ands a register and an immediate value and stores the result in a register
Operation:	\$t = \$s & imm; advance_pc (4);
Syntax:	andi \$t, \$s, imm
Encoding:	0011 00ss ssst tttt iiii iiii iiii

ORI -- Bitwise or immediate

Description:	Bitwise ors a register and an immediate value and stores the result in a register
Operation:	\$t = \$s imm; advance_pc (4);
Syntax:	ori \$t, \$s, imm
Encoding:	0011 01ss ssst tttt iiii iiii iiii

XORI -- Bitwise exclusive or immediate

Description:	Bitwise exclusive ors a register and an immediate value and stores the result in a register
Operation:	$\$t = \$s \wedge \text{imm}$; advance_pc (4);
Syntax:	xori \$t, \$s, imm
Encoding:	0011 10ss ssst tttt iiiiiiii iiiiiiii

LUI -- Load upper immediate

Description:	The immediate value is shifted left 16 bits and stored in the register. The lower 16 bits are zeroes.
Operation:	$\$t = (\text{imm} \ll 16)$; advance_pc (4);
Syntax:	lui \$t, imm
Encoding:	0011 11-- ---t tttt iiiiiiii iiiiiiii

SLTI -- Set on less than immediate (signed)

Description:	If \$s is less than immediate, \$t is set to one. It gets zero otherwise.
Operation:	if $\$s < \text{imm}$ $\$t = 1$; advance_pc (4); else $\$t = 0$; advance_pc (4);
Syntax:	slti \$t, \$s, imm
Encoding:	0010 10ss ssst tttt iiiiiiii iiiiiiii

BEQ -- Branch on equal

Description:	Branches if the two registers are equal
Operation:	if $\$s == \t advance_pc (offset $\ll 2$); else advance_pc (4);
Syntax:	beq \$s, \$t, offset
Encoding:	0001 00ss ssst tttt iiiiiiii iiiiiiii

BNE -- Branch on not equal

Description:	Branches if the two registers are not equal
Operation:	if $\$s \neq \t advance_pc (offset $\ll 2$); else advance_pc (4);
Syntax:	bne \$s, \$t, offset
Encoding:	0001 01ss ssst tttt iiiiiiii iiiiiiii

J -- Jump

Description:	Jumps to the calculated address
Operation:	$PC = nPC$; $nPC = (PC \& 0xf0000000) (\text{target} \ll 2)$;
Syntax:	j target
Encoding:	0000 10ii iiiiiiii iiiiiiii iiiiiiii

JAL -- *Jump and link*

Description:	Jumps to the calculated address and stores the return address in \$31
Operation:	$\$31 = PC + 8$ (or $nPC + 4$); $PC = nPC$; $nPC = (PC \& 0xf0000000) (target \ll 2)$;
Syntax:	jal target
Encoding:	0000 11ii iiiiiiii iiiiiiii iiiiiiii

JR -- *Jump register*

Description:	Jump to the address contained in register \$s
Operation:	$PC = nPC$; $nPC = \$s$;
Syntax:	jr \$s
Encoding:	0000 00ss sss0 0000 0000 0000 0000 1000

Prueba de Implementación

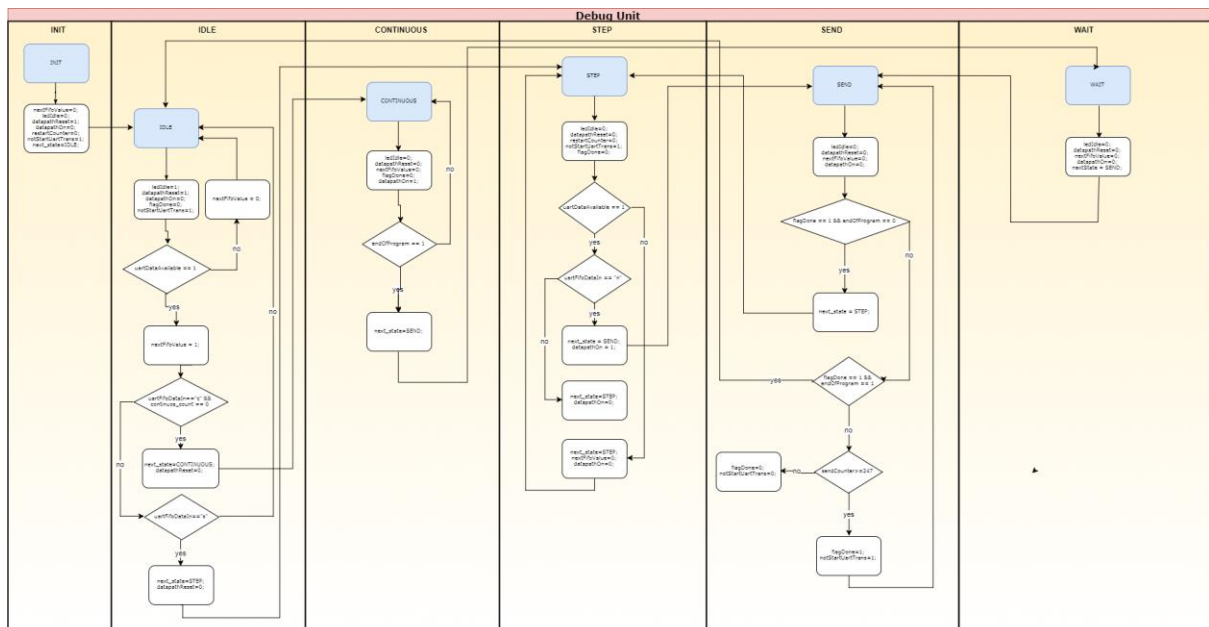
A continuación, se presenta la traducción de cada una de las instrucciones que deben permitirse, con valores numéricos de ejemplo:

Código Assembler	Código Hexadecimal	Código Binario de 32 bits
srl 6, 4, 2	00043082h	0000 0000 0000 0100 0011 0000 1000 0010
sra 6, 4, 2	00043083h	0000 0000 0000 0100 0011 0000 1000 0011
srlv 4, 4, 2	00442006h	0000 0000 0100 0100 0010 0000 0000 0110
srav 4, 6, 5	00a62007h	0000 0000 1010 0110 0010 0000 0000 0111
sllv 6, 2, 4	00823004h	0000 0000 1000 0010 0011 0000 0000 0100
add 10, 4, 8	00885020h	0000 0000 1000 1000 0101 0000 0010 0000
sub 12, 6, 4	00c46022h	0000 0000 1100 0100 0110 0000 0010 0010
and 2, 2, 2	00421024h	0000 0000 0100 0010 0001 0000 0010 0100
or 4, 8, 4	01042025h	0000 0001 0000 0100 0010 0000 0010 0101
xor 5, 6, 7	00c72826h	0000 0000 1100 0111 0010 1000 0010 0110
nor 2, 4, 8	00881027h	0000 0000 1000 1000 0001 0000 0010 0111
slt 6, 10, 12	014c302ah	0000 0001 0100 1100 0011 0000 0010 1010
sll 6, 4, 2	00043080h	0000 0000 0000 0100 0011 0000 1000 0000
lb 3, 0(4)	80830000h	1000 0000 1000 0011 0000 0000 0000 0000
lh 4, 4(4)	84840004h	1000 0100 1000 0100 0000 0000 0000 0100
lw 5, 8(4)	8c850008h	1000 1100 1000 0101 0000 0000 0000 1000
lbu 6, 0(5)	90a60000h	1001 0000 1010 0110 0000 0000 0000 0000
lhu 7, 0(6)	94c70000h	1001 0100 1100 0111 0000 0000 0000 0000

lwu 8, 0(7)	9ce80000h	1001 1100 1110 1000 0000 0000 0000 0000
sb 9, 0(9)	a1290000h	1010 0001 0010 1001 0000 0000 0000 0000
sh 10, 0(10)	a54a0000h	1010 0101 0100 1010 0000 0000 0000 0000
sw 11, 0(11)	ad6b0000h	1010 1101 0110 1011 0000 0000 0000 0000
addi 4, 4, 2	20840002h	0010 0000 1000 0100 0000 0000 0000 0010
andi 2, 4, 6	30820006h	0011 0000 1000 0010 0000 0000 0000 0110
ori 4, 6, 1256	34c404e8h	0011 0100 1100 0100 0000 0100 1110 1000
xori 4, 7, 6	38e40006h	0011 1000 1110 0100 0000 0000 0000 0110
lui 4, 8	3c040008h	0011 1100 0000 0100 0000 0000 0000 1000
stli 4, 4, 2	28840002h	0010 1000 1000 0100 0000 0000 0000 0010
beq 4, 5, 8	10850008h	0001 0000 1000 0101 0000 0000 0000 1000
bne 4, 6, 10	1486000ah	0001 0100 1000 0110 0000 0000 0000 1010
j 10	0800000ah	0000 1000 0000 0000 0000 0000 0000 1010
jal 24	0c000018h	0000 1100 0000 0000 0000 0000 0001 1000
jr 6	00c00008h	0000 0000 1100 0000 0000 0000 0000 1000
jalr 2, 12	01801009h	0000 0001 1000 0000 0001 0000 0000 1001
END	ffffffffh	1111 1111 1111 1111 1111 1111 1111 1111

Unidad de Debug

Diagrama de Estado



Como se puede ver en el diagrama de estados, la unidad de debug se puede comportar de dos maneras. Si por la uart le llega un carácter “c”, significa que se activa el modo continuo. En este modo se esperará a que el flag de end of program se active, para recién comenzar a transmitir todos los datos de los registros. Por otro lado, si por la uart se le envía un carácter “s”, esta se pondrá a la espera de otra letra, la cual es el carácter “n” de next clock, por lo que entre clock y clock, se irá enviando por la uart el estado de todos los registros. Para controlar la ejecución del pipeline, la unidad de debug habilita o deshabilita la acción del clock sobre el datapath a través de la variable “**datapathOn**”. Esto permite ejecutar instrucciones paso a paso y frenar la ejecución completamente a demanda del usuario. Por otro lado, este módulo cuenta con conexiones a los registros del procesador, registros de propósito general y a la memoria RAM para extraer datos de ellos.

La máquina de estados de la imagen anterior se compone de los siguientes:

- ☐ INIT: Inicialización de valores de la unidad de debug, para luego pasar al estado IDLE.
- ☐ IDLE: En este estado es donde se espera por el modo de ejecución a utilizar. Como se describió recientemente:
 - ☐ Si recibe “c” pasa a modo continuo
 - ☐ Si recibe “s” pasa a modo paso a paso (step)
- ☐ CONTINUO: habilita al pipeline y no lo interrumpe hasta que se llegue a la última instrucción END, momento en el cual se activa la bandera “**endOfProgram**” y se pasa al estado SEND.
- ☐ STEP: En este estado se queda a la espera del carácter “n”, ya que solo se llega al mismo cuando el modo paso a paso es activado. En este estado se mantiene el pipeline inhabilitado. Cuando se detecta que llegó el carácter, habilita el datapath y pasa al estado WAIT (el cual desactiva el datapath ni bien se ingresa a este estado), lo que produce que se ejecute sólo un ciclo de instrucciones.
- ☐ WAIT: Se desactiva el datapath, y se pasa al estado SEND. Este estado es para esperar un ciclo de clock de modo que los datos estén listos.
- ☐ SEND: En la unidad de debug, se tiene inicializado un array con los datos de todos los registros a enviar, por lo que, cuando se llega a este estado se envía cada uno de los valores a la UART. Cuando un dato está listo se procede a enviar el próximo hasta completar la lista de todos los datos (recordar que por UART se envía de a 8 bits).

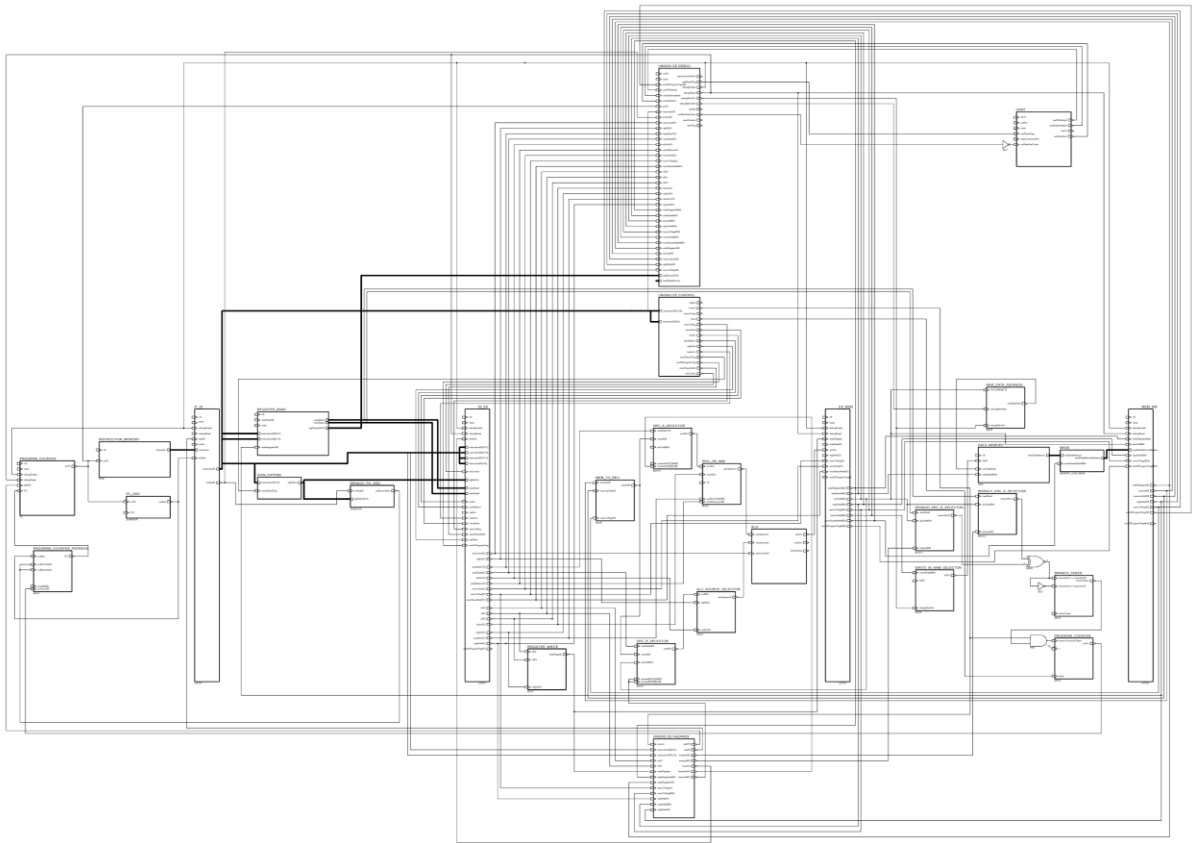
Un caso particular es el envío de datos de la memoria RAM. Para lo cual existe un multiplexor “ramDataAddressMux” para darle el control a la unidad de Debug cuando se encuentre en estado SEND.

Una vez finalizado:

- ☐ Si se detectó el fin del programa se pasa a IDLE
- ☐ De otro modo se pasa a STEP

Pipeline

Diagrama esquemático



Se presenta en un anexo, en formato PDF, el diagrama en máxima resolución (tamaño A0). La misma representa la interconexión de todas las unidades que conforman el procesador MIPS simplificado que se ha desarrollado.

Unidad de detección de riesgos (Hazards)

La unidad de detección de riesgos es la encargada de prevenir que no ocurran los diferentes riesgos identificados en un pipeline, los cuales pueden ser de datos o de control. Para ello tenemos tres tipos de control:

- **Stall:** Detener el avance de determinados datos que son inválidos momentáneamente, debido a que la operación actual necesita datos que todavía se están calculando dentro del pipeline. Se puede detener tanto el PC como cualquiera de los “Latches”.
- **Flush:** Utilizado para introducir una burbuja, es decir, ingresando ceros a la “etapa de ejecución”. Ésto se logra limpiando la salida del latch “ID_EX” cuando se activa la bandera correspondiente de “flush”, evitando así que se propague información errónea.
- **Forward (corto-circuito):** Se encarga de proporcionar los datos necesarios en la “etapa de ejecución”, proveniente de la etapa “Memory Access”, cuando aún una instrucción no ha finalizado las 5 etapas del pipeline, pero estos datos ya están listos y no van a ser modificados.

Análisis del reporte de tiempos

Como se puede observar en el reporte obtenido del IDE Xilinx ISE (Design Summary -> Synthesis Report), la frecuencia máxima que es capaz de soportar nuestro pipeline es de 77.857 MHz. Esto es debido a los retardos que se tienen por los tiempos de propagación a lo largo de todo el pipeline.

Timing Summary:

Speed Grade: -1

Minimum period: 12.844ns (Maximum Frequency: 77.857MHz) ← Máxima frecuencia del clock

Minimum input arrival time before clock: 4.508ns
Maximum output required time after clock: 3.615ns
Maximum combinational path delay: 0.001ns

Timing Details:

All values displayed in nanoseconds (ns)

Timing constraint: Default period analysis for Clock 'clock/clkout0'
Clock period: 12.844ns (frequency: 77.857MHz)
Total number of paths / destination ports: 455574 / 2826

Delay: 6.422ns (Levels of Logic = 17) ← Flujo de compuertas que más tiempo consume

Source: datapath/reg_bank/reg_file_0_832 (FF)
Destination: datapath/if_id/instruction_out_31 (FF)
Source Clock: clock/clkout0 rising
Destination Clock: clock/clkout0 falling

Data Path: datapath/reg_bank/reg_file_0_832 to datapath/if_id/instruction_out_31

Cell:in->out	Gate	Net	fanout	Delay	Logical Name (Net Name)
FDCE:C->Q	4	0.478	0.796	datapath/reg_bank/reg_file_0_832 (datapath/reg_bank/reg_file_0_832)	
LUT6:I2->O	1	0.124	0.776	datapath/reg_bank/Mmux_read_addr1[4]_reg_file[31][31]_wide_mux_38_OUT_81	
(datapath/reg_bank/Mmux_read_addr1[4]_reg_file[31][31]_wide_mux_38_OUT_81)					
LUT6:I2->O	2	0.124	0.542	datapath/reg_bank/Mmux_read_addr1[4]_reg_file[31][31]_wide_mux_38_OUT_3	
(datapath/reg_bank/Mmux_read_addr1[4]_reg_file[31][31]_wide_mux_38_OUT_3)					
LUT6:I4->O	1	0.124	0.939	datapath/branchSrcASelectorMux/Mmux_Salida110 (datapath/branchSrcA<0>)	
LUT6:I0->O	1	0.124	0.000	datapath/Mcompar_branchSrcA[31]_branchSrcB[31]_equal_2_o_lut<0> (datapath/Mcompar_branchSrcA[31]_branchSrcB[31]_equal_2_o_lut<0>)	
MUXCY:S->O	1	0.472	0.000	datapath/Mcompar_branchSrcA[31]_branchSrcB[31]_equal_2_o_cy<0> (datapath/Mcompar_branchSrcA[31]_branchSrcB[31]_equal_2_o_cy<0>)	
MUXCY:CI->O	1	0.029	0.000	datapath/Mcompar_branchSrcA[31]_branchSrcB[31]_equal_2_o_cy<1> (datapath/Mcompar_branchSrcA[31]_branchSrcB[31]_equal_2_o_cy<1>)	
MUXCY:CI->O	1	0.029	0.000	datapath/Mcompar_branchSrcA[31]_branchSrcB[31]_equal_2_o_cy<2> (datapath/Mcompar_branchSrcA[31]_branchSrcB[31]_equal_2_o_cy<2>)	
MUXCY:CI->O	1	0.029	0.000	datapath/Mcompar_branchSrcA[31]_branchSrcB[31]_equal_2_o_cy<3> (datapath/Mcompar_branchSrcA[31]_branchSrcB[31]_equal_2_o_cy<3>)	
MUXCY:CI->O	1	0.029	0.000	datapath/Mcompar_branchSrcA[31]_branchSrcB[31]_equal_2_o_cy<4> (datapath/Mcompar_branchSrcA[31]_branchSrcB[31]_equal_2_o_cy<4>)	
MUXCY:CI->O	1	0.029	0.000	datapath/Mcompar_branchSrcA[31]_branchSrcB[31]_equal_2_o_cy<5> (datapath/Mcompar_branchSrcA[31]_branchSrcB[31]_equal_2_o_cy<5>)	
MUXCY:CI->O	1	0.029	0.000	datapath/Mcompar_branchSrcA[31]_branchSrcB[31]_equal_2_o_cy<6> (datapath/Mcompar_branchSrcA[31]_branchSrcB[31]_equal_2_o_cy<6>)	
MUXCY:CI->O	1	0.029	0.000	datapath/Mcompar_branchSrcA[31]_branchSrcB[31]_equal_2_o_cy<7> (datapath/Mcompar_branchSrcA[31]_branchSrcB[31]_equal_2_o_cy<7>)	
MUXCY:CI->O	1	0.029	0.000	datapath/Mcompar_branchSrcA[31]_branchSrcB[31]_equal_2_o_cy<8> (datapath/Mcompar_branchSrcA[31]_branchSrcB[31]_equal_2_o_cy<8>)	
MUXCY:CI->O	1	0.029	0.000	datapath/Mcompar_branchSrcA[31]_branchSrcB[31]_equal_2_o_cy<9> (datapath/Mcompar_branchSrcA[31]_branchSrcB[31]_equal_2_o_cy<9>)	

MUXCY:Cl->O	9	0.334	0.474	datapath/Mcompar_branchSrcA[31]_branchSrcB[31]_equal_2_o_cy<10>	(datapath/branchSrcA[31]_branchSrcB[31]_equal_2_o)
LUT6:I5->O	32	0.124	0.574	datapath/Reset_OR_DriverANDClockEnable321	(datapath/Reset_OR_DriverANDClockEnable)
LUT5:I4->O	1	0.124	0.000	datapath/if_id/instruction_out_31_rstpot	(datapath/if_id/instruction_out_31_rstpot)
FD_1:D		0.030		datapath/if_id/instruction_out_31	
<hr/>					
Total		6.422ns (2.322ns logic, 4.101ns route)			
		(36.1% logic, 63.9% route)			

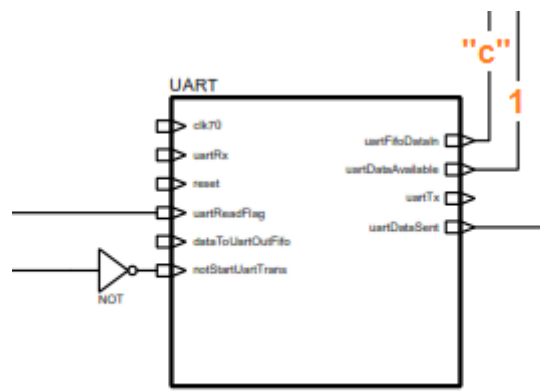
A su vez, se puede observar el flujo que más tiempo consume. Este es el que va desde el “register_bank”, analizando los parámetros de la instrucción, hacia el circuito combinacional que analiza si el salto puede ser tomado y el cálculo de la dirección del mismo, llegando el resultado al latch IF_ID. Este tiempo es de 6.422ns.

Test Bench

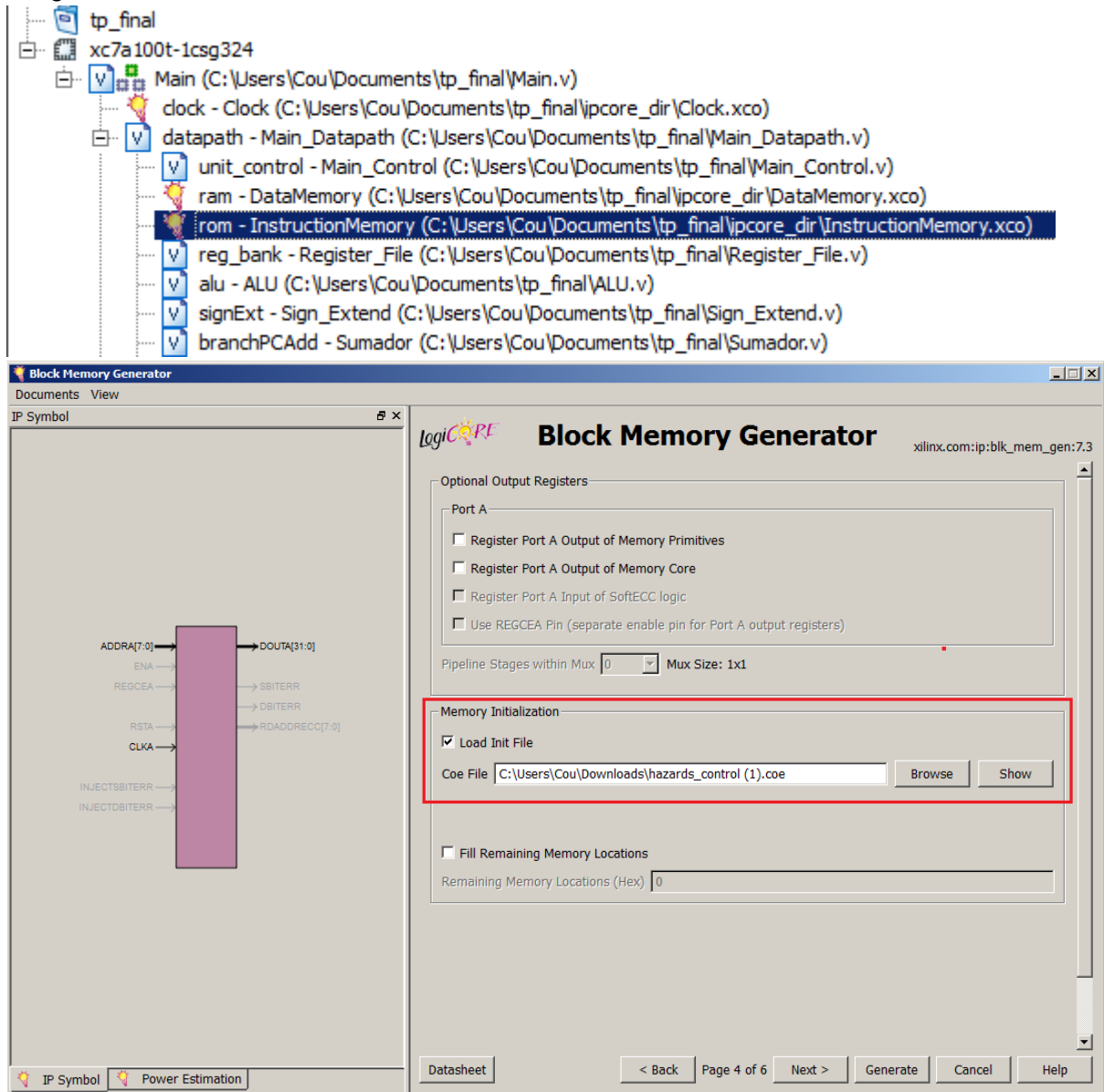
A través del programa ensamblador, se genera un archivo en formato “.coe” el cual contiene las siguientes instrucciones:

addi 1, 1, 3000	20210bb8h
addi 2, 2, 1500	204205dch
addi 3, 3, 1500	206305dch
beq 2, 3, 1	10430001h
addi 4, 4, 1	20840001h
sub 5, 1, 3	00232822h
beq 1, 2, 2	10220002h
addi 6, 6, 1	20c60001h
addi 7, 7, 1	20e70001h
END	ffffffff;

En primera instancia, para poder simular localmente, se deben prefijar dos valores a la salida de la unidad UART. Uno de ellos es la bandera “uartDataAvailable” y se fija el valor “c” en la salida de la unidad UART, para simular el ingreso por teclado de la letra “c” y se desencadene así el modo de funcionamiento “continuo” que permite debuguear paso a paso el programa precargado en la memoria ROM de instrucciones que se auto generó mediante la herramienta IP Core (Intellectual Property Cores) provista por el framework Xilinx.



La carga de las instrucciones se realiza abriendo el módulo de memoria ROM IP Core e indicando la ubicación del archivo de programación como se muestra en la siguiente imagen:



Una vez seleccionado el archivo se presiona el botón “Generate” y se corre el test “Main_Datapath_Test”.





































































A continuación se muestra el paso a paso (por clock) del banco de registros del procesador.

clock 0

clock with reset

clock 6

clock 14

clock 15			clock 16		
	clk	1		clk	1
	reset	0		reset	0
	wr_enable3	1		wr_enable3	0
	read_addr1[4:0]	31		read_addr1[4:0]	31
	read_addr2[4:0]	31		read_addr2[4:0]	31
	write_addr3[4:0]	7		write_addr3[4:0]	31
	write_data3[31:0]	1		write_data3[31:0]	65535
	read_data1[31:0]	0		read_data1[31:0]	0
	read_data2[31:0]	0		read_data2[31:0]	0
	read_data_to_debug_0[31:0]	0		read_data_to_debug_0[31:0]	0
	read_data_to_debug_1[31:0]	3000		read_data_to_debug_1[31:0]	3000
	read_data_to_debug_2[31:0]	1500		read_data_to_debug_2[31:0]	1500
	read_data_to_debug_3[31:0]	1500		read_data_to_debug_3[31:0]	1500
	read_data_to_debug_4[31:0]	0		read_data_to_debug_4[31:0]	0
	read_data_to_debug_5[31:0]	1500		read_data_to_debug_5[31:0]	1500
	read_data_to_debug_6[31:0]	1		read_data_to_debug_6[31:0]	1
	read_data_to_debug_7[31:0]	0		read_data_to_debug_7[31:0]	1
	read_data_to_debug_8[31:0]	0		read_data_to_debug_8[31:0]	0
	read_data_to_debug_9[31:0]	0		read_data_to_debug_9[31:0]	0
	read_data_to_debug_10[31:0]	0		read_data_to_debug_10[31:0]	0
	read_data_to_debug_11[31:0]	0		read_data_to_debug_11[31:0]	0
	read_data_to_debug_12[31:0]	0		read_data_to_debug_12[31:0]	0
	read_data_to_debug_13[31:0]	0		read_data_to_debug_13[31:0]	0
	read_data_to_debug_14[31:0]	0		read_data_to_debug_14[31:0]	0
	read_data_to_debug_15[31:0]	0		read_data_to_debug_15[31:0]	0
	read_data_to_debug_16[31:0]	0		read_data_to_debug_16[31:0]	0
	read_data_to_debug_17[31:0]	0		read_data_to_debug_17[31:0]	0
	read_data_to_debug_18[31:0]	0		read_data_to_debug_18[31:0]	0
	read_data_to_debug_19[31:0]	0		read_data_to_debug_19[31:0]	0
	read_data_to_debug_20[31:0]	0		read_data_to_debug_20[31:0]	0
	read_data_to_debug_21[31:0]	0		read_data_to_debug_21[31:0]	0
	read_data_to_debug_22[31:0]	0		read_data_to_debug_22[31:0]	0
	read_data_to_debug_23[31:0]	0		read_data_to_debug_23[31:0]	0
	read_data_to_debug_24[31:0]	0		read_data_to_debug_24[31:0]	0

Prueba de Implementación

Código assembler cargado en la placa de desarrollo

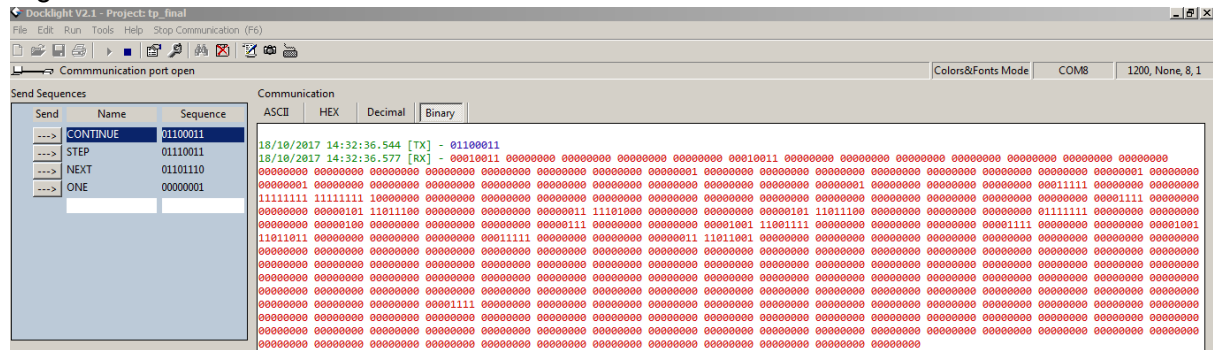
Archivo de Texto de Entrada (fuente.asm)	Archivo de Salida para Cargar en la Placa (salida.coe)
addi 1, 1, 3000	20210bb8,
addi 2, 2, 1500	204205dc,
addi 3, 3, 1000	206303e8,
addi 7, 7, 7	20e70007,
addi 9, 9, 15	2129000f,
addi 11, 11, 31	216b001f,
sw 9, 0(0)	ac090000,
add 1, 2, 3	00430820
sub 4, 1, 3	00232022
and 6, 1, 7	00273024
or 8, 1, 9	00294025
xor 10, 1, 11	002b5026,
addi 5, 5, 127	20a5007f,
lw 1, 0(0)	8c010000,
sub 12, 3, 1	00616022
END	fffffff;

Ejecución del ensamblador:

```
$ python mipsAssembler.py fuente.asm -o salida.coe
```


Funcionamiento en modo “continuo”:

Se envió el caracter “c” por UART, y se recibio el codigo binario de todos los registros:



Copiando el resultado en un archivo de texto plano con el nombre “binario.txt” y ejecutando “Debugger.py” (en la misma carpeta), se obtiene lo siguiente:

```
PC COUNTER
FE_pc / pcFE: 3

LATCH IF-ID
IF_ID_instr / instrID: 206303e8
IF_ID_pcNext / pcNextID: 3

LATCH ID-EX
ID_EX_aluControl / aluControlEX: 3
ID_EX_sigImm / SigExtEX: 1500
ID_EX_readData1 / readData1EX: 0
ID_EX_readData2 / readData2EX: 0
ID_EX_aluSrc / aluSrcEX: 1
ID_EX_aluShiftImm / aluShiftImmEX: 0
ID_EX_memWrite / memWriteEX: 0
ID_EX_memToReg / memToRegEX: 0
ID_EX_memReadWidth / memReadWidthEX: 0
ID_EX_rs / rsEX: 2
ID_EX_rt / rtEX: 2
ID_EX_rd / rdEX: 0
ID_EX_sa / shamtEX: 23
ID_EX_regDst / regDstEX: 0
ID_EX_loadImm / loadImmEX: 0
ID_EX_regWrite / regWriteEX: 1

LATCH EX-MEM
EX_MEM_writeReg / writeRegisterMEM: 1
EX_MEM_writeData / writeDataMEM: 0
EX_MEM_aluOut / aluOutMEM: 3000
EX_MEM_regWrite / regWriteMEM: 1
EX_MEM_memWrite / memWriteMEM: 0
EX_MEM_memReadWidth / memReadWidthMEM: 0

LATCH MEM-WB
MEM_WB_writeReg / writeRegisterWB: 0
MEM_WB_aluOut / aluOutWB: 0
MEM_WB_readData / memoryOutWB: 15
MEM_WB_regWrite / regWriteWB: 1
MEM_WB_memToReg / memToRegWB: 0

REGISTROS
reg0: 0
reg1: 0
reg2: 0
reg3: 0
reg4: 0
reg5: 0
reg6: 0
reg7: 0
reg8: 0
reg9: 0
reg10: 0
reg11: 0
reg12: 0
reg13: 0
reg14: 0
reg15: 0
reg16: 0
reg17: 0
reg18: 0
```

```

reg19: 0
reg20: 0
reg21: 0
reg22: 0
reg23: 0
reg24: 0
reg25: 0
reg26: 0
reg27: 0
reg28: 0
reg29: 0
reg30: 0
reg31: 0

```

```

MEMORIA RAM
memoryRamData[0]: 15
memoryRamData[1]: 0
memoryRamData[2]: 0
memoryRamData[3]: 0
memoryRamData[4]: 0
memoryRamData[5]: 0
memoryRamData[6]: 0
memoryRamData[7]: 0
memoryRamData[8]: 0
memoryRamData[9]: 0
memoryRamData[10]: 0
memoryRamData[11]: 0
memoryRamData[12]: 0
memoryRamData[13]: 0
memoryRamData[14]: 0
memoryRamData[15]: 0

```

Lo cual si lo comparamos con los registros de la simulación, obtenemos los mismos resultados:

<i>clk</i>	1
<i>reset</i>	0
<i>wr_enable3</i>	0
<i>read_addr1[4:0]</i>	11111
<i>read_addr2[4:0]</i>	11111
<i>write_addr3[4:0]</i>	11111
<i>write_data3[31:0]</i>	65535
<i>read_data1[31:0]</i>	0
<i>read_data2[31:0]</i>	0
<i>read_data_to_debug_0[31:0]</i>	0
<i>read_data_to_debug_1[31:0]</i>	15
<i>read_data_to_debug_2[31:0]</i>	1500
<i>read_data_to_debug_3[31:0]</i>	1000
<i>read_data_to_debug_4[31:0]</i>	1500
<i>read_data_to_debug_5[31:0]</i>	127
<i>read_data_to_debug_6[31:0]</i>	4
<i>read_data_to_debug_7[31:0]</i>	7
<i>read_data_to_debug_8[31:0]</i>	2511
<i>read_data_to_debug_9[31:0]</i>	15
<i>read_data_to_debug_10[31:0]</i>	2523
<i>read_data_to_debug_11[31:0]</i>	31
<i>read_data_to_debug_12[31:0]</i>	985
<i>read_data_to_debug_13[31:0]</i>	0
<i>read_data_to_debug_14[31:0]</i>	0
<i>read_data_to_debug_15[31:0]</i>	0

Prueba de Implementación 2

Código assembler cargado en la placa de desarrollo

Archivo de Texto de Entrada (fuente.asm)	Archivo de Salida para Cargar en la Placa (salida.coe)
addi 1, 1, 3000	20210bb8,
addi 2, 2, 1500	204205dc,
addi 3, 3, 1000	206303e8,
beq 2, 3, 1	10430001,
addi 4, 4, 1	20840001,
sub 5, 1, 3	00232822,
beq 1, 1, 2	10220002,
addi 6, 6, 7	20c60001,
addi 6, 6, 7	20e70001,
END	ffffff;

Resultado obtenido

```

PC COUNTER
FE_pc / pcFE: 13

LATCH IF-ID
IF_ID_instr / instrID: 0
IF_ID_pcNext / pcNextID: 13

LATCH ID-EX
ID_EX_aluControl / aluControlEX: 0
ID_EX_signImm / SigExtEX: 0
ID_EX_readData1 / readData1EX: 0
ID_EX_readData2 / readData2EX: 0
ID_EX_aluSrc / aluSrcEX: 0
ID_EX_aluShiftImm / aluShiftImmEX: 1
ID_EX_memWrite / memWriteEX: 0
ID_EX_memToReg / memToRegEX: 0
ID_EX_memReadWidth / memReadWidthEX: 0
ID_EX_rs / rsEX: 0
ID_EX_rt / rtEX: 0
ID_EX_rd / rdEX: 0
ID_EX_sa / shamtEX: 0
ID_EX_regDst / regDstEX: 1
ID_EX_loadImm / loadImmEX: 0
ID_EX_regWrite / regWriteEX: 1

LATCH EX-MEM
EX_MEM_writeReg / writeRegisterMEM: 0
EX_MEM_writeData / writeDataMEM: 0
EX_MEM_aluOut / aluOutMEM: 0
EX_MEM_regWrite / regWriteMEM: 1
EX_MEM_memWrite / memWriteMEM: 0
EX_MEM_memReadWidth / memReadWidthMEM: 0

LATCH MEM-WB
MEM_WB_writeReg / writeRegisterWB: 31
MEM_WB_aluOut / aluOutWB: 65535
MEM_WB_readData / memoryOutWB: 2147483648
MEM_WB_regWrite / regWriteWB: 0
MEM_WB_memToReg / memToRegWB: 0

REGISTROS
reg0: 0
reg1: 3000
reg2: 1500
reg3: 1500
reg4: 0
reg5: 1500
reg6: 1
reg7: 1
reg8: 0
reg9: 0
reg10: 0
reg11: 0
reg12: 0
reg13: 0
reg14: 0
reg15: 0

```



```
reg16: 0  
reg17: 0  
reg18: 0  
reg19: 0  
reg20: 0  
reg21: 0  
reg22: 0  
reg23: 0  
reg24: 0  
reg25: 0  
reg26: 0  
reg27: 0  
reg28: 0  
reg29: 0  
reg30: 0  
reg31: 0
```

MEMORIA RAM

```
memoryRamData[0]: 0  
memoryRamData[1]: 0  
memoryRamData[2]: 0  
memoryRamData[3]: 0  
memoryRamData[4]: 0  
memoryRamData[5]: 0  
memoryRamData[6]: 0  
memoryRamData[7]: 0  
memoryRamData[8]: 0  
memoryRamData[9]: 0  
memoryRamData[10]: 0  
memoryRamData[11]: 0  
memoryRamData[12]: 0  
memoryRamData[13]: 0  
memoryRamData[14]: 0  
memoryRamData[15]: 0
```

Conclusión

Con el presente trabajo práctico, se ha podido dilucidar las bases que rigen el funcionamiento de los procesadores al día de hoy y nos ha permitido lograr entender cuáles son las limitaciones que se tienen a la hora de diseñar un procesador en cuanto a tiempos de respuesta de los circuitos, sincronización, consistencia y coherencia de datos, entre otros.