Trabajo Práctico 3

Procesador B.I.P.

Materia: **Arquitectura de Computadoras**

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

Profesor: **Santiago Rodriguez**

Año Lectivo: **2016**

Introducción

En el presente trabajo práctico se implementará un procesador BIP (Basic Instruction Procesor) sobre la placa de desarrollo Nexys 3, la cual contiene una FPGA Spartan E6.

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

Desarrollo

Para el desarrollo de este trabajo práctico, se siguió el esquema de bloques obtenido del paper “A Basic Processor for Teaching Digital Circuits and Systems Design with FPGA“.

El diagrama en bloques es el siguiente:

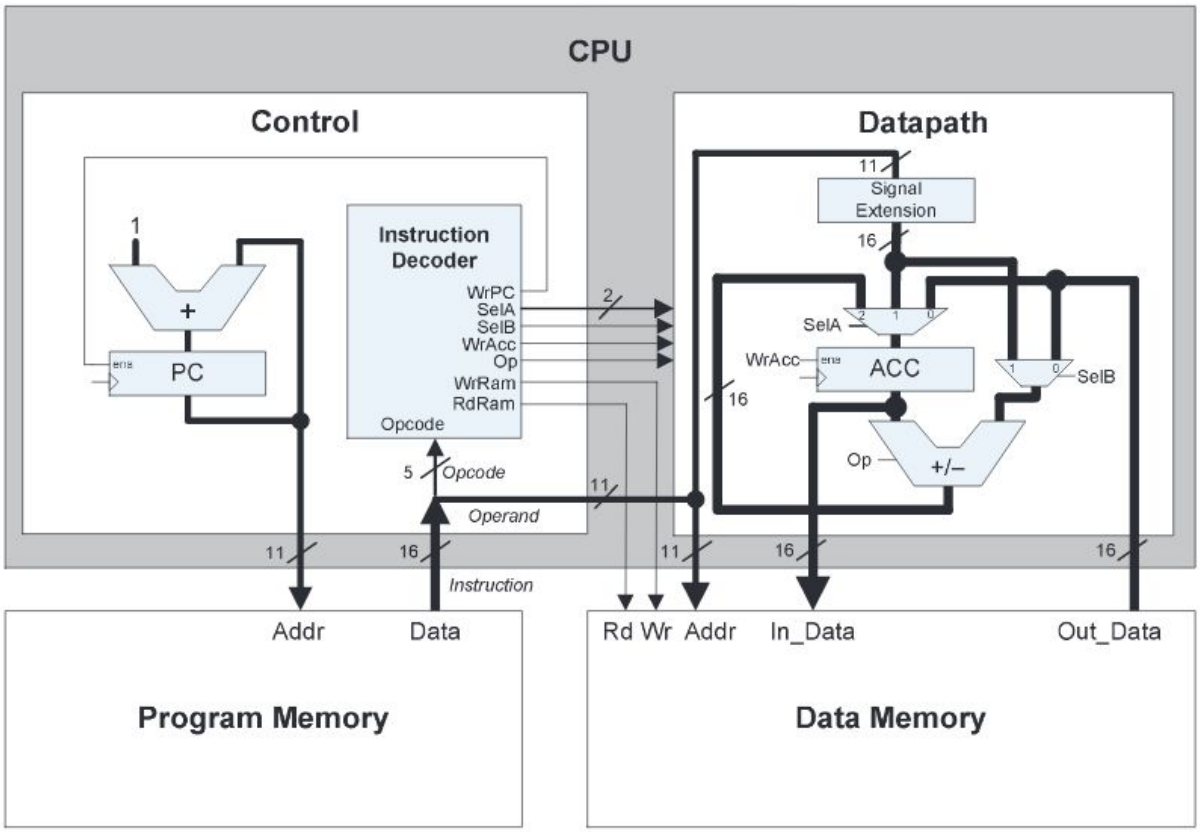


Figure 3. BIP I organization

Las instrucciones soportadas por este procesador son:

TABLE I. INSTRUCTION SET

Operation	Opcode	Instruction	Data Memory (DM) and Accumulator (ACC) Updating	Program Counter (PC) updating	Affected Flags	BIP Model
Halt	00000	**HLT**		PC ← PC		I, II
Store Variable	00001	**STO** operand	DM[operand] ← ACC	PC ← PC + 1		I, II
Load Variable	00010	**LD** operand	ACC ← DM[operand]	PC ← PC + 1		I, II
Load Immediate	00011	**LDI** operand	ACC ← operand	PC ← PC + 1		I, II
Add Variable	00100	**ADD** operand	ACC ← ACC + DM[operand]	PC ← PC + 1	Z,N	I, II
Add Immediate	00101	**ADDI** operand	ACC ← ACC + DM	PC ← PC + 1	Z,N	I, II
Subtract Variable	00110	**SUB** operand	ACC ← ACC - DM[operand]	PC ← PC + 1	Z,N	I, II
Subtract Immediate	00111	**SUBI** operand	ACC ← ACC - operand	PC ← PC + 1	Z,N	I, II

A continuación se explicarán los bloques del sistema.

Main

Es el bloque encargado de unir los 4 bloques principales, es decir, Data Memory, Program Memory, Control Block y Datapath. Además de que se establecen los valores iniciales de los parámetros para hacer parametrizable el sistema (AB y DB).

```
`timescale 1ns / 1ps
//////////////////////////////////////////////////////////////////////////////////
// Company:
// Engineer:
//
// Create Date:    18:39:02 11/03/2016
module Main(clk, rx, reset, tx);
//---------------------------------- Parametros ---------------------------------//
parameter AB = 11; //Address Bus
parameter DB = 16; //Data Bus
//--------------------------------Entradas y Salidas--------------------------//
//Del bloque Control Block
input clk;
//Del bloque Datapath
//Para debugger con UART
input rx;
input reset;
output tx;
//---------------------------------Conectores-----------------------------------//
wire [1:0] SelA;
wire SelB;
wire WrAcc;
wire Op;
reg Clear = 0;
wire WrRam;
wire RdRam;
wire [DB-1:0] Out_Data;
wire [DB-1:0] In_Data;
wire [AB-1:0] Addr_DM;
wire [DB-1:0] Data;
wire [AB-1:0] Addr;
//Para debugger con UART
wire wr_uart;
wire start_bip;

//---------------------------------Bloques--------------------------------------//
Control_Block #(AB) CB (clk, Data[15:11], SelA, SelB, WrAcc, Op, WrRam, RdRam, Addr, start_bip, wr_uart);
Datapath #(AB, DB) DP (SelA, SelB, WrAcc, Op, Clear, clk, Out_Data, In_Data, Data[10:0], Addr_DM);
Data_Memory #(AB, DB) DM (RdRam, WrRam, Addr_DM, In_Data, clk, Out_Data);
Program_Memory #(AB, DB) PM (Addr, clk, Data);
//Para debuggear
UART Uart (rx, clk, reset, tx, In_Data, wr_uart, start_bip);

endmodule
```

Data Memory

Este bloque será manejado por el Datapath. Tiene 2 flags (RdRam y WrRam), que permite leer y escribir datos de la memoria.

```
module Data_Memory #(parameter AB = 11, parameter DB = 16) (RdRam, WrRam, Addr, In_Data, clk, Out_Data);
//---------------------------------- Parametros --------------------------------//
//----------------------------------Entradas y Salidas--------------------------//
input RdRam;
input WrRam;
input [AB-1:0] Addr;
input [DB-1:0] In_Data;
input clk;
output reg [DB-1:0] Out_Data;
//----------------------------------Conectores----------------------------------//
reg [DB-1:0] Mem [0:100]; //2048 palabras de 16 bits
integer i;
//-----------------------------------Logica-------------------------------------//

//Para simulacion
initial
   begin
      Out_Data <= 0;
      Mem[4] = 'b0011111111000000;
      Mem[0] = 'b0000000000000000;
      Mem[1] = 'b0000000000000000;
      Mem[2] = 'b0000000000000000;
      Mem[3] = 'b0000000000000000;
      Mem[5] = 'b0000000000000000;
      Mem[6] = 'b0000000000000000;
      Mem[7] = 'b0000000000000000;
      Mem[8] = 'b0000000000000000;
   end

always@(negedge clk)
   begin
      if(WrRam == 1)
         Mem[Addr] = In_Data;
      else if(RdRam == 1)
         Out_Data = Mem[Addr];
   end

endmodule
```

Como se ve en el código, se inicializan los primeros 8 valores del Data Memory. En la posición 4 se carga un operando en particular para la operación de SUMA.

Program Memory

De este bloque, se obtendrán las instrucciones a realizar, donde los primeros 5 bits, corresponden al tipo de operación, los cuales irán al bloque “Instruction Decoder”, y los últimos 11 bits, serán la dirección de memoria que irá al Datapath o un valor inmediato que desea cargarse en el Acumulador.

```
module Program_Memory #(parameter AB = 11, parameter DB = 16)(
    input [AB-1:0] Addr,
    input clk,
    output reg [DB-1:0] Data
    );

//----------------------------------Conectores-----------------------------------//
reg [DB-1:0] Mem[0:100]; //2048 palabras de 16 bits
integer i;
//-------------------------------------Logica------------------------------------//

//Para simulacion
initial
   begin
      //Cargamos datos de prueba;
      Mem[0]= 'b0000100000000001;
      Mem[1]= 'b0001000000000010;
      Mem[2]= 'b0001100000000011;
      Mem[3]= 'b0010000000000100;
      Mem[4]= 'b0010100000000101;
      Mem[5]= 'b0011000000000110;
      Mem[6]= 'b0011100000000111;
      Mem[7]= 'b0000000000000000;
   end

   always @(negedge clk)
   begin
      Data [DB-1:0] = Mem[Addr];
   end
endmodule
```

Se cargaron en el código las 7 instrucciones que permite nuestro BIP para corroborar el funcionamiento completo. Entre cada una de las instrucciones se colocó una instrucción “Halt” para detener el PC y visualizar por UART el valor del “Acumulador”.

Control Block

El bloque de control, se encarga de la sincronización de todo el procesador, a través de dos bloques esenciales:

- PC: El program counter nos da la siguiente dirección de instrucción a realizar. La calcula realizando una suma a través del bloque ALU.
- Instruction Decoder: Este bloque, setea todos los flags, habilitando entradas y salidas, según la instrucción a realizar.

```
module Control_Block #(parameter AB = 11)(clk, OpCode, SelA, SelB, WrAcc, Op, WrRam, RdRam, Addr, WrPC_Input, WrPC_Output);
//---------------------------------Entradas y Salidas----------------------------//
//Del bloque Program Counter
input clk;
output [AB-1:0] Addr;
//Del bloque Instruction Decoder
input [4:0] OpCode;
output [1:0] SelA;
output SelB;
output WrAcc;
output Op;
output WrRam;
output RdRam;
//Del bloque ALU
reg [AB-1:0] increment = 1;
reg [5:0] operando_suma = 'b100000;
//---------------------------------Conectores-----------------------------------//
output WrPC_Output;//PASARA POR UNA AND
input WrPC_Input;
wire [AB-1:0] address_output;

//---------------------------------Bloques--------------------------------------//
ALU #(AB) alu_pc (Addr, increment, operando_suma, address_output);
Program_Counter #(AB) PC (clk, address_output, WrPC_Input, Addr);
Instruction_Decoder ID (OpCode, WrPC_Output, SelA, SelB, WrAcc, Op, WrRam, RdRam);

endmodule
```

WrPC_Input y WrPC_Output: El enable “WrPC” del contador de programa saldrá del “Intruction_Decoder” y del “Control_Block” para pasar a través del módulo UART (dentro del cual se encuentra una operación AND entre el “WrPC” del “Instruction Decoder” y lo que se reciba por la UART (VER MÓDULO UART).

Program Counter

```
module Program_Counter #(parameter AB = 11)(clk, address_bus, WrPC, Addr);
//----------------------------------Entradas y Salidas----------------------//
input clk;
input WrPC;
input [AB-1:0] address_bus;
output reg [AB-1:0] Addr = 0;

//----------------------------------Conectores-----------------------------//

//-------------------------------------Logica-----------------------------//
always @(posedge clk) //El Program Counter coloca en Addr la direccion que se leerá en el próximo clock.
   begin
      if (WrPC == 1)
         begin
            Addr = address_bus;
         end
   end

endmodule
```

Éste módulo es sencillo, simplemente incrementará el “Addr” si WrPC se encuentra en 1. (WrPC proviene del “Instruction Decoder”).

Instruction Decoder

```
module Instruction_Decoder(OpCode, WrPC, SelA, SelB, WrAcc, Op, WrRam, RdRam);
//Declaración de Entradas y Salidas
input [4:0] OpCode;

output reg WrPC = 1; //ARRANCA EN UNO PARA QUE NO IMPRIMA POR UART NI BIEN ARRANCA LA PLACA
output reg [1:0] SelA;
output reg SelB;
output reg WrAcc = 0;
output reg Op; //1 para Suma, 0 para resta
output reg WrRam = 0;
output reg RdRam = 0;

//Logica
always @(*)
begin
   //Analizamos el OpCode para ver como decodificar la instruccion
   case(OpCode)
      'b00000: //HLT
         begin
            WrPC = 0; //No hacemos nada, ni permitimos que el PC incremente
            WrAcc = 0;
            WrRam = 0;
            RdRam = 0;
         end
      'b00001: //STO
         begin
            WrPC = 1; //Incrementamos el PC
            WrRam = 1; //Y hablilitamos la escritura en la DM, entonces se guarda la salida del ACC y la ADDR
            RdRam = 0;
            WrAcc = 0;
         end
      'b00010: //LD
         begin
            WrPC = 1; //Incrementamos el PC
            RdRam = 1; //Habilitamos lectura, para poder sacar el operando de DM (Out_Data)
            WrAcc = 1; //Procesamos lo que hay en el acumulador
            WrRam = 0;
            SelA = 0; //Se selecciona la entrada al mux1 para que vaya al acumulador
         end
      'b00011: //LDI
         begin
            WrPC = 1; //Incrementamos el PC
            WrAcc = 1; //Procesamos lo que hay en el acumulador
            RdRam = 0;
            WrRam = 0;
            SelA = 1; //Se selecciona la entrada al mux1 para que vaya al acumulador
         end
      'b00100: //ADD
         begin
            WrPC = 1; //Incrementamos el PC
            WrAcc = 1; //Procesamos lo que hay en el acumulador
            RdRam = 1; //Habilitamos lectura, para poder sacar el operando de DM (Out_Data)
            WrRam = 0;
            SelB = 0; //Seleccionamos la entrada del multiplexor2
            SelA = 2; //Se selecciona la entrada al mux1 para que vaya al acumulador
            Op = 1; //Hacemos la suma
         end
      'b00101: //ADDI
         begin
            WrPC = 1; //Incrementamos el PC
            WrAcc = 1; //Procesamos lo que hay en el acumulador
            RdRam = 0;
            WrRam = 0;
            SelB = 1; //Seleccionamos la entrada del multiplexor2
            SelA = 2;
            Op = 1; //Hacemos la suma
         end
      'b00110: //SUB
         begin
            WrPC = 1; //Incrementamos el PC
            WrAcc = 1; //Procesamos lo que hay en el acumulador
            RdRam = 1; //Habilitamos lectura, para poder sacar el operando de DM (Out_Data)
            WrRam = 0;
            SelB = 0; //Seleccionamos la entrada del multiplexor2
            SelA = 2;
            Op = 0; //Hacemos la resta
         end
      'b00111: //SUBI
         begin
            WrPC = 1; //Incrementamos el PC
            WrAcc = 1; //Procesamos lo que hay en el acumulador
            RdRam = 0;
            WrRam = 0;
            SelB = 1; //Seleccionamos la entrada del multiplexor2
            SelA = 2;
            Op = 0; //Hacemos la resta
         end
      default:
         begin
            WrPC = 0; //No hacemos nada, ni permitimos que el PC incremente
            WrAcc = 0;
            WrRam = 0;
            RdRam = 0;
         end

   endcase
end

endmodule
```

Sus entradas y salidas son iguales a las del diagrama de bloques.

Datapath

En el datapath, se realiza el procesamiento de los datos, y según qué instrucción se realice, se guarda o no el dato en la memoria de datos. Todos los flags en juego en este bloque, provienen del bloque del control. El datapath se compone de los siguientes subbloques:

- Signal Extension: Es el encargado de convertir un valor de 11 bits, a 16 bits, replicando el bit más significativo.
- Multiplexores: Cuenta con 2 multiplexores, uno de 3 entradas, y otro de 2, ambos de 1 salida. El primero sirve para seleccionar el dato a procesar por el acumulador y el segundo el dato hacia la ALU.
- Acumulador: Este bloque, básicamente, lo que hace es almacenar la entrada y mostrarlo a su salida.
- ALU: Bloque sumador o restador, según la operación que se indique en la instrucción. Y realiza la operación con la salida del acumulador y la salida del multiplexor de 2 entradas.

```
module Datapath #(parameter AB = 11, parameter DB = 16) (SelA, SelB, WrAcc, Op, Clear, clk, Out_Data, In_Data, Addr, Addr_DM);
//---------------------------------Parametros----------------------------------//
//---------------------------Entradas y Salidas--------------------------------//
input [1:0] SelA;
input SelB;
input WrAcc;
input Op;
input Clear;
input clk;
input [DB-1:0] Out_Data;
input [AB-1:0] Addr;
output reg [DB-1:0] In_Data = 16'b0;
output reg [AB-1:0] Addr_DM;
//------------------------------ Conectores ----------------------------------//
wire [DB-1:0] salida_signalextension;
wire [DB-1:0] salida_ALU;
wire [DB-1:0] salida_mux1;
wire [DB-1:0] salida_mux2;
wire [DB-1:0] salida_acc;
reg [5:0] operacion;
//-------------------------------Uso de modulos--------------------------------//
Signal_Extension #(AB, DB) signalextension (Addr, salida_signalextension);
Multiplexor_3in_1out #(DB) mux1 (salida_ALU, salida_signalextension, Out_Data, SelA, salida_mux1);
Multiplexor_2in_1out #(DB) mux2 (salida_signalextension, Out_Data, SelB, salida_mux2);
ACC #(DB) acumulador (salida_mux1, clk, WrAcc, Clear, salida_acc);
ALU #(DB) alu (salida_acc, salida_mux2, operacion, salida_ALU);
//---------------------------------Logica--------------------------------------//


//Para saber opearcion de ALU, si es suma o resta
always @(Op)
begin
   if (Op == 1)
      operacion <= 'b100000;
   else
      operacion <= 'b100010;
end

always @(Addr)
begin
   Addr_DM = Addr;
end

always @(salida_acc)
begin
   In_Data = salida_acc;
end

endmodule
```

Signal Extension

```
module Signal_Extension #(parameter AB = 11, parameter DB = 16) (
    input [AB-1:0] Addr,
    output [DB-1:0] Salida
    );

assign Salida = {{DB-AB{Addr[AB-1]}}, Addr};

endmodule
```

Multiplexores

```
module Multiplexor_3in_1out #(parameter DB = 16)(
    input [DB-1:0] DatoA,
    input [DB-1:0] DatoB,
    input [DB-1:0] DatoC,
    input [1:0] Sel,
    output reg [DB-1:0] Salida
    );

always @(*)
begin
   if(Sel == 2)
      Salida = DatoA;
   else if(Sel == 1)
      Salida = DatoB;
   else if(Sel == 0)
      Salida = DatoC;
end

endmodule
```

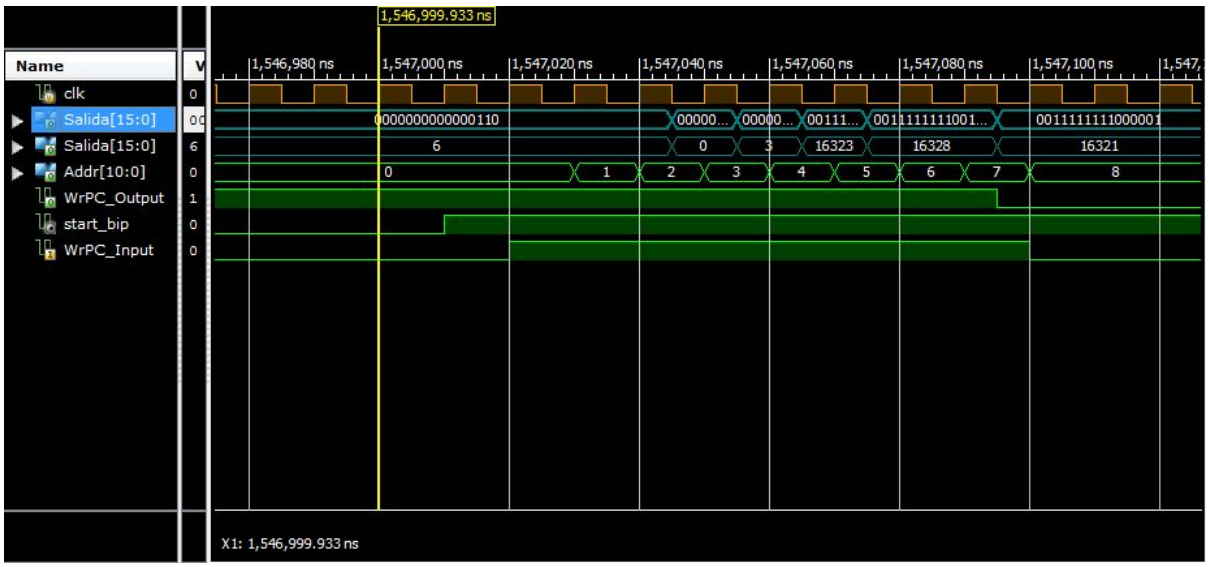
```
module Multiplexor_2in_1out #(parameter DB = 16)(
    input [DB-1:0] DatoA,
    input [DB-1:0] DatoB,
    input Sel,
    output reg [DB-1:0] Salida
    );

always @(*)
begin
   if(Sel == 1)
      Salida = DatoA;
   else
      Salida = DatoB;
end

endmodule
```

Acumulador

```
module ACC #(parameter DB = 16)(Entrada, clk, WrAcc, Clear, Salida);

input [DB-1:0] Entrada;
input clk;
input WrAcc;
input Clear;
output reg [DB-1:0] Salida = 0;

reg dos_clock = 1;

always @(posedge clk)
begin
   if(Clear == 1)
      begin
         Salida <= 16'b0;
      end
   if (WrAcc == 1)
      begin
         if (dos_clock == 1)
            begin
               Salida <= Entrada;
               dos_clock = 0;
            end
         else
            dos_clock = 1;
      end
end

endmodule
```

En este caso, se realiza algo similar que en el PC, es decir, se utiliza un registro “dos_clock” con el que el Acumulador funcionará durante un clock, y durante el siguiente no (actúa de manera alternada) por el retraso de lectura del “Program_Memory”.

Test Bench



Aquí podemos visualizar el funcionamiento más importante. A grandes rasgos, nuestro procesador BIP resuelve las operaciones en un ciclo de reloj.

Cuando “WrPC_Input” se pone a 1, le indica al Program Counter que inicie su conteo. A partir de este momento, por cada ciclo se realiza una operación. Éstas son las siguientes:

Referencia:
Negro: Operando
Rojo: Operador

Aclaración:
El acumulador está inicializado con el valor 6.

Mem[0]= 'b**00001**00000000001; ← **Storage,** guarda en memoria lo del Acumulador en la posición **1**. **Resultado Parcial del ACC= 6**

Mem[1]= 'b**00010**00000000010; ← **Load**, carga en el ACC el valor de la posición de memoria **2** del Data Memory. **Resultado Parcial del ACC= 0**

Mem[2]= 'b**00011**00000000011; ← **Load Inmediate**, carga en el ACC el valor **3**. **Resultado Parcial del ACC= 3**

Mem[3]= 'b**00100**00000000100; ← **Add**, suma al valor de ACC el valor de la posición **4** del Data Memory y lo guarda en el ACC. **Resultado Parcial del ACC= 16323**

Mem[4]= 'b**00101**00000000101; ← **Add Inmediate**, suma al valor de ACC el valor **5** y lo guarda en el ACC. **Resultado Parcial del ACC= 16328**

Mem[5]= 'b**00110**00000000110; ← **Sub**, resta al valor de ACC el valor de la posición **6** del Data Memory y lo guarda en el ACC. **Resultado Parcial del ACC= 16328**

Mem[6]= 'b**00111**00000000111; ← **Sub Inmediate**, resta al valor de ACC el valor **7** y lo guarda en el ACC. **Resultado Parcial del ACC= 16321**

Mem[7]= 'b**00000**00000000000; ← **Halt**, no aumenta el PC, y envía el resultado del acumulador por la UART.

Resultado Final: **0011111111000001 (16321)**

Prueba de Implementación