

A dark blue vertical bar runs down the left side of the page. A blue arrow points to the right from this bar, containing the date.

24-5-2015

# Práctica 2

Apartados 2-6

Several thin, curved lines in dark blue and light grey originate from the bottom left corner and sweep upwards and to the right.

Guillermo Meléndez Morales y Susana Pineda De  
Luelmo

## Parte 2: Diseño Del Circuito De Cálculo de PCSrc.

El grupo de prácticas deberá implementar una versión estructural de la arquitectura del circuito llamado nextPC, que, tomando como entradas las señales BranchEQ, BranchNE, Jump, JumpReg, generadas por la unidad de control, más la señal Zero, generada por la UAL, calcula la señal PCSrc, que gobierna el multiplexor cuya salida se grabará en el PC.

Puesto que la señal PCSrc está formada por otras 5 señales, debemos tomar la tabla de verdad que forman estas señales y calcular el circuito que genera dicha señal.

Para ello contamos con la siguiente tabla de verdad:

JumpReg	Jump	BranchEQ	BranchNE	Zero	PCSrc
0	0	0	0	X	00 PC+4
0	0	1	0	1	01 PC ramificación (beq)
0	0	0	1	0	01 PC ramificación (bne)
0	1	X	X	X	10 Dirección de salto
1	X	X	X	X	11 Registro de enlace
Resto	De	Combinaciones			XX Valor indiferente

Los valores indiferentes dados por las combinaciones no registradas (última opción de la tabla) se tomarán como si la señal PCSrc estuviese a 00, aprovechando que esta señal es el PC incrementado, por lo que, si alguna combinación importante se hubiese quedado sin especificar en la tabla al menos continuaríamos con la instrucción siguiente.

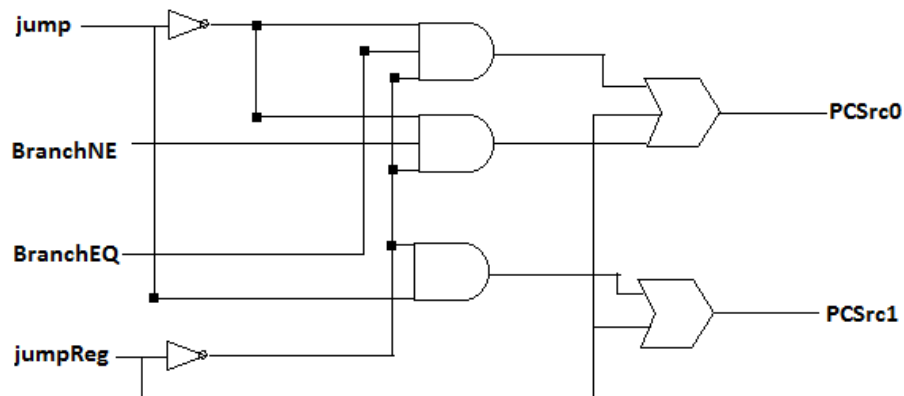
Las señales de entrada que vienen marcadas con una x se entienden como que, sea cual sea su valor, la salida va a ser la misma. Por ejemplo, en el caso de PCSrc “10” las señales que lo forman vienen dadas por “0” “1” “x” “x” “x” (en orden de aparición en la tabla) lo que quiere decir, que la señal PCSrc estará en PCSrc “10” con todas las combinaciones posibles de dichas x, como por ejemplo “0” “1” “1” “1” “1”.

Para crear el circuito que genere la señal PCSrc a partir de las otras señales, tomamos todos los valores para los que su señal de salida sea 1 y simplificamos, lo que nos da como resultado los dos bits de salida de la señal PCSrc:

$PCSrc0 = (\text{not\_Jump} \text{ and } \text{BranchEQ} \text{ and } \text{not\_JumpReg}) \text{ or } (\text{not\_Jump} \text{ and } \text{BranchNE} \text{ and } \text{not\_JumpReg}) \text{ or } \text{JumpReg}.$

$PCSrc1 = (\text{not\_JumpReg} \text{ and } \text{Jump}) \text{ or } \text{JumpReg}.$

El circuito combinacional que darían lugar estas dos ecuaciones booleanas sería el siguiente:



A continuación pasaremos estos resultados a VHDL para obtener el modelo de dicho circuito. El fragmento de código correspondiente a este circuito sería el siguiente:

```

architecture estructural of next_PC is
  signal not_JumpReg, not_Jump, not_BranchEQ, not_BranchNE, not_Zero:
    std_logic;
  signal s1, s2, s3, s4, s5 : std_logic;

begin
  inv0: entity work.not1 port map (JumpReg, not_JumpReg); --inversor
  JumpReg
  inv1: entity work.not1 port map (Jump, not_Jump); --inversor Jump
  inv2: entity work.not1 port map (BranchEQ, not_BranchEQ); --BranchEQ
  inv3: entity work.not1 port map (JumpReg, not_JumpReg); --JumpReg
  and0: entity work.and2 port map (not_JumpReg, Jump, s1); -- J*not_JR

  and1: entity work.and3 port map (not_Jump, BranchEQ, not_JumpReg, s2);
  and2: entity work.and3 port map (not_Jump, BranchNE, not_JumpReg, s3);
  or1: entity work.or3 port map (s2, s3, JumpReg, PCSrc(0));
  or0: entity work.or2 port map (JumpReg, s1, PCSrc(1));

end architecture estructural;

```

The screenshot displays the Waveform Editor interface. On the left, a signal hierarchy is shown with the following signals: `/test_nextpc/branch_eq`, `/test_nextpc/branch_ne`, `/test_nextpc/jump`, `/test_nextpc/jump_reg`, `/test_nextpc/zero`, `/test_nextpc/pcsrc_c`, `/test_nextpc/pcsrc_e`, and `/test_nextpc/pcsrc_o`. The main area shows a timing diagram with a yellow cursor at 58 ns. The bottom status bar indicates the time range from 0 ns to 468 ns.

En el cronograma podemos comprobar que la señal del test de la parte estructural coincide con la de comportamiento, a excepción de los valores indeterminados que aparecen en la tabla (salidas XX). En nuestro caso, dichas salidas se han tomado dependiendo de la tabla de verdad y de los mapas de Karnaugh para facilitar la simplificación del circuito.

Comprobamos que todas las salidas se corresponden con la tabla de verdad de la señal PCSrc, sobre todo las que no coinciden para asegurarnos de que dichas señales son aquellas que no están definidas y que todo lo demás coincide.

### Parte 3: Diseño de un Multiplexor de 4 entradas de n bits.

En este apartado, el grupo de prácticas deberá implementar una arquitectura estructural de un multiplexor de 4 entradas de ancho genérico. La entidad del circuito se llama `mux_4xn`, y está incorporada en el fichero `combinacionales.vhd`, donde además se presenta una arquitectura funcional del mismo.

La arquitectura estructural pedida se constituirá como una red de multiplexores 4 a 1.

También se realizará en este apartado un modelo estructural de un multiplexor de 2 a 1, cuya entidad se llama `mux_2x1` y se encuentra en el fichero `combinacionales.vhd`

Lo primero que vamos a hacer en este apartado va a ser la arquitectura estructural del multiplexor de 2 entradas de 1 bit utilizando puertas lógicas.

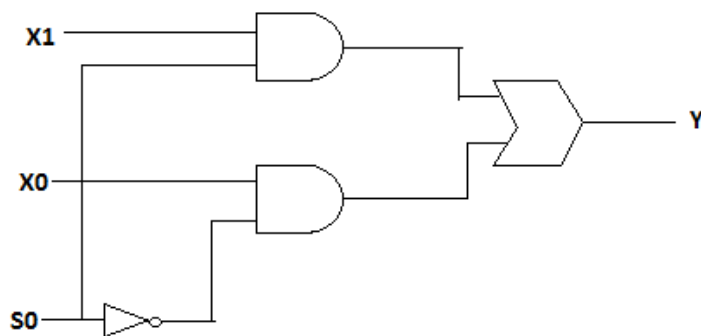
Para empezar haremos la tabla de verdad de dicho multiplexor, tomando como entradas `X1` y `X0` y `S0` como entrada de selección. Por otra parte tendremos `Y` como salida del multiplexor de un bit.

Tabla de verdad de multiplexor de 2x1:

X1	X0	S0	Y
0	0	0	0
0	0	1	0
0	1	0	1
0	1	1	0
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	1

Una vez tenemos la tabla de verdad simplificamos y obtenemos la ecuación booleana que forma dicho circuito:

$$Y = (X1 \text{ and } X0) \text{ or } (X0 \text{ and not\_}S0)$$

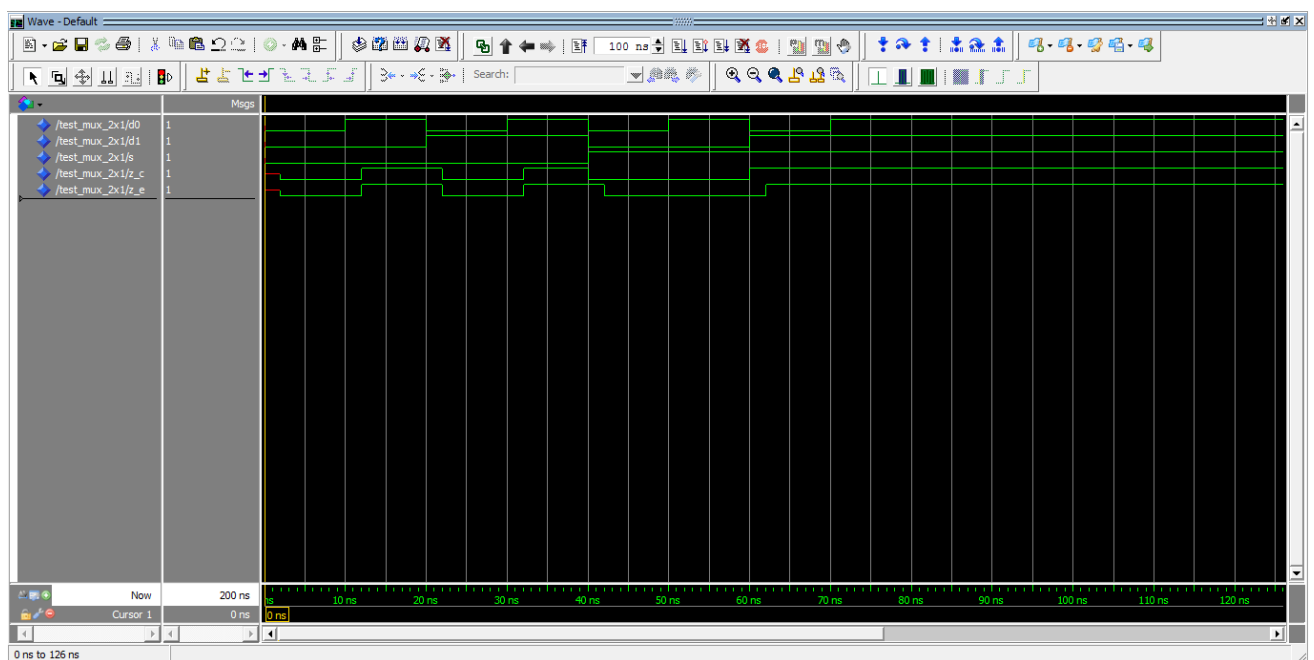


Una vez tenemos la ecuación booleana del circuito pasamos a implementarlo en VHDL, lo que nos da como resultado el siguiente fragmento de código:

```

architecture estructural of mux_2x1 is
  signal not_s: std_logic;
  signal a1, a2: std_logic;
begin
  inv0: entity work.not1 port map (s, not_s);
  and1: entity work.and2 port map (d0, not_s, a1);
  and2: entity work.and2 port map (d1, s, a2);
  or1: entity work.or2 port map (a1, a2, z);
end architecture estructural;
  
```

A continuación pasamos a comprobar el resultado del multiplexor, para ello analizamos las ondas generadas por el test del multiplexor de 2 a 1:



Comprobamos que las ondas son iguales según van cambiando las entradas a pesar de un leve retardo en la señal.

A continuación pasamos a implementar el multiplexor de 4 entradas de 1 bit utilizando puertas lógicas, para ello repetimos el mismo proceso de antes:

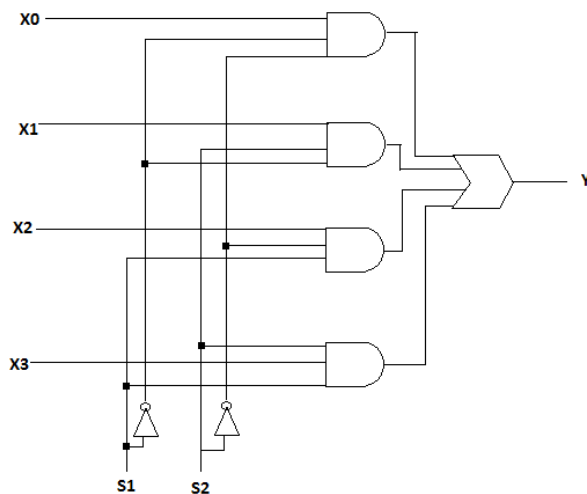
En primer lugar calculamos la tabla de verdad, simplificamos y calculamos la ecuación booleana que forma nuestro circuito:

Puesto que el circuito que buscamos cuenta con las entradas X0, X1, X2, X3 y las entradas de selección S0 y S1 e incluir la tabla de verdad sería muy farragoso, solo mostramos la ecuación resultante que nos da el comportamiento de un multiplexor de 4

a 1:

$$\overline{X0} \overline{S1} \overline{S0} + X1 \overline{S1} \overline{S0} + X2 S1 \overline{S0} + X3 S1 S0$$

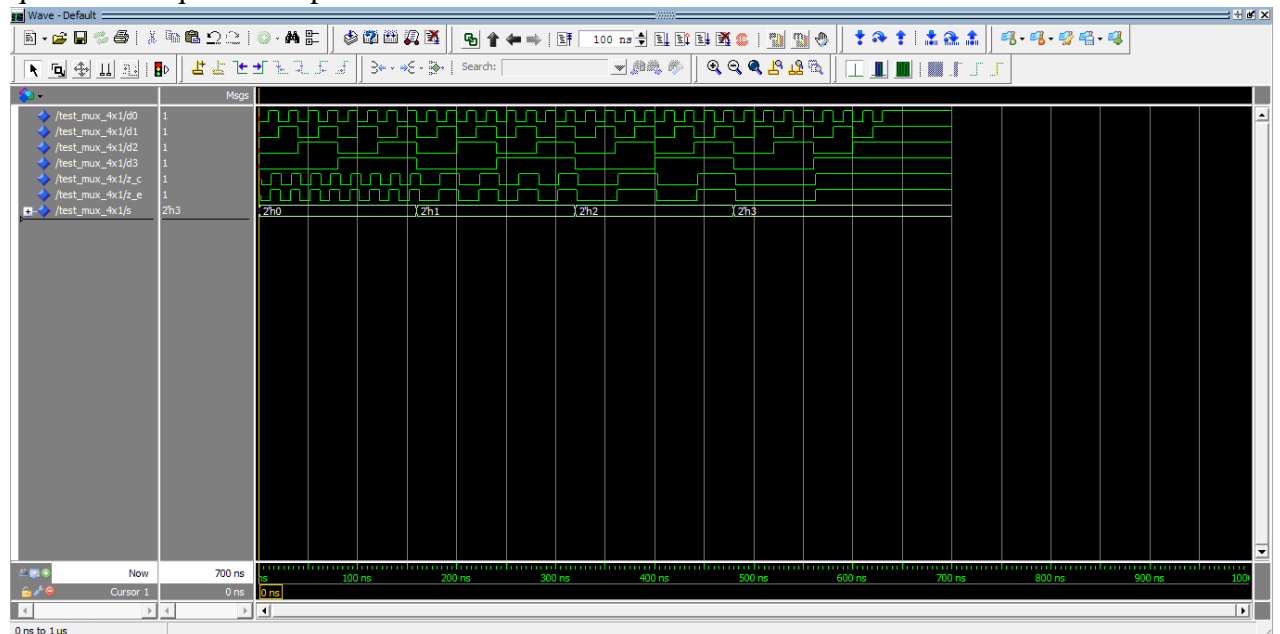
Y el circuito resultante:



Una vez obtenida la ecuación booleana que nos muestra el comportamiento del multiplexor de 4 a 1, pasamos a implementarlo en VHDL, obteniendo como resultado el siguiente fragmento de código:

```
architecture estructural of mux_4x1 is
signal not_s0, not_s1: std_logic;
signal a1, a2, a3, a4: std_logic;
begin
inv0: entity work.not1 port map (s(0), not_s0);
inv1: entity work.not1 port map (s(1), not_s1);
and1: entity work.and3 port map (d0, not_s0, not_s1, a1);
and2: entity work.and3 port map (d1, not_s1, s(0), a2);
and3: entity work.and3 port map (d2, s(1), not_s0, a3);
and4: entity work.and3 port map (d3, s(0), s(1), a4);
or1: entity work.or4 port map (a1, a2, a3, a4, z);
end architecture estructural;
```

Una vez obtenido el código que implementa nuestro circuito pasamos a comprobar el resultado mediante el test. Si las señales de comportamiento y estructural coinciden quiere decir que el comportamiento es el adecuado:



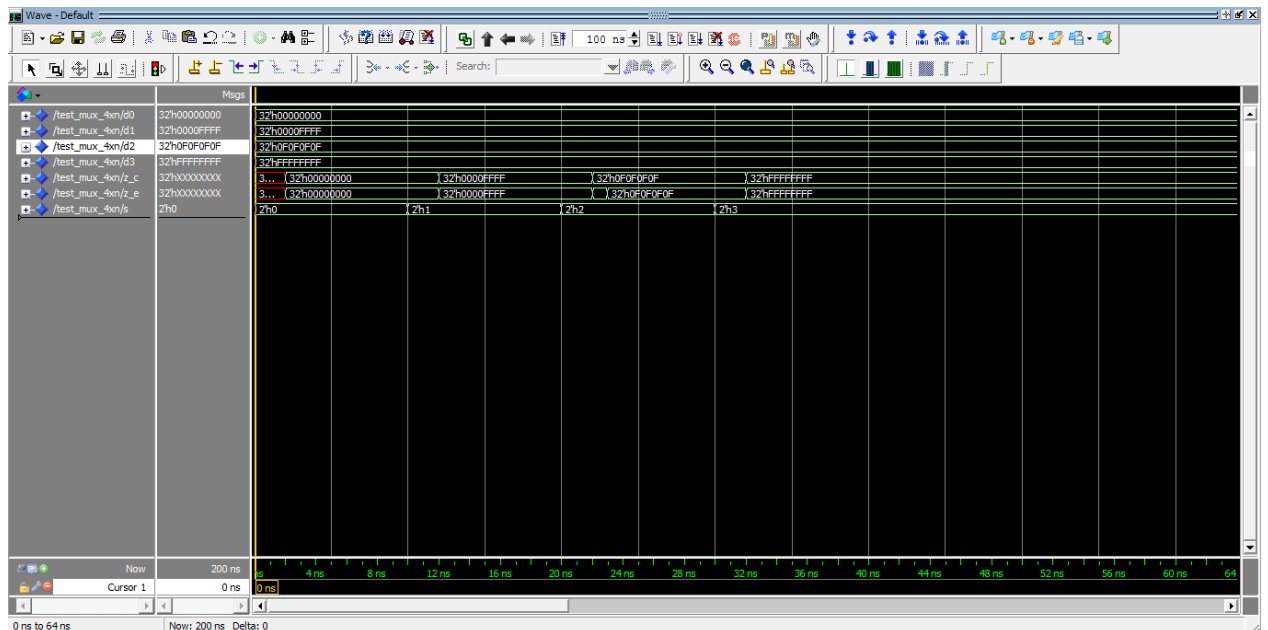
Puesto que las ondas coinciden y el comportamiento es el deseado, es decir, según van cambiando las entradas de selección a la salida muestra la entrada X correspondiente a dicha combinación de entradas de selección, damos por correcto el comportamiento del multiplexor de 4 a 1.

Por último se nos pide que escribamos una estructura combinacional para el multiplexor de 4 entradas de n bits que utilice la arquitectura estructural del multiplexor de 4 a 1. Para ello encadenaremos los multiplexores de forma que las entradas cuenten con n bits de ancho. Esta estructura queda reflejada en el siguiente fragmento de código que nos muestra el comportamiento de los multiplexores encadenados:

```
architecture estructural of mux_4xn is
    component mux_4x1 is
        generic (retardo_base: time := 1 ns);
        port
            (d0, d1,
             d2, d3: in std_logic;
             s:      in std_logic_vector(1 downto 0);
             z:      out std_logic);
    end component mux_4x1;
BEGIN
mux_4xn:
for n in 0 to ancho_datos-1 generate
mux: mux_4x1
port map (d0 => d0 (n),
          d1 => d1 (n),
          d2 => d2 (n),
          d3 => d3 (n),
          z  => z  (n),
          s  => s);
end generate mux_4xn;
end architecture estructural;
```



Por ultimo para comprobar el funcionamiento del multiplexor utilizaremos el test proporcionado. Dicho test genera las siguientes ondas que tenemos que analizar:



**Nota: captura de pantalla adjunta en el archivo de prácticas.**

Finalmente, comprobamos que las señales de comportamiento y estructural coincidan. Una vez comprobado esto, y que el resultado es correcto, podemos dar por finalizada la implementación del multiplexor de 4 entradas de n bits de ancho cada una. El multiplexor funciona correctamente a excepción de un pequeño fallo que da en la opción 2 (entradas de selección “10”) en el que se pone la salida a 0 antes de mostrar el resultado correcto, pero dentro del mismo ciclo, por lo que no afecta al comportamiento final del multiplexor.

### Ventana Transcript de la simulación:

```
Transcript
# // INFORMATION THAT ARE PRIVILEGED, CONFIDENTIAL, AND EXEMPT FROM
# // DISCLOSURE UNDER THE FREEDOM OF INFORMATION ACT, 5 U.S.C. SECTION 552.
# // FURTHERMORE, THIS INFORMATION IS PROHIBITED FROM DISCLOSURE UNDER
# // THE TRADE SECRETS ACT, 18 U.S.C. SECTION 1905.
# //
# // NOT FOR CORPORATE OR PRODUCTION USE.
# // THE ModelSim PE Student Edition IS NOT A SUPPORTED PRODUCT.
# // FOR HIGHER EDUCATION PURPOSES ONLY
# //
# vsim -gui
# Start time: 16:53:16 on May 25,2015
# Loading std.standard
# Loading std.textio(body)
# Loading ieee.std_logic_1164(body)
# Loading ieee.numeric_std(body)
# Loading work.mips_settings
# Loading work.test_mux_4xn(test)
# Loading work.mux_4xn(comportamiento)
# Loading work.mux_4xn(estructural)
# Loading work.mux_4x1(estructural)
# Loading work.not1(fd)
# Loading work.and3(fd)
# Loading work.or4(fd)
# ** Warning: Design size of 12178 statements or 259 leaf instances exceeds ModelSim PE Student Edition recommen
ded capacity.
# Expect performance to be quite adversely affected.
add wave sim:/test_mux_4xn/*
VSIM 3> run
run
VSIM 4> run

VSIM 4>]
```

## Parte 4: Diseño de un sumador de dos números de n bits.

**El grupo de prácticas deberá implementar una versión estructural de la arquitectura de un sumador de dos números de n bits.**

Lo primero que vamos a hacer en este apartado es Escribir una arquitectura estructural para el semisumador. Para ello, realizamos una tabla de verdad que recoja el comportamiento del semisumador, simplificamos y mostramos como un circuito para facilitar su comprensión.

Nuestro semisumador contará con dos entradas A y B que serán los dos operandos de la suma, y dos salidas S que mostrará el resultado de la suma y Cout que nos indicará si se ha producido desbordamiento en dicha operación.

Tabla de verdad semisumador:

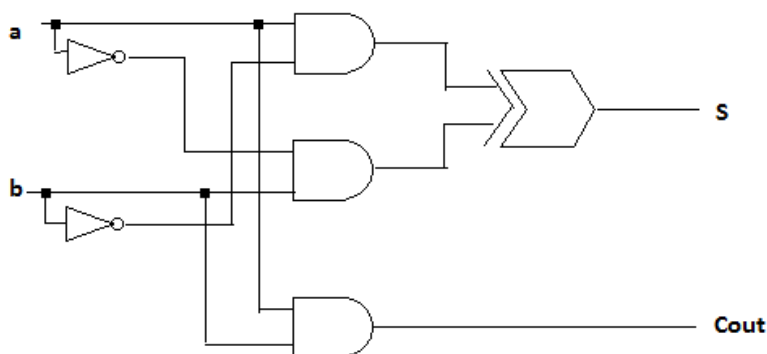
A	B	S	Cout
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	0

Expresión booleana resultante:

$$S = \overline{a}b + a\overline{b}$$

$$Cout = ab$$

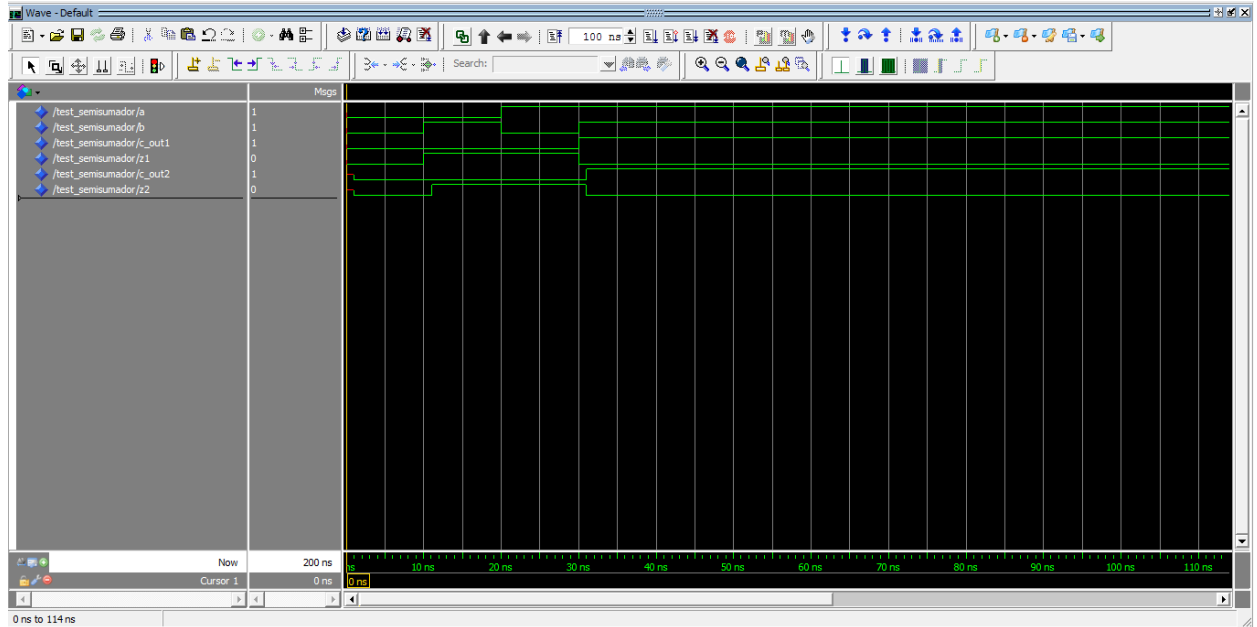
Circuito resultante:



Una vez que tenemos la expresión booleana que forma nuestro semisumador, pasaremos a implementarlo en VHDL, dando como resultado el siguiente fragmento de código:

```
architecture estructural of semisumador is
begin
xor1: entity work.xor2 port map (a, b, z);
and1: entity work.and2 port map (a, b, c_out);
end estructural;
```

Una vez implementado el circuito en VHDL pasamos a comprobar su funcionamiento mediante el test proporcionado en la práctica. Para ello compararemos las ondas obtenidas en la señal de comportamiento y en la estructural, ambas señales deben coincidir y mostrar el resultado de la suma aritmética de las dos entradas, y en el caso de existir desbordamiento, el desbordamiento producido.



Podemos observar que las ondas coinciden y que muestran en resultado esperado. A pesar de un pequeño retardo en la segunda representación del semisumador podemos dar por buena la implementación.

Una vez terminada la implementación de un semisumador pasaremos a implementar un sumador completo, para ello, repetiremos los mismos pasos que en el apartado anterior. Primero, calcularemos la tabla de verdad, simplificaremos y obtendremos la ecuación booleana que corresponda a nuestro circuito, seguidamente se mostrará un esquema de la forma del circuito, se mostrará el circuito implementado en VHDL, y por último se comprobará su funcionamiento mediante el test proporcionado para la práctica.

El sumador contará con las entradas A y B que serán los operandos de la suma, y Cin que será el acarreo de la suma anterior. Por otra parte contará con las salidas Z que será el resultado de la suma, y por otro la salida Cout que será la señal encargada de mostrar si hay acarreo o no.

Tabla de verdad del sumador completo:

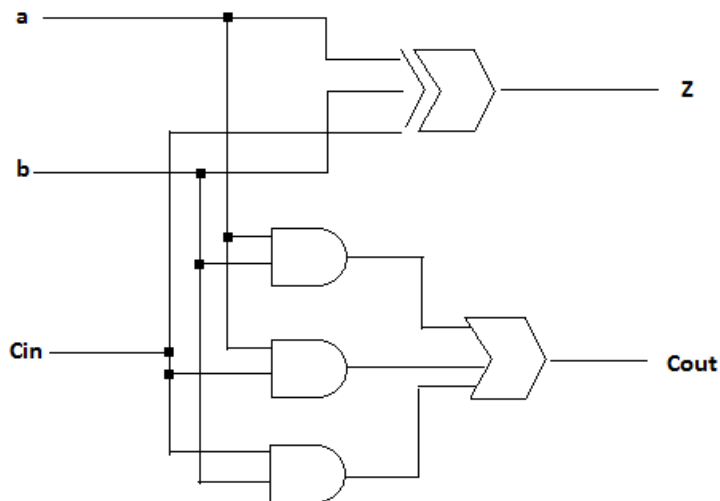
A	B	Cin	Z	Cout
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

Expresión booleana resultante:

$$Z = a \oplus b \oplus \text{Cin}$$

$$\text{Cout} = ab + a\text{Cin} + b\text{Cin}$$

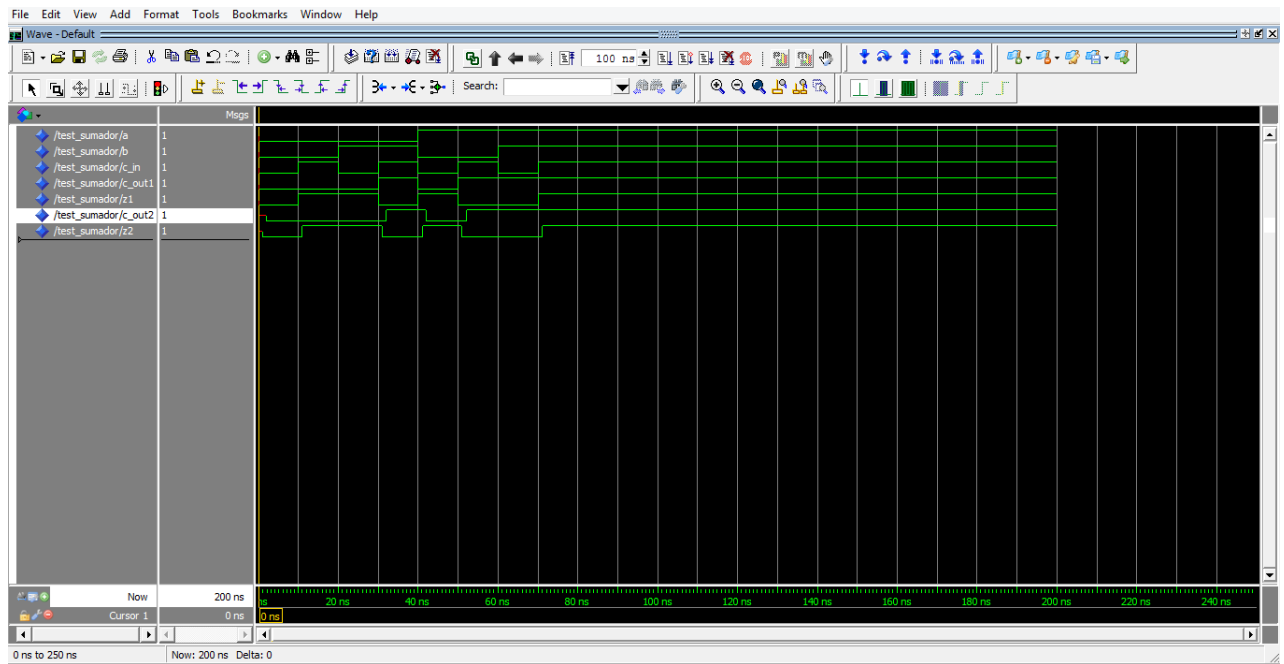
Circuito final:



Una vez calculado el circuito y su ecuación booleana, pasamos a implementarlo en VHDL dando como resultado el siguiente fragmento de código:

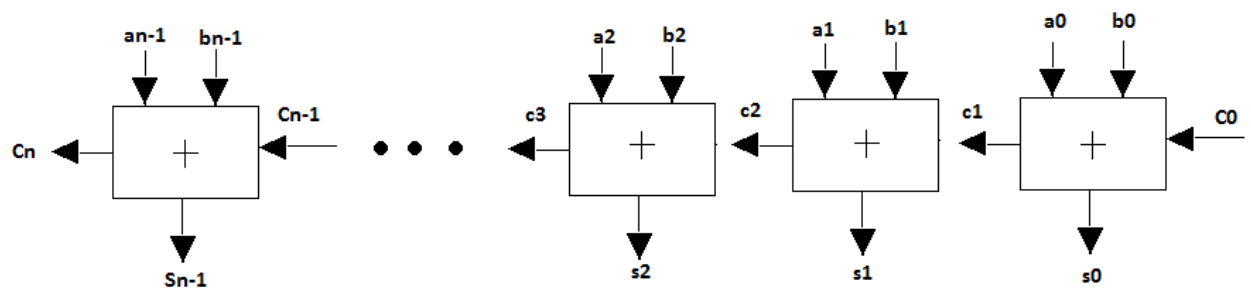
```
architecture estructural of sumador is
  signal a1, a2, a3: std_logic;
begin
  xor1: entity work.xor3 port map (a, b, c_in, z);
  and1: entity work.and2 port map (a, b, a1);
  and2: entity work.and2 port map (a, c_in, a2);
  and3: entity work.and2 port map (b, c_in, a3);
  or1: entity work.or3 port map (a1, a2, a3, c_out);
end estructural;
```

Una vez implementado el código en VHDL pasamos a ejecutarlo y a comprobar las señales que nos da como resultado.



Como podemos comprobar, las señales de las salidas coinciden y muestran lo esperado tras el cálculo de la tabla de verdad. A pesar de un pequeño retardo en las salidas los resultados son los esperados por lo que se puede dar por buena la implementación del sumador.

Una vez implementado el sumador de 1 bit pasamos a implementar el sumador de  $n$  bits de ancho. Para ello encadenaremos los sumadores de 1 bit de forma que podamos sumar dos operandos de  $n$  bits de ancho. Siguiendo el siguiente esquema:

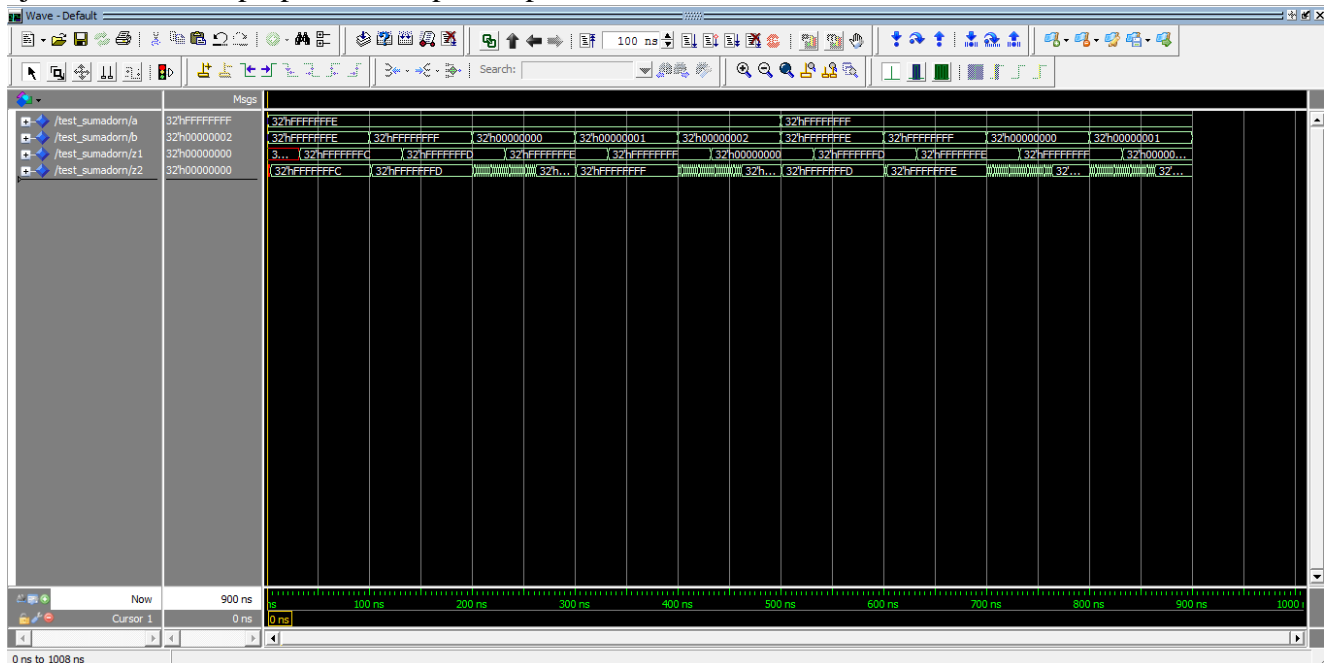


Obtendremos así un sumador con dos operandos de entrada de  $n$  bits cada uno y un resultado de  $n$  bits.

Para implementar dicho circuito empleamos la siguiente estructura en VHDL dando como resultado un sumador de n bits. Dicho fragmento de código es el siguiente:

```
architecture estructural of sumadorN is
component sumador is
    generic (retardo_base: time := 1 ns);
    port    (a, b, c_in: in  std_logic;
             c_out, z:  out std_logic);
end component sumador;
signal c: std_logic_vector (ancho downto 1);
BEGIN
sumador1:
component sumador
    port map (a => a(0),
             b => b(0),
             c_in => '0',
             z => z(0),
             c_out => c (1));
sumadorn: for n in 1 to ancho-1 generate
sum: sumador
port map (a => a (n),
         b => b (n),
         c_in => c (n),
         z => z (n),
         c_out => c (n+1));
end generate sumadorn;
end architecture estructural;
```

Una vez implementado en VHDL pasamos a comprobar los resultados mediante la ejecución del test proporcionado para la práctica:

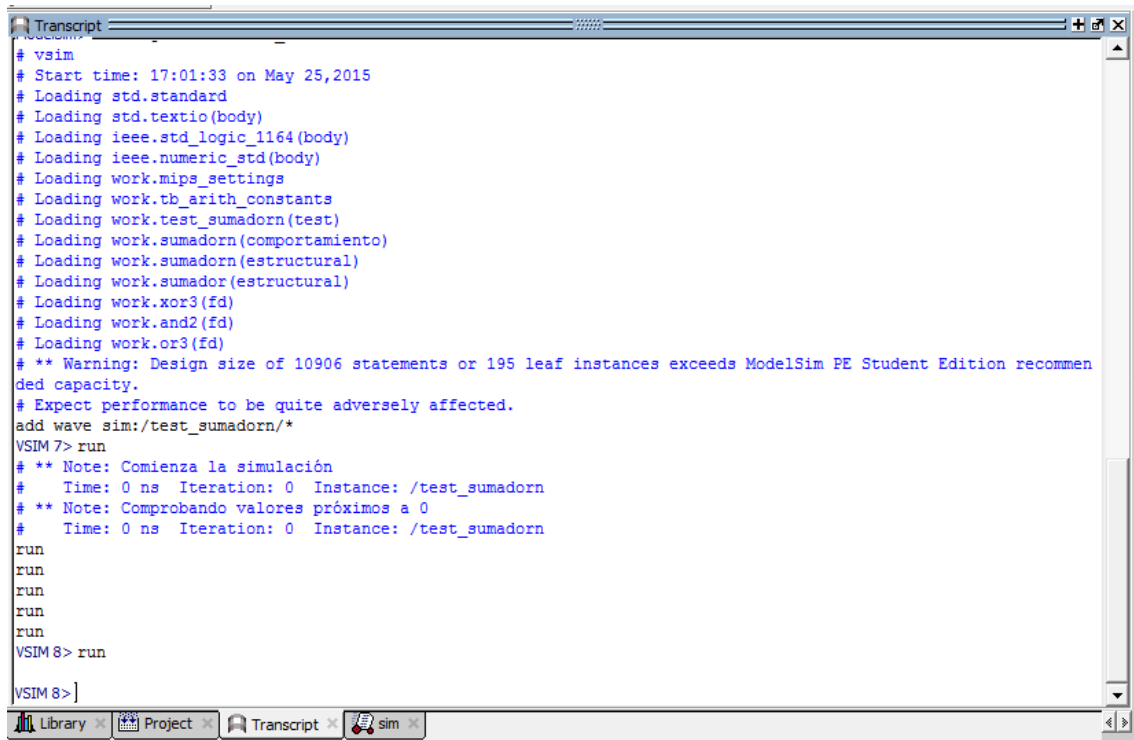


**Nota: imagen adjunta a la práctica.**

Como podemos comprobar en la captura de pantalla de las ondas, estas coinciden con el resultado esperado a pesar de unas pequeñas variaciones que acaban por estabilizarse, por lo que no afectan al funcionamiento general del circuito.

Una vez comprobado el comportamiento del circuito puede darse por finalizada la implementación del sumador de n bits de ancho.

### **Ventana Transcript de la simulación:**



```
# vsim
# Start time: 17:01:33 on May 25,2015
# Loading std.standard
# Loading std.textio(body)
# Loading ieee.std_logic_1164(body)
# Loading ieee.numeric_std(body)
# Loading work.mips_settings
# Loading work.tb_arith_constants
# Loading work.test_sumadorn(test)
# Loading work.sumadorn(comportamiento)
# Loading work.sumadorn(estructural)
# Loading work.sumador(estructural)
# Loading work.xor3(fd)
# Loading work.and2(fd)
# Loading work.or3(fd)
# ** Warning: Design size of 10906 statements or 195 leaf instances exceeds ModelSim PE Student Edition recommended capacity.
# Expect performance to be quite adversely affected.
add wave sim:/test_sumadorn/*
VSIM 7> run
# ** Note: Comienza la simulación
# Time: 0 ns Iteration: 0 Instance: /test_sumadorn
# ** Note: Comprobando valores próximos a 0
# Time: 0 ns Iteration: 0 Instance: /test_sumadorn
run
run
run
run
run
run
VSIM 8> run

VSIM 8>]
```

## Parte 5: Diseño de una Unidad Aritmético lógica de 32 bits.

**El grupo de prácticas deberá implementar una versión estructural de la arquitectura de una unidad aritmético lógica de 32 bits que realice las operaciones indicadas en la tabla correspondiente.**

La unidad aritmético lógica de 32 bits estará formada por unidades aritmético lógicas de un solo bit de ancho unidas de forma que creen una ALU de 32 bits.

Para comenzar implementamos la ALU de 1 bit que utilizaremos como base para nuestro circuito.

Lo primero que debemos hacer, es plantear una tabla de verdad que recoja las diferentes combinaciones y resultados que soporta nuestra ALU, para ello contamos con la tabla de verdad de la ALU.

Tabla de verdad de la Unidad Aritmético Lógica.

ALUop	Operación Realizada
000	A and B
001	A or B
010	A xor B
011	A nor B
100	A plus B
101	A minus B
110	A slt B
111	A sltu B

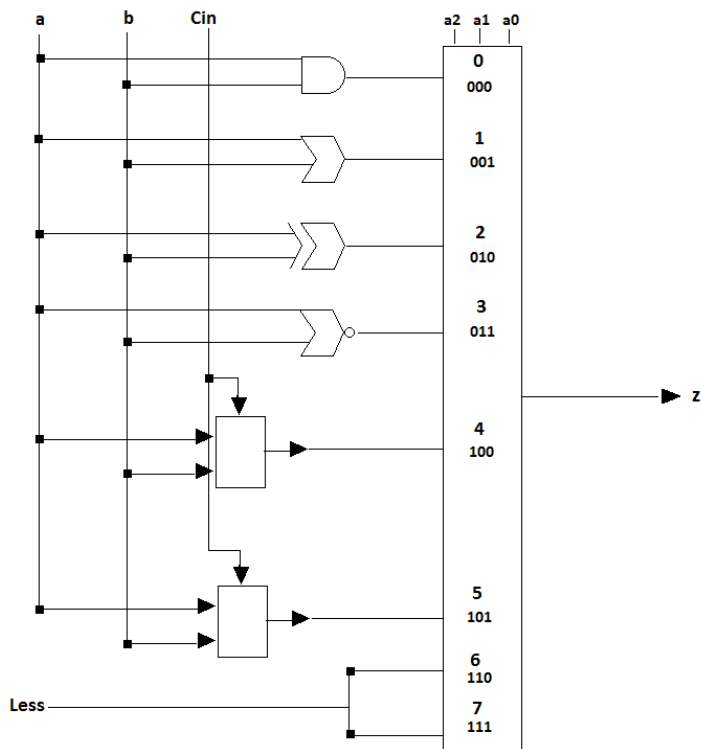
Comenzamos con las operaciones lógicas ( and, or, xor, nor) cuya implementación va a ser muy sencilla, ya que únicamente tenemos que conectar las entradas a una puerta de dicho tipo y las salidas a un multiplexor que se active mediante la señal ALUop y nos marque que señal debe salir en cada momento.

Por otra parte, tendremos un circuito paralelo que nos muestre la salida C\_Out en los casos de suma, resta y slt y sltu, y en el caso de and, or , xor y nor ponga dicha salida a 0.

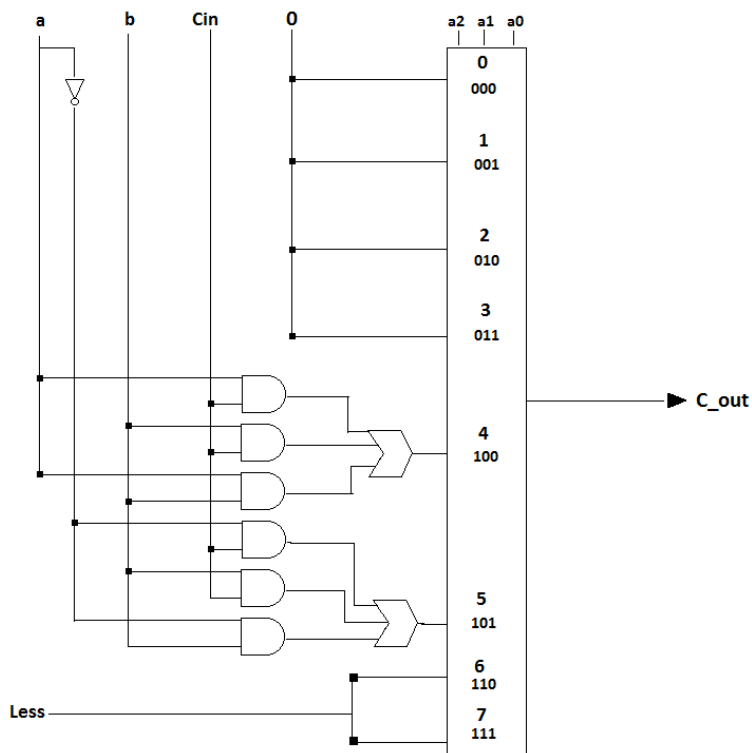


Un esquema simplificado de la ALU1 sería el siguiente:

Circuito de salida Z:



Circuito de Salida C\_out:



Una vez tenemos el circuito que vamos a implementar de forma esquemática pasamos a implementarlo en VHDL para poder comprobar el resultado.

Dicha implementación da como resultado el siguiente fragmento de código:

architecture estructural of ALU1 is

```
signal not_b, not_ALU_Operation0, not_ALU_Operation1, not_ALU_Operation2, not_cin,
not_cout: std_logic;
signal c1, c2, c3, c4, c5, c6, cout, ac, not_d6, not_c6: std_logic;
signal d1, d2, d3, d4, d5, d6: std_logic; -- c_out
signal s1, s2, s3, s4, s5, s6, s7, s8, s9, s10: std_logic;
signal e1, e2, e3, e4, e5, e6, e7, e8, e9, e10: std_logic; -- c_out
begin

inv2:      entity work.not1      port map (ALU_Operation (0), not_ALU_Operation0);
inv3:      entity work.not1      port map (ALU_Operation (1), not_ALU_Operation1);
inv4:      entity work.not1      port map (ALU_Operation (2), not_ALU_Operation2);
inv5:      entity work.not1      port map (c_in, not_cin);

-- And (Aluop => 000)
and1:      entity work.and2      port map (a, b, c1);
-- Or  (Aluop => 001)
or1:      entity work.or2        port map (a, b, c2);
-- Xor (Aluop => 010)
xor1:      entity work.xor2      port map (a, b, c3);
-- Nor (Aluop => 011)
nor1:      entity work.nor2      port map (a, b, c4);
-- Add (Aluop => 100)
add1:      entity work.sumador   port map (a, b, c_in, ac, c5);
add1c:     entity work.sumador   port map (a, b, c_in, d5, ac);
-- Sub (Aluop => 101)
ca2:      entity work.xor2      port map (b, '1', not_b);
sub1:      entity work.sumador   port map (a, not_b, not_cin, ac, c6);
sub1c:     entity work.sumador   port map (a, not_b, c_in, d6, ac);
inver:     entity work.not1      port map (c6, not_c6);
d1 <= ALU_Operation (2);
d2 <= ALU_Operation (2);
d3 <= ALU_Operation (2);
d4 <= ALU_Operation (2);

-- Multiplexor
and000:     entity work.and4      port map (c1, not_ALU_Operation2,
not_ALU_Operation1, not_ALU_Operation0, s1);
and000c:    entity work.and4      port map (d1, not_ALU_Operation2,
not_ALU_Operation1, not_ALU_Operation0, e1);

and001:     entity work.and4      port map (c2, not_ALU_Operation2,
not_ALU_Operation1, ALU_Operation (0), s2);
and001c:    entity work.and4      port map (d2, not_ALU_Operation2,
not_ALU_Operation1, ALU_Operation (0), e2);

and010:     entity work.and4      port map (c3, not_ALU_Operation2, ALU_Operation
(1), not_ALU_Operation0, s3);
and010c:    entity work.and4      port map (d3, not_ALU_Operation2, ALU_Operation
(1), not_ALU_Operation0, e3);

and011:     entity work.and4      port map (c4, not_ALU_Operation2, ALU_Operation
(1), ALU_Operation (0), s4);
and011c:    entity work.and4      port map (d4, not_ALU_Operation2, ALU_Operation
(1), ALU_Operation (0), e4);

and100:     entity work.and4      port map (c5, ALU_Operation (2),
not_ALU_Operation1, not_ALU_Operation0, s5);
and100c:    entity work.and4      port map (d5, ALU_Operation (2),
not_ALU_Operation1, not_ALU_Operation0, e5);

and101:     entity work.and4      port map (not_c6, ALU_Operation (2),
not_ALU_Operation1, ALU_Operation (0), s6);
and101c:    entity work.and4      port map (d6, ALU_Operation (2),
not_ALU_Operation1, ALU_Operation (0), e6);

and110:     entity work.and4      port map (less, ALU_Operation (2), ALU_Operation
(1), not_ALU_Operation0, s7);
```

```

and110c:      entity work.and4      port map (d6, ALU_Operation (2), ALU_Operation (1),
not_ALU_Operation0, e7);

and111:      entity work.and4      port map (less, ALU_Operation (2), ALU_Operation
(1), ALU_Operation (0), s8);
and111c:      entity work.and4      port map (d6, ALU_Operation (2), ALU_Operation (1),
ALU_Operation (0), e8);

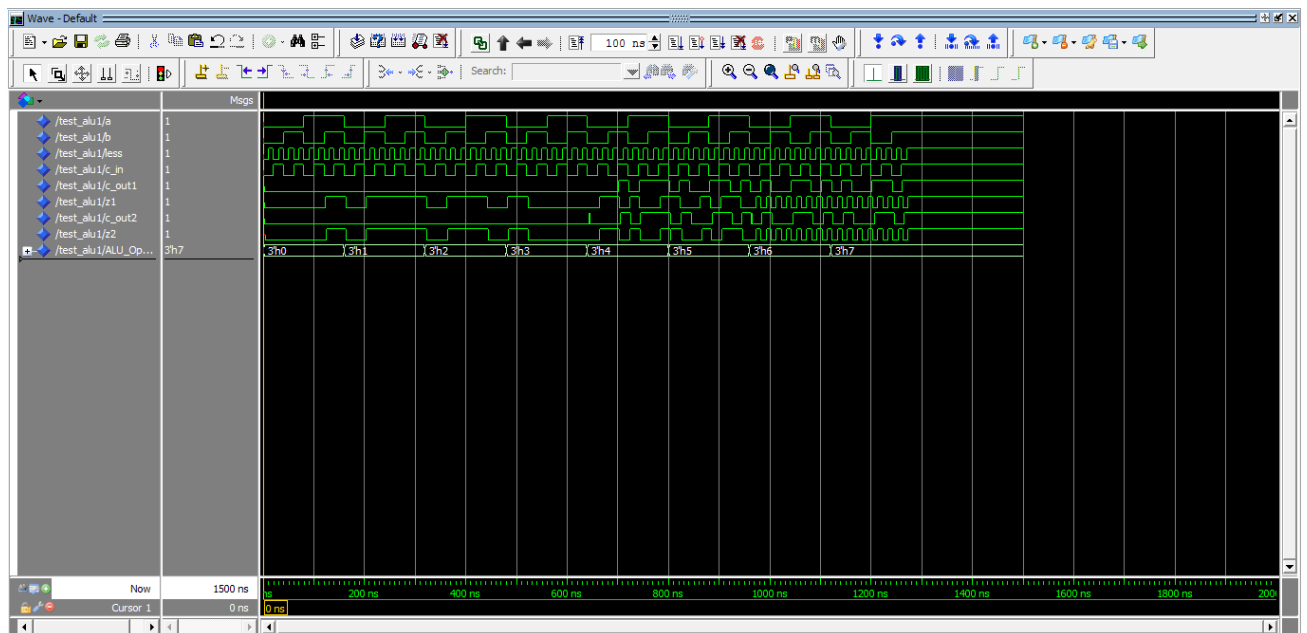
orfin1:      entity work.or4      port map ( s1, s2, s3, s4, s9);
orfin1c:      entity work.or4      port map ( e1, e2, e3, e4, e9);

orfin2:      entity work.or4      port map ( s5, s6, s7, s8, s10);
orfin2c:      entity work.or4      port map ( e5, e6, e7, e8, e10);
orfinal:      entity work.or2      port map (s9, s10, z);
orfinalc:      entity work.or2      port map (e9, e10, c_out);
end architecture estructural;

```

Una vez tenemos el código, pasamos a comprobar que las ondas muestran el resultado esperado. Para ello, por una parte tenemos que comprobar que las ondas de la arquitectura de comportamiento y estructural coinciden y además hay que comprobar que el resultado obtenido es el deseado, es decir, que se realizan las operaciones pedidas por ALUop y que estas se realizan de forma correcta.

A continuación se muestra el cronograma que obtenemos al simular el test ofrecido para comprobar la práctica:



**Nota: captura incluida en el archivo de la práctica**

Como podemos comprobar, las ondas coinciden a pesar de algún pico que no afecta al funcionamiento general de la ALU.

Por una parte observamos que las operaciones “000” “001” “010” y “011” muestran en la salida Z el resultado de la operación lógica correspondiente y en C\_out tiene una salida constante de 0, ya que en estas operaciones no tenemos en cuenta el acarreo de entrada ni de salida.

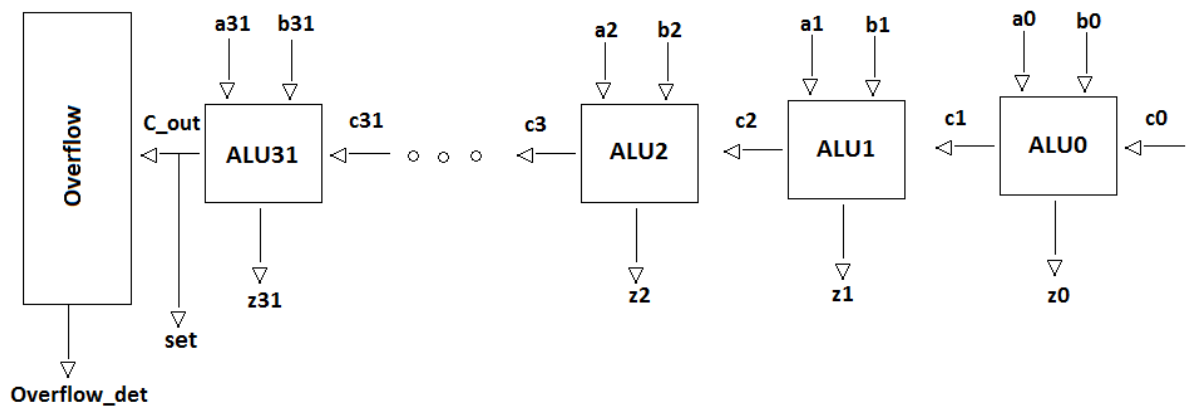
Por otra parte observamos que el resto de operaciones ponen en funcionamiento las dos salidas. La salida Z muestra el resultado de la operación aritmética realizada y la salida C\_out muestra el acarreo producido por dicha operación.

Nos aseguramos de que las operaciones muestran los resultados esperados y que las operaciones son las correspondientes a las entradas de ALUop. Una vez comprobado esto, pasamos a implementar la ALU de 32 bits.

Para la implementación de la ALU de 32 bits encadenaremos ALUs de un bit para conseguir el ancho deseado. Al final del encadenamiento de ALUs contaremos con un circuito, Overflow, encargado de detectar si se ha producido desbordamiento en la suma o en la resta aritmética, además de para esta función, el circuito Overflow nos va a servir para detectar si un número es menos que otro, lo cual utilizaremos para las operaciones slt y sltu.

Por otra parte, la ALU32 contará con una señal de salida adicional (Set) que nos indicará el signo del resultado.

A continuación se muestra un esquema del encadenamiento que se va a realizar de las ALUs.



Una vez tenemos el esquema del circuito que vamos a implementar, pasamos a implementarlo en VHDL. Para ello, utilizamos la estructura *generate* para las ALUs de la 0 a la 30 y después se implementará la ALU31 de forma separada para incluir la señal set y el detector de desbordamiento.

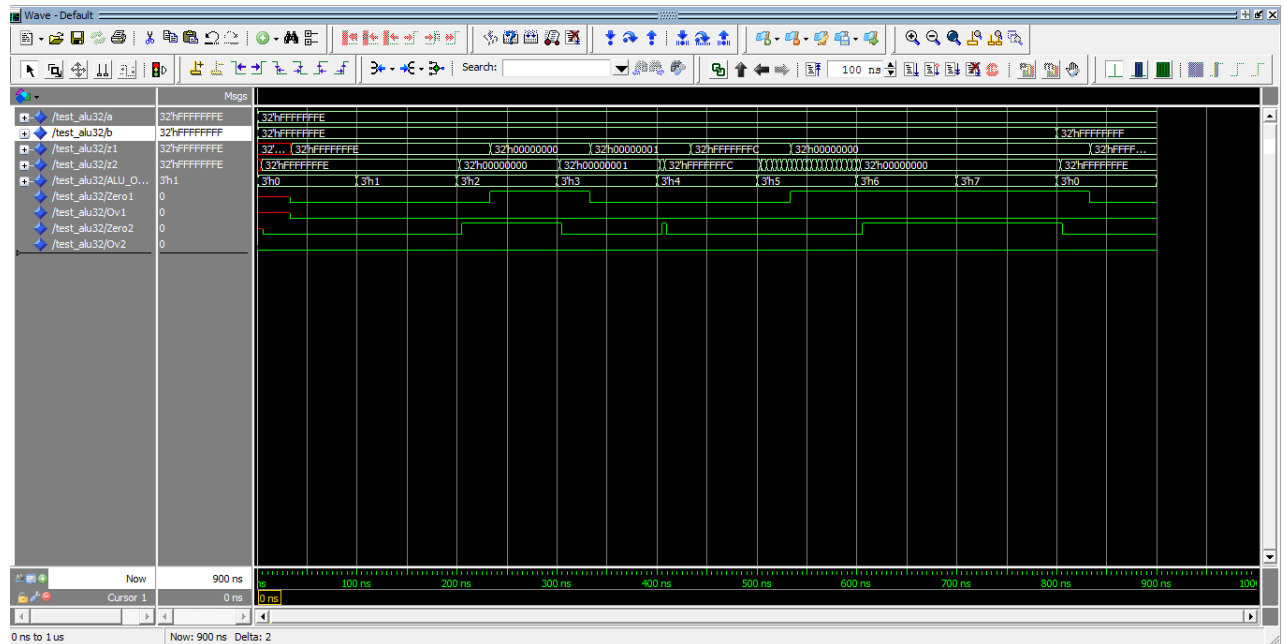
A continuación mostramos el fragmento de código que representa dicho circuito:

architecture estructural of ALU32 is

```
    component ALU1 is
    generic (retardo_base : time := 1 ns);
    port      (a      : in  std_logic;
               b      : in  std_logic;
               less   : in  std_logic;
               c_in   : in  std_logic;
               ALU_Operation: in  std_logic_vector      (2 downto 0);
               c_out  : out std_logic;
               z      : out std_logic);
    end component ALU1;
    signal c: std_logic_vector (32 downto 1);
    signal      Result, b, not_ALU_Src2, not_Result: std_logic;
BEGIN

    resta:          entity work.not1 port map (ALU_Src2 (31),
    not_ALU_Src2);
    resta1:         entity work.semisumador port map (ALU_Src1(31) ,
    not_ALU_Src2, b, Result);
    inv1:           entity work.not1 port map (Result, not_Result);
    ALUo:
    component ALU1
        port map (a => ALU_Src1 (0),
                  b => ALU_Src2 (0),
                  less => not_Result ,
                  c_in => '0',
                  ALU_Operation => ALU_Operation,
                  c_out => c (1),
                  z => ALU_Result (0));
    ALU31: for n in 1 to 31 generate
    ALU30: ALU1
        port map (a => ALU_Src1 (n),
                  b => ALU_Src2 (n),
                  less => '0',
                  c_in => c (n),
                  ALU_Operation => ALU_Operation,
                  c_out => c (n+1),
                  z => ALU_Result (n));
    end generate ALU31;
    Overflow <= '0';
end architecture estructural;
```

Una vez que tenemos el código y comprobamos que compila, pasamos a simularlo en ModelSim y comprobamos los resultados.



Como podemos observar las señales coinciden a excepción de la resta que en vez de restar va añadiendo ceros a la derecha, por lo que, al realizar la simulación global del circuito, no vamos a utilizar esta ALU, ya que causa un fallo general en todo el circuito.

```

# Loading work.alu32(comportamiento)
# Loading work.alu32(estructural)
# Loading work.not1(fd)
# Loading work.semisumador(estructural)
# Loading work.xor2(fd)
# Loading work.and2(fd)
# Loading work.alu1(estructural)
# Loading work.or2(fd)
# Loading work.nor2(fd)
# Loading work.sumador(estructural)
# Loading work.xor3(fd)
# Loading work.or3(fd)
# Loading work.and4(fd)
# Loading work.or4(fd)
# ** Warning: Design size of 40960 statements or 1835 leaf instances exceeds ModelSim PE Student Edition recommended capacity.
# Expect performance to be quite adversely affected.
add wave sim:/test_alu32/*
VSI11> run
# ** Note: Comienza la simulación
# Time: 0 ns Iteration: 0 Instance: /test_alu32
# ** Note: Comprobando valores próximos a 0
# Time: 0 ns Iteration: 0 Instance: /test_alu32
# ** Warning: NUMERIC_STD."-": metavalu detected, returning FALSE
# Time: 0 ns Iteration: 0 Instance: /test_alu32/UUT1
run
run
run
# ** Warning: NUMERIC_STD.TO_SIGNED: vector truncated
# Time: 400 ns Iteration: 0 Instance: /test_alu32
run
# ** Warning: NUMERIC_STD.TO_SIGNED: vector truncated
# Time: 500 ns Iteration: 0 Instance: /test_alu32
run
# ** Note: ERROR en t = 600 ns\n      a = -2 / b = -2 / Op = 5 / R1 = 0 / zero1 = '1' / ov1 = '0' / R2 = 2097151 / zero2 = '0' / ov2 = '0'
# Time: 600 ns Iteration: 0 Instance: /test_alu32
# ** Warning: NUMERIC_STD.TO_SIGNED: vector truncated
# Time: 600 ns Iteration: 0 Instance: /test_alu32
run
# ** Warning: NUMERIC_STD.TO_SIGNED: vector truncated
# Time: 700 ns Iteration: 0 Instance: /test_alu32
run

```

## PARTE 6: Incorporación de los modelos estructurales al camino de datos unicity.

En este apartado, el grupo de prácticas analizará el comportamiento del camino de datos para el caso de prueba caso2.asm, pero incorporando los modelos estructurales realizados en los pasos 2 al 5 de esta práctica en sustitución de los correspondientes modelos funcionales utilizados en el apartado 1.

Para analizar el comportamiento del camino de datos con la nueva estructura, compararemos las ondas generadas en el primer apartado de la práctica con las generadas en este último paso.

Para ello comenzaremos analizando el circuito con la representación estructural con la de comportamiento.

El caso que vamos a utilizar para comprobar el resultado de los apartado del 2 al 5 es el caso2 de la parte 1 de esta práctica. El código correspondiente en MIPS es el siguiente:

```
.data
dato:      .word 4
res:       .space      4
.text
main:      lw      $s0,dato
           or      $s1,$zero,$zero
           ori     $t0,$zero,0
           xori    $t0,$zero,0
bucle:     beq     $t0,$s0,e2
           e2:
           sltiu   $s0, $s0, 1
           slti    $s0, $s0, 1

           addiu   $s1, $s1, 1
           bne     $s1, $zero, e
           e:
           sltu    $s1, $s1, $zero
           sub     $s1, $s1, $zero
           subu    $s1, $s1, $zero
           xor     $s1, $zero, $s1
           slt     $s1, $zero, $s1
           nor     $s1, $zero, $s1
           and     $s1, $s1, $zero
           addu    $s1, $s1, $zero

           add     $s1,$t0,$s1
           andi    $s1, $s1, 1

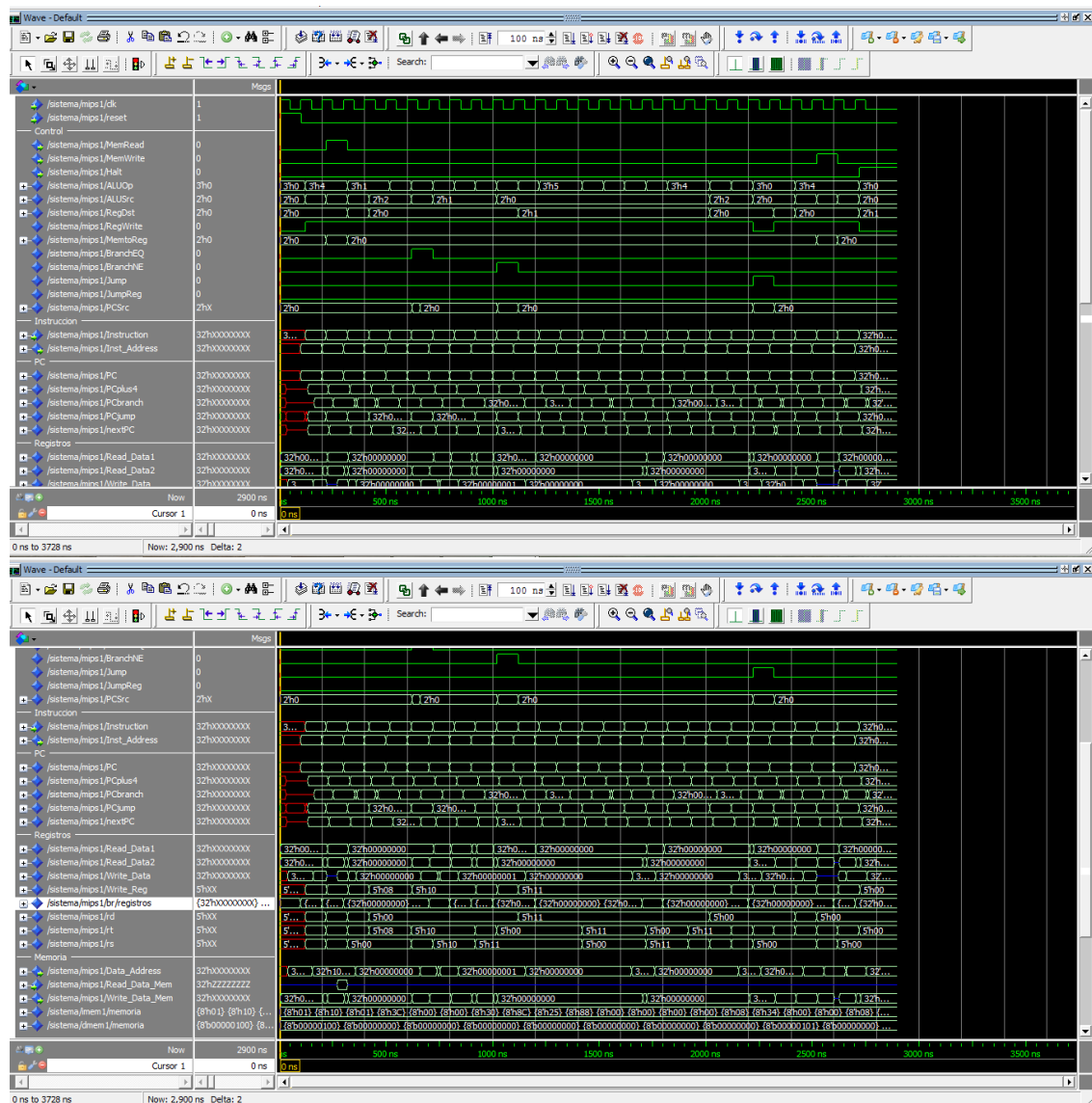
           addi    $t0,$t0,1
           j       e1
           e1:nop
           sw      $s1,res
           lui     $v0,10
           syscall
```

Comenzaremos analizando el caso2 con el modelo de comportamiento del circuito para asegurarnos de que el resultado es el correcto y a continuación pasaremos a analizar dicho caso con el modelo estructural del circuito (a excepción de la ALU). Para comprobar el segundo modelo del circuito solo tendremos que compararlo con el analizado previamente y analizar los posibles errores o diferencias.

Para comenzar a analizar el caso2 lo primero que tenemos que hacer es saber el número de instrucciones que tiene nuestro caso, para ello utilizaremos el programa MARS, el cual también utilizaremos para ir comprobando los resultados obtenidos en Modelsim. Nuestro caso de prueba cuenta con 27 instrucciones, pero algunas de ellas no se pueden ejecutar en un solo ciclo de reloj, por lo que finalmente contamos con 34 pseudoinstrucciones de MIPS. A pesar de esto, nuestro circuito contará con 27 ciclos de reloj para llevar a cabo el caso de prueba.

A continuación pasaremos a simular nuestro circuito y posteriormente pasaremos a analizarlo para comprobar que todas las instrucciones son las correctas.

Tras la ejecución del modelo de comportamiento del circuito obtenemos las siguientes ondas:





Una vez tenemos en funcionamiento el circuito pasamos a analizar cada una de las instrucciones del caso para comprobar que el funcionamiento es el correcto.

Comenzamos con la primera instrucción (lw) dicha instrucción se divide en dos ciclos de reloj, ya que el repertorio de MIPS no soporta dicha instrucción en un solo ciclo, por lo tanto estará formada por lui y lw.

#### **Instrucción lui:**

Carga un inmediato a un registro.

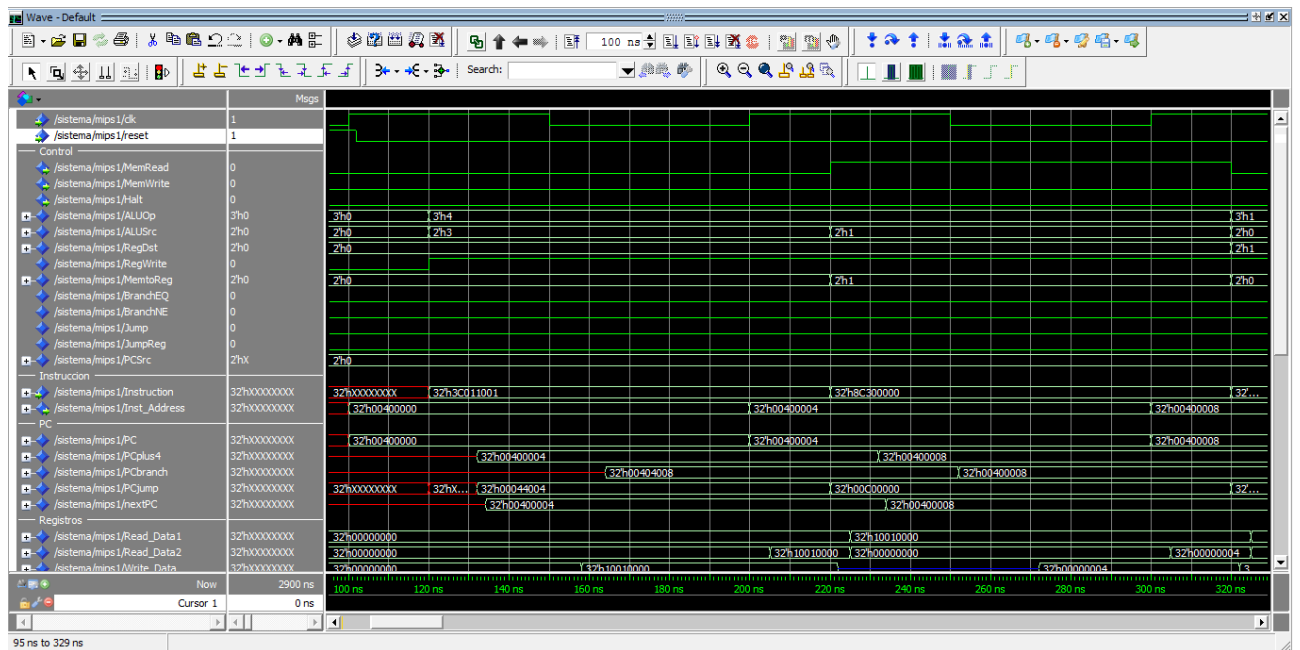
Comienzo a los 100 ns y termina a los 200. Durante este tiempo pone las señales ALUOp a "100" para que haga una suma, ALUSrc a "11" para que pase al segundo operador de la ALU el dato con extensión en ceros. RegDst a "00" para indicar el registro en el que se va a guardar. RegWrite a "1" para que funcione la parte de los registros. MemtoReg a "00" para que tome el valor de salida de la ALU y lo guarde en el registro indicado por RegDst. MemRead y MemWrite se ponen a "0" ya que la información que circulará por dichas señales no va a ser relevante (la información importante es la que sale de la ALU). PCSrc se pondrá a "0" para que pase PC+4 al siguiente PC y poder continuar con la ejecución del programa. Para que PCSrc esté a 0 todas las señales que la conforman tienen que estar a 0.

#### **Instrucción lw:**

Carga en el registro el contenido de la palabra de memoria que indica la dirección.

Comienza a los 200 ns y termina a los 300. Durante este tiempo pone las señales ALUOp a "100" para que haga una suma, ALUSrc a "01" para que pase al segundo operando de la ALU el dato con extensión en signo. RegDst a "00" para indicar el registro en el que se va a guardar. Regwrite a "1" para que funcione la parte de los registros. MemtoReg a "01" para que tome el valor que sale de la memoria y los escriba en el registro que indica Write Register. MemRead estará a "0" ya que lo que queremos es escribir en memoria y MemWrite estará a 1. PCSrc se pondrá a "0" para que pase PC+4 al siguiente PC y poder continuar con la ejecución del programa. Para que PCSrc esté a 0 todas las señales que la conforman tienen que estar a 0.

*A continuación se muestran estas dos instrucciones con más detalle. El primer ciclo de reloj se corresponde con un reset inicial para poner todas las señales a 0, por lo que mostramos solo las dos primeras instrucciones (100-300 ns).*



### Instrucción or:

Guarda en rd el resultado de aplicar la suma lógica de rs y rt.

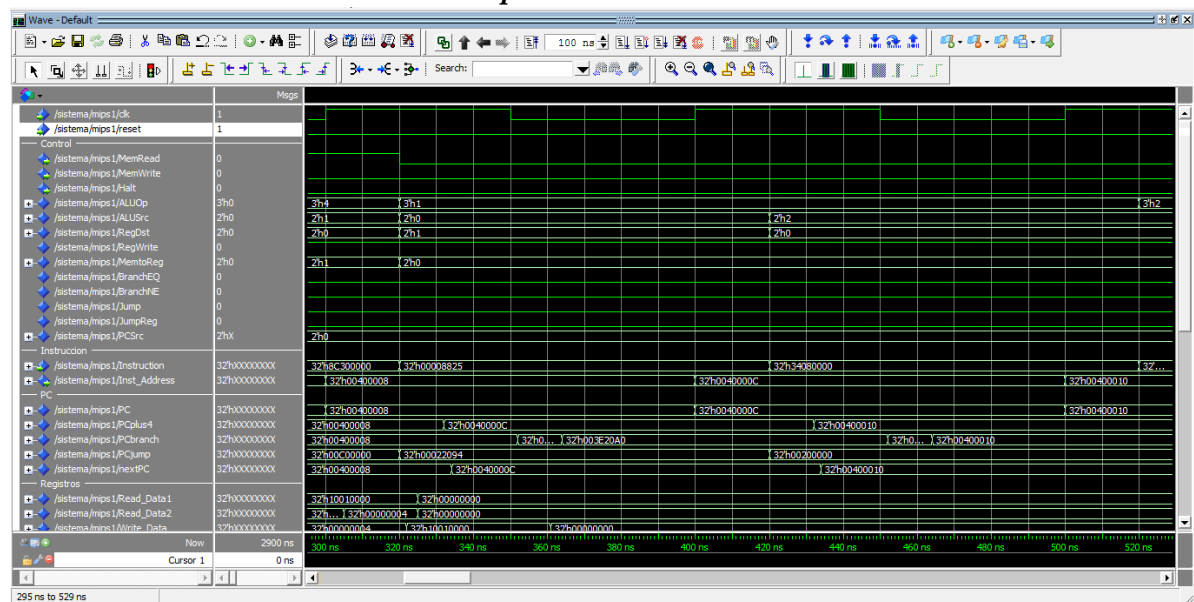
Comienza a los 300 ns y termina a los 400. Durante este tiempo pone las señales ALUOp a "001" para que haga una suma lógica (OR), ALUSrc a "00" para que pase al segundo operando de la ALU el registro rt. RegDst a "01" para indicar el registro en el que se va a guardar (rd). Regwrite a "1" para que funcione la parte de los registros. MemtoReg a "00" para que tome el valor nada mas salir de la ALU y lo escriba en rd. MemRead y MemWrite estará a "0" ya que lo que queremos es el valor que nos da la ALU y podemos omitir esa parte. PCSrc se pondrá a "0" para que pase PC+4 al siguiente PC y poder continuar con la ejecución del programa. Para que PCSrc esté a 0 todas las señales que la conforman tienen que estar a 0.

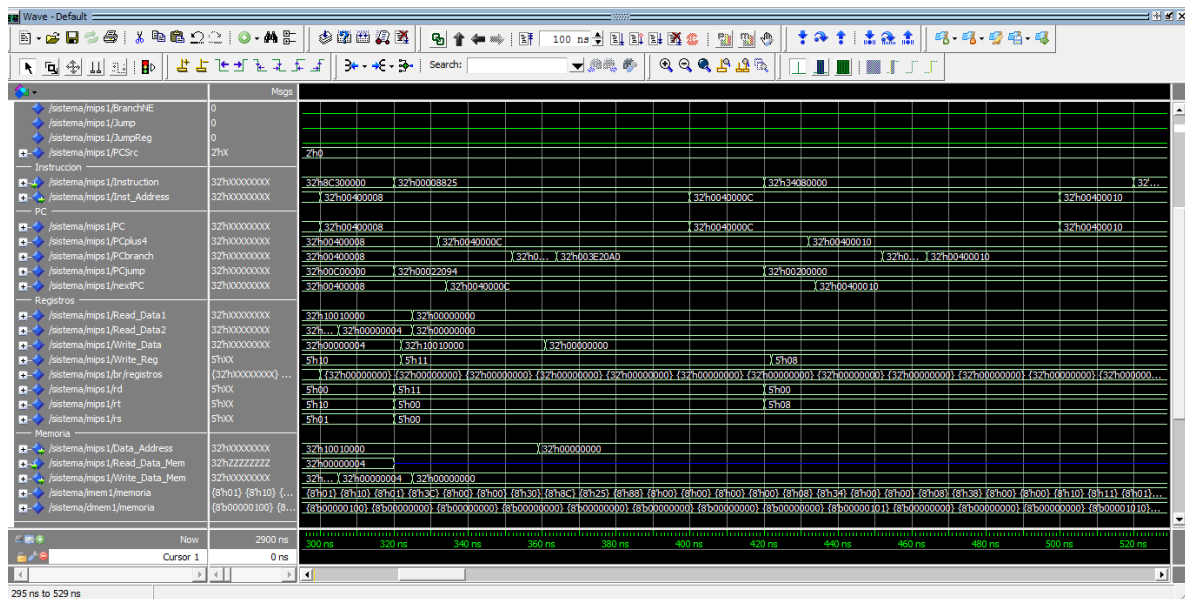
### Instrucción Ori:

Suma lógica bit a bit con inmediato.

Comienza a los 400 ns y termina a los 500. Durante este tiempo pone las señales ALUOp a "001" para que haga una suma lógica (OR), ALUSrc a "10" para que pase al segundo operando de la ALU el inmediato con extensión en ceros. RegDst a "00" para indicar el registro en el que se va a guardar. Regwrite a "1" para que funcione la parte de los registros. MemtoReg a "00" para que tome el valor nada mas salir de la ALU. MemRead y MemWrite estará a "0" ya que lo que queremos es el valor que nos da la ALU y podemos omitir esa parte. PCSrc se pondrá a "0" para que pase PC+4 al siguiente PC y poder continuar con la ejecución del programa. Para que PCSrc esté a 0 todas las señales que la conforman tienen que estar a 0.

*A continuación se muestran las dos operaciones con más detalle:*





### Instrucción Xori:

Suma lógica exclusiva bit a bit con inmediato.

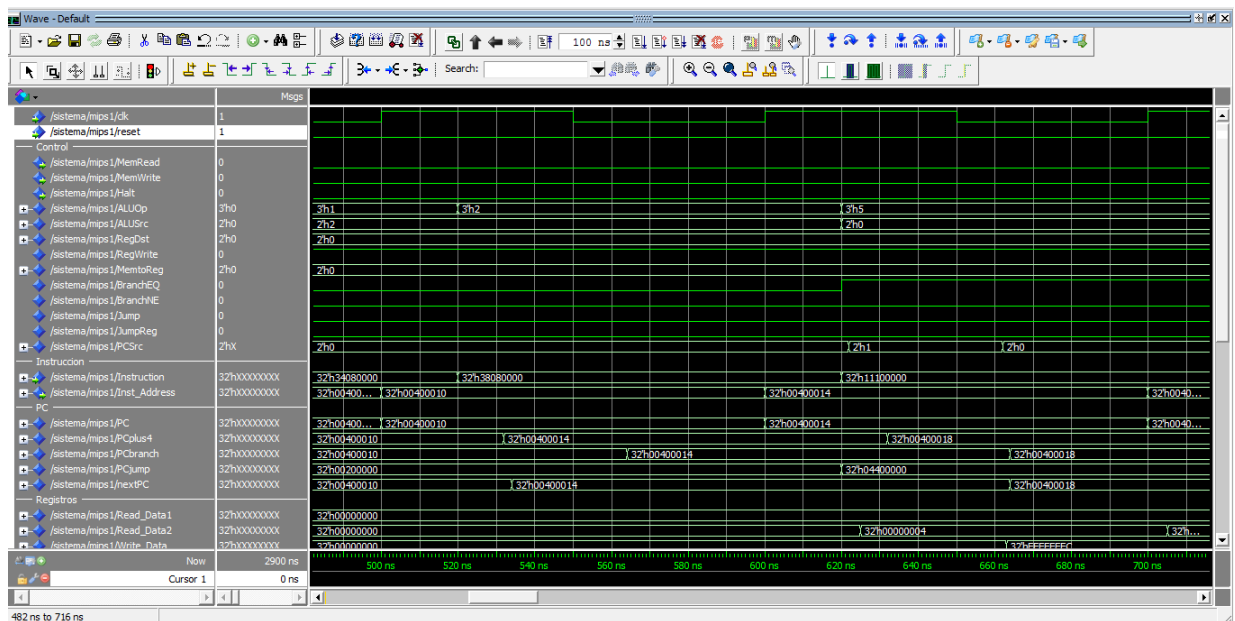
Comienza a los 500 ns y termina a los 600. Durante este tiempo pone las señales ALUOp a “010” para que haga XOR, ALUSrc a “10” para que pase al segundo operando de la ALU el inmediato con extensión en ceros. RegDst a “00” para indicar el registro en el que se va a guardar. Regwrite a “1” para que funcione la parte de los registros. MemtoReg a “00” para que tome el valor nada mas salir de la ALU. MemRead y MemWrite estará a “0” ya que lo que queremos es el valor que nos da la ALU y podemos omitir esa parte. PCSrc se pondrá a “0” para que pase PC+4 al siguiente PC y poder continuar con la ejecución del programa. Para que PCSrc esté a 0 todas las señales que la conforman tienen que estar a 0.

### Instrucción beq:

Ramifica a la instrucción de la etiqueta si rs y rt son iguales.

Comienza a los 600 ns y termina a los 700. Durante este tiempo pone las señales ALUOp a “101” para que la ALU reste rs y rt y comprobar si son iguales. ALUSrc a “00” para que pase rt al segundo operando de la ALU. RegDst a “00” para que se guarde el registro de la última operación. RegWrite a “1” para activar la parte de los registros. MemtoReg a “10” para guardar en rt la dirección de la siguiente operación antes de entrar al beq. MemRead y MemWrite a “0” ya que esa parte no nos va a interesar. PCSrc a “01” para que tome PC+4 y lo sume a la dirección de la etiqueta con extensión en signo. Para que PCSrc esté a 01 las señales que lo conforman tienen que ser: Zero “1”, BranchEQ “1”, BranchNE “0”, Jump “0”, JumpReg “0”, y Halt a “0”.

*A continuación se muestran estas dos operaciones en más detalle:*



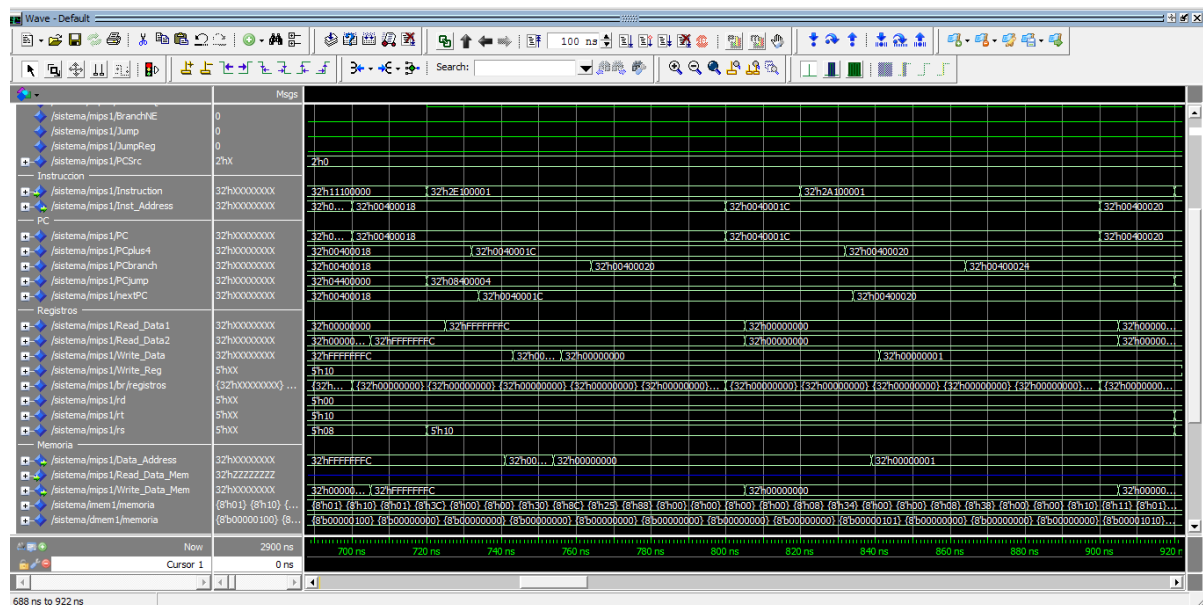
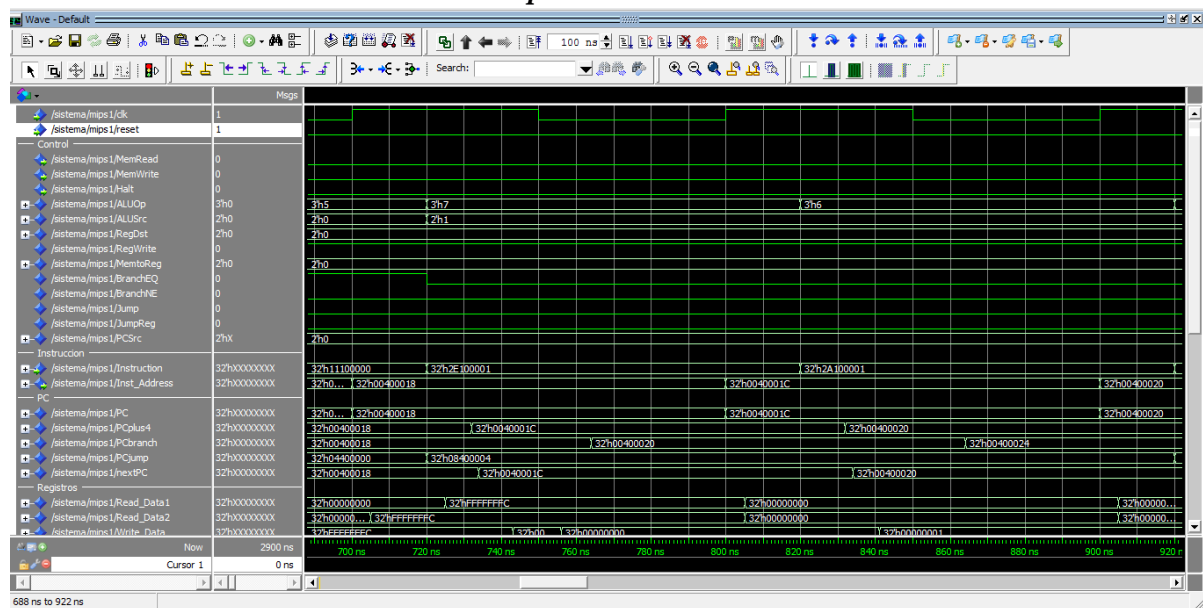
Comienza a los 700 ns y termina a los 800. Durante este tiempo pone las señales ALUOp a “111” para que compare los operandos, ALUSrc a “01” para que pase al segundo operando de la ALU el inmediato con extensión en signo. RegDst a “00”. Regwrite a “1” para que funcione la parte de los registros. MemtoReg a “00” para que tome el valor nada mas salir de la ALU y lo escriba en rd. MemRead y MemWrite estará a “0” ya que lo que queremos es el valor que nos da la ALU y podemos omitir esa parte. PCSrc se pondrá a “0” para que pase PC+4 al siguiente PC y poder continuar con la ejecución del programa. Para que PCSrc esté a 0 todas las señales que la conforman tienen que estar a 0.

## Instrucción slti:

Pone a 1 rt si rs es menor que el inmediato con extensión en signo suponiendo que están representados en complemento a 2.

Comienza a los 800 ns y termina a los 900. Durante este tiempo pone las señales ALUOp a “110” para que compare los operandos, ALUSrc a “01” para que pase al segundo operando de la ALU el inmediato con extensión en signo. RegDst a “00”. Regwrite a “1” para que funcione la parte de los registros. MemtoReg a “00” para que tome el valor nada mas salir de la ALU y lo escriba en rd. MemRead y MemWrite estará a “0” ya que lo que queremos es el valor que nos da la ALU y podemos omitir esa parte. PCSrc se pondrá a “0” para que pase PC+4 al siguiente PC y poder continuar con la ejecución del programa. Para que PCSrc esté a 0 todas las señales que la conforman tienen que estar a 0.

*A continuación se muestran estas dos operaciones con más detalle:*



### Instrucción addiu:

Suma con inmediato, genera un bit de desbordamiento pero como nuestro modelo no lo utiliza va a funcionar igual que la instrucción addi.

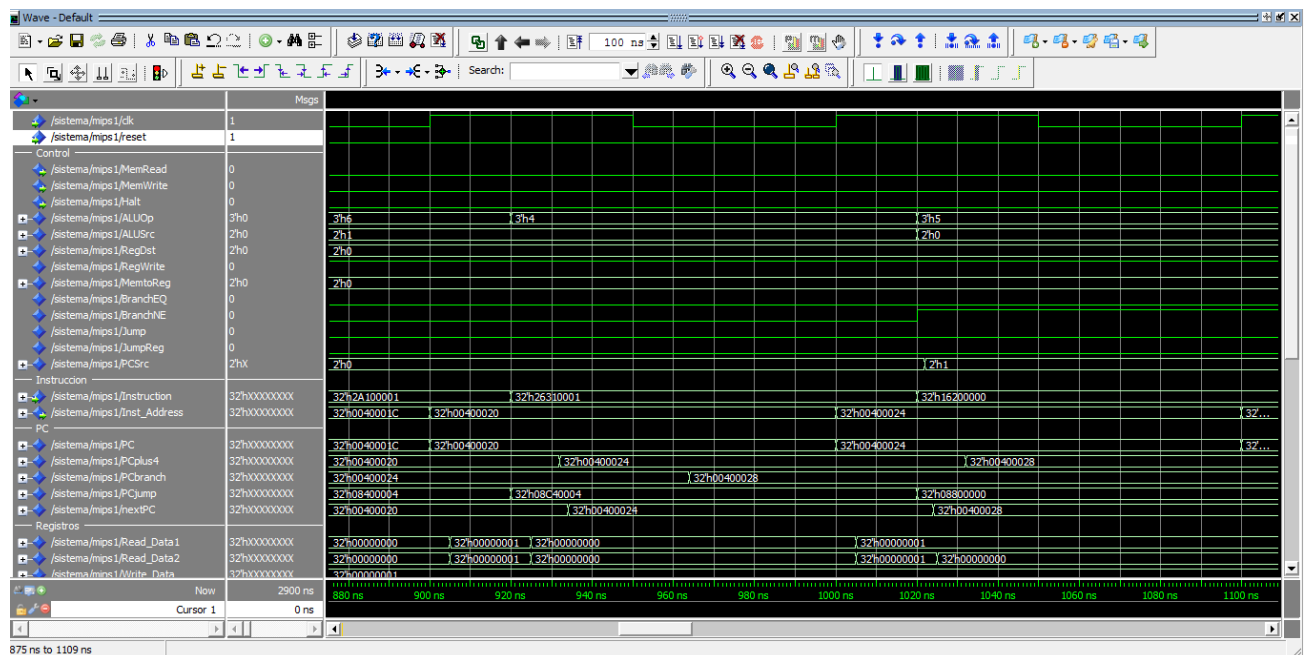
Comienza a los 900 ns y termina a los 1000. Durante este tiempo pone las señales ALUOp a "000" para que haga un producto lógico (AND), ALUSrc a "10" para que pase al segundo operando de la ALU el inmediato con extensión en ceros. RegDst a "00". Regwrite a "1" para que funcione la parte de los registros. MemtoReg a "00" para que tome el valor nada mas salir de la ALU y lo escriba en rd. MemRead y MemWrite estará a "0" ya que lo que queremos es el valor que nos da la ALU y podemos omitir esa parte. PCSrc se pondrá a "0" para que pase PC+4 al siguiente PC y poder continuar con la ejecución del programa. Para que PCSrc esté a 0 todas las señales que la conforman tienen que estar a 0.

### Instrucción bne:

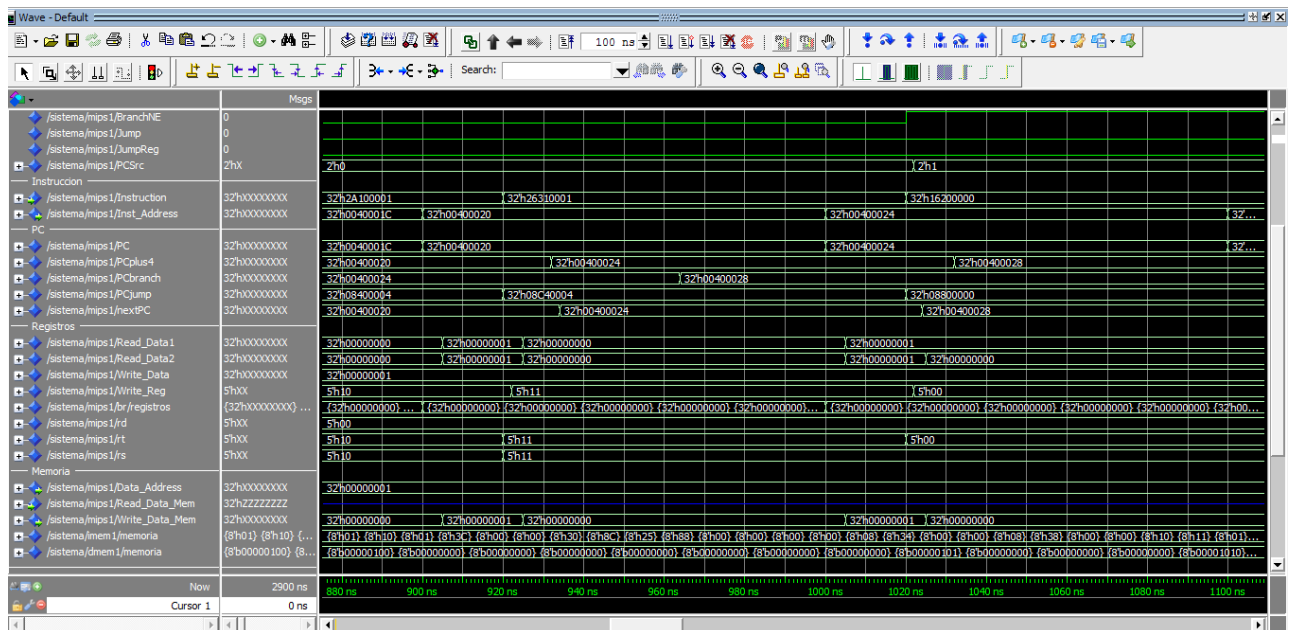
Ramifica a la instrucción de la etiqueta si rs y rt son diferentes.

Comienza a los 1000 ns y termina a los 1100. Durante este tiempo pone las señales ALUOp a "101" para que la ALU reste rs y rt y comprobar si son iguales. ALUSrc a "00" para que pase rt al segundo operando de la ALU. RegDst a "00" para que se guarde el registro de la última operación. RegWrite a "1" para activar la parte de los registros. MemtoReg a "10" para guardar en rt la dirección de la siguiente operación antes de entrar al beq. MemRead y MemWrite a "0" ya que esa parte no nos va a interesar. PCSrc a "01" para que tome PC+4 y lo sume a la dirección de la etiqueta con extensión en signo. Para que PCSrc esté a 01 las señales que lo conforman tienen que ser: Zero "0", BranchEQ "0", BranchNE "1", Jump "0", JumpReg "0", y Halt a "0".

*A continuación se muestran las dos instrucciones con más detalle:*







### Instrucción sltu:

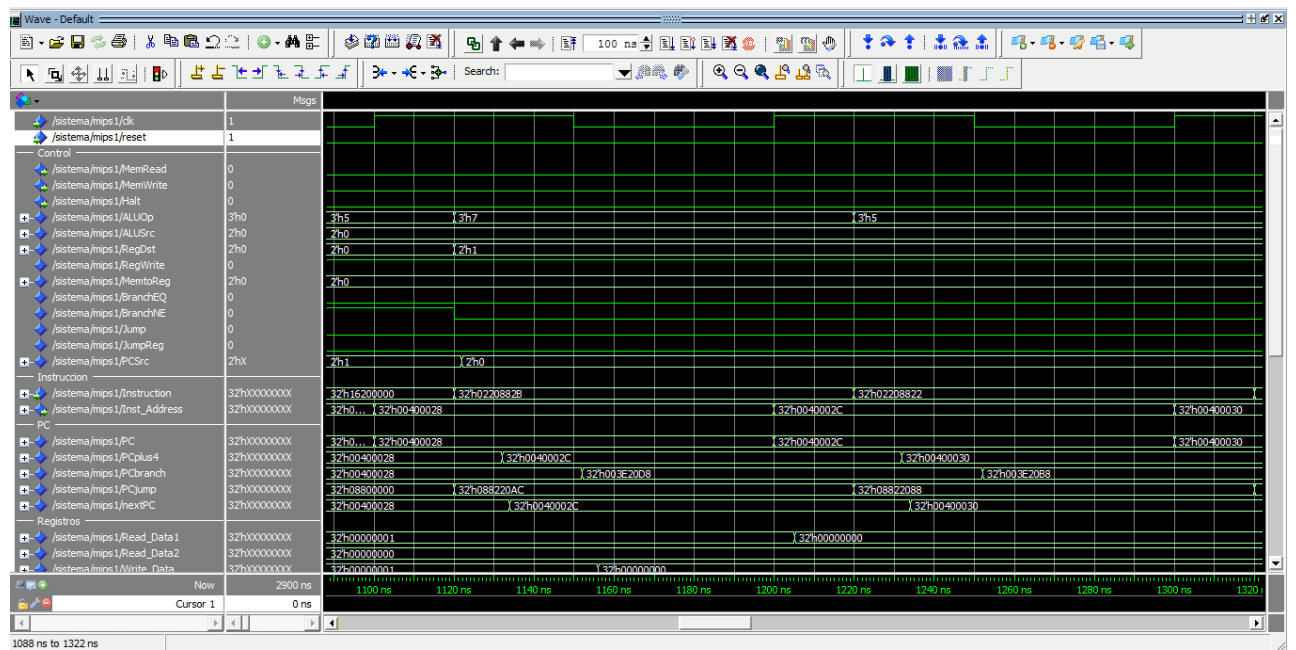
Esta instrucción compara dos operandos suponiendo que están representados en binario puro. Si *rs* es menor que *rt* entonces se guarda 1 en *rd*, si no, se guarda 0 en *rd*. Comienza a los 1100 ns y termina a los 1200. Durante este tiempo pone la señales ALUOp a "111" para que compare los operandos en binario puro, ALUSrc a "00" para que pase al segundo operando de la ALU el registro *rt*. RegDst a "01" para indicar el registro en el que se va a guardar (*rd*). Regwrite a "1" para que funcione la parte de los registros. MemtoReg a "00" para que tome el valor nada mas salir de la ALU y lo escriba en *rd*. MemRead y MemWrite estará a "0" ya que lo que queremos es el valor que nos da la ALU y podemos omitir esa parte. PCSrc se pondrá a "0" para que pase PC+4 al siguiente PC y poder continuar con la ejecución del programa. Para que PCSrc esté a 0 todas las señales que la conforman tienen que estar a 0.

### Instrucción sub:

Resta el contenido de *rs* y *rt* y lo guarda en *rd*. Comienza a los 1200 ns y termina a los 1300. Durante este tiempo pone las señales ALUOp a "101" para que haga una resta, ALUSrc a "00" para que pase al segundo operando de la ALU el registro *rt*. RegDst a "01" para indicar el registro en el que se va a guardar (*rd*). Regwrite a "1" para que funcione la parte de los registros. MemtoReg a "00" para que tome el valor nada mas salir de la ALU y lo escriba en *rd*. MemRead y MemWrite estará a "0" ya que lo que queremos es el valor que nos da la ALU y podemos omitir esa parte. PCSrc se pondrá a "0" para que pase PC+4 al siguiente PC y poder continuar con la ejecución del programa. Para que PCSrc esté a 0 todas las señales que la conforman tienen que estar a 0.

*A continuación se muestran las dos operaciones con más detalle:*





**Instrucción subu:**

Resta el contenido de rs y rt y lo guarda en rd. Es igual que la instrucción sub pero sin tener en cuenta el desbordamiento. Como nuestro circuito no contempla el desbordamiento las dos instrucciones serán iguales.

Comienza a los 1300 ns y termina a los 1400. Las señales toman los mismos valores que en sub.

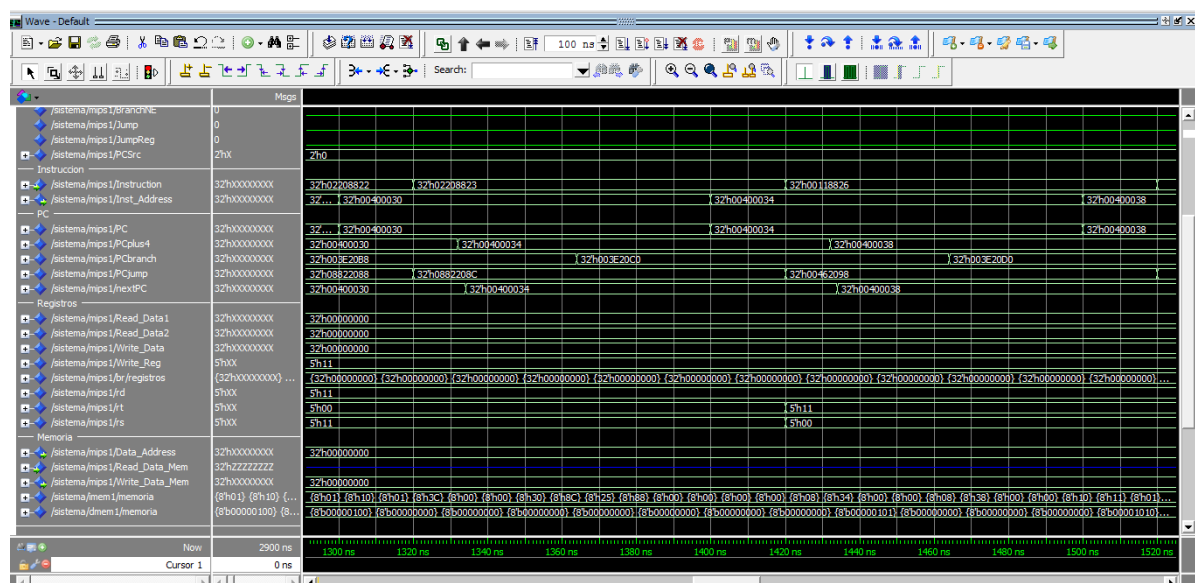
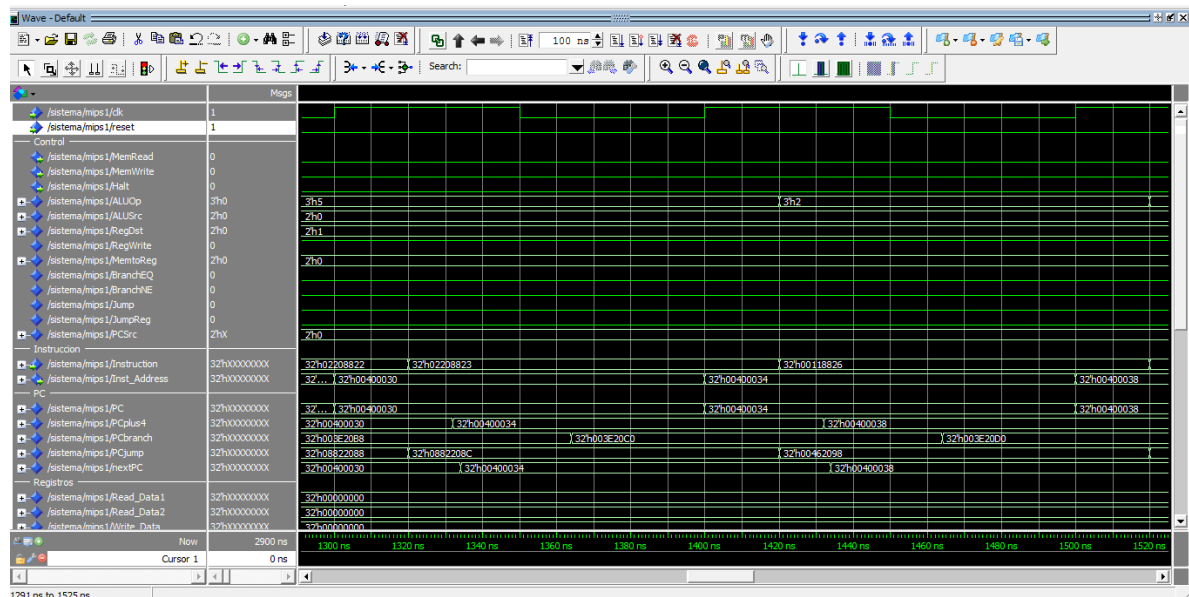
### **Instrucción xor:**

Guarda en rd el resultado de aplicar un xor lógico a rs y rt.

Comienza a los 1400 ns y termina a los 1500. Durante este tiempo pone las señales ALUop a

“010” para que haga un xor lógico. ALUsrc a “00” para que pase al segundo operando de la ALU el registro rt. RegDst a “01” para indicar el registro en el que se va a guardar (rd).

Regwrite a “1” para que funcione la parte de los registros. MemtoReg a “00” para que tome el valor nada mas salir de la ALU y lo escriba en rd. MemRead y MemWrite estará a “0” ya que lo que queremos es el valor que nos da la ALU y podemos omitir esa parte. PCSrc se pondrá a “0” para que pase PC+4 al siguiente PC y poder continuar con la ejecución del programa. Para que PCSrc esté a 0 todas las señales que la conforman tienen que estar a 0.



### Instrucción slt:

Esta instrucción compara dos operandos suponiendo que están representados en complemento a 2. Si rs es menor que rt entonces se guarda 1 en rd, si no, se guarda 0 en rd.

Comienza a los 1500 ns y termina a los 1600. Durante este tiempo pone la señales ALUOp a "110" para que compare los operandos, ALUSrc a "00" para que pase al segundo operando de la ALU el registro rt. RegDst a "01" para indicar el registro en el que se va a guardar (rd).

Regwrite a "1" para que funcione la parte de los registros. MemtoReg a "00" para que tome el valor nada mas salir de la ALU y lo escriba en rd. MemRead y MemWrite estará a "0" ya que lo que queremos es el valor que nos da la ALU y podemos omitir esa parte. PCSrc se pondrá a "0" para que pase PC+4 al siguiente PC y poder continuar con la ejecución del programa. Para que PCSrc esté a 0 todas las señales que la conforman tienen que estar a 0.

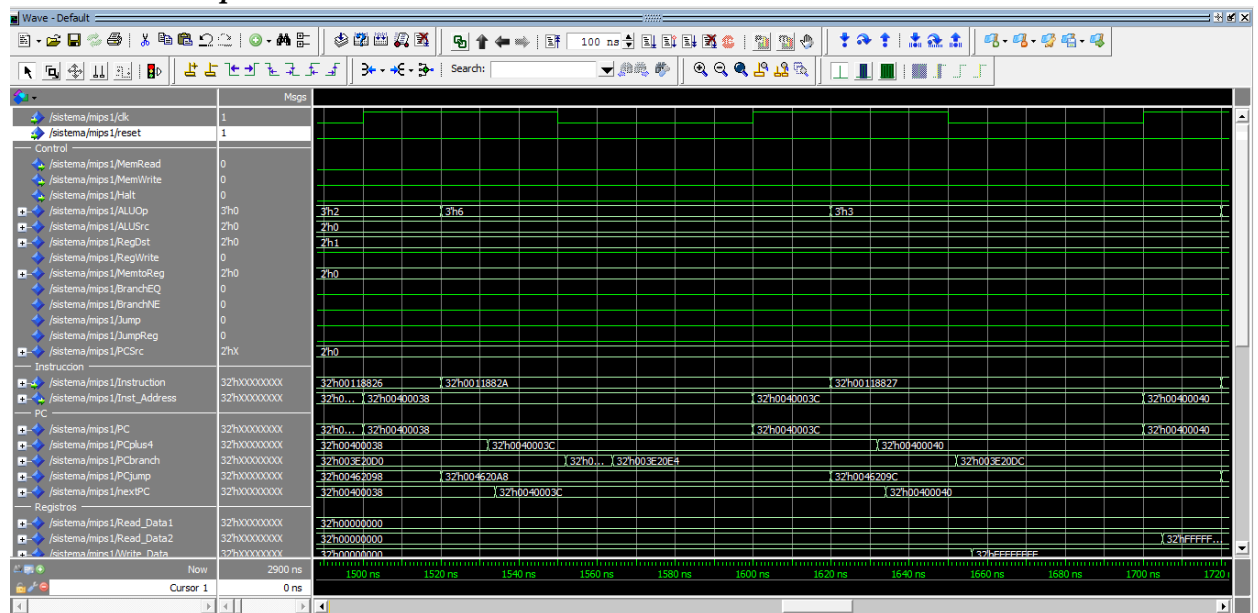
### Instrucción nor:

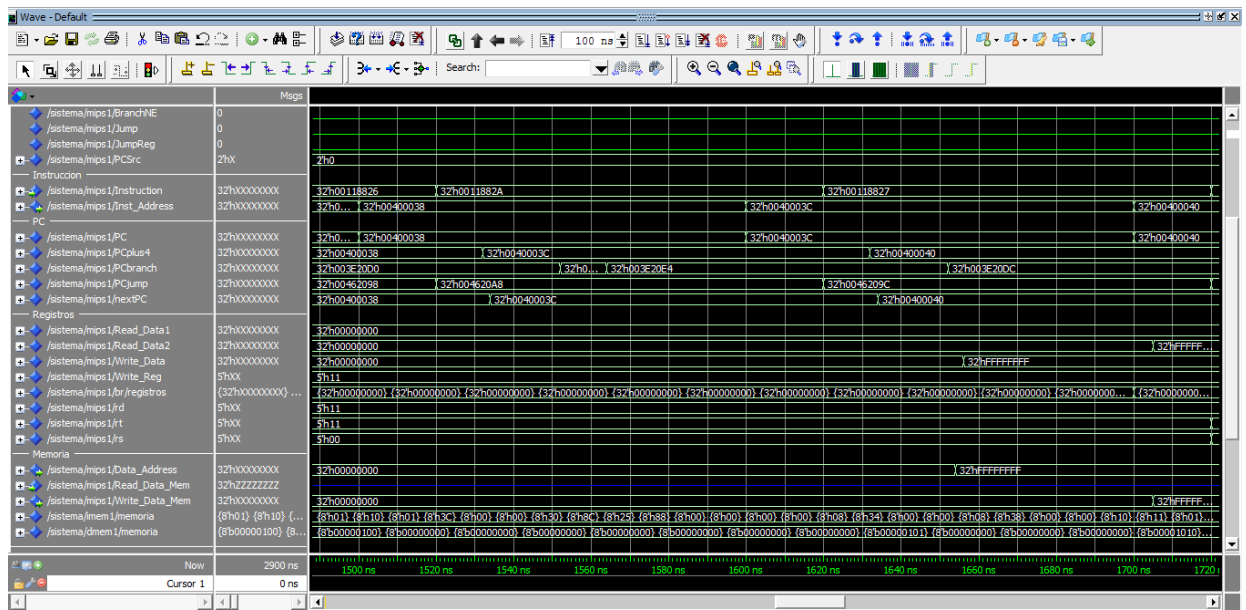
Guarda en rd el resultado de aplicar un nor lógico a rs y rt.

Comienza a los 1600 ns y termina a los 1700. Durante este tiempo pone las señales ALUOp a "011" para que haga un nor lógico. ALUSrc a "00" para que pase al segundo operando de la ALU el registro rt. RegDst a "01" para indicar el registro en el que se va a guardar (rd).

Regwrite a "1" para que funcione la parte de los registros. MemtoReg a "00" para que tome el valor nada mas salir de la ALU y lo escriba en rd. MemRead y MemWrite estará a "0" ya que lo que queremos es el valor que nos da la ALU y podemos omitir esa parte. PCSrc se pondrá a "0" para que pase PC+4 al siguiente PC y poder continuar con la ejecución del programa. Para que PCSrc esté a 0 todas las señales que la conforman tienen que estar a 0.

*A continuación podemos ver las dos señales con más detalle:*





### Instrucción and:

Guarda en rd el resultado de aplicar el producto lógico de rs y rt.

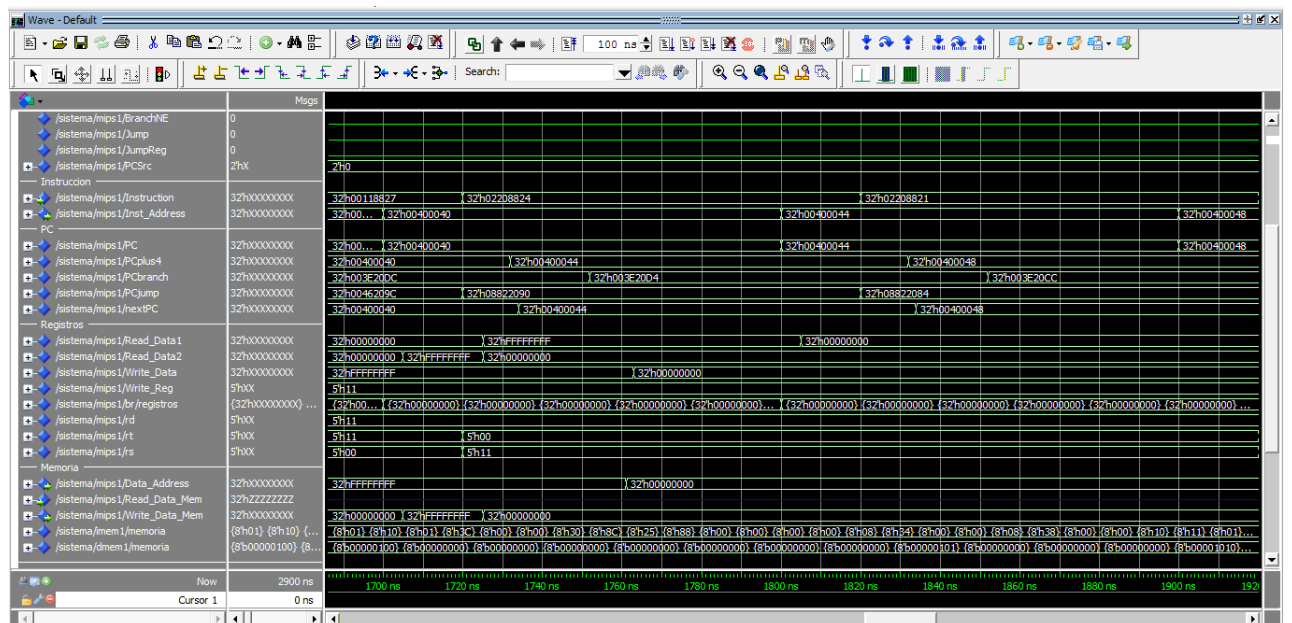
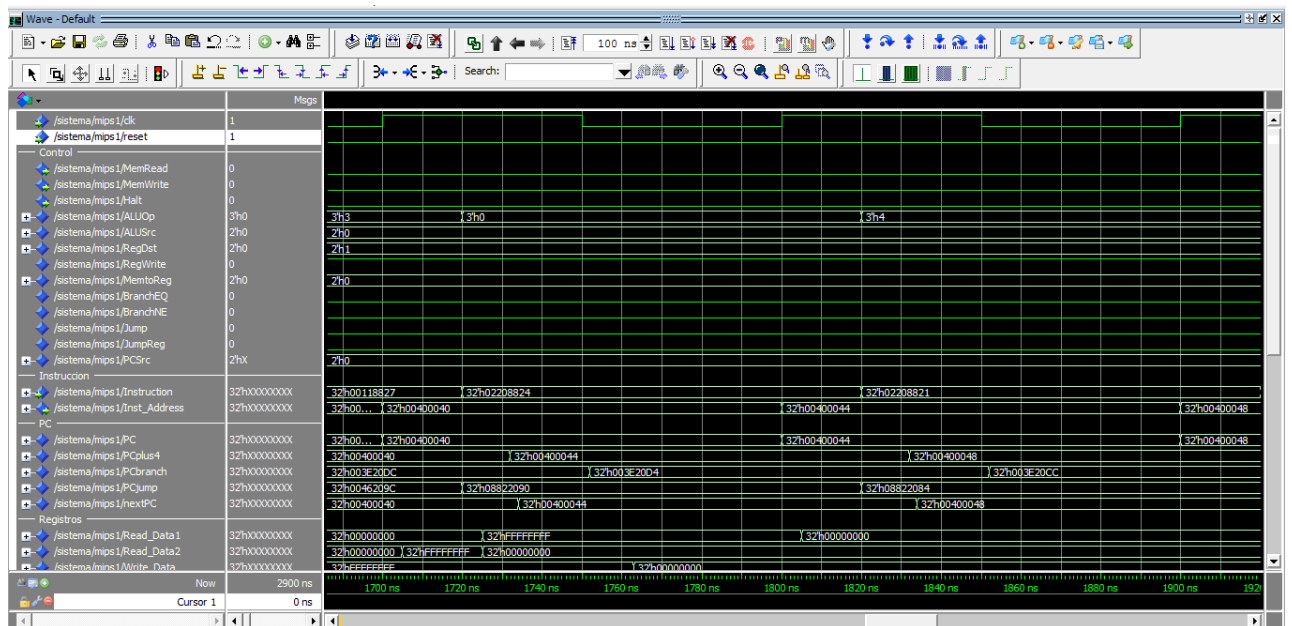
Comienza a los 1700 ns y termina a los 1800. Durante este tiempo pone las señales ALUOp a “000 ” para que haga un producto lógico (AND), ALUSrc a “00” para que pase al segundo operando de la ALU el registro rt. RegDst a “01” para indicar el registro en el que se va a guardar (rd). Regwrite a “1” para que funcione la parte de los registros. MemtoReg a “00” para que tome el valor nada mas salir de la ALU y lo escriba en rd. MemRead y MemWrite estará a “0” ya que lo que queremos es el valor que nos da la ALU y podemos omitir esa parte. PCSrc se pondrá a “0” para que pase PC+4 al siguiente PC y poder continuar con la ejecución del programa. Para que PCSrc esté a 0 todas las señales que la conforman tienen que estar a 0.

### Instrucción addu:

Suma el contenido de rs y rt y lo guarda en rd.

Comienza a los 1800 ns y termina a los 1900. Durante este tiempo pone las señales ALUOp a “100” para que haga una suma, ALUSrc a “00” para que pase al segundo operando de la ALU el registro rt. RegDst a “01” para indicar el registro en el que se va a guardar (rd). Regwrite a “1” para que funcione la parte de los registros. MemtoReg a “00” para que tome el valor nada mas salir de la ALU y lo escriba en rd. MemRead y MemWrite estará a “0” ya que lo que queremos es el valor que nos da la ALU y podemos omitir esa parte. PCSrc se pondrá a “0” para que pase PC+4 al siguiente PC y poder continuar con la ejecución del programa. Para que PCSrc esté a 0 todas las señales que la conforman tienen que estar a 0.

*A continuación se muestra el cronograma de las dos instrucciones con más detalle.*



### Instrucción add:

Suma el contenido de rs y rt y lo guarda en rd. Es igual que addu pero teniendo en cuenta el desbordamiento, ya que nuestro circuito no lo tiene en cuenta, las dos instrucciones son iguales.

Comienza a los 1900 ns y termina a los 2000. Las señales que hemos modificado toman los mismos valores que en add.

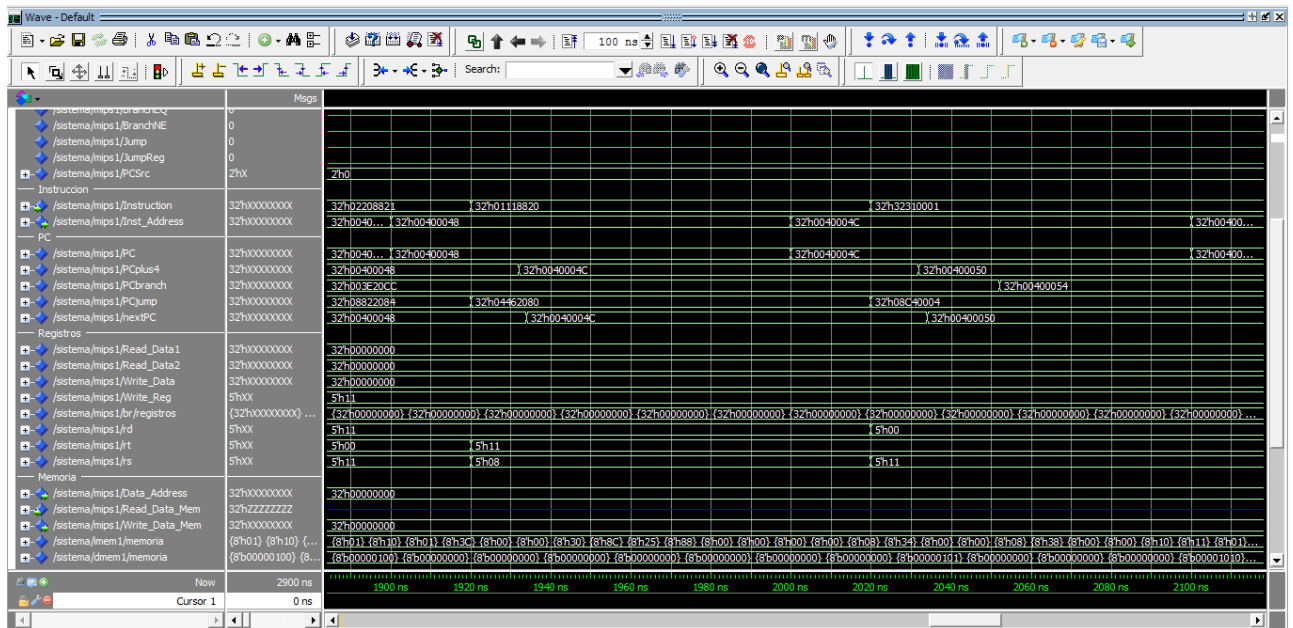
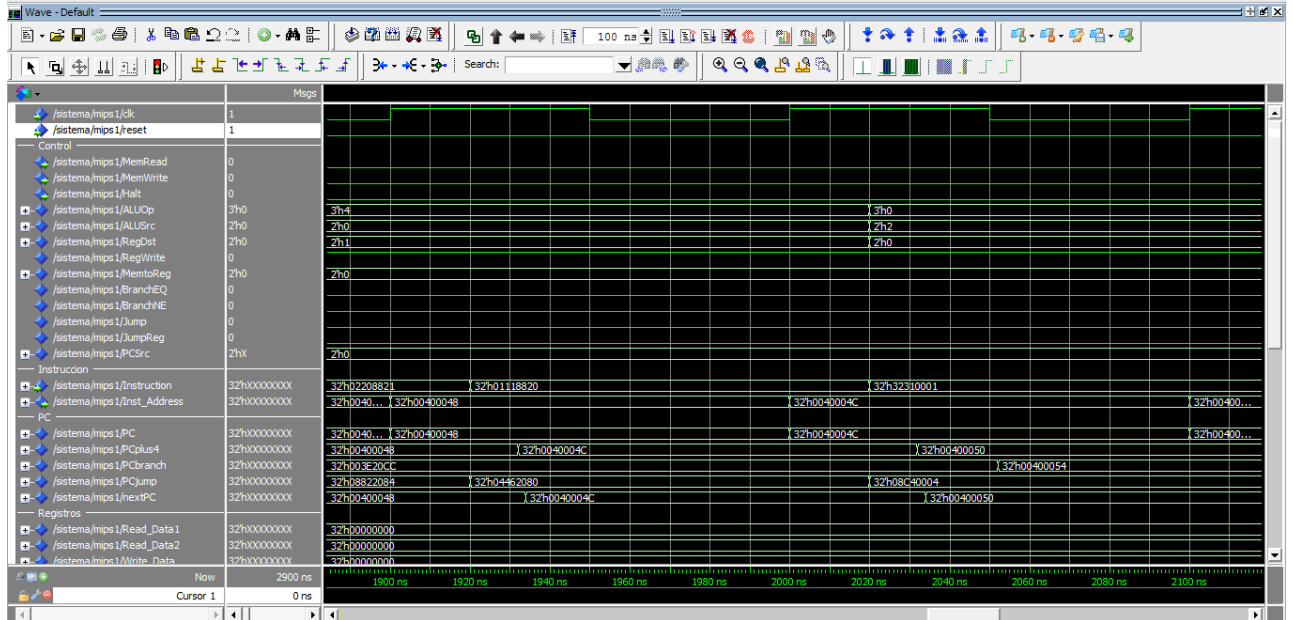
### Instrucción andi:

Hace el producto lógico bit a bit con inmediato.

Comienza a los 2000 ns y termina a los 2100. Durante este tiempo pone las señales ALUOp a “000” para que haga un producto lógico (AND), ALUSrc a “10” para que pase al segundo operando de la ALU el inmediato con extensión en ceros. RegDst a “00” para indicar el registro en el que se va a guardar. RegWrite a “1” para que funcione la parte de los registros. MemtoReg a “00” para que tome el valor nada mas salir de la ALU. MemRead y MemWrite estará a “0” ya

que lo que queremos es el valor que nos da la ALU y podemos omitir esa parte. PCSrc se pondrá a “0” para que pase PC+4 al siguiente PC y poder continuar con la ejecución del programa. Para que PCSrc esté a 0 todas las señales que la conforman tienen que estar a 0.

*A continuación mostramos los resultados obtenidos en la simulación:*





### Instrucción addi:

Suma con inmediato, genera un bit de desbordamiento pero como nuestro modelo no lo utiliza va a funcionar igual que la instrucción addiu.

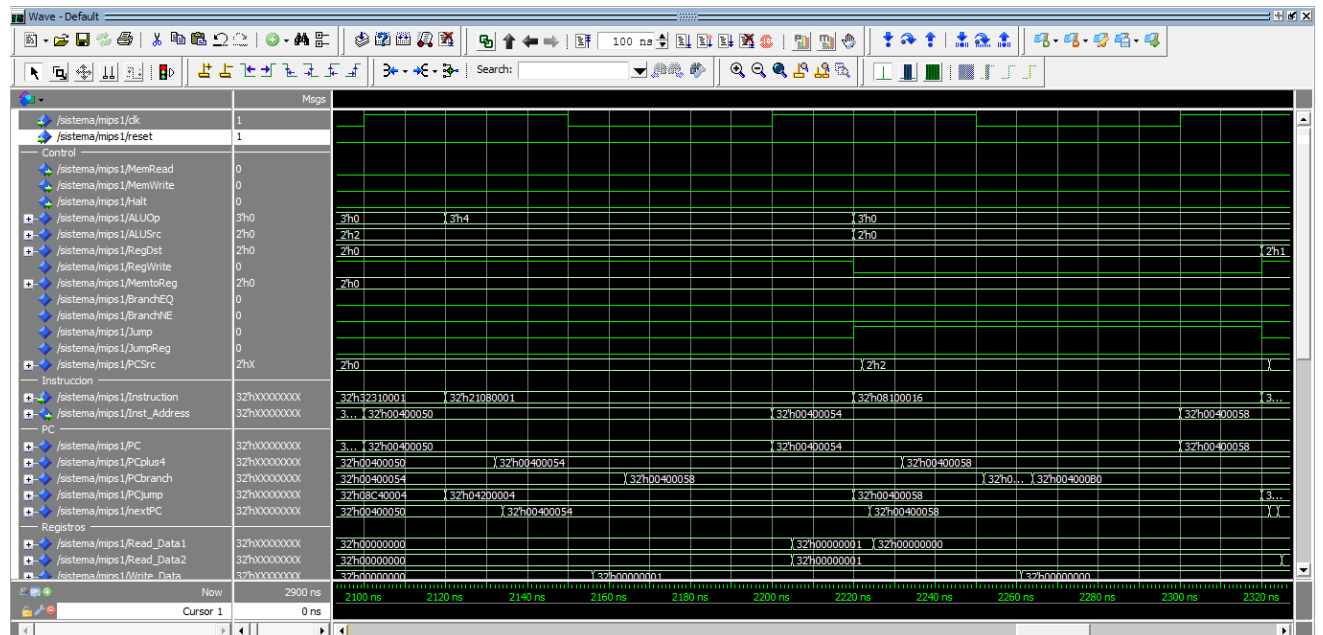
Comienza a los 2100 ns y termina a los 2200. Durante este tiempo pone las señales ALUOp a "000" para que haga un producto lógico (AND), ALUSrc a "10" para que pase al segundo operando de la ALU el inmediato con extensión en ceros. RegDst a "00". Regwrite a "1" para que funcione la parte de los registros. MemtoReg a "00" para que tome el valor nada mas salir de la ALU y lo escriba en rd. MemRead y MemWrite estará a "0" ya que lo que queremos es el valor que nos da la ALU y podemos omitir esa parte. PCSrc se pondrá a "0" para que pase PC+4 al siguiente PC y poder continuar con la ejecución del programa. Para que PCSrc esté a 0 todas las señales que la conforman tienen que estar a 0.

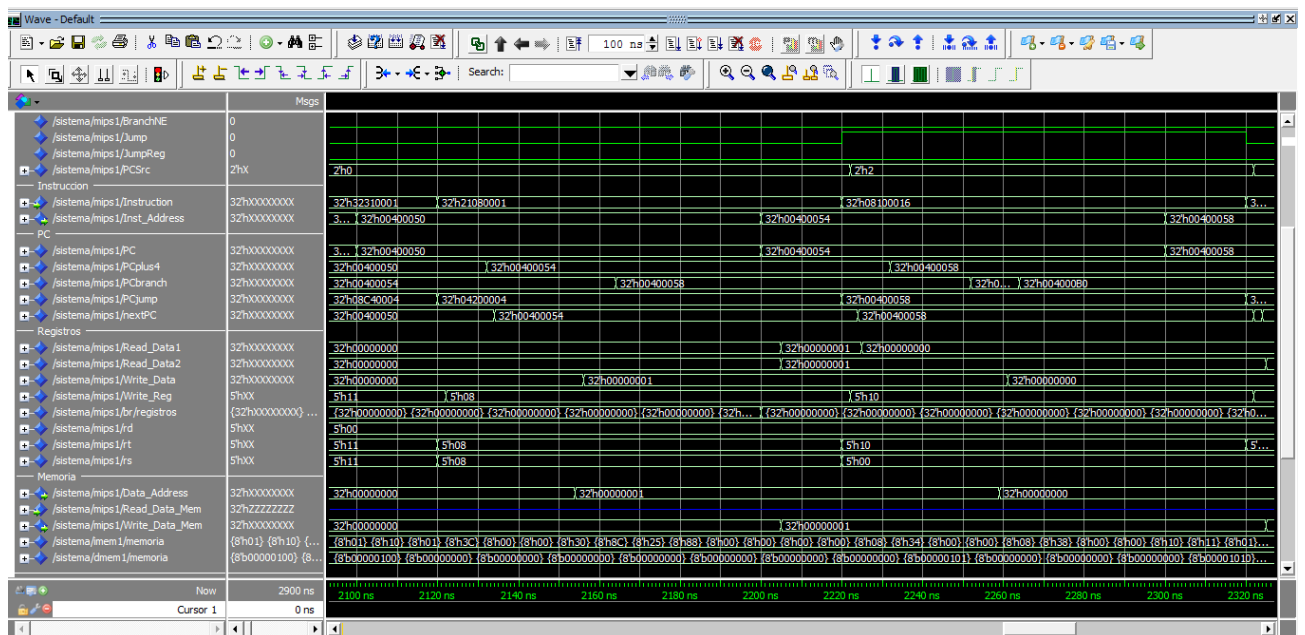
### Instrucción jump:

Salta a un destino.

Comienza a los 2200 ns y termina a los 2300. Durante este tiempo pone las señales ALUOp a "000" como valor indiferente, ya que la salida de la ALU no nos va a interesar, ALUSrc a "00" también como valor indiferente. RegDst a "00". Regwrite a "0" para que no se active la parte de los registros. MemtoReg a "00" Como valor indiferente de la señal, ya que no vamos a necesitar dicho valor. MemRead y MemWrite estará a "0". PCSrc se pondrá a "10" para que pase la dirección de destino al nextPC. Para que PCSrc esté a "10" las señales que lo forman tienen que tomar los siguientes valores: Zero = "0", BranchEQ = "0", BranchNE = "0", Jump = "1", JumpReg = "0", y Halt = "0".

*A continuación se muestran estas dos operaciones más detalladamente:*





### Instrucción nop:

Es una instrucción de tipo R con todos sus bits a 0.

Comienza a los 2300 ns y termina a los 2400. Durante este tiempo pone las señales ALUOp a “000” como valor indiferente, ya que no nos importa el valor de la ALU, ALUSrc a “00” para que pase al segundo operando de la ALU el segundo registro leído. RegDst a “01”. RegWrite a “0” para que no funcione la parte de los registros. MemtoReg a “00” para que tome el valor nada mas salir de la ALU y lo escriba en rd. MemRead y MemWrite estará a “0” ya que lo que queremos es el valor que nos da la ALU y podemos omitir esa parte. PCSrc se pondrá a “0” para que pase PC+4 al siguiente PC y poder continuar con la ejecución del programa. Para que PCSrc esté a 0 todas las señales que la conforman tienen que estar a 0.

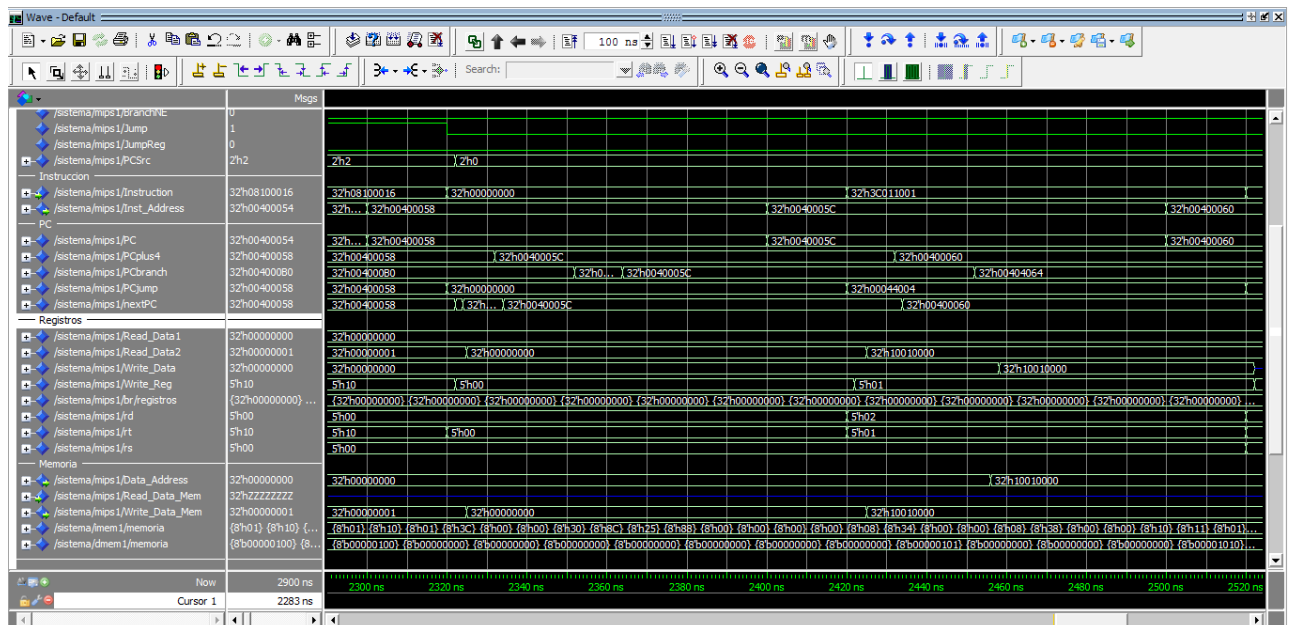
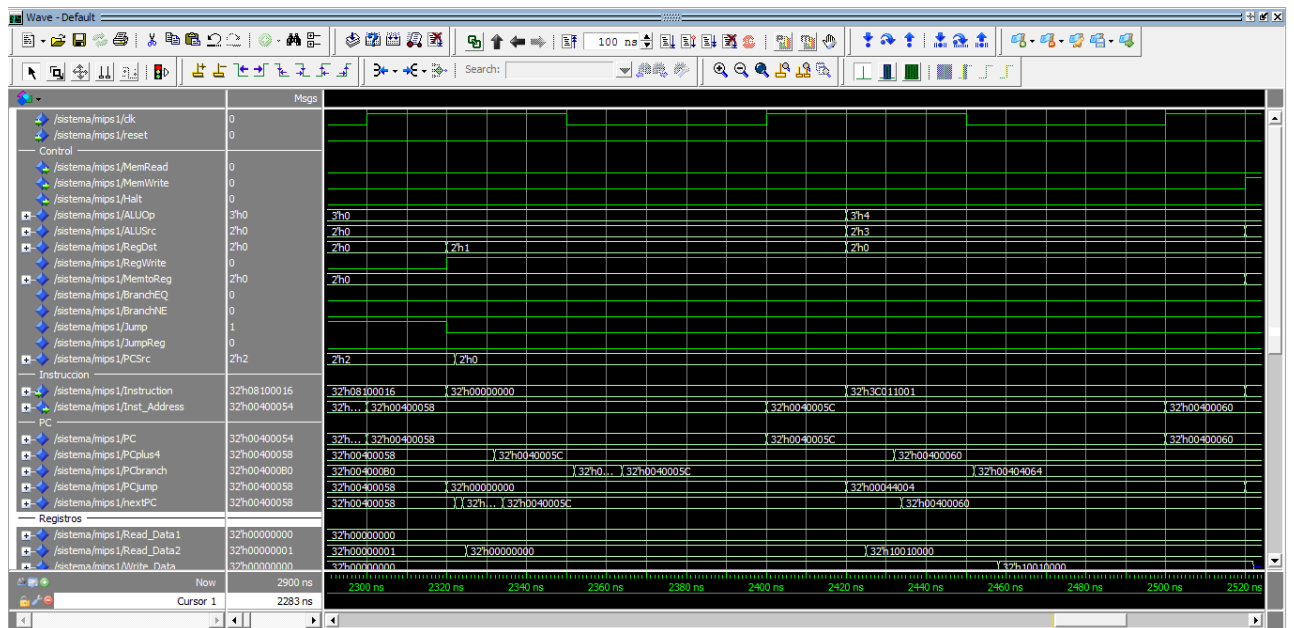
### Instrucción lui:

Carga un inmediato a un registro.

Comienzo a los 2400 ns y termina a los 2500. Durante este tiempo pone las señales ALUOp a “100” para que haga una suma, ALUSrc a “11” para que pase al segundo operador de la ALU el dato con extensión en ceros. RegDst a “00” para indicar el registro en el que se va a guardar. RegWrite a “1” para que funcione la parte de los registros. MemtoReg a “00” para que tome el valor de salida de la ALU y lo guarde en el registro indicado por RegDst. MemRead y MemWrite se ponen a “0” ya que la información que circulará por dichas señales no va a ser relevante (la información importante es la que sale de la ALU). PCSrc se pondrá a “0” para que pase PC+4 al siguiente PC y poder continuar con la ejecución del programa. Para que PCSrc esté a 0 todas las señales que la conforman tienen que estar a 0.

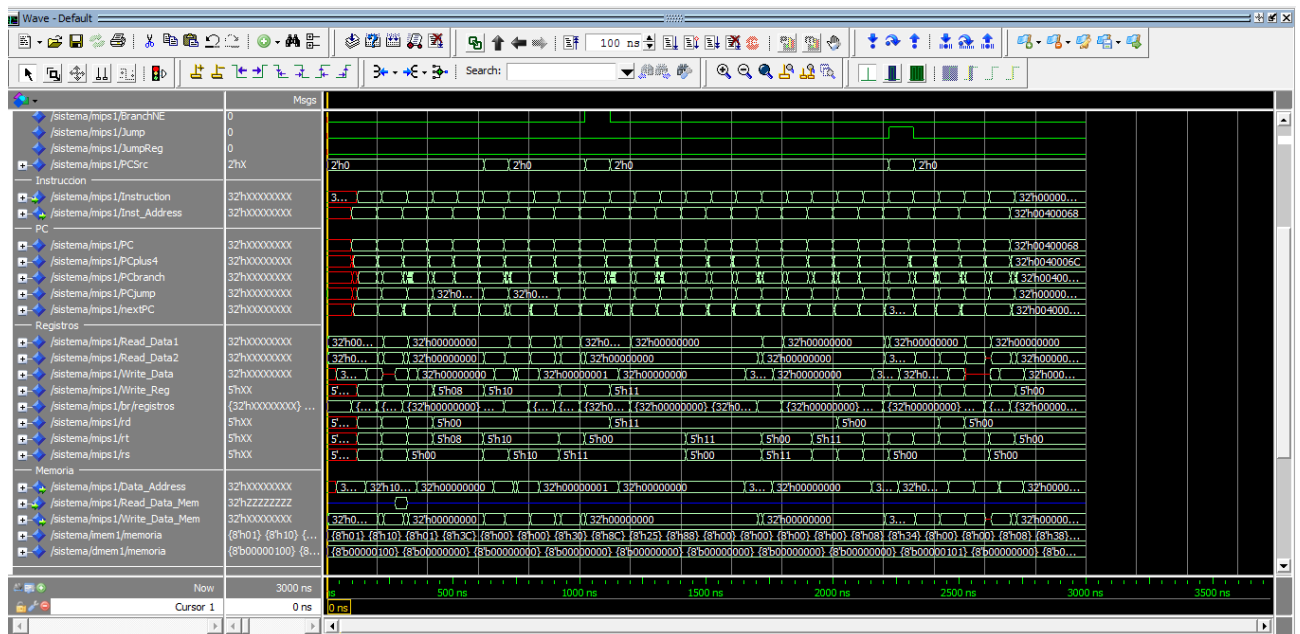
*A continuación se muestran las dos instrucciones:*





Una vez que tenemos analizado el caso con el modelo de comportamiento del circuito pasamos a compararlo con el modelo estructural. Puesto que ya está analizado uno de los modelos ahora solo quedaría comparar las ondas que nos dan como resultado los dos modelos y comprobar que el resultado coincide.

Para ello, pasamos al modelo estructural y lo simulamos en VHDL. Las ondas que nos da como resultado son las siguientes:



### **Opinión personal:**

Durante la realización de esta práctica hemos comprendido mejor los conceptos que no teníamos completamente asimilados, así como aumentar los conocimientos que teníamos de VHDL.

Debido a los cambios producidos en la práctica hemos adquirido cierta experiencia analizando problemas inesperados y buscando soluciones fuera del entorno habitual de trabajo.

En conclusión, la práctica nos ha sido muy útil para asentar los conocimientos básicos de la asignatura.