Implementación procesador MIPS
Facultad de Ciencias Exactas, Físicas y Naturales

Sardoy, Juan Manuel[1], **Perez, Federico**[1]

[1]Arquitectura de computadoras - FCEFyN
Av. Velez Sarsfield 1600 – Córdoba – Argentina

jmsardoy@gmail.com, 0xfede07c8@gmail.com

Resumen. *Este documento describe, de manera sintética, la implementación práctica del pipeline de un procesador que cumple con un subconjunto del set de instrucciones de la arquitectura MIPS IV de 32 bits. La implementación está basada en tecnologia FPGA, mediante el lenguaje Verilog. Este trabajo tiene como objeto el aprendizaje de los diferentes elementos más comunes de la arquitectura de computadoras, dado el encuadre académico del mismo.*
Palabras clave*: MIPS, FPGA, pipeline.*

Abstract. *This document describes, in a synthetic way, a practical implementation of the pipeline of a processor that is compliant with a subset of the MIPS IV 32bits architecture instruction set. This job has as objective, to build the basic knowledge of the computer's architecture cope of study.*
Keywords*: MIPS, FPGA, pipeline.*

1. Introducción

El trabajo, se encuentra dentro del marco de la asignatura de Arquitectura de Computadoras, de la FCEFyN de la Universidad Nacional de Córdoba, Argentina. Corresponde al trabajo final de la misma, el cual es requerimiento para su aprobación.

Además de la implementación del procesador en sí, se mostrará como implementar los elementos auxiliares requeridos para su funcionamiento y programación, entre otras utilidades. El hardware utilizado será un dispositivo FPGA, de la marca Xilinx, al ser de esta empresa, con los que cuentan los autores del trabajo.

Dentro de los elementos auxiliares, encontraremos el desarrollo de un sistema de comunicación directo con el procesador, mediante el protocolo UART. Además de este, un mecanismo para cargar un programa en la RAM del procesador, como asi también funcionalidad de debugging (ejecución paso a paso y salída de valores de registros internos). También se mostrará la implementación y uso de software auxiliar para el ensamblado de los programas a ejecutar.

Todo esto compondrá una suite básica de desarrollo de software de propósito general, cuyo entendimiento es el objetivo principal del trabajo y la materia.

2. Referencias Teóricas

La técnica de *pipeline* es la más común utilizada para la implementación de procesadores modernos de diferentes arquitecturas, dado el gran aumento de *IPC* (instrucciones por ciclo) que este provee.

MIPS son las siglas de *Microprocessor without interlocked pipeline stages*. Es una arquitectura RISC (*reduced instruction set computer* desarrollada por MIPS Computer Systems. Existen versiones de 32 y 64 bits de dicha arquitectura. La primera versión de fue desarrollada en la Universidad de Stanford en 1981 por un equipo liderado por John. L. Hennesy. La idea era diseñar un procesador segmentado (idea ya muy conocida en esos momentos) pero sin interbloqueo entre capas. En esa época eran comunes los procesadores donde cada instrucción iba ejecutandose por cada capaz, mientras las otras permanecian inactivas. Esto llevaba a que el tiempo que demora en ejecutar la instrucción fuera el del tiempo en el que tarda en pasar por todas las capas, en contraste con el tiempo del camino crítico, como ocurre sin interbloqueo.

Por estos motivos este trabajo tendrá como cuerpo principal, la aplicación de dicha técnica, para lo cual, a modo de guía, se seguirá la bibliografía de *Patterson y Hennesy*, dado que expone y soluciona los problemas más comunes a la hora de la creación de un *pipeline* genérico. Sin embargo, muchas partes de la implementación, quedan a cargo, y serán solucionadas por los autores, lo cual lo hace una implementación única de la arquitectura.

3. Metodología

Se utilizará una metodología incremental y modular para la implementación del pipeline y sus módulos complementarios. Esto mejora el entendimiento y fragmentación del problema, asi como también facilta su implementación, dado la complejidad del mismo. También esto es muy util a la hora del testing y el debugeado del sistema.

4. Requerimientos del trabajo

Los requerimientos provistos por la cátedra fueron los siguientes:

1. Implementar el pipeline de un procesador MIPS, segmentado en las siguientes etapas:
    a) **IF** (Instruction Fetch): Búsqueda de la instrucción en la memoria de programa.
    b) **ID** (Instruction Decode): Decodificación de la instrucción y lectura de registros.
    c) **EX** (Excecute): Ejecución de la instrucción propiamente dicha.
    d) **MEM** (Memory Access): Lectura o escritura desde/hacia la memoria de datos.
    e) **WB** (Write back): Escritura de resultados en los registros.
2. En dicha arquitectura, implementar las siguientes instrucciones:
    a) **R-type** (SLL, SRL, SRA, SLLV, SRLV, SRAV, ADDU, SUBU, AND, OR, XOR, NOR, SLT?
    b) **I-Type** (LB, LH, LW, LWU, LBU, LHU, SB, SH, SW, ADDI, ANDI, ORI, XORI, LUI, SLTI, BEQ, BNE, J, JAL)
    c) **J-Type** (JR, JALR)
3. Debe poseer soporte (detección y manejo) de los siguientes tipos de riesgos:
    a) **Estructurales**: Cuando dos o más instrucciones tratan de utilizar el mismo recurso en el mismo ciclo.
    b) **Datos**: Una etapa desea utilizar un dato antes de que esté listo. Se debe mantener el orden estricto de las lecturas y escrituras.

   c) **Control**: Capacidad para tomar una decición sobre una condicion no evaluada.
   Para esto debe implementar:
   d) **Unidad de detección** riesgos: Detecta los posibles riesgos de cada tipo, y efectua los controles necesarios.
   e) **Unidad de cortocircuitos**: Ayuda en el transpaso de datos de una etapa a otra, de ser requeridos inmediatamente. Elimina la necesidad de que los datos lleguen a la última etapa de escritura de registros para que estos sean usados por otros.
4. **Ensablador:** Debe implementarse un programa ensamblador, en el cual dado una entrada en assembler de MIPS, se obtenga una salida de código máquina, apto para ser ejecutado.
5. **Unidad de Debug:** Administra y monitorea la ejecución del procesador mediante una interfaz UART. Debe ser capaz de cargar un programa en memoria, correrlo en varios modos, y mostrar el estado del procesador.

   Los datos que se deben envíar son:

   a) Contenido de los 32 registros.
   b) Contenido de los latches intermedios.
   c) Program counter.
   d) Contenido de la memoria de datos.
   e) Cantidad de clocks desde el inicio.

   Los modos de operación que debe soportar son:

   a) **Continuo:** Una vez cargado el programa, se envía un comando que comienza la ejecución del mismo. Al terminar, se envía la información del estado del procesador.
   b) **Paso a paso:** Se comienza con la ejecución, pero esta vez, con cada comando, se ejecuta un ciclo del clock, y se envía la informacion del estado del procesador.

   Además de estos requerimientos, los autores, a modo de mejoras, incluyen los siguientes:

6. **Interfaz gráfica WEB para el debugger:** Desde esta interfaz se puede cargar un programa, ejecutarlo e inspeccionar los resultados.
7. **Tests automatizados**: Capacidad de correr una tanda de tests sobre la implementación del MIPS y obtener un resumen de los resultados obtenidos. Esto sirve para corroborar el funcionamiento del procesador, y también que los cambios aplicados no modifiquen el comportamiento correcto comportamiento del mismo. Nótese que estos test no son lo mismo que un *test bench* de Verilog. La diferencia entre ambos es que unos son simulaciones del diseño, y el otro sobre el procesador ya sintetizado.

5. Pipeline de cinco etapas: Descripción general

En la (Figura 1) se puede apreciar un diagrama simplificado y general del sistema que compone a un procesador segmentado en cinco etapas. En las siguientes subsecciones se explicará cada una de ellas. Más adelante se agregará complejidad a este diagrama, dado los problemas que irán surgiendo y su solución.

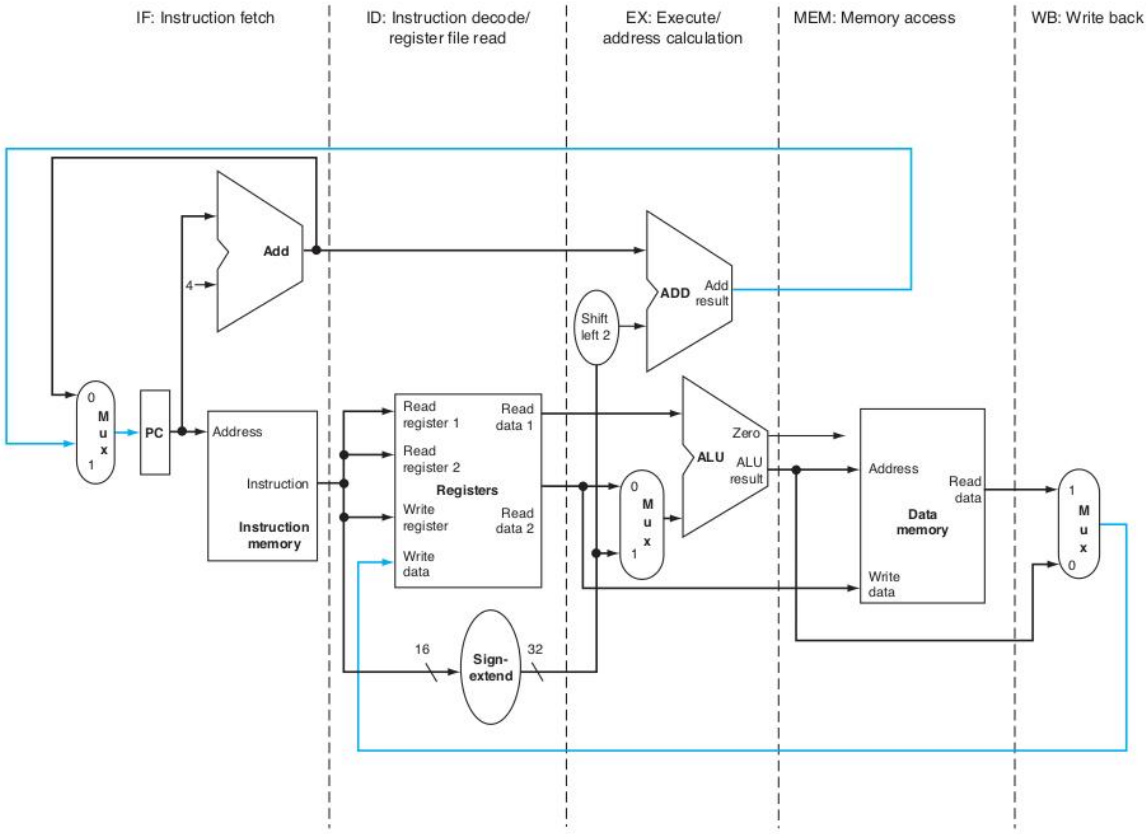

Figura 1. Pipeline de cinco etapas: Diagrama [David A. Patterson 2009]

5.1. *IF o Instruction Fetch*

Es la primer etapa del pipeline. Se busca la siguiente instrucción a ejecutar en la memoria de programa. En ésta etapa se ubica el contador de programa o *Program Counter*, mejor conocido como PC, el cual es un registro que contiene la dirección actual en la memoria de programa en la que se encuentra el programa en ejecución. La siguiente instrucción a ejecutar antes del ciclo es PC + WORD_SIZE (4 para un procesador de 32bits), en caso que no haya un salto. Para mejor claridad, los pasos que ocurren en el ciclo son:

1. Se calcula el nuevo valor de PC. Para eso existe un multiplexor a la entrada del PC que toma el valor desde dos fuentes. La primera es el módulo sumador, que simplemente devuelve la siguiente instrucción, y la otra fuente es una dirección arbitraria que viene desde la etapa de ejecución, producto de una instrucción de salto, condicional o absoluta.
2. Con el valor del PC se accede a la memoria de programa, se obtiene el valor en dicha dirección, y se entrega a la siguiente etapa.

5.2. *ID o Instruction Decode*

Etapa en donde se decodifica la instrucción y se cargan los registros correspondientes. Aqui tiene lugar el File Register, el módulo encargado de contener los 32 registros de 32bits del MIPS. Cada instrucción se transforma en sus correspondientes señales en la Unidad de Control, el cual comanda todo el procesador, según que instrucción se esté ejecutando.

5.3. *EX o Excecute*

Unidad que ejecuta la instrucción, osea, que efectua el cálculo aritmético, lógico, o de resolución de dirección de offset. Vemos que una entrada de la ALU de propósito general puede ser un valor inmediato, con el signo extendido, o un registro, mientras que la otra entrada es siempre un registro. Existe otra ALU que unicamente suma, la cual cálcula nuevas direcciones para el PC, el cual le suma a este un valor inmediato, el cual corresponde al salto a efectuar.

5.4. *MEM o Memory Access*

Unidad de lectura o escritura desde/hacia la memoria de datos. En ésta etapa se lee o escribe un dato de la memoria de acuerdo a la instrucción ejecutada. Si la instrucción no es de acceso a memoria, se hace un bypass de ésta etapa hacia la siguiente.

5.5. *WB o Write back*

Escritura de resultados en los registros. Los registros pueden ser escritos desde un dato en memoria, si la instrucción era de acceso a memoria, o desde la ALU si no lo eran. Se llama Write Back porque escriben sobre una etapa anterior del pipeline.

6. Pipeline de cinco etapas: Problema completo y solución

El primer pipeline presentado, es simple, funciona y expone correctamente la teória básica de la segmentación en el diseño de un procesador, lo cual es necesario para el marco de este trabajo, pero presenta problemas muy grandes, que de no ser solucionados, prácticamente elimina las ventajas del pipeline debido a su ineficiencia a la hora de enfrentar ciertos casos comunes.

El presente trabajo no planea ser una explicación teórica de todas las técnicas utilizadas y los problemas resueltos, por eso se irá al grano de la implementación completa, pero dejando al lector la posibilidad de consultar la bibliografía en el caso de ser necesario. Sin embargo, se explicará brevemente el motivo de cada uno de los elementos incluidos en el mismo y luego se discutirá la implementación desarrollada. No se incluirán en el informe todos los detalles implementativos, dado que estos pueden consultarse en el código del proyecto anexo, pero se mostraran ejemplos variados y detalles a tener en cuenta.

Un esquema aproximado de la implementación completa del pipeline es el de la (Figura 2).

Podemos ver como se agregan diferentes unidades y complejidad que resuelven problemas determinados. Los mas significativos son los latches intermedios *etapa-1/etapa*, la unidad de detección de riesgos o *hazard detection unit*, la unidad de control, la

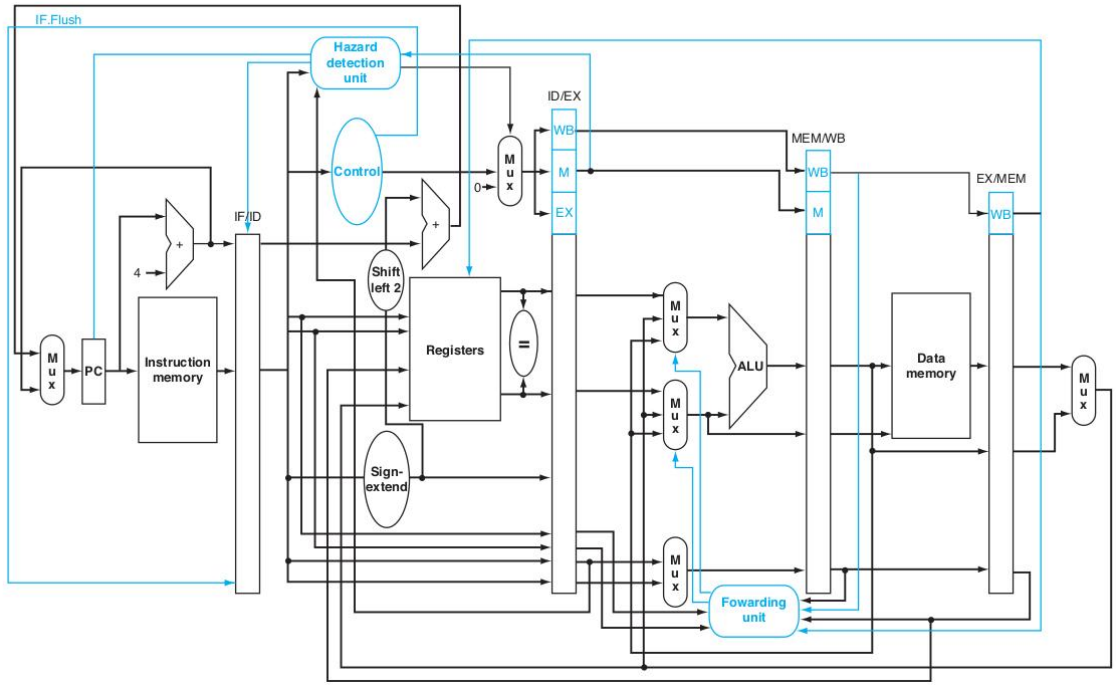

Figura 2. Pipeline de cinco etapas completo [David A. Patterson 2009]

unidad de forwarding, y en nuestro caso, la unidad de saltos o *branching unit*. A continuación se hablara de cada una de esas unidades, explicando brevemente el funcionamiento de las mismas.

6.1. Latches intermedios

Las etapas deben pasarse datos unas a las otras, y mas de una vez, una etapa debe obtener datos de mas de una etapa anterior a ella misma. Si una etapa deseara hacer esto, sin interbloqueo, le resultaría imposible, dado que el estado de una etapa anterior ya ha cambiado con respecto a la instrucción que le corresponde. Por este motivo se incluyen registros o latches intermedios entre cada etapa, que almacenan los datos necesarios para el flujo del pipeline. Sirven de buffer general a las señales que se deseen conservar.

Estos latches se nombran de acuerdo a las etapas que unen. Por ejemplo, el registro que une la etapa IF con la etapa ID se llama IF/ID. Veamos la implementación del *latch* IF/ID:

IF_ID.v

```
1  module IF_ID
2  #(
3      parameter PC_BITS          = `PC_BITS,
4      parameter INSTRUCTION_BITS = `INSTRUCTION_BITS
5  )
6  (
7      input clk,
8      input rst,
9      input enable,
10     input flush,
11     input                                  i_if_id_write,
```

```
    input wire [PC_BITS - 1 : 0]            i_PCNext,
    input wire [INSTRUCTION_BITS - 1 : 0]   i_instruction,
    output reg [PC_BITS - 1 : 0]            o_PCNext,
    output reg [INSTRUCTION_BITS - 1 : 0]   o_instruction
);

    always@(posedge clk) begin
        if (~rst) begin
            o_PCNext <= 0;
            o_instruction <= 0;
        end
        else begin
            if(enable) begin
                if (flush) begin
                    o_PCNext <= 0;
                    o_instruction <= 0;
                end
                else if (i_if_id_write) begin
                    o_PCNext <= i_PCNext;
                    o_instruction <= i_instruction;
                end
            end
        end
    end

endmodule
```

Todos los *latches* simplemente pone a su salida los registros de entrada luego de un ciclo de clock, si esta el *write enable* activado. Tambien tienen capacidad de flush

6.2. Hazard detection unit

Para explicar qué hace esta unidad tenemos que analizar algunos conceptos teóricos. El más importante es el de *hazard*, o riesgo. Un riesgo ocurre cuando una instrucción no se puede ejecutar inmediatamente despues que otra, debido a dependencias no resueltas que existen entre ambas. Existen varios tipos de riesgos, los cuales se explicaran a continuacion:

6.2.1. Tipos de riesgos

1. **Riesgo estructural:** Este tipo de riesgo ocurre cuando el hardware no puede ejecutar la combinacion de instrucciones que tenemos en el pipeline en un ciclo de reloj. Por ejemplo en el caso que en dos etapas diferentes se utilice un mismo recurso de hardware. Estos casos no se presentan a menudo dado que la idea primordial del MIPS es que sea segmentado y se eviten los riesgos estructurales.
2. **Riesgo de datos:** Ocurren cuando el pipeline debe ser detenido porque una etapa debe esperar a que otra etapa termine.

Por ejemplo:

```
    sub   $2, $1,$3
    and   $12,$2,$5
```

La instruccion *sub* no escribe su resultado hasta la quinta etapa (WB), entonces se deberia detener el pipeline durante tres ciclos hasta que termine, para luego ejecutar la *and*, dañando severamente la performance. El compilador podria mitigar mucho este problema, pero ocurre demasiado a menudo y no se puede confiar en que lo resuelva siempre. Existen dos maneras de solucionar este problema, haciendo una parada del pipeline (poco eficiente) o implementando una *forwarding unit*. Ésta es la técnica utilizada en este trabajo, de la cual se hablara más adelante.

3. **Riesgo de control:** Se basa en la necesidad de tomar una desicion sin conocer el resultado de una instruccion que esta ejecutandose. Estas instrucciones, mas especificamente, son las de branching o de salto. Cuando entra una instruccion de salto al pipeline, dependiendo de si ese salto es tomado o no, la etapa IF debe cargar nuevas instrucciones de un lado de la memoria u otro. Para resolver este problema se usan predictores de salto, que tratan de adivinar cual sera el resultado del salto. En el caso de adivinar correctamente, la performance no se ve afectada, si no es asi, deben desecharse las instrucciones que habia en el pipeline hasta ese momento, y comenzar a cargar las nuevas. Los tipos de predictores no seran explicados aqui, salvo el elegido por los autores, el cual es llamado *branch not taken*. este tipo de predictor siempre supone que el branch no es tomado. Es menos eficiente que soluciones más complejas, pero es simple de implementar y en bajo en consumo de recursos de hardware.

En vista de los tipos de riesgos que hay, y sus necesidades, nuestra *hazard detection unit* se ubicará en la etapa de ID, y solo contendrá lógica para la detección y salida de control de los riesgos de datos:

HazardDetector.v

```
module HazardDetector
#(
    parameter INSTRUCTION_BITS = `INSTRUCTION_BITS,
    parameter REG_ADDRS_BITS   = `REG_ADDRS_BITS
)
(
    input wire [REG_ADDRS_BITS - 1 : 0] i_instruction_rs,
    input wire [REG_ADDRS_BITS - 1 : 0] i_instruction_rt,
    input wire [REG_ADDRS_BITS - 1 : 0] i_id_ex_rt,
    input wire                          i_id_ex_MemRead,
    output reg                          o_PCWrite,
    output reg                          o_if_id_write,
    output reg                          o_control_mux
);

    //control_mux = 0 when we want to set all control lines to 0
       and
    //control mux = 1 when we want all control lines from the
       control unit
    always@* begin
        if  (
```

```
                i_id_ex_MemRead &&
                (
                    (i_id_ex_rt == i_instruction_rs) ||
                    (i_id_ex_rt == i_instruction_rt)
                )
            )
        begin
            o_PCWrite     = 0;
            o_if_id_write = 0;
            o_control_mux = 0;
        end
        else begin
            o_PCWrite     = 1;
            o_if_id_write = 1;
            o_control_mux = 1;
        end
    end
endmodule
```

Se evalúan las situaciones peligrosas:

- Si la instrucción anterior quiere leer (*i_id_ex_MemRead*) y:
  - Si el registro RT de la instruccion anterior (*i_id_ex_rt*) es igual al RS de la actual (*i_instruction_rs*) o...
  - Si el registro RT de la instruccion anterior (*i_id_ex_rt*) es igual al RT de la actual (*i_instruction_rt*).

Si se cumple alguno de esos casos el riesgo fue detectado y se toman las medidas correspondientes.

6.3. Control Unit

Para explicar la Unidad de Control, debemos abordar un enfoque de más pequeño a más grande. Existe una Unidad de Control principal y una Unidad de Control de ALU (no representada en el diagrama general) pero sobre la cual se hablará a continuación. Para tener un mejor entendimiento del tema, es bueno entender como se componen binariamente las instrucciones del set de instrucciones MIPS de 32 bits, según el tipo de instrucción que sean. La (Figura 3) detalla esta estructura. Prestar atencion al campo *op*, *shamt*, y *funct*, dado que son campos de los que no se ha hablado hasta ahora.

6.3.1. ALU Control

Dependiendo del tipo de instrucción a ejecutar, la ALU deberá efectuar una de ciertas operaciones definidas. Éstas operaciones se seleccionan mediante lineas de control de la ALU. El (Cuadro 1) las define, según el standard MIPS.

Que familias de instrucciones mapeen a las mismas operaciones de ALU nos permite generar otro nivel de abstracción para el control de la ALU y las instrucciones a ejecutar. Con una señal de dos bits llamada *ALUOp* podemos mapear a dichas operaciones. La tabla (Cuadro 2) muestra este nuevo concepto. También se intruduce el concepto

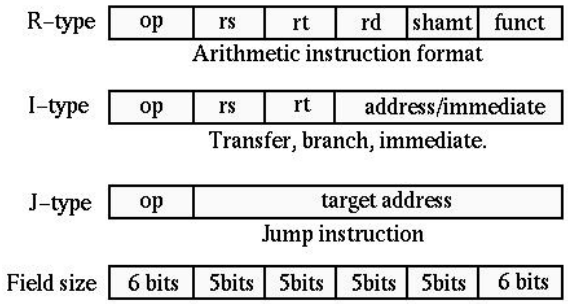

Figura 3. Estructura binaria de cada tipo de instruccion

Cuadro 1. Combinación de entradas de control de ALU y funciones a las que mapean

Control ALU	Función
0000	AND
0001	OR
0010	add
0110	substract
0111	set on less than
1100	NOR

de Funct Field, el cual es un campo de cada instrucción de MIPS el cual, junto con el Opcode, definen la instrucción a ejecutar.

Cuadro 2. Mapeo *ALUOps* a Funct field y entradas de control de ALU

Opcode	ALUOp	Operación	Funct field	Operación ALU	Control ALU
LW	00	load word	XXXXXX	add	0010
SW	00	store word	XXXXXX	add	0010
Branch equal	01	branch equal	XXXXXX	substract	0110
Tipo R	10	add	100000	add	0010
Tipo R	10	substract	100010	substract	0110
Tipo R	10	AND	100100	AND	0000
Tipo R	10	OR	100101	OR	0001
Tipo R	10	set on less than	101010	set on less than	0111

La implementación es extensa debido a las definiciones requeridas, por eso se omiten algunas partes del código:

ALUControl.v

```
module ALUControl
(
    input wire [3 : 0] i_ALUOp,
    input wire [5 : 0] i_funct,

    output reg [3 : 0] o_operation
);
    localparam ALU_ADD  = 'b0000;
    localparam ALU_SUB  = 'b0001;
    // ... resto de AluOP
    localparam FUNCT_ADD  = 'b100000;
    localparam FUNCT_ADDU = 'b100001;
    // ... resto de Functs
    localparam INMED_ADD  = 'b000;
    localparam INMED_AND  = 'b100;
    // ... resto de AluOp

    always@* begin
        if (i_ALUOp == 'b0000) begin
            case (i_funct)
                FUNCT_ADD:  o_operation = ALU_ADD;
                FUNCT_ADDU: o_operation = ALU_ADD;
                // ... resto del mapeo
                default:    o_operation = 0;
            endcase
        end

        else if (i_ALUOp == 'b0001) begin
            o_operation = ALU_ADD;
        end

        else if (i_ALUOp[3] == 1) begin
            case (i_ALUOp[2:0])
                INMED_ADD: o_operation = ALU_ADD;
                INMED_AND: o_operation = ALU_AND;
                // ... resto del mapeo
                default:   o_operation = 0;
            endcase
        end
        else o_operation = 0;
   end
endmodule
```

Se ve como se hace un mapeo, teniendo en cuenta las diferentes variaciones que hay en el set de instrucciones, ALUOps, Functs y controles de la ALU.

6.3.2. Main Control Unit

Ahora que ya definimos los controles de la ALU, y como interpretarlos a partir de los bits de la instruccion, debemos definir como se controlan todas las otras partes del datapath. En la (Figura 4) se observan en celeste, cuales son las principales señales de

control de los diferentes elementos del pipeline.

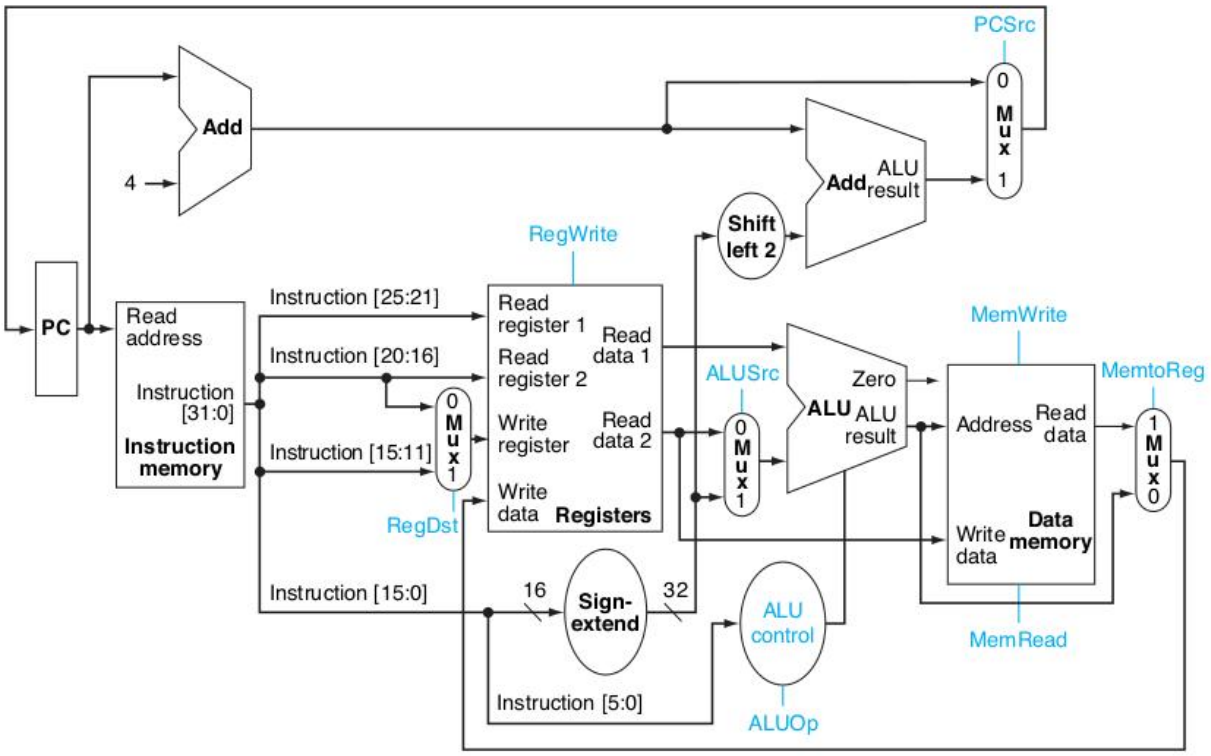

Figura 4. Señales de control del pipeline [David A. Patterson 2009]

Éstas siete señales tienen diferentes efectos, los cuales se explican en el (Cuadro 3).

Habiendo entendido cuales son las entradas y salidas que gobiernan el módulo de control, veremos la implementación del mismo. Tener en cuenta que tambien se producen truncaciones en el código (no sin comentarlas y no sin que ya haya sido explicada como completarlo) para brevedad.

Control.v

```
1  `include "constants.vh"
2
3  module Control
4  #(
5      parameter OPCODE_BITS = `OPCODE_BITS
6  )
7  (
8      input wire [OPCODE_BITS - 1 : 0] i_opcode,
9      input wire [5: 0]                i_funct,
10     input wire                       i_control_mux,
11     output reg                       o_RegDst,
12     output reg                       o_RegWrite,
13     output reg                       o_MemRead,
14     output reg                       o_MemWrite,
15     output reg                       o_MemtoReg,
16     output reg [3:0]                 o_ALUOp,
```

[H]

Cuadro 3. Señales de control de la *Main Control Unit*

Nombre de la señal	Efecto en bajo	Efecto en alto
RegDst	El número del registro de destino viene del campo RT	Idem RD
RegWrite	Nada	El registro en la entrada es escrito con el valor del dato
ALUSrc	El segundo operando de la ALU viene de la segunda salida de los registros	Idem del extensor de signo
PCSrc	El PC es reemplazado con la salida del sumador PC+4	Idem con el objetivo del salto
MemRead	Nada	Se lee la memoria a partir de sus entradas
MemWrite	Nada	Se escribe la memoria a partir de sus entradas
MemToReg	El valor entregado a los datos de escritura del registro vienen de la ALU	Idem memoria de datos

```
17      output reg                      o_ALUSrc,
18      output reg                      o_Shamt,
19      output reg [2:0]                o_ls_filter_op
20  );
21
22      localparam LW       = 'b100;
23      localparam SW       = 'b101;
24      localparam INMED    = 'b001;
25      localparam OTHER    = 'b000;
26
27      always@* begin
28          o_ls_filter_op  = i_opcode[2:0];
29
30          if (i_control_mux) begin
31              case (i_opcode[5:3])
32                  LW: begin
33                      // El primero se expone para claridad
34                      // Los otros casos se colapsan
35                      o_RegDst    = 0;
36                      o_RegWrite  = 1;
37                      o_MemRead   = 1;
38                      o_MemWrite  = 0;
39                      o_MemtoReg  = 1;
40                      o_ALUOp     = 'b0001;
41                      o_ALUSrc    = 1;
42                      o_Shamt     = 0;
43                  end
```

```
44
45                  SW: begin
46                      // Caso de instruccion SW
47                  end
48
49                  INMED: begin
50                      // Caso de instruccion inmediata
51                  end
52
53                  OTHER: begin
54                      if (i_opcode[2]) begin
55                          // Caso de branch
56                      end
57
58                      else if (i_opcode[1]) begin
59                        //Caso de J-JAL
60                      end
61
62                      else if (i_opcode == 0) begin
63                          if (i_funct[5:1] == 'b00100) begin
64                             // Caso de JR-JRAL
65                          end
66
67                          else begin
68                            // Caso de RTYPE
69                          end
70                      end
71                      else begin
72                          // Caso ninguna instruccion. Stall
73                      end
74                  end
75
76                  default: begin
77                      // Caso ninguna instruccion. Stall
78                  end
79              endcase
80          end
81
82          else begin
83              // Caso ninguna instruccion. Stall
84          end
85
86      end
87
88  endmodule
```

6.4. Forwarding Unit

Como se mencionó anteriormente, existen dos maneras de lidiar con los riesgos de datos. Parar el pipeline hasta que se resuelvan las dependencias al escribirse nuevamente en la etapa WB, o implementar una *forwarding unit*. Lo que hace este modulo es detectar si alguna instrucción que se encuentre en el pipeline va a necesitar un dato que este ya esté en EX o WB, y lo envía directamente, sin necesidad de que este haya sido escrito

todavia en un registro. Nótese que existe el caso donde el dato requerido por la instrucción siguiente viene a partir de un load. Los load se procesan en la etapa siguiente a EX, en MEM, lo que pone dos etapas de distancia entre la instrucción que requiere el dato, y el dato en sí. Esto obliga a que haya una parada de un ciclo, hasta que se resuelve el load, y pueda ser forwardeado a la etapa de ejecución.

ForwardingUnit.v

```
module FowardingUnit
#(
    parameter REG_ADDRS_BITS = `REG_ADDRS_BITS
)
(
    input wire i_ex_mem_RegWrite,
    input wire i_mem_wb_RegWrite,
    input wire [REG_ADDRS_BITS - 1 : 0] i_id_ex_rs,
    input wire [REG_ADDRS_BITS - 1 : 0] i_id_ex_rt,
    input wire [REG_ADDRS_BITS - 1 : 0] i_ex_mem_rd,
    input wire [REG_ADDRS_BITS - 1 : 0] i_mem_wb_rd,

    output reg [1:0] o_foward_A,
    output reg [1:0] o_foward_B
);

    reg mem_foward_a;
    reg mem_foward_b;
    reg wb_foward_a;
    reg wb_foward_b;

    always@* begin

        mem_foward_a = i_ex_mem_RegWrite && (i_ex_mem_rd != 0) &&
                       (i_ex_mem_rd == i_id_ex_rs);
        mem_foward_b = i_ex_mem_RegWrite && (i_ex_mem_rd != 0) &&
                       (i_ex_mem_rd == i_id_ex_rt);
        wb_foward_a  = i_mem_wb_RegWrite && (i_mem_wb_rd != 0) &&
                       (i_mem_wb_rd == i_id_ex_rs);
        wb_foward_b  = i_mem_wb_RegWrite && (i_mem_wb_rd != 0) &&
                       (i_mem_wb_rd == i_id_ex_rt);

        if (mem_foward_a)     o_foward_A = 'b10;
        else if (wb_foward_a) o_foward_A = 'b01;
        else                  o_foward_A = 'b00;

        if (mem_foward_b)     o_foward_B = 'b10;
        else if (wb_foward_b) o_foward_B = 'b01;
        else                  o_foward_B = 'b00;
    end
endmodule
```

Como se explicó el módulo analiza las necesidades de las etapas anteriores en busqueda de una dependencia que puede ser cumplida inmediatamente. De ser así, envía ese dato a la entrada de la ALU correspondiente.

6.5. Branching Unit

Esta unidad es una solución propia para resolver el control de saltos. Interpreta los *opcodes* de cada instrucción y controla las señales correspondientes. Esto alivia trabajo de la Unidad de Control. Además contiene la lógica del predictor de saltos. Decide si se va a interpretar como *taken* o *not taken*.

BranchUnit.v

```
1  `include "constants.vh"
2
3  module BranchUnit
4  #(
5      parameter OPCODE_BITS = `OPCODE_BITS,
6      parameter PROC_BITS   = `PROC_BITS,
7      parameter PC_BITS     = `PC_BITS,
8      parameter FUNCT_BITS  = `FUNCT_BITS
9  )
10 (
11     input wire                          i_branch_enable,
12     input wire [OPCODE_BITS - 1 : 0] i_opcode,
13     input wire [FUNCT_BITS - 1 : 0]  i_funct,
14     input wire [PC_BITS - 1 : 0]     i_pc_next,
15     input wire [PROC_BITS - 1 : 0]   i_immediate,
16     input wire [PROC_BITS - 1 : 0]   i_jump_address,
17     input wire [PROC_BITS - 1 : 0]   i_data_rs,
18     input wire [PROC_BITS - 1 : 0]   i_data_rt,
19
20     output reg o_taken,
21     output reg o_pc_to_reg,
22     output reg [PC_BITS - 1 : 0] o_jump_address,
23     output reg [PC_BITS - 1 : 0] o_pc_return,
24     output reg o_pc_reg_sel
25
26 );
27
28     localparam BEQ     = 'b000100;
29     localparam BNE     = 'b000101;
30     localparam J       = 'b000010;
31     localparam JAL     = 'b000011;
32     localparam JR_JALR = 'b000000;
33
34     always@* begin
35         o_pc_return = i_pc_next;
36         if (i_branch_enable) begin
37             case (i_opcode)
38                 BEQ: begin
39                     o_taken = i_data_rs == i_data_rt;
40                     o_pc_to_reg = 0;
41                     o_jump_address = (i_pc_next + i_immediate);
42                     o_pc_reg_sel = 0;
43                 end
44                 BNE: begin
45                     o_taken = i_data_rs != i_data_rt;
46                     o_pc_to_reg = 0;
47                     o_jump_address = (i_pc_next + i_immediate);
48                     o_pc_reg_sel = 0;
```

```
                end
                J: begin
                    o_taken = 1;
                    o_pc_to_reg = 0;
                    o_jump_address = i_jump_address;
                    o_pc_reg_sel = 0;
                end
                JAL: begin
                    o_taken = 1;
                    o_pc_to_reg = 1;
                    o_jump_address = i_jump_address;
                    o_pc_reg_sel = 1;
                end
                JR_JALR: begin
                    //JR
                    if (i_funct == 'b001000) begin
                        o_taken = 1;
                        o_pc_to_reg = 0;
                        o_jump_address = i_data_rs;
                        o_pc_reg_sel = 0;
                    end
                    //JALR
                    else if (i_funct == 'b001001) begin
                        o_taken = 1;
                        o_pc_to_reg = 1;
                        o_jump_address = i_data_rs;
                        o_pc_reg_sel = 0;
                    end
                    //NO JUMP
                    else begin
                        o_taken = 0;
                        o_pc_to_reg = 0;
                        o_jump_address = 0;
                        o_pc_reg_sel = 0;
                    end
                end
                //NO JUMP
                default: begin
                    o_taken = 0;
                    o_pc_to_reg = 0;
                    o_jump_address = 0;
                    o_pc_reg_sel = 0;

                end
            endcase
        end
        else begin
            o_taken = 0;
            o_pc_to_reg = 0;
            o_jump_address = 0;
            o_pc_reg_sel = 0;
        end
    end
```

```
endmodule
```

7. Debug Unit

Como se comento anteriormente, la Debug Unit es fundamental para el analísis y utilización del pipeline sintetizado. Sin este, es un proyecto inerte e inutilizable.

Dicha unidad se conecta, en el módulo principal, a las lineas del Datapath que existen para este propósito. El Datapath debió ser modificado de manera importante para la inclusión de ésta unidad, dado que los datos que se desean extraer de la misma son de variada indole y procedencia.

La Debug Unit esta modelada como una máquina de estados finita, cuyo diagrama se presenta en el (Diagrama 7).

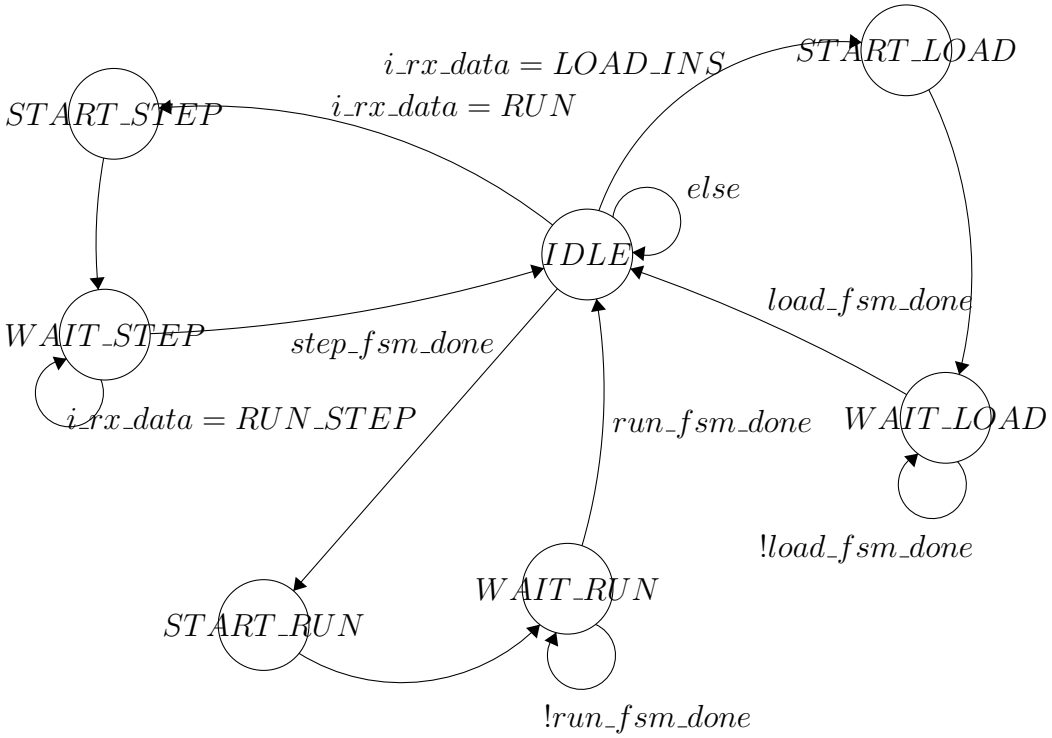

Diagrama de la FSM de la DebugUnit

Esta máquina de estados modela el comportamiento de la unidad, pero luego existen módulos que entran en acción según en que estado se encuentre la misma. Por ejemplo, enviar datos por UART, cargar un programa, ejecutar el programa, son todas tareas comandadas por la máquina de estados general. Para conocer más sobre los detalles de la implementación, consultar el código anexo.

8. Test Benches

Los test benches son una parte fundamental de cada diseño en cualquier HDL. Estos componen el primer paso en la validacion del sistema, y deben ser pasados para continuar a cualquier etapa posterior. Consisten en una representación lógica de cada módulo, a

la cual luego se lo ataca con señales de entrada simuladas, pensadas para contemplar la mayor cantidad de casos posibles. El resultado de la prueba es exitoso si la salída esperada se corresponde con la salída real. La suite de desarrollo que utilizamos cuenta con herramientas de simulación, muy utiles para ésta tarea.

Debe tenerse en cuenta que un test bench exitoso no corresponde a una implementación sintetizada exitosa. Existen problemas de hardware (timing, conexiones, constraints, espacio) que no son contemplados en la simulación.

En el marco de este trabajo se generaron test benches para cada uno de los módulos utilizados. A continuación se muestra un ejemplo de estos tests, quedando a disposición el resto en el código fuente anexo.

tb_HazardDetector.v

```
module tb_HazardDetector();

    reg [4:0] rs;
    reg [4:0] rt;
    reg [4:0] id_ex_rt;
    reg id_ex_MemRead;
    wire PCWrite;
    wire if_id_write;
    wire control_mux;

    initial begin
        //testing MemRead == 0
        id_ex_MemRead = 0;
        rs = 0;
        rt = 0;
        id_ex_rt = 0;

        //testing all posibilities

        //all register diferent
        #20
        id_ex_MemRead = 1;
        rs = 1;
        rt = 2;
        id_ex_rt = 3;

        //rs == id_ex_rt
        #20
        id_ex_MemRead = 1;
        rs = 10;
        rt = 2;
        id_ex_rt = 10;

        //rt == id_ex_rt
        #20
        id_ex_MemRead = 1;
        rs = 10;
        rt = 25;
        id_ex_rt = 25;

        //rs == rt == id_ex_rt
```

```
42          #20
43          id_ex_MemRead = 1;
44          rs = 15;
45          rt = 15;
46          id_ex_rt = 15;
47
48          //rs == rt == id_ex_rt and MemRead == 0
49          #20
50          id_ex_MemRead = 0;
51          rs = 15;
52          rt = 15;
53          id_ex_rt = 15;
54      end
55
56      HazardDetector hazard_detector_u(
57          .i_instruction_rs(rs),
58          .i_instruction_rt(rt),
59          .i_id_ex_rt(id_ex_rt),
60          .i_id_ex_MemRead(id_ex_MemRead),
61          .o_PCWrite(PCWrite),
62          .o_if_id_write(if_id_write),
63          .o_control_mux(control_mux)
64      );
65  endmodule
```

Se ve a simple vista como las entradas del modulo son confeccionadas para probar cada caso de interés. En la (Figura 5) vemos la salída gráfica que nos provee la herramienta de simulación para analizar los resultados obtenidos. También existen formas de incluir asserts para generar tests automatizados de los módulos y no tener que analizar las gráficas todo el tiempo, pero en el módulo mostrado en este informe no se utilizan por conveniencia.

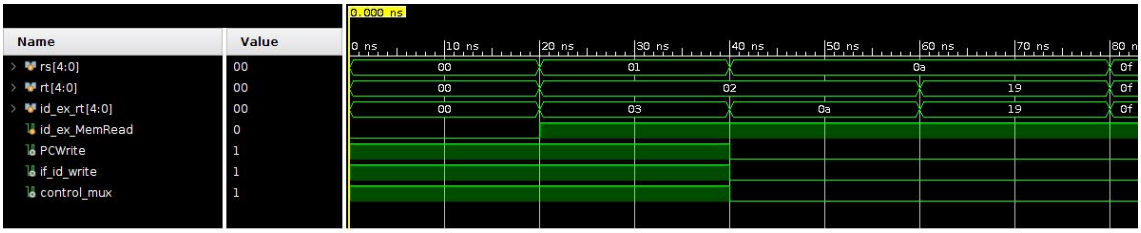

Figura 5. Simulación módulo HazardDetector.v [Xilinx Vivado]

Nótese también, como en éstas simulaciones se pueden realizar cosas imposibles en módulos que planeen ser sintetizados. Por ejemplo, incluir delays de tiempo (con *#xx*) no es sintetizable en hardware.

9. Tests sobre el modelo sintetizado

Una vez sintetizado y cargado en el embebido FPGA, es necesario comprobar el correcto funcionamiento del pipeline. Una buena forma de realizar esto es automatizar la tarea utilizando los datos obtenidos de la Debug Unit mediante UART. Pudiendo cargar

un programa, y obtener resultados, es totalmente posible generar un pequeño programa controlador y comparador. Creando un script en Python y utilizando una librería de test genéricos llamada *pytest* se creó esta utilidad.

Los resultados de la herramienta son de muy simple lectura, como se puede ver en la (Figura 6), lo que facilita la tarea del desarrollador a la hora de aplicar cambios, y comprobar que estos no afecten el comportamiento del procesador.

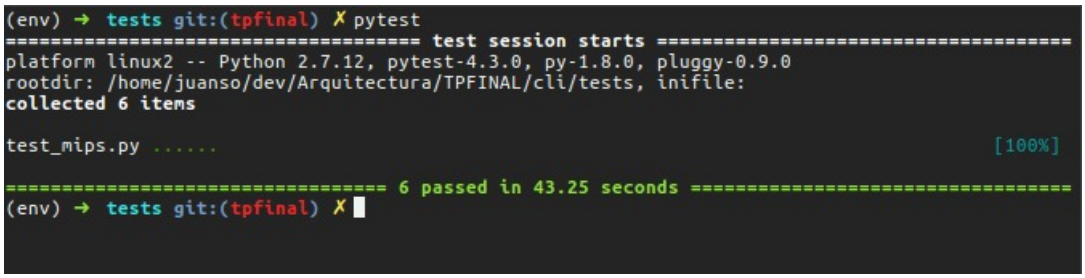

Figura 6. Tests automatizados sobre el hardware sintetizado

10. Ensamblador

Crear programas en código máquina es una tarea muy engorrosa, dado el manejo de bits que esto conlleva. Por ésto es que existen abstracciones que van de menor a mayor en nivel, siendo el lenguaje ensamblador la siguiente en la escala a partir del lenguaje de máquina. El lenguaje ensamblador consiste en la resolución de los símbolos mnemonicos de las instrucciones y los operandos de las mismas a una línea de lenguaje máquina. En la (Figura 7) vemos como luce una sección de código en ensamblador, y como se traduce a lenguaje máquina.

Assembler:

```
addi   $a0,   $a0,   -1
addi   $a1,   $zero,   1
```

Lenguaje Máquina:

```
2084ffff
20050001
```

Figura 7. Paso de ensamblador a lenguaje máquina

Para resolver esta tarea se generó, también con el lenguaje Python, una utilidad ensambladora básica. Hay herramientas existentes muchos más poderosas para ésta tarea, pero para este trabajo se pedía que la misma fuera generada en su marco. La implementación también se encuentra anexa en el código fuente del trabajo.

11. Interfaz Gráfica

Con los elementos presentados hasta el momento ya se cuenta con una implementación totalmente funcional de un pipeline de MIPS para la cual es posible debuggear y desarrollar. Para facilitar este proceso se decidió mejorar la experiencia del usuario en la parte del debugging. En una primer versión, los datos del estado interno se mostraban por terminal. Esta solución parece ineficiente desde el punto de vista del usuario, dado el poco atractivo grafico, y la dificultad de leer texto tan condensado.

Por esto se decidió crear una aplicación WEB, que soporte la misma funcionalidad que la Debug Unit y sirva de interfaz para la misma. Las herramientas utilizadas fueron de nuevo Python, agregando Flask para el desarrollo web. En la (Figura 8) se puede ver el aspecto de dicha interfaz, en pleno proceso de debugging paso a paso sobre un programa.

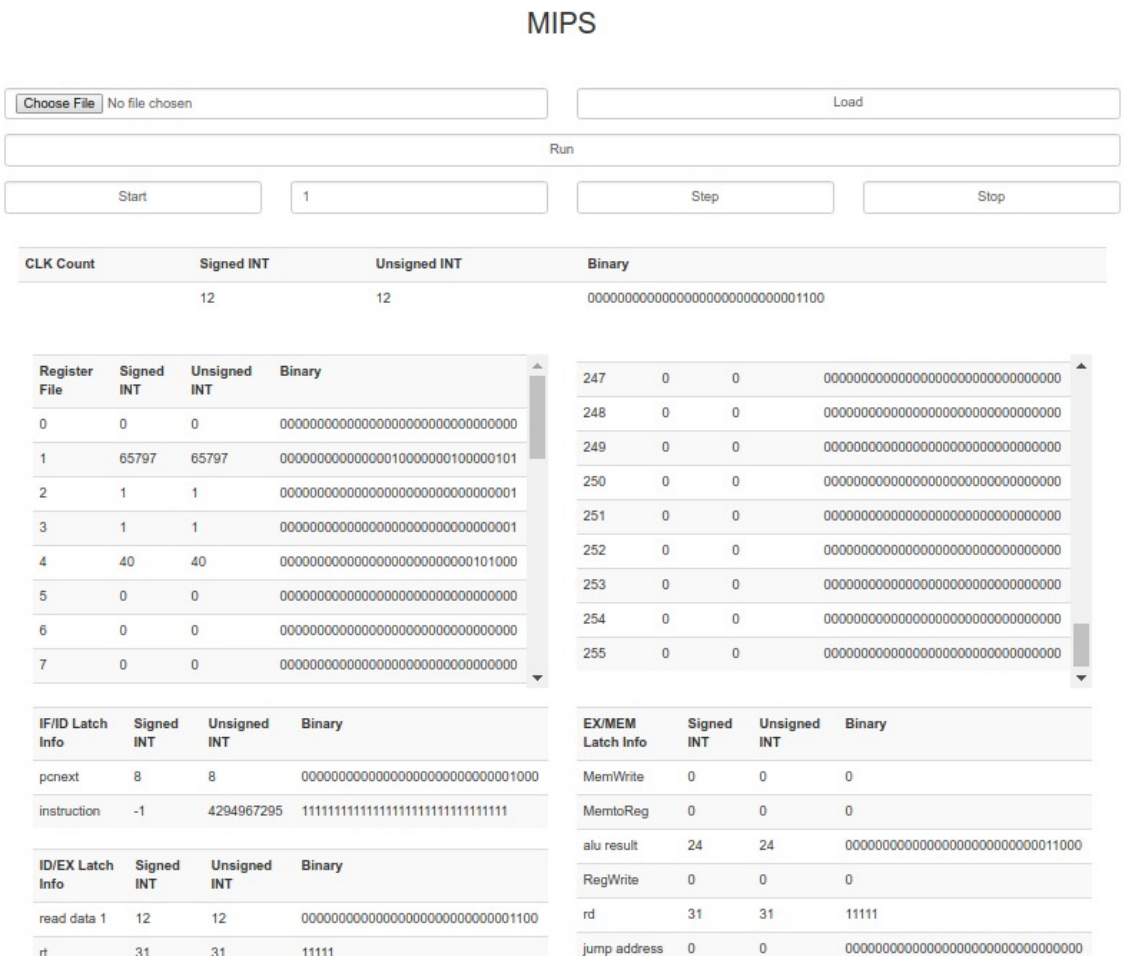

Figura 8. Interfaz gráfica WEB: Debugging

12. Consideraciones finales

Este trabajo reúne los conocimientos básicos de la arquitectura de computadoras y su resolución es muy útil para su aprendizaje. Es un desafío bastante grande encarar un proyecto tan complejo, pero con ayuda de la bibliografía de referencia, y tiempo y dedicación puede lograrse. El mismo asienta las bases de un buen conocimiento en el

area de procesadores de propósito general y de prototipado de hardware, las cuales son fundamentales en la formación de un Ingeniero en Computación.

Referencias

David A. Patterson, J. L. H. (2009). *Computer Organization and Design*. Morgan-Kaufmann, 4th edition.