Proyecto-ARCH

HELLO!

Lapa Romero, Julisa	100%
Calixto Rojas, Neftali	100%

Prof: Jorge Luis Gonzalez Reaño

Índice

Índice

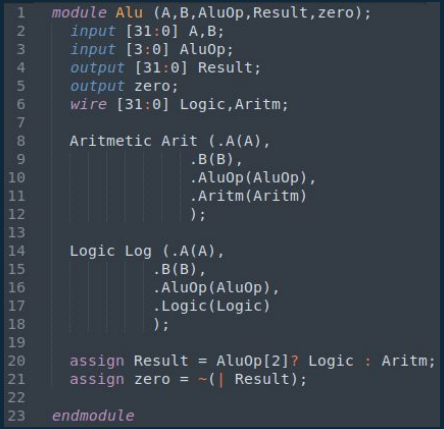
Índice

1

MIPS Single Cycle

Implementación en Verilog HDL

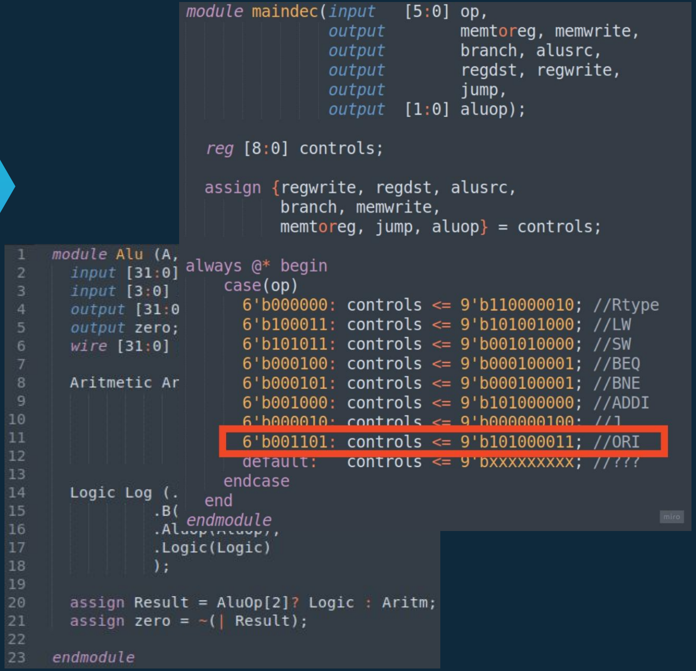
MIPS Single Cycle



```verilog
module Alu (A,B,AluOp,Result,zero);
  input [31:0] A,B;
  input [3:0] AluOp;
  output [31:0] Result;
  output zero;
  wire [31:0] Logic,Aritm;

  Aritmetic Arit (.A(A),
                  .B(B),
                  .AluOp(AluOp),
                  .Aritm(Aritm)
                  );

  Logic Log (.A(A),
             .B(B),
             .AluOp(AluOp),
             .Logic(Logic)
             );

  assign Result = AluOp[2]? Logic : Aritm;
  assign zero = ~(| Result);

endmodule
```

Implementación en Verilog HDL

MIPS Single Cycle

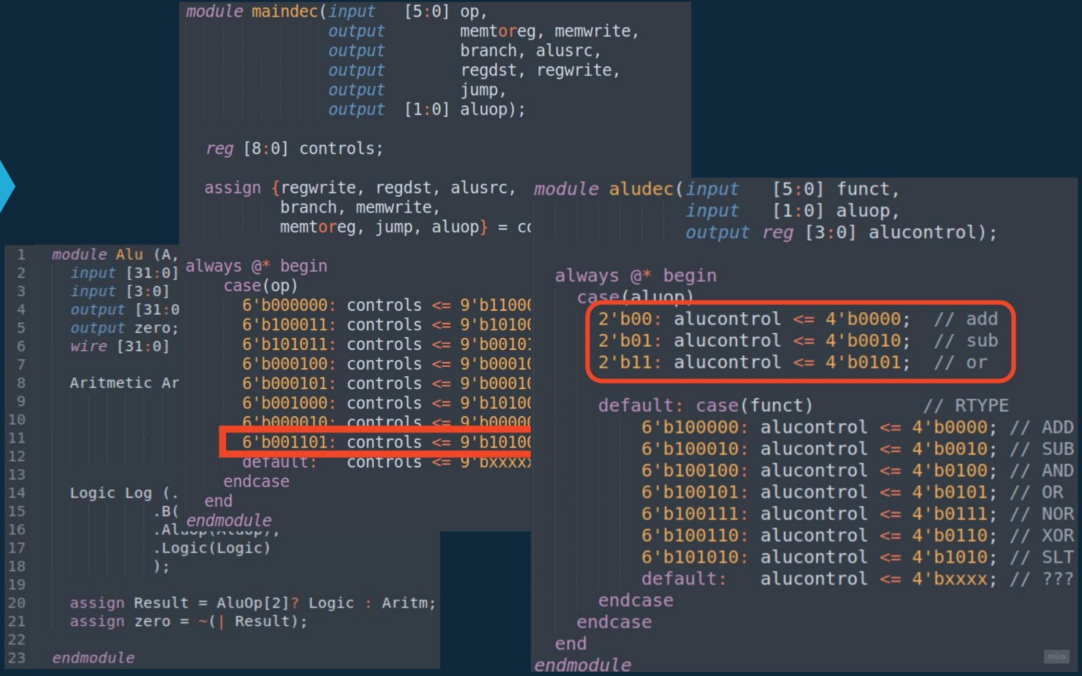


```
module maindec(input   [5:0] op,
               output        memtoreg, memwrite,
               output        branch, alusrc,
               output        regdst, regwrite,
               output        jump,
               output  [1:0] aluop);

  reg [8:0] controls;

  assign {regwrite, regdst, alusrc,
          branch, memwrite,
          memtoreg, jump, aluop} = controls;

always @* begin
    case(op)
      6'b000000: controls <= 9'b110000010; //Rtype
      6'b100011: controls <= 9'b101001000; //LW
      6'b101011: controls <= 9'b001010000; //SW
      6'b000100: controls <= 9'b000100001; //BEQ
      6'b000101: controls <= 9'b000100001; //BNE
      6'b001000: controls <= 9'b101000000; //ADDI
      6'b000010: controls <= 9'b000000100; //J
      6'b001101: controls <= 9'b101000011; //ORI
      default:   controls <= 9'bxxxxxxxxx; //???
    endcase
  end
endmodule
```

```
module Alu (A,
  input [31:0]
  input [3:0]
  output [31:0
  output zero;
  wire [31:0]

  Aritmetic Ar




  Logic Log (.
            .B(
            .AluOp(AluOp),
            .Logic(Logic)
            );

  assign Result = AluOp[2]? Logic : Aritm;
  assign zero = ~(| Result);

endmodule
```

Implementación en Verilog HDL

MIPS Single Cycle



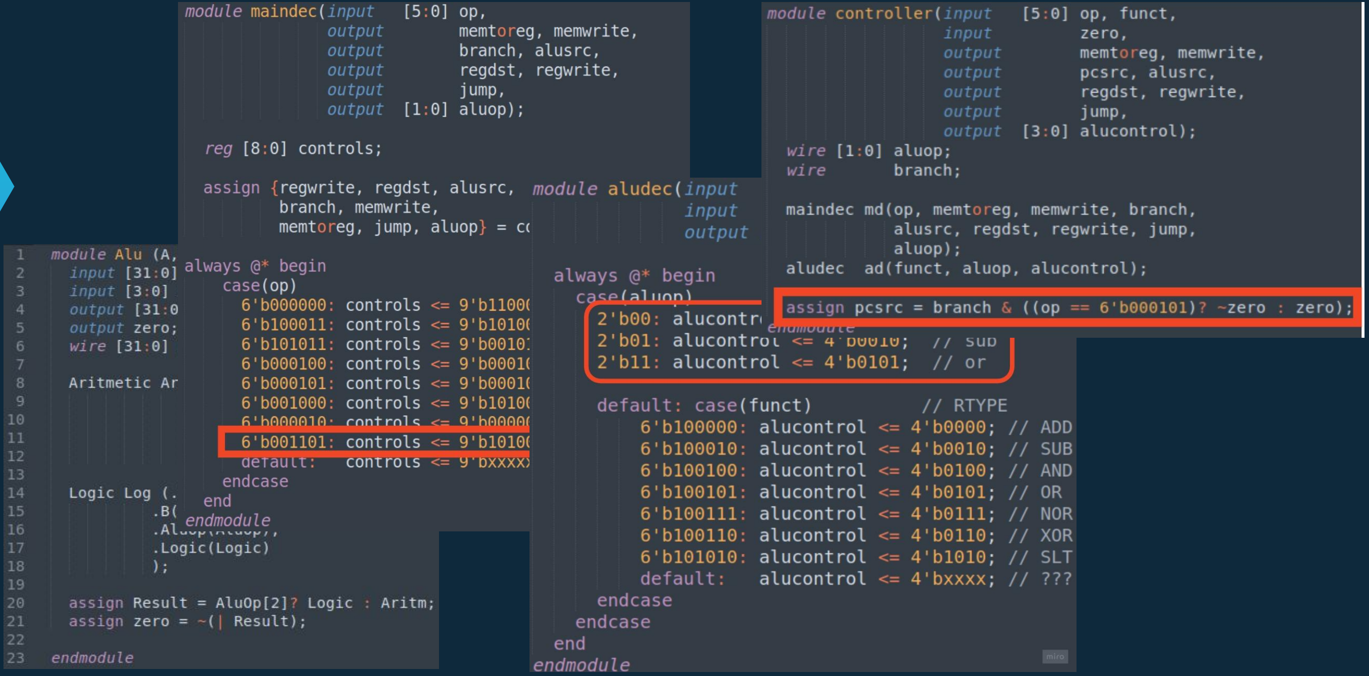
```
module maindec(input   [5:0] op,
               output        memtoreg, memwrite,
               output        branch, alusrc,
               output        regdst, regwrite,
               output        jump,
               output  [1:0] aluop);

  reg [8:0] controls;

  assign {regwrite, regdst, alusrc,
          branch, memwrite,
          memtoreg, jump, aluop} = cc

always @* begin
    case(op)
      6'b000000: controls <= 9'b11000
      6'b100011: controls <= 9'b10100
      6'b101011: controls <= 9'b00101
      6'b000100: controls <= 9'b00010
      6'b000101: controls <= 9'b00010
      6'b001000: controls <= 9'b10100
      6'b000010: controls <= 9'b00000
      6'b001101: controls <= 9'b10100
      default:   controls <= 9'bxxxxx
    endcase
  end
endmodule
```

```
module Alu (A,
  input [31:0]
  input [3:0]
  output [31:0
  output zero;
  wire [31:0]

  Aritmetic Ar



  Logic Log (.
             .B(
             .Al
             .Logic(Logic)
             );

  assign Result = AluOp[2]? Logic : Aritm;
  assign zero = ~(| Result);

endmodule
```

```
module aludec(input      [5:0] funct,
              input      [1:0] aluop,
              output reg [3:0] alucontrol);

  always @* begin
    case(aluop)
      2'b00: alucontrol <= 4'b0000;  // add
      2'b01: alucontrol <= 4'b0010;  // sub
      2'b11: alucontrol <= 4'b0101;  // or

      default: case(funct)          // RTYPE
          6'b100000: alucontrol <= 4'b0000; // ADD
          6'b100010: alucontrol <= 4'b0010; // SUB
          6'b100100: alucontrol <= 4'b0100; // AND
          6'b100101: alucontrol <= 4'b0101; // OR
          6'b100111: alucontrol <= 4'b0111; // NOR
          6'b100110: alucontrol <= 4'b0110; // XOR
          6'b101010: alucontrol <= 4'b1010; // SLT
          default:   alucontrol <= 4'bxxxx; // ???
        endcase
      endcase
    end
endmodule
```

Implementación en Verilog HDL

MIPS Single Cycle



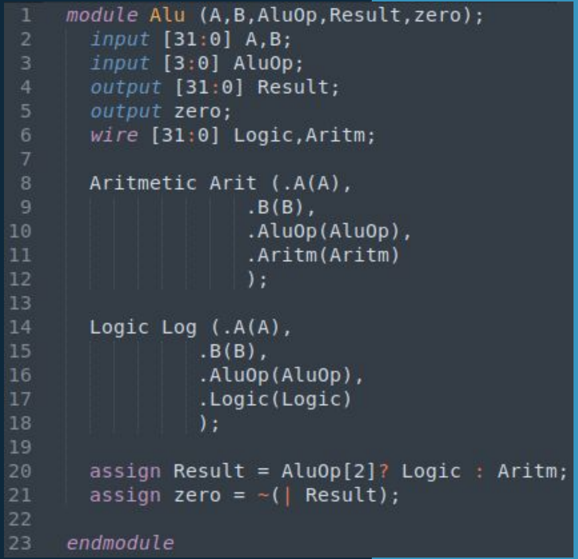
```verilog
module maindec(input       [5:0] op,
               output            memtoreg, memwrite,
               output            branch, alusrc,
               output            regdst, regwrite,
               output            jump,
               output      [1:0] aluop);

  reg [8:0] controls;

  assign {regwrite, regdst, alusrc,
          branch, memwrite,
          memtoreg, jump, aluop} = cc

always @* begin
    case(op)
      6'b000000: controls <= 9'b11000
      6'b100011: controls <= 9'b10100
      6'b101011: controls <= 9'b00101
      6'b000100: controls <= 9'b00010
      6'b000101: controls <= 9'b00010
      6'b001000: controls <= 9'b10100
      6'b000010: controls <= 9'b00000
      6'b001101: controls <= 9'b10100
      default:   controls <= 9'bxxxxx
    endcase
  end
endmodule
```

```verilog
module Alu (A,
  input [31:0]
  input [3:0]
  output [31:0
  output zero;
  wire [31:0]

  Aritmetic Ar

  Logic Log (.
            .B(
            .Al
            .Logic(Logic)
            );

  assign Result = AluOp[2]? Logic : Aritm;
  assign zero = ~(| Result);

endmodule
```

```verilog
module aludec(input
              input
              output

  always @* begin
    case(aluop)
      2'b00: alucontr
      2'b01: alucontrol <= 4'b0010;  // sub
      2'b11: alucontrol <= 4'b0101;  // or

      default: case(funct)          // RTYPE
          6'b100000: alucontrol <= 4'b0000; // ADD
          6'b100010: alucontrol <= 4'b0010; // SUB
          6'b100100: alucontrol <= 4'b0100; // AND
          6'b100101: alucontrol <= 4'b0101; // OR
          6'b100111: alucontrol <= 4'b0111; // NOR
          6'b100110: alucontrol <= 4'b0110; // XOR
          6'b101010: alucontrol <= 4'b1010; // SLT
          default:   alucontrol <= 4'bxxxx; // ???
      endcase
    endcase
  end
endmodule
```

```verilog
module controller(input   [5:0] op, funct,
                  input         zero,
                  output        memtoreg, memwrite,
                  output        pcsrc, alusrc,
                  output        regdst, regwrite,
                  output        jump,
                  output  [3:0] alucontrol);
  wire [1:0] aluop;
  wire       branch;

  maindec md(op, memtoreg, memwrite, branch,
             alusrc, regdst, regwrite, jump,
             aluop);
  aludec  ad(funct, aluop, alucontrol);

  assign pcsrc = branch & ((op == 6'b000101)? ~zero : zero);
endmodule
```

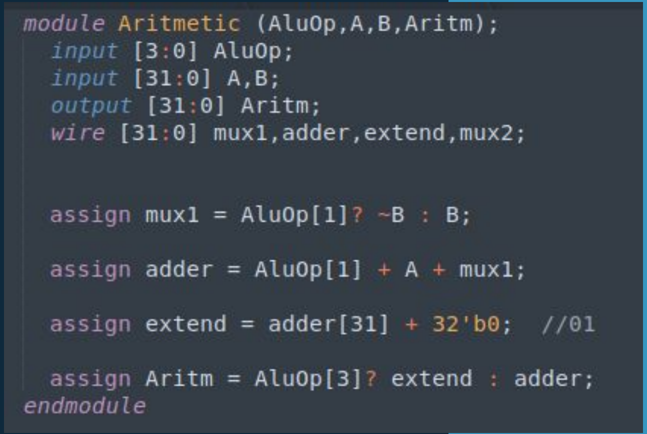
Alu



```
module Alu (A,B,AluOp,Result,zero);
  input [31:0] A,B;
  input [3:0] AluOp;
  output [31:0] Result;
  output zero;
  wire [31:0] Logic,Aritm;

  Aritmetic Arit (.A(A),
                  .B(B),
                  .AluOp(AluOp),
                  .Aritm(Aritm)
                  );

  Logic Log (.A(A),
             .B(B),
             .AluOp(AluOp),
             .Logic(Logic)
             );

  assign Result = AluOp[2]? Logic : Aritm;
  assign zero = ~(| Result);

endmodule
```

Al usar un Alu de 4 bits se adaptó el alucontrol

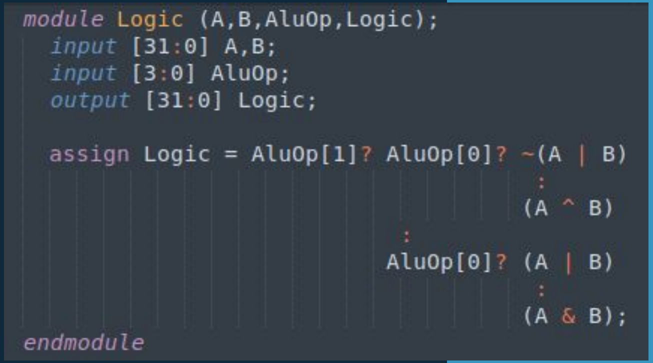
Alu



```
module Aritmetic (AluOp,A,B,Aritm);
  input [3:0] AluOp;
  input [31:0] A,B;
  output [31:0] Aritm;
  wire [31:0] mux1,adder,extend,mux2;


  assign mux1 = AluOp[1]? ~B : B;

  assign adder = AluOp[1] + A + mux1;

  assign extend = adder[31] + 32'b0;  //01

  assign Aritm = AluOp[3]? extend : adder;
endmodule
```

Módulo Aritmetic

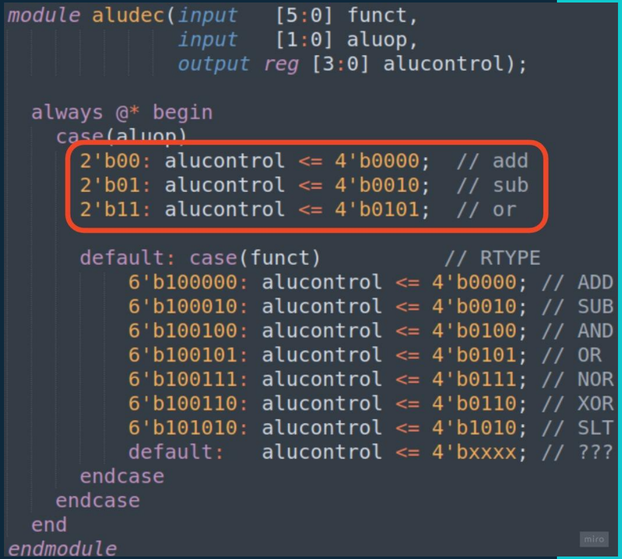
Alu



```
module Logic (A,B,AluOp,Logic);
  input [31:0] A,B;
  input [3:0] AluOp;
  output [31:0] Logic;

  assign Logic = AluOp[1]? AluOp[0]? ~(A | B)
                                    :
                                    (A ^ B)
                          :
                          AluOp[0]? (A | B)
                                    :
                                    (A & B);
endmodule
```

Módulo Logic

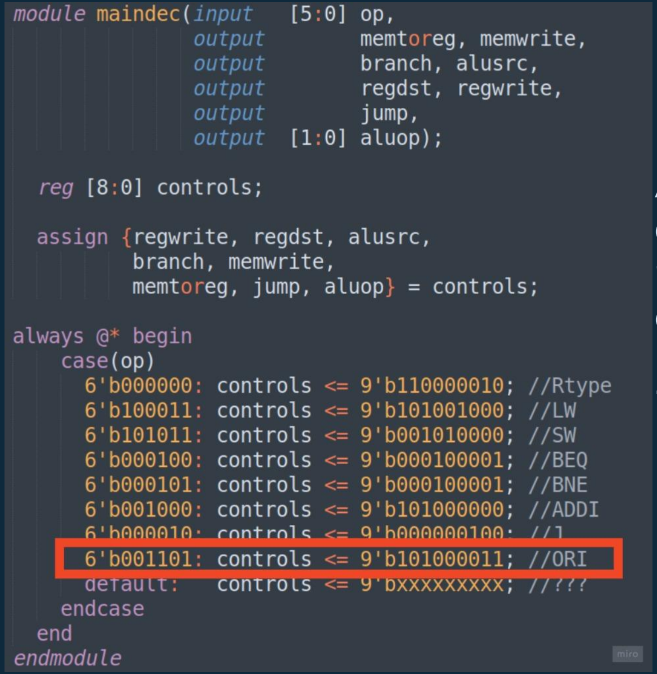
aludec



```
module aludec(input       [5:0] funct,
              input       [1:0] aluop,
              output reg [3:0] alucontrol);

  always @* begin
    case(aluop)
      2'b00: alucontrol <= 4'b0000;  // add
      2'b01: alucontrol <= 4'b0010;  // sub
      2'b11: alucontrol <= 4'b0101;  // or

      default: case(funct)          // RTYPE
          6'b100000: alucontrol <= 4'b0000; // ADD
          6'b100010: alucontrol <= 4'b0010; // SUB
          6'b100100: alucontrol <= 4'b0100; // AND
          6'b100101: alucontrol <= 4'b0101; // OR
          6'b100111: alucontrol <= 4'b0111; // NOR
          6'b100110: alucontrol <= 4'b0110; // XOR
          6'b101010: alucontrol <= 4'b1010; // SLT
          default:   alucontrol <= 4'bxxxx; // ???
      endcase
    endcase
  end
endmodule
```

En este módulo agregamos un case al aluop para hacer un or

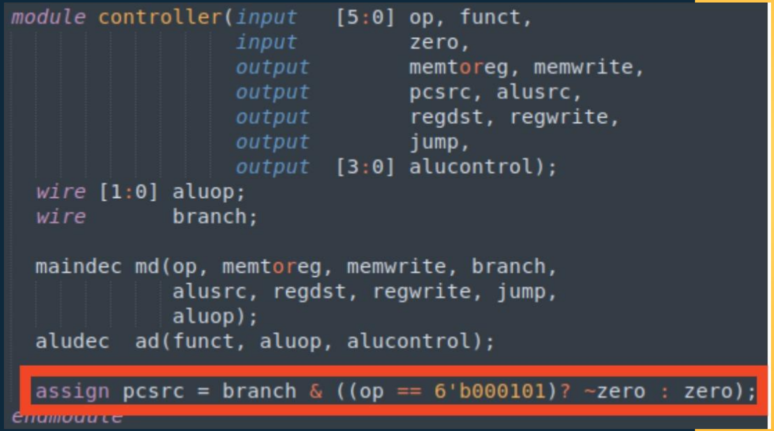
maindec



```
module maindec(input   [5:0] op,
               output        memtoreg, memwrite,
               output        branch, alusrc,
               output        regdst, regwrite,
               output        jump,
               output  [1:0] aluop);

  reg [8:0] controls;

  assign {regwrite, regdst, alusrc,
          branch, memwrite,
          memtoreg, jump, aluop} = controls;

always @* begin
    case(op)
      6'b000000: controls <= 9'b110000010; //Rtype
      6'b100011: controls <= 9'b101001000; //LW
      6'b101011: controls <= 9'b001010000; //SW
      6'b000100: controls <= 9'b000100001; //BEQ
      6'b000101: controls <= 9'b000100001; //BNE
      6'b001000: controls <= 9'b101000000; //ADDI
      6'b000010: controls <= 9'b000000100; //J
      6'b001101: controls <= 9'b101000011; //ORI
      default:   controls <= 9'bxxxxxxxxx; //???
    endcase
  end
endmodule
```

Al analizar el diagrama podemos saber el controls de las nuevas instrucciones .

controller



```
module controller(input   [5:0] op, funct,
                  input         zero,
                  output        memtoreg, memwrite,
                  output        pcsrc, alusrc,
                  output        regdst, regwrite,
                  output        jump,
                  output  [3:0] alucontrol);
  wire [1:0] aluop;
  wire       branch;

  maindec md(op, memtoreg, memwrite, branch,
             alusrc, regdst, regwrite, jump,
             aluop);
  aludec  ad(funct, aluop, alucontrol);

  assign pcsrc = branch & ((op == 6'b000101)? ~zero : zero);
endmodule
```

En este módulo modificamos el pcsrc para implementar el branch

BNE & ORI

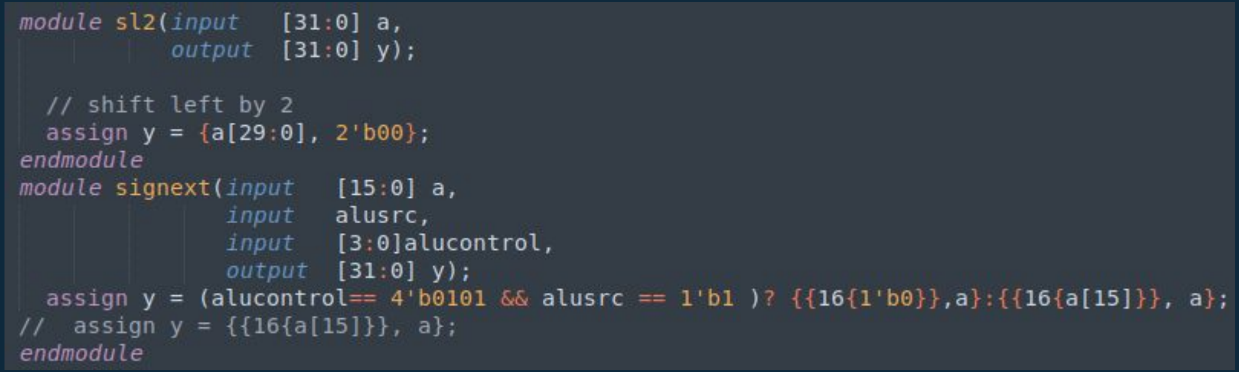
Ori :

Ori :

```
rt = rs | immediate
```

Ori :

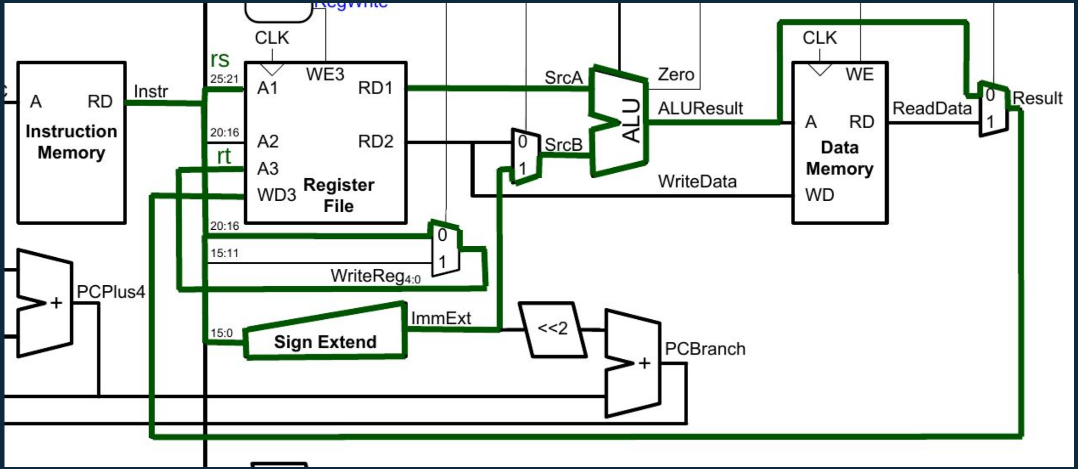
```
rt = rs | immediate
```

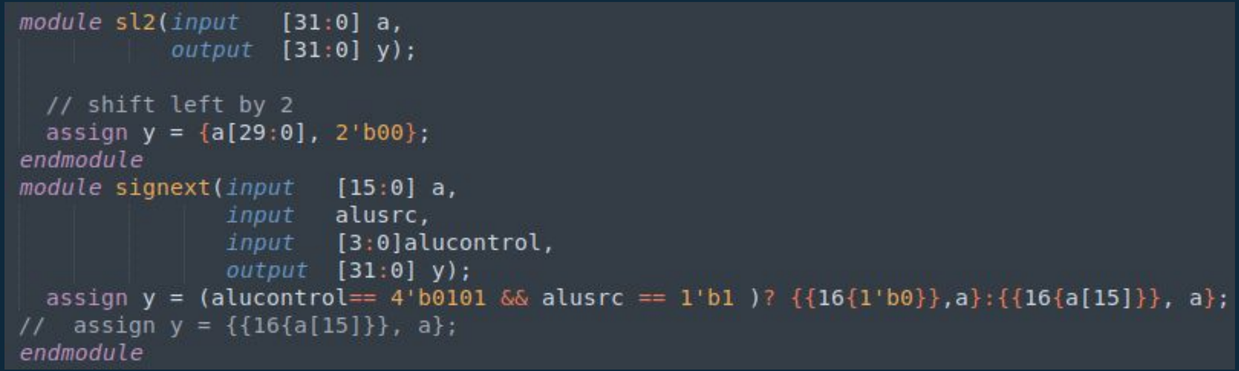


```
module sl2(input   [31:0] a,
           output  [31:0] y);

  // shift left by 2
  assign y = {a[29:0], 2'b00};
endmodule
module signext(input   [15:0] a,
               input   alusrc,
               input   [3:0]alucontrol,
               output  [31:0] y);
  assign y = (alucontrol== 4'b0101 && alusrc == 1'b1 )? {{16{1'b0}},a}:{{16{a[15]}}, a};
//  assign y = {{16{a[15]}}, a};
endmodule
```

Ori:

```
rt = rs | immediate
```





```
module sl2(input   [31:0] a,
           output  [31:0] y);

  // shift left by 2
  assign y = {a[29:0], 2'b00};
endmodule
module signext(input   [15:0] a,
               input   alusrc,
               input   [3:0]alucontrol,
               output  [31:0] y);
  assign y = (alucontrol== 4'b0101 && alusrc == 1'b1 )? {{16{1'b0}},a}:{{16{a[15]}}, a};
//  assign y = {{16{a[15]}}, a};
endmodule
```

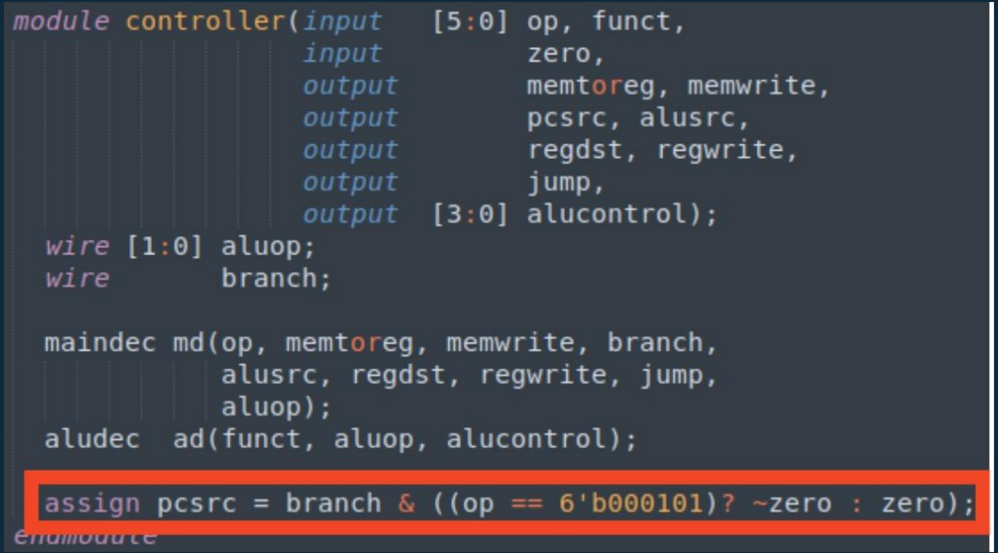
Bne :

Bne :

```
If rs = rt -> j label
If rs ≠ rt -> j PC+4
```

Bne :

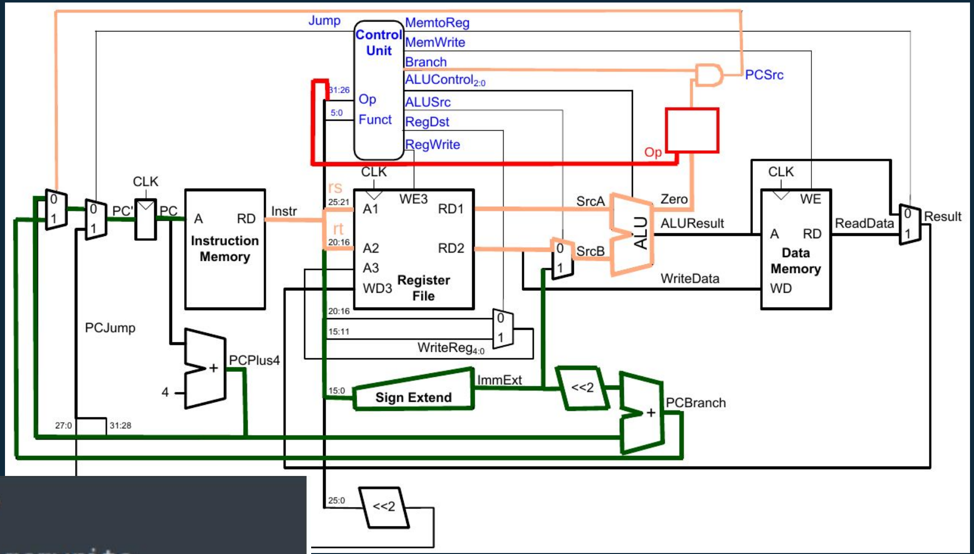
If rs = rt -> j label
If rs ≠ rt -> j PC+4

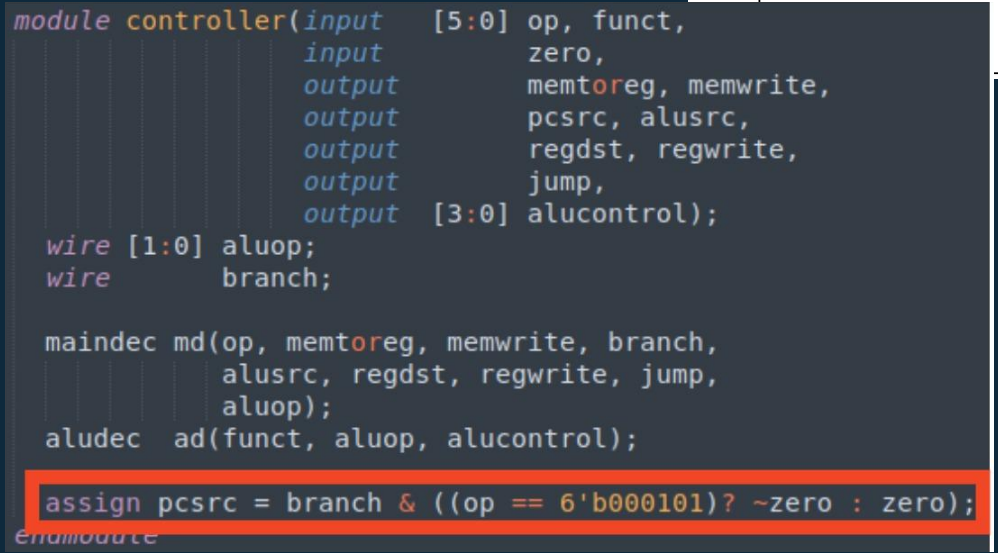


```
module controller(input   [5:0] op, funct,
                  input         zero,
                  output        memtoreg, memwrite,
                  output        pcsrc, alusrc,
                  output        regdst, regwrite,
                  output        jump,
                  output  [3:0] alucontrol);
  wire [1:0] aluop;
  wire       branch;

  maindec md(op, memtoreg, memwrite, branch,
             alusrc, regdst, regwrite, jump,
             aluop);
  aludec  ad(funct, aluop, alucontrol);

  assign pcsrc = branch & ((op == 6'b000101)? ~zero : zero);
endmodule
```

Bne :

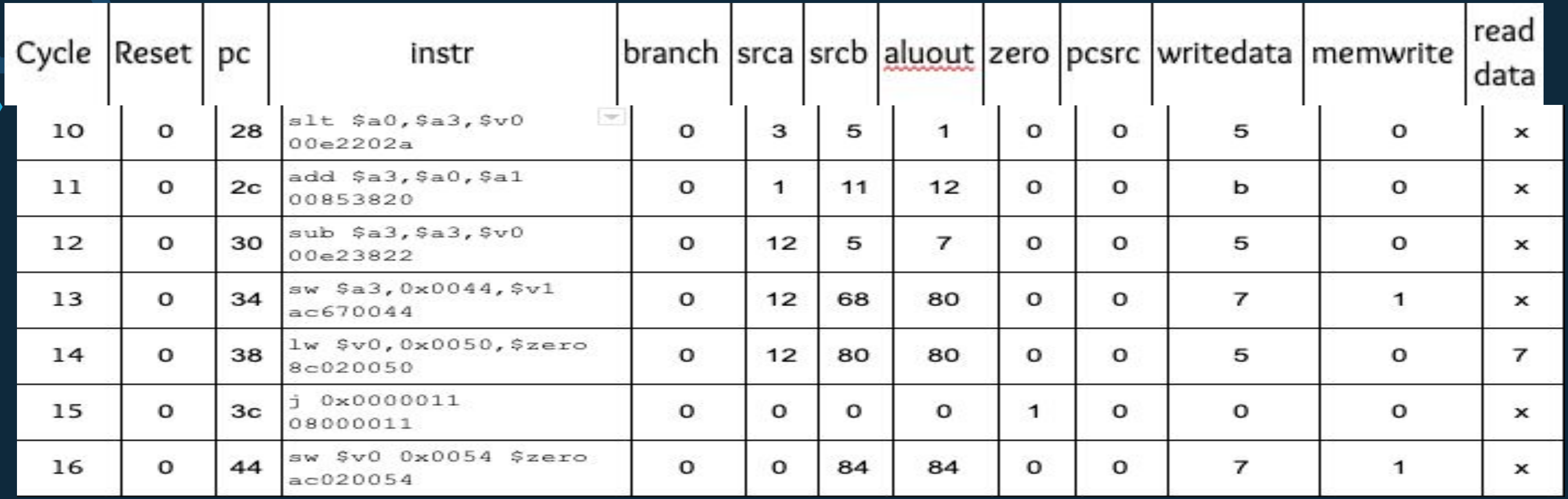
```
If rs = rt -> j label
If rs ≠ rt -> j PC+4
```



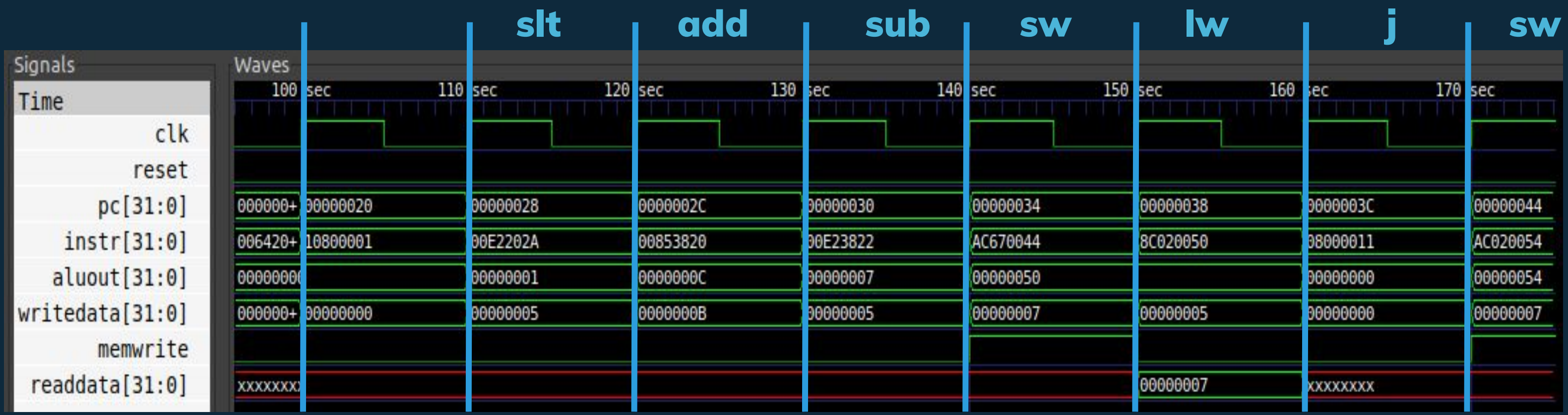


```
module controller(input   [5:0] op, funct,
                  input         zero,
                  output        memtoreg, memwrite,
                  output        pcsrc, alusrc,
                  output        regdst, regwrite,
                  output        jump,
                  output  [3:0] alucontrol);
  wire [1:0] aluop;
  wire       branch;

  maindec md(op, memtoreg, memwrite, branch,
             alusrc, regdst, regwrite, jump,
             aluop);
  aludec  ad(funct, aluop, alucontrol);

  assign pcsrc = branch & ((op == 6'b000101)? ~zero : zero);
endmodule
```

Waveforms memfile.dat

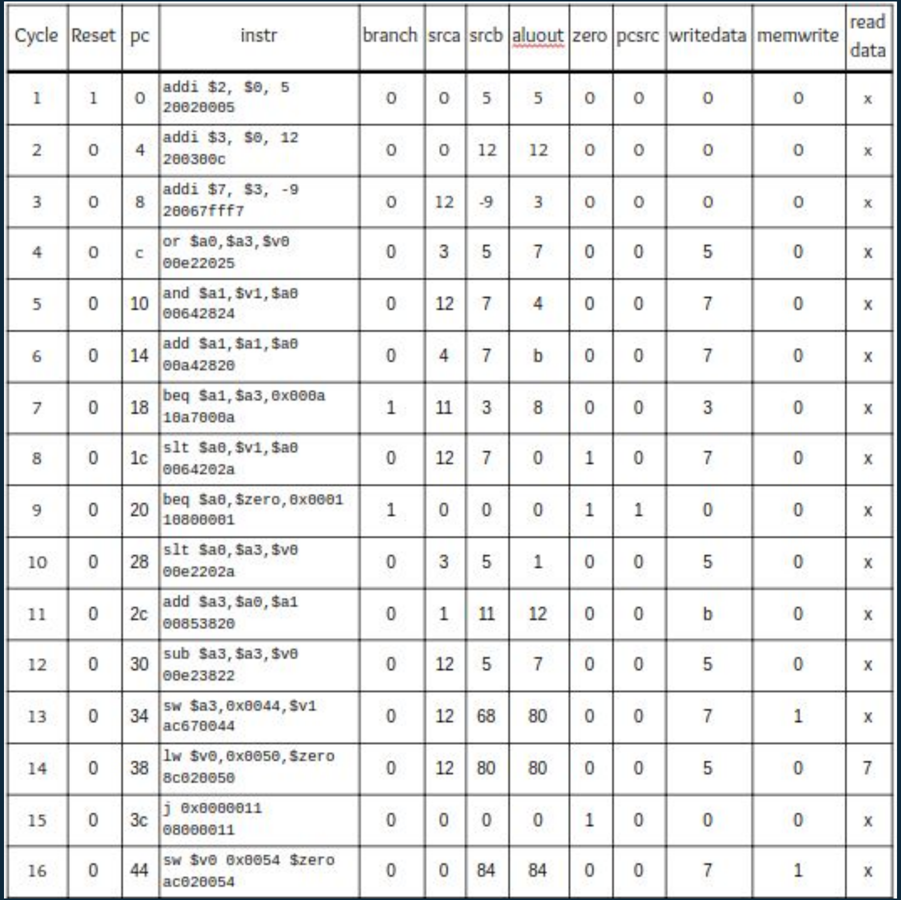


Cycle	Reset	pc	instr	branch	srca	srcb	aluout	zero	pcsrc	writedata	memwrite	read data
10	0	28	slt $a0,$a3,$v0 00e2202a	0	3	5	1	0	0	5	0	x
11	0	2c	add $a3,$a0,$a1 00853820	0	1	11	12	0	0	b	0	x
12	0	30	sub $a3,$a3,$v0 00e23822	0	12	5	7	0	0	5	0	x
13	0	34	sw $a3,0x0044,$v1 ac670044	0	12	68	80	0	0	7	1	x
14	0	38	lw $v0,0x0050,$zero 8c020050	0	12	80	80	0	0	5	0	7
15	0	3c	j 0x0000011 08000011	0	0	0	0	1	0	0	0	x
16	0	44	sw $v0 0x0054 $zero ac020054	0	0	84	84	0	0	7	1	x



Tablas:

Tabla 1: Primeros 16 ciclos del mipstest.asm

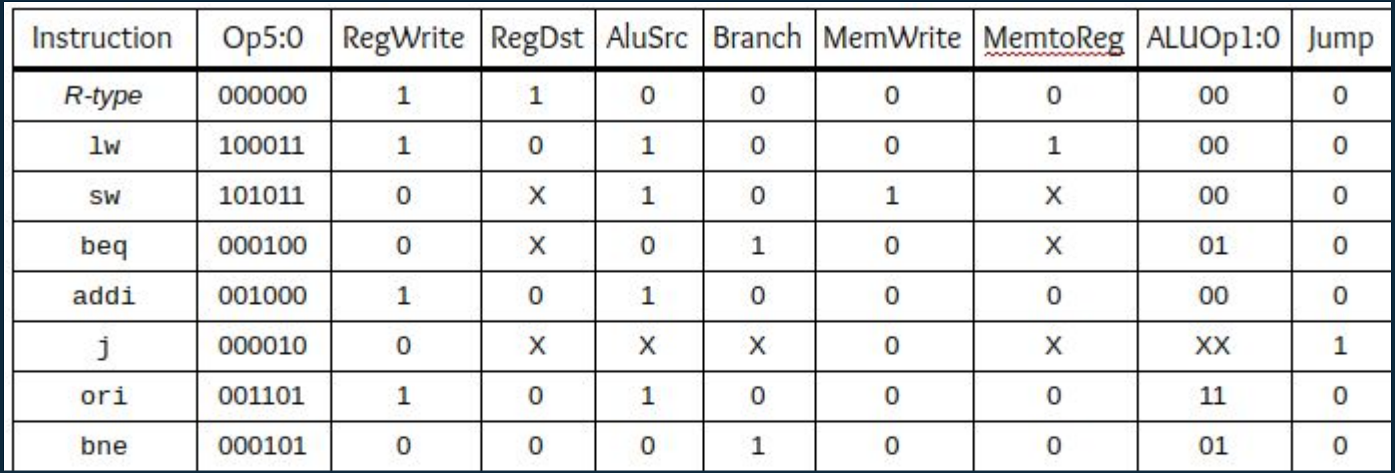


Cycle	Reset	pc	instr	branch	srca	srcb	aluout	zero	pcsrc	writedata	memwrite	read data
1	1	0	addi $2, $0, 5 20020005	0	0	5	5	0	0	0	0	x
2	0	4	addi $3, $0, 12 2003000c	0	0	12	12	0	0	0	0	x
3	0	8	addi $7, $3, -9 20067fff7	0	12	-9	3	0	0	0	0	x
4	0	c	or $a0,$a3,$v0 00e22025	0	3	5	7	0	0	5	0	x
5	0	10	and $a1,$v1,$a0 00642824	0	12	7	4	0	0	7	0	x
6	0	14	add $a1,$a1,$a0 00a42820	0	4	7	b	0	0	7	0	x
7	0	18	beq $a1,$a3,0x000a 10a7000a	1	11	3	8	0	0	3	0	x
8	0	1c	slt $a0,$v1,$a0 0064202a	0	12	7	0	1	0	7	0	x
9	0	20	beq $a0,$zero,0x0001 10800001	1	0	0	0	1	1	0	0	x
10	0	28	slt $a0,$a3,$v0 00e2202a	0	3	5	1	0	0	5	0	x
11	0	2c	add $a3,$a0,$a1 00853820	0	1	11	12	0	0	b	0	x
12	0	30	sub $a3,$a3,$v0 00e23822	0	12	5	7	0	0	5	0	x
13	0	34	sw $a3,0x0044,$v1 ac670044	0	12	68	80	0	0	7	1	x
14	0	38	lw $v0,0x0050,$zero 8c020050	0	12	80	80	0	0	5	0	7
15	0	3c	j 0x0000011 08000011	0	0	0	0	1	0	0	0	x
16	0	44	sw $v0 0x0054 $zero ac020054	0	0	84	84	0	0	7	1	x

¿Qué dirección escribirá la instrucción final sw y qué valor será?

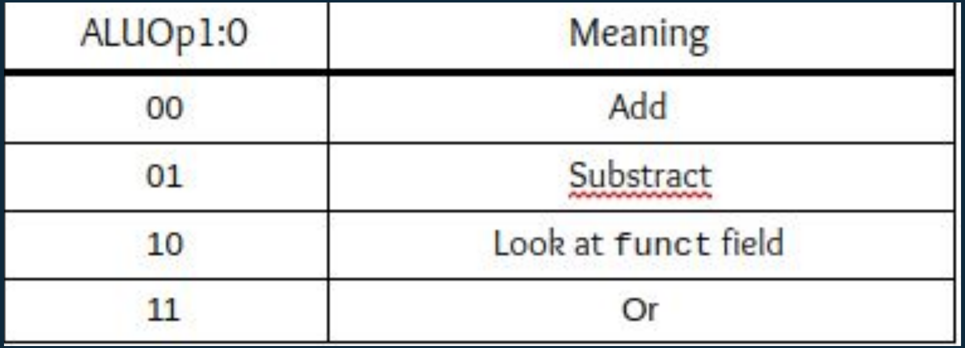
La dirección de la instrucción final se obtiene del aluout, esta es 84 en decimal. El valor está mostrado por el writedata, este es 7.

Tabla 2: Extended Functionality. Main Decoder



Instruction	Op5:0	RegWrite	RegDst	AluSrc	Branch	MemWrite	MemtoReg	ALUOp1:0	Jump
R-type	000000	1	1	0	0	0	0	00	0
lw	100011	1	0	1	0	0	1	00	0
sw	101011	0	X	1	0	1	X	00	0
beq	000100	0	X	0	1	0	X	01	0
addi	001000	1	0	1	0	0	0	00	0
j	000010	0	X	X	X	0	X	XX	1
ori	001101	1	0	1	0	0	0	11	0
bne	000101	0	0	0	1	0	0	01	0

Tabla 2: Extended Functionality. Alu Decoder

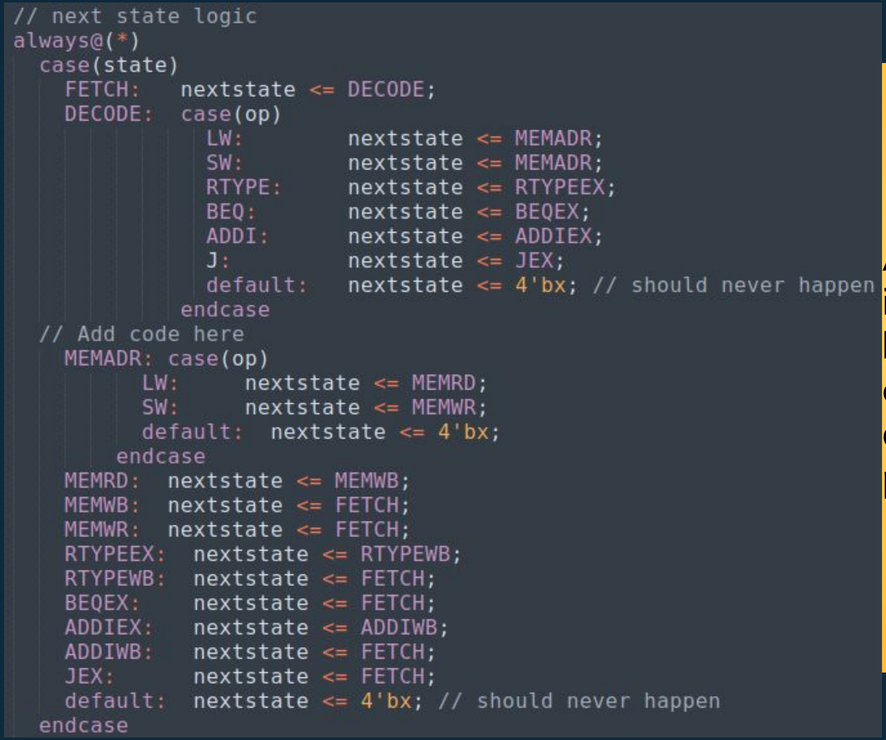


ALUOp1:0	Meaning
00	Add
01	Substract
10	Look at funct field
11	Or

2

MIPS Multi Cycle

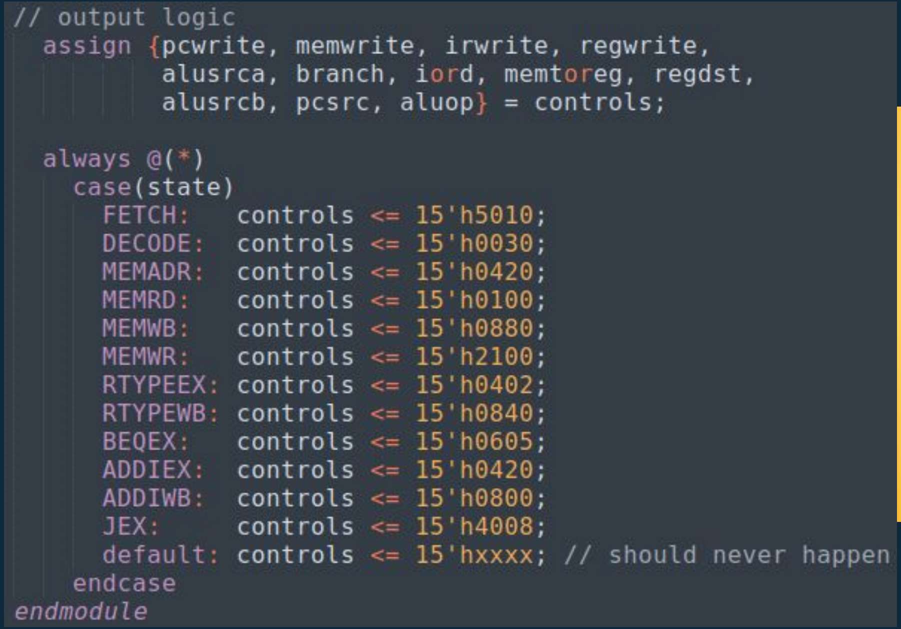
controller



```
// 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
  // Add code here
    MEMADR: case(op)
            LW:     nextstate <= MEMRD;
            SW:     nextstate <= MEMWR;
            default:  nextstate <= 4'bx;
          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
```

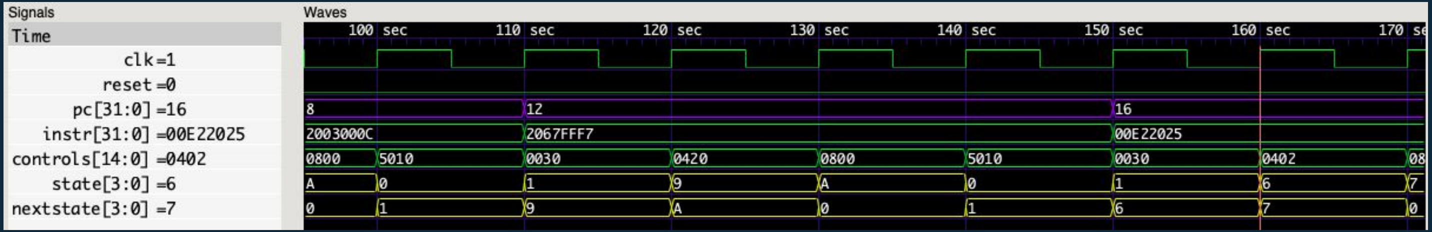
Aquí implementamos las señales de control usando el diagrama provisto

controller



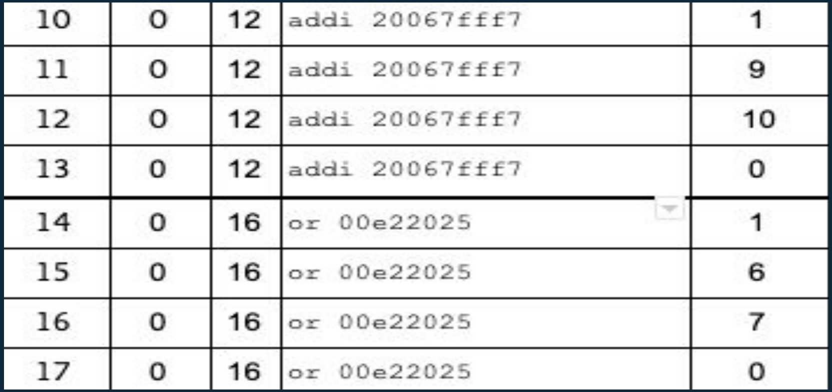
```
// output logic
  assign {pcwrite, memwrite, irwrite, regwrite,
          alusrca, branch, iord, memtoreg, regdst,
          alusrcb, pcsrc, aluop} = controls;

  always @(*)
    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'h0800;
      JEX:      controls <= 15'h4008;
      default:  controls <= 15'hxxxx; // should never happen
    endcase
endmodule
```

Se le asigna las señales al controls



```
Time
clk=1
reset=0
pc[31:0]=16
instr[31:0]=00E22025
controls[14:0]=0402
state[3:0]=6
nextstate[3:0]=7
```

```
pc:        8 | 12 | 16
instr:     2003000C | 2067FFF7 | 00E22025
controls:  0800 | 5010 | 0030 | 0420 | 0800 | 5010 | 0030 | 0402 | 08
state:     A | 0 | 1 | 9 | A | 0 | 1 | 6 | 7
nextstate: 0 | 1 | 9 | A | 0 | 1 | 6 | 7 | 0
```

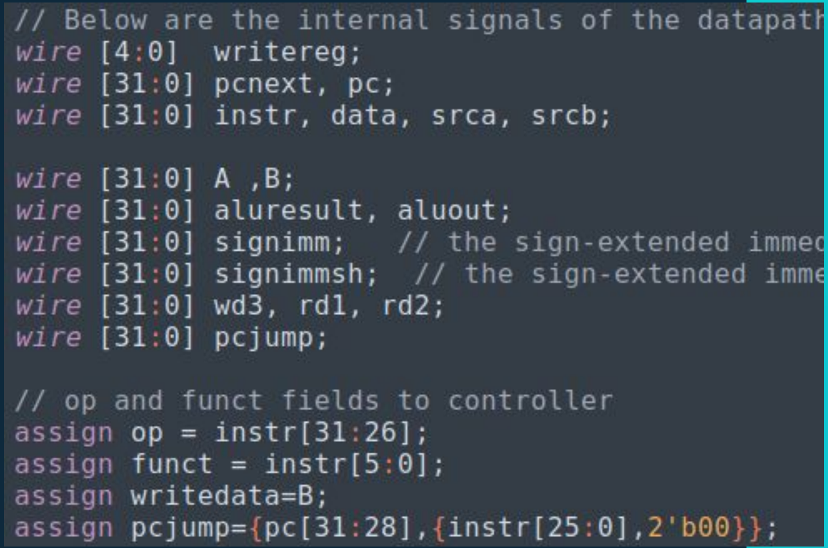


10	0	12	addi 20067fff7	1
11	0	12	addi 20067fff7	9
12	0	12	addi 20067fff7	10
13	0	12	addi 20067fff7	0
14	0	16	or 00e22025	1
15	0	16	or 00e22025	6
16	0	16	or 00e22025	7
17	0	16	or 00e22025	0

3

MIPS Multi Cycle Datapath

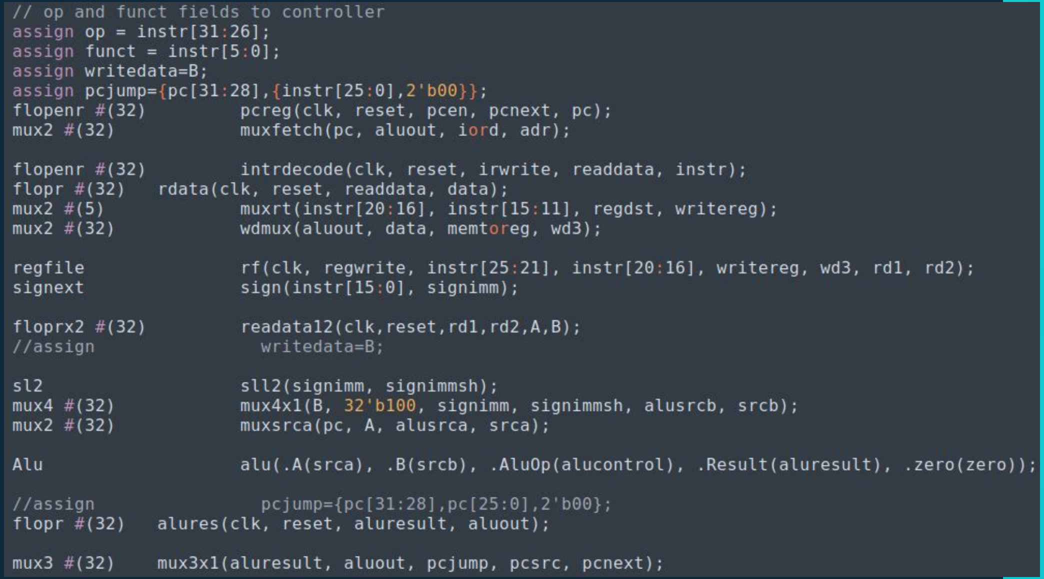
datapath



```
// Below are the internal signals of the datapath
wire [4:0]  writereg;
wire [31:0] pcnext, pc;
wire [31:0] instr, data, srca, srcb;

wire [31:0] A ,B;
wire [31:0] aluresult, aluout;
wire [31:0] signimm;   // the sign-extended immed
wire [31:0] signimmsh;  // the sign-extended imme
wire [31:0] wd3, rd1, rd2;
wire [31:0] pcjump;

// op and funct fields to controller
assign op = instr[31:26];
assign funct = instr[5:0];
assign writedata=B;
assign pcjump={pc[31:28],{instr[25:0],2'b00}};
```

Se creo el
wire pcjump
Para ser
fieles al
diagrama

datapath

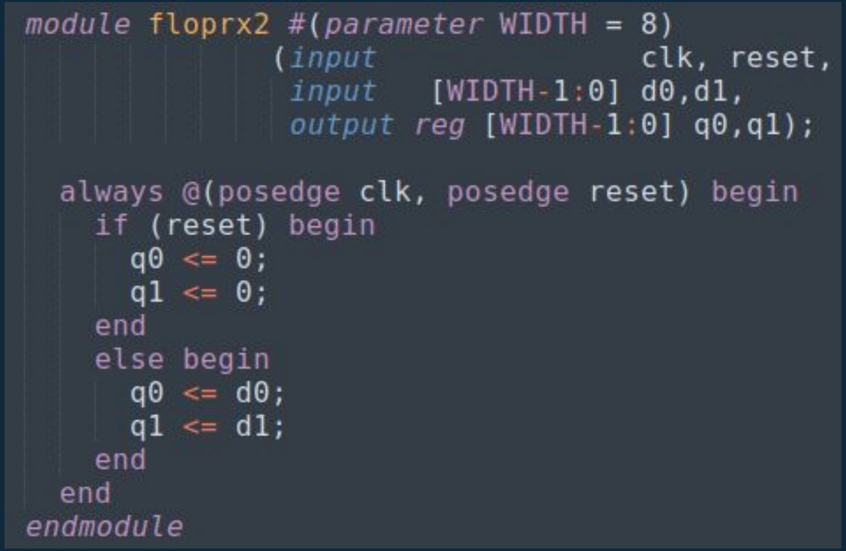


```
// op and funct fields to controller
assign op = instr[31:26];
assign funct = instr[5:0];
assign writedata=B;
assign pcjump={pc[31:28],{instr[25:0],2'b00}};
flopenr #(32)         pcreg(clk, reset, pcen, pcnext, pc);
mux2 #(32)            muxfetch(pc, aluout, iord, adr);

flopenr #(32)         intrdecode(clk, reset, irwrite, readdata, instr);
flopr #(32)   rdata(clk, reset, readdata, data);
mux2 #(5)             muxrt(instr[20:16], instr[15:11], regdst, writereg);
mux2 #(32)            wdmux(aluout, data, memtoreg, wd3);

regfile               rf(clk, regwrite, instr[25:21], instr[20:16], writereg, wd3, rd1, rd2);
signext               sign(instr[15:0], signimm);

floprx2 #(32)         readata12(clk,reset,rd1,rd2,A,B);
//assign                writedata=B;

sl2                   sll2(signimm, signimmsh);
mux4 #(32)            mux4x1(B, 32'b100, signimm, signimmsh, alusrcb, srcb);
mux2 #(32)            muxsrca(pc, A, alusrca, srca);

Alu                   alu(.A(srca), .B(srcb), .AluOp(alucontrol), .Result(aluresult), .zero(zero));

//assign                pcjump={pc[31:28],pc[25:0],2'b00};
flopr #(32)   alures(clk, reset, aluresult, aluout);

mux3 #(32)    mux3x1(aluresult, aluout, pcjump, pcsrc, pcnext);
```

Se puso en orden las unidades que sirven de control.

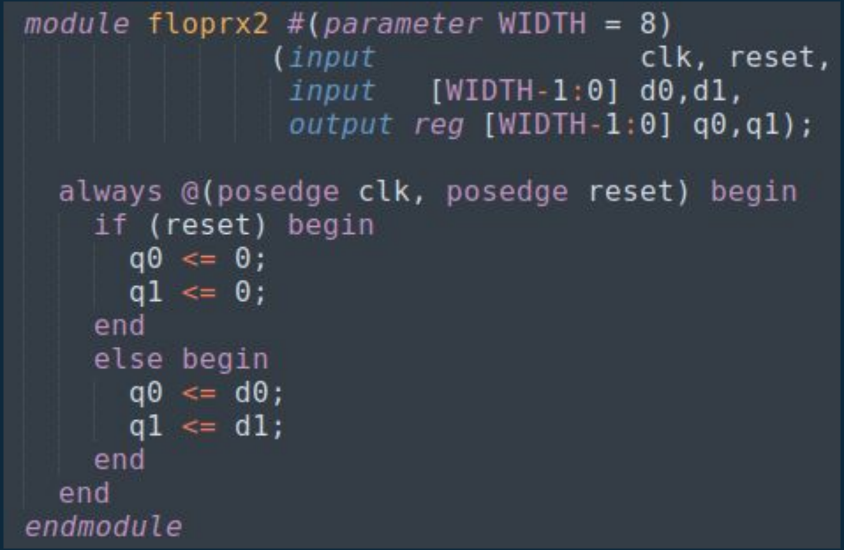
parts

parts

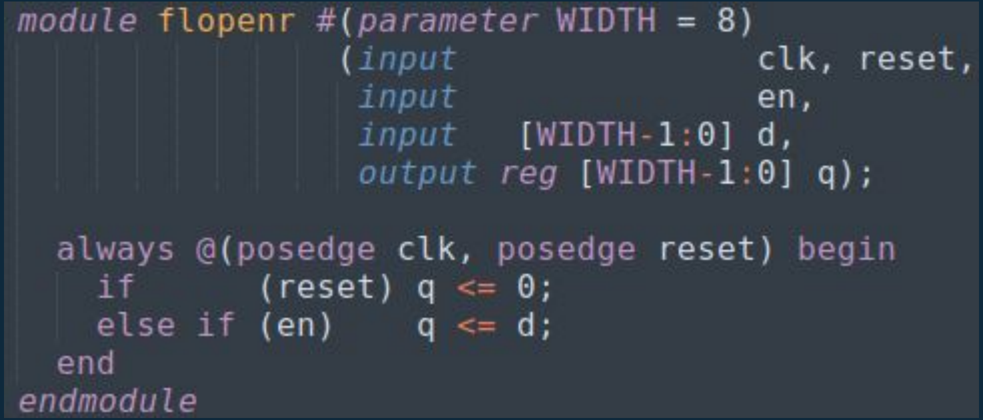


```
module floprx2 #(parameter WIDTH = 8)
               (input                    clk, reset,
                input    [WIDTH-1:0] d0,d1,
                output reg [WIDTH-1:0] q0,q1);

  always @(posedge clk, posedge reset) begin
    if (reset) begin
      q0 <= 0;
      q1 <= 0;
    end
    else begin
      q0 <= d0;
      q1 <= d1;
    end
  end
endmodule
```

parts

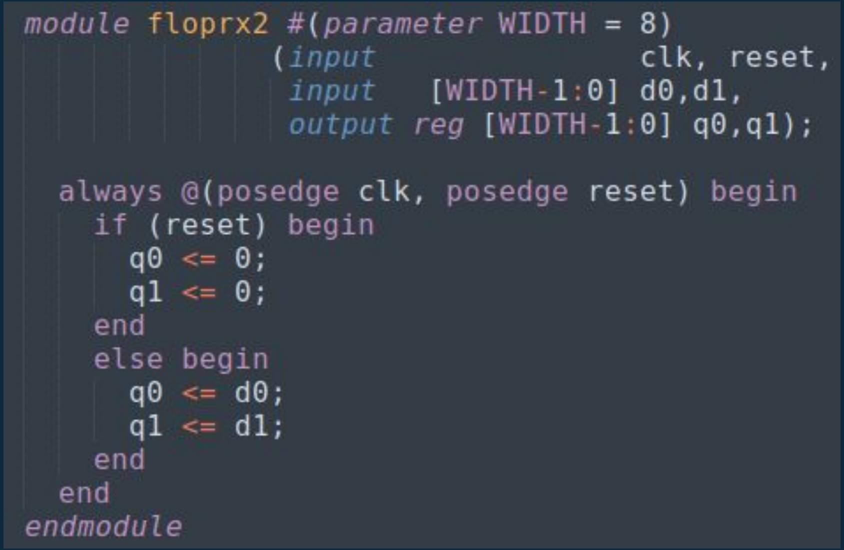


```
module floprx2 #(parameter WIDTH = 8)
               (input                    clk, reset,
                input      [WIDTH-1:0] d0,d1,
                output reg [WIDTH-1:0] q0,q1);

  always @(posedge clk, posedge reset) begin
    if (reset) begin
      q0 <= 0;
      q1 <= 0;
    end
    else begin
      q0 <= d0;
      q1 <= d1;
    end
  end
endmodule
```

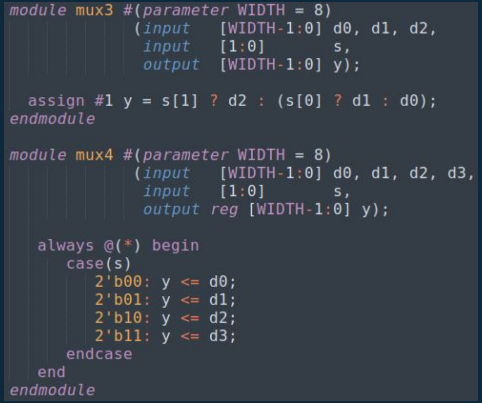


```
module flopenr #(parameter WIDTH = 8)
               (input                    clk, reset,
                input                    en,
                input      [WIDTH-1:0] d,
                output reg [WIDTH-1:0] q);

  always @(posedge clk, posedge reset) begin
    if       (reset) q <= 0;
    else if (en)     q <= d;
  end
endmodule
```

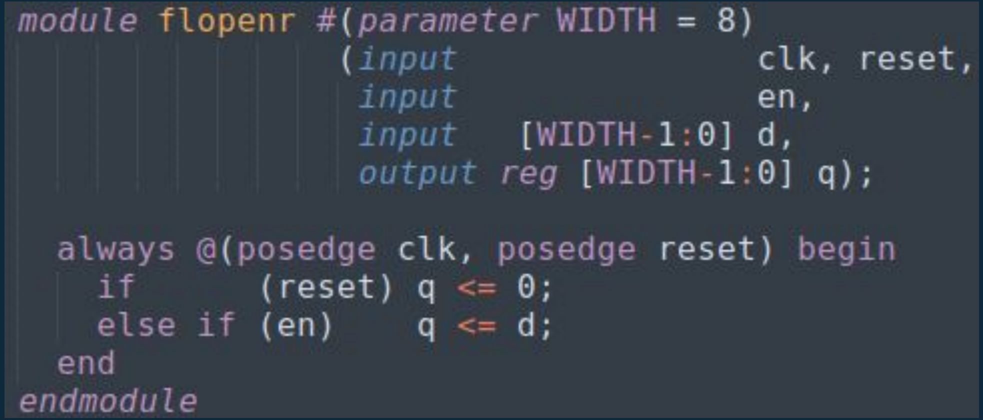
parts



```verilog
module floprx2 #(parameter WIDTH = 8)
               (input                    clk, reset,
                input     [WIDTH-1:0] d0,d1,
                output reg [WIDTH-1:0] q0,q1);

  always @(posedge clk, posedge reset) begin
    if (reset) begin
      q0 <= 0;
      q1 <= 0;
    end
    else begin
      q0 <= d0;
      q1 <= d1;
    end
  end
endmodule
```

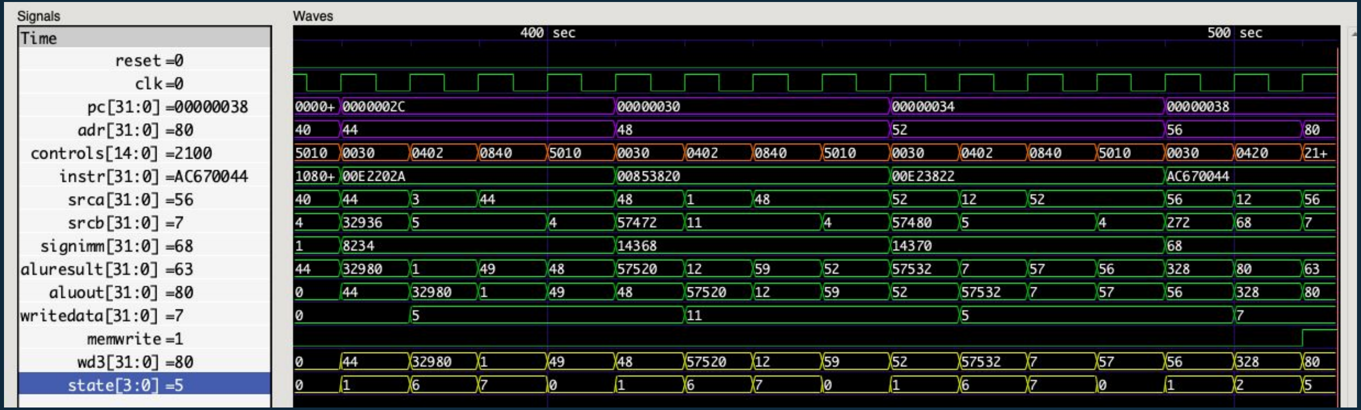


```verilog
module mux3 #(parameter WIDTH = 8)
            (input   [WIDTH-1:0] d0, d1, d2,
             input   [1:0]       s,
             output  [WIDTH-1:0] y);

  assign #1 y = s[1] ? d2 : (s[0] ? d1 : d0);
endmodule

module mux4 #(parameter WIDTH = 8)
            (input   [WIDTH-1:0] d0, d1, d2, d3,
             input   [1:0]       s,
             output reg [WIDTH-1:0] y);

   always @(*) begin
      case(s)
         2'b00: y <= d0;
         2'b01: y <= d1;
         2'b10: y <= d2;
         2'b11: y <= d3;
      endcase
   end
endmodule
```



```verilog
module flopenr #(parameter WIDTH = 8)
               (input                    clk, reset,
                input                    en,
                input      [WIDTH-1:0] d,
                output reg [WIDTH-1:0] q);

  always @(posedge clk, posedge reset) begin
    if       (reset) q <= 0;
    else if (en)     q <= d;
  end
endmodule
```

Waveform multicycle



```
Signals                        Waves
Time                           400 sec                                         500 sec
reset =0
clk =0
pc[31:0] =00000038             0000+ 0000002C 00000030 00000034 00000038
adr[31:0] =80                  40 44 48 52 56 80
controls[14:0] =2100           5010 0030 0402 0840 5010 0030 0402 0840 5010 0030 0402 0840 5010 0030 0420 21+
instr[31:0] =AC670044          1080+ 00E2202A 00853820 00E23822 AC670044
srca[31:0] =56                 40 44 3 44 48 1 48 52 12 52 56 12 56
srcb[31:0] =7                  4 32936 5 4 57472 11 4 57480 5 4 272 68 7
signimm[31:0] =68              1 8234 14368 14370 68
aluresult[31:0] =63            44 32980 1 49 48 57520 12 59 52 57532 7 57 56 328 80 63
aluout[31:0] =80               0 44 32980 1 49 48 57520 12 59 52 57532 7 57 56 328 80
writedata[31:0] =7             0 5 11 5 7
memwrite =1
wd3[31:0] =80                  0 44 32980 1 49 48 57520 12 59 52 57532 7 57 56 328 80
state[3:0] =5                  0 1 6 7 0 1 6 7 0 1 6 7 0 1 2 5
```

“Podría parecer que hemos llegado al límite de lo que es posible lograr con la tecnología informática, aunque hay que tener cuidado con tales declaraciones, ya que tienden a sonar bastante tontas en cinco años.”

– John von Neumann

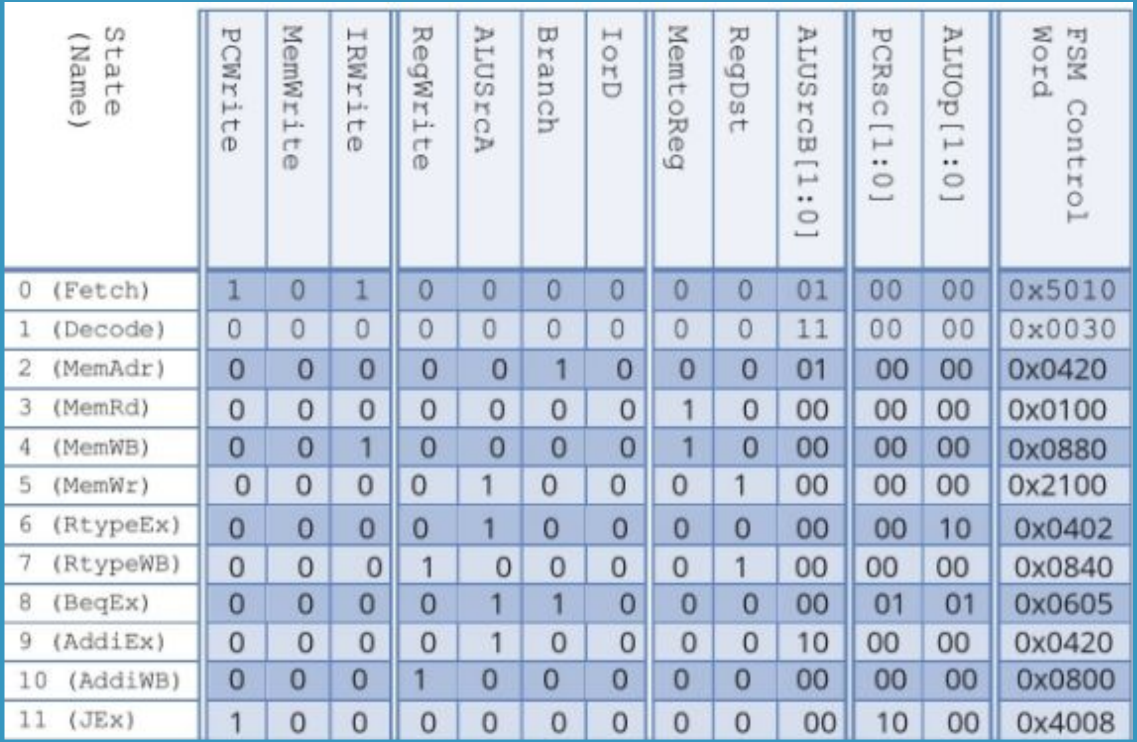
THANKS!

Any questions?

You can find us at

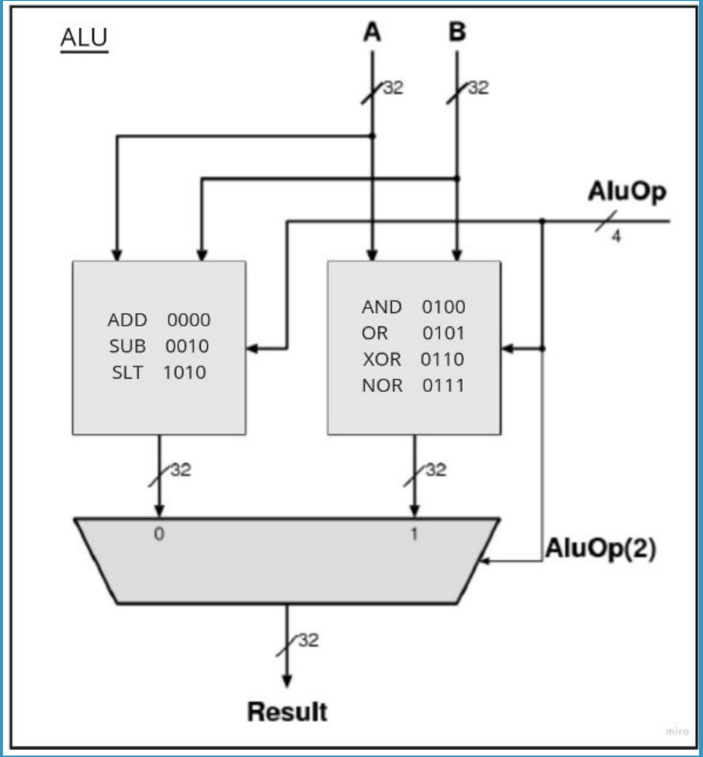
- julisa.lapa@utec.edu.pe
- neftali.calixto@utec.edu.pe

Main Decoder

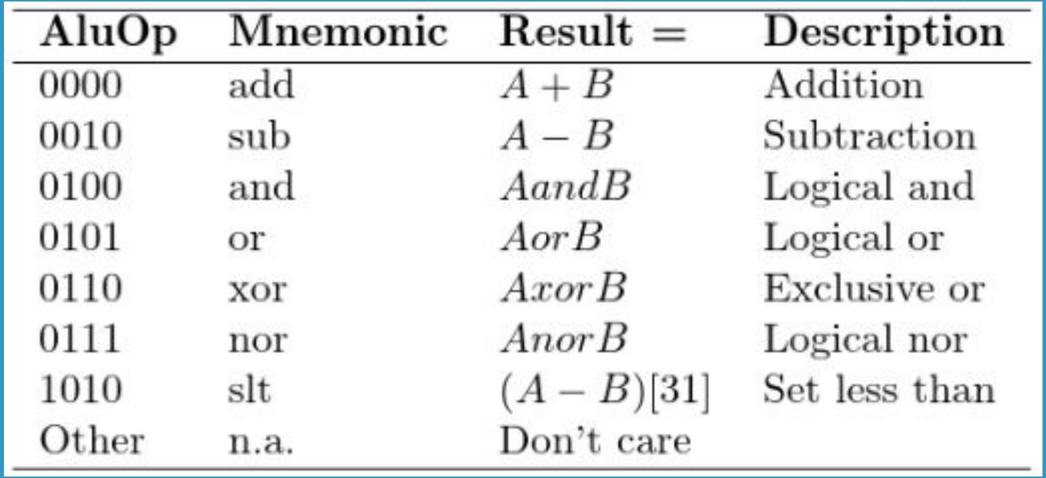


State (Name)	PCWrite	MemWrite	IRWrite	RegWrite	ALUSrcA	Branch	IorD	MemtoReg	RegDst	ALUSrcB[1:0]	PCSrc[1:0]	ALUOp[1:0]	FSM Control Word
0 (Fetch)	1	0	1	0	0	0	0	0	0	01	00	00	0x5010
1 (Decode)	0	0	0	0	0	0	0	0	0	11	00	00	0x0030
2 (MemAdr)	0	0	0	0	0	1	0	0	0	01	00	00	0x0420
3 (MemRd)	0	0	0	0	0	0	0	1	0	00	00	00	0x0100
4 (MemWB)	0	0	1	0	0	0	0	1	0	00	00	00	0x0880
5 (MemWr)	0	0	0	0	1	0	0	0	1	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	01	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	0x0800
11 (JEx)	1	0	0	0	0	0	0	0	0	00	10	00	0x4008

ALU

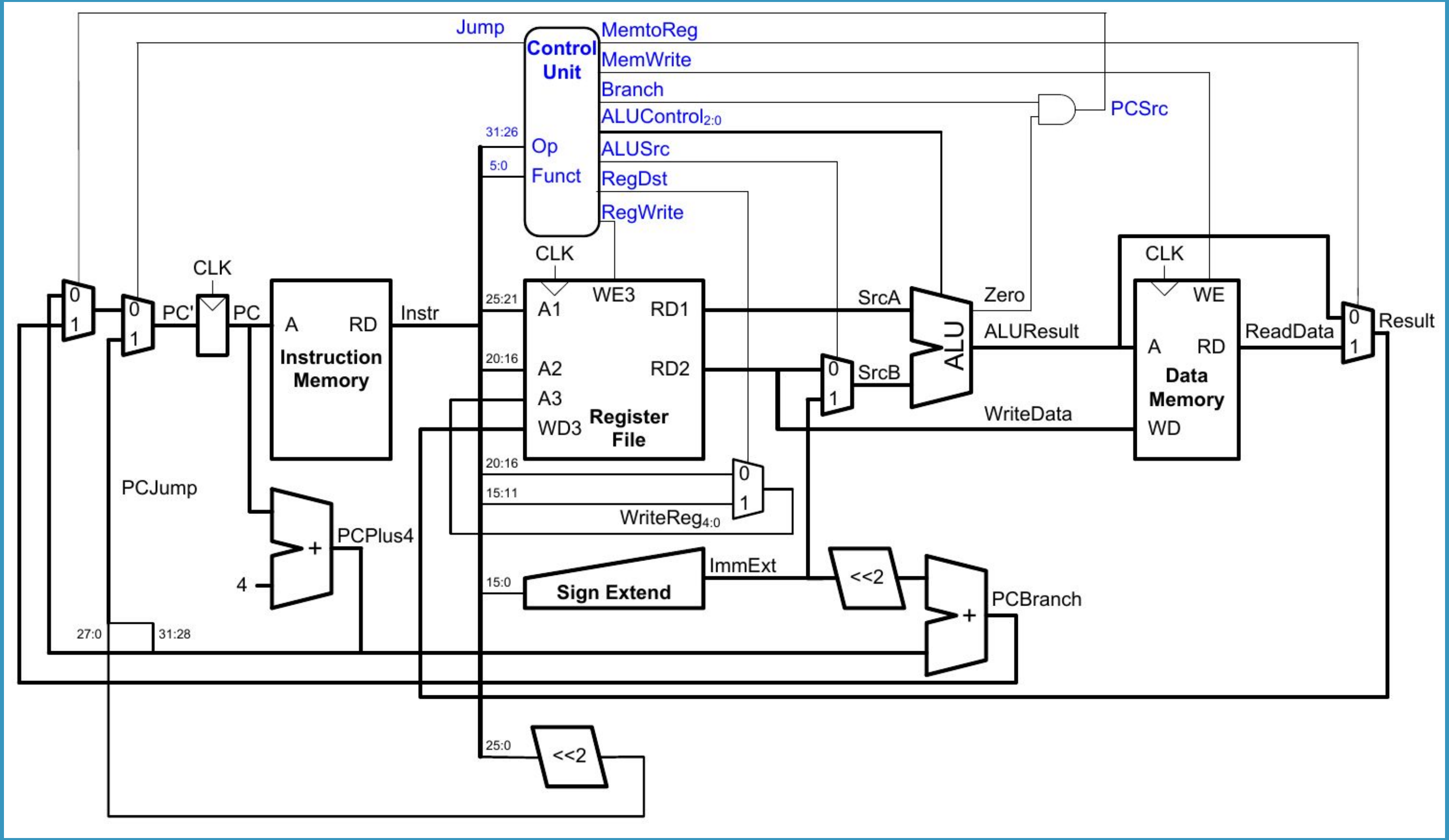


Instrucciones:

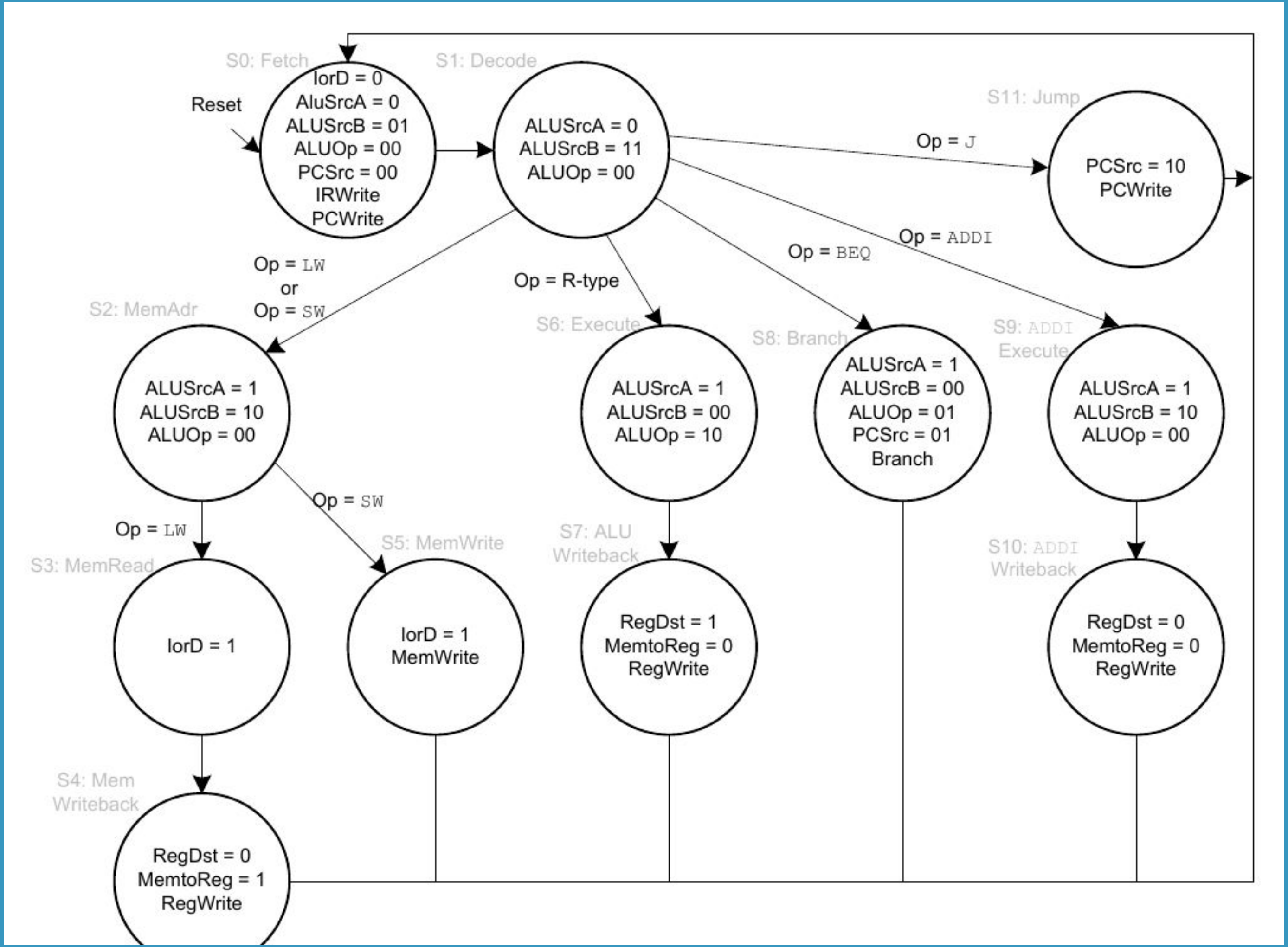


AluOp	Mnemonic	Result =	Description
0000	add	$A + B$	Addition
0010	sub	$A - B$	Subtraction
0100	and	$A and B$	Logical and
0101	or	$A or B$	Logical or
0110	xor	$A xor B$	Exclusive or
0111	nor	$A nor B$	Logical nor
1010	slt	$(A - B)[31]$	Set less than
Other	n.a.	Don't care	

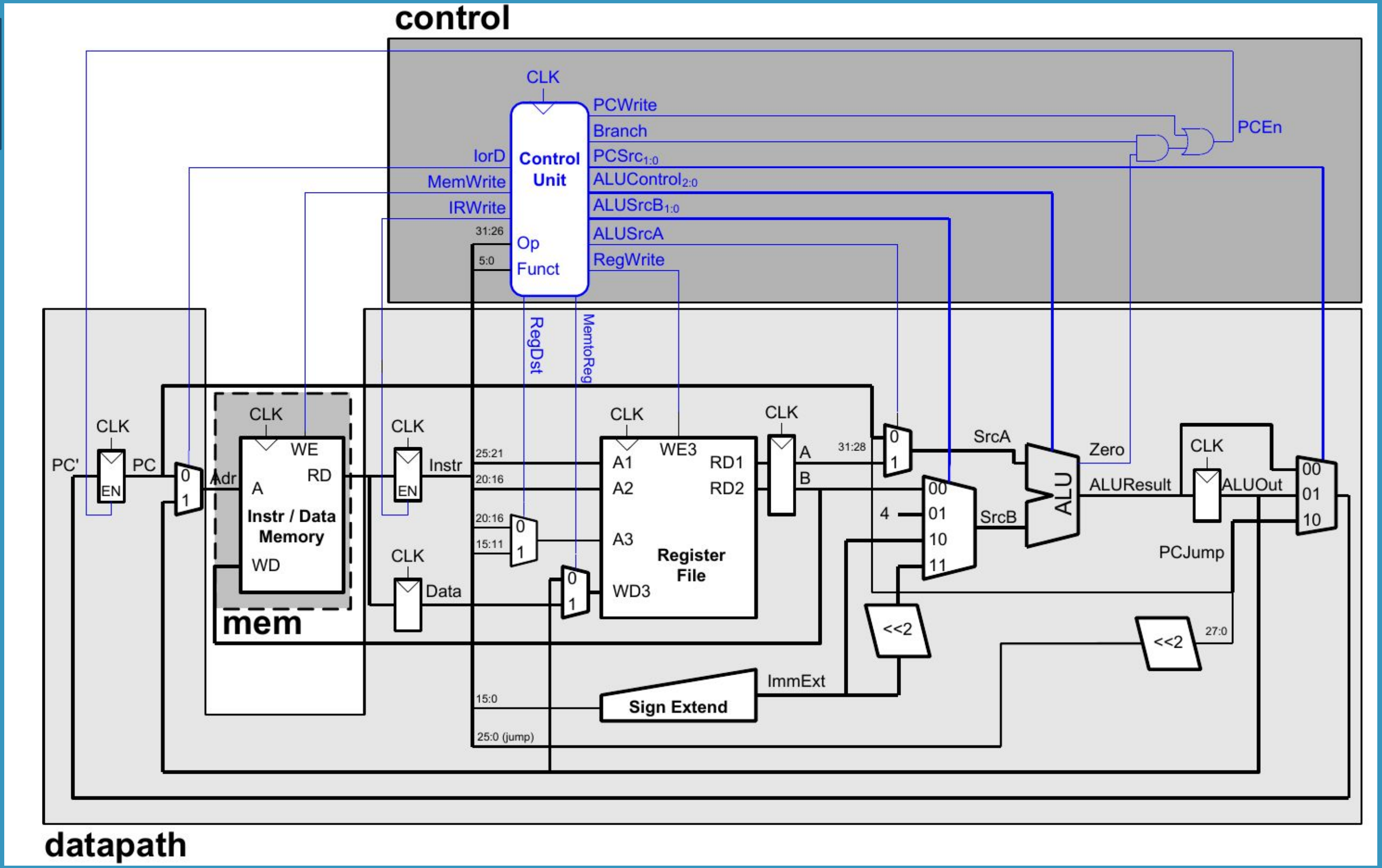
Single-cycle MIPS processor



Multi-cycle FSM



Multi-cycle MIPS processor


control
CLK
Control Unit
PCWrite
Branch
PCSrc1:0
ALUControl2:0
ALUSrcB1:0
ALUSrcA
RegWrite
IorD
MemWrite
IRWrite
31:26
5:0
Op
Funct
PCEn
RegDst
MemtoReg
PC'
PC
EN
Adr
A
WE
RD
Instr / Data Memory
WD
mem
Instr
Data
25:21
20:16
15:11
A1
A2
A3
WD3
WE3
RD1
RD2
Register File
31:28
SrcA
SrcB
4
ALU
Zero
ALUResult
ALUOut
PCJump
<<2
27:0
ImmExt
Sign Extend
15:0
25:0 (jump)
datapath