Final project - Microprocessor

Integrantes:

- Eduardo Sebastián Castro Quispe 201910111
- Efraín Córdova Amaya 201910129

Lab 7: Single - Cycle Processor

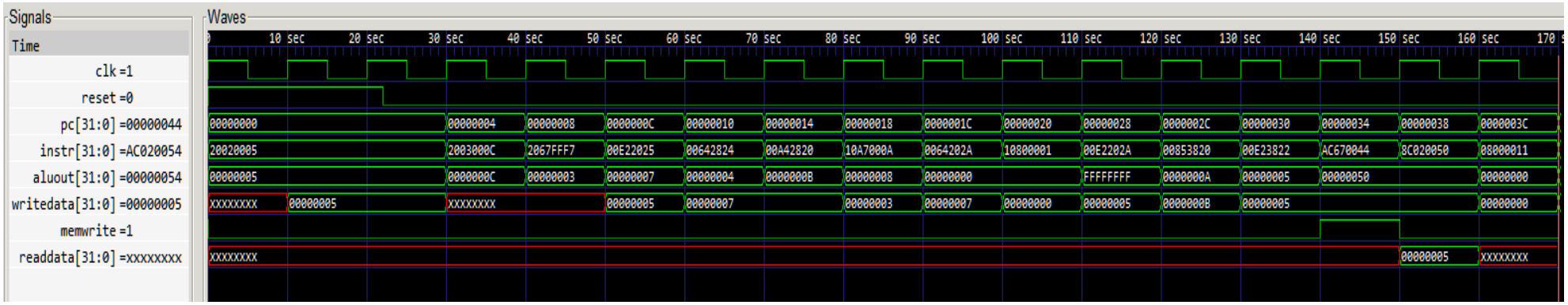
- Descripción de la tarea: Para este laboratorio, se nos pide conectar nuestro ALU del laboratorio 5 a un single-cycle processor. Luego, se debe crear un programa que testee el funcionamiento de este a través de una secuencia de instrucciones. Finalmente, se implementarán dos nuevas operaciones al microprocesador y se evaluarán con un segundo conjunto de instrucciones.

1 - Table 1 - Understanding instructions:

En los archivos del informe puede encontrar un .xlsx con la tabla en excel.

Cycle	Reset	PC (Hexadecimal)	Instr	Branch	Src A (Hexadecimal)	Src B (Hexadecimal)	Alu out (decimal)	Zero	PC Src (Hexadecimal)	Writedata	Memwrite	Read data
1	1	00	addi $s2, $0, 5 **20020005**	0	0	5	5	0	0	0	0	X
2	0	04	addi $s3, $0, 12 **2003000C**	0	0	c	c	0	0	0	0	X
3	0	08	addi $s7, $s3, -9 **20067FFF7**	0	c	-9	3	0	0	0	0	X
4	0	0C	or $s4 $s7 $s2 **00E22025**	0	3	5	7	0	0	0	0	X
5	0	10	and $s5, $s3, $s4 **00642824**	0	c	7	4	0	0	0	0	X
6	0	14	add $s5, $s5, $s4 **00A42820**	0	4	7	11	0	0	0	0	X
7	0	18	beq $s5, $s7, 0x000A **10A7000A**	0	b	3	0	0	18	0	0	X
8	0	1C	slt $s4, $s3, $s4 **0064202A**	0	c	7	0	1	0	0	0	X
9	0	20	beq $s4, $0, 0x0001 **10800001**	4 -> (PC´ = PC + 4 + 4)	0	0	1	0	20	0	0	X
10	0	28	slt $s4, $s7, $s2 **00E2202A**	0	3	5	-1 (FFFFFFFF)	1	0	0	0	X
11	0	2C	add $s7, $s4, $s5 **00853820**	0	-1	b	10	0	0	0	0	X
12	0	30	sub $s7, $s7, $s2 **00E23822**	0	a	5	5	0	0	0	0	X
13	0	34	sw $s7, 0x0044($s3) mem[68(0x44 on decimal) + 12 ($s3 data)] = 5 ($7 data) **AC670044**	0	x	x	x	x	0	1	1	0
14	0	38	lw $s3, 0x0050($0) $s3 data = mem[80 (0x50 on decimal)] **8C020050**	0	x	x	x	x	3C	x	x	1
15	0	3C	j 0x0000011 PC´ = 44 [11*4 (word allign)] **08000011**	PC´ = 44	x	x	x	x	x	x	x	X
16	0	44	sw $s7, 0x0054($0) mem[84 (0x54 on decimal)] = 5 ($s7 data) **AC020054**	0	x	x	x	x	x	1	1	0

2- Simulación del programa



Una vez conectamos nuestro ALU al procesador e identificamos posibles errores de implementación, obtenemos las siguientes señales en nuestro programa. Rápidamente, y tal como lo notamos al realizar la tabla 1, obtenemos un primer “Simulation failed” por parte del sistema. Ello se debe al valor final de writedata, el cual no coincide con el valor esperado (7). ¿Cómo explicamos esto? Recordemos, por un momento, el funcionamiento de la operación slt de nuestro ALU:

```
module slt_(input [31:0] slt_a, slt_b,
            output [31:0] result_slt);
  wire [31:0] slt_comp;
  wire [31:0] result;
  adder_ slt(slt_a,~slt_b,1'b1,slt_comp);
  msbext #(1,32) extsl(slt_comp[31], result);
  assign result_slt = result;
endmodule
```

Si nos fijamos en las operaciones, vemos que el output es de 32 bits, como debe ser todo resultado del ALU. Sin embargo, ¿cómo representamos 1-lógico en 32 bits? Para este slt, realizamos un sign (msb) extend. Ello implica que el valor que obtendremos en caso de cumplirse la condición de a < b es **32'hFFFFFFFF** y no **32'h00000001.** Recordemos que en nuestro procesador el trigésimo primer bit de un resultado es considerado como el signo negativo y que, además, usamos la regla de complemento 2 para este tipo de valores. En otras palabras, para operaciones del ALU, slt es -1 cuando la condición se cumple y 0 cuando no. Por ello, el resultado del addi $s7, $s4, $s5 es 10 (b-1) y no 12 (b+1). Esta política de implementación con el slt nos lleva a concluir que el valor que espera el programa se encuentra dos unidades por encima del real. Con el fin de mantener nuestro ALU como en anteriores labs, nos limitaremos a cambiar la condición que aprueba la funcionalidad del programa de writedata === 7 a **writedata === 5.** En los siguientes apartados de esta parte del informe, tendremos que definir otra política de extensión de bits menores a 32, y, para ese casos, resultará más interesante que hacer cambiar un condicional.

Entonces, cambiamos el módulo de forma que el condicional se active cuando writedata = 5

```
if (dataadr === 84 & writedata === 5) begin
    $display("Simulation succeeded");
    $stop;
```

Finalmente, y luego de cambiar el condicional, obtenemos un "Simulation succeeded" por parte del terminal.

```
Simulation succeeded
** VVP Stop(0) **
** Flushing output streams.
** Current simulation time is 175 ticks.
>
```

Recordar que el módulo testbench final sería:

```
module testbench();
    reg clk;
    reg reset;

    wire [31:0] writedata, dataadr;
    wire        memwrite;

    top dut(clk, reset, writedata, dataadr, memwrite);

    initial begin
        reset <= 1; #22; reset <= 0;
    end

    always begin
        clk <= 1; #5; clk <= 0; #5;
    end

    //In case of memfile
    always @ (negedge clk)
    begin
        if (memwrite) begin
            if (dataadr === 84 & writedata === 5) begin
                $display("Simulation succeeded");
                $stop;
            end else if (dataadr !== 80) begin
                $display("Simulation failed");
                $stop;
            end
        end
    end

    initial begin
      $dumpfile("mips.vcd");
      $dumpvars;
    end
endmodule
```

3 & 4 - Nuevas instrucciones:

ORI: Para esta instrucción, debemos implementar una lógica que nos permita realizar la operación lógica OR utilizando el aluop, de forma que podamos definirla en el main_decoder.v. Por otra parte, necesitamos implementar un zero-extend de 16 a 32 bits para el inmidiate que contiene instr[15:0]. Para el slt aún era debatible si aplicar un extend del msb era o no una política acertada, pero para esta implementación es necesario agregar un zero-extend. Adicionalmente, debemos cambiar la lógica del alusrc, ya que ahora será un selector de 2 bits, no de únicamente 1 bit, y cambiar srcbmux para que sea un mux3, no un mux2. Lo siguiente es implementar la lógica que seleccionará al zero-extend de instr[15:0] cuando alusrc coincida con el patrón que se decodifica cuando la operación es un ORI. Esto implica cambiar muy ligeramente la lógica del controller.

```
module aludec(input      [5:0]   funct,
              input      [1:0] aluop,
              output reg [2:0] alucontrol);
always@(*)
    case(aluop)
    2'b00:  alucontrol <= 3'b010; // add
    2'b01:  alucontrol <= 3'b110; // sub
    2'b11:  alucontrol <= 3'b001; // or (Our implementation)
    default: case(funct)
        6'b100000: alucontrol <= 3'b010; // add
        6'b100010: alucontrol <= 3'b110; // sub
        6'b100100: alucontrol <= 3'b000; // and
        6'b100101: alucontrol <= 3'b001; // or
        6'b101010: alucontrol <= 3'b111; // slt
        default:   alucontrol <= 3'bxxx; // ???
      endcase
    endcase
endmodule
```

```
//Maindec signals
always @ (*)
    case(op)                          alusrc [1:0]
    6'b000000: controls <= 11'b11000000010; //Rtyp
    6'b100011: controls <= 11'b10010001000; //LW
    6'b101011: controls <= 11'b00010010000; //SW
    6'b000100: controls <= 11'b00001000001; //BEQ
    6'b000101: controls <= 11'b00001100001; //BNE
    6'b001000: controls <= 11'b10010000000; //ADDI
    6'b000010: controls <= 11'b00000000100; //J
    6'b001101: controls <= 11'b10110000011; //ORI (Our implementation)
    default: controls <= 11'bxxxxxxxxxxx; //???
    endcase
endmodule
```

Se implementan estos dos módulos:

```
module zeroext (input [15:0] a,
                output [31:0] y);

assign y = {16'b0, a};

endmodule
```

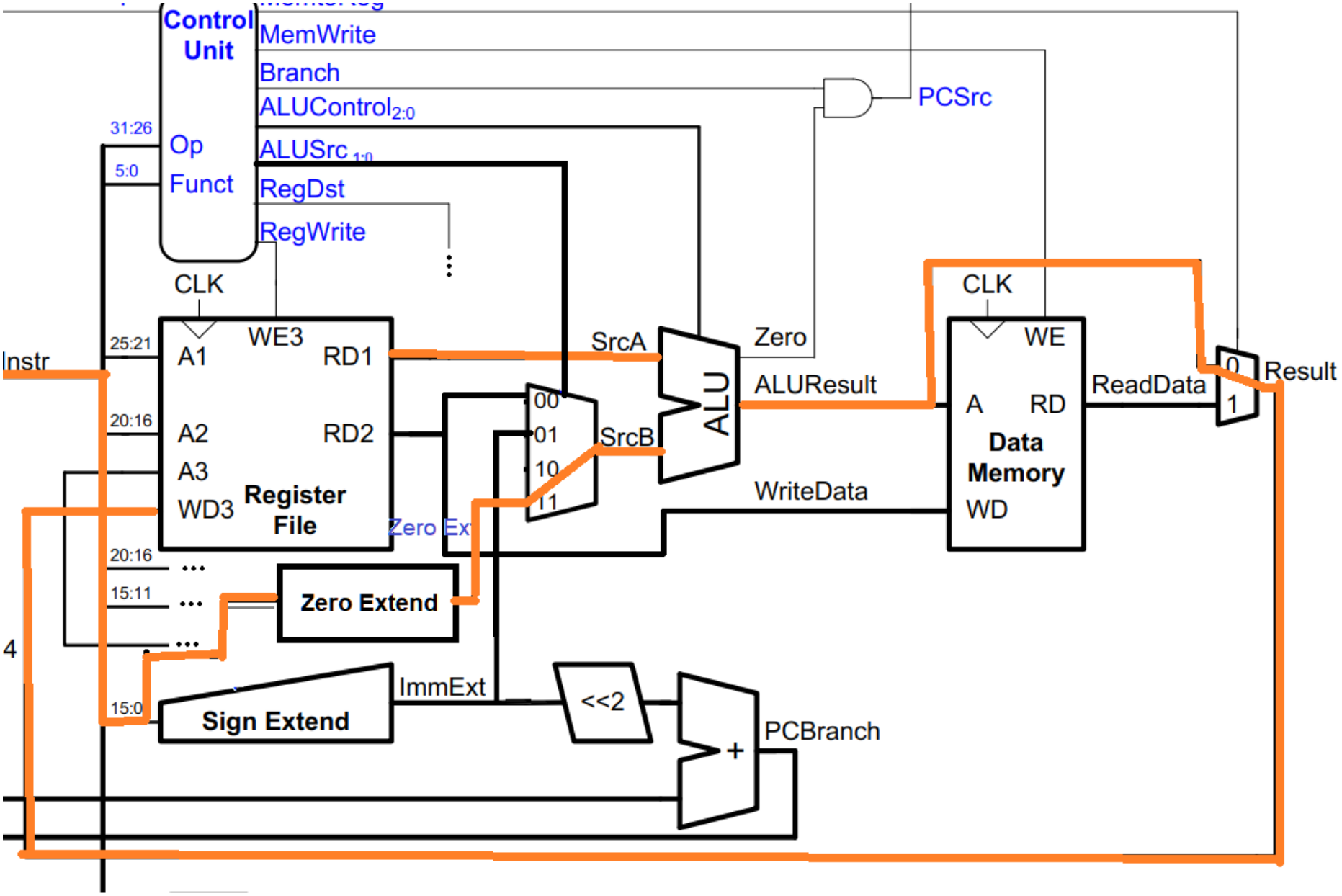
```
module mux3 # (parameter WIDTH = 8)
              (input [WIDTH-1:0] d0, d1, d2,
               input [1:0] s,
               output [WIDTH-1:0] y);
  assign y = s[1]? d2 : s[0]? d1: d0;
endmodule
```

Luego, adaptamos e implementamos los bloques en structural que los vayan a utilizar.

```
signext       se(instr[15:0], signimm);
zeroext       ze(instr[15:0], zeroimn);

// ALU logic
mux3 #(32)    srcbmux(writedata, signimm, zeroimn, alusrc, srcb);
```

El datapath para esta instrucción, sin contar la lógica del próximo pc, que sigue exactamente igual, es:


Control Unit
MemWrite
Branch
ALUControl2:0
ALUSrc1:0
RegDst
RegWrite
PCSrc
31:26
5:0
Op
Funct
CLK
Instr
25:21
20:16
A1
A2
A3
WD3
WE3
RD1
RD2
Register File
SrcA
SrcB
ALU
Zero
ALUResult
00
01
10
11
CLK
WE
A
RD
Data Memory
WD
ReadData
0
1
Result
WriteData
20:16
15:11
15:0
Zero Extend
Sign Extend
ImmExt
<<2
+
PCBranch
4

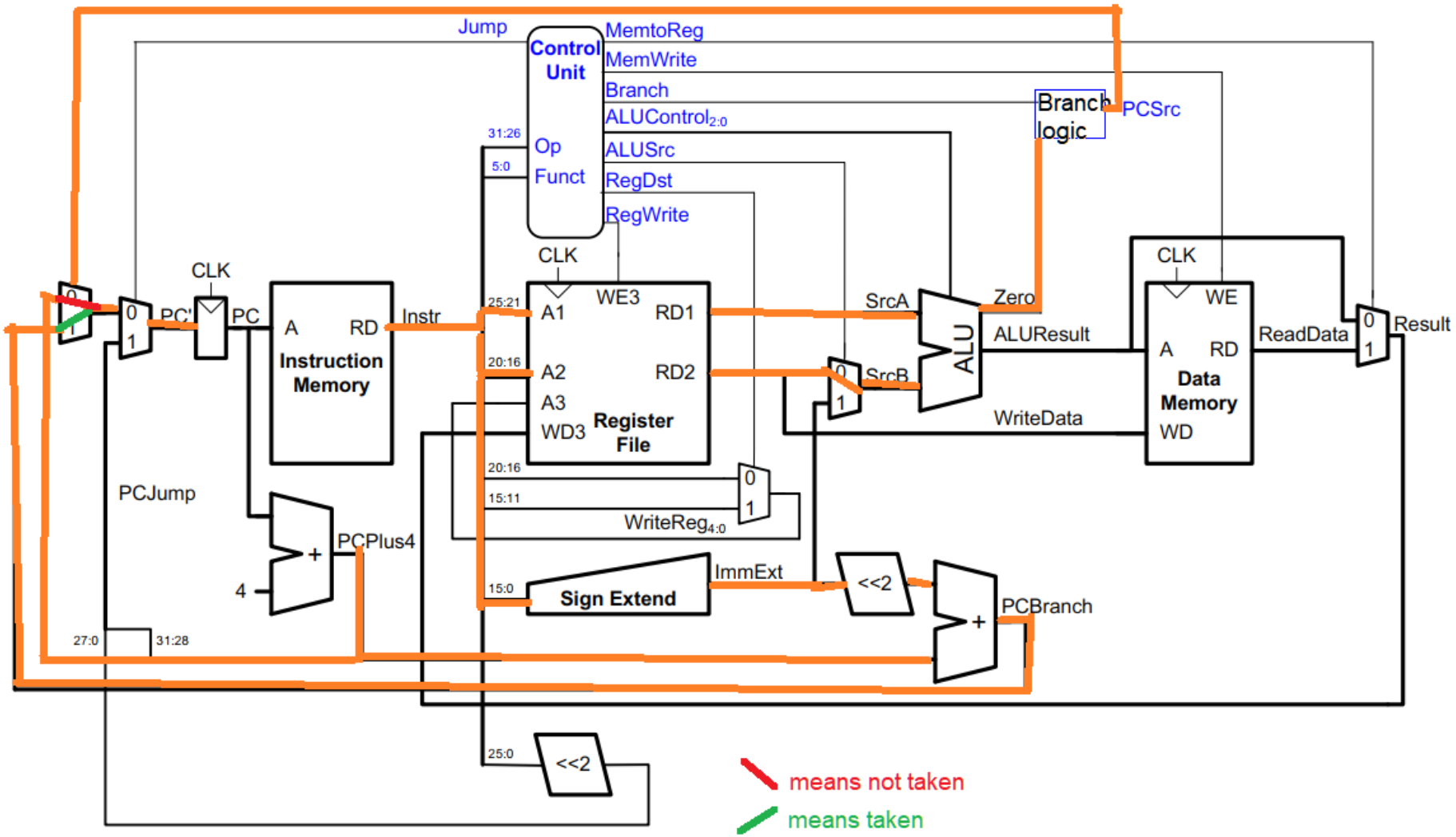
BNE: La lógica de implementación es similar a la realizada en ORI. Primero, debemos adaptar el branch para que sea un selector de dos bits y no únicamente de uno (esto nos permite elegir un resultado de entre 4 opciones como máximo), de forma analóga a alusrc en el ejercicio anterior. Luego, debemos hacer que durante la operación BNE el aluout sea de 01, para restar los operandos, al igual que en BEQ. A su vez, debemos implementar el branch de forma que su valor sea 11 para que la condición de comparación sea la de desigualdad. A diferencia de algunas señales, branch es un cable interno del módulo controller y es allí donde debemos limitar sus acciones e implementar los cambios referente a este. Anteriormente, teniamos que el pcsrc era equivalente a branch & zero. Esto no puede seguir siendo así, ya que branch tiene más posibles valores que sólo 0-lógico y 1-lógico. La idea es usar los bits de branch somo selectores y al literal zero y su complemento como operandos variables según esta señal. De esa forma, tendríamos implementada esta operación.

```
assign pcsrc = branch[0]? (branch[1] & ~zero) : (branch[1] & zero);
```

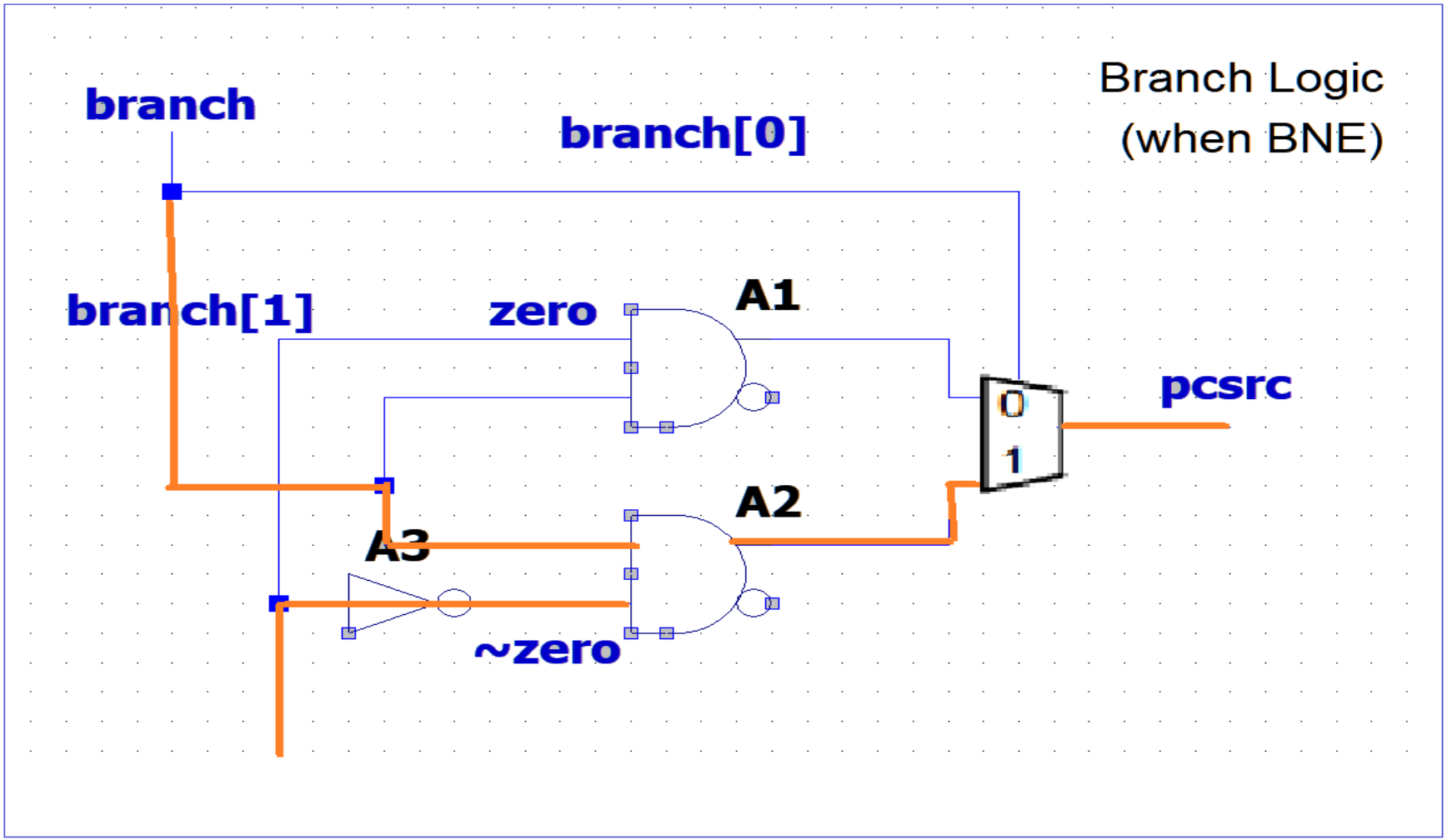
```
//Maindec signals
always @ (*)
    case(op)
    6'b000000: controls <= 11'b110_00_0_0010; //Rtyp
    6'b100011: controls <= 11'b100_00_1_1000; //LW
    6'b101011: controls <= 11'b000_00_1_0000; //SW
    6'b000100: controls <= 11'b000_10_0_0001; //BEQ
    6'b000101: controls <= 11'b000_11_0_0001; //BNE
    6'b001000: controls <= 11'b100_00_0_0000; //ADDI
    6'b000010: controls <= 11'b000_00_0_0100; //J
    6'b001101: controls <= 11'b101_00_0_0011; //ORI (Our implementation)
    default: controls <= 11'bxxxxxxxxxxx; //???
    endcase
```

branch [1:0]

Para concluir con BNE, se tiene el datapath de la instrucción:
(Nota: Este datapath cumple para branch taken y not taken, ya que ello depende del valor de pcsrc)


Jump
Control Unit
MemtoReg
MemWrite
Branch
ALUControl2:0
ALUSrc
RegDst
RegWrite
Op
Funct
31:26
5:0
Branch logic
PCSrc
CLK
PC'
PC
A
RD
Instruction Memory
Instr
25:21
20:16
A1
A2
A3
WD3
WE3
RD1
RD2
Register File
SrcA
SrcB
ALU
Zero
ALUResult
WriteData
CLK
WE
A
RD
Data Memory
WD
ReadData
Result
PCJump
PCPlus4
4
+
20:16
15:11
WriteReg4:0
15:0
Sign Extend
ImmExt
<<2
PCBranch
27:0
31:28
25:0
<<2
means not taken
means taken

El circuito combinacional branch logic sería:



Con nuestras instrucciones ya implementadas se nos pide completar las siguientes tablas:

ALUOp 1:0	Meaning
00	Add
01	Substract
10	Look at funct field
11	Or

Instruction	Op 5:0	RegWrite	RegDst	AluSrc 1:0	Branch 1:0	MemWrite	MemtoReg	ALUOp 1:0	Jump
R-type	000000	1	1	00	00	0	0	10	0
lw	100011	1	0	10	00	0	1	00	0
sw	101011	0	X	10	00	1	X	00	0
beq	000100	0	X	00	01	0	X	01	0
addi	001000	1	0	10	00	0	0	00	0
j	000010	0	X	X	XX	0	X	XX	1
ori	001101	1	0	11	11	0	0	11	0
bne	000101	0	X	00	00	0	X	01	0

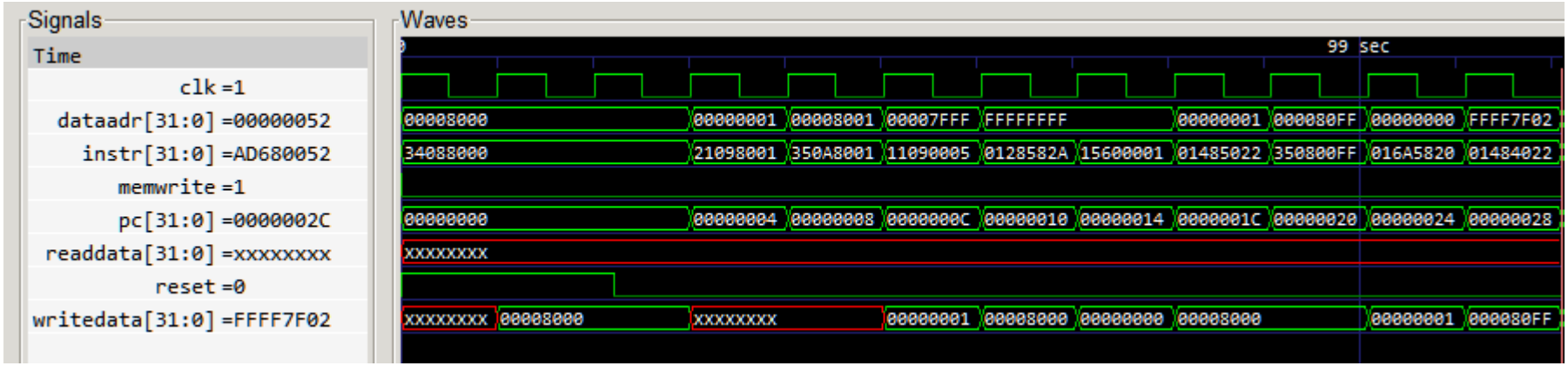
Y con ello, finalmente, podemos pasar al testeo de las instrucciones añadidas.

5 - Transcribir las instrucciones a hexadecimal:

```
#Assembly Code
main:       ori  $t0, $0,  0x8000     34088000
            addi $t1, $0,  -32768     20098001
            ori  $t2, $t0, 0x8001     350A8001
            beq  $t0, $t1, there      11090005
            slt  $t3, $t1, $t0        0128582A
            bne  $t3, $0,  here       15600001
            j    there                08000009
here:       sub  $t2, $t2, $t0        01485022
            ori  $t0, $t0, 0xFF       350800FF
there:      add  $t3, $t3, $t2        016A5820
            sub  $t0, $t2, $t0        01484022
            sw   $t0, 82($t3)         AD680052
```

(Nota: La segunda instrucción, addi $t1, $0, -32768; asume que -32768 = 15´h8001, ya que -32768 = -2^15)

6 - Simulación de las nuevas instrucciones:



Adicionalmente, se ha modificado el testbench para que arroje “Simulation succeeded” si el valor final es 15´hFFFF7F02.

Finalmente, se tiene el módulo testbench editado para estas nuevas instrucciones. Con esto último, hemos terminado la parte de Single-Cycle del informe.

```
module testbench();
    reg clk;
    reg reset;

    wire [31:0] writedata, dataadr;
    wire        memwrite;

    top dut(clk, reset, writedata, dataadr, memwrite);

    initial begin
        reset <= 1; #22; reset <= 0;
    end

    always begin
        clk <= 1; #5; clk <= 0; #5;
    end

    //In case of test2
    always @ (negedge clk)
    begin
        if (memwrite) begin
            if (dataadr === 32'h52 & writedata === 32'hFFFF7F02) begin
                $display("Simulation succeeded");
                $stop;
            end else begin
                $display("Simulation failed");
                $stop;
            end
        end
    end

    initial begin
      $dumpfile("test2.vcd");
      $dumpvars;
    end
endmodule
```

Lab 8: Multicycle Processor - Part 1: Controller

- Descripción de la tarea: En esta parte del MCP, se pide terminar el módulo controller. Ello contempla diseñar casi completamente la FSM del procesador, así como las señales que se activarán según que tipo de instrucción este siendo decodificada. Para concluir, se debe desarrollar un programa de test y comprobar el correcto funcionamiento del módulo y sus bloques subordinados.

1 - Main Decoder Output Table:

State Machine	PCWrite	MemWrite	IRWrite	RegWrite	ALUSrcA	Branch	IorD	MemtoReg	RegDst	ALUSrcB [1:0]	PCSrc [1:0]	ALUop [1:0]	FSM Control Word
0 (Fetch)	1	0	1	0	0	0	0	0	0	01	00	00	0x5010
1 (Decode)	0	0	0	0	0	0	0	0	0	11	00	00	0x0030
2 (MemAdr)	0	0	0	0	1	0	0	0	0	10	00	00	0x0420
3 (MemRd)	0	0	0	0	0	0	1	0	0	00	00	00	0x0100
4 (MemWB)	0	0	0	1	0	0	0	1	0	00	00	00	0x0880
5 (MemWr)	0	1	0	0	0	0	1	0	0	00	00	00	0x2100
6 (RtypeEx)	0	0	0	0	1	0	0	0	0	00	00	10	0x0402
7 (RtypeWB)	0	0	0	1	0	0	0	0	1	00	00	00	0x0840
8 (BeqEx)	0	0	0	0	1	1	0	0	0	00	00	01	0x0605
9 (AddiEx)	0	0	0	0	1	0	0	0	0	10	00	00	0x0420
10 (AddiWB)	0	0	0	1	0	0	0	0	0	00	00	00	0x1000
11 (JEx)	1	0	0	0	0	0	0	0	0	00	10	00	0x4008

(Nota: Control Word representa únicamente al output del main_decoder, no al de controller)

2 - Módulos MCP:

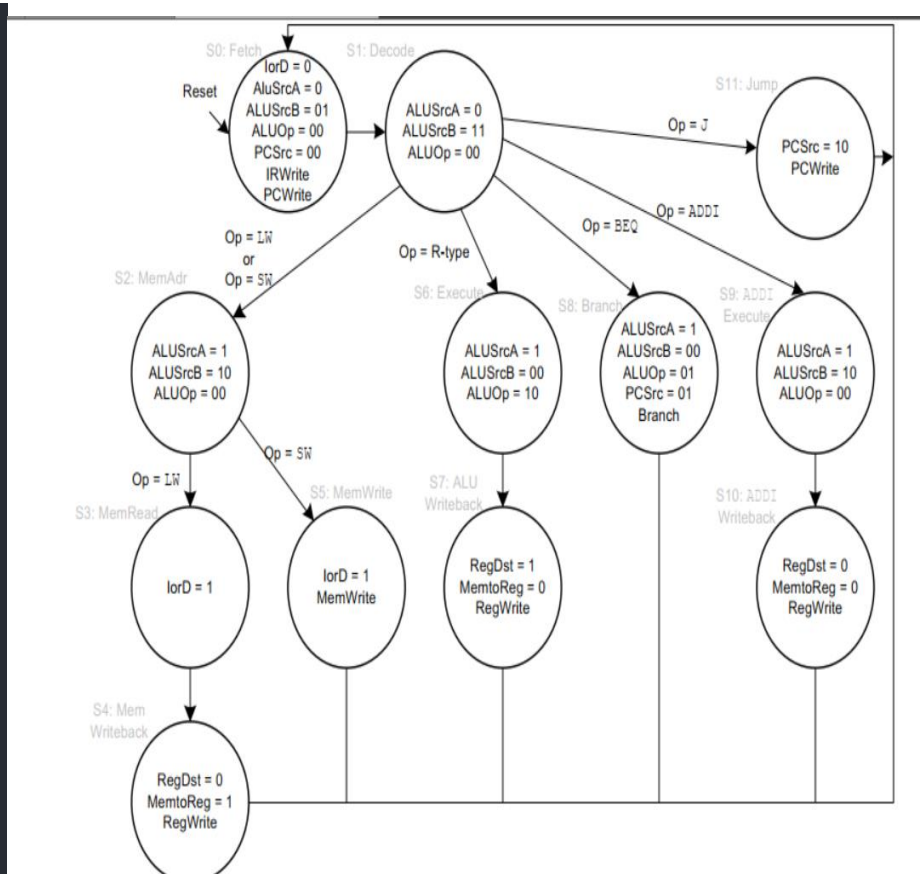
Módulo Aludec: Este módulo sigue siendo el mismo que en el laboratorio anterior (Single-cycle). No cuenta con las instrucciones ORI ni BNE.

```
module aludec(input  [5:0] funct,
              input  [1:0] aluop,
              output reg [2:0] alucontrol);

always@(*)
    case(aluop)
    2'b00:  alucontrol <= 3'b010; // add
    2'b01:  alucontrol <= 3'b110; // sub
    default: case(funct)
            6'b100000: alucontrol <= 3'b010; // add
            6'b100010: alucontrol <= 3'b110; // sub
            6'b100100: alucontrol <= 3'b000; // and
            6'b100101: alucontrol <= 3'b001; // or
            6'b101010: alucontrol <= 3'b111; // slt
            default:   alucontrol <= 3'bxxx; // ???
        endcase
    endcase
endmodule
```

Módulo maindec: Este módulo es el que contiene la FSM del procesador. Dados los parámetros de cada fase, el state register y la decodificación de los dos primeros estados, S0 (Fetch) y S1 (Decode), se pide implementar los otros 9 estados así como el output logic de la máquina de Moore.
Nos limitaremos únicamente a mostrar partes del código que hemos tenido que diseñar por nuestra cuenta. En ese sentido, se tiene el next state logic que hemos implementado para las instrucciones faltantes:

```
// next state logic
always @ (*)
  case(state)
    FETCH:   nextstate <= DECODE;
    DECODE:  case(op)
               LW:      nextstate <= MEMADR;
               SW:      nextstate <= MEMADR;
               RTYPE:   nextstate <= RTYPEEX;
               BEQ:     nextstate <= BEQEX;
               ADDI:    nextstate <= ADDIEX;
               J:       nextstate <= JEX;
               default: nextstate <= 4'bx; // should never happen
             endcase
    MEMADR: case(op)
               LW: nextstate <= MEMRD;
               SW: nextstate <= MEMWR;
               default: nextstate <= 4'bx; // should never happen
             endcase
    MEMRD: nextstate <= MEMWB;
    MEMWB: nextstate <= FETCH;
    MEMWR: nextstate <= FETCH;
    RTYPEEX: nextstate <= RTYPEWB;
    RTYPEWB: nextstate <= FETCH;
    BEQEX: nextstate <= FETCH;
    ADDIEX: nextstate <= ADDIWB;
    ADDIWB: nextstate <= FETCH;
    JEX: nextstate <= FETCH;
    default: nextstate <= 4'bx; // should never happen
  endcase
```



Luego, se tiene el output logic, que vendrían a ser las 15 señales respectivas a los estados que identificamos en la tabla 1:

```
// output logic

always @ (*) begin
    case(state)
        FETCH:    controls <= 15'h5010;
        DECODE:   controls <= 15'h0030;
        MEMADR:   controls <= 15'h0420;
        MEMRD:    controls <= 15'h0100;
        MEMWB:    controls <= 15'h0880;
        MEMWR:    controls <= 15'h2100;
        RTYPEEX:  controls <= 15'h0402;
        RTYPEWB:  controls <= 15'h0840;
        BEQEX:    controls <= 15'h0605;
        ADDIEX:   controls <= 15'h0420;
        ADDIWB:   controls <= 15'h1000;
        JEX:      controls <= 15'h4008;
        default: controls <= 15'hxxxx; // should never happen
    endcase
    pcwrite <= controls[14];
    memwrite <= controls[13];
    irwrite <= controls[12];
    regwrite <= controls[11];
    alusrca <= controls[10];
    branch <= controls[9];
    iord <= controls[8];
    memtoreg <= controls[7];
    regdst <= controls[6];
    alusrcb <= controls[5:4];
    pcsrc <= controls[3:2];
    aluop <= controls[1:0];
end
```

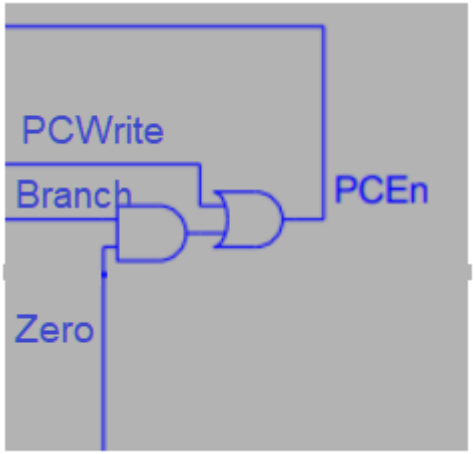
Módulo controller: Este módulo no es muy diferente a la versión final. Salvo los ajustes al pasarlo de SystemVerilog a Verilog, nuestra única modificación es implementar la lógica de pcen de acuerdo a la Figura 1

```
module controller(input        clk, reset,
                  input  [5:0] op, funct,
                  input        zero,
                  output       pcen, memwrite, irwrite, regwrite,
                  output       alusrca, iord, memtoreg, regdst,
                  output [1:0] alusrcb, pcsrc,
                  output [2:0] alucontrol);

  wire [1:0] aluop;
  wire       branch, pcwrite;

  // Main Decoder and ALU Decoder subunits.
  maindec md(clk, reset, op,
             pcwrite, memwrite, irwrite, regwrite,
             alusrca, branch, iord, memtoreg, regdst,
             alusrcb, pcsrc, aluop);
  aludec  ad(funct, aluop, alucontrol);

  assign pcen = (branch & zero) | pcwrite;
endmodule
```



Aunque el código implementado es autoexplicativo, hemos optado por hacer explícito lo que hemos diseñado. Para ello, se han elaborado tablas que resumen lo solicitado. Ha terminado siendo muy útil contar con ellas en el momento de insertar el código.

Table 2: State encoding of the Main decoder FSM

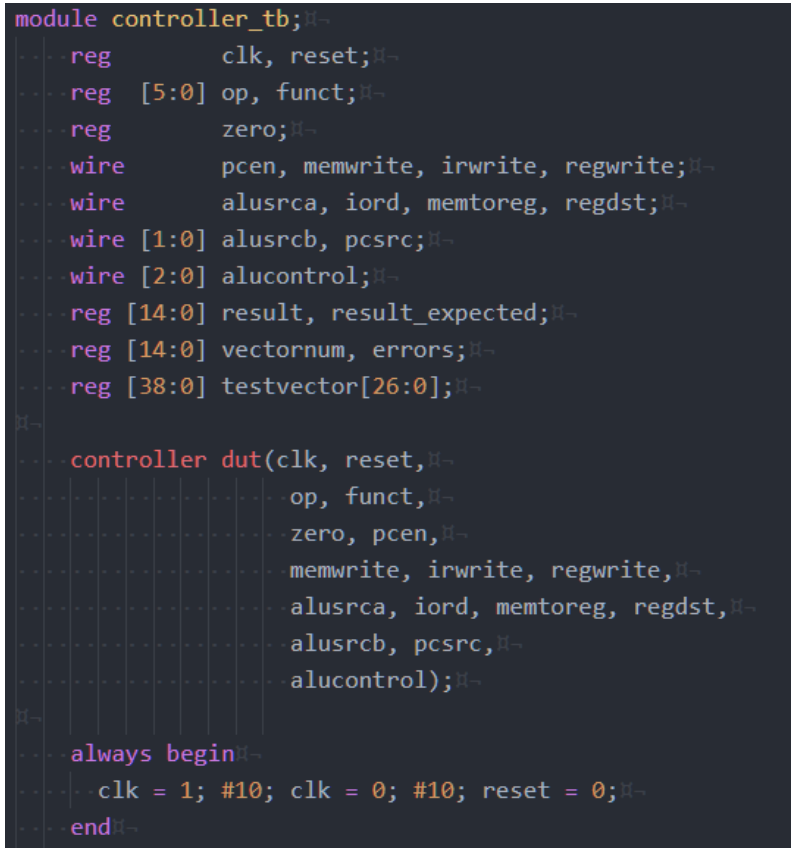
State	**Encoding $S_{3:0}$**
S0: Fetch	0000
S1: Decode	0001
S2: MemAdr	0010
S3: MemRead	0011
S4: Mem Writeback	0100
S5: MemWrite	0101
S6: Execute	0110
S7: ALU Writeback	0111
S8: Branch	1000
S9: ADDI Execute	1001
S10: ADDI Writeback	1010
S11: Jump	1011

Table 3: State transition with binary encoding

Current state | S	Inputs | $Op_{5:0}$	Next state | S'
S0: 0000	xxxxxx	S1: 0001
S1: 0001	10x011	S2: 0010
S1: 0001	000000	S6: 0110
S1: 0001	000100	S8: 1000
S1: 0001	001000	S9: 1001
S1: 0001	000010	S11: 1011
S2: 0010	100011	S3: 0011
S2: 0010	101011	S5: 0101
S3:0011	xxxxxx	S4: 0100
S4: 0100	xxxxxx	S0: 0000
S5: 0101	xxxxxx	S0: 0000
S6: 0110	xxxxxx	S7: 0111
S7: 0111	xxxxxx	S0: 0000
S8:1000	000100	S0: 0000
S9:1001	001000	S10: 1010
S10: 1010	XXXXXX	S0: 0000
S11: 1011	000010	S0: 0000

3 - Controllertest_xx testbench:

En esta ocasión, explicaremos más detenidamente el testbench, puesto que es de las pocas veces durante el proyecto que lo hemos implementado prácticamente desde cero y usando testvectors.
Lo primero viene a ser la inicialización del device under test, del clock y de nuestro reset.



```
module controller_tb;
    reg        clk, reset;
    reg  [5:0] op, funct;
    reg        zero;
    wire       pcen, memwrite, irwrite, regwrite;
    wire       alusrca, iord, memtoreg, regdst;
    wire [1:0] alusrcb, pcsrc;
    wire [2:0] alucontrol;
    reg [14:0] result, result_expected;
    reg [14:0] vectornum, errors;
    reg [38:0] testvector[26:0];

    controller dut(clk, reset,
                   op, funct,
                   zero, pcen,
                   memwrite, irwrite, regwrite,
                   alusrca, iord, memtoreg, regdst,
                   alusrcb, pcsrc,
                   alucontrol);

    always begin
        clk = 1; #10; clk = 0; #10; reset = 0;
    end
```

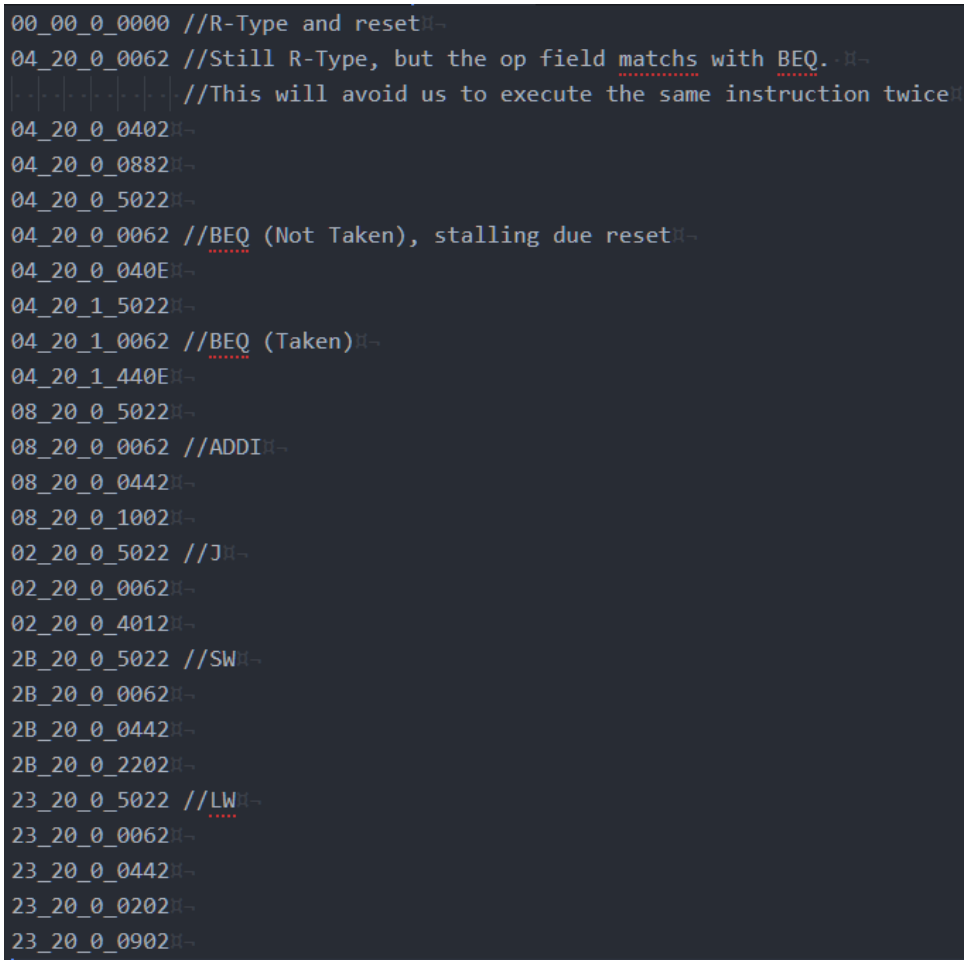
Después, excitamos la matriz testvector usando instrucciones generadas por nosotros. Esto lo hacemos a través de controller.tv, que contiene un total de 7 instrucciones que deberán ser cumplidas luego de 25 ciclos de reloj (reset incluido). Adicionalmente,

creamos una variable que contabilizará los errores de simulación. Finalmente, decodificamos la instrucción asignándoles a nuestras variables el valor que les corresponda.

```
initial begin
    $readmemh("controller.tv", testvector);
    //Vectornum is setted with -1 just to match with the correct clock cycle (after reset)
    vectornum = -1; errors = 0;
    reset = 1;
end

always @(posedge clk) begin
            op = testvector[vectornum][33:28];
            funct = testvector[vectornum][25:20];
            zero = testvector[vectornum][16];
            result_expected = testvector[vectornum][14:0];
end
```

Las instrucciones que contiene nuestro .tv, con el cual probamos nuestra implementación, son:



```
00_00_0_0000 //R-Type and reset
04_20_0_0062 //Still R-Type, but the op field matchs with BEQ.
             //This will avoid us to execute the same instruction twice
04_20_0_0402
04_20_0_0882
04_20_0_5022
04_20_0_0062 //BEQ (Not Taken), stalling due reset
04_20_0_040E
04_20_1_5022
04_20_1_0062 //BEQ (Taken)
04_20_1_440E
08_20_0_5022
08_20_0_0062 //ADDI
08_20_0_0442
08_20_0_1002
02_20_0_5022 //J
02_20_0_0062
02_20_0_4012
2B_20_0_5022 //SW
2B_20_0_0062
2B_20_0_0442
2B_20_0_2202
23_20_0_5022 //LW
23_20_0_0062
23_20_0_0442
23_20_0_0202
23_20_0_0902
```

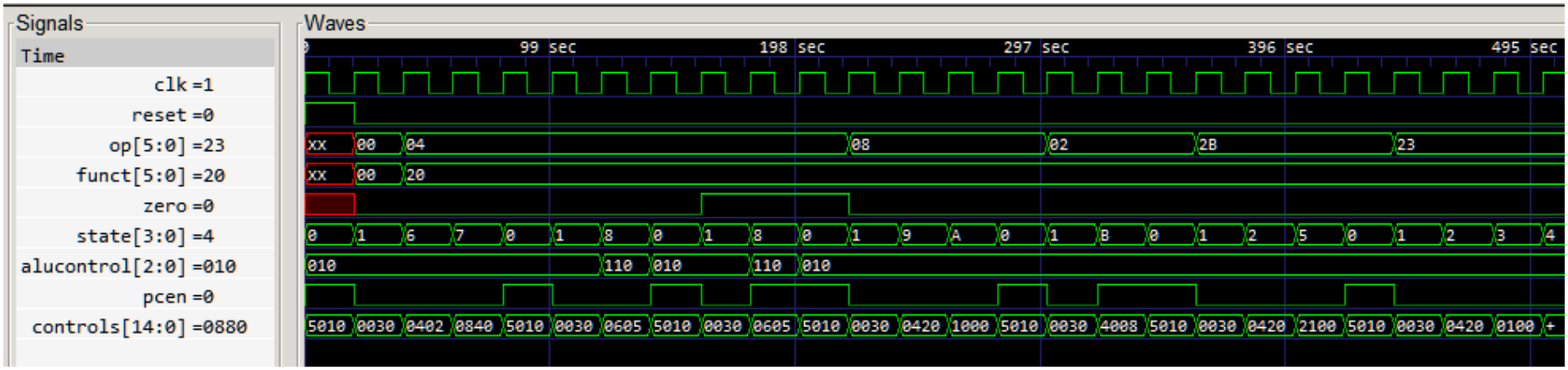
(Nota: Para evitar posibles errores de lectura, el .tv original no contiene ningún comentario. Insertamos el op de BEQ durante el reset para no perder esa instrucción en memoria)

Por último, comprobamos que el resultado sea el esperado. En caso de no serlo, nuestro contador errors irá aumentando. Imprimimos en la terminal la información necesaria para cada caso, generamos el .vcd respectivo y estamos listos para simular el programa.

```
always @(negedge clk) begin
    result <= {pcen, memwrite, irwrite, regwrite, alusrca, iord, memtoreg, regdst, alusrcb, pcsrc, alucontrol};
    if(vectornum) begin
        if (result !== result_expected) begin
            $display("\nErrors in vector %d", vectornum);
            $display(" Inputs: op = %b, funct = %b, zero = %b", op, funct, zero);
            $display(" Outputs: result = %h (%h expected)", result, result_expected);
            errors = errors + 1;
        end
    end
    vectornum = vectornum + 1;
    if (vectornum === 25) begin
        $display("%d tests (7 instructions) completed with %d error(s)",vectornum, errors);
        $finish;
    end
end

initial begin
    $dumpfile("controllertest.vcd");
    $dumpvars;
end
```

4 - Simulación del programa:



Podemos notar que la transición de estados es la esperada, así como la control word anexa a estos. La terminal también nos informa que no ha ocurrido ningún error en la simulación, lo que da cierre a la primera parte de la implementación del MIPS Multicycle.

```
VCD info: dumpfile controllertest.vcd opened for output.
   25 tests (7 instructions) completed with     0 error(s)
```

Lab 9: Multicycle Processor - Part 2: Datapath

- Descripción de la tarea: Como última parte del proyecto final, se pide implementar las conexiones entre las distintas partes del microprocesador a través de un bloque structural, el llamado datapath.

1 - Tabla 1 - Expected Outputs

Cycle	Reset	PC	Instr	(FSM) State	SrcA	SrcB	ALUResult	Zero	Control Word
1	1	00	0	0	00	04	04	0	0x5010
2	0	04	addi 20020005	1	00	05	05	0	0x0030
3	0	04	addi 20020005	9	00	05	05	0	0x0420
4	0	04	addi 20020005	10	00	05	05	0	0x0800
5	0	04	addi 20020005	0	00	05	05	0	0x5010
6	0	08	addi 2003000C	1	00	0C	0C	0	0x0030
7	0	08	addi 2003000C	9	00	0C	0C	0	0x0420
8	0	08	addi 2003000C	10	00	0C	0C	0	0x0800
9	0	08	addi 2003000C	0	00	0C	0C	0	0x5010
10	0	0C	addi 2067FF7	1	0C	-09	03	0	0x0030
11	0	0C	addi 2067FF7	9	0C	-09	03	0	0x0420
12	0	0C	addi 2067FF7	10	0C	-09	03	0	0x0800
13	0	0C	addi 2067FF7	0	0C	-09	03	0	0x5010
14	0	10	or 00E22025	1	03	05	07	0	0x0030
15	0	10	or 00E22025	6	03	05	07	0	0x0402
16	0	10	or 00E22025	7	03	05	07	0	0x0840
17	0	10	or 00E22025	0	03	05	07	0	0x5010
18	0	14	and 00642824	1	0C	07	04	0	0x0030
19	0	14	and 00642824	6	0C	07	04	0	0x0402
20	0	14	and 00642824	7	0C	07	04	0	0x0840
21	0	14	and 00642824	0	0C	07	04	0	0x5010
22	0	18	add 00A42820	1	04	07	0B	0	0x0030
23	0	18	add 00A42820	6	04	07	0B	0	0x0402
24	0	18	add 00A42820	7	04	07	0B	0	0x0840
25	0	18	add 00A42820	0	04	07	0B	0	0x5010
26	0	1C	beq 10A7000A	1	0B	03	09	0	0x0030
27	0	1C	beq 10A7000A	8	0B	03	09	0	0x0605
28	0	1C	beq 10A7000A	0	0B	03	09	0	0x5010
29	0	20	slt 0064202A	1	0C	07	-05	0	0x0030
30	0	20	slt 0064202A	9	0C	07	-05	0	0x0402
31	0	20	slt 0064202A	10	0C	07	-05	0	0x0840
32	0	20	slt 0064202A	0	0C	07	-05	0	0x5010
33	0	24	beq 10800001	1	00	00	FF	1	0x0030
34	0	24	beq 10800001	8	00	00	FF	1	0x0605
35	0	24	beq 10800001	0	00	00	FF	1	0x5010
36	0	2C	slt 00E2202A	1	03	05	FF	0	0x0030
37	0	2C	slt 00E2202A	9	03	05	FF	0	0x0402
38	0	2C	slt 00E2202A	10	03	05	FF	0	0x0840
39	0	2C	slt 00E2202A	0	03	05	FF	0	0x5010
40	0	30	add 00853820	1	-01	0B	10	0	0x0030
41	0	30	add 00853820	9	-01	0B	10	0	0x0402
42	0	30	add 00853820	10	-01	0B	10	0	0x0840
43	0	30	add 00853820	0	-01	0B	10	0	0x5010
44	0	34	sub 00E23822	1	0A	05	05	0	0x0030
45	0	34	sub 00E23822	9	0A	05	05	0	0x0402
46	0	34	sub 00E23822	10	0A	05	05	0	0x0840
47	0	34	sub 00E23822	0	0A	05	05	0	0x5010
48	0	38	sw AC670044	1	44	0C	50	0	0x0030
49	0	38	sw AC670044	2	44	0C	50	0	0x0420
50	0	38	sw AC670044	5	44	0C	50	0	0x2100
51	0	38	sw AC670044	0	44	0C	50	0	0x5010
52	0	3C	lw 8C020050	1	50	00	50	0	0x0030
53	0	3C	lw 8C020050	2	50	00	50	0	0x0420
54	0	3C	lw 8C020050	3	50	00	50	0	0x0100
55	0	3C	lw 8C020050	4	50	00	50	0	0x0880
56	0	3C	lw 8C020050	0	50	00	50	0	0x5010
57	0	40	j 08000011	1	44	00	44	0	0x0030
58	0	40	j 08000011	11	44	00	44	0	0x4008
59	0	40	j 08000011	0	44	00	44	0	0x5010
60	0	44	sw AC020054	1	54	00	54	0	0x0030
61	0	44	sw AC020054	2	54	00	54	0	0x0420
62	0	44	sw AC020054	5	54	00	54	0	0x2100

(Nota: Esta tabla y las demás se encuentran en sus respectivas carpetas según el laboratorio al que pertenecen)

2 - Structural Verilog Datapath:

Se implementó el diagrama del MIPS Multicycle que se nos facilitó en el laboratorio 8. Este módulo contiene las conexiones entre las diferentes partes del procesador y se encarga de gestionar su flujo.

Lo primero es proporcionarle al control unit las señales que necesita decodificar para asignar las señales. Adicionalmente, generamos la condición de reset para inicializar el pc en 0.

```
// Setting codes for control unit
assign op = instr[31:26];
assign funct = instr[5:0];

//"Activate" program when reset is asserted
always @ (*) begin
 if (reset) pc <= 0;
end
```

Luego, durante el posedge del clock signal, refrescamos y asignamos las respectivas para el siguiente ciclo

```
//Saving ALU output
always @ (posedge clk) begin
  aluout <= aluresult;
end

//Saving operands from register
always @ (posedge clk) begin
  a <= rd1;
  writedata <= rd2;
end

//Setting Instr and Data. Always on posedge
always @ (posedge clk) begin
 if(irwrite)  instr <=  readdata;
 data <= readdata;
end

//Going to next instruction
 always @ (posedge clk)begin
    if(pcen) pc <= pcnext;
 end
```

Debemos encargarnos de extender los 16 últimos bits de la instrucción para tenerlos hábiles si la instrucción así lo requiere. Algunas instrucciones requieren también que este pase por el módulo sl2.

```
//Adjusting some wires
signext    se(instr[15:0], signimm);

sl2        inmsl2(signimm, signimmsh);
```

A continuación llamamos al regfile para gestionar nuestra data memory. Este módulo almacena valores y los sobrescribe si la instrucción así lo amerita. También tenemos un mux de 5 (log2 de 32, puesto a que representan posiciones en memoria) para hacer fetch de los operandos. Luego, tenemos un mux que se encarga de hacer writeback para algunas operaciones del ALU.

```
//Regfile logic
regfile    regf(clk, regwrite, instr[25:21], instr[20:16], writereg, wd3, rd1, rd2);


mux2 #(5)  RegDstMux(instr[20:16], instr[15:11], regdst, writereg);

mux2 #(32) WD3Mux(aluout, data, memtoreg, wd3);
```

En relación a nuestro ALU, requerimos de dos multiplexores de 32 bits. El primero para filtrar si el primer operando será el operando que se solicita en la instrucción o el pc (en este último caso, se efectúa un pc + 4). El segundo mux hace lo propio con el segundo

operando, que esta vez puede ser el operando b, un 4 decimal, para sumarlo al pc y continuar leyendo el .dat o cualquiera de las dos señales extendidas, ya sea la sign extend o la sign extend shifted by 2. Y, por último, claramente debemos llamar al ALU per se.

```
//ALU Logic
mux2 #(32) SrcAmux(pc,a,alusrca,srca);

mux4 #(32) SrcBmux(writedata, 32'b100, signimm, signimmsh, alusrcb, srcb);

alu        alu(srca,srcb,alucontrol,aluresult,zero);
```

Finalmente, tenemos la lógica del próximo pc. El primer mux nos dirá si el pc next será igual a pc+4, pc+4+branch o un jump que se concatena en base a los primeros 4 bits del pc actual, los 25 bits de un immediate en J-instructions y 2 bits de 00 para que el valor sea word-aligned. También tenemos otro multiplexor que maneja la lógica del next address.

```
//NextPC Logic
mux3 #(32) Nextmux(aluresult, aluout, {pc[31:28], {instr[25:0], 2'b00} }, pcsrc, pcnext);


mux2 #(32) PCadrMux(pc,aluout,iord,adr);
```

Recordar el diagrama que se usó para la implementación:

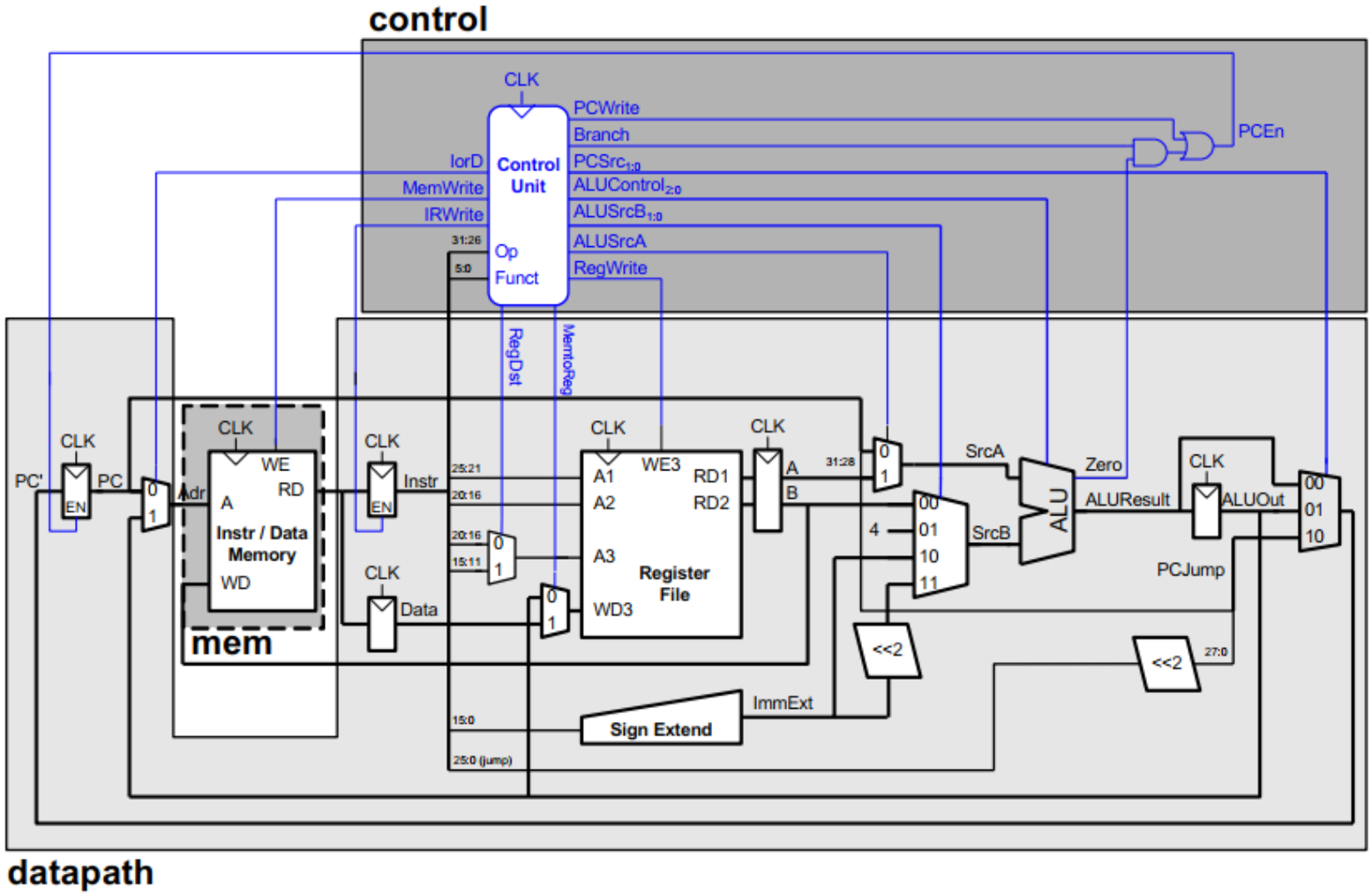
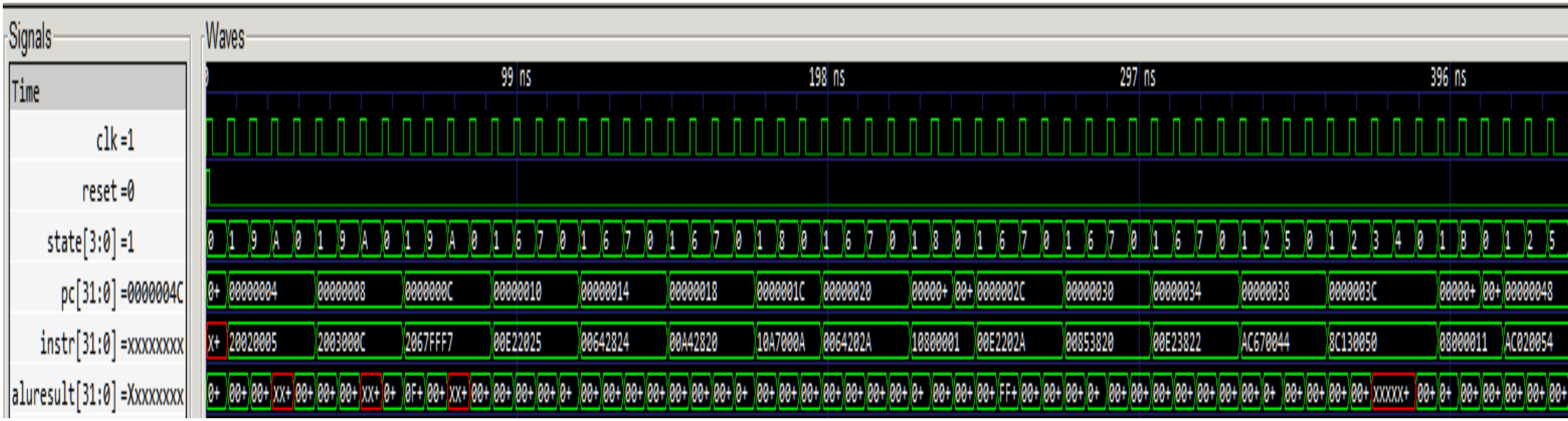


Figure 1. Multicycle Processor

3 - Simulación del programa:

Como podemos observar, la lectura de instrucciones, lógica de pc, salida de aluresult y la transición de estados es la correcta.



Para esta simulación, se realizó el siguiente testbench:

```
`timescale 1ns/1ns
module testbench;
    reg clk;
    reg reset;

    wire [31:0] writedata, dataadr;
    wire        memwrite;

    top dut(clk, reset, writedata, dataadr, memwrite);

    initial
        begin
            reset <= 1; #1; reset <= 0;
        end

    always
        begin
            clk <= 1; #2; clk <= 0; #5;
        end

    always @ (negedge clk)
    begin
        if (dataadr === 32'h4C & writedata===5) begin
            $display("Simulation suceeded");
            $stop;
        end
    end

    initial begin
        $dumpfile("mips.vcd");
        $dumpvars;
    end
endmodule
```

Con esto, damos por concluido el laboratorio 9 y, consecuentemente, el proyecto solicitado.

Bibliografía

- Harris, D. & Harris, S. (2012). Digital Design and Computer Architecture. 2nd Edition. San Francisco, Morgan Kaufmann.