TRABAJO PRACTICO 3
Arquitectura de computadoras

BIP - Basic Instruction-Set Processor

Integrantes:

- Sosa Ludueña Gabriel
- Cazajous Miguel

2016

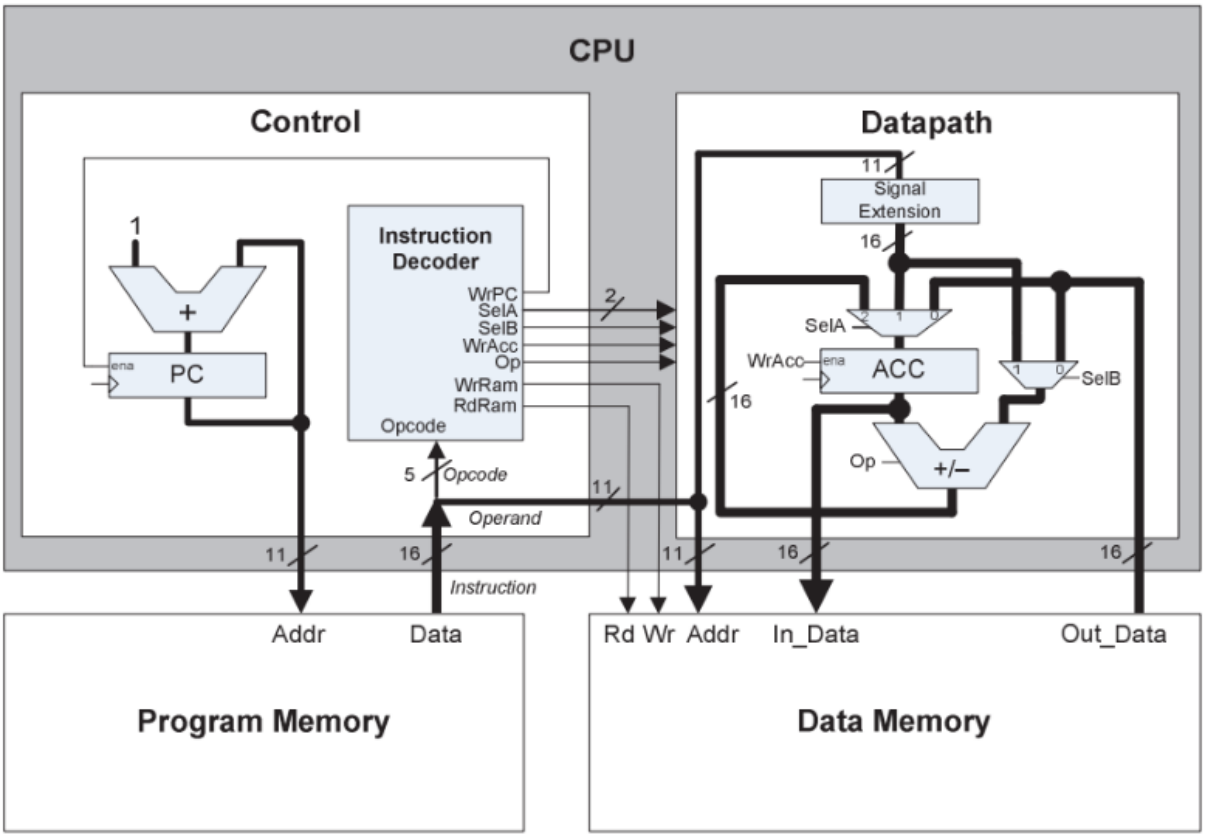
Introducción

En este trabajo se describe una arquitectura de procesador simple que, a primera vista, no tiene diferencias importantes con otros procesadores descritos en otros libros. Sin embargo, se distingue de los otros procesadores porque fue diseñado usando un enfoque interdisciplinario. Este procesador, llamado BIP (Basic Instruction-Set Processor), fue especificado por un equipo de profesores que trabajan con cursos de *Introducción a la Programación*, *Circuitos Digitales*, *Sistemas Digitales*, *Arquitectura y Organización de Computadores* y *Diseño*. Todas estas disciplinas de Ciencias de la Computación se tuvieron en cuenta en el momento de especificar una familia de procesadores con una arquitectura incremental que podría ser utilizada en las primeras semanas de un curso de licenciatura en Ciencias de la Computación. Además, consideramos su uso en cursos avanzados como un primer ejemplo de arquitectura y organización de procesadores, o como una alternativa para la generación de código de ensamblaje[1].

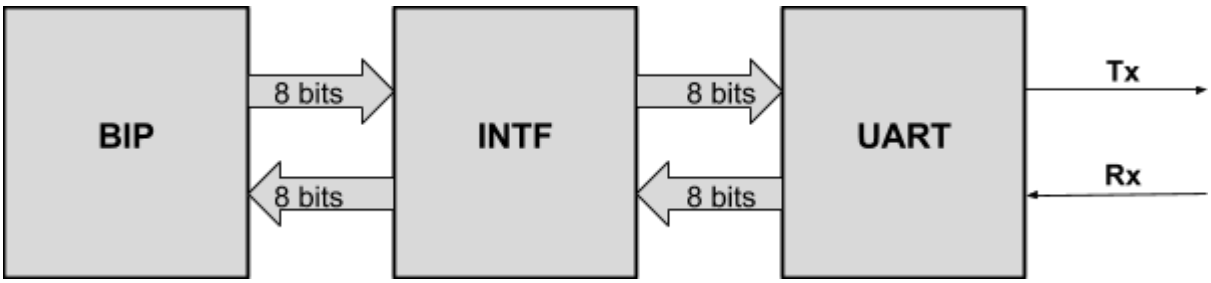
En el dominio de los circuitos y sistemas digitales, el BIP ha demostrado ser un ejemplo didáctico de procesador. Después de aprender los principales conceptos sobre circuitos digitales, tecnologías y metodologías de diseño y herramientas, cualquier estudiante puede aprender fácilmente la arquitectura BIP, diseñar su organización, describirla con cualquier HDL (o con una herramienta de entrada esquemática) y sintetizarla en una FPGA.

Consigna de trabajo

La consigna del presente trabajo consiste en diseñar el código verilog del BIP descrito de manera completa en la referencia [1], verificando y validando su funcionamiento a través de los correspondientes testbench y llevarlo a la práctica a través de una placa FPGA (como una Spartan 3 Basys II en nuestro caso).



Considerando la imagen anterior como el BIP para la validación del funcionamiento del mismo se deberá, de manera práctica, utilizar la UART diseñada en el TP2 de la materia y una interfaz de por medio, para que la placa FPGA transmita a la PC el último valor del registro ACC (registro acumulador) del BIP.



Desarrollo

A continuación detallamos el desarrollo de los códigos verilog realizados para llevar a cabo el diseño del BIP respetando las especificaciones (que responden al porqué se utiliza una arquitectura y organización simple), la arquitectura (todos los atributos arquitectónicos del procesador BIP como ancho de datos, ancho de instrucción, etc), la organización (como se asocia el CPU del BIP a la arquitectura de memoria, E/S, ect) y la programación, todo lo anterior detallado en la referencia [1].

BAU - Basic Aritmetic Unit (o unidad aritmética básica)

La BAU es un simplemente una unidad que suma o resta dos operando como entrada devolviendo el resultado como su salida. En verilog es un módulo combinacional muy similar a la ALU, que se realizó en el TP1 de la materia, pero la BAU solo dispone de dos operaciones aritméticas que le permitirán solo sumar o restar los operandos. El módulo es parametrizable, de manera que se permite variar la cantidad de bits de los operandos A, B que por defecto se establece el valor predeterminado de msb=10 (de manera que cada operando será de 11 bits).
Como consecuencia de la operación resta se incluye en resultado una cláusula signed para mostrar un resultado negativo en caso de que este suceda.

```
1.  module bloque_BAU
2.  #(parameter msb=10)
3.  (A,B,Op,Result);
4.
5.  // Entrada modulo
6.  input wire [msb:0] A;
7.  input wire [msb:0] B;
8.  input wire Op;
9.
10. // Salidas modulo
11. output reg signed [msb:0] Result;
12.
13. always @(*)
14.     case(Op)
15.         1: Result=A+B;//ADD
16.         0: Result=A-B;//SUB
17.         default: Result = 0;
18.     endcase
19. endmodule
```

Program Memory (o memoria de programa)

Para crear la memoria de programa según las especificaciones del BIT, la idea fue crear un módulo en verilog que nos permita tener almacenada un conjunto de instrucciones y que puedan ser accedidas a través de un puntero (dirección de la memoria de programa) mostrando por la salida la instrucción asociada a esa dirección.
Entonces este módulo representa el espacio de memoria destinado para guardar las instrucciones que componen un programa.

La lista de instrucciones del programa que el BIP se encargará de ejecutar se incluye directamente en el código (hard-code) en una lista de registros de 16 bits por registro. Estas instrucciones serán accedidas una a una y enviadas al bloque de *Control* de acuerdo al valor del PC, que es la dirección de la lista de registros que contiene cada instrucción.
Este modulo es de tipo secuencial, pues posee datos de clock de entrada y debe estar en sincronía con los demás bloques involucrados en el BIP.

```
1.  module program_memory
2.  (clk, reset, Addr, Data);
3.
4.  // Entradas modulo
5.  input wire clk, reset;
6.  input wire [10:0] Addr;
7.
8.  // Salidas modulo
9.  output reg [15:0] Data;
10.
11. // variable interna
12. reg [15:0] lista [9:0]; // lista de 10 registros de 16 bits cada registro (representa la
    memoria)
13.
14. always@(posedge clk, posedge reset)
15. begin
16.    if(reset)
17.    begin
18.       // Carga de valores de programa fijo
19.       lista[0] <= 16'b0001100000001010; // LDI 10
20.       lista[1] <= 16'b0000100000000001; // STO 1
21.       lista[2] <= 16'b0001000000000001; // LD 1
22.       lista[3] <= 16'b0010100000000101; // ADDI 2
23.       lista[4] <= 16'b0000100000000010; // STO 2
24.       lista[5] <= 16'b0001000000000010; // LD 2
25.       lista[6] <= 16'b0000000000000000; // HLT
26.       lista[7] <= 16'b0000000000000000; // HLT
27.       lista[8] <= 16'b0000000000000000; // HLT
28.       lista[9] <= 16'b0000000000000000; // HLT
29.
30.       // Se muestra primer instrucción de las cargadas inicialmente
31.       Data <= lista[0];
32.    end
33.
34.    else
35.       Data <= lista[Addr];
36.
37. end
38.
39.
40. endmodule
```

Data Memory (o memoria de datos)

Al igual que la memoria de programa la memoria de datos se pensó de la misma manera pero con la posibilidad de que esta pueda ser escrita y leída cuando se lo requiriera.
Este espacio de memoria cuenta al igual que la *Program Memory* con una lista de registros de 16 bits por registro y está destinado a almacenar valores constantes de datos. Los mismos son escritos (o almacenados) mediante la instrucción STO o también pueden ser leídos (o

accedidos) mediante la instrucción LD, o usados para ser operandos juntos con el valor que cuenta el registro ACC mediante las instrucciones ADD o SUB.
De acuerdo al *Opcode* que recibe el módulo *Instruction Decoder* es que se modifican las banderas *WrRam* y *RdRam* que ingresan a la *Data Memory*.
Este módulo también es del tipo secuencial ya que es una memoria sincrónica con los demás módulos.

```
1.  module data_memory
2.  (clk, reset, Rd, Wr, Addr, In_data, Out_data);
3.
4.     // Entradas modulo
5.     input wire clk, reset;
6.     input wire Rd, Wr;
7.     input wire [10:0] Addr;
8.     input wire [15:0] In_data;
9.
10.    // Salidas modulo
11.    output reg [15:0] Out_data;
12.
13.    // Variables internas
14.    reg [15:0] data_list [9:0]; // lista de datos (10 registros de 16 bits)
15.    reg [3:0] i;
16.
17.    always@(posedge clk, posedge reset)
18.    begin
19.       if(reset)
20.       begin
21.          for( i = 0 ; i< 9 ; i=i+1)
22.             data_list[i] <= 0;
23.
24.          data_list[9] <= 0;
25.          Out_data <= 0;
26.       end
27.
28.       else
29.       begin
30.          case ({Rd,Wr})
31.             2'b01: // Write
32.             begin
33.                data_list [Addr] <= In_data;
34.                Out_data <= In_data ;
35.             end
36.             2'b10:// Read
37.             begin
38.                Out_data <= data_list [Addr];
39.             end
40.       endcase
41.       end
42.    end
43.
44.
45. endmodule
```

CPU (o unidad de procesamiento central)

De manera abstracta la CPU es el módulo encargado de interaccionar con las memoria de datos y la memoria de programa. Es decir que la CPU a través del PC (Program Counter) selecciona una instrucción de la memoria de programa y según esa instrucción decide si realizar una lectura o escritura sobre la memoria de datos.

Este módulo engloba los módulos *Datapath* y *Control* y se conecta a los bloques de memoria *Data Memory* y *Program Memory* y es del tipo secuencial y combinacional.

```
1.  module CPU
2.  (clk, reset, instruction, DM_in, AddrPM, AddrDM, ACC, Rd_out, Wr_out, Operand, SelA, SelB,
    Op, WrAcc);
3.
4.     // Entradas modulo
5.     input wire clk, reset;
6.     input wire [15:0] instruction;
7.     input wire [15:0] DM_in;
8.
9.     // Salidas modulo
10.    output wire [10:0] AddrPM;
11.    output wire [10:0] AddrDM;
12.    output wire [15:0] ACC;
13.    output wire Rd_out;
14.    output wire Wr_out;
15.
16.    // Variables internas
17.    output wire [10:0] Operand;
18.    output wire WrAcc, SelB, Op;
19.    output wire [1:0] SelA;
20.
21.
22.    // Instanciacion Control
23.    Control C1(
24.       .clock(clk),
25.       .reset(reset),
26.       .Instruction(instruction),
27.       .SelA(SelA),
28.       .SelB(SelB),
29.       .Op(Op),
30.       .WrAcc(WrAcc),
31.       .WrRam(Wr_out),
32.       .RdRam(Rd_out),
33.       .Operand(Operand),
34.       .PC(AddrPM)
35.    );
36.
37.    // Instanciacion Datapath
38.    Datapath D1(
39.       .clk(clk),
40.       .reset(reset),
41.       .Operand_in(Operand),
42.       .DM_in(DM_in),
43.       .SelA_in(SelA),
44.       .SelB_in(SelB),
45.       .WrAcc_in(WrAcc),
46.       .Op_in(Op),
47.       .ACC(ACC),
48.       .Operand_out(AddrDM)
49.    );
50. endmodule
```

Instruction Decoder (o decodificador de instrucción)

Este módulo es un bloque de código puramente combinacional como lo indica la referencia [1] para simplicidad del procesador cuyo objetivo consiste en, como su nombre lo indica, interpretar un valor de 5 bits (Opcode) que recibe del bloque *Control* y modifica una serie de

banderas dependiendo del valor recibido.
La bandera nombrada como WrPC es utilizada por el bloque *Control* mientras que SelA, SelB, WrAcc y Op ingresan al *Datapath*. Las dos banderas restantes (WrRam y RdRam) se conectan con el bloque *Data memory*.

```
1.  module Instruction_decoder
2.  (
3.  output reg WrPC,
4.  output reg [1:0] SelA,
5.  output reg SelB,
6.  output reg WrAcc,
7.  output reg Op,
8.  output reg WrRam,
9.  output reg RdRam,
10. input wire [4:0]Opcode
11. );
12.
13. always@*
14. case(Opcode)
15.   //HALT: WrPC=0,WrRam=0,RdRam=0
16.   5'b00000:
17.   begin
18.           WrPC=0;
19.           //SelA=2'b00;
20.           //SelB=0;
21.           WrAcc=0; // 0
22.           Op=0;
23.           WrRam=0;
24.           RdRam=0;
25.   end
26.   //STO: WrPC=1,WrRam=1,RdRam=0,
27.   5'b00001:
28.   begin
29.           WrPC=1;
30.           SelA=2'b00;
31.           SelB=1'b1;
32.           WrAcc=0; // 0
33.           Op=0;
34.           WrRam=1;
35.           RdRam=0;
36.   end
37.   //LD: WrPC=1,WrRam=0,RdRam=1,SelA=2'b00,WrAcc=1
38.   5'b00010:
39.   begin
40.           WrPC=1;
41.           SelA=2'b00;
42.           SelB=1;
43.           WrAcc=1;
44.           RdRam=1;
45.           Op=0;
46.           WrRam=0;
47.   end
48.   //LDI: WrPC=1, WrRam=0, RdRam=0,WrAcc=1,SelA=2'b01
49.   5'b00011:
50.   begin
51.           WrPC=1;
52.           SelA=2'b01;
53.           SelB=0;
54.           WrAcc=1;
55.           Op=0;
56.           WrRam=0;
57.           RdRam=0;
58.   end
```

```
59.   //ADD: WrPC=1,WrRam=0,RdRam=1,SelA=2'b10,SelB=0,WrAcc=1,Op=1
60.   5'b00100:
61.   begin
62.           WrPC=1;
63.           SelA=2'b10;
64.           SelB=0;
65.           WrAcc=1;
66.           Op=1;
67.           WrRam=0;
68.           RdRam=1;
69.   end
70.   //ADDI: WrPC=1,WrRam=0,RdRam=0,SelA=2'b10,SelB=1,Op=1,WrAcc=1
71.   5'b00101:
72.   begin
73.           WrPC=1;
74.           RdRam=0;
75.           SelB=1;
76.           Op=1;
77.           SelA=2'b10;
78.           WrAcc=1;
79.           WrRam=0;
80.   end
81.   //SUB: WrPC=1,SelA=2'b10,SelB=0,WrAcc=1,Op=0,WrRam=0,RdRam=1
82.   5'b00110:
83.   begin
84.           WrPC=1;
85.           SelA=2'b10;
86.           SelB=0;
87.           WrAcc=1;
88.           Op=0;
89.           WrRam=0;
90.           RdRam=1;
91.   end
92.   //SUBI: WrPC=1,WrRam=0,RdRam=0,SelA=2'b10,SelB=1,WrAcc=1,Op=0
93.   5'b00111:
94.   begin
95.           WrPC=1;
96.           SelA=2'b10;
97.           SelB=1;
98.           WrAcc=1;
99.           Op=0;
100.              WrRam=0;
101.              RdRam=0;
102.      end
103.      default:
104.      begin
105.              WrPC=0;
106.              SelA=2'b11;
107.              SelB=0;
108.              WrAcc=0; // 0
109.              Op=0;
110.              WrRam=0;
111.              RdRam=0;
112.      end
113.      endcase
114.  endmodule
```

Control

De manera general el módulo *Control* es la parte del CPU que se encarga de ir incrementando en uno el PC por cada clock del BIP en general para ir ejecutando una instrucción por cada ciclo relog (clock). En este bloque se utiliza la bandera WrPC recibida del

Instruction Decoder para modificar o no el registro interno PC (Program Counter), para ello el *Instruction Decoder* recibe de *Control* el valor *Opcode* de 5 bits.
Este bloque es de tipo secuencial, ya que manipula el registro PC (Program Counter), pero también combinacional, debido a que hace uso de un bloque *BAU* y un bloque *Intruction Decoder*. Cuando el valor de entrada *reset* es puesto en uno, el valor del PC (registro de 11 bits) es inicializado en cero, *OpPC* en uno y *Cte* (registro 11 bits) con uno, estas asignaciones se conectan con BAU, teniendo para A el registro PC, para B un valor constante de uno y la operación de suma. El resultado de la operación se guarda en una variable interna *Result* que es asignada a PC si el valor de WrPC está en uno siguiendo una lógica *siguiente estado*, luego este valor del PC es utilizado por el bloque *Program Memory*.

```
1.  module Control
2.  (
3.     input wire clock,
4.     input wire reset,
5.     input wire [15:0] Instruction,
6.
7.     output wire [1:0] SelA,
8.     output wire SelB,
9.     output wire WrAcc,
10.    output wire Op,
11.    output wire WrRam,
12.    output wire RdRam,
13.    output reg [10:0] Operand,
14.    output reg [10:0] PC
15. );
16.
17.
18. wire [10:0] Result;
19. wire WrPC;
20.
21. reg [4:0] Opcode;
22. reg [10:0] PC_next;
23. reg OpPC;
24. reg [10:0] Cte;
25.
26. always@*
27. begin
28.    PC_next=PC;
29.    if(WrPC)
30.       PC_next=Result;
31. end
32.
33. always@(posedge clock,posedge reset)
34. begin
35.       if(reset)
36.       begin
37.          PC<=11'b00000000000;
38.          OpPC<=1;
39.          Cte<=11'b00000000001;
40.       end
41.       else
42.       begin
43.          PC<=PC_next;
44.          Opcode<=Instruction[15:11];
45.          Operand<=Instruction[10:0];
46.       end
47. end
48.
49. Instruction_decoder ID1
```

```
(
        .WrPC(WrPC),
        .SelA(SelA),
        .SelB(SelB),
        .WrAcc(WrAcc),
        .Op(Op),
        .WrRam(WrRam),
        .RdRam(RdRam),
        .Opcode(Opcode)
);

bloque_BAU BAU1
(
        .A(PC),
        .B(Cte),
        .Op(OpPC),
        .Result(Result)
);

endmodule
```

Datapath (o ruta de datos)

Este módulo es la otra parte fundamental de la CPU que consiste en llevar a cabo las operaciones relacionadas con los datos almacenados en la memoria de datos y el registro ACC (registro acumulador) o con este último y algún operando (o dato) literal. Todo esto según qué instrucción se esté ejecutando. Luego que el bloque *Control* separa la instrucción obtenida desde *Program Memory*, se envía desde *Control* a *Datapath* los 11 primeros bits de la instrucción correspondientes al operando, ese valor es usado para acceder a la *Data Memory* para instrucciones LD, ADD, y/o SUB, o como valor inmediato para ser utilizado en instrucciones LDI, ADDI, y/o SUBI.
El valor inmediato antes de ser operado es convertido. Los valores de las banderas SelA y SelB proporcionan información al *Datapath* para conocer el flujo de los datos correcto que debe realizar para cada instrucción. De igual manera con los valores ACC, el cual es un registro interno y un valor inmediato u obtenido de *Data Memory* y Op se le indicará a BAU la operación que debe realizar. La bandera WrAcc indica en qué casos se guarda un valor resultado en ACC, siguiendo la lógica *siguiente estado* utilizada en el bloque *Control* para actualizar el valor PC.
El bloque es de tipo secuencial, por el registro ACC (), y también combinacional por el bloque BAU y por los multiplexores.

```
module Datapath
(clk, reset, Operand_in, DM_in, SelA_in, SelB_in, WrAcc_in, Op_in, ACC, Operand_out);

   // Entradas modulo
   input wire clk, reset;
   input wire [10:0] Operand_in;
   input wire [15:0] DM_in;
   input wire [1:0] SelA_in;
   input wire SelB_in;
   input wire WrAcc_in;
   input wire Op_in;

```

```
13.
14.    // Salidas modulo
15.    output reg [15:0] ACC; // registrio ACC
16.    output wire [10:0] Operand_out;
17.
18.    //Parametro local
19.    localparam msb_BAU = 15;
20.
21.    // Variables internas
22.    wire [15:0] signal_ext; // Signal Extension
23.    wire [15:0] salida_mux3;
24.    wire [15:0] salida_mux2;
25.    reg [15:0] ACC_next;
26.    wire [15:0] result_BAU;
27.
28.    assign signal_ext = {5'b00000, Operand_in};
29.
30.
31.    always@(posedge clk, posedge reset)
32.    begin
33.       if(reset)
34.          ACC <= 0;
35.
36.       else
37.          ACC <= ACC_next;
38.    end
39.
40.    // Instaciacion de multiplexor mux3
41.    mux3 m1(
42.       .A(DM_in),
43.       .B(signal_ext),
44.       .C(result_BAU),
45.       .sel(SelA_in),
46.       .salida(salida_mux3)
47.    );
48.
49.    // Instaciacion de multiplexor mux2
50.    mux2 m2(
51.       .A(signal_ext),
52.       .B(DM_in),
53.       .sel(SelB_in),
54.       .salida(salida_mux2)
55.    );
56.
57.
58.    // Instanciacion de bloque BAU
59.    bloque_BAU #(.msb(msb_BAU)) // paso de parametro, 16 bits en este caso
60.    BAU1(
61.       .A(ACC),
62.       .B(salida_mux2),
63.       .Op(Op_in),
64.       .Result(result_BAU)
65.    );
66.
67.    always@(*)
68.    begin
69.       ACC_next = ACC;
70.       if(WrAcc_in)
71.          ACC_next = salida_mux3;
72.    end
73.
74.    // Salidas
75.    assign Operand_out = Operand_in;
76.
77. endmodule
```

BIP - Bassic Instruction-Set Processor

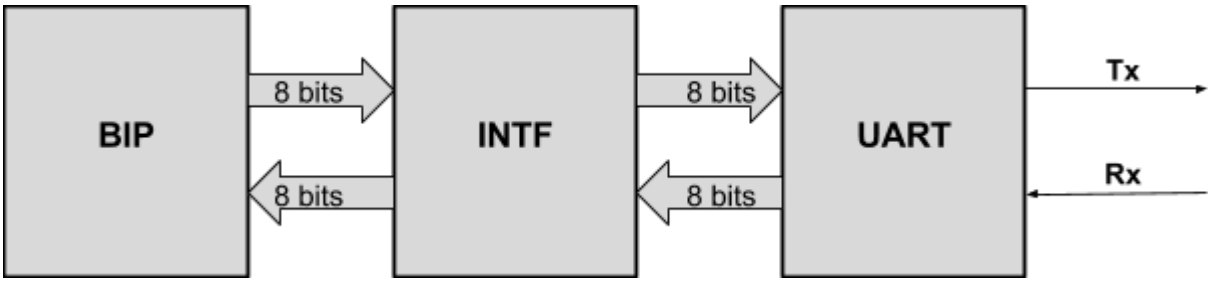
Este módulo está formado por el bloque CPU, la memoria de dato y la memoria de programa. Estos bloques en conjunto conforman la organización del BIP como se detalla en la referencia [1]. El módulo en verilog consiste simplemente en instanciar cada uno de los bloques y generar las relaciones a través de datos *wire* de manera adecuada asociar unos con otros como corresponde.

```
1.  module BIP
2.  (clk, reset, ACC, instruction, PC);
3.
4.      // Entradas modulo
5.      input clk, reset;
6.
7.      // Salidas modulo
8.      //output wire [15:0] result;
9.
10.     // Variables internas
11.     wire Rd, Wr;
12.     output wire [15:0] instruction;
13.     wire [10:0] AddrPM, AddrDM;
14.     output wire [15:0] ACC;
15.     wire [15:0] Out_data;
16.     output wire [10:0] PC;
17.
18.
19.     // Instanciacion de PM
20.     program_memory PM(
21.         .clk(clk),
22.         .reset(reset),
23.         .Addr(AddrPM),
24.         .Data(instruction)
25.     );
26.
27.     // Instanciacion de DM
28.     data_memory2 DM(
29.         .clk(clk),
30.         .reset(reset),
31.         .Rd(Rd),
32.         .Wr(Wr),
33.         .Addr(AddrDM),
34.         .In_data(ACC),
35.         .Out_data(Out_data)
36.     );
37.
38.     // Instanciacion de CPU
39.     CPU CPU1(
40.         .clk(clk),
41.         .reset(reset),
42.         .instruction(instruction),
43.         .DM_in(Out_data),
44.         .AddrPM(AddrPM),
45.         .AddrDM(AddrDM),
46.         .ACC(ACC),
47.         .Rd_out(Rd),
48.         .Wr_out(Wr)
```

```
49.    );
50.
51.    assign PC = AddrPM;
52. endmodule
```

Bloque BIP - INTF - UART

Por último como se realizó en el TP2 de la materia, generamos un bloque que instancia los tres bloques de la imagen al mismo nivel para poder comunicar de esta forma una PC con la placa FPGA utilizando este módulo para darle acceso al procesador BIP. Para dicho requerimiento se debió crear un módulo interfaz (INTF) muy similar al del TP2 que consiste en una máquina de estado a la espera de comandos para cumplir con ciertas funcionalidades. En este caso la idea es introducir un comando para que se indique el inicio de la ejecución de instrucciones en el procesador BIP y al finalizar este de ejecutar todas sus instrucciones muestre cual es el último valor del registro ACC trasmitiendolo por el UART.



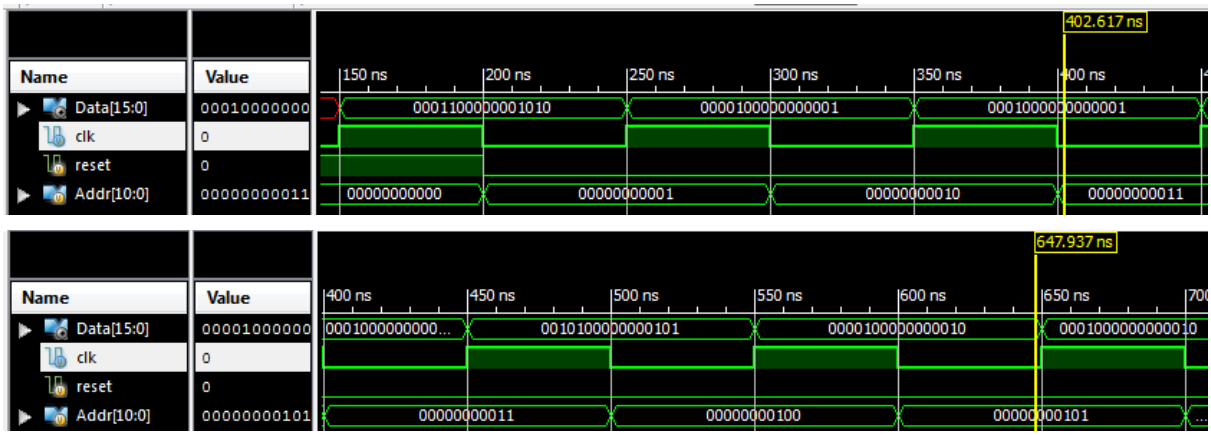
Validación y verificación de funcionalidad

La implementación de los módulos explicados anteriormente se realizó en una placa Spartan 3 - Basys II, complementada de un módulo interfaz CP2102 para lograr funcionalidad UART.
Ante la cantidad de módulos se hizo menester la implementación de test para la gran mayoría de ellos con el fin de asegurarse un funcionamiento óptimo y evitar o disminuir la posibilidad de errores al combinarlos.
A continuación mostramos los resultados de los testbench más importantes que consisten en:

- Verificación de una instrucción por ciclo reloj en la memoria de programa.
- Verificación de sincronismos en las memoria de datos.
- Verificación del funcionamiento deseado en la CPU.

Testbench que verifica una instrucción por ciclo reloj en la memoria de programa

Este testbench consiste en verificar si se cumple que por cada clock del procesador BIP se va cambiando a la instrucción siguiente lo que es equivalente a que en cada ciclo de reloj se ejecutará una instrucción diferente. Ejecutando el testbench desde la herramienta de simulación que brinda el software utilizado (ISE 14.7) se obtuvo el siguiente resultado.



De esta manera se puede observar que por cada nueva dirección de memoria (Addr) varía la instrucción (Data) y también se nota que lo hace por cada clock, es decir que en cada ciclo de reloj nuevo se observa una nueva instrucción.

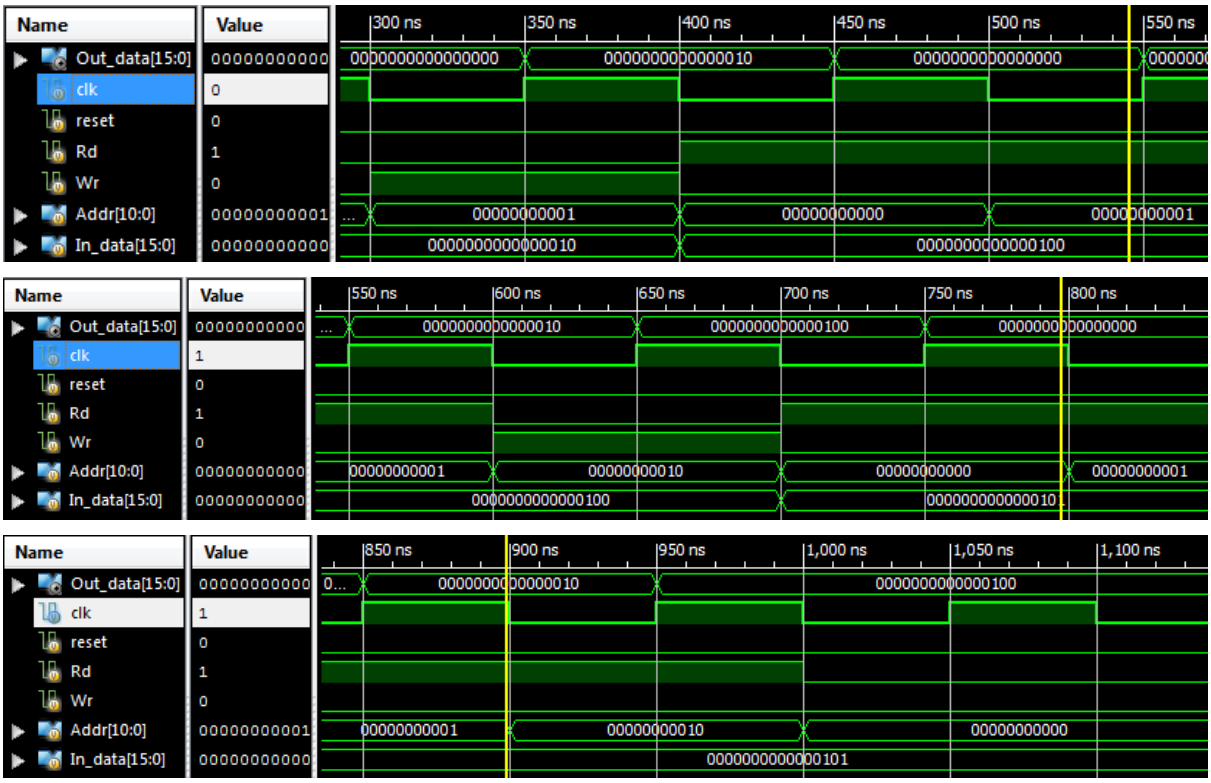
El código del testbench que lleva a cabo la verificación previamente descrita es el siguiente.

```
1.  module test_program_memory;
2.
3.      // Inputs
4.      reg clk;
5.      reg reset;
6.      reg [10:0] Addr;
7.
8.      // Outputs
9.      wire [15:0] Data;
10.
11.     // Instantiate the Unit Under Test (UUT)
```

```
12.    program_memory uut (
13.        .clk(clk),
14.        .reset(reset),
15.        .Addr(Addr),
16.        .Data(Data)
17.    );
18.
19.    initial begin
20.        // Initialize Inputs
21.        reset = 0;
22.        Addr = 0;
23.
24.        // Wait 100 ns for global reset to finish
25.        #100;
26.
27.        // Add stimulus here
28.        reset = 1; // Se recetea Datapath
29.
30.        #100;
31.        reset = 0; // termina reseteo
32.        Addr = 1;
33.
34.        #100;
35.        Addr = 2;
36.
37.        #100;
38.        Addr = 3;
39.
40.        #100;
41.        Addr = 4;
42.
43.        #100;
44.        Addr = 5;
45.
46.        #100;
47.        Addr = 6;
48.
49.    end
50.
51.    // Generacion de clock en circuito secuencial
52.    always begin
53.        clk = 1'b0;
54.        #(100/2) clk = 1'b1;
55.        #(100/2);
56.
57.    end
58.
59. endmodule
```

Testbench que verifica sincronismo en la memoria de datos

Este testbench consiste en verificar si se cumple que por cada clock del procesador BIP se van obteniendo las salidas correctas según las indicaciones en las entradas de la memoria de datos. Con la misma herramienta del test anterior se obtuvo el siguiente resultado.



Como podemos observar se verifica que según las indicaciones de lectura, escritura y el dato de entrada la memoria almacena (Wr- write) la entrada (In_data) o muestra (Rd - read) la salida (Out_data) en la dirección indicada (Addr) y también se verifica que esto lo realiza de manera sincrónica es decir que en cada ciclo reloj tiene la capacidad de llevar a cabo cualquiera de las operaciones de lectura o escritura.

En el ejemplo simulado la idea fue que por cada ciclo de reloj en primer instancia se almacene una entrada sobre In_data (el valor de 2 - b10 como primer caso) en la dirección 1, luego, leer lo que hay en la dirección cero (que como no se almacena nada previamente tiene su valor por defecto que es cero), recien despues verificamos si el valor en la dirección 1 es el previamente cargado (que debería ser el 2) viendo la simulación se cumple de manera correcta y respetando en cada uno de los casos el sincronismo ya que en cada indicación (Rd - Wr) varía la salida por cada ciclo reloj como debería ser. Por último se almacena un nuevo valor (en este caso 4 - b100) sobre la dirección 2, y luego se pide ver el valor de la dirección 0, en el próximo ciclo de la dirección 1 y por último, en un nuevo ciclo, de la dirección 2 para ver si en cada ciclo reloj se verifica el sincronismo y la consistencia de los datos. Observando la simulación queda verificado de manera correcta todas las situaciones.

El código del testbench que lleva a cabo la verificación previamente descrita es el siguiente.

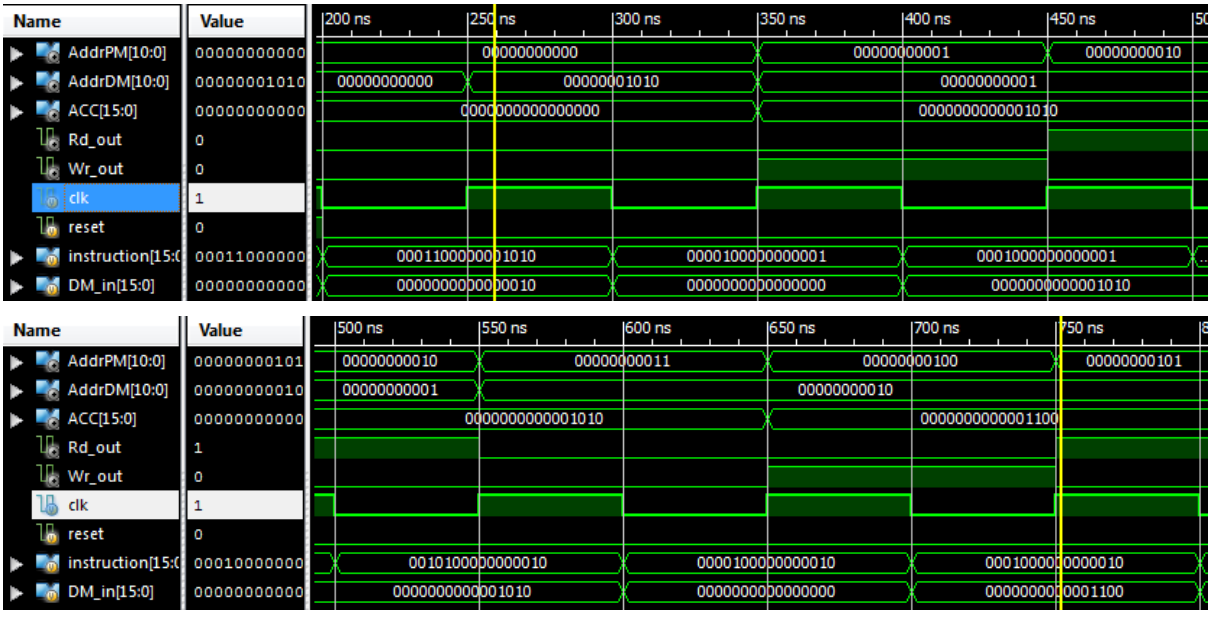
```
1.  module test_data_memory;
2.
3.     // Inputs
4.     reg clk;
5.     reg reset;
6.     reg Rd;
7.     reg Wr;
8.     reg [10:0] Addr;
9.     reg [15:0] In_data;
10.
```

```
11.   // Outputs
12.   wire [15:0] Out_data;
13.
14.   // Instantiate the Unit Under Test (UUT)
15.   data_memory2 uut (
16.       .clk(clk),
17.       .reset(reset),
18.       .Rd(Rd),
19.       .Wr(Wr),
20.       .Addr(Addr),
21.       .In_data(In_data),
22.       .Out_data(Out_data)
23.   );
24.
25.   initial begin
26.       // Initialize Inputs
27.       reset = 0;
28.       Rd = 0;
29.       Wr = 0;
30.       Addr = 0;
31.       In_data = 0;
32.
33.       // Wait 100 ns for global reset to finish
34.       #100;
35.
36.       // Add stimulus here
37.       // Add stimulus here
38.       reset = 1; // Se recetea Datapath
39.
40.       #100;
41.       reset = 0; // termina reseteo
42.       Rd = 0;
43.       Wr = 0;
44.       Addr = 0;
45.       In_data = 2;
46.
47.       #100;
48.       Rd = 0;
49.       Wr = 1;
50.       Addr = 1;
51.       In_data = 2;
52.
53.       #100;
54.       Rd = 1;
55.       Wr = 0;
56.       Addr = 0;
57.       In_data = 4;
58.
59.       #100;
60.       Rd = 1;
61.       Wr = 0;
62.       Addr = 1;
63.       In_data = 4;
64.
65.       #100;
66.       Rd = 0;
67.       Wr = 1;
68.       Addr = 2;
69.       In_data = 4;
70.
71.       #100;
72.       Rd = 1;
73.       Wr = 0;
74.       Addr = 0;
75.       In_data = 5;
```

```
76.
77.      #100;
78.      Rd = 1;
79.      Wr = 0;
80.      Addr = 1;
81.      In_data = 5;
82.
83.      #100;
84.      Rd = 1;
85.      Wr = 0;
86.      Addr = 2;
87.      In_data = 5;
88.
89.      #100;
90.      Rd = 0;
91.      Wr = 0;
92.      Addr = 0;
93.      In_data = 5;
94.
95.    end
96.
97.    // Generacion de clock en circuito secuencial
98.    always begin
99.       clk = 1'b0;
100.          #(100/2) clk = 1'b1;
101.          #(100/2);
102.
103.       end
104.
105.   endmodule
```

Testbench que verifica el funcionamiento correcto de la CPU

Este testbench consiste en verificar si se cumple que la CPU tiene la capacidad de ejecutar las instrucciones por cada ciclo reloj de manera correcta. Para llevar a cabo la verificación el tesbtench consiste en un conjunto de 6 instrucciones que deberá llevar a cabo obteniendo los resultados correctos en 6 ciclos reloj ya que es una instrucción por ciclo. Utilizando la misma herramiento que en los test anteriores se obtiene la siguiente simulación.



Lo que está mostrando la simulación es el conjunto de instrucciones que sigue:

- LDI 10
- STO 1
- LD 1
- ADDI 2
- STO 2
- LD 2
- HLT

De manera que se verifica tanto el resultado obtenido como también el tiempo ya que las 6 instrucciones se llevan a cabo en 6 ciclos reloj considerando la instrucción HALT como la instrucción de fin de programa que detiene el program counter (registro PC).

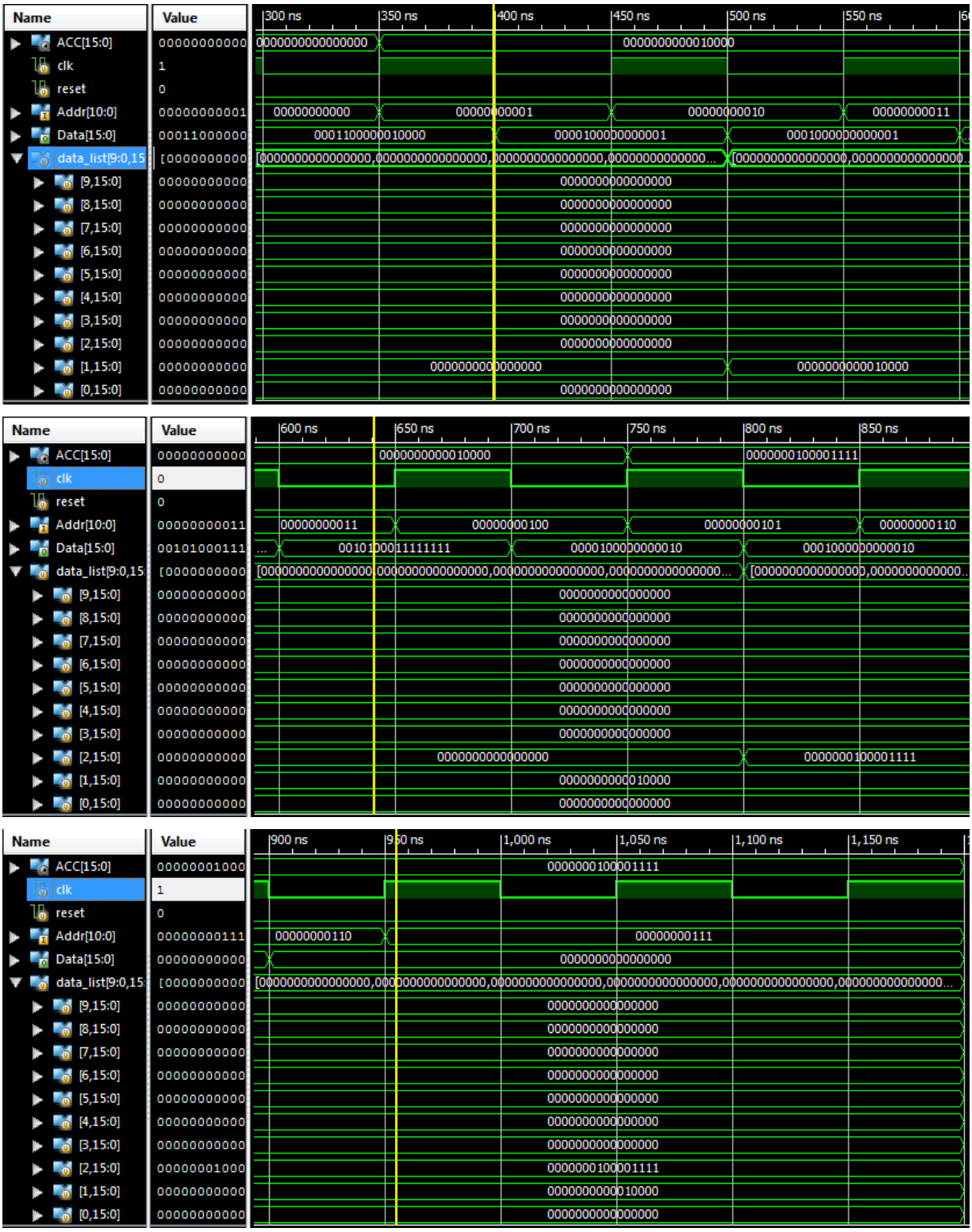
El código del testbench que lleva a cabo la verificación previamente descrita es el siguiente.

```
1.  module test_CPU;
2.
3.      // Inputs
4.      reg clk;
5.      reg reset;
6.      reg [15:0] instruction;
7.      reg [15:0] DM_in;
8.
9.      // Outputs
10.     wire [10:0] AddrPM;
11.     wire [10:0] AddrDM;
12.     wire [15:0] ACC;
13.     wire Rd_out;
14.     wire Wr_out;
15.
16.     // Instantiate the Unit Under Test (UUT)
17.     CPU uut (
18.         .clk(clk),
19.         .reset(reset),
20.         .instruction(instruction),
21.         .DM_in(DM_in),
22.         .AddrPM(AddrPM),
23.         .AddrDM(AddrDM),
24.         .ACC(ACC),
25.         .Rd_out(Rd_out),
26.         .Wr_out(Wr_out)
27.     );
28.
29.     initial begin
30.         // Initialize Inputs
31.         reset = 0;
32.         instruction = 0;
33.         DM_in = 0;
34.
35.         // Wait 100 ns for global reset to finish
36.         #100;
37.
38.         // Add stimulus here
39.         reset = 1; // Se recetea Datapath
40.
41.
42.         #100;
43.         reset = 0; // termina reseteo
```

```
44.       instruction = 16'b0001100000001010; // LDI 10;
45.       DM_in = 2;
46.
47.       #100;
48.       instruction = 16'b0000100000000001; // STO 1
49.       DM_in = 0;
50.
51.       #100;
52.       instruction = 16'b0001000000000001; // LD 1
53.       DM_in = 10;
54.
55.       #100;
56.       instruction = 16'b0010100000000010; // ADDI 2
57.       DM_in = 10;
58.
59.       #100;
60.       instruction = 16'b0000100000000010; // STO 2
61.       DM_in = 0;
62.
63.       #100;
64.       instruction = 16'b0001000000000010; // LD 2
65.       DM_in = 12;
66.
67.       #100;
68.       instruction = 16'b0000000000000000; // HALT
69.       DM_in = 0;
70.
71.       #100;
72.       instruction = 16'b0000000000000000; // HALT
73.       DM_in = 0;
74.
75.    end
76.
77.       // Generacion de clock en circuito secuencial
78.    always begin
79.       clk = 1'b0;
80.       #(100/2) clk = 1'b1;
81.       #(100/2);
82.
83.    end
84.
85.
86. endmodule
```

Testbench de BIP

Este testbench consiste en verificar si se cumple que el conjunto de todos los bloques testados anteriormente funcionan de la manera adecuada según las especificaciones de la referencia [1]. Para llevar a cabo la verificación el tesbtench consiste en verificar si se cumple el sincronismo de las memorias y la ejecución de manera adecuada de las instrucciones por ciclo reloj. Utilizando la misma herramienta que en los test anteriores se obtiene la siguiente simulación.



Como se observa en las imágenes de la simulación podemos observar cómo se va cumpliendo cada una de las instrucciones en un ciclo reloj como también cómo interactúan las memorias de manera correcta manteniendo la consistencia de los datos según las indicaciones solicitadas en las instrucciones.

Conclusión de los testbench

Con todos los tesbench pudimos verificar que el funcionamiento de los bloques más importantes fue el correcto según las especificaciones en la referencia [1], de manera que verificamos también que en la generación del bloque BIP su funcionamiento fue el correcto cumpliendo tanto con tiempos como con resultados.

Referencias

1. Documento de Pereira, M. C., P. V. Viera, A. L. A. Raabe, and C. A. Zeferino. "A Basic Processor for Teaching Digital Circuits and Systems Design with FPGA." 2012 VIII Southern Conference on Programmable Logic (2012). Print.