

Diseño Digital – Lab 3

El objetivo de esta práctica consiste en realizar un procesador RISC segmentado, con un número reducido de instrucciones.

Las instrucciones que realizará dicho procesador son:

- Add = Suma
- Sub = Resta
- And
- Or
- Slt = Set Less Than (1 si $a < b$)
- Lw = Load Word (Cargar desde memoria)
- Sw = Store Word (Guardar en memoria)
- Beq = Branch on equal (Salta si $a = b$)

Y su pipeline constará de 5 etapas:

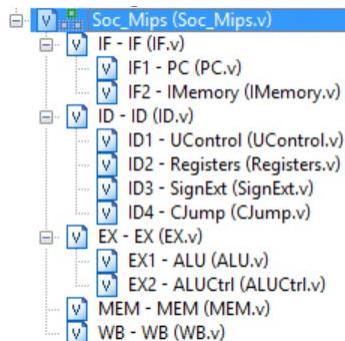
- IF = Instruction Fetch
- ID = Instruction Decode
- EX = Execute
- MEM = Memoria
- WB = Write Back

Y tendrá también gestión de riesgo de datos, control de anticipación y detección de riesgos y soporte de excepciones.

La realización del procesador se ha dividido en 6 versiones:

- V1 = Procesador Monociclo
- V2 = Procesador Segmentado
- V3 = Control de anticipación y riesgos
- V4 = Limpieza y mejora del código y soporte de riesgo de saltos
- V5 = Soporte excepciones. Overflow, instrucción invalida y direccionamiento inválido
- V6 = Prototipado en la placa FPGA

Versión 1 - Monociclo



- IF -> PC y Memoria Instrucciones
- ID -> Unidad Control, Banco Registros, Extensor de Signo y cálculo de dirección de salto condicional
- EX -> ALU y Control de la ALU
- MEM -> Memoria de datos
- WB -> Writeback, dato a escribir en el banco de registros

- Etapa IF

o PC – Contador de Programa

```

module PC
  #(parameter PC_WIDTH = 6)
  (input clk, rst,
   input [PC_WIDTH-1:0] PCin,
   output reg [PC_WIDTH-1:0] PCout
  );
  always@(posedge clk or posedge rst)
    if(rst)
      PCout = 0;
    else
      PCout = PCin;
endmodule

```

Pone a la salida lo que tiene en la entrada en cada ciclo de reloj

El valor que entra se calcula en el salto condicional

o Memoria de Instrucciones

```

always@(Address)
begin
  case(Address)
    0:Instruction = 'CODE_0;
    4:Instruction = 'CODE_1;
    8:Instruction = 'CODE_2;
    12:Instruction = 'CODE_3;
    16:Instruction = 'CODE_4;
    20:Instruction = 'CODE_5;
    24:Instruction = 'CODE_6;
    28:Instruction = 'CODE_7;
    32:Instruction = 'CODE_8;
    36:Instruction = 'CODE_9;
    40:Instruction = 'CODE_A;
    44:Instruction = 'CODE_B;
    48:Instruction = 'CODE_C;
    52:Instruction = 'CODE_D;
    56:Instruction = 'CODE_E;
    60:Instruction = 'CODE_F;
    default: Instruction = 32'h00;
  endcase
end

```

Es la memoria en donde está el programa que va a ejecutar el procesador.

Se utilizan parámetros para simplificar el código

```

#define CODE_0 32'h8C010010 // lw $1,$0(10) # a = M[10]
#define CODE_1 32'h8C020011 // lw $2,$0(14) # b = M[11]
#define CODE_2 32'h0041182A // slt $3,$2,$1 # a > b?
#define CODE_3 32'h10600009 // beq $3,$0,L1/9 # a == b
#define CODE_4 32'hAC010012 // sw $1,$0(18) # M[12] = a
#define CODE_5 32'h00222022 // sub $4,$1,$2 # c = a - b
#define CODE_6 32'h0082182A // slt $5,$2,$4 # c < b?
#define CODE_7 32'h10600002 // beq $5,$0,L0/2 # c >= b
#define CODE_8 32'hAC040010 // sw $4,$0(10) # M[10] = c
#define CODE_9 32'h10000005 // beq $0,$0,L2/5 # j L2

```

o Bloque IF

```

PC IF1(
  .clk(clk),
  .rst(rst),
  .PCin(jmp_address),
  .PCout(PCout)
);

assign PCnext = PCout + 4;
IMemory IF2 (
  .Address(PCout),
  .Instruction(Instruction)
);

```

El valor que entra en el PC es el valor que da el CJump.

En el caso de que la instrucción sea un branch, y ambos registros sean iguales, jmp_address será la dirección a saltar, en caso contrario, será PCnext.

- Etapa ID

- o Unidad de Control

```

assign Rtype = ( op == 6'b000000 )? 1:0;
assign lw = ( op == 6'b100011 )? 1:0;
assign sw = ( op == 6'b101011 )? 1:0;

assign RegWrite = ( Rtype || lw )? 1:0;
assign ALUSrc = ( lw || sw )? 1:0;
assign RegDst = ( Rtype )? 1:0;
assign MemtoReg = ( lw )? 1:0;
assign MemWrite = ( sw )? 1:0;
assign Branch = ( op == 6'b000100 )? 1:0;
assign MemRead = ( lw )? 1:0;
assign ALUop[1] = ( Rtype )? 1:0;
assign ALUop[0] = ( op == 6'b000100 )? 1:0;

```

Se asignan las distintas variables que controlan la escritura de los registros y la memoria, la lectura de memoria, los datos de entrada de la ALU y la operación de la ALU.

- o Banco de Registros

```

reg [REG_WIDTH-1:0] RegFile [0:REG_FILE_DEPTH-1];
integer i;

always@(posedge clk or posedge rst)
  begin
    if(rst)
      begin
        for(i=0;i<(REG_FILE_DEPTH);i=i+1)
          RegFile[i] <= 0;
      end
    else if ( RegWrite == 1 )
      RegFile [writer] <= writedata;
  end
  assign readd1 = ( readr1 == 0 )?0:RegFile[readr1];
  assign readd2 = ( readr2 == 0 )?0:RegFile[readr2];

```

Se declara e inicializa el banco de registros en un reset.

La lectura es asíncrona, y la escritura síncrona.

El registro 0 siempre vale 0.

- o Extensor de Signo

```

module SignExt
  #(parameter EXT_IN_WIDTH = 6,
  parameter EXT_OUT_WIDTH = 8)
  (input [EXT_IN_WIDTH-1:0] In,
  output [EXT_OUT_WIDTH-1:0] Out);

  assign Out = {((EXT_OUT_WIDTH-EXT_IN_WIDTH){In[EXT_IN_WIDTH-1]}),In};
endmodule

```

Se amplía el dato de entrada, colocando tantas veces el bit más significativo como sea necesario.

- o CJump – Salto Condicional

```

(input [SH_IN_WIDTH-1:0] ShiftIn,
input [PC_WIDTH-1:0] PCNext,
input Branch,Zero,
output [SH_OUT_WIDTH-1:0] PCJout
);

wire [SH_OUT_WIDTH-1:0] ShiftOut;
wire [SH_OUT_WIDTH-1:0] ALUR;

assign ShiftOut = ShiftIn << 2;
assign ALUR = ShiftOut + PCNext;
assign PCJout = ( Branch == 1 && Zero == 1 )? ALUR:PCNext;

```

Se hacen dos "Shifts" a la izquierda para multiplicar por 4 y así obtener la cantidad que hay que saltar el PC.

Se suma al siguiente

valor de PC para obtener la dirección de salto.

Se coloca dicho valor recién calculado o el valor actual del PC dependiendo de si es una instrucción de Branch y ambos registros son iguales.

- o Bloque ID

Se interconexinan todos los bloques anteriores.

- Etapa EX

- o ALU – Unidad Aritmética Lógica

```

always@(a or b or op)
begin
    case(op)
        0: result = a & b;
        1: result = a | b;
        2: begin
                result = a + b;
                if ( (a[ALU_WIDTH-1] == b[ALU_WIDTH-1]) && (a[ALU_WIDTH-1] != result[ALU_WIDTH-1]) )
                    Ov = 1;
                else
                    Ov = 0;
            end
        begin
            result = a - b;
            if ( (a[ALU_WIDTH-1] != b[ALU_WIDTH-1]) && (b[ALU_WIDTH-1] == result[ALU_WIDTH-1]) )
                Ov = 1;
            else
                Ov = 0;
        end
        7: if( a < b )
            result = 1;
        else
            result = 0;
        default: result = 0;
    endcase
end

assign Zero = ( result == 0 )? 1:0;

```

- o ALUCtrl

```

always@(ALUop or funct)
begin
    case(ALUop)
        0: ALUCtrl = 3'b010;
        1: ALUCtrl = 3'b110;
        2: case(funct)
                32: ALUCtrl = 3'b010;
                34: ALUCtrl = 3'b110;
                36: ALUCtrl = 3'b000;
                37: ALUCtrl = 3'b001;
                42: ALUCtrl = 3'b111;
                default: ALUCtrl = 3'b101;
            endcase
        default: ALUCtrl = 3'b101;
    endcase
end

```

- o Bloque EX

`assign data2 = (ALUSrc == 0)? readd2:SignExtendOut;` Controla el valor que le entra al segundo dato de la ALU según la variable ALUSrc.

En las instrucciones de lectura o escritura de datos, se debe poner la dirección a la que va referida en la salida de la ALU.

- Etapa MEM

```

reg [DATA_WIDTH-1:0] RegFile [0:DATA_DEPTH-1];
integer i;

always@(posedge rst or posedge clk)
    if(rst)
        begin
            for(i=0;i<(DATA_DEPTH);i=i+1)
                RegFile[i] <= 0;
        end
    else if (MemWrite == 1)
        begin
            RegFile[Address[1:0]] <= WriteData;
        end
assign ReadData = (MemRead == 1)?RegFile[Address[1:0]]:0;

```

Calcula el resultado según el valor de OP.

Soporte para Cero y Overflow.

Controla el valor de "OP" que le llega a la ALU.

En las operaciones tipo R, proviene de los bits de "Function", y en resto viene dentro del propio código de operación.

Declaración e inicialización de la memoria de datos.

Lectura asíncrona y escritura síncrona.

La lectura y escritura vienen controladas por las variables "MemRead" y "MemWrite"

- Etapa WB

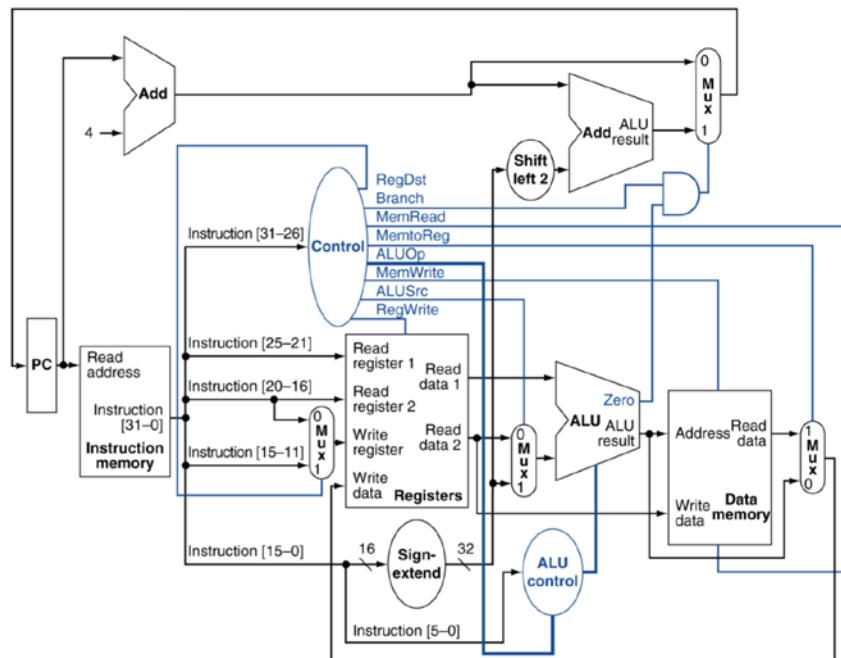
```
assign WriteBack = ( MemtoReg == 0 ) ? ALUr:ReadD;
```

pone el valor requerido en el WriteBack.

En los casos en los que haya que escribir en el banco de registros,

- Bloque General

Se interconexinan todas las entradas y salidas de las etapas anteriores siguiendo el siguiente esquema:



No es necesario añadir nada más entre las distintas etapas.

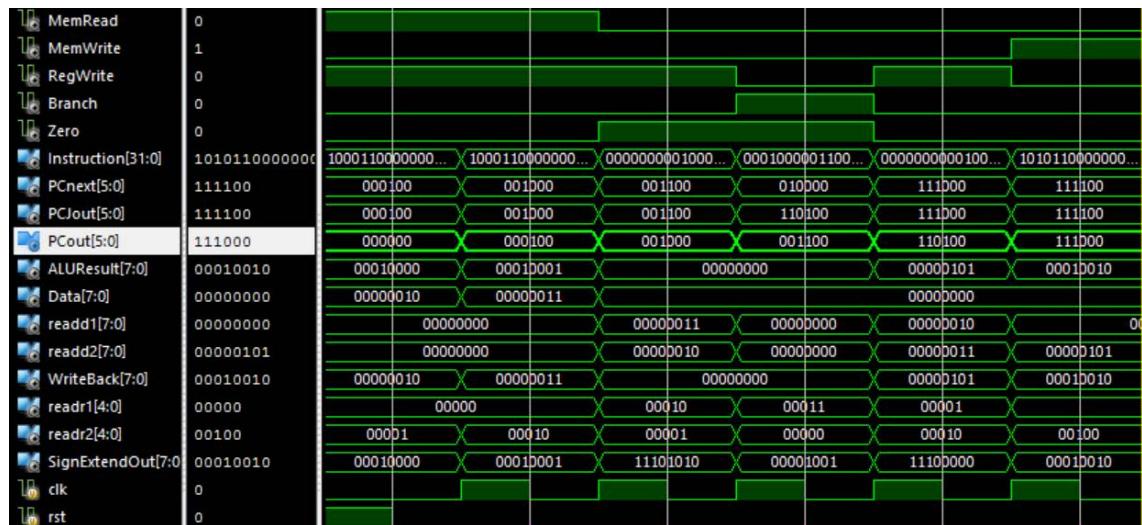
Versión 1 TestBench

Extensor de Signo:

	Out[7:0]	11101010	00000000 X 00000111 X 11101010
	In[5:0]	101010	000000 X 000111 X 101010

Se puede ver cómo los valores añadidos son iguales al bit más significativo.

Procesador Monociclo:



Siendo Mem[0] = 2 y Mem[1] = 3;

En el tercer ciclo se puede ver como se han cargado ambos datos de la memoria, en las salidas readd1 y readd2.

En el ciclo 4, como 3 no es menor que 2, ALUResult da 0, lo que provoca el salto en la siguiente instrucción

Se pude ver como entre el 4º y el 5º se efectúa efectivamente el salto debido a la instrucción Beq. Pasa de un PCnext de 010000 a 111000 (16 a 56 -> Instrucción 5 a la 15. Por lo que salta hasta L1)

En la última instrucción, se guarda el valor de readd2 en la dirección que dan los dos últimos bits de ALUResult, por lo que se guarda un 5 en la dirección 2. Lo que es correcto.

Por lo tanto el procesador monociclo funciona correctamente.

Versión 2 - Segmentado

Para segmentar las distintas etapas del procesador, se añade un registro entre cada etapa.

En total hay 4 registros, IFID, IDEX, EXMEM y MEMWB que son los que se encargan de pasar los datos de una etapa a otra. De esta manera se consigue que todas las etapas del procesador puedan trabajar al mismo tiempo, aunque a su vez conlleva riesgos a la hora de trabajar con los datos, que serán arreglados después con los soportes de anticipación, riesgos y riesgos de saltos

Además de los registros de segmentación, se ha movido la ALU en donde se calculaba la dirección de salto a la etapa EX, ya que para saber si se va a saltar o no es necesario el resultado de la ALU.

Los registros de segmentación se han declarado como cuatro registros únicos:

```
parameter IFIDsize = 32+PC_WIDTH;
parameter IDEXsize = 9+PC_WIDTH+REG_WIDTH+REG_WIDTH+EXT_OUT_WIDTH+REG_DIR_WIDTH+REG_DIR_WIDTH;
parameter EXMEMsize = 5+PC_WIDTH+1+ALU_WIDTH+REG_WIDTH+REG_DIR_WIDTH;
parameter MEMWBsize = 2+DATA_WIDTH+ALU_WIDTH+REG_DIR_WIDTH;

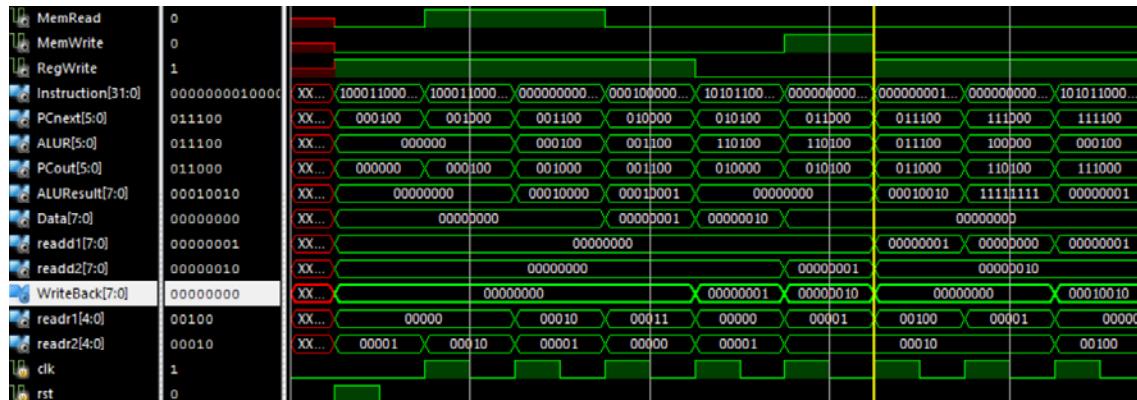
reg [IFIDsize-1:0] IFID;
reg [IDEXsize-1:0] IDEX;
reg [EXMEMsize-1:0] EXMEM;
reg [MEMWBsize-1:0] MEMWB;
```

Al ser una manera bastante ineficiente de trabajar con ellos, se cambiaron en versiones siguientes.

En cada ciclo de reloj, se pasa el contenido de los registros de las etapas, y de las etapas a los registros siguientes.

```
always@(posedge rst or posedge clk)
  if(rst)
    IDEX <= 0;
  else
    begin
      IDEX[INDEXsize-1:INDEXsize-9] <= (RegDat,ALUOp[1:0],ALUSrc,Branch,MemRead,MemWrite,RegWrite,MemtoReg);
      IDEX[INDEXsize-10:INDEXsize-9-PC_WIDTH] <= IFID[IFIDsize-1:32]; //Se pasa el PCnext
      IDEX[INDEXsize-PC_WIDTH-10:INDEXsize-9-PC_WIDTH-REG_WIDTH] <= read1; //Dato 1
      IDEX[INDEXsize-PC_WIDTH-REG_WIDTH-10:INDEXsize-9-PC_WIDTH-REG_WIDTH-REG_WIDTH] <= read2; //Dato 2
      IDEX[INDEXsize-PC_WIDTH-REG_WIDTH-REG_WIDTH-10:INDEXsize-9-PC_WIDTH-REG_WIDTH-REG_WIDTH-REG_WIDTH-EXT_OUT_WIDTH] <= SignExtendOut;
      IDEX[REG_DIR_WIDTH+REG_DIR_WIDTH-1:REG_DIR_WIDTH] <= IFID[REG_DIR_WIDTH+15:16]; //Instrucciones 20-16
      IDEX[REG_DIR_WIDTH-1:0] <= IFID[REG_DIR_WIDTH+10:11]; //Instrucciones 15-11
    end
```

Versión 2 TestBench



Debido a que aún no están implementados los sistemas de riesgos y anticipación, se puede ver que el procesador no funciona como debería, ya que no carga los valores correctos.

Las dos primeras instrucciones son "lw" que como se puede ver en la simulación, dichos valores se guardan en el banco de registros en el cuarto y quinto ciclo de reloj.

En el segundo ciclo de reloj se carga la instrucción de "slt", por lo que en el tercer ciclo de reloj cargará los registros necesarios del banco de registros. Como los valores reales no se cargan hasta el cuarto y el quinto ciclo, ninguno de los dos registros que carga la instrucción son válidos.

En este caso da la casualidad de que la instrucción "slt" da el mismo resultado que el que debería dar, pero en la instrucción siguiente, que es el "beq", en ALUResult tenemos un 0, que es incorrecto, ya que "a" es menor que "b". Por lo que no realiza la instrucción de salto, cuando sí debería realizarla.

Pero a pesar de los fallos por anticipación y riesgos, se puede ver como el funcionamiento segmentado funciona como debería. Por ejemplo en la primera instrucción de "lw", el valor se guarda después de 4 ciclos de reloj (se ve en "WriteBack"), que es lo que tarda en pasar por todas las etapas.

Versión 3 – Anticipación y Riesgos

Hay que evitar que se trabaje con valores erróneos, esto se consigue con el sistema de anticipación:

```
assign Forward_A = ( (EXMEMReqWrite == 1) && (EXMEMRegisterRd != 0) && (EXMEMRegisterRd == IDEXRegisterRs) ) ? 2:  
    ( (MEMWBReqWrite == 1) && (MEMWBRegisterRd != 0) && (EXMEMRegisterRd != IDEXRegisterRs) &&  
    (MEMWBRegisterRd == IDEXRegisterRs) ) ? 1:0;  
assign Forward_B = ( (EXMEMReqWrite == 1) && (EXMEMRegisterRd != 0) && (EXMEMRegisterRd == IDEXRegisterRt) ) ? 2:  
    ( (MEMWBReqWrite == 1) && (MEMWBRegisterRd != 0) && (EXMEMRegisterRd != IDEXRegisterRt) &&  
    (MEMWBRegisterRd == IDEXRegisterRt) ) ? 1:0;
```

Las señales "Forward_A" y "Forward_B" se encargan de cambiar el dato a las entradas de la ALU para así operar con los valores correctos.

```
assign data1 = ( Forward_A == 0 )? readd1:  
    ( Forward_A == 1 )? WBData:  
    ( Forward_A == 2 )? Address: 0;  
  
assign data2 = ( Forward_B == 0 )? readd2:  
    ( Forward_B == 1 )? WBData:  
    ( Forward_B == 2 )? Address: 0;  
  
assign data2_2 = (ALUSrc == 0)? data2:SignExtendOut;
```

"data1" es el valor a la primera entrada de la ALU y "data2_2" el valor que entra en el segundo.

Las señales que pueden redireccionarse son la que salen de la ALU y la señal del "WriteBack".

Por ejemplo:

- Add \$4, \$1, \$2
- Add \$5, \$4, \$1

En este caso, el valor obtenido para \$4 se necesita en la ALU en el ciclo siguiente, por lo tanto para este caso se deberá redireccionar "Address" que es el dato de salida de la ALU.

Para este otro caso:

- Add \$4, \$1, \$2
- Nop
- Add \$5, \$4, \$1

El valor de \$4 se necesitaría justo en el ciclo en el que se estaría guardando en el banco de registros, por lo que es necesario coger el valor desde el "WriteBack"

El sistema de riesgos está para las ocasiones en las que no es posible evitar el problema a través de una anticipación, por ejemplo:

- Lw \$1, \$0(0)
- Add \$3, \$1, \$2

En este caso, se requiere el valor de \$1 en el mismo ciclo en el que se está obteniendo de la memoria, por lo tanto no es posible anticiparlo. En este caso es necesario introducir un "bubble", que es de lo que se encarga el sistema de riesgos:

```
assign PCWrite = ( (IDEXMemRead == 1) && ( (IDEXRegisterRt == IFIDRegisterRs) || (IDEXRegisterRt == IFIDRegisterRt) ) )? 0:1;
assign IFIDwrite = ( (IDEXMemRead == 1) && ( (IDEXRegisterRt == IFIDRegisterRs) || (IDEXRegisterRt == IFIDRegisterRt) ) )? 0:1;
assign CControl = ( (IDEXMemRead == 1) && ( (IDEXRegisterRt == IFIDRegisterRs) || (IDEXRegisterRt == IFIDRegisterRt) ) )? 0:1;
```

Los "bubble" se introducen anulando todas las señales de control de dicha instrucción, evitando que avance el PC, de esta forma se repite la misma instrucción en el ciclo siguiente y para cuando se necesite el dato, éste estará en la etapa "WB" por lo que sí es posible anticiparlo.

Control del PC:

```
always@(posedge clk or posedge rst)
  if(rst)
    PCout = 0;
  else if(PCWrite == 1)
    PCout = PCin;
```

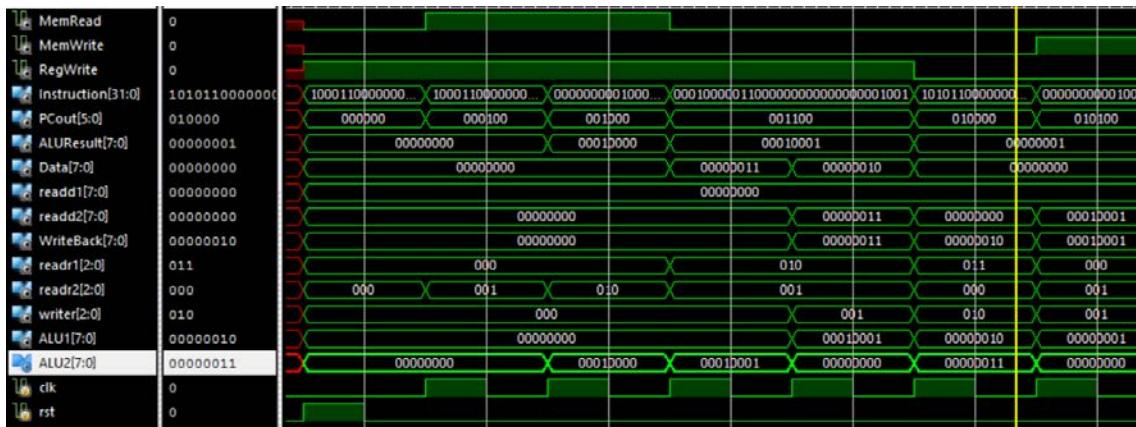
Anulación de las señales de control:

```
always@(posedge rst or posedge clk)
  if(rst)
    begin
      IDEX <= 0;
      IDEXRegisterRs <= 0;
    end
  else
    begin
      if(CControl == 1)
        IDEX[IDEXsize-1:IDEXsize-9] <= {RegDst,ALUop[1:0],ALUSrc,Branch,MemRead,MemWrite,RegWrite,MemoReg};
      else if (CControl == 0)
        IDEX[IDEXsize-1:IDEXsize-9] <= 0;
      IDEX[IDEXsize-10:IDEXsize-9-PC_WIDTH] <= IFID[IFIDsize-1:32]; //Se pasa el PCnext
      IDEX[IDEXsize-PC_WIDTH-10:IDEXsize-9-PC_WIDTH-REG_WIDTH] <= readd1; //Dato 1
      IDEX[IDEXsize-PC_WIDTH-REG_WIDTH-10:IDEXsize-9-PC_WIDTH-REG_WIDTH-REG_WIDTH] <= readd2; //Dato 2
      IDEX[IDEXsize-PC_WIDTH-REG_WIDTH-REG_WIDTH-10:IDEXsize-9-PC_WIDTH-REG_WIDTH-REG_WIDTH-EXT_OUT_WIDTH] <= SignExtendOut;
      IDEX[REG_DIR_WIDTH+REG_DIR_WIDTH-1:REG_DIR_WIDTH] <= IFID[REG_DIR_WIDTH+15:16]; //Instrucciones 20-16 //Rt
      IDEX[REG_DIR_WIDTH-1:0] <= IFID[REG_DIR_WIDTH+10:11]; //Instrucciones 15-11 //Rd
      IDEXRegisterRs <= IFID[REG_DIR_WIDTH+20:21]; //Rs
    end
```

Evitar que entre otra instrucción:

```
always@(posedge clk or posedge rst)
  if(rst)
    IFID <= 0;
  else if(IFIDwrite == 1)
    begin
      IFID[IFIDsize-1:32] <= PCnext;
      IFID[31:0] <= Instruction;
    end
```

Versión 3 – TestBench



Se puede ver como en el tercer ciclo, al cargarse la instrucción de "slt" el sistema de riesgo comprueba que se necesita un registro que aún está en memoria, por lo tanto se introduce un "bubble" en el cuarto ciclo.

En el quinto ciclo, la instrucción de "slt" llega a la etapa EX, y podemos ver como a la entrada de la ALU ("ALU1" y "ALU2") tenemos un 2 y un 3, que son Mem[1] y Mem[0] respectivamente.

Por lo que se comprueba que tanto el sistema de anticipación como el de riesgos funcionan correctamente.

Versión 4 – Riesgo de Saltos

En esta versión se ha mejorado el código general del programa, y se ha incluido el sistema de riesgo de saltos.

La mejora del código es principalmente en la estructura de los registros de segmentación.

En vez de utilizar un registro para cada etapa en el que se guardan todos los valores, se ha simplificado y se ha creado un registro para cada valor:

```
reg [31:0]           IFID_Instruction,
reg [PC_WIDTH-1:0]   IFID_FCNext,

reg [7:0]            IDEX_ControlSignals,
reg [REG_WIDTH-1:0]  IDEX_ReadData1,
reg [REG_WIDTH-1:0]  IDEX_ReadData2,
reg [EXT_OUT_WIDTH-1:0] IDEX_SignExtend,
reg [REG_DIR_WIDTH-1:0] IDEX_RegisterRs,
reg [REG_DIR_WIDTH-1:0] IDEX_RegisterRt,
reg [REG_DIR_WIDTH-1:0] IDEX_RegisterRd,

reg [3:0]            EXMEM_ControlSignals,
reg [ALU_WIDTH-1:0]  EXMEM_ALUResult,
reg [REG_WIDTH-1:0]  EXMEM_ReadData2,
reg [REG_DIR_WIDTH-1:0] EXMEM_RegisterRd,

reg [1:0]             MEMWB_ControlSignals,
reg [DATA_WIDTH-1:0]  MEMWB_ReadData,
reg [ALU_WIDTH-1:0]  MEMWB_Address,
reg [REG_DIR_WIDTH-1:0] MEMWB_RegisterRd
```

De esta forma es mucho más fácil entender y trabajar con el código ya que no es necesario estar contando los bits que ocupa cada variable:

```
always@(posedge rst or posedge clk)
if(rst)
begin
    IDEX_ControlSignals <= 0;
    IDEX_ReadData1    <= 0;
    IDEX_ReadData2    <= 0;
    IDEX_SignExtend   <= 0;
    IDEX_RegisterRt  <= 0;
    IDEX_RegisterRd  <= 0;
    IDEX_RegisterRs  <= 0;
end
else
begin
    if(CControl == 1)
        IDEX_ControlSignals <= {RegDst, ALUop[1:0], ALUSrc, MemRead, MemWrite, RegWrite, MemtoReg};
    else if (CControl == 0)
        IDEX_ControlSignals <= 0;

    IDEX_ReadData1    <= readd1;
    IDEX_ReadData2    <= readd2;
    IDEX_SignExtend   <= SignExtendOut;
    IDEX_RegisterRs  <= IFID_Instruction[REG_DIR_WIDTH+20:21];
    IDEX_RegisterRt  <= IFID_Instruction[REG_DIR_WIDTH+15:16];
    IDEX_RegisterRd  <= IFID_Instruction[REG_DIR_WIDTH+10:11];
end
```

El riesgo de saltos ocurre cuando se produce un salto de instrucciones.

Al estar segmentado el proceso, la comprobación del salto se realiza en la etapa EX, por lo que entre la instrucción de salto y la instrucción a saltar se introducen varias instrucciones indeseadas que pueden causar problemas.

Para evitar esto, primero se va a desplazar la comprobación de registros iguales a la etapa ID para así reducir las instrucciones que entran antes del salto.

Así mismo, hay que mover la ALU que calcula el valor de la instrucción a saltar a esta etapa también.

A la salida del banco de registros se va a colocar un bloque que compruebe en todo momento si el valor de ambos registros es el mismo.

Pero al igual que antes, es posible que el valor que se necesita aún no se haya guardado en el banco de registros, por lo tanto también es necesario hacer un sistema de anticipación para este bloque:

```
assign reg1 = (IDEX_RegisterRd == IFID_RegisterRs) ? ALUResult:readd1;
assign reg2 = (IDEX_RegisterRd == IFID_RegisterRt) ? ALUResult:readd2;

assign Iguales = ( reg1 == reg2 ) ? 1:0;
```

En el caso de que haya una instrucción de salto, y se cumpla que ambos registros sean iguales, se cumple la condición de "Branch", y se cambiará el valor de PC.

A pesar de adelantar la comprobación, sigue introduciéndose una instrucción indeseada, por lo que es necesario borrarla.

Para ello se crea una variable IF_Flush, que será la encargada de borrar el contenido del registro IF en el caso de que se produzca un salto:

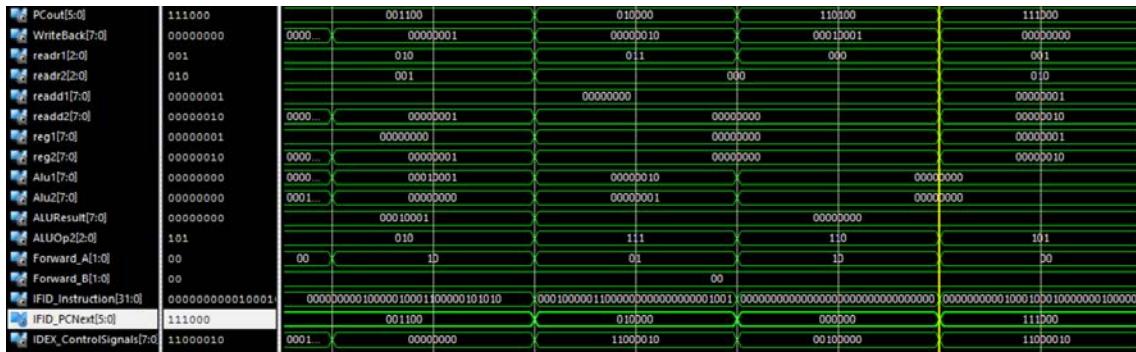
```
assign IF_Flush = ( beq == 1 && Iguales == 1 ) ? 1:0;

-----

always@(posedge clk or posedge rst)
  if(rst)
    begin
      IFID_Instruction  <= 0;
      IFID_PCNext      <= 0;
    end
  else if(IFIDWrite == 1 && IF_Flush == 0)
    begin
      IFID_PCNext      <= PCnext;
      IFID_Instruction <= Instruction;
    end
  else if(IF_Flush == 1)
    begin
      IFID_Instruction <= 0;
      IFID_PCNext      <= 0;
    end
```

De esta forma entre la instrucción de salto, y la instrucción a saltar, sólo habrá un "nop" que no interferirá en la evolución del programa.

Versión 4 – TestBench



Para ver si funciona de manera correcta, se va a establecer Mem[0] = 1 y Mem[1] = 2, de esta manera se debería saltar en la instrucción "beq".

En los dos ciclos del medio que se ven en la imagen, podemos ver como PCout pasa de 010000 a 110100, por lo que se produce el salto, y al mismo tiempo, se puede ver como el valor de IFID_Instruction es 0, por lo que se comprueba que el sistema de riesgo de saltos funciona correctamente.

Versión 5 – Excepciones

En esta versión se añade el soporte para excepciones.

Las excepciones que pueden producirse son las siguientes:

- Instrucción Inválida: Ya sea por un “OPCode” inválido, o por un “Function” inválido para las instrucciones tipo R
- Overflow en la ALU
- Direccionamiento del banco de registros inválido: En nuestro caso por ejemplo tenemos 8 registros, por lo que se puede direccionar con 3 bits pero el procesador es capaz de direccionar hasta 5 bits. Por lo tanto la instrucción se produce si se intenta direccionar algún registro por encima de los 8 de los que se dispone por ejemplo en este caso.

Al producirse una excepción, se guarda el valor del PC de la instrucción que ha provocado dicha instrucción, además se guardará en otro registro la causa de dicha instrucción.

El PC deberá saltar a una dirección determinada cuando se produzca una interrupción, y el programa deberá pararse hasta que se solucione dicha excepción, ya sea por el usuario, o por la rutina de excepciones, que es a donde se direcciona el PC.

- ExceptionCause = 1 -----> OPCode inválido
- ExceptionCause = 2 -----> Function inválido en instrucción tipo R
- ExceptionCause = 3 -----> Dirección registro inválido
- ExceptionCause = 4 -----> Overflow

El bloque "ExceptionUnit", es el que genera las señales que controlan las excepciones y en donde se generan las salidas que indican la causa y el PC de la excepción.

```

always@(OPCode or RegisterRs or RegisterRt or RegisterRd or Function or rst or Ov)
  if( rst )
    begin
      ExceptionPC <= 0;
      ExceptionCause <= 0;
    end

  else if( Ov == 1 && ExceptionCause == 0)      //Excepción si hay overflow
    begin
      ExceptionPC <= INDEX_PC;
      ExceptionCause <= 4;
    end

  //Excepción si OPCode no es ninguna de las opciones implementadas

  else if( OPCode != RTYPE && OPCode != LW && OPCode != SW && OPCode != BEQ && ExceptionCause == 0)
    begin
      ExceptionPC <= IFID_PC;
      ExceptionCause <= 1;
    end

  //Excepción si Function no es ninguna de las implementadas en el caso de una operación tipo R
  //El caso de Function 0 se descarta ya que empieza en cero tras un reset, para evitar que salte una excepción ahí

  else if( (OPCode == RTYPE) && (Function != ADD && Function != SUB && Function != AND && Function != OR &&
          Function != SLT && Function != 0) && ExceptionCause == 0)
    begin
      ExceptionPC <= IFID_PC;
      ExceptionCause <= 2;
    end

  //Excepción en el caso de que se dirccione un registro no existente. Operaciones tipo R

  else if( (OPCode == RTYPE) && (REG_DIR_WIDTH < 5) && ExceptionCause == 0)
    begin
      if( RegisterRs[4:REG_DIR_WIDTH] != 0 || RegisterRt[4:REG_DIR_WIDTH] != 0 || RegisterRd[4:REG_DIR_WIDTH] != 0 )
        begin
          ExceptionPC <= IFID_PC;
          ExceptionCause <= 3;
        end
    end

  //Igual que el de arriba pero para operaciones tipo I

  else if( (OPCode == LW || OPCode == SW || OPCode == BEQ) && (REG_DIR_WIDTH < 5) && ExceptionCause == 0)
    begin
      if( RegisterRs[4:REG_DIR_WIDTH] != 0 || RegisterRt[4:REG_DIR_WIDTH] != 0 )
        begin
          ExceptionPC <= IFID_PC;
          ExceptionCause <= 3;
        end
    end
end

```

La señal "ExceptionCause" va hasta la unidad de control:

```

assign IF_Flush = ( (beq == 1 && Iguales == 1) || ExceptionCause != 0 )? 1:0;      //Señal que limpia en condición de salto
assign ID_Flush = ( ExceptionCause != 0 )? 1:0;          //Señales que paran el programa en caso de excepción
assign EX_Flush = ( ExceptionCause != 0 )? 1:0;

```

Donde se generan las señales "ID_Flush" y "EX_Flush", el "IF_Flush" es el mismo que el que se utilizó para el riesgo de saltos, ya que su función es la misma.

La limpieza del registro IDEX:

```
always@(posedge rst or posedge clk)
  if(rst)
    begin
      IDEX_ControlSignals <= 0;
      IDEX_ReadData1     <= 0;
      IDEX_ReadData2     <= 0;
      IDEX_SignExtend    <= 0;
      IDEX_RegisterRt   <= 0;
      IDEX_RegisterRd   <= 0;
      IDEX_RegisterRs   <= 0;
      IDEX_PC            <= 0;
    end
  else
    begin
      if(CControl == 1 && ID_Flush == 0)
        IDEX_ControlSignals <= {RegDst,ALUop[1:0],ALUSrc,MemRead,MemWrite,RegWrite,MemtoReg};
      else if (CControl == 0 || ID_Flush == 1)
        IDEX_ControlSignals <= 0;
      if ( ID_Flush == 0 )
        IDEX_PC            <= IFID_PC;
      else if ( ID_Flush == 1 )
        IDEX_PC            <= IDEX_PC;

      IDEX_ReadData1     <= readd1;
      IDEX_ReadData2     <= readd2;
      IDEX_SignExtend    <= SignExtendOut;
      IDEX_RegisterRs   <= IFID_Instruction[REG_DIR_WIDTH+20:21];
      IDEX_RegisterRt   <= IFID_Instruction[REG_DIR_WIDTH+15:16];
      IDEX_RegisterRd   <= IFID_Instruction[REG_DIR_WIDTH+10:11];
    end
```

En el caso de que "ID_Flush" sea 1, el PC en este registro se mantiene constante mientras esté activa la excepción ya que es el valor del PC que provocó la excepción.

Y la limpieza del registro EXMEM:

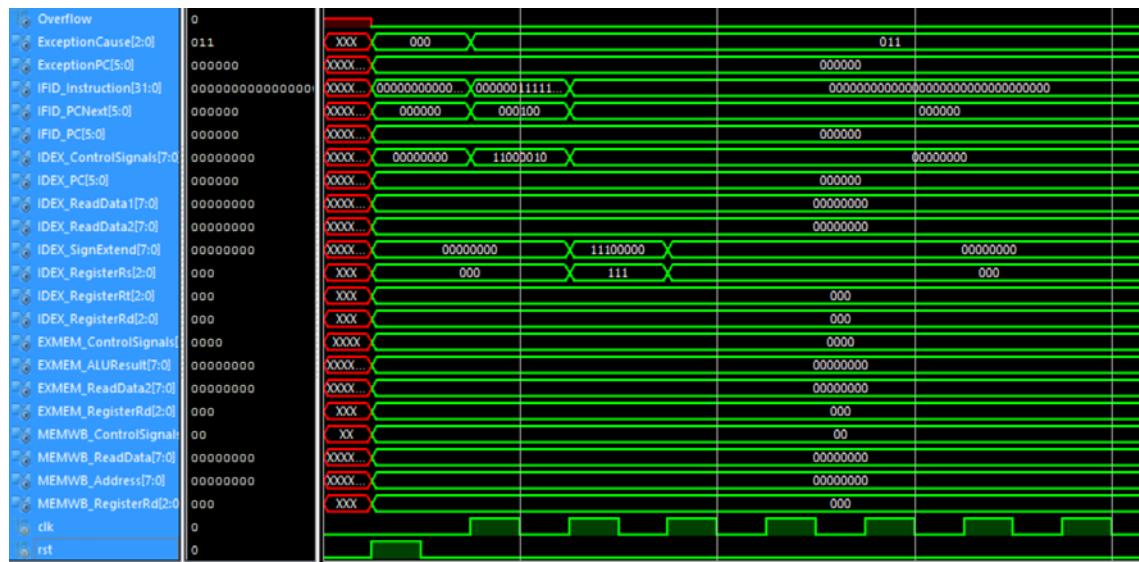
```
always@(posedge clk or posedge rst)
  if(rst)
    begin
      EXMEM_ControlSignals <= 0;
      EXMEM_ALUResult     <= 0;
      EXMEM_ReadData2     <= 0;
      EXMEM_RegisterRd   <= 0;
    end
  else
    begin
      if( EX_Flush == 1 )
        EXMEM_ControlSignals <= 0;
      else if( EX_Flush == 0 )
        EXMEM_ControlSignals <= IDEX_ControlSignals[3:0];           //MemRead,MemWrite,RegWrite,MemtoReg
      EXMEM_ALUResult      <= ALUResult;
      EXMEM_ReadData2      <= data2;
      EXMEM_RegisterRd    <= WriteReg;
    end
```

Versión 5 – TestBench:

Para probar de una forma más sencilla el funcionamiento de las excepciones, se va a modificar la memoria de instrucciones:

```
case(Address)
  0:Instruction = `CODE_0;
  //0:Instruction = 32'b000000111100000000000000100000; //Prueba para Excepción Registro Rs
  //0:Instruction = 32'b000000000001111000000000100000; //Prueba para Excepción Registro Rt
  //0:Instruction = 32'b000000000000000111100000100000; //Prueba para Excepción Registro Rd
  //0:Instruction = 32'b1111110000000000000000000000000100000; //Prueba para Excepción OpCode
  //0:Instruction = 32'b0000000000000000000000000000000000000000000000000000000000000001; //Prueba para Excepción Function
  4:Instruction = `CODE_1;
```

Para el primer, segundo y tercer caso, obtenemos:



Como se pude ver, la excepción salta en el primer ciclo de reloj ya que la instrucción modificada es la primera. Se puede ver como el valor de la excepción es 3, que es la de registro inválido, y cómo todos los registros se limpian una vez ocurre dicha excepción.

Para el cuarto caso:



La excepción ocurre de la misma forma, pero ahora su valor es 1.

Para el quinto caso:



Igual, pero ahora vale 2.

Para comprobar el funcionamiento ante un “overflow” se va a cargar un 126 en la posición 0 de memoria, y un 127 en la primera.

De esta forma al sumarse ambos valores, debería producirse un overflow.

Aquí podemos ver como el programa evoluciona de forma normal, se produce el salto de instrucción debido al "beq" y en la instrucción "add", se produce un "overflow" lo que conlleva una excepción de valor 4.

El valor del "ExceptionPC" es el de "IDEX_PC" ya que la interrupción se produce en la ALU.

También se puede ver cómo se limpian todos los registros de manera progresiva cuando se produce la interrupción.

Por lo tanto se han probado todas las excepciones posibles, y visto que todas funcionan de manera correcta.

Versión 6 – Prototipado

En esta versión se va a modificar el programa para que se pueda integrar en la placa FPGA.

Se va a modificar de tal forma que con el codificador rotatorio se pueda elegir y ver valores internos del programa:

Los registros que pueden verse son:

- Status -----> 000
- Registros 1, 2, 3 y 4 -----> 001, 010, 011, 100
- WriteBack -----> 101
- ALUResult -----> 111

Los 3 últimos LEDs servirán para saber qué registros se está viendo, y los 5 primeros, para ver el valor de dichos registros.

Además, se implementará la pantalla LCD gracias a los archivos suministrados.

También se ha añadido una forma de modificar los valores de Mem[0] a Mem[3] desde la propia placa.

```
always@(posedge clk or posedge rst)
  if(rst)
    begin
      state <= 0;
      clock <= 0;
    end
  else if(rotary_event)
    if(rotary_right && !rotary_press)
      state <= state +1;
    else if(!rotary_right && !rotary_press)
      if( clock == 1 )
        clock <= 0;
      else if ( clock == 0 )
        clock <= 1;

  assign MemSet = ( op[3] == 1 )? 1:0;
  assign MemVal = state;
  assign MemNum = ( op[0] == 1 )? 0:
    ( op[1] == 1 )? 1:
    ( op[2] == 1 )? 2:3;

  assign led[7:0] = (state == 0)? {3'b000,clock,1'b0,Status}:
    (state == 1)? {3'b001,Register1[4:0]}:
    (state == 2)? {3'b010,Register2[4:0]}:
    (state == 3)? {3'b011,Register3[4:0]}:
    (state == 4)? {3'b100,Register4[4:0]}:
    (state == 5)? {3'b101,PCout[4:0]}:
    (state == 6)? {3'b110,WriteBack[4:0]}:{3'b111,ALUResult[4:0]};
```

Con una máquina de estados se establece qué es lo que se quiere ver en los LEDs.

Los LEDs que indican qué valor es el que se está viendo será el valor que se introducirá en la memoria. Para elegir qué dirección de memoria se quiere modificar, se utilizarán los 3 primeros “switches”, el último “switch” servirá para indicar si se quiere o no modificar el valor de la memoria.

```

always@(posedge rst or posedge clk)
  if(rst)
    begin
      for(i=0;i<(DATA_DEPTH);i=i+1)
        RegFile[i] <= 0;
      RegFile[0] <= 126;
      RegFile[1] <= 127;
    end
  else if (MemWrite == 1)
    RegFile[Address[1:0]] <= WriteData;
  else if (MemSet == 1)
    RegFile[MemNum] <= MemVal;           //Esto está para modificar los valores desde la propia placa

  assign ReadData = (MemRead == 1)?RegFile[Address[1:0]]:0;

```

Si se gira el codificador en sentido horario, se modifican los LEDs, y girándolo en sentido anti horario, se envía la señal de reloj que utiliza el procesador como sincronismo.

Poder controlar la señal de reloj de manera manual es necesario para poder comprobar después en la placa si el procesador funciona o no de manera correcta una vez ya implementado.

Con unos valores de Memoria:

Mem[0] = 1;

Mem[1] = 2;

Cargamos el programa, pulsamos el botón de reset y comprobamos el funcionamiento del mismo.

Vamos girando el codificador en sentido anti horario para enviarle pulsos de reloj al procesador.

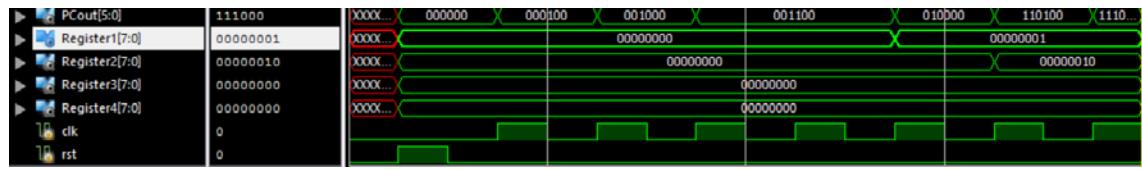
El primer cambio lo vemos aquí:



Se puede ver que el display funciona correctamente, y que hay dos números asignados a cada parámetro que queremos ver. Estos números están codificados en hexadecimal.

En PC = h10 = d16 = 5^a Instrucción -> Se carga el primer valor en el banco de registros.

Vamos a compararlo con lo obtenido en simulación:



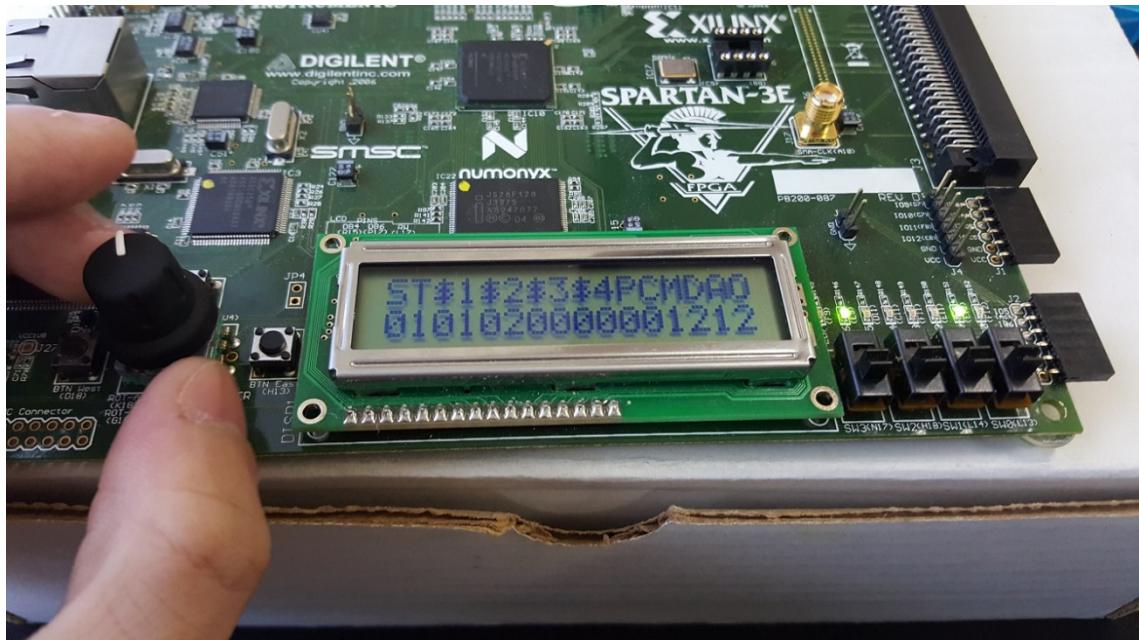
Se puede ver como el registro 1 cambia de valor en PC = 16 y el registro 2 en el siguiente, que en este caso no se ve en dicha salida ya que ocurre en el mismo ciclo que la instrucción de salto.



Vemos que ciertamente ocurre lo mismo en la placa.

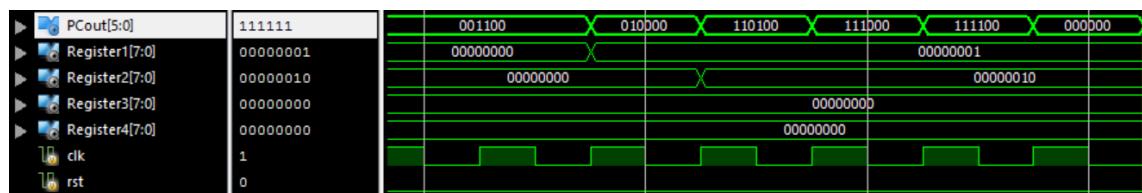
Seguimos avanzando...





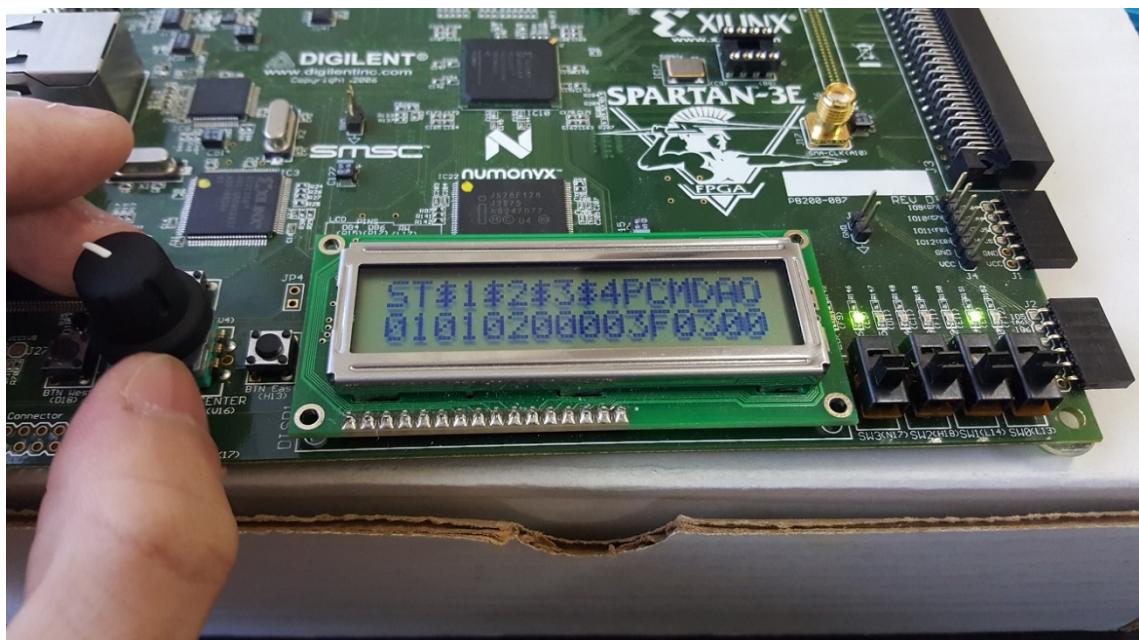
Y después del 3C vemos como pasa directamente a 00, ya que PC = h3C = d60, que es la última instrucción.

Comprobamos esto en simulación:



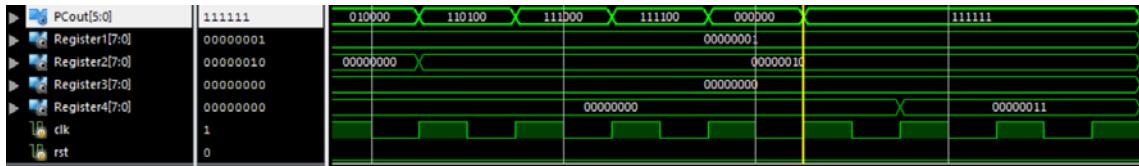
Y efectivamente el PC pasa a 0.

Si seguimos avanzando:

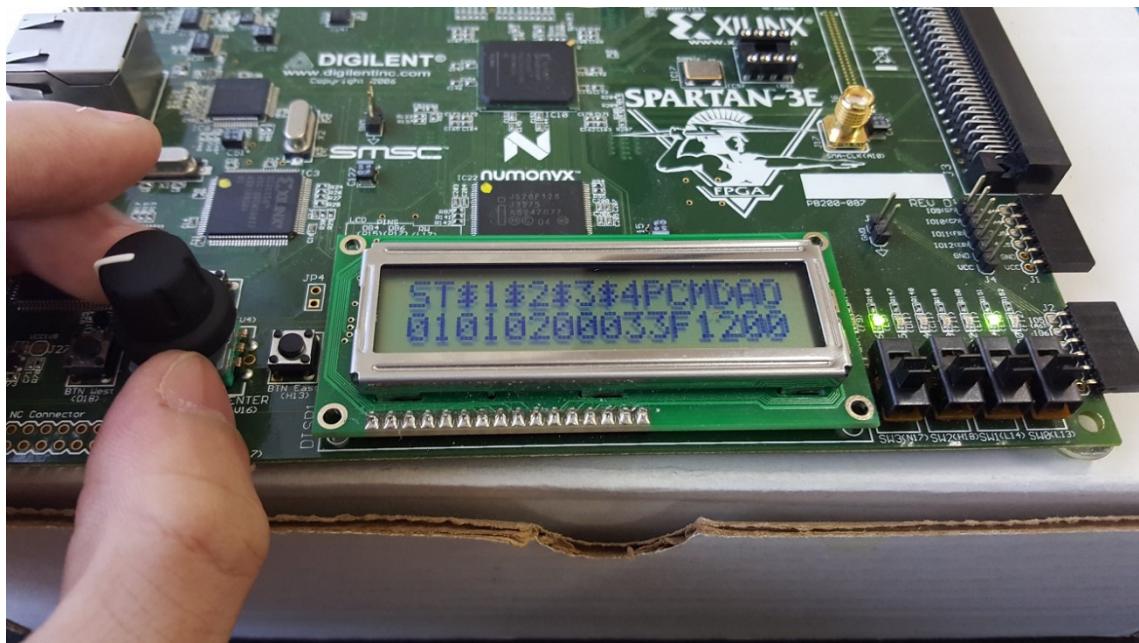


Vemos como el "Status" marca un 1, y el PC ha pasado a 3F, lo que significa que ha saltado una excepción, de valor 1 ("OPCode" inválido) y ha saltado hasta la dirección de excepciones.

Si miramos de nuevo la simulación:



Vemos que ocurre lo mismo, sólo que en simulación el registro 4 pasa a valer 3 un par de ciclos después, cosa que podemos comprobar si seguimos avanzando en la placa.



Por lo tanto se comprueba que el funcionamiento del programa es correcto.

En los LEDs también se puede ver en todo momento el valor de dichos parámetros, aunque es más sencillo mirarlos en el display.