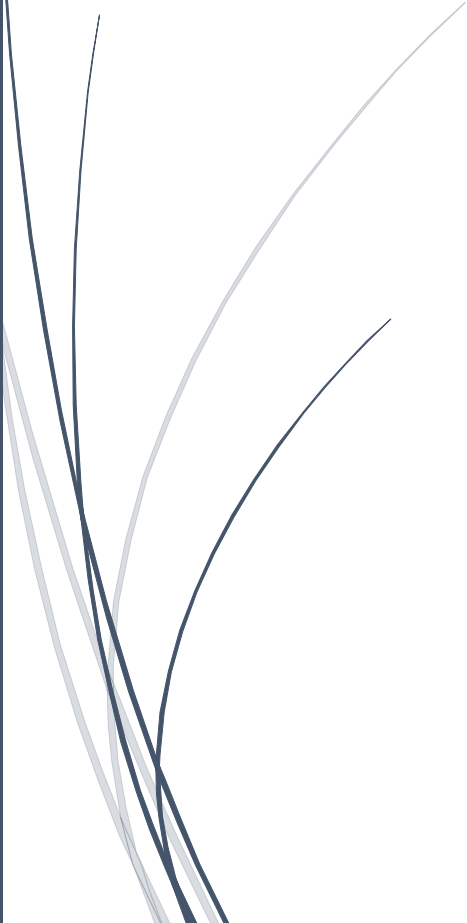


A dark blue vertical bar on the left side of the page. A blue arrow points to the right from the bar, containing the date.

23-4-2015

Práctica 2

Estructura de Computadores

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

Guillermo Meléndez Morales y Susana Pineda De Luelmo

La señal **ALUop** tiene 3 bits de salida y **ALUsrc**, **RegDst** y **MemToReg** tienen 2 bits de salida para poder controlar los multiplexores y las operaciones de la ALU. Además, contamos con la señal **PCSrc** compuesta por **JumpReg**, **Jump**, **BranchNE**, **BranchEQ**. Para controlar estas señales contaremos con las tablas de verdad de la señal **ALUop** y la señal **PCSrc**:

ALUop:

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 (res=1 si A<b con signo)
111	A sltu B (res=1 si A<b sin signo)

Las señales RegWrite, MemWrite y MemRead responden a diferentes funciones:

RegWrite: Cuando está inactiva no tiene ningún efecto, y cuando está activa el registro destino se escribe con el valor correspondiente (Write register).

MemWrite: No tiene ningún efecto cuando esta inactiva, y cuando está activa se escribe una posición de memoria con el valor dado en la entrada de datos.

MemRead: Cuando está inactiva no tiene ningún efecto, y cuando está activa se lee una posición de memoria y su contenido se coloca a la salida de datos.

El resto de señales se controlarán dependiendo de la entrada que queramos poner a la salida del multiplexor.

Tras estudiar una a una las instrucciones **Add, Addu, And, Break, Jalr, Jr, nor, nop, Or, Slt, Xor, Subu, Sub, Slu, Syscall, J, Jal, Beq, Bne, Addi, Addiu, Sltu, Andi, Ori, Xori, Lui, Lw y Sw** y ver su recorrido por el camino de datos unificado, hemos creado una tabla que recoge los diferentes códigos de operación, las señales de salida y las instrucciones correspondiente con el fin de facilitar la implementación en VHDL:

CodOP	funct	ALUOP <i>Op₁Op₂Op₃</i>	ALUSrc <i>Src₁Src₂</i>	RegDst <i>Dest₁Dest₂</i>	RegWrite	MemtoReg <i>Mr₁Mr₂</i>	MemRead	MemWrite	Zero	BranchEQ	BranchNE	Jump	JumpReg	Halt	
000000	100000	100	00	01	1	00	x	x	x	0	0	0	0	0	Add
000000	100001	100	00	01	1	00	x	x	x	0	0	0	0	0	Addu
000000	100100	000	00	01	1	00	x	x	x	0	0	0	0	0	And
000000	001101	xxx	xx	xx	x	xx	x	x	x	0	0	0	0	1	Break
000000	001001	xxx	xx	01	1	10	x	x	x	x	x	x	1	0	Jalr
000000	001000	xxx	xx	01	1	10	x	x	x	x	x	x	1	0	Jr
000000	100111	011	00	01	1	00	x	x	x	x	x	x	x	0	Nor
000000	000000	xxx	00	01	0	xx	0	0	x	0	0	0	0	0	Nop
000000	100101	001	00	01	1	00	x	x	x	0	0	0	0	0	Or
000000	101010	110	00	01	1	00	x	x	x	0	0	0	0	0	Slt
000000	100110	010	00	01	1	00	x	x	x	0	0	0	0	0	Xor
000000	100011	101	00	01	1	00	x	x	x	0	0	0	0	0	Subu
000000	100010	101	00	01	1	00	x	x	x	0	0	0	0	0	Sub
000000	101011	111	00	01	1	00	x	x	x	0	0	0	0	0	Sltu
000000	001100	xxx	xx	xx	x	xx	x	x	x	x	x	x	x	1	Syscall
000010	xxxxxxxx	xxx	xx	xx	0	xx	x	x	x	x	x	1	0	0	J
000011	xxxxxxxx	xxx	xx	10	1	10	x	x	x	x	x	1	0	0	Jal
000100	xxxxxxxx	101	00	xx	1	10	x	x	1	1	0	0	0	0	Beq
000101	xxxxxxxx	101	00	xx	1	10	x	x	0	0	1	0	0	0	Bne
001000	xxxxxxxx	000	10	00	1	00	x	x	x	0	0	0	0	0	Addi
001001	xxxxxxxx	100	01	00	1	00	x	x	x	0	0	0	0	0	Addiu
001010	xxxxxxxx	110	01	00	1	00	x	x	x	0	0	0	0	0	Slti
001011	xxxxxxxx	111	01	00	1	00	x	x	x	0	0	0	0	0	Sltiu
001100	xxxxxxxx	000	10	00	1	00	x	x	0	0	0	0	0	0	Andi
001101	xxxxxxxx	001	10	00	1	00	x	x	0	0	0	0	0	0	Ori
001110	xxxxxxxx	010	10	00	1	00	x	x	0	0	0	0	0	0	Xori
001111	xxxxxxxx	100	10	00	1	00	x	x	0	0	0	0	0	0	Lui
100011	xxxxxxxx	100	01	00	1	01	1	0	x	0	0	0	0	0	Lw
101011	xxxxxxxx	100	01	00	1	01	0	1	x	0	0	0	0	0	Sw

Imagen incluida en el zip, para que se pueda ver mejor.

Los elementos que aparecen con señales no definidas, es decir que no importa qué valor tomen (x) se han puesto en el modelo VHDL como señales a 0.

La implementación de la unidad de control se ha realizado mediante una descripción funcional del circuito mediante la estructura **Case** Introduciendo así la salida de todas las señales para

los diferentes códigos de operaciones y en el caso de las operaciones de tipo R teniendo en cuenta también funct (6 bits menos significativos de la instrucción), dando como resultado el siguiente código:

```
...
case opcode is
  when "000000" => -- Operación tipo R
    case funct is
      when "000000" => -- nop
        NULL;
      when "001000" => --jr
        ALUOp <= "000"; ALUSrc <= "00"; RegDst <= "01"; RegWrite <= '1';
        MemtoReg <= "10"; MemRead <= '0'; MemWrite <= '0'; BranchEQ <= '0';
        BranchNE <= '0'; Jump <= '0'; JumpReg <= '1'; Halt <= '0';
      when "001001" => --jalr
    ...
```

Una vez hecho esto y habiendo comprobado que el código compila pasamos a crear los casos de prueba para comprobar que el diseño de la unidad de control es el correcto. Para ello utilizaremos el programa MARS y dos casos para comprobar el resultado de las operaciones.

En nuestro caso, los códigos utilizados son los siguientes:

<pre>.data dato1: .word 5 dato2: .word 10 res: .space 4 .text main: lw \$s0, dato1 lw \$s1, dato2 lui \$t0, 1 add \$s2, \$s0, \$s1 addu \$s2, \$s2, \$s1 sub \$s2, \$s2, \$s1 subu \$s2, \$s0, \$s1 and \$s2, \$s0, \$s1 or \$s2, \$s0, \$s1 xor \$s2, \$s0, \$s1 nor \$s2, \$s0, \$s1 slt \$s2, \$s1, \$s0 sltu \$s2, \$s1, \$s0 beq \$s2, \$s1, e e: bne \$s2, \$s1, e1 e1: addi \$s2, \$s1, 1 e2: addiu \$s2, \$s1, 1 slti \$s2, \$s1, 1 sltiu \$s2, \$s1, 1 andi \$s2, \$s1, 1 ori \$s2, \$s1, 1 xori \$s2, \$s1, 1 sw \$s2, res li \$v0, 10 syscall</pre>	<pre>.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</pre>
--	---

Caso1.asm:

El primer caso de prueba cuenta con 28 instrucciones. El contenido final de los registros y las variables de memoria son los siguientes:

Registers	Coproc 1	Coproc 0	
Name	Number	Value	
\$zero	0	0x00000000	
\$at	1	0x10010000	
\$v0	2	0x0000000a	
\$v1	3	0x00000000	
\$a0	4	0x00000000	
\$a1	5	0x00000000	
\$a2	6	0x00000000	
\$a3	7	0x00000000	
\$t0	8	0x00010000	
\$t1	9	0x00000000	
\$t2	10	0x00000000	
\$t3	11	0x00000000	
\$t4	12	0x00000000	
\$t5	13	0x00000000	
\$t6	14	0x00000000	
\$t7	15	0x00000000	
\$s0	16	0x00000005	
\$s1	17	0x0000000a	
\$s2	18	0x0000000b	
\$s3	19	0x00000000	
\$s4	20	0x00000000	
\$s5	21	0x00000000	
\$s6	22	0x00000000	
\$s7	23	0x00000000	
\$t8	24	0x00000000	
\$t9	25	0x00000000	
\$k0	26	0x00000000	
\$k1	27	0x00000000	
\$gp	28	0x10008000	
\$sp	29	0x7ffefffc	
\$fp	30	0x00000000	
\$ra	31	0x00000000	
pc		0x00400070	
hi		0x00000000	
lo		0x00000000	

Labels	
Label	Address ▲
fuente.asm	
main	0x00400000
e	0x00400040
e1	0x00400044
e2	0x00400048
dato1	0x10010000
dato2	0x10010004
res	0x10010008
<input checked="" type="checkbox"/> Data <input checked="" type="checkbox"/> Text	

Caso 2:

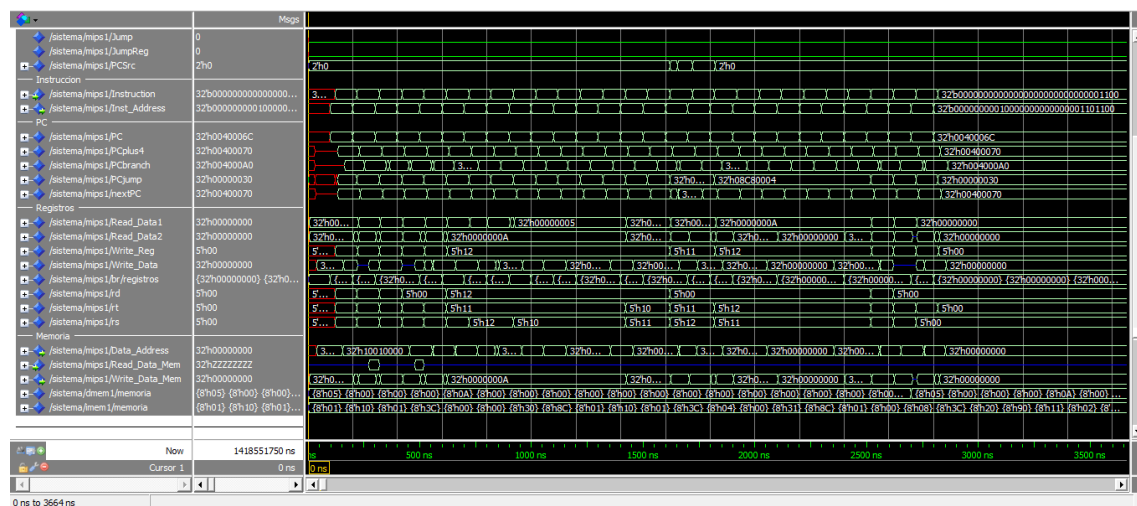
El segundo caso de prueba tiene 27 instrucciones. El contenido final de los registros y las variables de memoria son las siguientes:

Registers	Coproc 1	Coproc 0
Name	Number	Value
\$zero	0	0x00000000
\$at	1	0x10010000
\$v0	2	0x000a0000
\$v1	3	0x00000000
\$a0	4	0x00000000
\$a1	5	0x00000000
\$a2	6	0x00000000
\$a3	7	0x00000000
\$t0	8	0x00000001
\$t1	9	0x00000000
\$t2	10	0x00000000
\$t3	11	0x00000000
\$t4	12	0x00000000
\$t5	13	0x00000000
\$t6	14	0x00000000
\$t7	15	0x00000000
\$s0	16	0x00000001
\$s1	17	0x00000000
\$s2	18	0x00000000
\$s3	19	0x00000000
\$s4	20	0x00000000
\$s5	21	0x00000000
\$s6	22	0x00000000
\$s7	23	0x00000000
\$t8	24	0x00000000
\$t9	25	0x00000000
\$k0	26	0x00000000
\$k1	27	0x00000000
\$gp	28	0x10008000
\$sp	29	0x7ffefffc
\$fp	30	0x00000000
\$ra	31	0x00000000
pc		0x0040006c
hi		0x00000000
lo		0x00000000

Label	Address
Caso2.asm.asm	
main	0x00400000
bucle	0x00400014
e2	0x00400018
e	0x00400028
e1	0x00400058
dato	0x10010000
res	0x10010004
Caso1.asm.asm	
main	0x0040006c
e	0x004000ac
e1	0x004000b0
e2	0x004000b4
dato1	0x10010008
dato2	0x1001000c
res	0x10010010

☒ Data ☒ Text

The screenshot displays the Logic Analyzer tool interface, showing a complex digital logic circuit simulation. The top section lists the signals being monitored, including system bus signals like `/system/mps1/cdk`, `/system/mps1/reset`, and various memory/register signals. The middle section shows the waveform for these signals, with a time scale of 500 ns. The bottom section shows the waveform for the instruction and instruction address signals, with a time scale of 1000 ns. The signals are color-coded: green for logic 1, red for logic 0, and blue for high-impedance (Z). The waveform shows a sequence of events, including a reset signal, followed by memory reads and writes, and finally a jump instruction.



En el cronograma podemos comprobar que para cada operación que empleamos con el código utilizamos un ciclo de reloj (en algunos casos como lw utilizamos dos ya que la instrucción se divide a su vez en otras dos)

Instrucción lui:

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

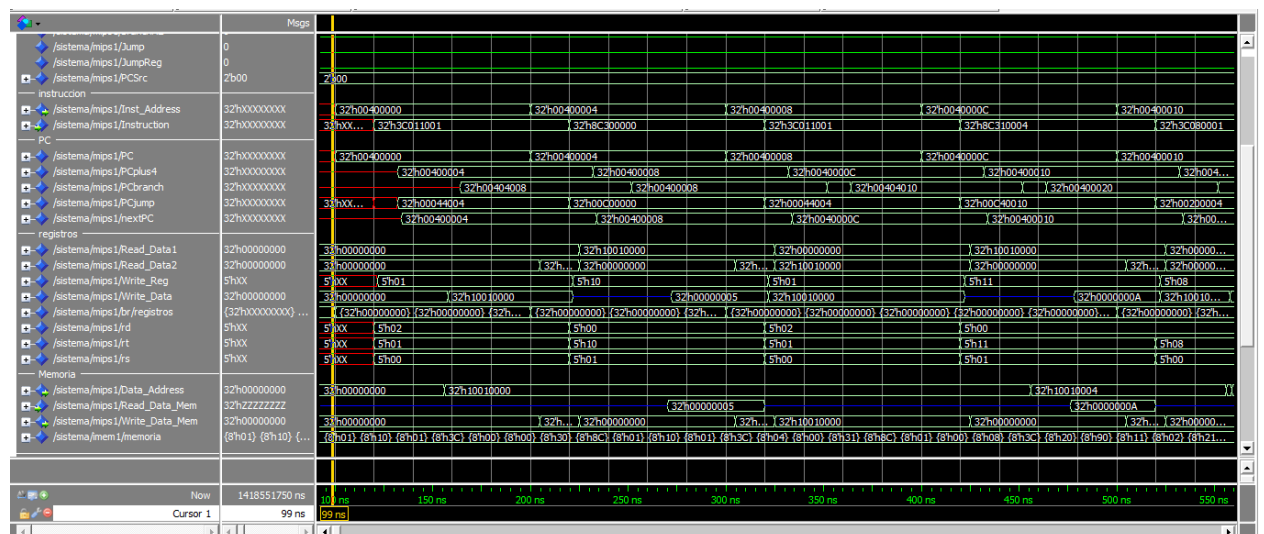
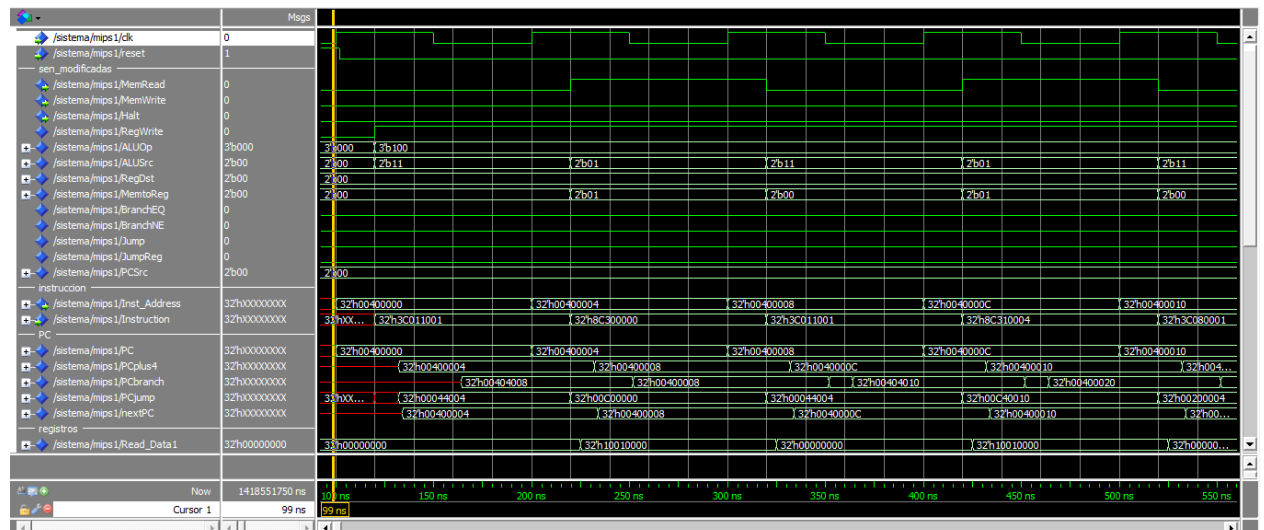
siguiente PC y poder continuar con la ejecución del programa. Para que PCSrc esté a 0 todas las señales que la conforman tienen que estar a 0.

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.

Los dos siguientes ciclos de reloj se corresponden con las mismas operaciones que los dos primeros lui y lw, por lo que se van a omitir, al igual que el 5º ciclo de reloj que también se corresponde a la instrucción lui. A continuación se muestra una captura de pantalla con estas dos operaciones más detalladas:



Instrucción add:

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

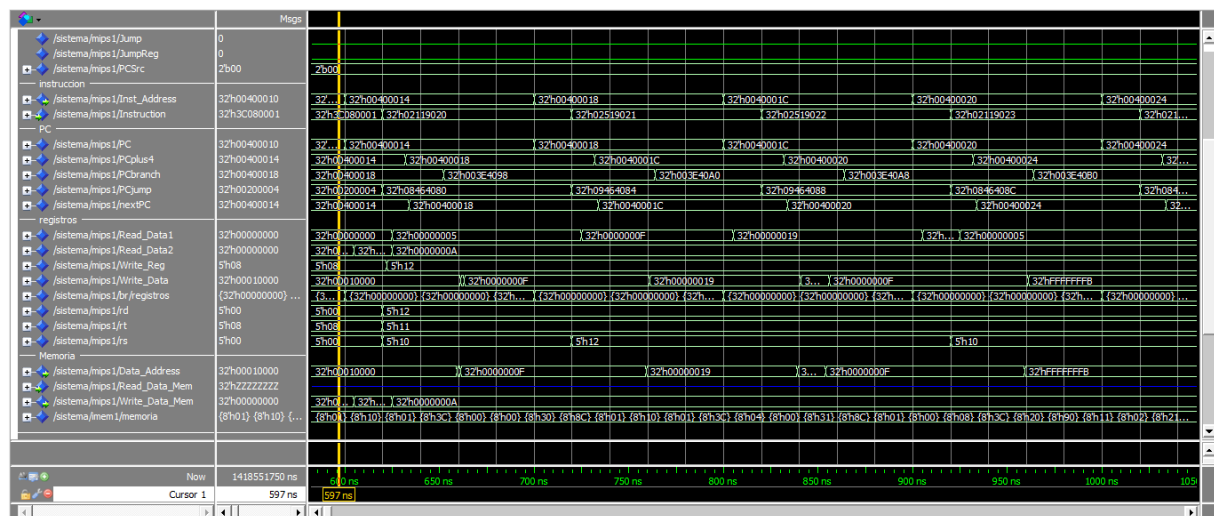
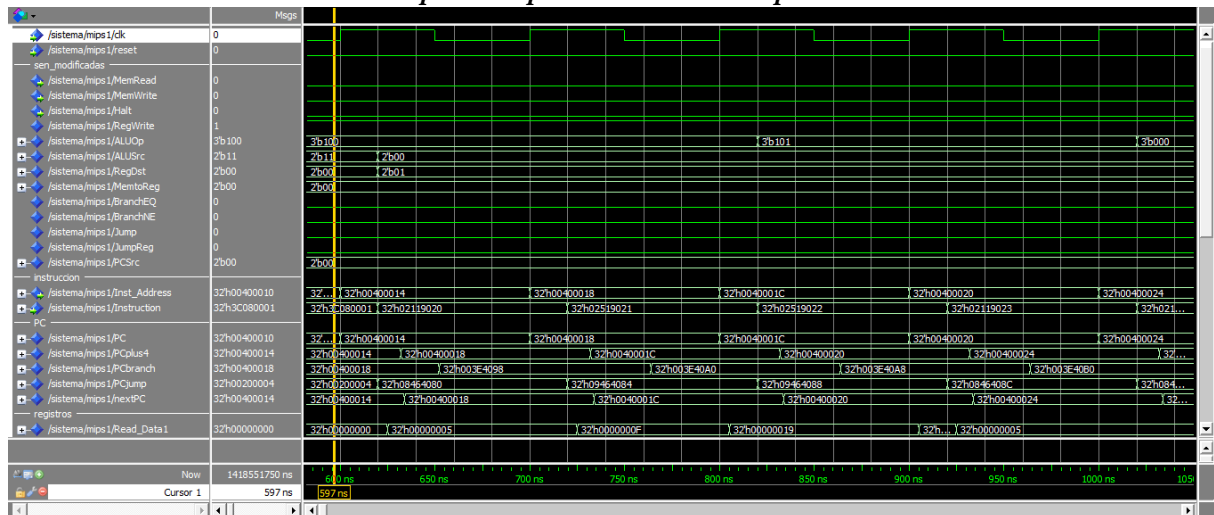
Comienza a los 600 ns y termina a los 700. Durante este tiempo pone las señales ALUOp a “100” para que haga una suma, ALUSrc a “00” para que pase al segundo operando de la ALU el registro rt. RegDst a “01” para indicar el registro en el que se va a guardar (rd). Regwrite a “1” para que funcione la parte 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.

Instrucción addu:

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

Comienza a los 700 ns y termina a los 800. Las señales que hemos modificado toman los mismos valores que en add.

A continuación se muestra una captura de pantalla de las dos operaciones con más detalle:



Instrucción sub:

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

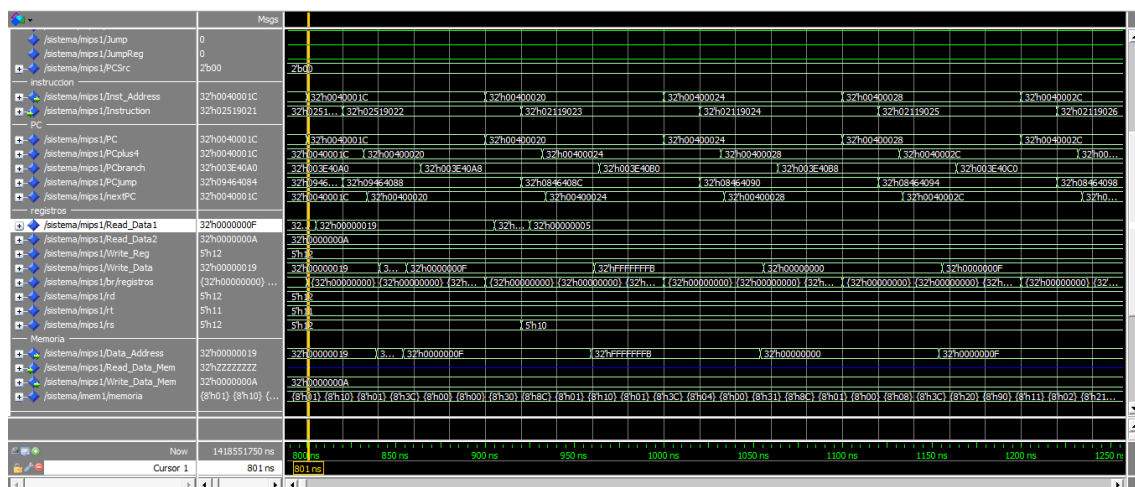
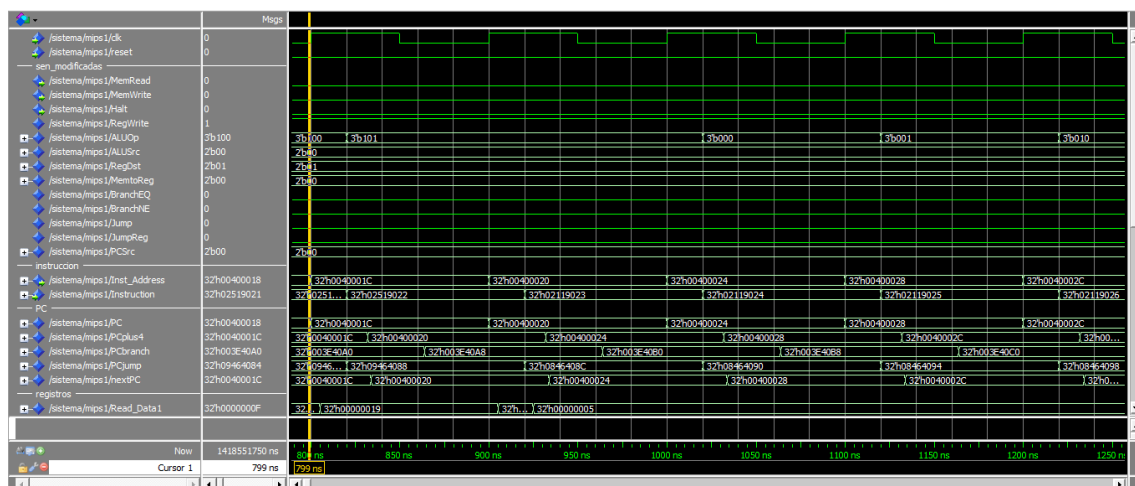
Comienza a los 800 ns y termina a los 900. Durante este tiempo pone las señales ALUOp a “101” para que haga una resta, ALUSrc a “00” para que pase al segundo operando de la ALU el registro rt. RegDst a “01” para indicar el registro en el que se va a guardar (rd). Regwrite a “1” para que funcione la parte 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.

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 900 ns y termina a los 1000. Las señales toman los mismos valores que en sub.

A continuación se muestra una captura de pantalla de las dos operaciones en VHDL con más detalle:



Instrucción and:

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

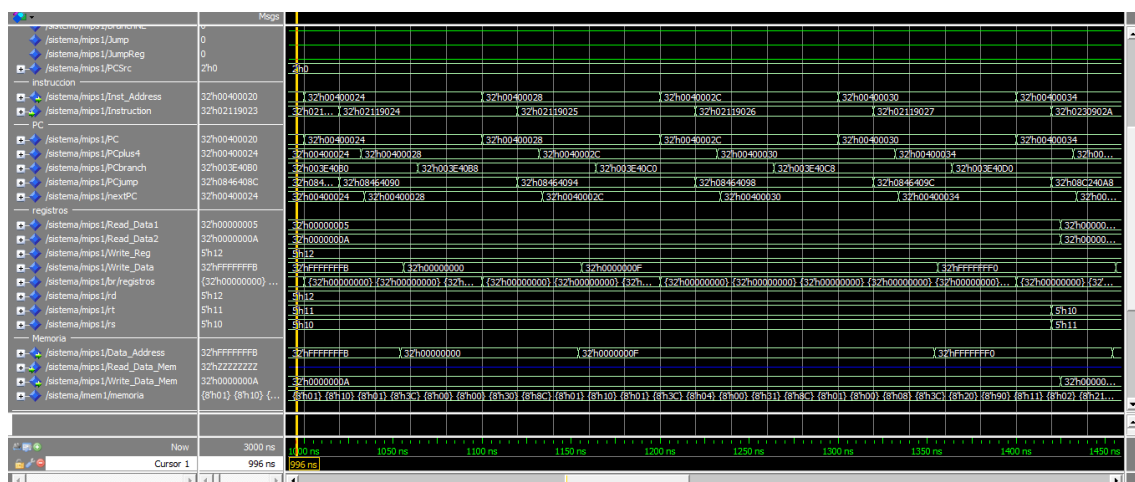
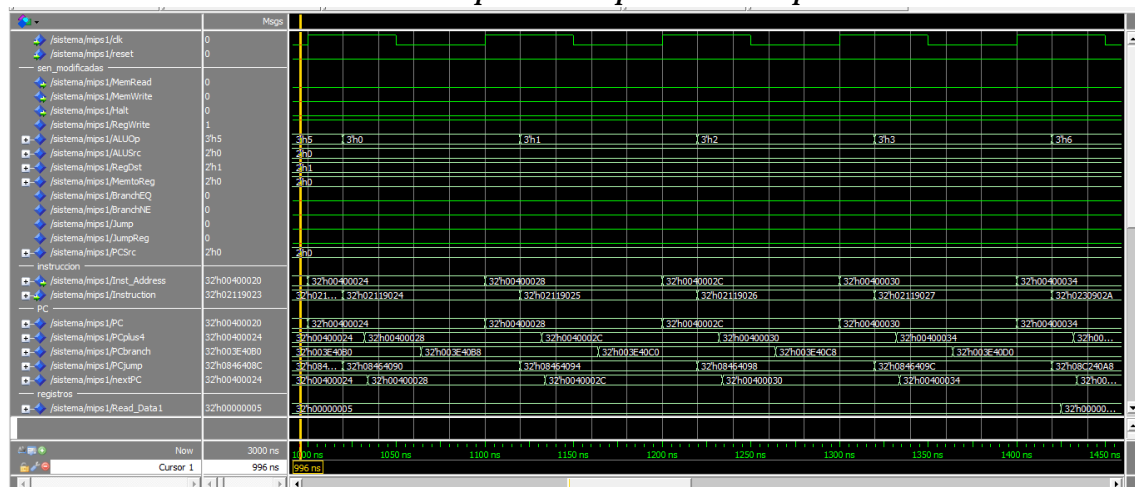
Comienza a los 1000 ns y termina a los 1100. Durante este tiempo pone las señales ALUOp a “000” para que haga un producto lógico (AND), ALUSrc a “00” para que pase al segundo operando de la ALU el registro rt. RegDst a “01” para indicar el registro en el que se va a guardar (rd). RegWrite a “1” para 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 or:

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

Comienza a los 1100 ns y termina a los 1200. 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.

A continuación se muestran las ondas producidas por estas dos operaciones con más detalle:



Instrucción xor:

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

Comienza a los 1200 ns y termina a los 1300. 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.

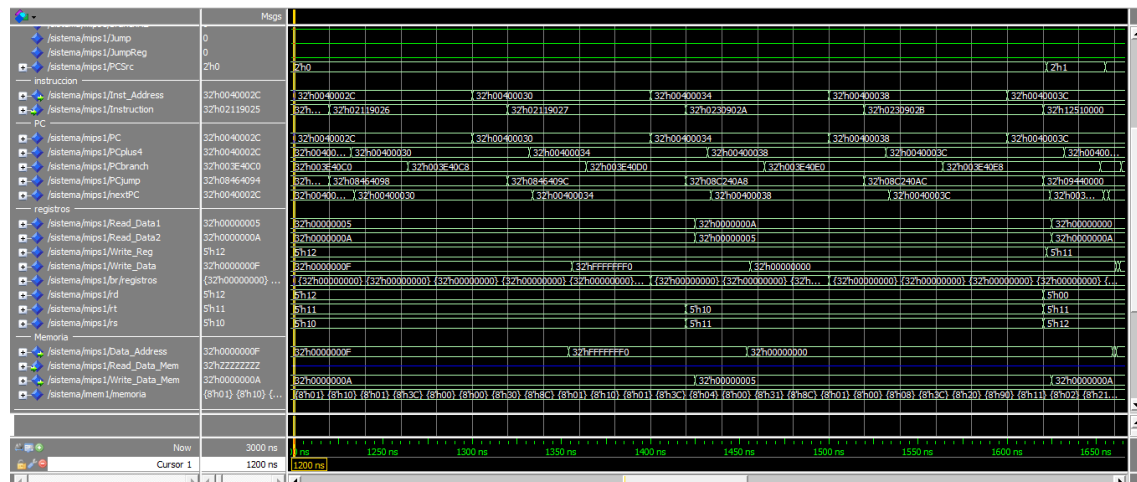
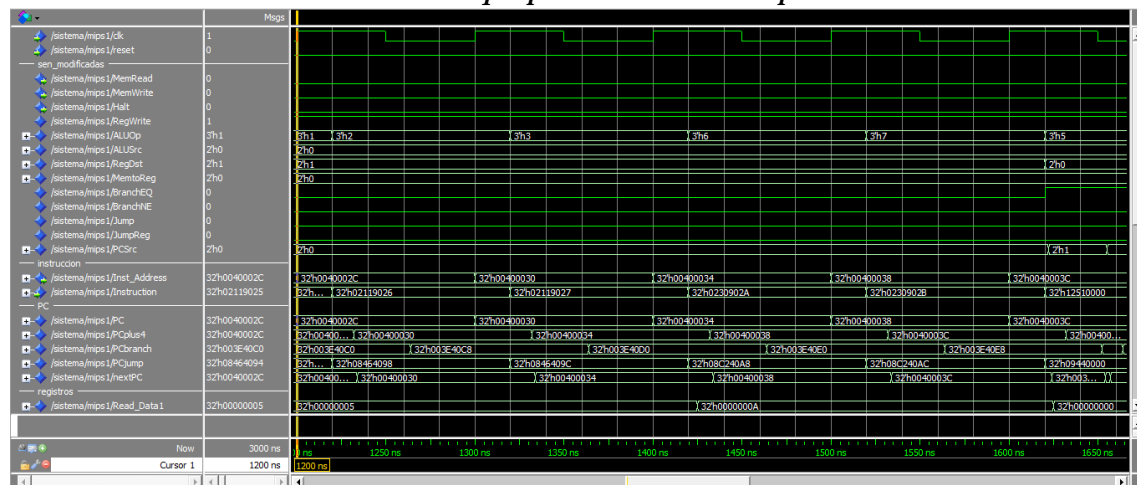
Instrucción nor:

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

Comienza a los 1300 ns y termina a los 1400. 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 se muestran las ondas que producen estas dos operaciones con más detalle:



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 1400 ns y termina a los 1500. 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).

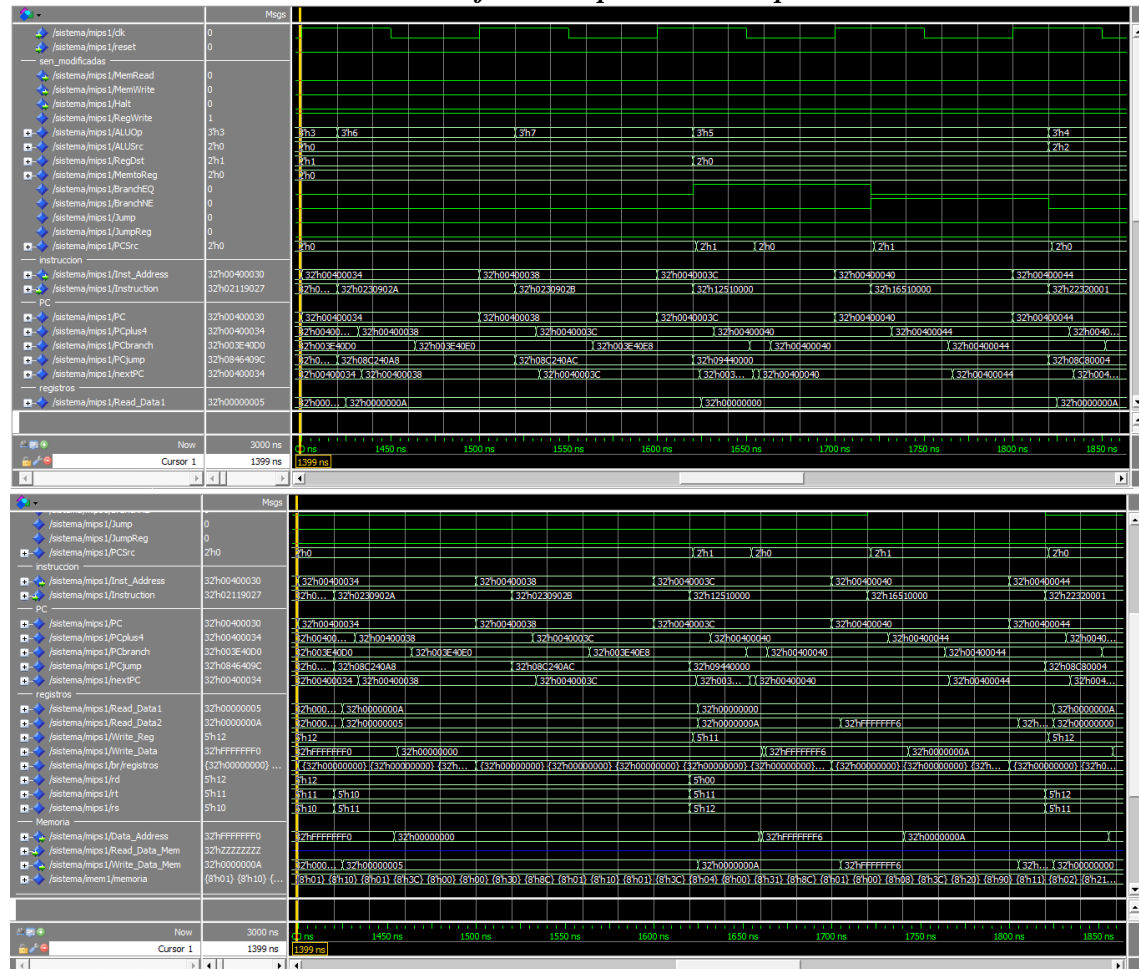
Regwrite a “1” para que funcione la parte de los registros. MemtoReg a “00” para que tome el valor nada mas salir de la ALU y lo escriba en rd. MemRead y MemWrite estará a “0” ya que lo que queremos es el valor que nos da la ALU y podemos omitir esa parte. PCSrc se pondrá a “0” para que pase PC+4 al 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 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 1500 ns y termina a los 1600. 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.

A continuación se muestran las ondas formadas por estas dos operaciones con más detalle:



Instrucción beq:

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

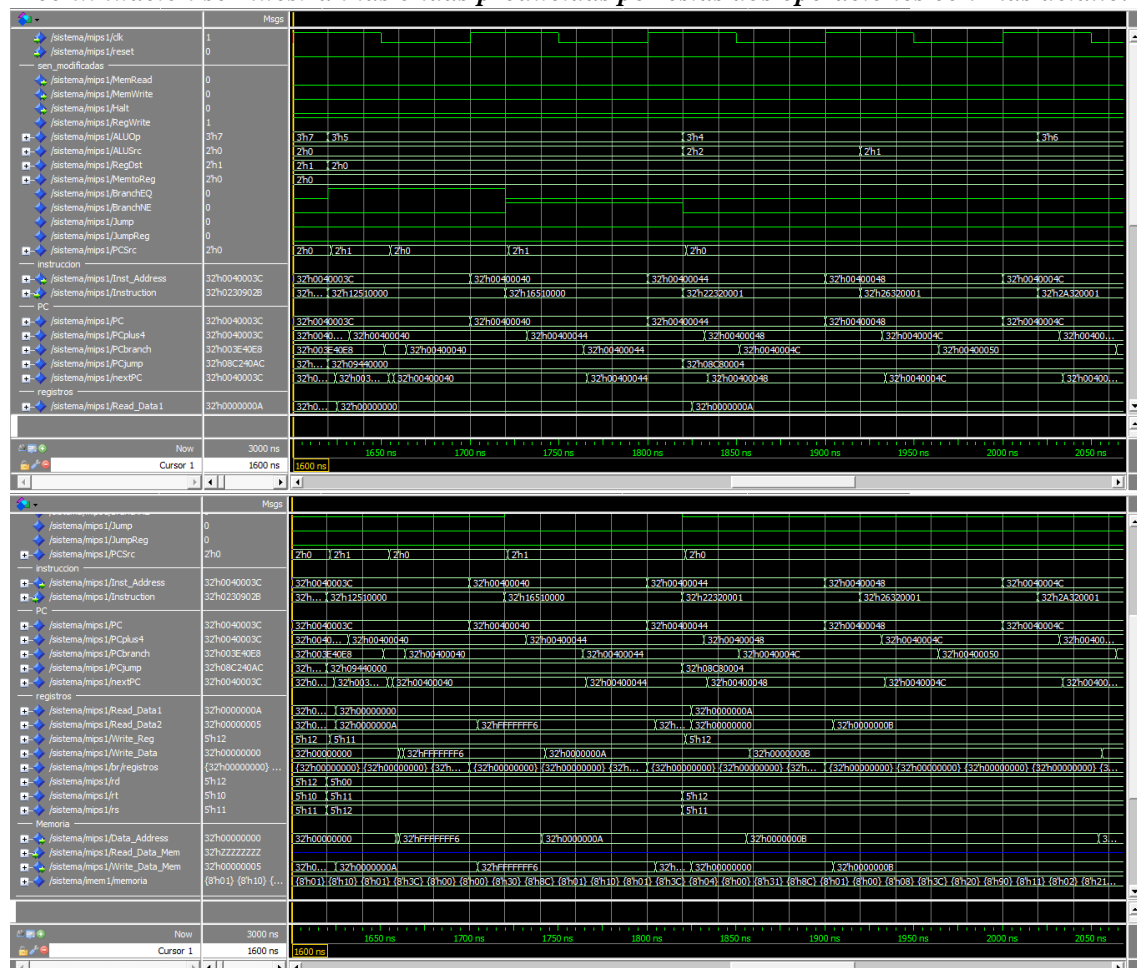
Comienza a los 1600 ns y termina a los 1700. Durante este tiempo pone las señales ALUOp a “101” para que la ALU reste rs y rt y comprobar si son iguales. ALUSrc a “00” para que pase rt al segundo operando de la ALU. RegDst a “00” para que se guarde el registro de la última operación. RegWrite a “1” para 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”.

Instrucción bne:

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

Comienza a los 1700 ns y termina a los 1800. 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 ondas producidas por estas dos operaciones con mas detalle:



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.

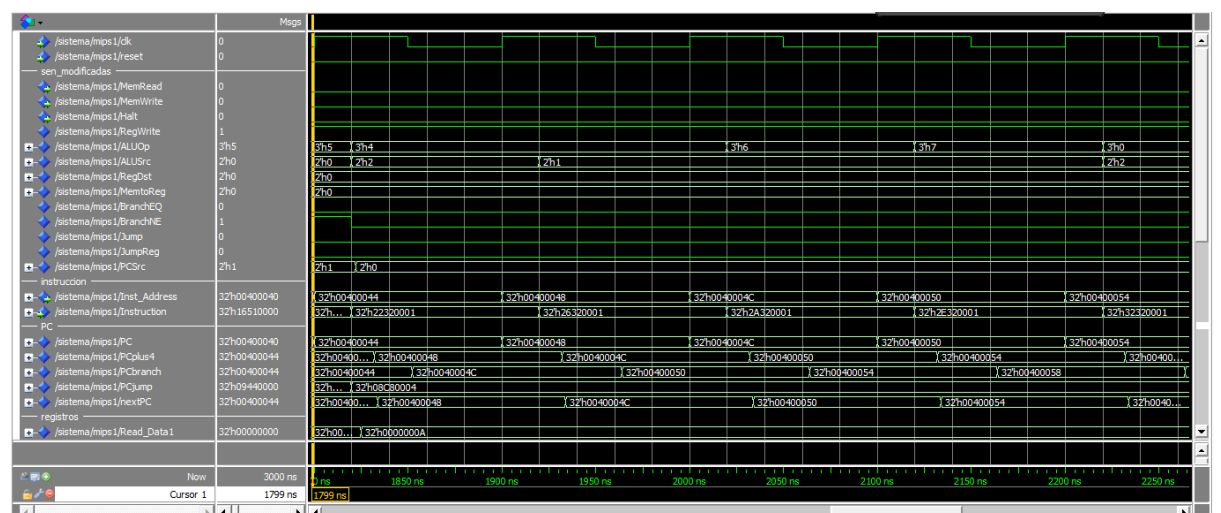
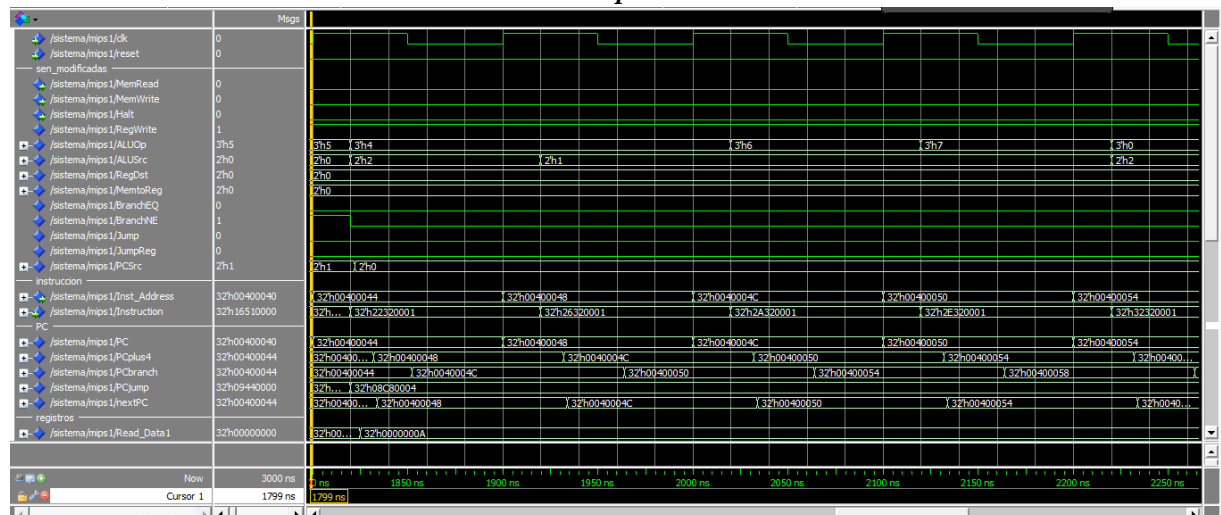
Comienza a los 1800 ns y termina a los 1900. Durante este tiempo pone las señales ALUOp a “000” para que haga un producto lógico (AND), ALUSrc a “10” para que pase al segundo operando de la ALU el inmediato con extensión en ceros. RegDst a “00”. Regwrite a “1” para que funcione la parte de los registros. 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 addiu:

Suma con inmediato. En nuestro circuito va a funcionar exactamente igual que addi.

Comienza a los 1900 ns y termina a los 2000. Las señales se ponen a los mismos valores que addi.

A continuación se muestran las ondas de las dos operaciones con más detalle:



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.

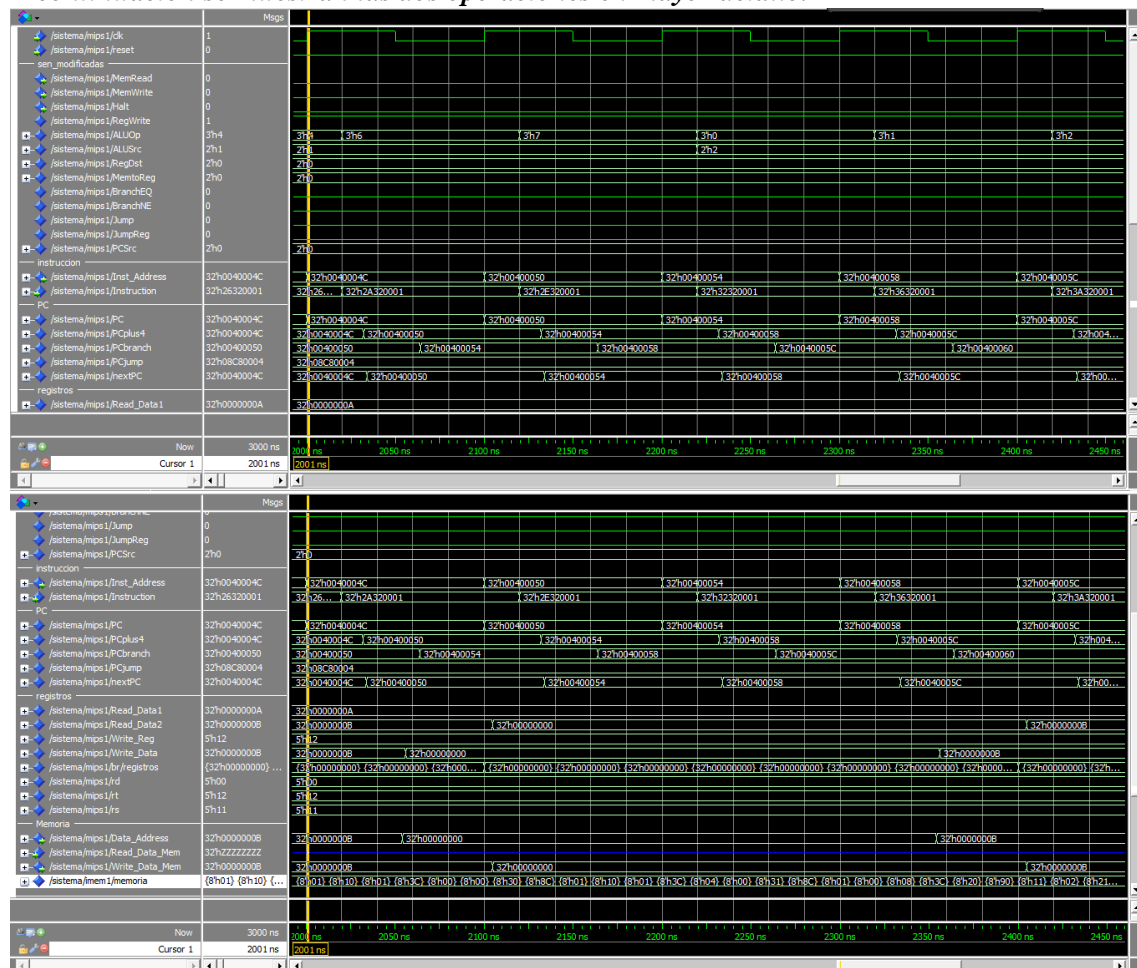
Comienza a los 2000 ns y termina a los 2100. Durante este tiempo pone las señales ALUop a "110" para que compare los operandos, ALUsrc a "01" para que pase al segundo operando de la ALU el inmediato con extensión en signo. RegDst a "00". Regwrite a "1" para que funcione la parte de los registros. MemtoReg 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.

Instrucción sltiu:

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

Comienza a los 2100 ns y termina a los 2200. 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.

A continuación se muestran las dos operaciones en mayor detalle:



Instrucción andi:

Hace el producto lógico bit a bit con inmediato.

Comienza a los 2300 ns y termina a los 2400. 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.

Instrucción Ori:

Suma lógica bit a bit con inmediato.

Comienza a los 2400 ns y termina a los 2500. 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.

Instrucción Xori:

Suma lógica exclusiva bit a bit con inmediato.

Comienza a los 2500 ns y termina a los 2600. 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.

A continuación se muestran las 3 operaciones con mas detalle en la simulación de multisim :

