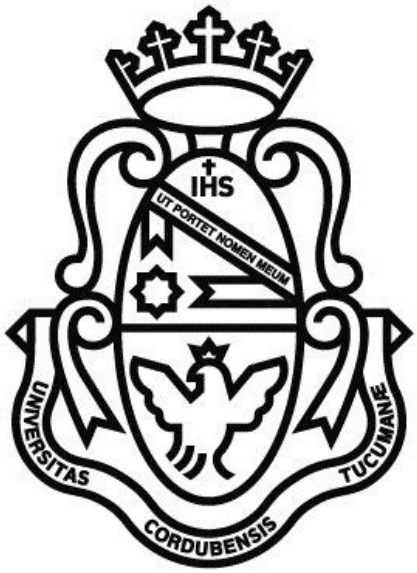
UNIVERSIDAD NACIONAL DE CÓRDOBA
Facultad de Ciencias Exactas, Físicas y Naturales

Ingeniería en Computación



ARQUITECTURA DE COMPUTADORAS

TRABAJO FINAL

“Descripción del microprocesador MIPS en FPGA”

INTEGRANTES:

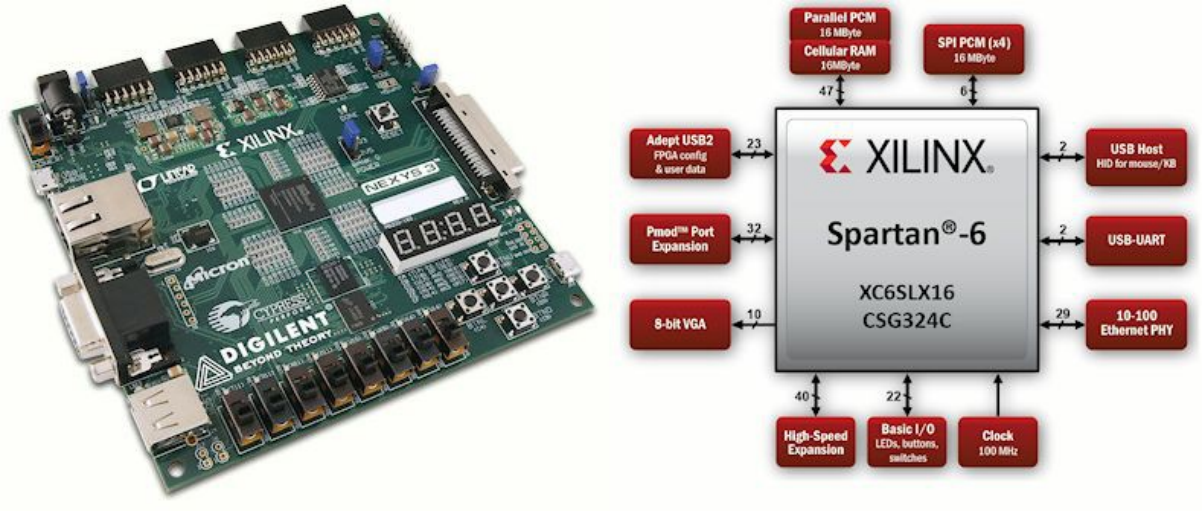
Provinciani, Diego Alejandro	33656734
Remonda, Aurelio Celestino	33012672
Serjoy, Héctor Andrés	34818490

DOCENTES:

Dr. Ing. Micolini, Orlando	Profesor de Teórico
Ing. Bechler, Renzo	Profesor de Práctico

Introducción

Durante el cursado de la asignatura Arquitectura de Computadoras y para el aprendizaje de la arquitectura interna de un microprocesador y su funcionamiento, se tomó como referencia el microprocesador MIPS desarrollado por la Universidad de Stanford, el cual implementa un pipeline para la ejecución de las instrucciones.
En el presente trabajo, y como cierre de dicha asignatura, nos planteamos describir en HDL un MIPS para luego ejecutarlo sobre una FPGA y mediante una interfaz de debbug, observar el live status del pipeline en ejecución.
El chip FPGA a utilizar es el Spartan6 provisto por la placa XILINX Nexis 3 que puede observarse en la imagen a continuación.



1 Placa XILINX Nexis 3 con chip FPGA Spartan 6.

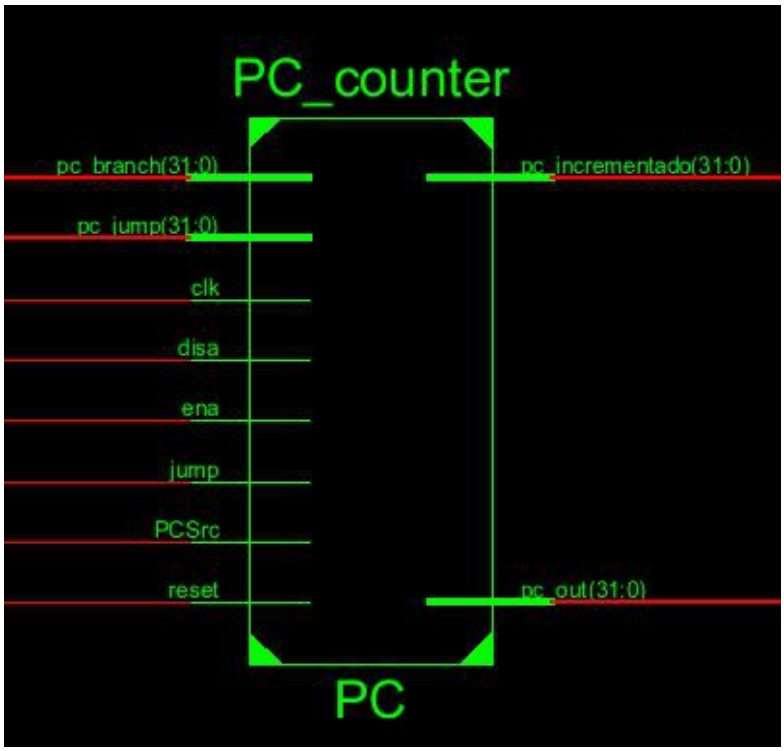
Por otra parte, como puede verse en la imagen 2, si bien el set de instrucciones a implementar no es el total de instrucciones que implementa MIPS, es un subconjunto que contiene instrucciones de los tres tipos (R, I y J).
Para poder hacer ejecutar un programa en el MIPS que se pretende describir, debió crearse un compilador que transforma las instrucciones en assembler a código máquina binario que es interpretado por el pipeline, para ésto la tecnología utilizada fue Python 2.7.

R-TYPE		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
SLL	Shift Left Logical	0 0 0 0 0 0		0 0 0 0 0	rd ← rs << shamt		0 0 0 0 0 0
SRL	Shift Right Logical	0 0 0 0 0 0		0 0 0 0 0	rd ← rs << shamt		0 0 0 0 1 0
SRA	Shift Right Arithmetic	0 0 0 0 0 0		0 0 0 0 0	rd ← rs << shamt		0 0 0 0 1 1
SLLV	Shift Left Logical Variable	0 0 0 0 0 0	rd ← rs << rt			0 0 0 0 0	0 0 0 1 0 0
SRLV	Shift Right Logical Variable	0 0 0 0 0 0	rd ← rs >> rt			0 0 0 0 0	0 0 0 1 1 0
SRAV	Shift Right Arithmetic Variable	0 0 0 0 0 0	rd ← rs >> rt			0 0 0 0 0	0 0 0 1 1 1
JR	Jump Register	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	Jump and Link Register	0 0 0 0 0 0	rd ← ret_addr	0 0 0 0 0	PC ← rs	0 0 0 0 0	0 0 1 0 0 1
ADD	Add	0 0 0 0 0 0	rd ← rs + rt			0 0 0 0 0	1 0 0 0 0 0
SUB	Subtract	0 0 0 0 0 0	rd ← rs - rt			0 0 0 0 0	1 0 0 0 1 0
AND	And	0 0 0 0 0 0	rd ← rs AND rt			0 0 0 0 0	1 0 0 1 0 0
OR	Or	0 0 0 0 0 0	rd ← rs OR rt			0 0 0 0 0	1 0 0 1 0 1
XOR	Exclusive Or	0 0 0 0 0 0	rd ← rs XOR rt			0 0 0 0 0	1 0 0 1 1 0
NOR	Not Or	0 0 0 0 0 0	rd ← rs NOR rt			0 0 0 0 0	1 0 0 1 1 1
SLT	Set on Less Than	0 0 0 0 0 0	rd ← (rs < rt)			0 0 0 0 0	1 0 1 0 1 0
	I-TYPE	Opcode	BASE o RS	RT	OFFSET o IMMEDIATE		
BEQ	Branch on Equal	4 0 0 0 1 0 0	if (rs = rt) then branch				
BNE	Branch on Not Equal	5 0 0 0 1 0 1	if (rs ≠ rt) then branch				
ADDI	Add Immediate Word	8 0 0 1 0 0 0	rt ← rs + immediate				
SLTI	Set on Less Than Immediate	10 0 0 1 0 1 0	rt ← (rs < immediate)				
ANDI	And Immediate	12 0 0 1 1 0 0	rt ← rs AND immediate				
ORI	Or Immediate	13 0 0 1 1 0 1	rt ← rs OR immediate				
XORI	Exclusive Or Immediate	14 0 0 1 1 1 0	rt ← rs XOR immediate				
LUI	Load Upper Immediate	15 0 0 1 1 1 1	0 0 0 0 0	rt ← immediate << 0^16			
LB	Load Byte	32 1 0 0 0 0 0	rt ← memory[base+offset]				
LH	Load Halfword	33 1 0 0 0 0 1	rt ← memory[base+offset]				
LW	Load Word	35 1 0 0 0 1 1	rt ← memory[base+offset]				
LBU	Load Byte Unsigned	36 1 0 0 1 0 0	rt ← memory[base+offset]				
LHU	Load Halfword Unsigned	37 1 0 0 1 0 1	rt ← memory[base+offset]				
LWU	Load Word Unsigned	39 1 0 0 1 1 1	rt ← memory[base+offset]				
SB	Store Byte	40 1 0 1 0 0 0	memory[base+offset] ← rt				
SH	Store Halfword	41 1 0 1 0 0 1	memory[base+offset] ← rt				
SW	Store Word	43 1 0 1 0 1 1	memory[base+offset] ← rt				
	J-TYPE	Opcode	INSTR_INDEX				
J	Jump	2 0 0 0 0 1 0					
JAL	Jump and Link	3 0 0 0 0 1 1					

2 Set de instrucciones implementadas en el pipeline.

Módulos del pipeline

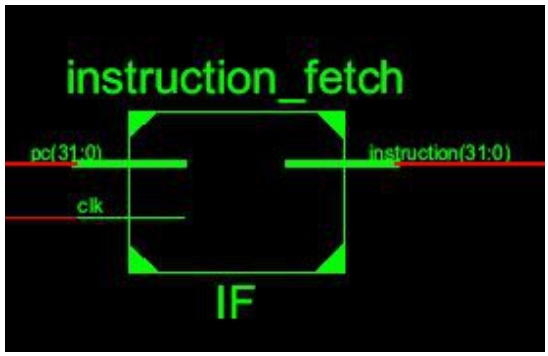
Program Counter



3 Esquemático de inputs y outputs para el Program Counter.

Módulo que selecciona el valor del *Program Counter*, de 32 bits, de acuerdo al orden de ejecución del programa. Las instrucciones de salto y salto condicional ponen en alto a las señales de control *jump* o *PCSrc* según sea el caso. El módulo PC se encarga de incrementar el *Program Counter* y seleccionar, de acuerdo a las señales de control anteriormente mencionadas, que valor de *Program Counter* de los ingresados corresponde poner en la salida *pc_out*.

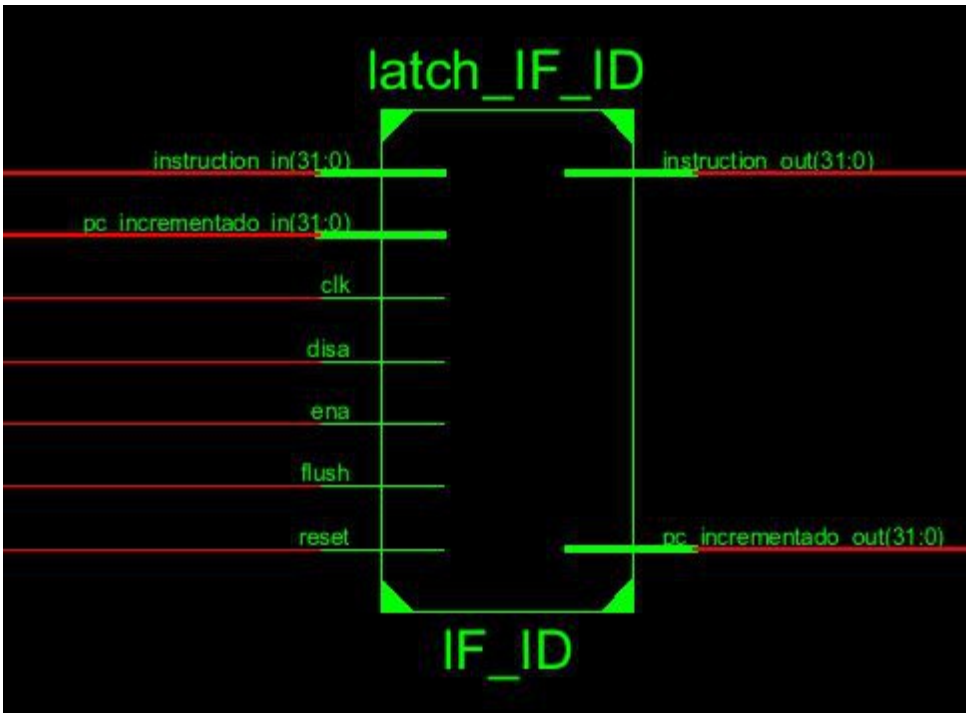
Instruction Fetch



4 Esquemático de inputs y outputs para la etapa N° 1 IF.

Este módulo implementa la memoria de instrucciones del programa, es decir la memoria que contiene el código binario de las instrucciones correspondientes al programa assembler que se desea ejecutar. La misma se encuentra implementada haciendo uso de un IP Core Memory, los cuales nos permiten cargar a la memoria de instrucciones los programas a ejecutar haciendo uso de archivos *.coe* y luego el programa es ejecutado de acuerdo a la entrada pc de este módulo. La salida del módulo es la instrucción leída del IP Core acorde a la posición indicada por el PC.

IF/ID Register



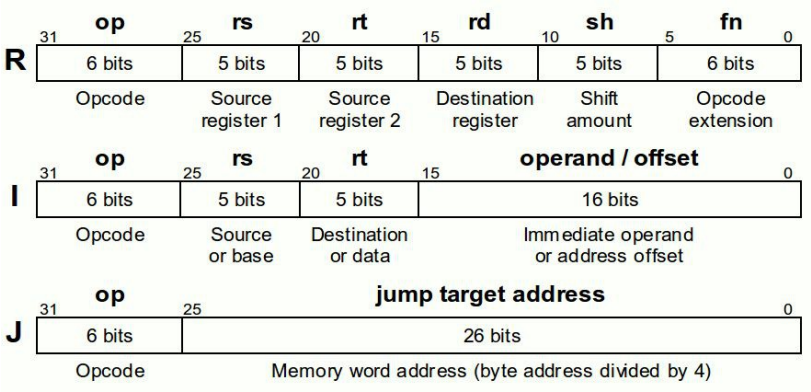
5 Esquemático de inputs y outputs para el latch que une las etapas IF ID.

Este, al igual que todos los latches ubicados entre etapas, se encarga de contener las salidas de la etapa anterior, para proveerles a la siguiente en el momento adecuado. En éste caso particular, el latch *IFID* se encuentra ubicado como su nombre lo indica entre las etapas de *Instruction Fetch* y *Instruction Decode*, y como puede observarse, se tienen como entradas al mismo la instrucción provista por la *Instruction Memory*, el valor del *Program Counter* incrementado en cuatro que será utilizado para realizar la búsqueda de la siguiente instrucción a ejecutar, demás señales de control y fundamentalmente el clock, que regirá el comportamiento de la toma de datos de la etapa anterior y la puesta a disposición para la siguiente.
El comportamiento de los latches es básicamente el mismo para todas las transiciones entre las distintas etapas del pipeline, por lo que para los demás, se proveen esquemáticos que muestran los datos que son temporalmente almacenados en los mismos y luego provistos a la siguiente etapa.

Instruction Decode

Inputs	Outputs
address_write[4:0]	addResult_Pc_Branch_D[31:0]
alu_result_EX[31:0]	ex_ALUOp_out[5:0]
data_write[31:0]	func_out[5:0]
instruction[31:0]	opcode_out[5:0]
pc_incrementado[31:0]	pc_jump[31:0]
reg_muxes_b[31:0]	rd[4:0]
clk	reg_data1[31:0]
ForwardAD	reg_data2[31:0]
ForwardBD	reg_0[31:0] ~ reg_31[31:0]
RegWrite	rs[4:0]
reset	rt[4:0]
	sgn_extend_data_imm[31:0]
	branch_out
	branch_taken_out
	ex_ALUSrc_out
	ex_RegDst_out
	jump_out
	m_MemWrite_out
	wb_MemtoReg_out
	wb_RegWrite_out

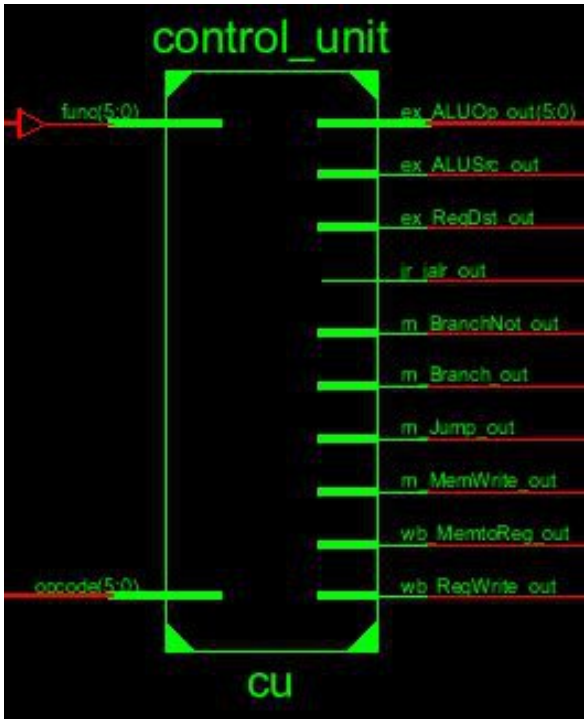
La presente etapa es la encargada de llevar a cabo la decodificación de las instrucciones, la misma parsea el binario de la instrucción de 32 bits que le llega de la etapa de instruction fetch en diferentes fragmentos que luego serán utilizados para realizar las operaciones correspondientes o como dato para realizar distintas acciones.
Las instrucciones según su tipo (R, I o J) tienen la siguiente estructura:



6 Campos de datos de las instrucciones según su tipo.

En la etapa se obtienen los datos de la instrucción que servirán para la ejecución de la misma y la escritura de datos en memoria o en los registros. La unidad de Hazard (HZU) trabaja en conjunto con ésta etapa para detectar y resolver los riesgos. ver Hazards Unit.

El módulo control unit (CU), interno de ésta etapa, es el encargado de generar las señales de control que le dirán al pipeline como se ejecutará la instrucción. Tiene como entradas los campos opcode y funct obtenidos de la instrucción.



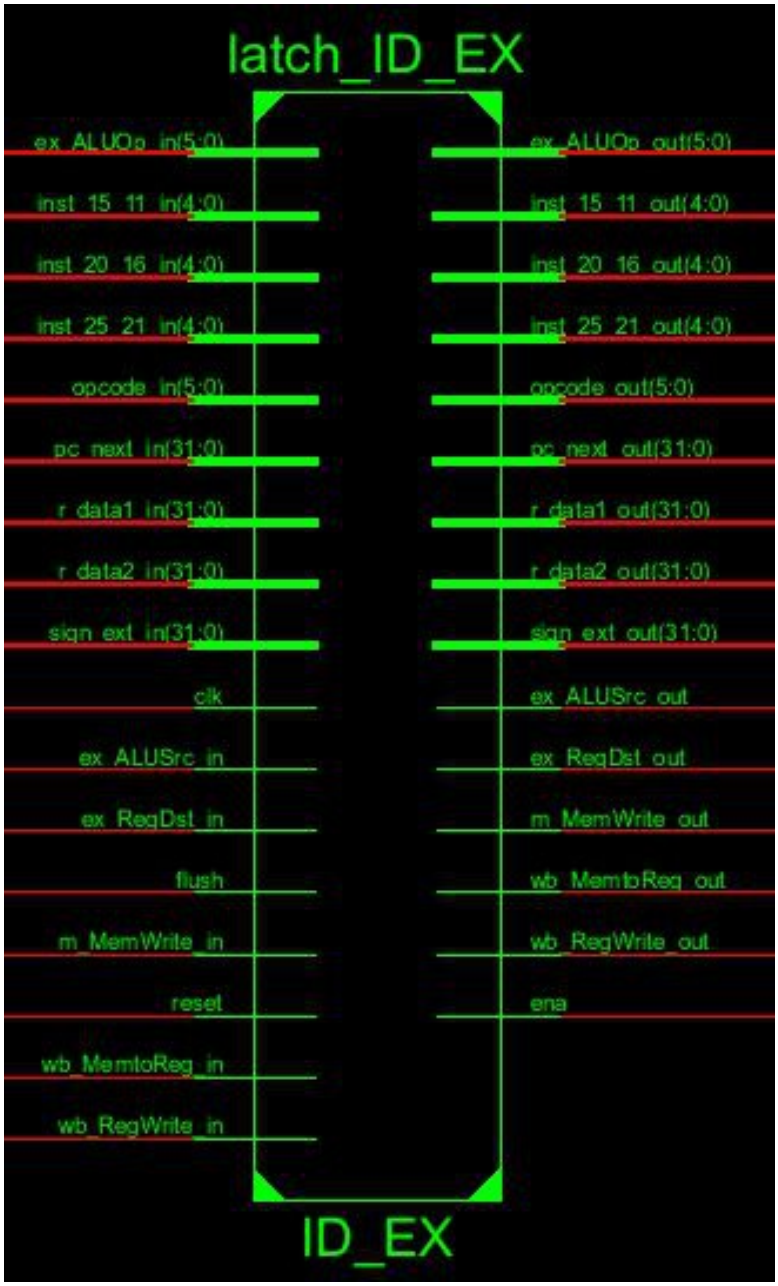
7 Esquemático de inputs y outputs del módulo generador de señales de control.

Estas señales de control (outputs de CU) se propagan a través del pipeline pasando de registro a registro, así cada etapa puede tomar decisiones a partir de ésta información. Por ejemplo en una instrucción Jump, la CU pone en alto m_Jump_out. Ésta información es tomada por el módulo PC e interpretada como un salto para así asignar el pc correspondiente.

También en ésta etapa se encuentra el banco de registros de 32 bits de ancho de palabra y con una capacidad de 32 palabras. Todos los registros son de propósitos generales.
Las instrucciones de salto, condicionales y no condicionales, son resueltas también en ID valiéndose de HZU para ejecutar flush y stalls.

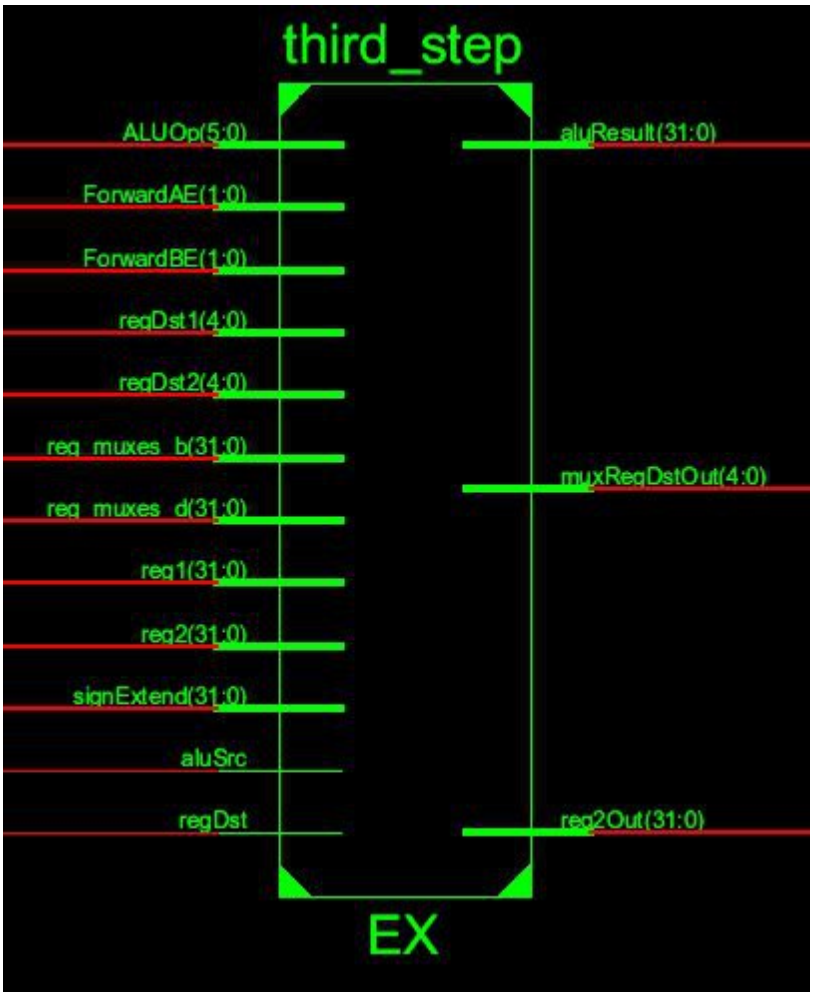
ID/EX Register

Ver IF/ID Register



8 Esquemático de inputs y outputs para el latch que une las etapas ID EX.

Instruction Execution



9 Esquemático de inputs y outputs para la etapa N° 3 EX.

En la presente etapa de ejecución, de acuerdo al valor de las señales de control la misma puede u optar por llevar a cabo una operación con la ALU, o realizar el cálculo de una dirección para efectuar un salto.

En el primer caso, la etapa se vale no solo del campo *funct* de la instrucción, sino también del *opcode*. Esto se debe a que si bien las las instrucciones que fundamentalmente hacen uso de la ALU son las de tipo R, y la operación que pretenden realizar puede ser determinada a través del campo *funct*, las operaciones de tipo I también llevan a cabo operaciones aritméticas o lógicas, y las mismas no cuentan con el campo *funct* entre sus valores, por ello es que se decidió hacer uso también en éste caso del *opcode*.

Luego, si el *opcode* indica que estamos frente a una instrucción de tipo R, evaluamos el campo *funct* para indicar a la ALU la operación que debe ejecutar, de lo contrario, la operación es determinada solamente haciendo uso del campo *opcode.*

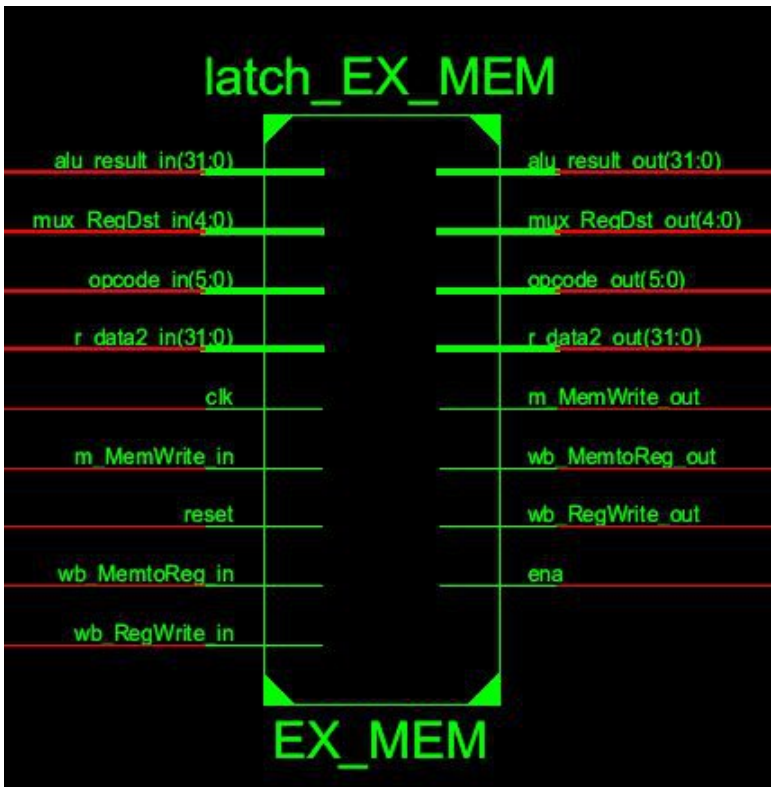
Todo el proceso antes mencionado es llevado a cabo por una pequeña unidad de control específica para la ALU contenida en ésta etapa. La cual indica a la ALU la operación que debe ejecutar haciendo uso del siguiente código:

```
alu_control == 4'b0000  →  ADD
alu_control == 4'b0001  →  SUB
alu_control == 4'b0010  →  AND
alu_control == 4'b0011  →  OR
alu_control == 4'b0100  →  XOR
alu_control == 4'b0101  →  NOR
alu_control == 4'b0110  →  SLT
alu_control == 4'b0111  →  SLL
alu_control == 4'b1000  →  SRL
alu_control == 4'b1001  →  SRA
alu_control == 4'b1010  →  SLLV
alu_control == 4'b1011  →  SRLV
alu_control == 4'b1100  →  SRAV
alu_control == 4'b1101  →  LUI
```

Por otra parte, puede ocurrir el caso de esta frente a una instrucción de tipo J en la que se deba ejecutar un salto y consecuentemente calcular la instrucción de destino. En éste caso, la etapa lleva a cabo, fuera de la ALU, un shift y una suma con el valor actual del PC para calcular su nuevo valor, el cual es enviado a través de una conección con el latch que comunica con la etapa de acceso a memoria.

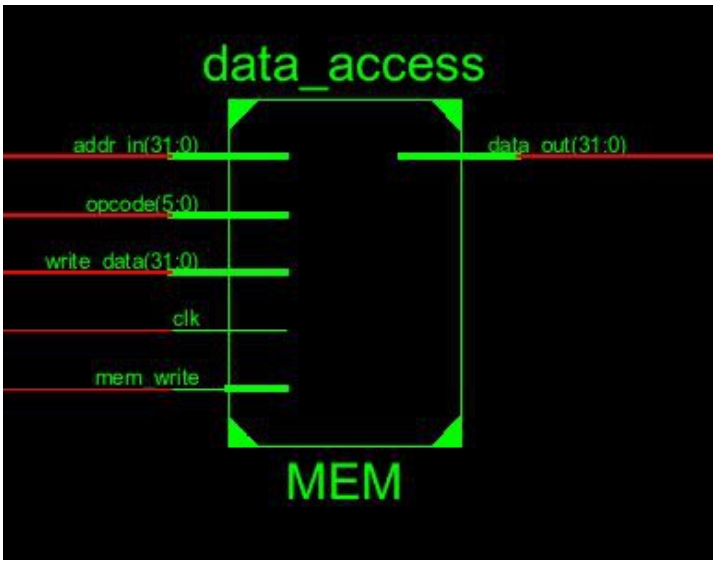
EX/MEM Register

Ver IF/ID Register



10 Esquemático de inputs y outputs para el latch que une las etapas EX MEM.

Memory Stage

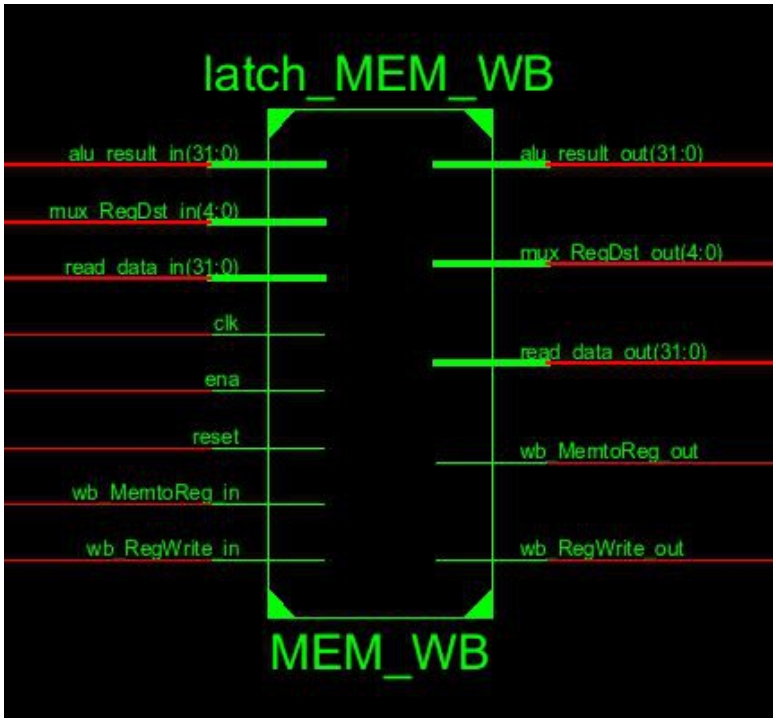


11 Esquemático de inputs y outputs para la etapa de acceso a Data Memory.

La presente es una etapa muy simple en la que pueden llevarse a cabo fundamentalmente dos tareas, o escribir datos en memoria como resultados de operaciones que deben ser almacenados para luego ser utilizados, o lectura de datos previamente almacenados en memoria para ser utilizados en las siguiente instrucciones.

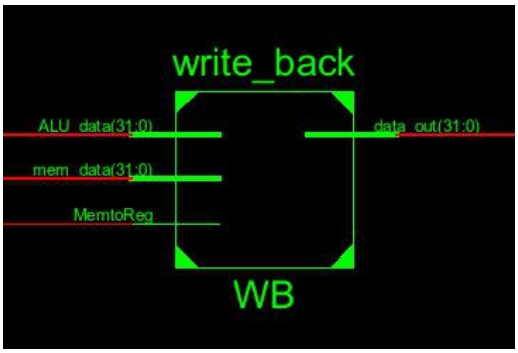
MEM/WB Register

Ver IF/ID Register



12 Esquemático de inputs y outputs para el latch que une las etapas MEM WB.

Write Back



13 Esquemático de inputs y outputs para la etapa de Write Back.

Al igual que la anterior, la actual es una etapa muy simple que permite concluir con el uso de una instrucción. En ésta siendo la última etapa, como se observa en el esquemático se tienen fundamentalmente dos datos de entrada, uno de ellos el resultado de la operación ejecutada en la ALU, y el otro, el dato proporcionado por la memoria de datos.
La señal de control *MemtoReg*, será utilizada para definir cual de los dos datos antes mencionados es utilizado y puesto a disposición en *data_out* para en ese caso, ser almacenado en alguno de los registros.

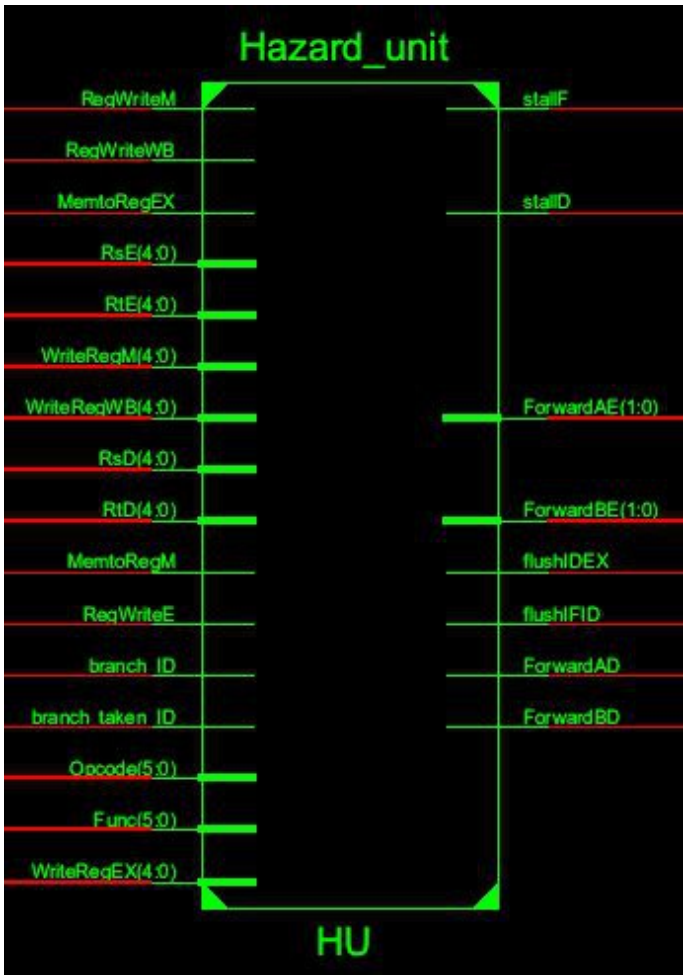
Hazards Unit

Ésta unidad es la encargada de detectar los riesgos y enviar señales para resolverlos.
Aquí se comparan los registros utilizados por la instrucción (rs, rd, rt) en las etapas ID y EX, para detectar dependencias y actuar en consecuencia haciendo stall (detención de una instrucción en una etapa), forwarding (cortocircuitar valores de una etapa posterior hacia una anterior) y flush (limpiar un latch) según cual sea el riesgo.

Riesgo de dependencia de datos, cuando una instrucción en etapa de ejecución utiliza como operador a un registro que todavía no fue escrito en el banco, es decir rs o rt de la instrucción en EX es igual a rs en etapa MEM o WB. Si no se toma alguna decisión (frenar el pipe o cortocircuitar resultados) la instrucción tomará valores incorrectos de los registros. Si se detecta un riesgo de este tipo la HZU envía las señales de control (ForwardAE o ForwardBE) a los muxes de la etapa EX para seleccionar como operador el valor cortocircuitado directamente de la etapa MEM o WB.

Las instrucciones load generan un riesgo particular, una lw recibe datos desde la memoria en la etapa MEM, la siguiente instrucción necesita ese dato en etapa EX. Cortocircuitar no es suficiente para éste caso y es necesario detener el pipeline durante un ciclo. Para esto la HZU utiliza las señales stallD y stallF que mantienen las instrucciones que se encuentran en las etapas decode y fetch durante un ciclo. Además se pone en alto la señal flushIDEX que inserta una “burbuja”, es decir todas las entradas a EX en el siguiente ciclo se ponen en cero.

Las instrucciones de salto al ser detectadas en la etapa ID requieren algún mecanismo para eliminar la instrucción anterior que ya está en la etapa IF. La HZU detecta este caso y envía un alto en la señal flushIFID que está conectada al flush del la***14*** tch IFID.



14 Esquemático de inputs y outputs para la unidad de control de riesgos.

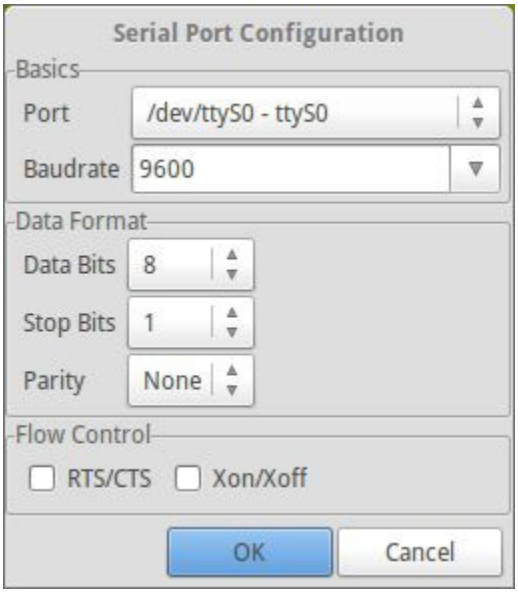
Unidad debbuger

Para la prueba y el testeo del correcto funcionamiento de los programas cargados a la placa, fue desarrollada una interfaz que permite observar el estado en tiempo real del pipeline.

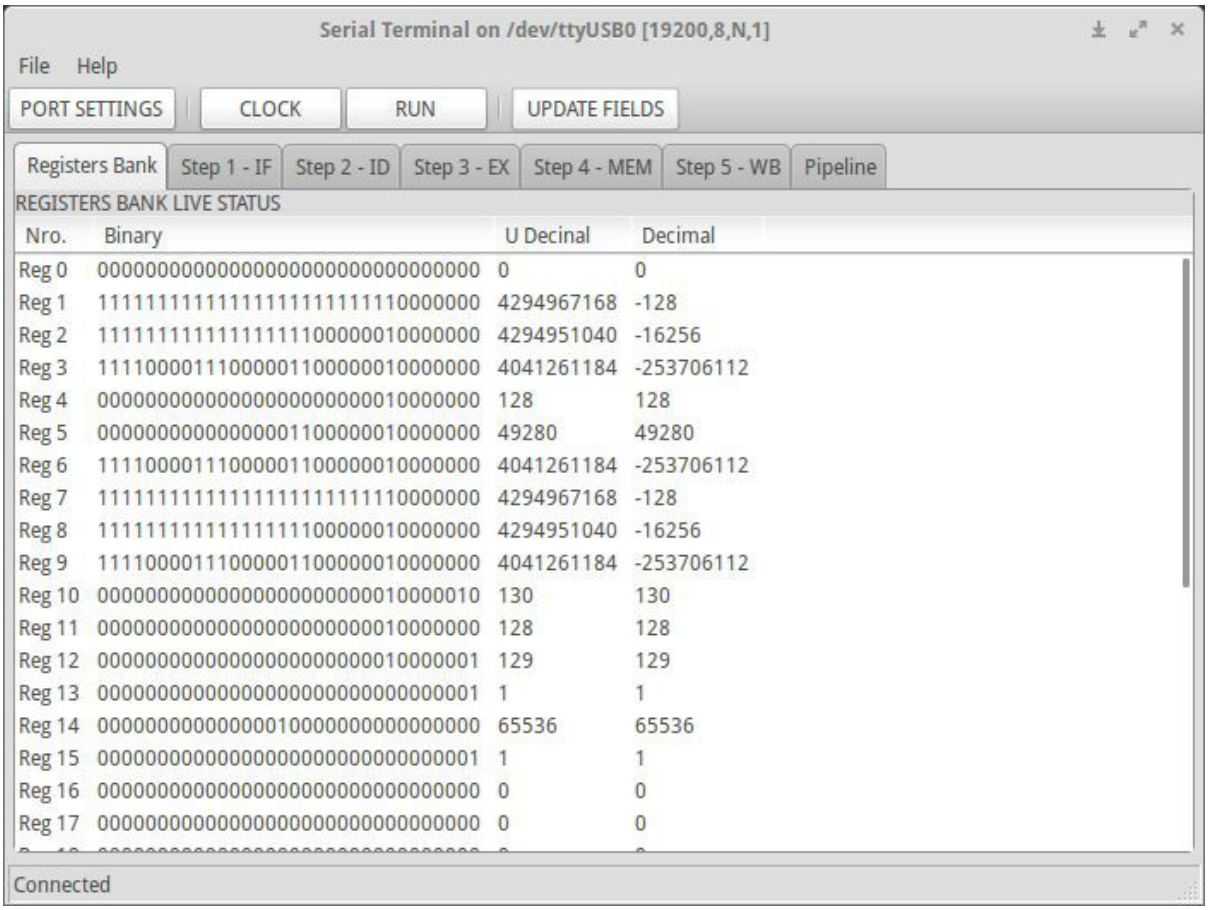
En dicha interfaz, pueden verse los valores almacenados en los 32 registros, las señales de control junto a las entradas y salidas de cada etapa y además un live status del pipeline que nos muestra las 5 instrucciones que se encuentran siendo ejecutadas en un instante determinado, indicando la etapa en la que se encuentran y la lista de las que ya han finalizado su ejecución.

Por último, para facilitar la configuración del puerto serie a utilizar, se provee un menú que permite seleccionar al usuario el puerto, la velocidad de transmisión de símbolos, cantidad de bits de datos, bits de stop, etc.

A continuación se muestran capturas de ambas i***15*** nterfaces.



15 Interfaz de configuración de pue16 rto serie.



16 Interfaz que muestra el live status del pipeline.

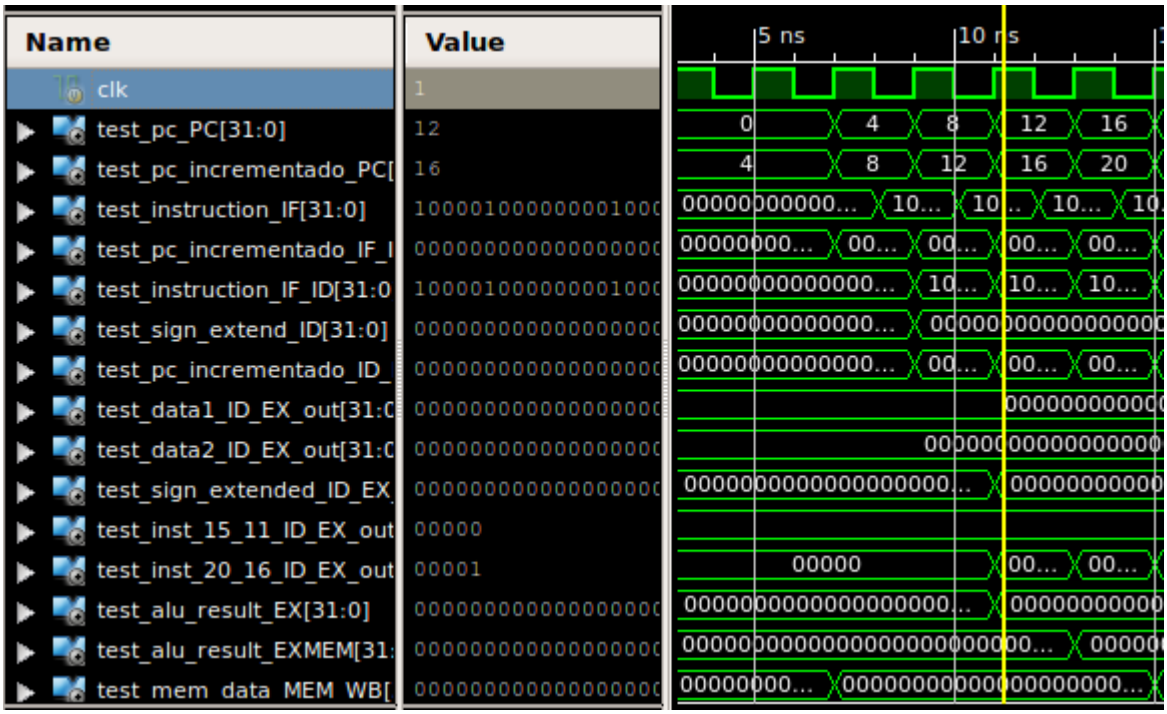
Testing

Test Bench

Para comprobar el funcionamiento del pipeline en primer lugar, y del proyecto finalizado con unidad de debugger y pipeline implementado en segundo lugar, se realizaron dos test bench, pipeline_tb.v y tp_final_tb.v.

Test bench del modulo pipeline

Id Caso de Prueba	01	
Tipo de prueba	Test Bench	
Objetivos de la prueba	Comprobar el correcto funcionamiento del modulo “pipeline”	
Descripción	Se instancia el modulo pipeline. Se simula un clock, se habilita el pipeline y se ejecuta el programa de forma continua.	
Prerrequisitos de la prueba		
1. dataMemory.coe cargado en ipcore dataMemory 2. test_1.coe cargado en ipcore instructionMemory		
Procedimientos		
1. Abrir el proyecto en Xilinx 2. Ejecutar pipeline_tb.v 3. Verificar las distintas señales y su avance clock a clock 4. Verificar el estado final de los registros		
Resultado esperado		
Estado final de los registros reg_17 = 2 reg_18 = 3 reg_19 = 5 reg_20 = 7 reg_22 = 5 reg_23 = 3 reg_24 = 0 reg_25 = -2 Todos los demas: 0		
Resultados obtenidos		
Fueron los esperados.		
Observaciones		
Resultado de la prueba	x	Aprobado
		No aprobado



En el test se puede ver un gran numero de señales provenientes de los distintos modulos del pipeline
La convencion que se utilizo para nombrarlas es la siguiente:

“test_” + “descripcion” + “etapa o registro del que sale la señal”

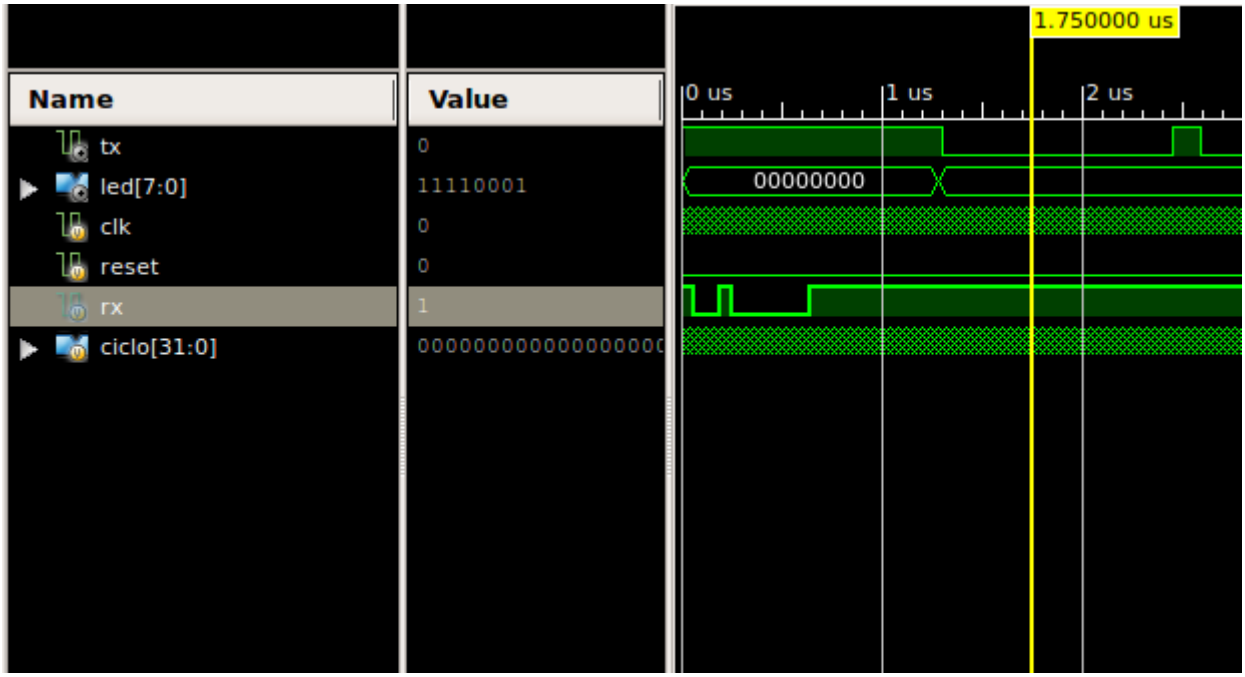
Por ejemplo, el resultado de la operación realizada en la ALU se puede ver en el test bench como

test_alu_result_EX

Es posible testear nuevas señales agregandolas tanto a la salida del pipeline como al test bench.

Test bench del modulo tp final

Id Caso de Prueba	02	
Tipo de prueba	Test Bench	
Objetivos de la prueba	Comprobar la recepcion y envio de datos en el modulo “tp_final” y la correcta ejecucion del pipeline	
Descripción	Se instancia el modulo tp_final. Se simula un clock y se simula el envio de datos hacia “tp_final” para luego comprobar la devolucion de datos del mismo.	
Prerrequisitos de la prueba		
1. dataMemory.coe cargado en ipcore dataMemory 2. test_1.coe cargado en ipcore instructionMemory		
Procedimientos		
1. Abrir el proyecto en Xilinx 2. Ejecutar tp_final_tb.v 3. Corroborar que se comienze una transmision cada vez que se recibe un dato. Si rx recibe un dato, tx comienza a transmitir		
Resultado esperado		
Envio de datos cada vez que se recibe un dato.		
Resultados obtenidos		
Fueron los esperados.		
Observaciones		
Resultado de la prueba	x	Aprobado
		No aprobado



System Tests

Para comprobar el correcto funcionamiento del pipeline se realizaron cuatro tests. En los mismos se prueba la correcta ejecucion de todas las instrucciones implementadas asi como tambien el correcto manejo de los hazards que se presentan en los distintos casos de ejecucion.

Id Caso de Prueba	01
Tipo de prueba	Sistema
Objetivos de la prueba	Verificar el correcto funcionamiento del pipeline.
Descripción	Test breve (ocho instrucciones) donde se comprueba el correcto funcionamiento del pipeline a travez de la verificacion del estado final de los registros al final de la ejecuccion.
Prerrequisitos de la prueba	
3. Cargar el archivo “test_1.bit” (adjunto con la documentacion del proyecto) en la placa FPGA. 4. Abrir el debugger (adjunto con la documentacion del proyecto)	
Procedimientos	
1. En el debugger ejecutar un “clock” haciendo click sobre dicho boton. 2. Refrescar la pantalla con el boton “refrescar” 3. Verificar el estado del pipeline en sus distintas etapas 4. Repetir el paso 1 a 3 hasta finalizar la ejecucion del programa 5. Verificar el estado final de los registros	
Resultado esperado	
Estado final de los registros reg_17 = 2 reg_18 = 3 reg_19 = 5 reg_20 = 7 reg_22 = 5 reg_23 = 3 reg_24 = 0 reg_25 = -2 Todos los demas: 0	
Resultados obtenidos	
Fueron los esperados.	
Observaciones	
Resultado de la prueba	x Aprobado
	No aprobado

Id Caso de Prueba	02	
Tipo de prueba	Sistema	
Objetivos de la prueba	Verificar que las instrucciones de salto se ejecuten correctamente.	
Descripción	Instrucciones probadas: J, JR, BEQ, BNE. Hazards probados: stall para BEQ y BNE. Flush de la instrucción inmediatamente posterior.	
Prerrequisitos de la prueba		
1. Cargar el archivo “test_2.bit” (adjunto con la documentacion del proyecto) en la placa FPGA. 2. Abrir el debugger (adjunto con la documentacion del proyecto)		
Procedimientos		
1. En el debugger ejecutar un “clock” haciendo click sobre dicho boton. 2. Refrescar la pantalla con el boton “refrescar” 3. Verificar el estado del pipeline en sus distintas etapas 4. Repetir el paso 1 a 3 hasta finalizar la ejecucion del programa 5. Verificar el estado final de los registros		
Resultado esperado		
Estado final de los registros reg_16 = 142 reg_17 = 2 reg_18 = 5 reg_19 = 5 reg_20 = 7 reg_21 = 2 reg_22 = 7 reg_23 = 60 reg_24 = 5 reg_25 = 62 reg_26 = 80 Todos los demas: 0		
Resultados obtenidos		
Fueron los esperados.		
Observaciones		
Resultado de la prueba	x	Aprobado
		No aprobado

Id Caso de Prueba	03	
Tipo de prueba	Sistema	
Objetivos de la prueba	Verificar que las instrucciones tipo-R se ejecuten correctamente. Verificar que los hazards se manejen correctamente	
Descripción	Instrucciones probadas: LW, SLL, SRL, SRA, SLLV, SRLV, SRAV, ADD, SUB, AND, OR, XOR, NOR, SLT Hazards probados: • stall y flush para LW. • Cortocircuitos para dependencias.	
Prerrequisitos de la prueba		
1. Cargar el archivo “test_3.bit” (adjunto con la documentacion del proyecto) en la placa FPGA. 2. Abrir el debugger (adjunto con la documentacion del proyecto)		
Procedimientos		
1. En el debugger ejecutar un “clock” haciendo click sobre dicho boton. 2. Refrescar la pantalla con el boton “refrescar” 3. Verificar el estado del pipeline en sus distintas etapas 4. Repetir el paso 1 a 3 hasta finalizar la ejecucion del programa 5. Verificar el estado final de los registros		
Resultado esperado		
Estado final de los registros reg_1 = 2 reg_2 = 3 reg_3 = 5 reg_4 = 7 reg_5 = 10000000_00000000_00000000_00000001 reg_10= 00000000_00000000_00000000_00010000 reg_11 = 00001000_00000000_00000000_00000000 reg_12= 11111000_00000000_00000000_00000000 reg_13= 00000000_00000000_00000000_00001000 reg_14= 00010000_00000000_00000000_00000000 reg_15= 11110000_00000000_00000000_00000000 reg_16= 10 reg_17= 7 reg_18= 2 reg_19= 10 reg_21= 7 reg_22= 1 reg_23= 0		
Resultados obtenidos		
Fueron los esperados.		
Resultado de la prueba	x	Aprobado
		No aprobado

Id Caso de Prueba	04	
Tipo de prueba	Sistema	
Objetivos de la prueba	Verificar que las instrucciones tipo-I se ejecuten correctamente. Verificar que los hazards se manejen correctamente.	
Descripción	Instrucciones probadas: LB, LH, LW, LBU, LHU, LWU, SB, SH, SW, ADDI, ANDI, ORI, XORI, LUI, SLTI. Hazards probados: • stall y flush para LW. • Cortocircuitos para dependencias.	
Prerrequisitos de la prueba		
1. Cargar el archivo “test_4.bit” (adjunto con la documentacion del proyecto) en la placa FPGA. 2. Abrir el debugger (adjunto con la documentacion del proyecto)		
Procedimientos		
1. En el debugger ejecutar un “clock” haciendo click sobre dicho boton. 2. Refrescar la pantalla con el boton “refrescar” 3. Verificar el estado del pipeline en sus distintas etapas 4. Repetir el paso 1 a 3 hasta finalizar la ejecucion del programa 5. Verificar el estado final de los registros		
Resultado esperado		
Estado final de los registros reg_1 11111111_11111111_11111111_10000000 reg_2 11111111_11111111_11000000_10000000 reg_3 11110000_11100000_11000000_10000000 reg_4 00000000_00000000_00000000_10000000 reg_5 00000000_00000000_11000000_10000000 reg_6 11110000_11100000_11000000_10000000 reg_7 00000000_00000000_00000000_10000000 reg_8 00000000_00000000_11000000_10000000 reg_9 11110000_11100000_11000000_10000000 reg_10 130 reg_11 128 reg_12 129 reg_13 1 reg_1400000000_00000001_00000000_00000000 reg_15 1		
Resultados obtenidos		
Fueron los esperados.		
Resultado de la prueba	x	Aprobado
		No aprobado

Codigo fuente System Test

Test 1

Codigo	Operacion
`NOP ;//NOP para que func. correctamente la interfaz grafica del debugger`	
`LW $17, 0($16);`	`//2 -> $17 (carga un 2 en el registro 20)`
`LW $18, 4($16);`	`//3 -> $18`
`LW $19, 8($16);`	`//5 -> $19`
`LW $20, 12($16);`	`//7 -> $20`
`ADD $22, $17, $18;`	`//2 + 3 = 5 -> $22`
`OR  $23, $18, $17;`	`//010 || 011 = 011 -> $23`
`AND $24, $17, $19;`	`//010 || 101 = 000 -> $24`
`SUB $25, $19, $20;`	`//5 - 7 = -2 -> $25`

Test 4

Codigo	Operacion
`LB $1, 28($0);`	`//11111111_11111111_11111111_10000000 -> $1`
`LH $2, 28($0);`	`//11111111_11111111_11000000_10000000 -> $2`
`LW $3, 28($0);`	`//11110000_11100000_11000000_10000000 -> $3`
`LBU $4, 28($0);`	`//00000000_00000000_00000000_10000000 -> $4`
`LHU $5, 28($0);`	`//00000000_00000000_11000000_10000000 -> $5`
`LWU $6, 28($0);`	`//11110000_11100000_11000000_10000000 -> $6`
`SB $6, 32($0);`	`//00000000_00000000_00000000_10000000 -> memoria`
`SH $6, 36($0);`	`//00000000_00000000_11000000_10000000 -> memoria`
`SW $6, 40($0);`	`//11110000_11100000_11000000_10000000 -> memoria`
`ADDI $10, $4, 2;`	`//130 -> $10`
`ANDI $11, $4, 129;`	`//128 -> $11`
`ORI $12, $4, 1;`	`//129 -> $12`
`XORI $13, $4, 129;`	`//1 -> $13`
`LUI $14, 1;`	`//00000000_00000001_00000000_00000000 -> $14`
`SLTI $15, $4, 129;`	`//1 -> $15`
`LW $7, 32($0);`	`//00000000_00000000_00000000_10000000 -> $7`
`LW $8, 36($0);`	`//00000000_00000000_11000000_10000000 -> $8`
`LW $9, 40($0);`	`//11110000_11100000_11000000_10000000 -> $9`

Test 2

Codigo	Linea	Operacion
LW $19, 4($16);	0	//3 -> $19 (carga un 3 en el reg 19)
LW $18, 8($16);		//5 -> $18
LW $20, 12($16);		//7 -> $20
LW $25, 20($16);	12	//62 -> $25
LW $26, 16($16);		//80 -> $26
LW $17, 0($16);	20	//2 -> $17
AND $21, $17, $19;		//2 & 3 = 2 -> $21
OR $22, $18, $17;	28	//5 | 2 = 7 -> $22
SUB $23, $25, $17;		//62 - 2 = 60 -> $23
JR $23; 36		//jump a 60
AND $16, $21, $19;	40	
AND $16, $22, $19;		
AND $16, $23, $19;	48	
AND $16, $17, $19;		
AND $16, $20, $19;	56	
JR $26; 60		//jump a 80
AND $16, $21, $19;	64	
AND $16, $22, $19;		
AND $16, $23, $19;	72	
AND $16, $20, $19;		
AND $24, $20, $18;	80	//7 & 5 = 5 -> $24
BEQ $24, $18, 4;	84	//$24 == $18 (5 == 5) ? salto a 104 (se toma el salto)
J 20;	88	
J 32;		
J 64;		
J 160;		
BEQ $26, $25, 5;	104	//$26 == $25 (80 == 62) ? salto 108 (el salto no se toma)
AND $16, $21, $19;	108	//2 & 3 = 2 -> $16
AND $16, $22, $19;	112	//7 & 3 = 3 -> $16
AND $16, $23, $19;	116	//60 & 3 = 0 -> $16
AND $16, $20, $19;	120	//7 & 3 = 3 -> $16
OR $22, $18, $17;	124	//5 | 2 = 7 -> $22
SUB $23, $25, $17;	128	//62 - 2 = 60 -> $23
J 37;	132	//jump a 148
AND $16, $21, $19;	136	
AND $16, $22, $19;	140	
AND $16, $23, $19;	144	
ADD $16, $25, $26;	148	//62 + 80 = 142 -> $16
AND $19, $20, $18;	152	//7 & 5 = 5 -> $19
BNE $16, $19, 4;	156	//$16 != $19 (142 != 5) ? salto a 176 (el salto se toma)
J 20;	160	
J 12;	164	
J 64;	168	
J 4;	172	
BNE $20, $22, 4;	176	$20 = 7 == $22 = 7 ? no! sigue a 180
AND $16, $21, $19;	180	$16 = 2 & 5 = 0
AND $16, $22, $19;	184	$16 = 7 & 5 = 5
AND $16, $23, $19;	188	$16 = 60 & 5 = 0
ADD $16, $25, $26;	192	$16 = 62 + 80 = 142
AND $19, $20, $18;	196	$19 = 7 | 5 = 5

Test 3

Codigo	**Operacion**

```
NOP ;
LW $1, 0($0);          //2 -> $1
LW $2, 4($0);          //3 -> $2
LW $3, 8($0);          //5 -> $3
LW $4, 12($0);         //7 -> $4
LW $5, 24($0);         //10000000_00000000_00000000_00000001 -> $5
SLL $10, $5, 4;        //00000000_00000000_00000000_00010000 -> $10
SRL $11, $5, 4;        //00001000_00000000_00000000_00000000 -> $11
SRA $12, $5, 4;        //11111000_00000000_00000000_00000000 -> $12
SLLV $13, $5, $2;      //00000000_00000000_00000000_00001000 -> $13
SRLV $14, $5, $2;      //00010000_00000000_00000000_00000000 -> $14
SRAV $15, $5, $2;      //11110000_00000000_00000000_00000000 -> $15
ADD $16, $4, $2;       //7 + 3 = 10 -> $16
SUB $17, $16, $2;      //10 - 3 = 7 -> $17
AND $18, $17, $16;     //0111 & 1010 = 2 -> $18
OR $19, $18, $16;      //0010 | 1010 = 10 -> $19
XOR $20, $1, $3;       //010 xor 101 = 111 (7) -> $20
NOR $21, $1, $3;       //00...0010 nor 00...0101 = 11...1000 (-8) -> $21
SLT $22, $1, $3;       //2 < 5 = 1 -> $22
SLT $23, $3, $1;       //5 < 2 = 0 -> $23
```

Conclusión

El uso de la fpga en un proyecto tan extenso como la implementación de un procesador MIPSlike nos dio una muestra de las posibilidades que da esta tecnología. Nos dejó ver la versatilidad que da un lenguaje de descripción de hardware al llevar a cabo funciones lógicas simples y reprogramar el chip para hacer cosas tan complejas como un procesador. Es muy importante notar y comprender que un dispositivo con estas características, nos da la posibilidad de llevar a cabo desarrollos de ingeniería que luego pueden ser implementados una vez que fueron ampliamente testeados en todos los casos y situaciones a las que puede estar sometido en su uso real.

Por otra parte, el desarrollo del pipeline en conjunto con el compilador de instrucciones y el debugger que permite ver el live status del mismo, nos ayudó a comprender mucho más sobre la arquitectura de los procesadores en general y de muchos de los temas que abarcan el plan teórico de la materia.

Además, comprendemos como al partir de un buen diseño, es muy simple llevar a cabo una implementación, que si bien puede tener sus inconvenientes durante el proceso de desarrollo, los mismos son inherentes a la codificación en sí y no al funcionamiento del diseño que se desea implementar.