

LA MAQUINA SENCILLA:
 Introducción a la Estructura Básica
 de un Computador.

COMPUTADORS

Máquina Sencilla: de 1988 a 2016

La Máquina Sencilla
 Trabajo Final de los alumnos de
 Diseño de Sistemas con FPGA
 Primer cuatrimestre 2016
 Profesora Patricia Borensztein

LA MAQUINA SENCILLA:
 Introducción a la Estructura Básica
 de un Computador.

Miguel Valero
 Eduard Ayguadé

COMPUTADORS

La Máquina Sencilla

- La Máquina Sencilla fue diseñada por Miguel Valero y Eduard Ayguadé, ambos profesores de la Facultad de Informática de Barcelona (UPC, Cataluña, España)
- Fue diseñada en 1988 para que el alumno aprendiera los fundamentos de la estructura básica de un computador.

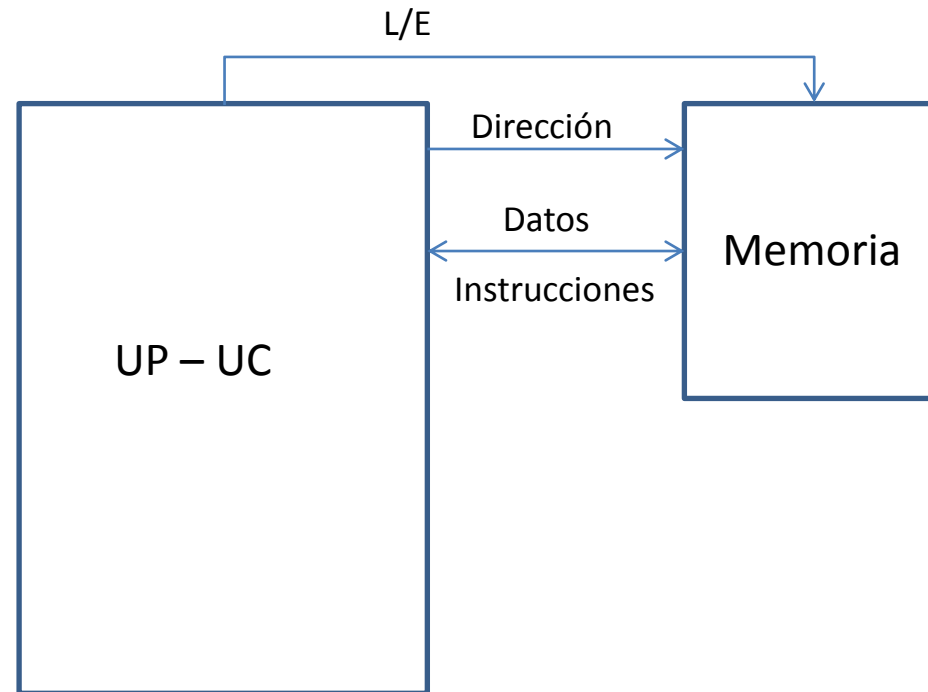
La Máquina Sencilla

- El primer diseño presenta una CPU que puede ejecutar cuatro instrucciones: ADD, MOV, CMP y BEQ
- Tanto las instrucciones como los datos están en memoria, y las instrucciones son memoria-memoria. Es decir, no hay banco de registros visibles a nivel lenguaje máquina.
- Dado que una FPGA es básicamente memoria estática (LUTs, FF, BRAM) estas se adaptan perfectamente a la estructura y arquitectura original de la máquina sencilla.

Estructura del Trabajo

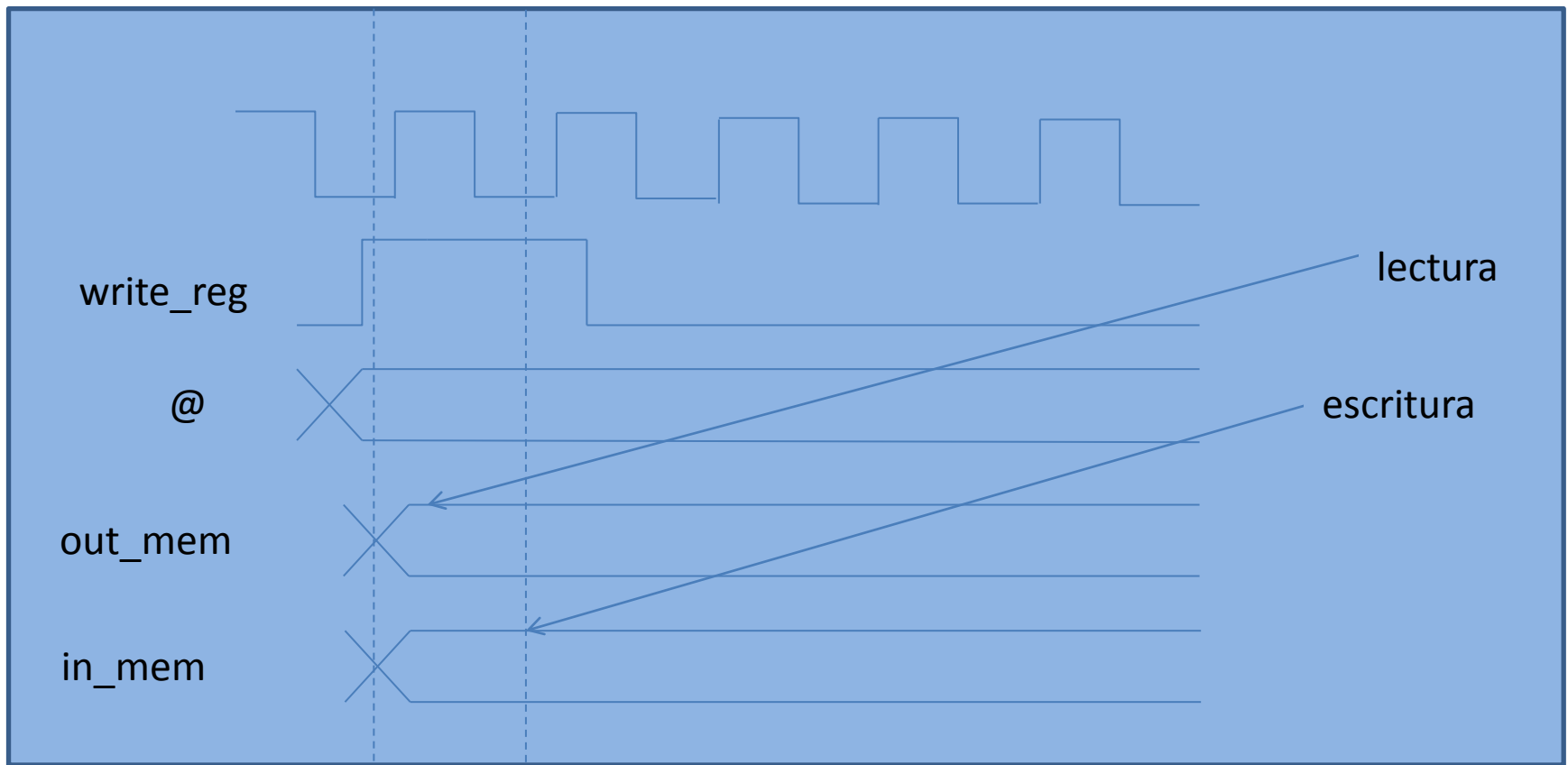
- Primera Parte:
 - Implementación de la Máquina Sencilla con sus extensiones, instrucciones de entrada y salida y soporte para subrutinas
- Segunda Parte:
 - Definición e Implementación de una interface (bus) para los controladores de entrada/salida
 - Implementación de controladores básicos: leds, switches, pushButtons, uart, 7-segmentos.
- Tercera Parte:
 - Programas de Prueba de controladores
 - Incorporación de nuevos controladores
- Cuarta Parte:
 - Ensamblador: funcionalidades
 - Otras herramientas: desensamblador, simulador

La Máquina Sencilla: Primera Parte



La Memoria

- Las instrucciones y datos se guardaran en Memoria Distribuida (Lut Ram) debido a que el tamaño total requerido para la memoria es pequeño (128 words de 16 bits)
- La salida de la memoria se escribe en los registros al final del ciclo
- La lectura es combinacional, es decir, asíncrona



La Memoria

```
module RAM(  
    input clk, le,  
    input [6:0] dir,  
    input [15:0] ent,  
    output [15:0] sal  
);  
  
parameter RAM_WIDTH = 16;  
parameter RAM_ADDR_BITS = 7;  
  
reg [RAM_WIDTH-1:0] M[(2**RAM_ADDR_BITS)-1:0];  
  
initial  
    begin  
        $readmemb("shifter.hex",M);  
    end  
  
always @(posedge clk)  
    if (le)  
        M[dir] <= ent;  
  
assign sal = M[dir];  
  
endmodule
```

Instrucciones y Formato

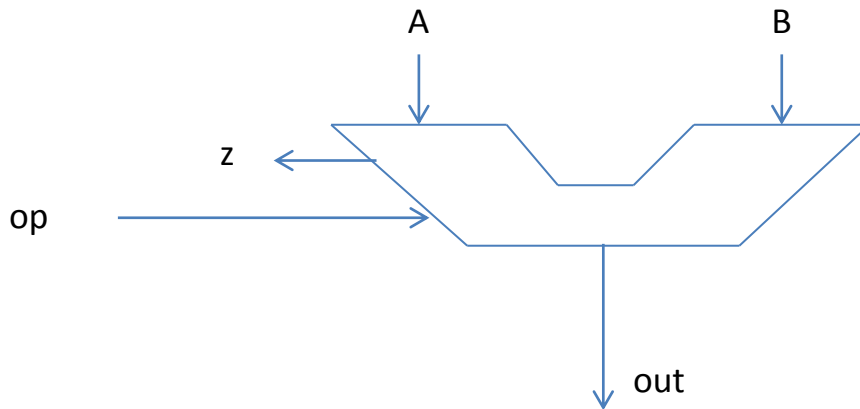
- Las instrucciones son memoria-memoria:
 - Tamaño: 16 bits
 - Código de operación: 2 bits
 - Campo para especificar dirección de memoria: 7 bits
- Instrucciones
 - ADD F, D ; $(D) \leftarrow (F) + (D)$
 - MOV F, D ; $(D) \leftarrow (F)$
 - CMP F, D ; $FZ \leftarrow (F) \text{ xor } (D)$
 - BEQ D ; si (FZ) $PC \leftarrow D$

La Unidad de Proceso

- Registros (no visibles al programador)
 - PC: dirección de instrucción
 - IR: instrucción
 - A, B: registros fuentes para ALU
 - FZ: flag de cero
- ALU (Unidad Aritmético Lógica)
 - Soporta las tres operaciones básicas : sumar, xor, dejar pasar un dato de la entrada a la salida
- Multiplexores
 - Selección de la dirección de memoria

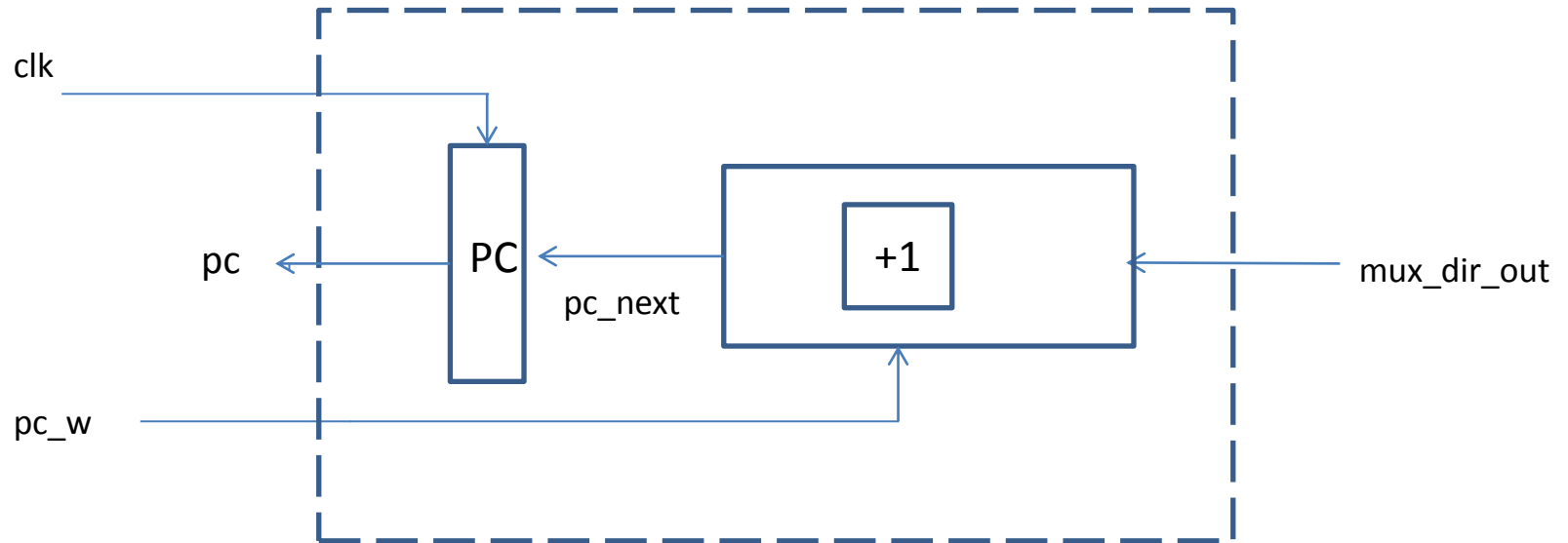
UP: ALU

```
module Alu(  
    input [15:0] A,  
    input [15:0] B,  
    input [1:0] op,  
    output z,  
    output reg [15:0] out  
);
```



```
always @*  
begin  
    case (op)  
        0: // Suma  
            out=(A+B);  
        1: // XOR  
            out=A^B;  
        2: // B  
            out=B;  
        default:  
            out=16'b0;  
    endcase  
end  
  
assign z = (!out)? 1'b1: 1'b0;  
  
endmodule
```

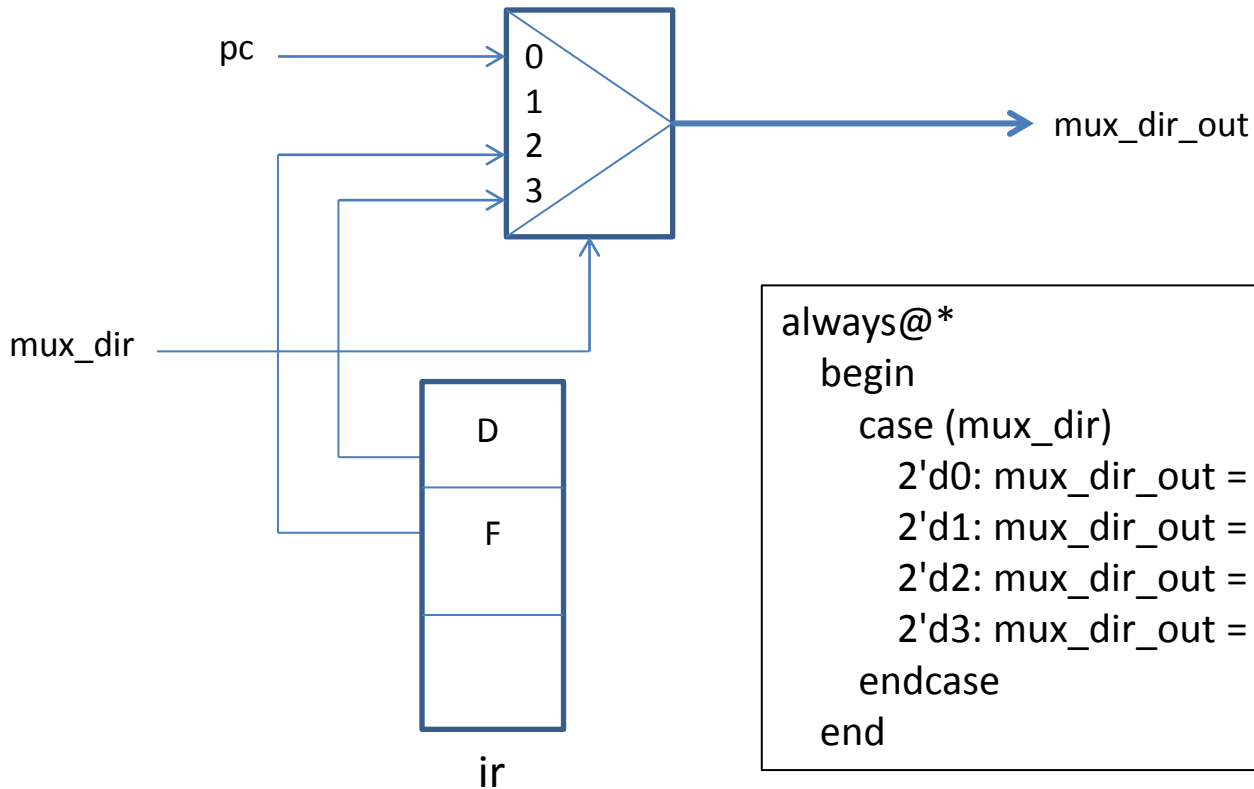
UP: registro PC



```
always @ (posedge clk)
begin
    if (reset)
        pc<=0;
    else
        pc<=pc_next;
end
```

```
always @*
pc_next = pc;
if (pc_w)
    pc_next = mux_dir_out + 1'd1;
```

UP: Selector de Direcciones



```
always@*
begin
  case (mux_dir)
    2'd0: mux_dir_out = pc;
    2'd1: mux_dir_out = 7'b0; // vacía
    2'd2: mux_dir_out = ir[13:7];
    2'd3: mux_dir_out = ir[6:0];
  endcase
end
```

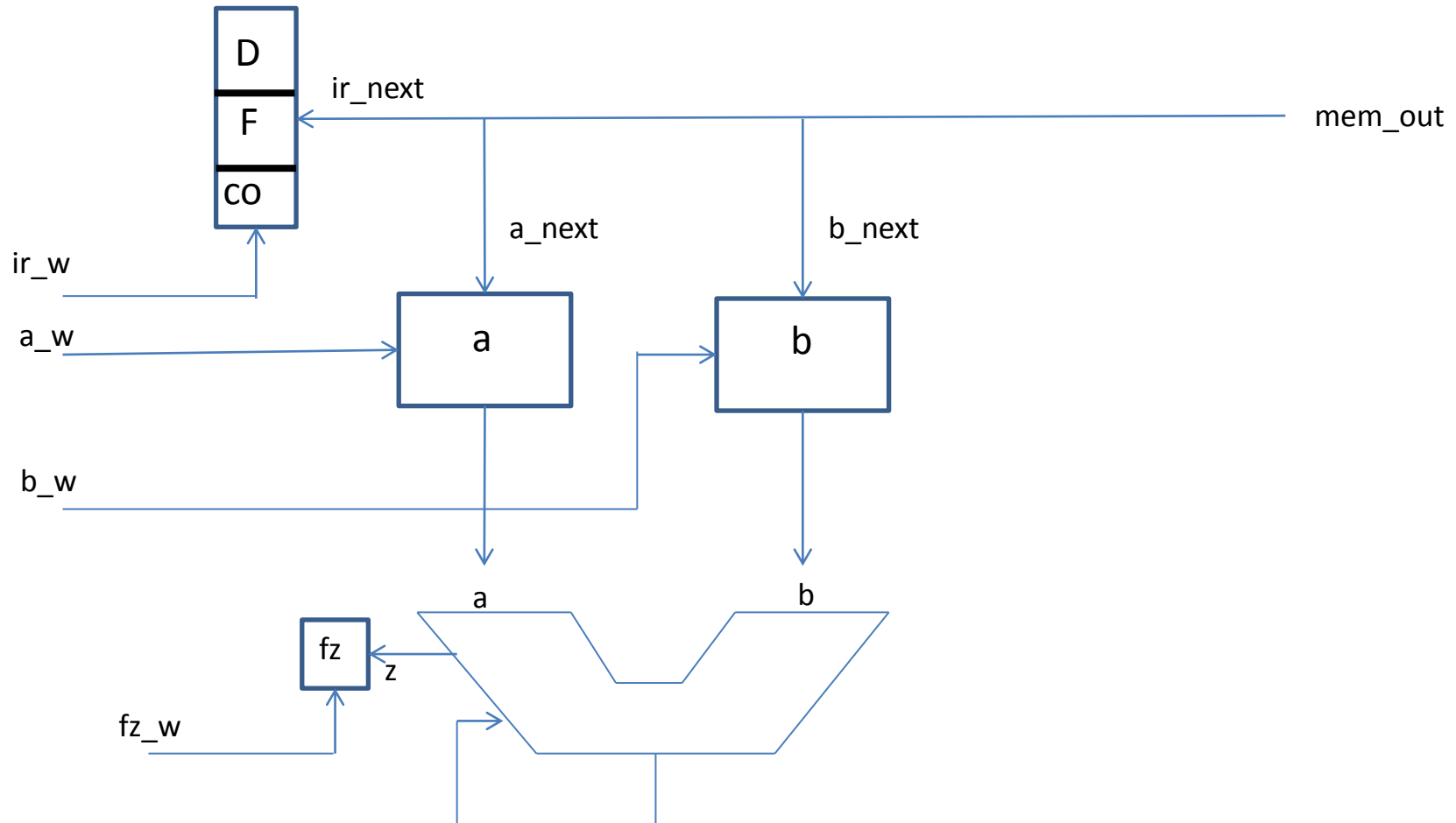
UP: registros pc, a, b, ir, fz

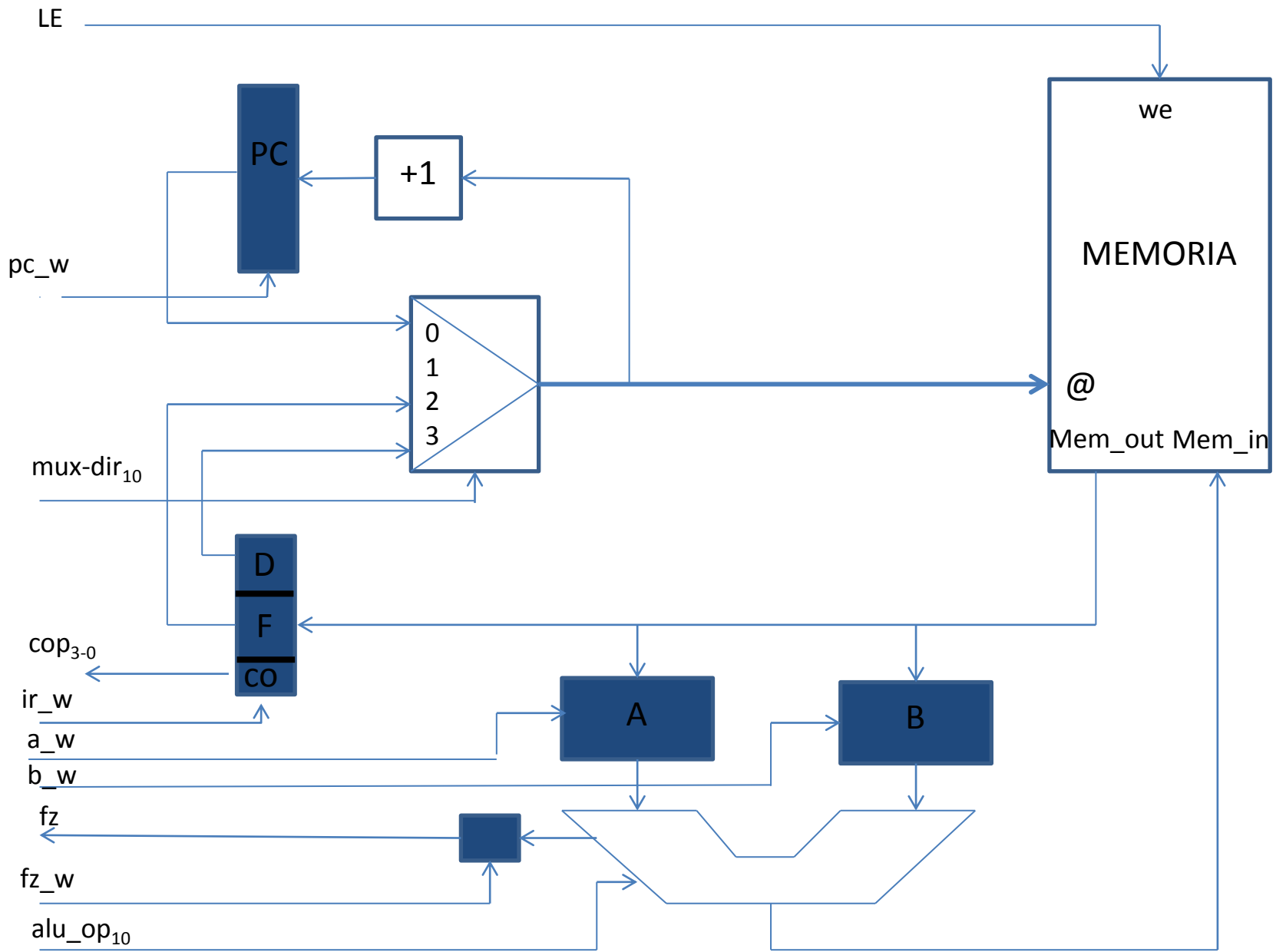
```
always@(posedge clk)
begin
  if (reset)
  begin
    ir <= 16'd0;
    pc <= 7'd0;
    a <= 16'd0;
    b <= 16'd0;
    fz <= 1'd0;
  end
  else
  begin
    ir <= ir_next;
    pc <= pc_next;
    a <= a_next;
    b <= b_next;
    fz <= fz_next;
  end
end
```

```
always@*
begin
  pc_next = pc;
  ir_next = ir;
  a_next = a;
  b_next = b;
  fz_next = fz;

  if (pc_w)
    pc_next = mux_dir_out + 1'd1;
  if (ir_w)
    ir_next = mem_out;
  if (a_w)
    a_next = mem_out;
  if (b_w)
    b_next = mem_out;
  if (fz_w)
    fz_next = alu_z;
end
```

UP: registros A,B,IR

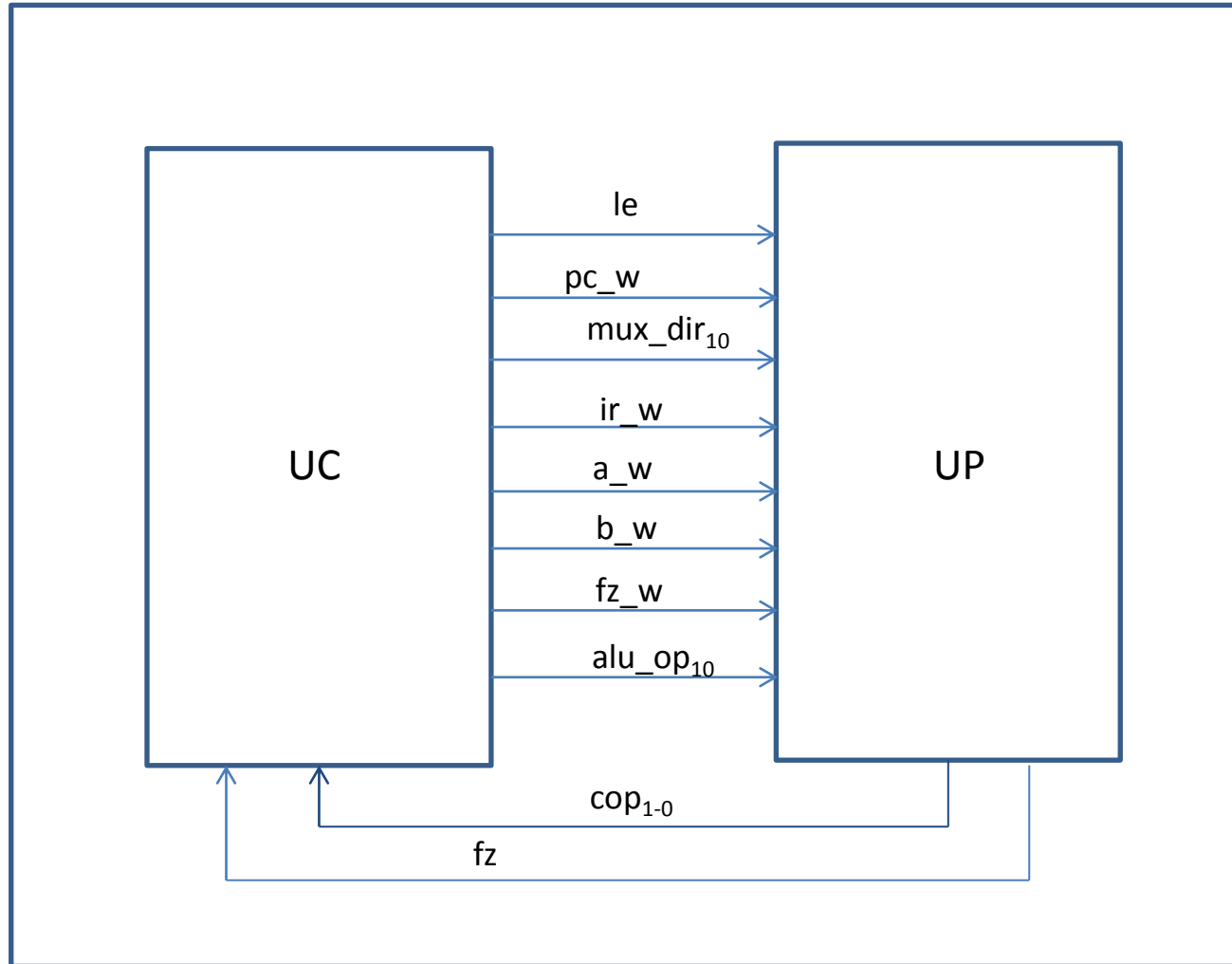




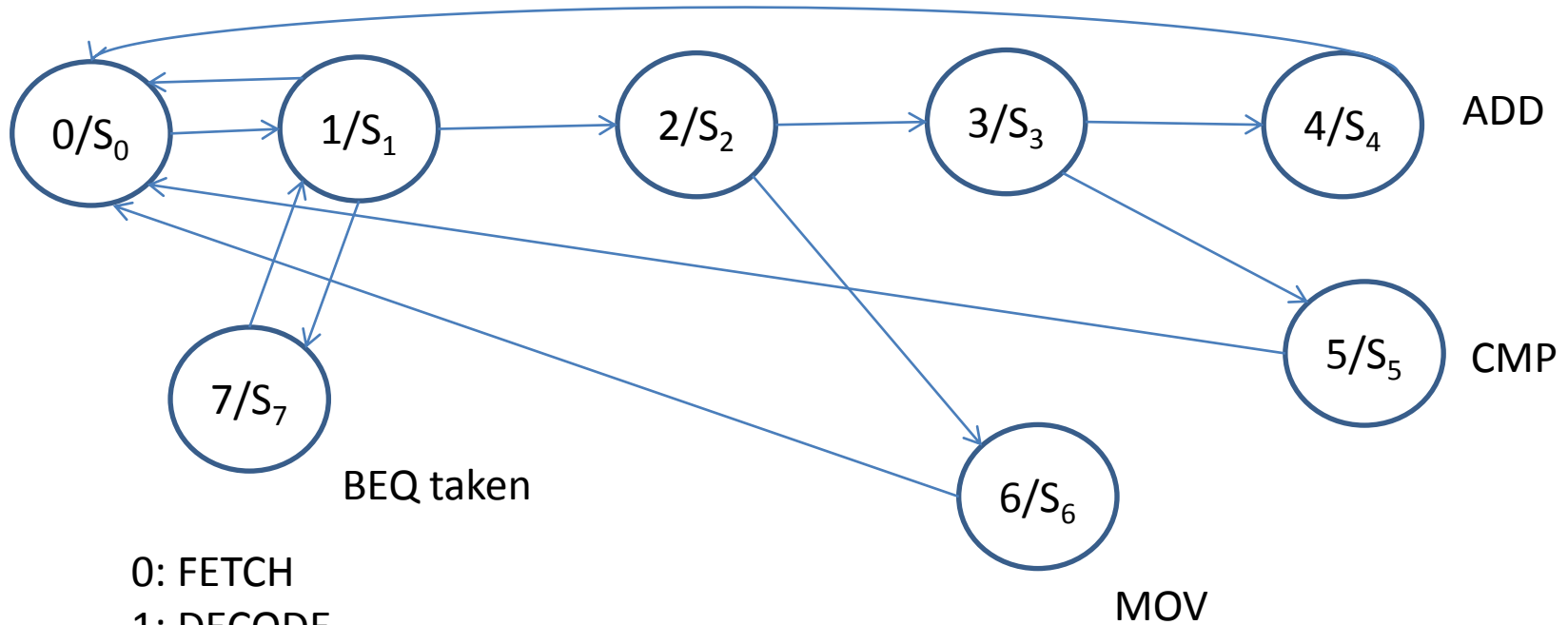
UP

```
module UP(  
    input clk,  
    input reset,  
    input [1:0] mux_dir, alu_op,  
    input le, pc_w, ir_w, a_w, b_w, fz_w,  
    output reg fz,  
    output [1:0] cop  
);
```


UP-UC



UC: Grafo de Estados



0: FETCH

1: DECODE

2: BUSQUEDA OP 1

3: BUSQUEDA OP 2

4: EJECCIÓN ADD

5: EJECCIÓN CMP

6: EJECCIÓN MOV

7: EJECCIÓN BEQ

CPI por instrucción

- Una métrica en las arquitecturas (sencillas) es el CPI (Ciclos por Instrucción), es decir, el número de ciclos que tarda una instrucción en ejecutarse
- En la MS, las instrucciones tienen CPI variable:
 - ADD: 5 ciclos
 - CMP: 5 ciclos
 - MOV: 4 ciclos
 - BEQ taken: 3 ciclos, pero el tercer ciclo es FETCH, por lo tanto, son 2 ciclos.
 - BEQ not taken: 2 ciclos
- El CPI medio, es una medida que depende de la mezcla de instrucciones de un determinado programa.
- Dado que la MS tiene un clk de 100 MHz, su tiempo de ciclo es de 10nseg.

UC: ESTADOS

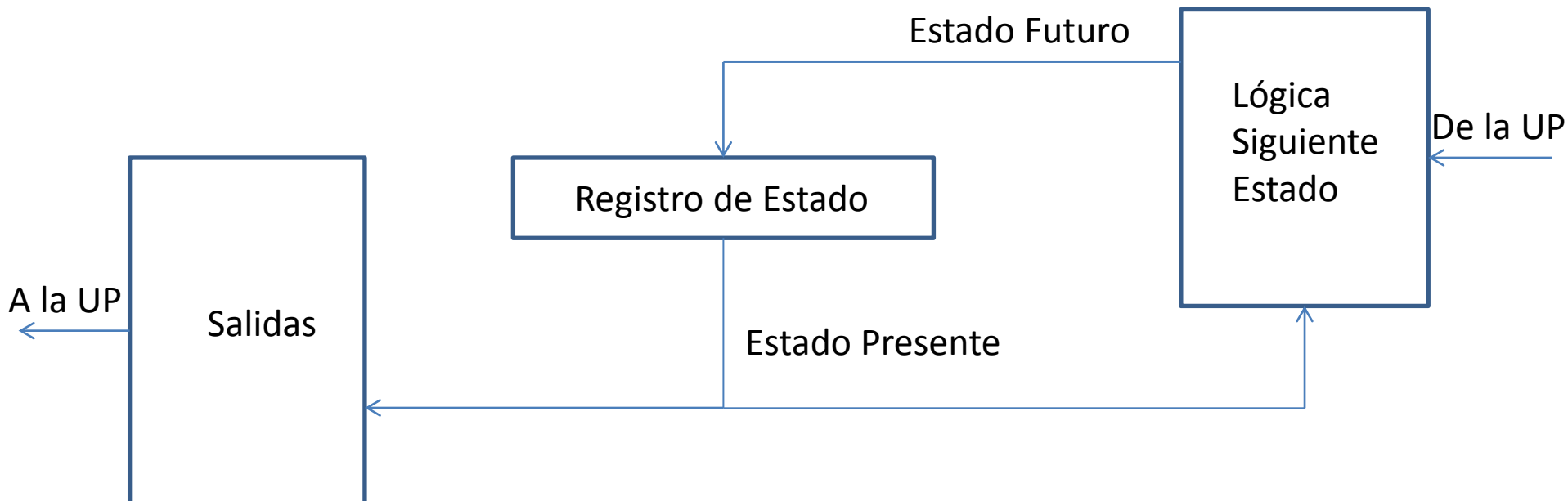
```
fetch  = 5'd0, // FETCH
decode = 5'd1, // DECODE y CONSULTA de FZ
load_f = 5'd2, // LOAD F PRIMER OPERANDO
load_d = 5'd3, // LOAD D SEGUNDO OPERANDO
add    = 5'd4, // SUMA
cmp    = 5'd5, // CMP
mov    = 5'd6, // MOV
JMP    = 5'd7 // SALTO
```

UC: salidas

	S0	S1	S2	S3	S4	S5	S6	S7
le	0	0	0	0	1	0	1	0
pc-w	1	0	0	0	0	0	0	1
mux_dir	00	00	10	11	11	11	11	11
ir_w	1	0	0	0	0	0	0	1
a_w	0	0	0	1	0	0	0	0
b_w	0	0	1	0	0	0	0	0
fz_w	0	0	0	0	1	1	1	0
alu_op	00	00	00	00	00	01	10	00

Unidad de Control

- Está formada por:
 - Un registro de estado
 - Lógica combinacional del siguiente estado
 - Lógica combinacional de las salidas



Unidad de Control: Registro de Estado

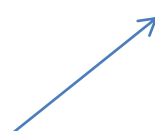
```
always@(posedge clk)
begin
    if (reset)
        estado <= 5'd0;
    else
        estado <= estado_next;
end
```

UC: lógica del siguiente estado

```
always @ *
begin
    case (estado)
    FETCH: estado_next=DECOD;
    DECOD:  casex ({COp[1],COp[0],FZ})
                3'b0xx: estado_next=LOAD_F;// ADD y CMP
                3'b10x: estado_next=LOAD_F;// MOV
                3'b110: estado_next=FETCH;// NO SALTA
                3'b111: estado_next=JMP;// SALTA
            endcase
    LOAD_F:  casex ({COp[1],COp[0],FZ})
                3'b0xx: estado_next=OPE2;// ADD y CMP
                3'b1xx: estado_next=MOV;// MOV
            endcase
    LOAD_D:  casex ({COp[1],COp[0],FZ})
                3'bx0x: estado_next=ADD;// ADD
                3'bx1x: estado_next=COMP;// MOV
            endcase
    ADD:     estado_next=FETCH;
    COMP:    estado_next=FETCH;
    MOV:     estado_next=FETCH;
    BEQ:     estado_next=DECOD;
end
```


UC: lógica de las Salidas

- Las salidas pueden realizarse de tres maneras (al menos)
 1. Mediante expresiones lógicas (ecuaciones) en función del estado: por ejemplo
assign pc_load=(estado==FETCH)?1:0;
 2. Definiendo todas las salidas explícitamente para cada estado: por ejemplo:
 3. Mediante el uso de una ROM para contener todas las salidas. Se direcciona con el registro de estado



```
case (current_state)
    FETCH:
begin
    le=1'b0;
    pc_w=1'b1;
    mux_dir=2'b00;
    ir_w=1'b1;
    a_w=1'b0;
    b_w=1'b0;
    fz_w=1'b0;
    alu_op=2'b00;
end
```

UC con salidas en ROM

UC

```
ROMControl Salidas_MS
(
    .addr(current_state),
    .dout(salida)
);

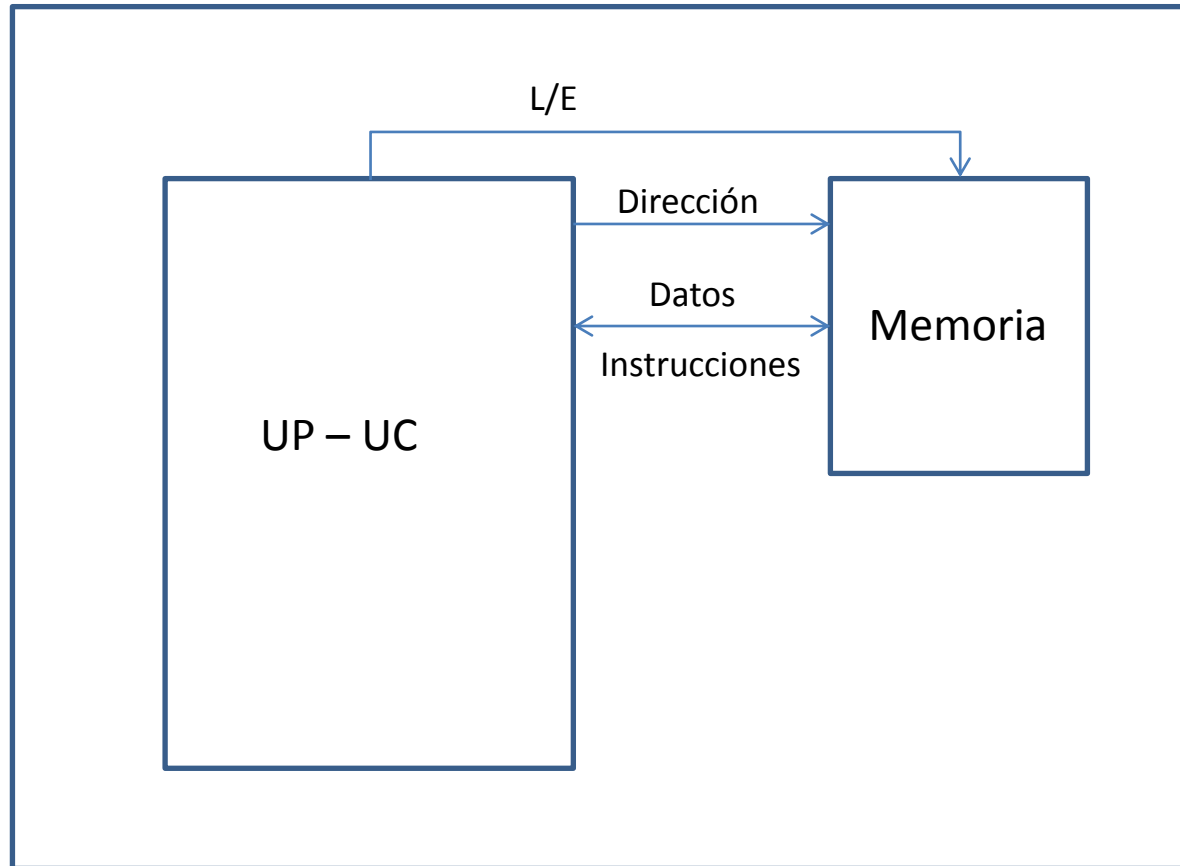
assign mux_dir[1]=salida[9];
assign mux_dir[0]=salida[8];
assign alu_op[1]=salida[7];
assign alu_op [0]=salida[6];
assign le=salida[5];
assign pc_w=salida[4];
assign ir_w=salida[3];
assign a_w=salida[2];
assign b_w=salida[1];
assign fz_w=salida[0];
```

```
module ROMControl(
input [2:0] addr,
output [9:0] dout
);
reg [9:0] ram [7:0]; // 8 palabras (estados) de 10
bits (señales de control)
initial
    begin
        $readmemb("rom.dat",ram, 0, 7);
    end
assign dout = ram[addr];
endmodule
```

rom.dat

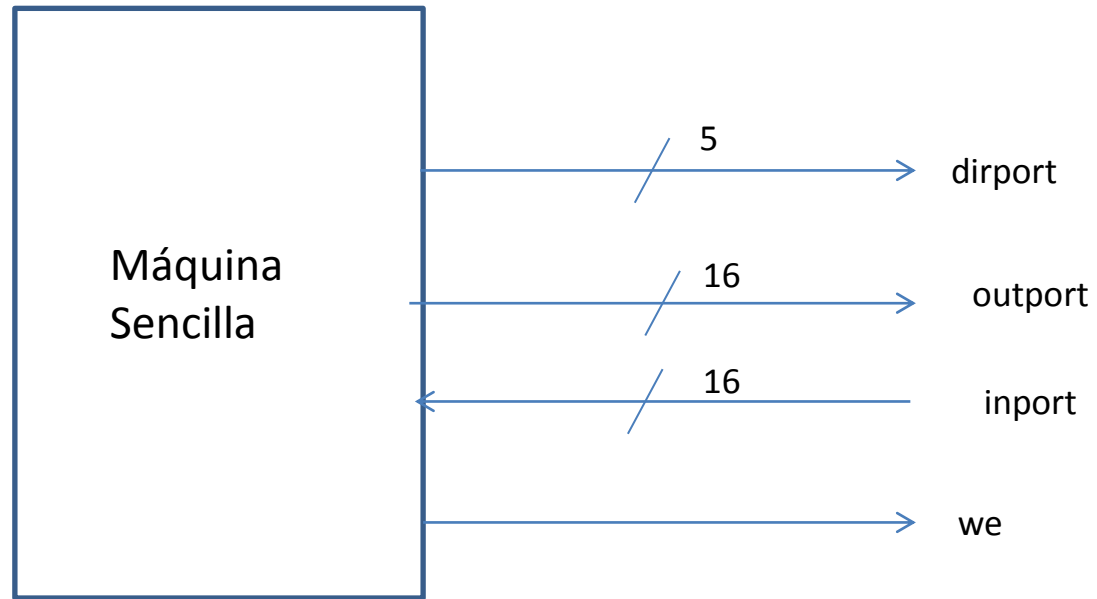
```
0000011000
0000000000
1000000010
1100000100
1100100001
0001000001
1110100001
1100011000
```

Máquina Sencilla

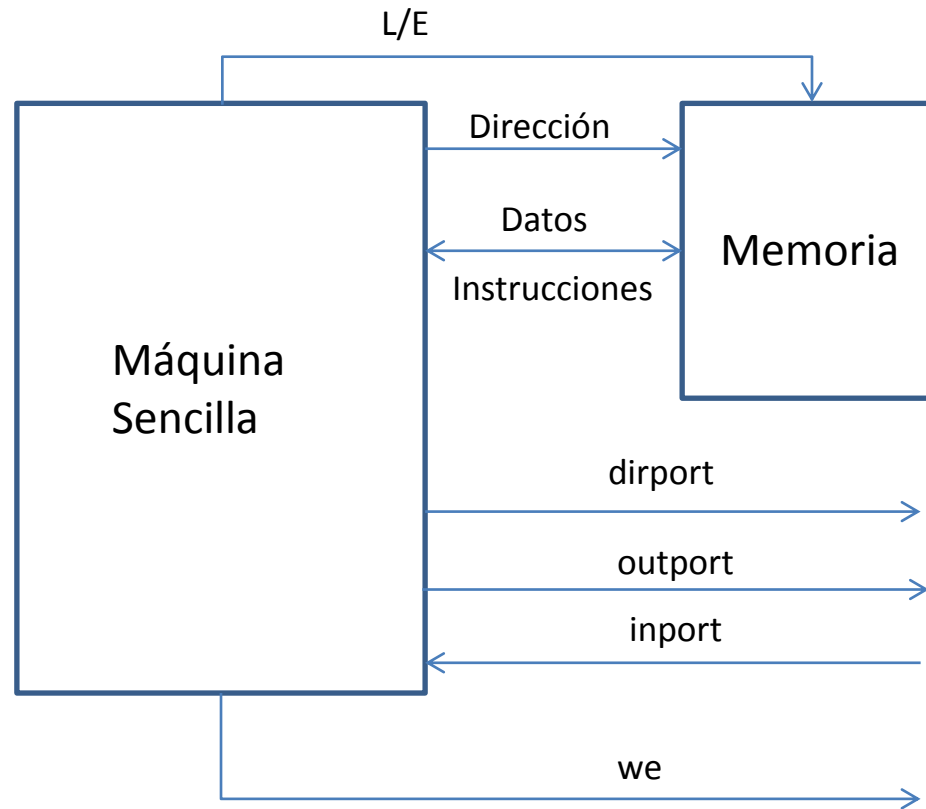


- No tiene entradas ni salidas.....es una caja negra...

Máquina Sencilla con E/S



Máquina Sencilla



Instrucciones de E/S

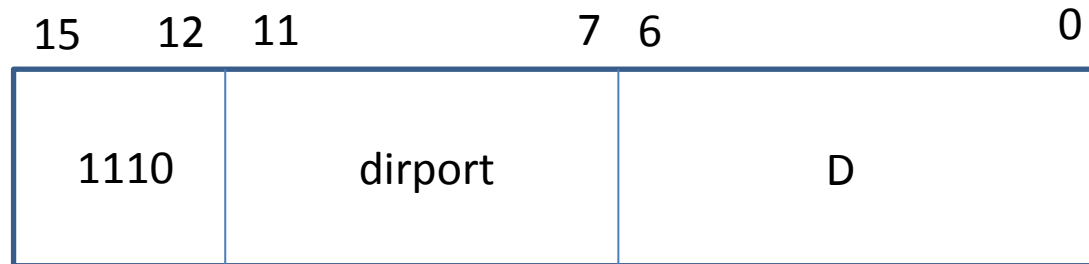
- Se decide:
 - El espacio de direcciones de E/S es independiente al de memoria
 - Se definen las instrucciones IN , OUT
 - IN dirport, D ;;; (D) ← inport
 - OUT dirport, D ;;; outport← (D)
 - Se modifica el formato de instrucción de forma : el código de operación ocupa 4 bits.

Formato de Instrucción

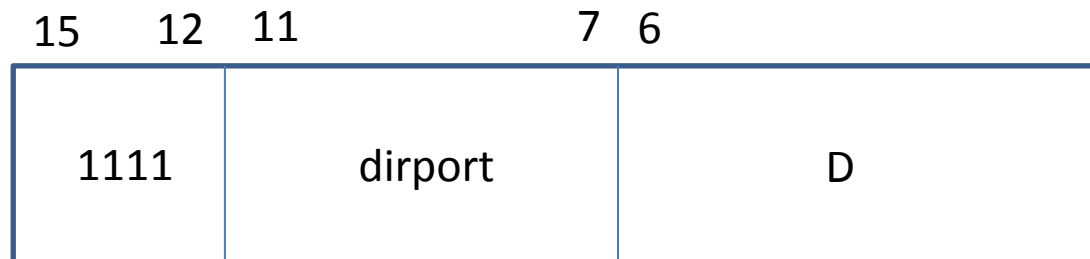
- Los códigos de operación de las seis instrucciones se definen de la siguiente manera:
 - 00xx add
 - 01xx cmp
 - 10xx mov
 - 1100 beq
 - 1110 in
 - 1111 out

Formato de Instrucciones IN-OUT

- Formato IN



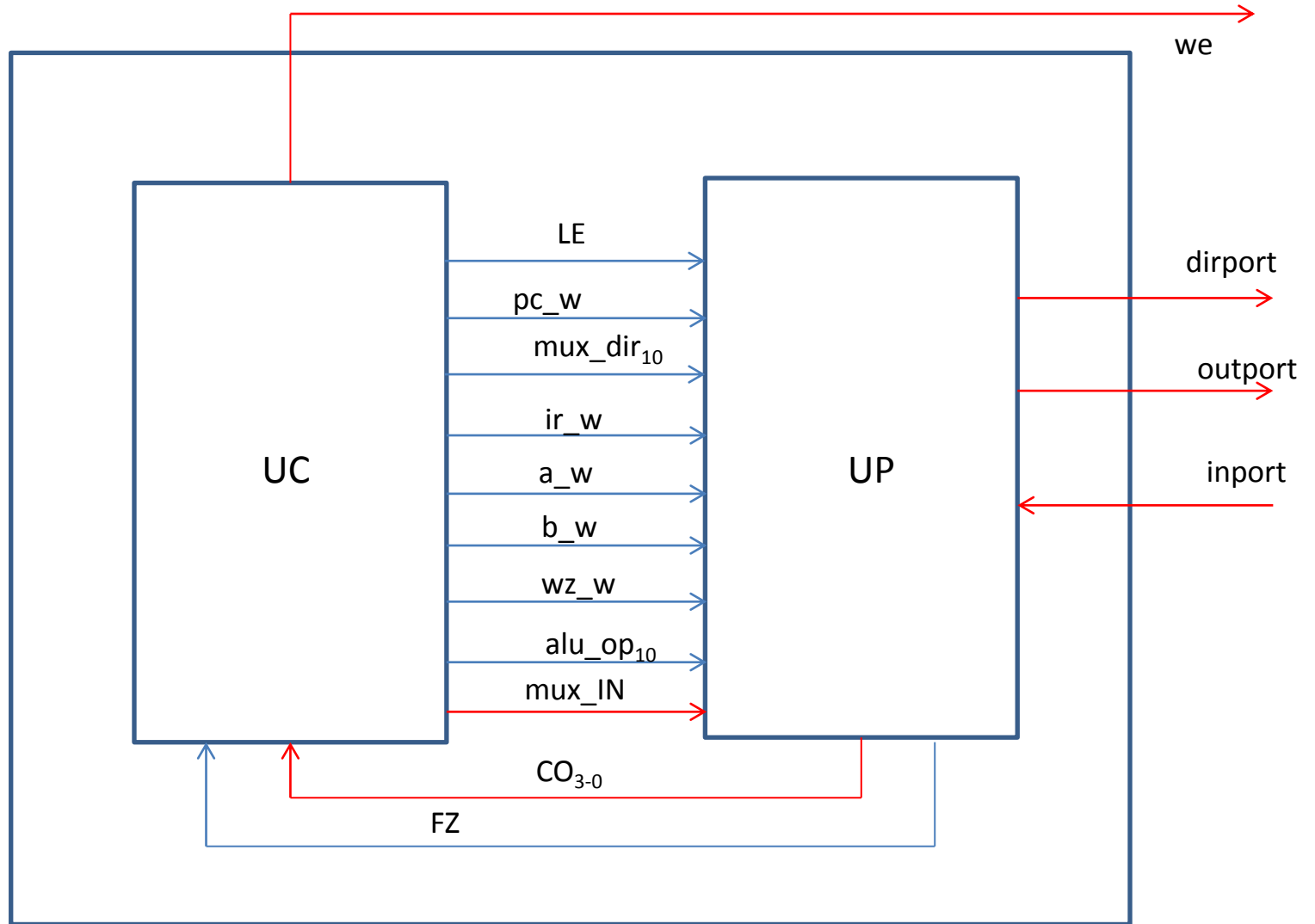
- Formato OUT

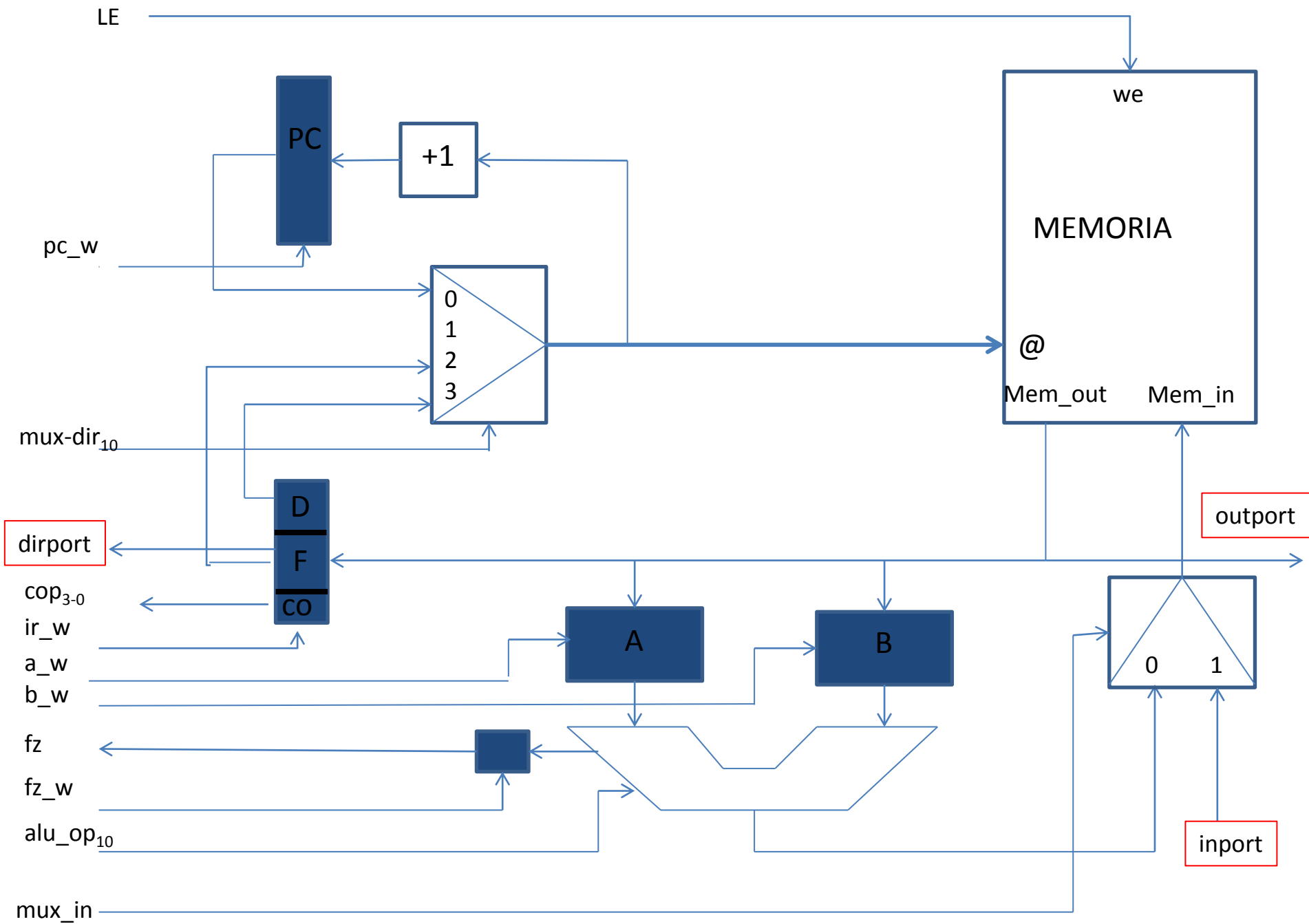


Modificaciones a la UP

- Entrada de Datos:
 - inport: dato de 16 bits proveniente del espacio de E/S
- Salida de Datos:
 - outport: dato de 16 bits proveniente de la Memoria
 - dirport: dato de 5 bits proveniente de los bits 11..7 del registro IR
- Bloques Combinacionales
 - Multiplexor que selecciona si el dato que se escribe en memoria proviene de la ALU o bien de la entrada inport
- Salidas hacia la UC
 - Código de operación: bits 15 a 12
- Entradas desde la UC:
 - Señal de control del multiplexor añadido

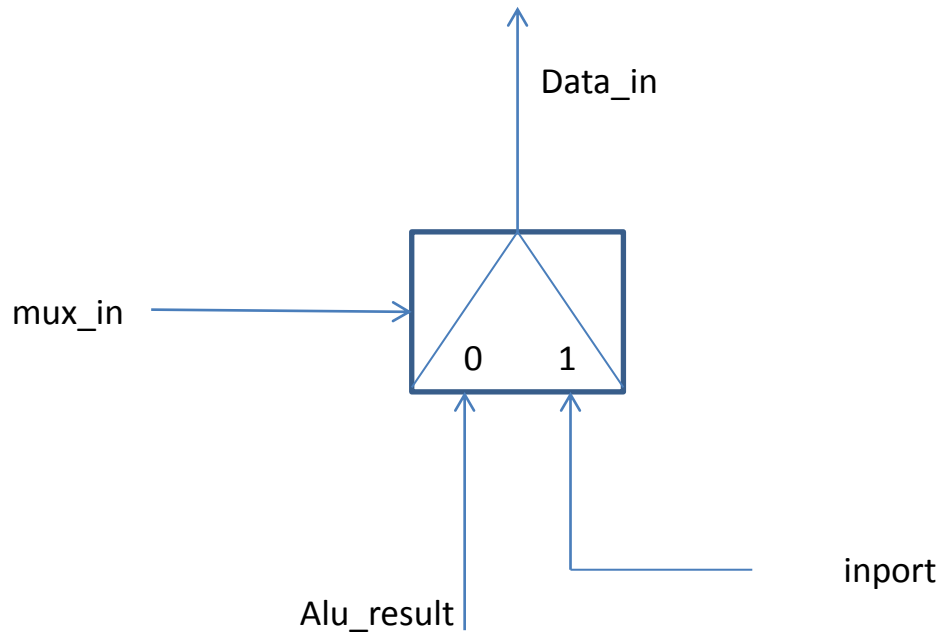
UP-UC





UP: selección de dato de entrada a Memoria

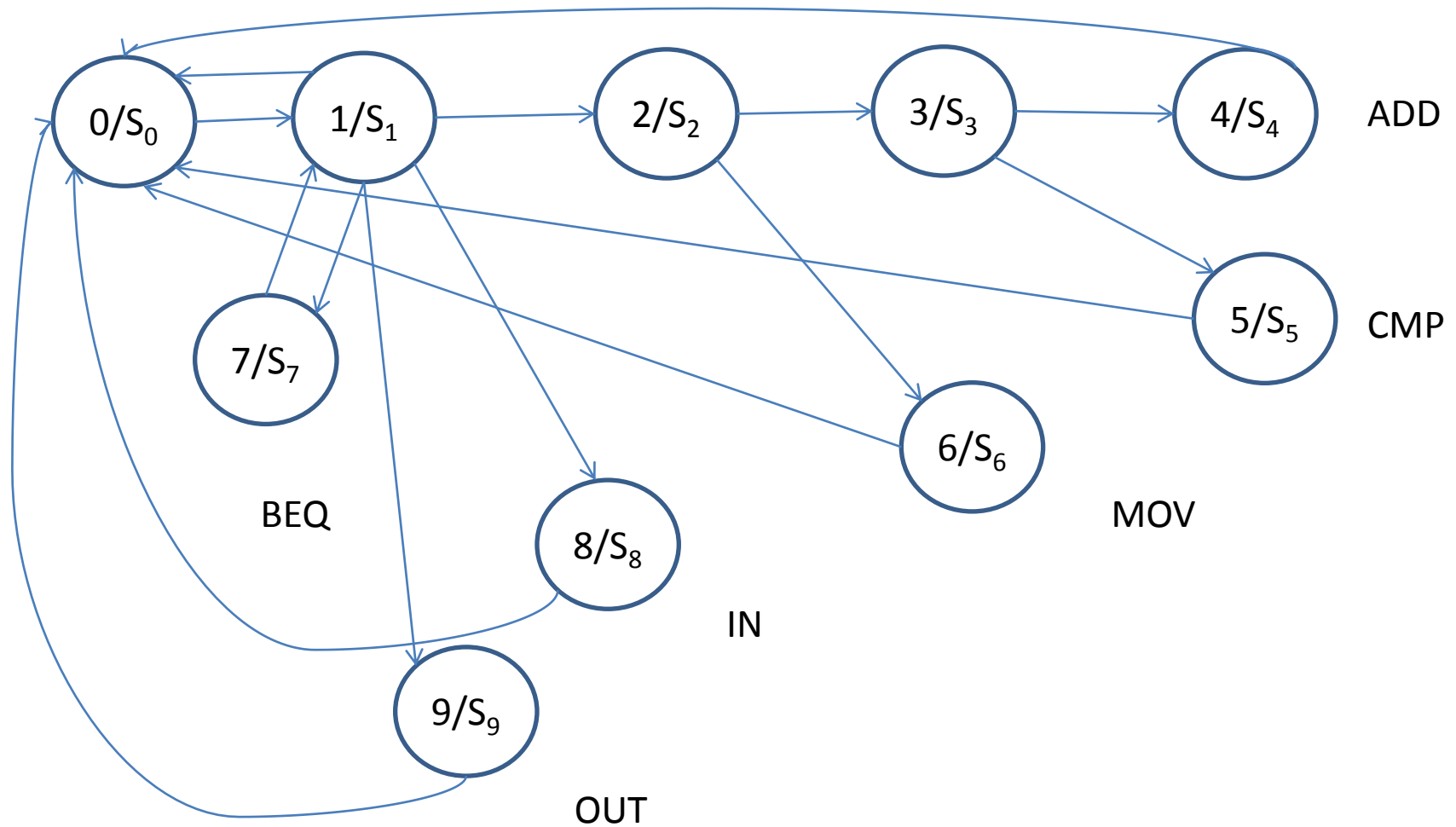
```
// multiplexor para din: ALU o E/S
always @*
begin
    case (mux_in)
    0: Data_in=Alu_result;
    1: Data_in=in_port;
    endcase
end
```



Modificaciones a la UC

- Dos nuevos estados:
 - Las instrucciones IN y OUT se ejecutan en 3 ciclos.
 - Estado de ejecución de IN: (estado 8)
 - mux_in=1; mux_dir=11; LE=1;
 - Estado de ejecución de OUT (estado 9)
 - mux_dir=11; we=1
- Nuevas señales hacia la UP:
 - mux_in
- Nuevas señales hacia el espacio de E/S
 - we

UC: Grafo de Estados



ESTADOS

FETCH=4'b0000, // Fetch: S0

DECOD=4'b0001, // Decodificación y consulta de FZ: S1

LOAD_F=4'b0010, // Búsqueda del Primer Operando S2 : ADD, MOV y CMP

LOAD_D=4'b0011, // Búsqueda del Segundo Operando S3: ADD, MOV

ADD=4'b0100, // ejecución suma S4

CMP=4'b0101, // ejecución cmp S5

MOV=4'b0110, // ejecución mov S6

BEQ=4'b0111, // ejecución beq S7

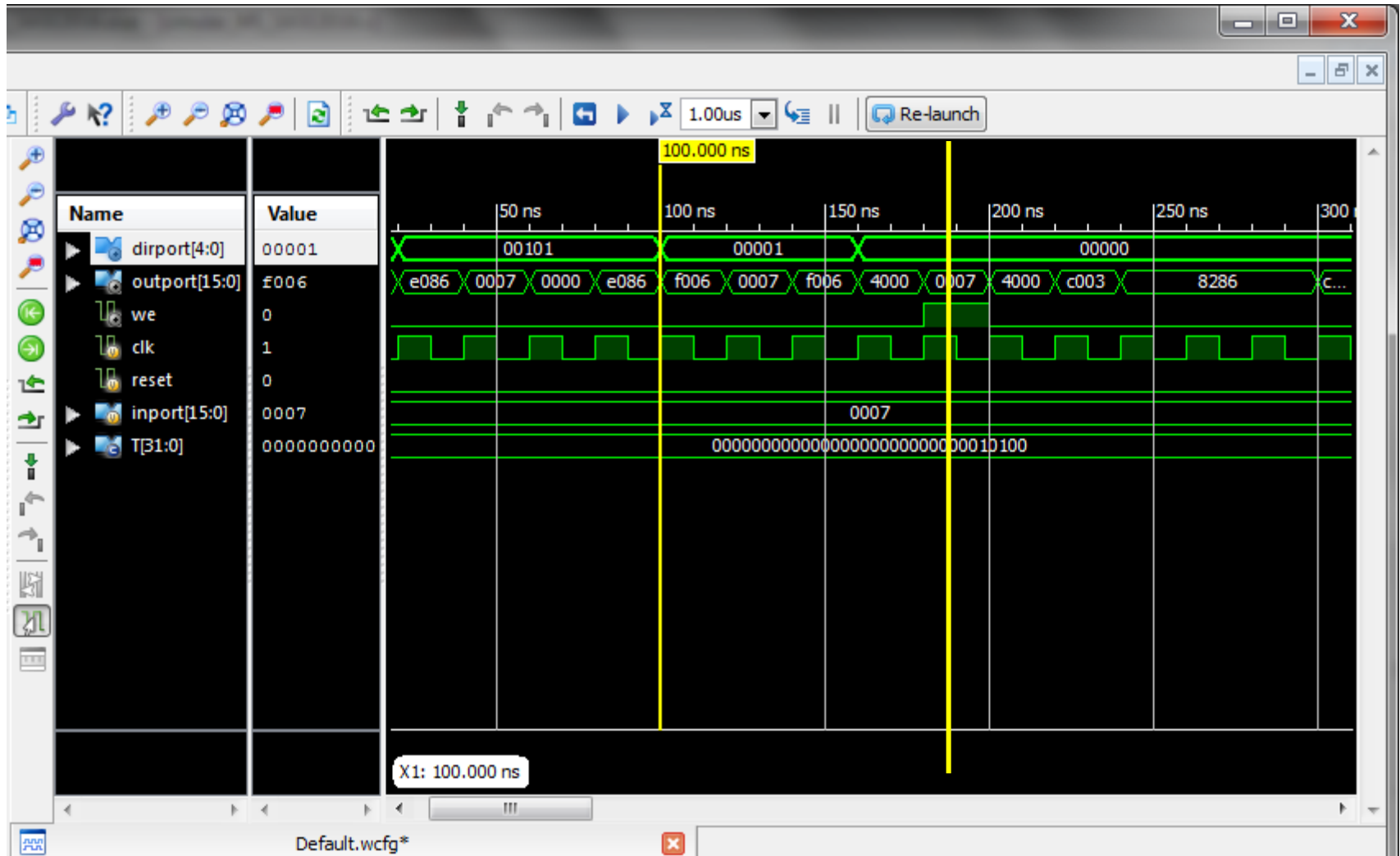
IN=4'b1000, // ejecución in S8

OUT=4'b1001 // S9

UC: salidas

	S0	S1	S2	S3	S4	S5	S6	S7	S8	S9
LE	0	0	0	0	1	0	1	0	1	0
pc_w	1	0	0	0	0	0	0	1	0	0
mux_dir	00	00	10	11	11	11	11	11	11	11
ir_w	1	0	0	0	0	0	0	1	0	0
a_w	0	0	0	1	0	0	0	0	0	0
b_w	0	0	1	0	0	0	0	0	0	0
fz_w	0	0	0	0	1	1	1	0	0	0
mux_in	0	0	0	0	0	0	0	0	1	0
we	0	0	0	0	0	0	0	0	0	1
Alu_op	00	00	00	00	00	01	10	00	00	00

Simulación con E/S: inport=7



Simulación

- La línea amarilla indica el comienzo de la instrucción IN , fase de decodificación: dirport sale fuera.
- OUT. En el tercer ciclo de la ejecución se activa we, y la segunda línea amarilla muestra el valor del dato (outport) valiendo 7

IN 1, a OUT 0, a siempre:JMP siempre
--

Soporte para subrutinas: CALL y RET

- Para soporte de subrutinas es necesario:
 - Ampliar repertorio de instrucciones con dos nuevas instrucciones: CALL y RET
 - Elegir un sitio para salvar y restaurar direcciones de retorno
- Se decidió:
 - Utilizar las posiciones mas altas de memoria para implementar una pila de direcciones de retorno
 - No se soporta recursividad ni tampoco una cantidad mayor a N de llamadas identadas.

Instrucciones de CALL y RET

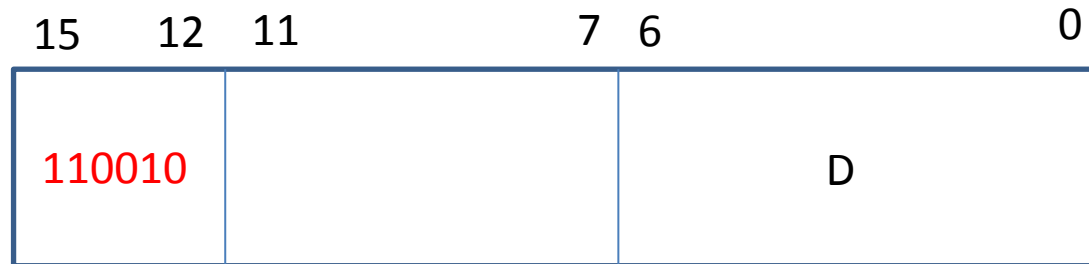
- Se definen las instrucciones CALL, RET
 - CALL D ;;; PUSH PC; IR \leftarrow (D);
 - RET ;;; POP PC
- Para implementar las instrucciones de PUSH y POP se utiliza un nuevo registro SP
- SP está inicializado en el reset a 0
- La operación PUSH, decrementa PC y escribe el contenido del PC en la memoria
- La operación POP lee la memoria apuntada por SP y carga PC con el valor. Luego incrementa el SP.

Formato de Instrucción

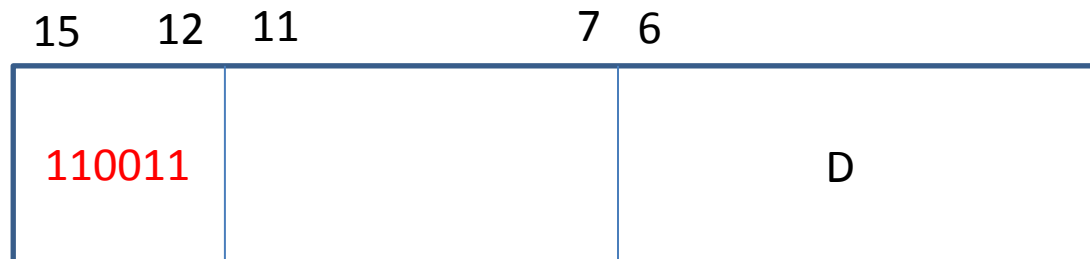
- Los códigos de operación de las ocho instrucciones se definen de la siguiente manera:
 - 00xx add
 - 01xx cmp
 - 10xx mov
 - 1100 beq
 - 1110 in
 - 1111 out
 - 110010 call
 - 110011 ret

Formato de Instrucciones CALL-RET

- Formato CALL

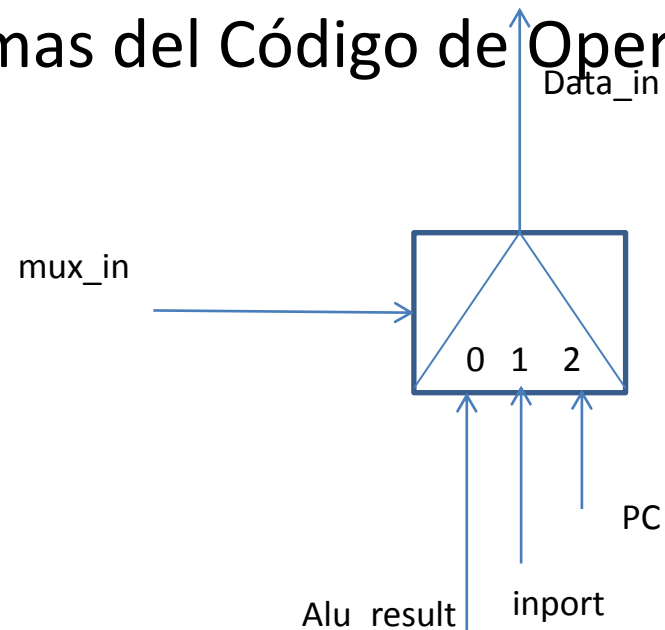
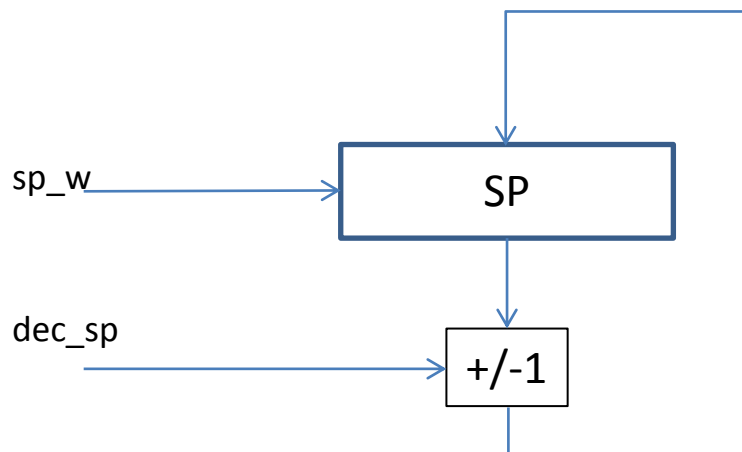


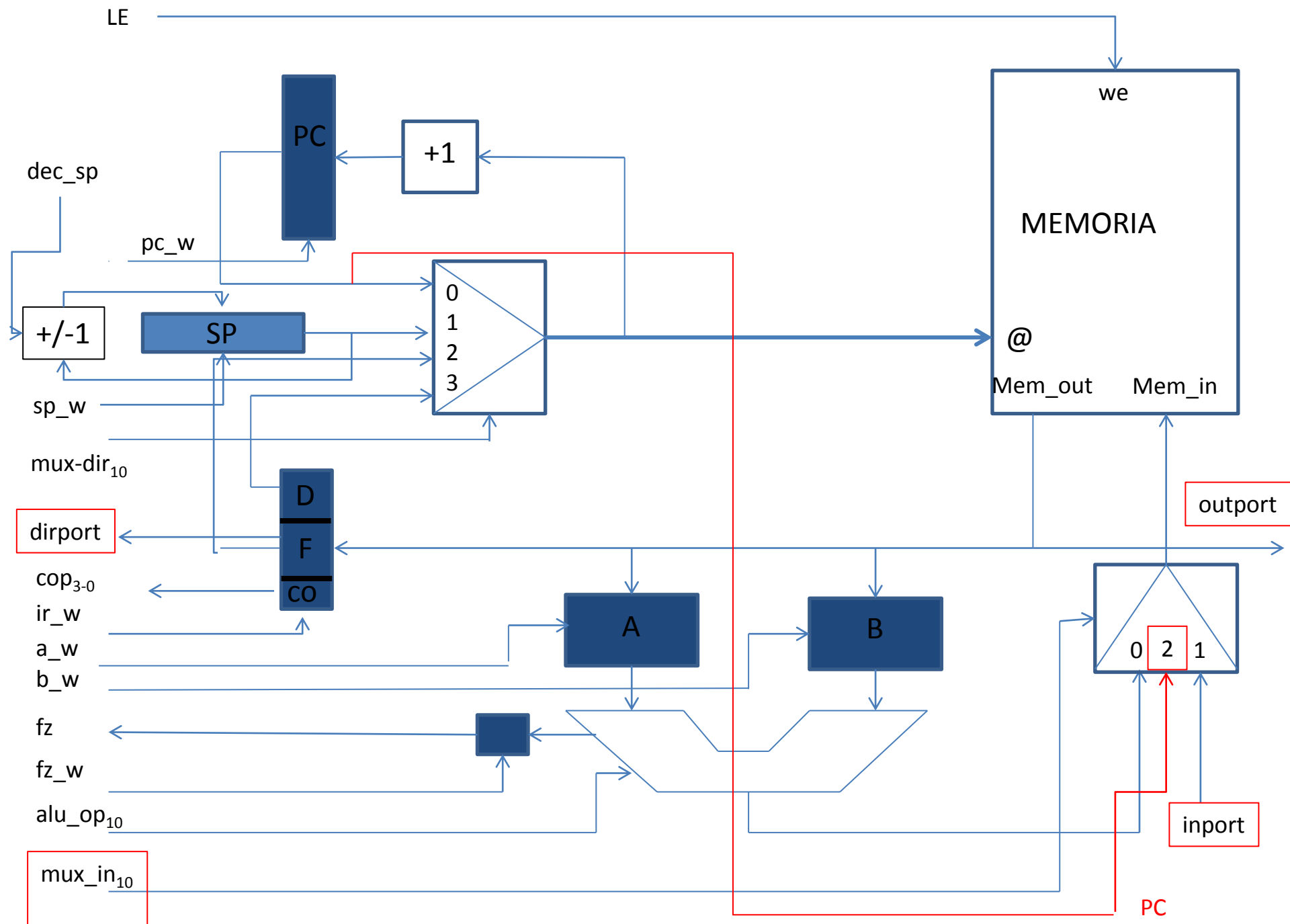
- Formato RET



Cambios en la UP

- Registro SP, puntero a memoria, con lógica para sumar/restar 1. El registro utiliza la entrada 1 del multiplexor de direcciones.
- Una entrada mas en el multiplexor de entrada a la Memoria, proveniente del PC
- Dos (podría ser uno) bits mas del Código de Operación hacia la UC

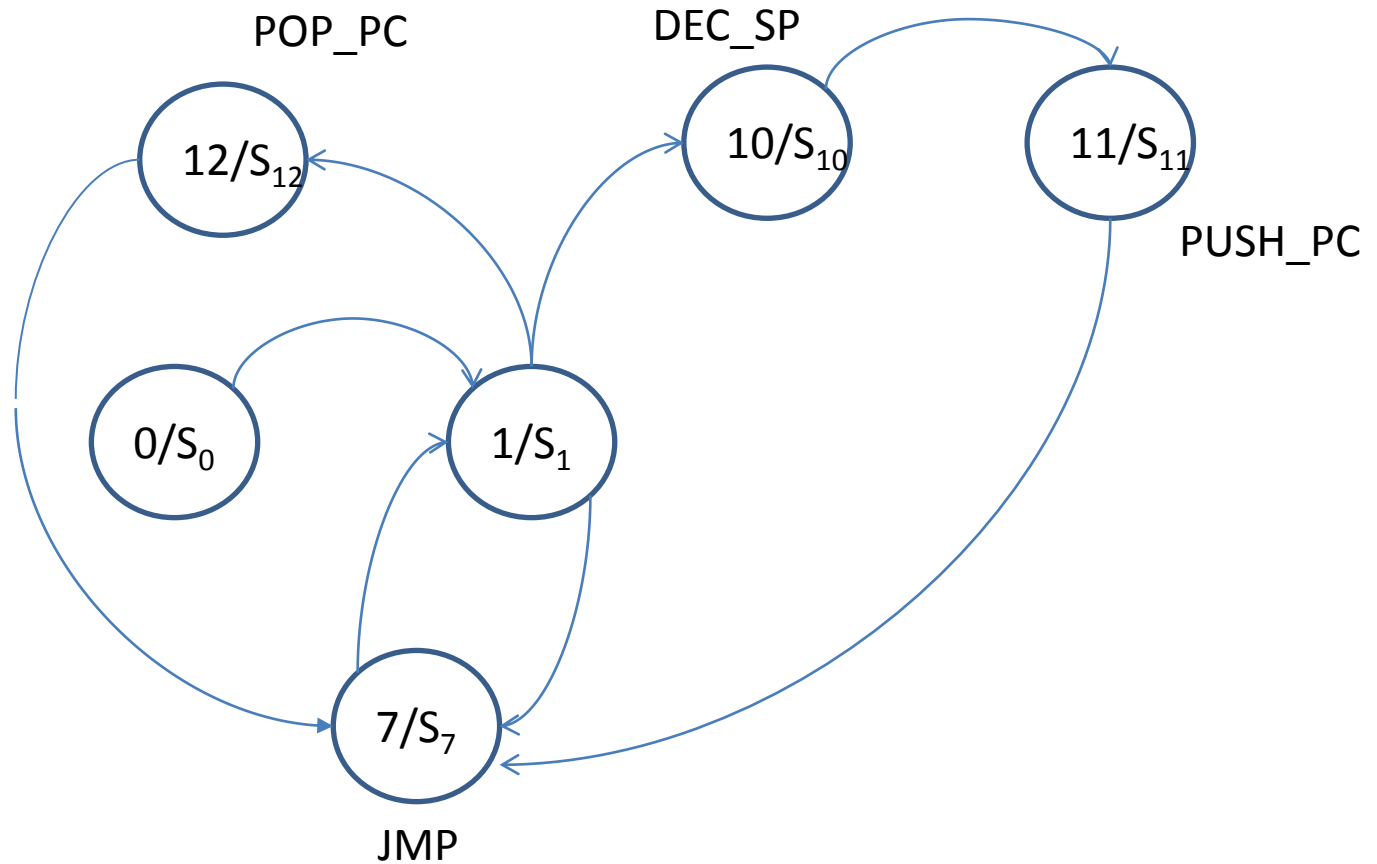




Cambios en la UC

- La instrucción CALL necesita 4 ciclos de ejecución: Fetch, Decode, Dec_SP, PUSH_PC, JMP
- La instrucción RET necesita 3 ciclos de ejecución: FETCH, Decode, POP y JMP (igual que en el caso del BEQ, se resta un ciclo de la instrucción siguiente)
- 3 nuevas señales de control hacia la UP:
 - sp_w: señal de carga del registro. Se activa en los estados Dec_SP y RET
 - dec_sp: resta cuando se activa. Se activa en el estado Des_SP
 - mux_in[1]: un bit mas de selección para el multiplexor. La entrada 2 la ocupa el PC.

Grafo de estados para CALL, RET y BEQ

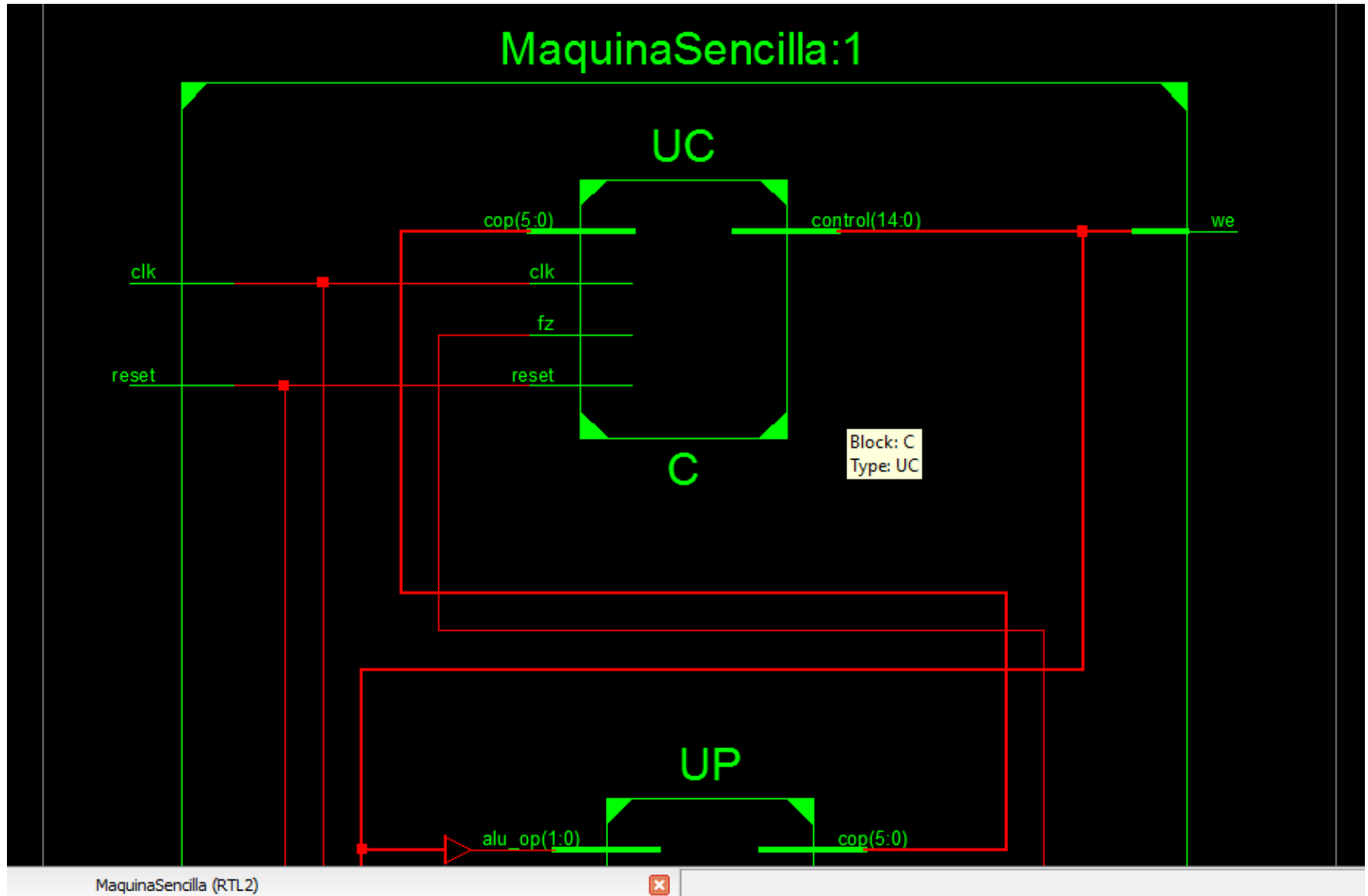


The diagram illustrates the internal structure of the **MaquinaSencilla:1**. It is composed of four main functional blocks: **UC** (Control Unit), **C** (Cache), **UP** (Processing Unit), and **P** (Peripheral).

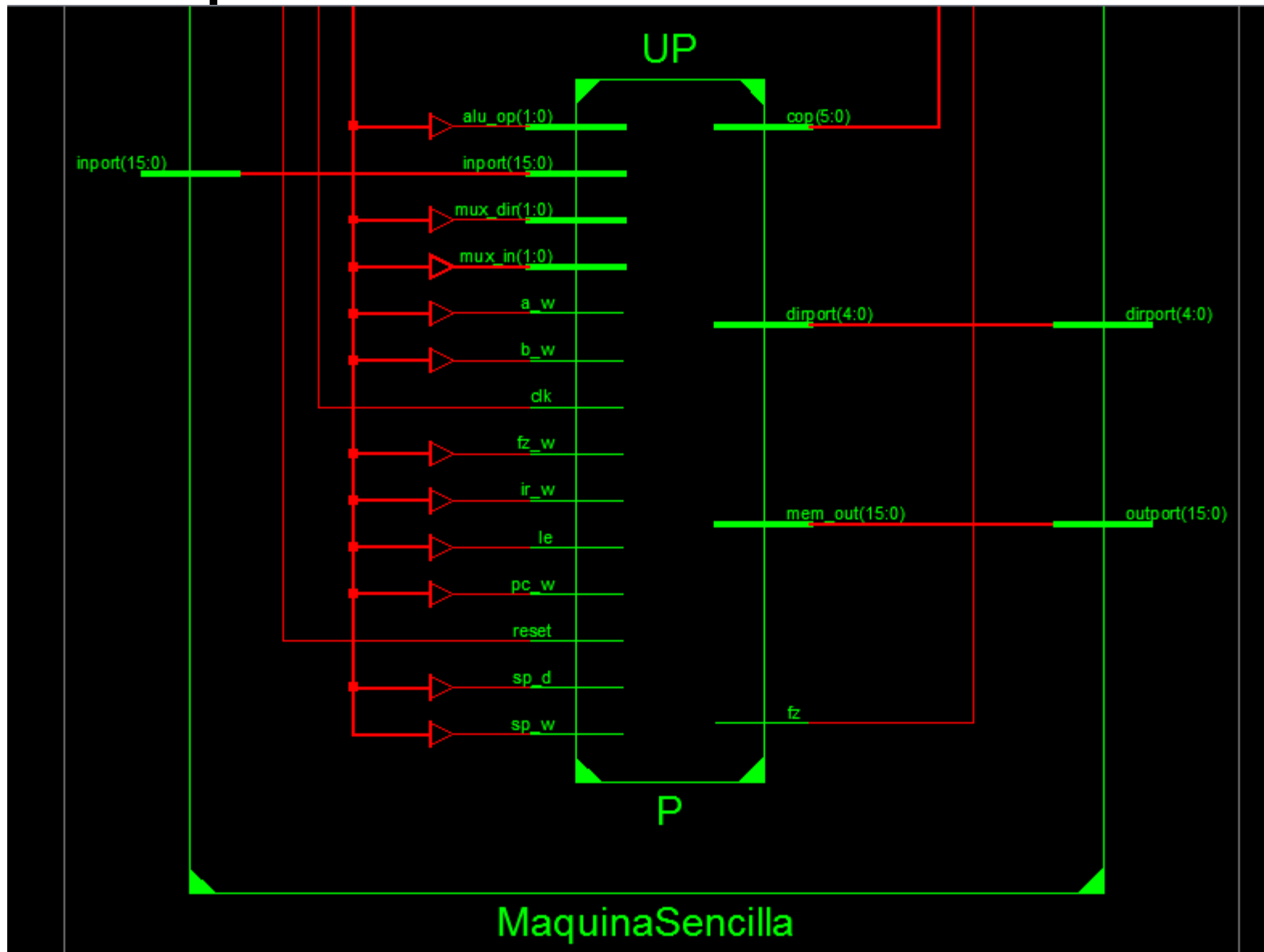
- UC (Control Unit):** Receives external **clk** and **reset** signals. It has two 15-bit data buses: **cpu_in(15.0)** and **cpu_out(15.0)**.
- C (Cache):** Connected to the **UC** and the **UP**. It has a 15-bit data bus **cpu_in(15.0)** and a 15-bit data bus **cpu_out(15.0)**.
- UP (Processing Unit):** Receives a 15-bit **input(15.0)** signal. It has multiple control inputs: **alu_op(1.0)**, **mul_div(1.0)**, **mul_sh(1.0)**, **ls_op**, **rs_op**, **fs_op**, **rs**, **pc_op**, **reset**, **sp_op**, and **sp_op**. It has two 4-bit data buses: **data_in(4.0)** and **data_out(4.0)**. It also has a 15-bit data bus **cpu_out(15.0)** and a 15-bit data bus **cpu_in(15.0)**.
- P (Peripheral):** Connected to the **UP** and the **UC**. It has a 15-bit data bus **cpu_in(15.0)** and a 15-bit data bus **cpu_out(15.0)**.

The diagram shows the interconnections between these components, including the flow of data and control signals. The **UC** and **UP** are connected to the **C** and **P** blocks. The **UP** block is connected to the **C** and **P** blocks. The **UC** block is connected to the **C** and **P** blocks. The **UP** block is connected to the **C** and **P** blocks.

Esquemático RTL , Detalle UC



Esquemático RTL: Detalle UP



Device Utilization

Device utilization summary:

Selected Device : 6slx16csg324-3

Slice Logic Utilization:

Number of Slice Registers:	77 out of 18224	0%
Number of Slice LUTs:	134 out of 9112	1%
Number used as Logic:	102 out of 9112	1%
Number used as Memory:	32 out of 2176	1%
Number used as RAM:	32	

Device Utilization

IO Utilization:

Number of IOs:	40		
Number of bonded IOBs:	40	out of	232 17%

Specific Feature Utilization:

Number of BUFG/BUFGCTRLs:	1	out of	16 6%
---------------------------	---	--------	-------

Timing

Asynchronous Control Signals Information:

No asynchronous control signals found in this design

Timing Summary:

Speed Grade: -3

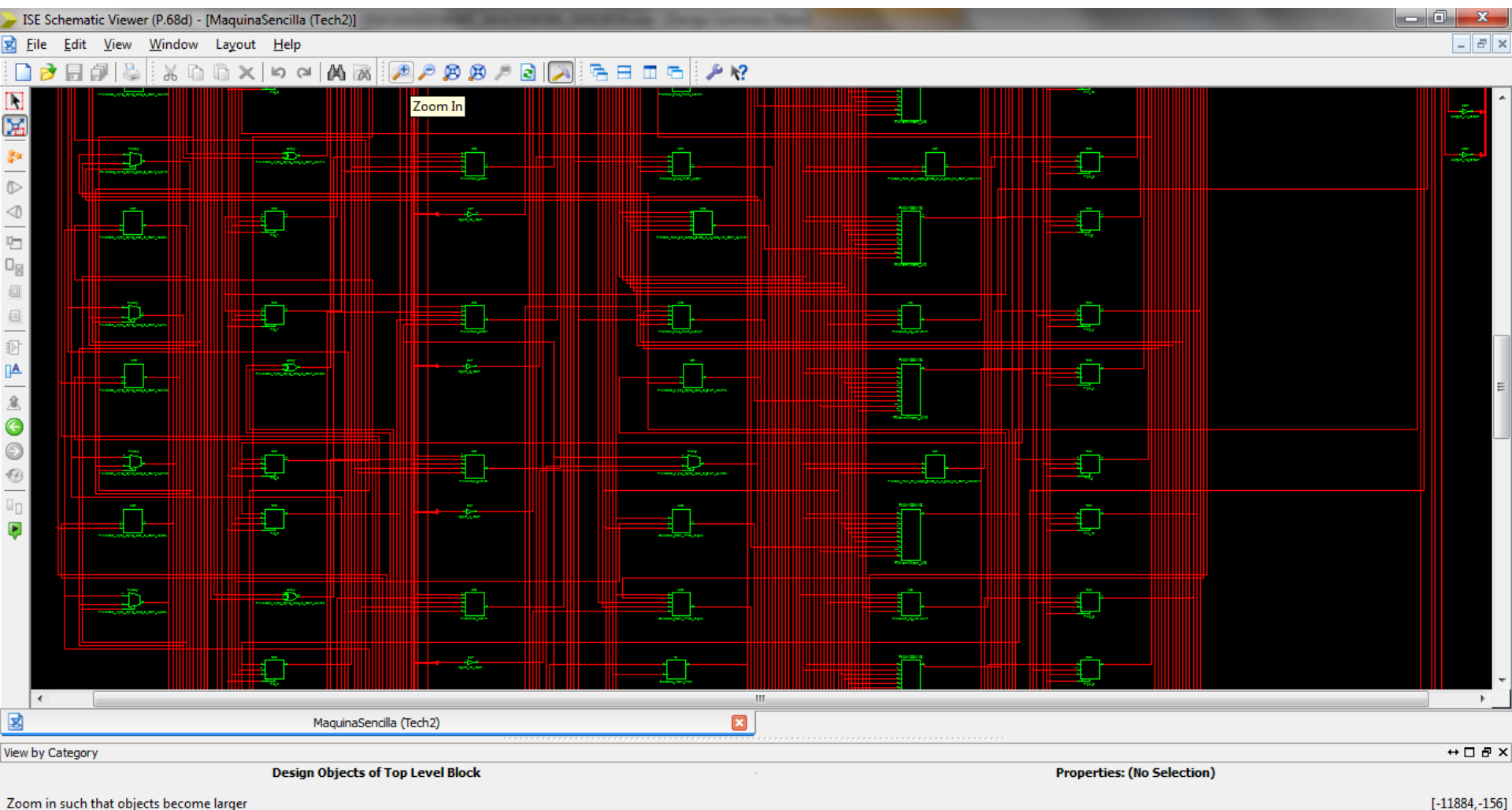
Minimum period: 6.124ns (Maximum Frequency: 163.285MHz)

Minimum input arrival time before clock: 3.590ns

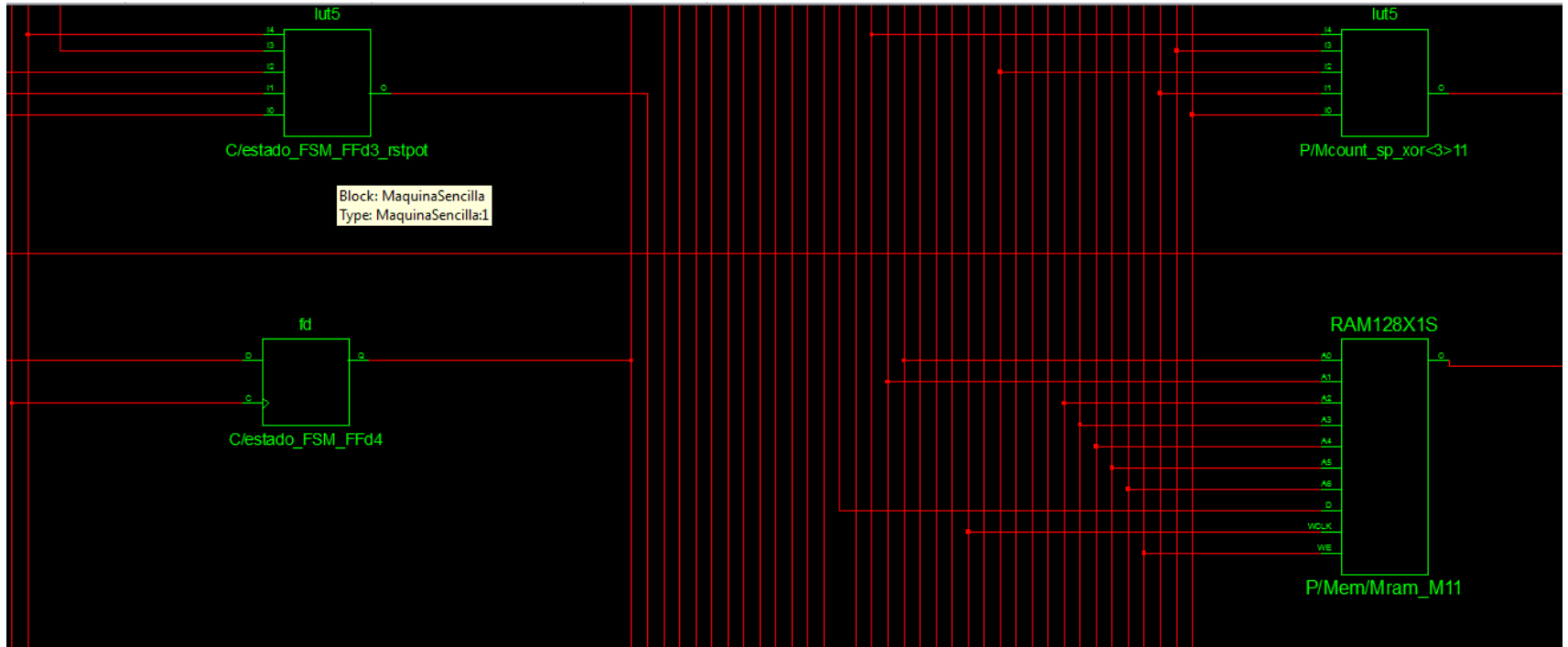
Maximum output required time after clock: 8.341ns

Maximum combinational path delay: No path found

Technological View



Technological View



Diseño de Máquina Sencilla

Segunda Parte

Segunda Parte

- Objetivo:
 - Definir una interface común para todos los dispositivos de entrada/salida
 - Implementar el decodificador de direcciones y ruteo de datos entre el procesador y los controladores
 - Utilizar la misma interface tanto para controladores de entrada/salida como para módulos en verilog

Interface hacia los Módulos

- CS : un bit para seleccionar el controlador (1 bit)
- Dir_reg: dos bits para identificar 4 registros del controlador (2 bits)
- We: escritura del registro especificado
- Datos: data_in, data_out(16 bits)

LógicoIO: Interface con MS

- Entradas desde la MS
 - **dev_sel: dirport[4:2]** identifica al dispositivo permitiendo solo 8 posibles dispositivos conectados simultáneamente. Con estos 3 bits se activa la señal CS (chip select) del dispositivo seleccionado.

Recordamos que **dirport** está especificada en la instrucción
OUTPUT dirport, A
siendo **dirport** las señales que se exportan a la interface de la MS y A la dirección de memoria del dato
 - **reg_sel: dirport[1:0]** identifica uno de los 4 registros disponibles del controlador. Se exporta al controlador del dispositivo directamente.
 - **data_out:** dato de 16 bits .

Es el dato que se envía al dispositivo seleccionado. Si el dispositivo es de 8 bits, el programa deberá hacerse cargo de la conversión.
 - **we:** señal de escritura

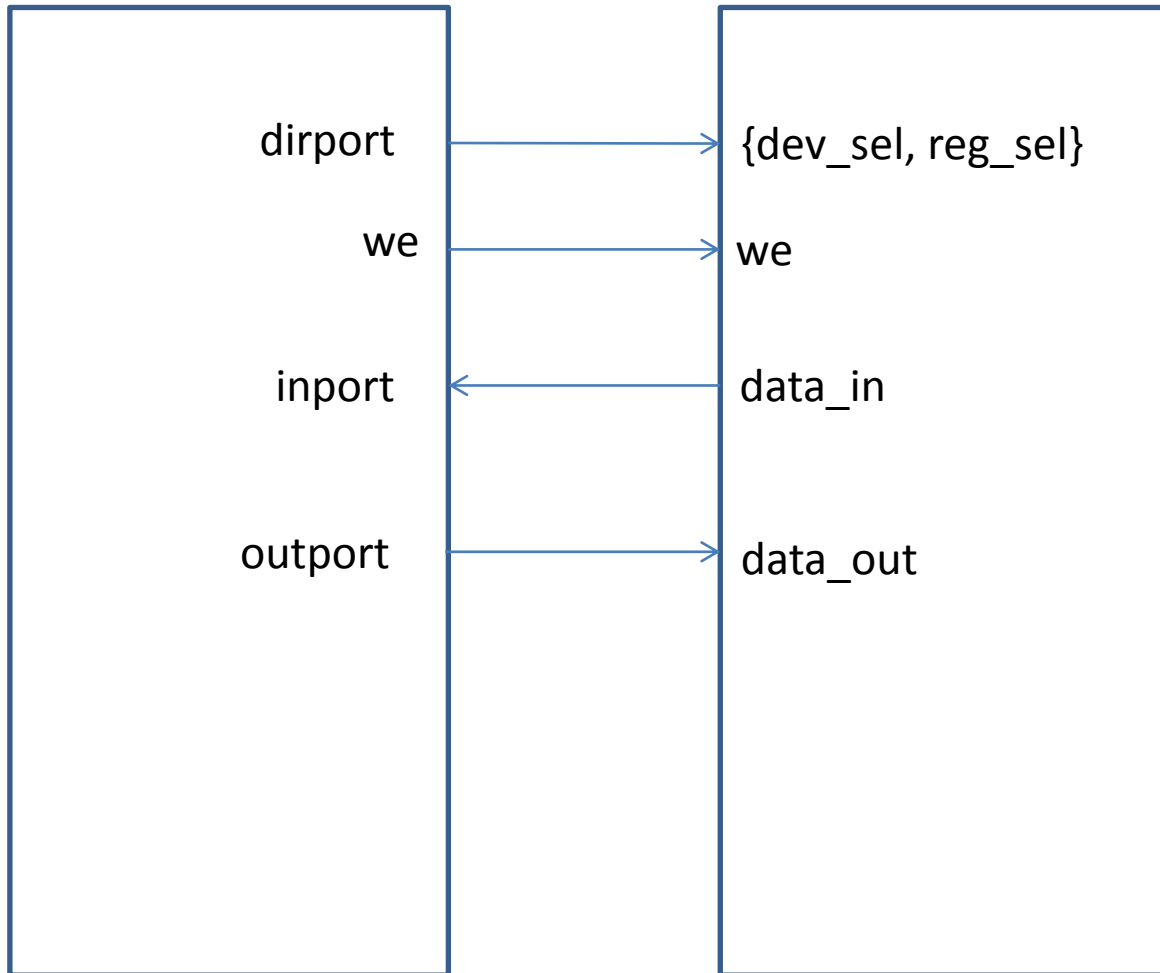
LógicoI/O: Interface con MS

- Salidas hacia la MS

- data_in: dato de 16 bits .

Es el dato que envía el dispositivo seleccionado. Si el dispositivo es de 8 bits, el programa recibirá ceros en los bits mas significativos.

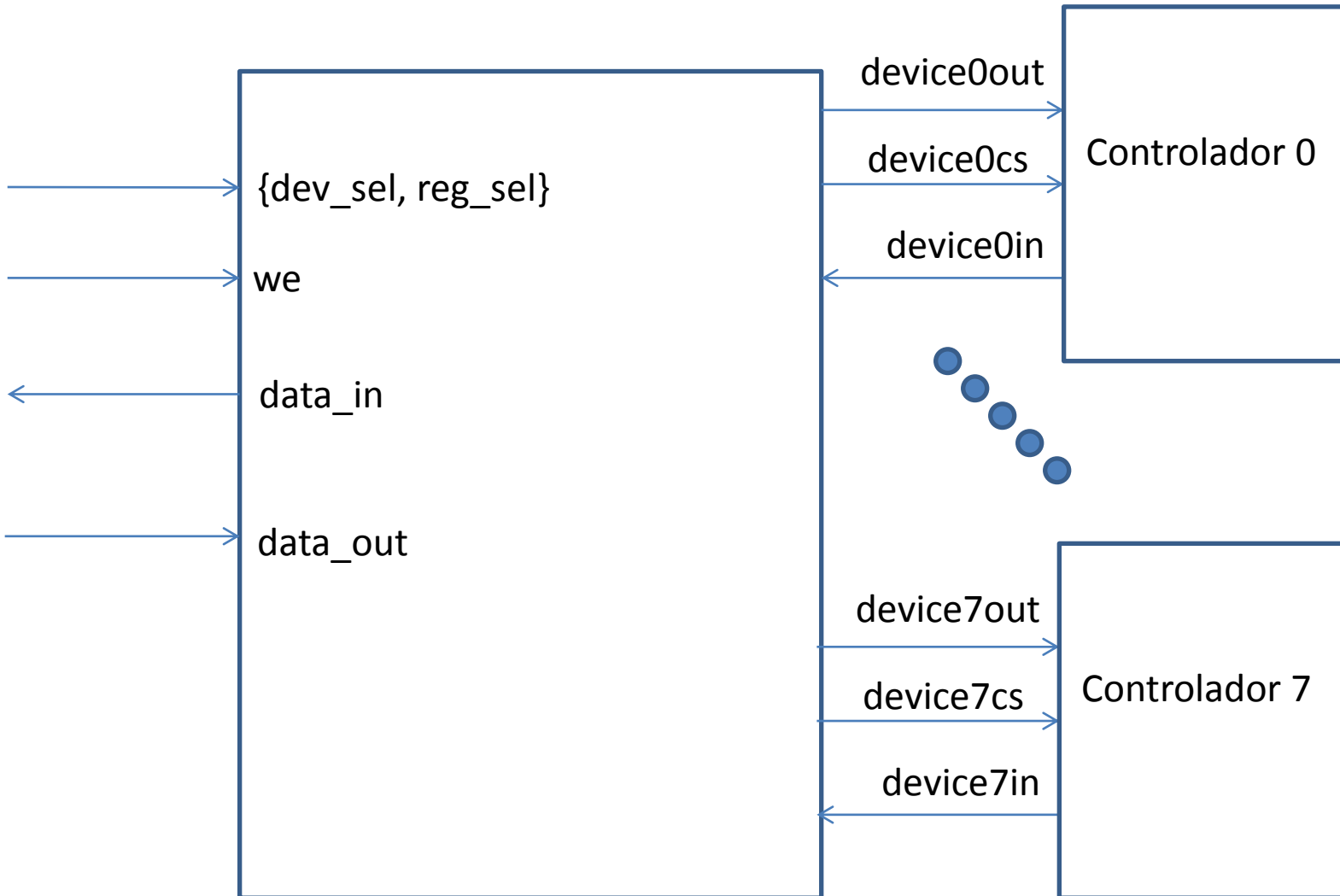
Interface MS-LógicoIO



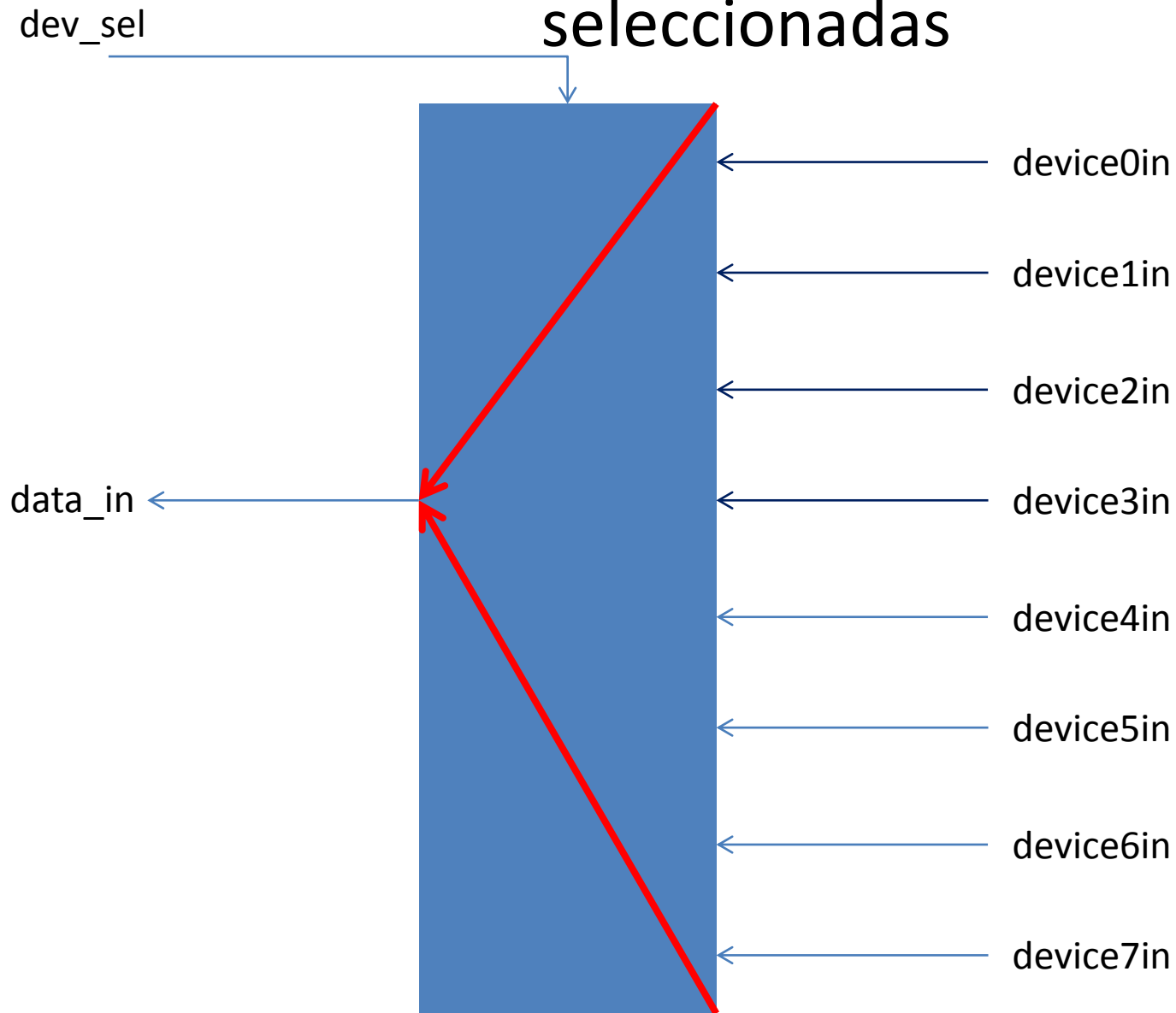
LógicoO: Interface con Controladores

- Permite hasta 8 controladores de dispositivos conectados
- Salidas al controlador:
 - deviceNout: donde N es un número del 0 al 7
Formado por 3 campos:
bit 18= we
bit 17:16= reg_sel
bits 15:0 = datos al controlador
 - deviceNcs: habilitación del dispositivo
- Entradas del controlador:
 - deviceNin [15:0] = datos del controlador

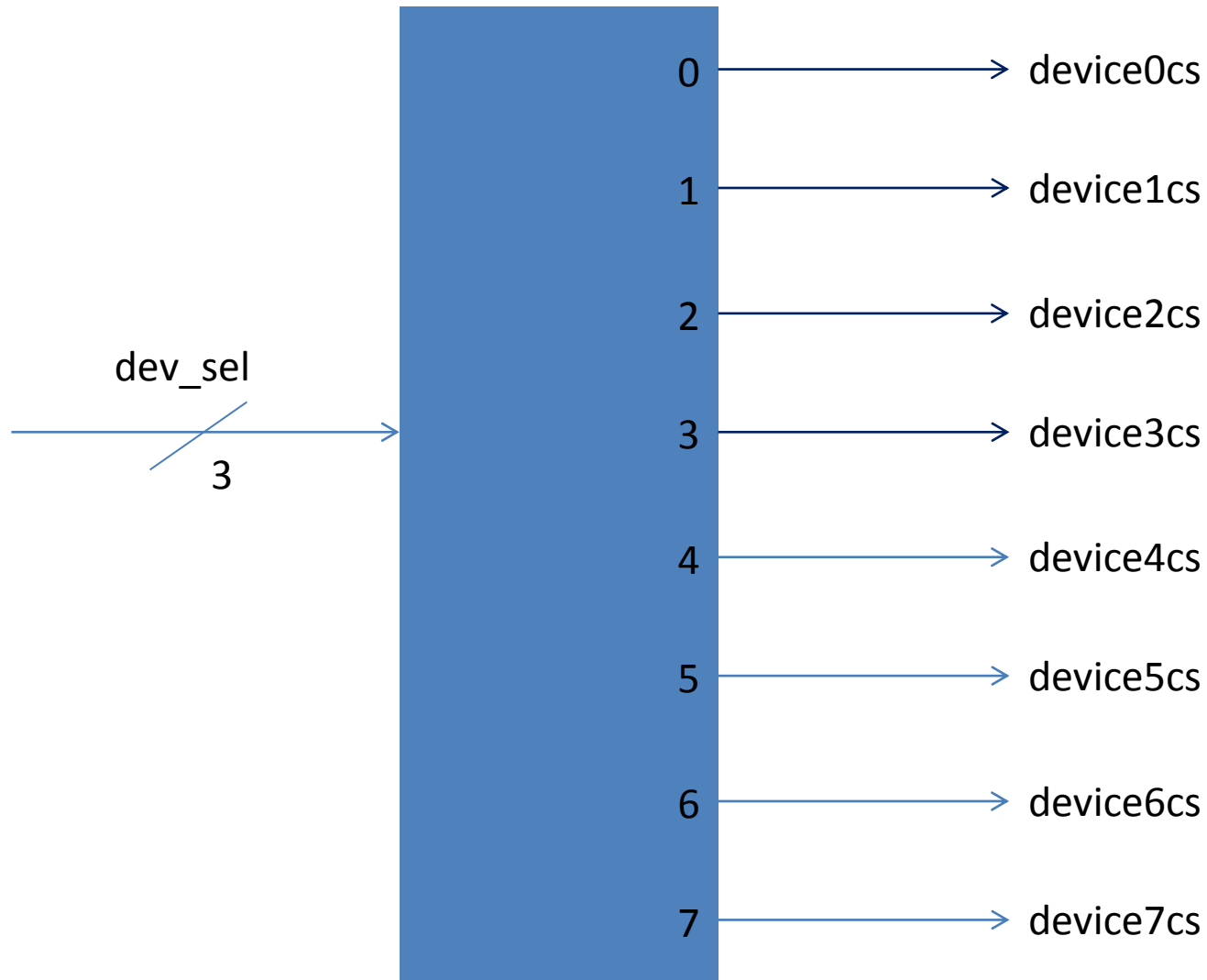
LógicoIO



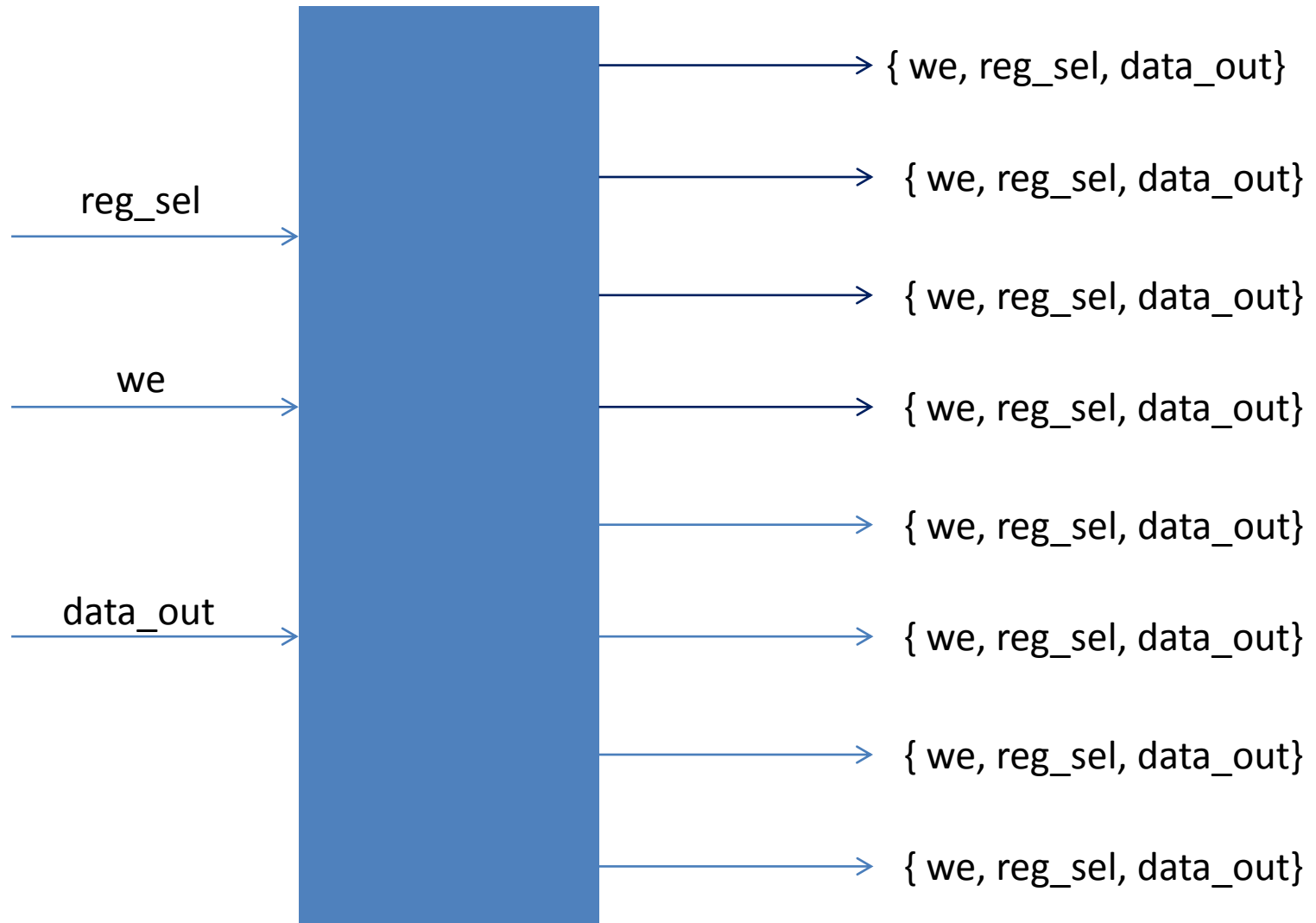
LógicaIO: multiplexación de las entradas seleccionadas

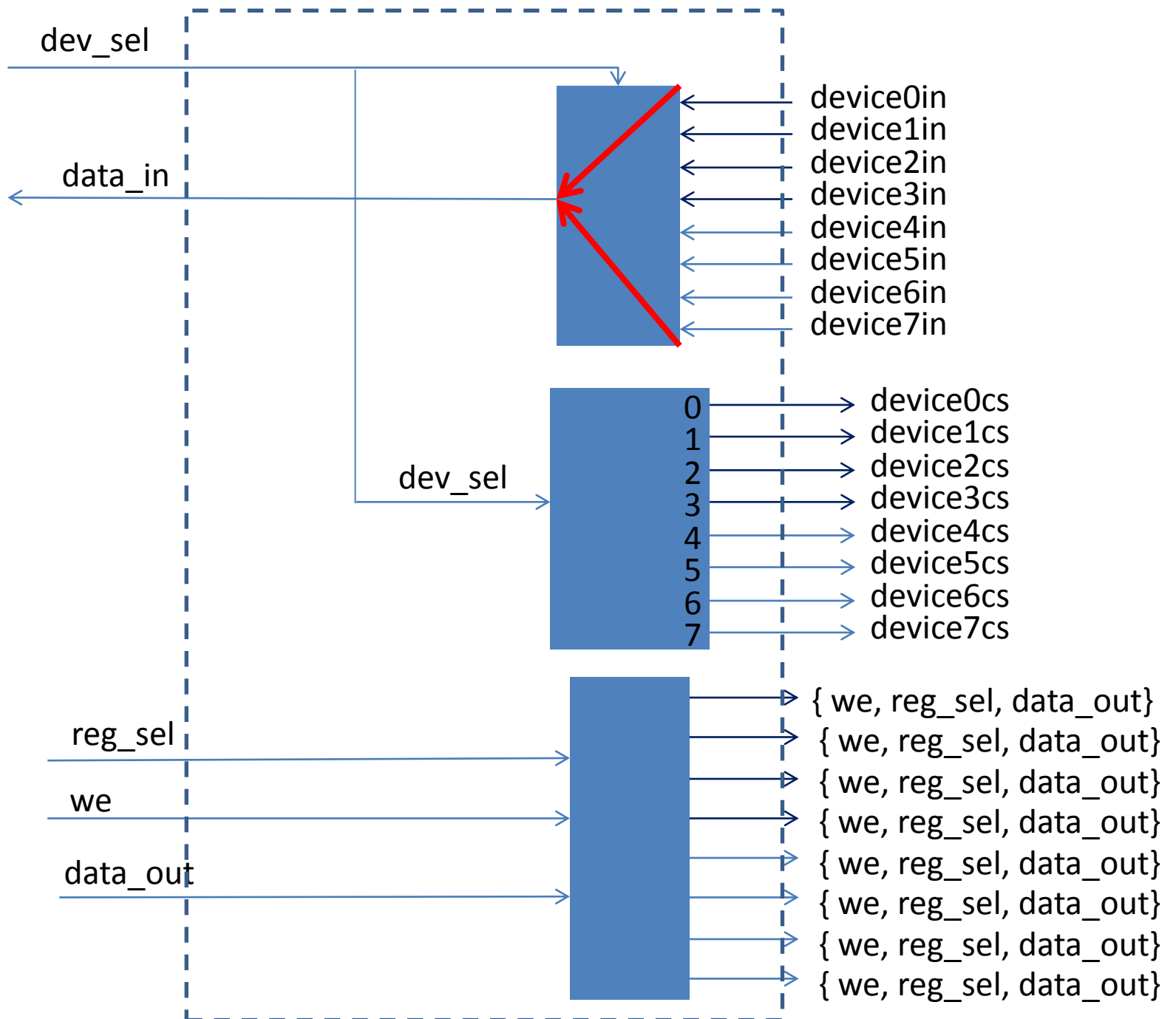


LógicaIO: generación del CS



LógicaIO: deviceNout





Controladores

- Se conectan a la LógicaIO
- Además de las señales especificadas , toman un reloj y un reset
- Para incluir un nuevo controlador, es necesario asignarle una dirección en el Mapa de direcciones

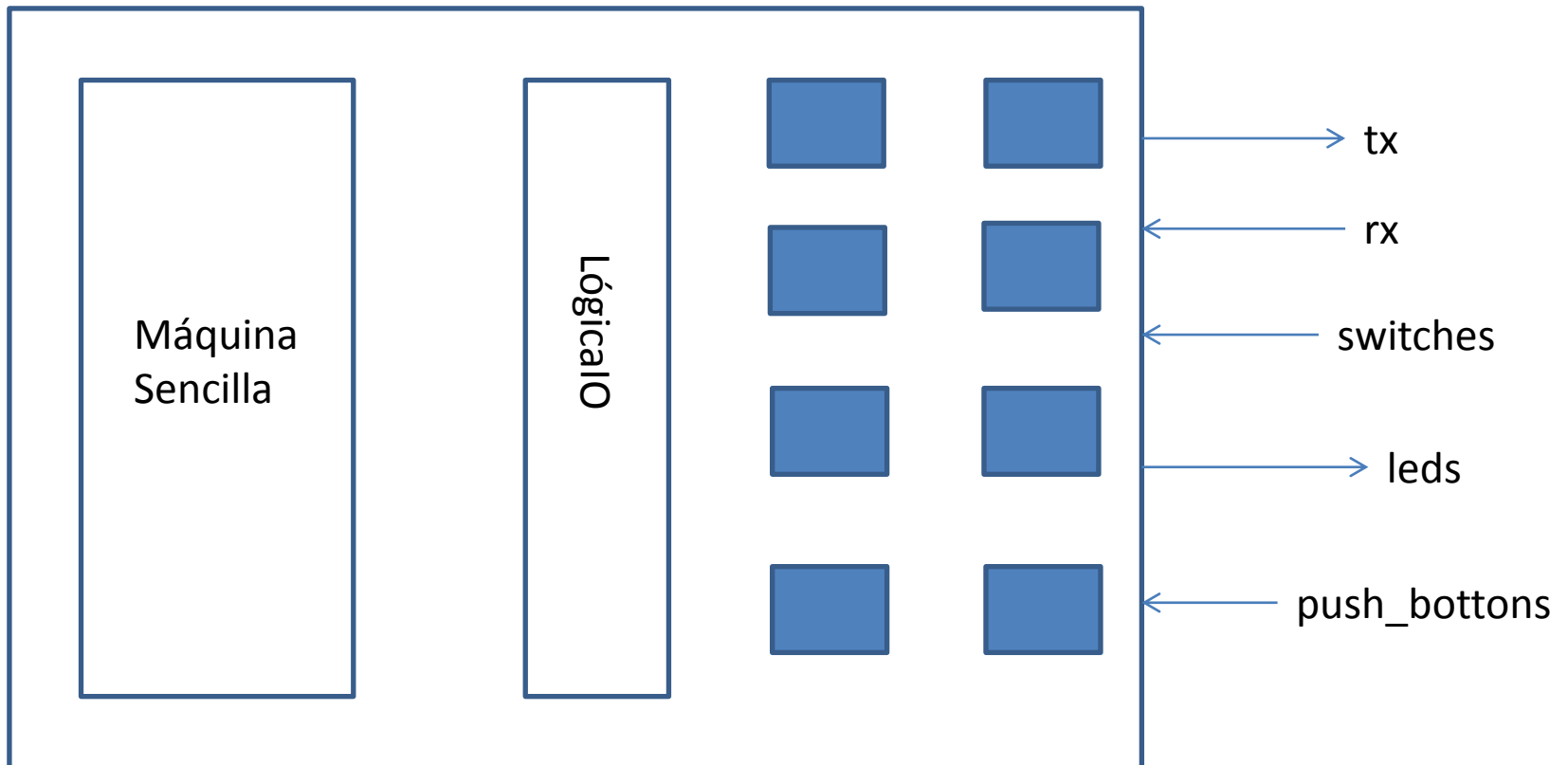
Mapa de Direcciones de Dispositivos

Las direcciones de entrada/salida son de 5 bits de ancho y tienen la forma {d[2:0], r[1:0]}, donde los 3 bits más significativos identifican un dispositivo y los 2 menos significativos seleccionan un registro dentro de él.

Dirección	Dispositivo	Reg. 0	Reg. 1	Reg. 2	Reg. 3	Observaciones
0	ALM	d0(w)	d1(w)	op(w)	out(r)	Ver Docs/ALM.md
1	Shifter	op(w)	value(w)	data_out(r)		op = {{11{x}}, 1'b(-l/r), 4'b(#shifts)}
2	Timer	rounds(r) rounds_goal(w)	prescaler(r) prescaler_goal(w)	status(r) start(w)	done(r) reset(w)	Ver Docs/timer-calc.html para el cálculo de tiempos
3	UART	tx_data(w)	rx_data(r) reset(w)	tx_full(r)	rx_empty(r)	Opera con los 8 bits <i>menos</i> significativos
4	7 Segmentos	7seg_data(w)				
5	Botones	btnU(r)	btnD(r)	btnL(r)	btnR(r)	Expande con ceros
6	Switches	swt_data(r)				
7	LEDs	led_data(w)				

Placa

- La interface con la placa es responsabilidad del módulo placa.



Placa

- El módulo Placa instancia:
 - La máquina sencilla
 - La LógicaIO
 - Los controladores
- Y, exporta las señales que deben salir al exterior de la FPGA
- Actualmente, la placa exporta las siguientes señales (en el UCF)
 - Leds
 - Switches
 - Push Bottons
 - Reset
 - Tx, Rx
 - Siete Segmentos

Placa: Interface

```
module Placa(  
    input clk,  
    input reset,  
    input [7:0] sw,  
    input [3:0] btn,  
    output [7:0] Led, sseg,  
    output [3:0] an,  
    input rx,  
    output tx  
);
```

Placa: Instanciación de Módulos

MaquinaSencilla MS

```
(.clk(clk), .reset(reset), .inport(inport), .dirport(dirport), .outport(outport), .we(we));
```

```
LogicalO L(.dev_sel(dirport[4:2]), .reg_sel(dirport[1:0]), .we(we), .data_out(outport), .data_in(inport),  
  .device0in(device0out), .device0out(device0in), .device0cs(device0cs),  
  .device1in(device1out), .device1out(device1in), .device1cs(device1cs),  
  .device2in(device2out), .device2out(device2in), .device2cs(device2cs),  
  .device3in(device3out), .device3out(device3in), .device3cs(device3cs),  
  .device4in(device4out), .device4out(device4in), .device4cs(device4cs),  
  .device5in(device5out), .device5out(device5in), .device5cs(device5cs),  
  .device6in(device6out), .device6out(device6in), .device6cs(device6cs),  
  .device7in(device7out), .device7out(device7in), .device7cs(device7cs)  
  );
```

Placa: Instanciación de Módulos

```
// DEVICE 7: LEDS
```

```
    ControladorLED CntrlLed(.clk(clk), .reset(reset), .we(device7in[18]), .reg_sel(device7in[17:16]), .cs(device7cs),  
.in(device7in[15:0]), .out(device7out));
```

```
// DEVICE 6: SWITCHES
```

```
    ControladorSwitches CntrlSwt(.clk(clk), .reset(reset), .we(device6in[18]), .reg_sel(device6in[17:16]), .cs(device6cs),  
.in({8'b0, sw}), .out(device6out));
```

```
// DEVICE 5 : BOTONES
```

```
    ControladorBotones CntrlBtn(.clk(clk), .reset(reset), .we(device5in[18]), .reg_sel(device5in[17:16]), .cs(device5cs),  
.in({12'b0, btn}), .out(device5out));
```

```
// DEVICE 4: SEVEN SEGMENTS
```

```
    ControladorSSEG CntrlSSEG(.clk(clk), .reset(reset), .we(device4in[18]), .reg_sel(device4in[17:16]), .cs(device4cs),  
.in(device4in[15:0]), .out(device4out));
```

```
// DEVICE 3: UART
```

```
    ControladorUART CntrlUART(.clk(clk), .reset(reset), .we(device3in[18]), .reg_sel({device3in[17:16]}), .cs(device3cs),  
.in({8'b0, device3in[7:0]}), .out(device3out), .tx(tx), .rx(rx));
```

```
// DEVICE 2:
```

```
    Timer TimerModule(.clk(clk), .reset(reset), .we(device2in[18]), .reg_sel({device2in[17:16]}), .cs(device2cs),  
.in({device2in[15:0]}), .out(device2out));
```

```
// DEVICE 1: -----
```

```
// DEVICE 0: -----
```

Para crear un nuevo dispositivo

1. Se deberá crear el controlador con la siguiente interface:

```
Controlador_Nuevo (  
    input clk, reset, we, cs,  
    input [1:0] reg_sel,  
    input [15:0] in,  
    output [15:0] out )
```

2. Se deberá asignar una dirección al controlador (si fuere necesario , se re-utilizará una dirección de alguno de los controladores por defecto no utilizados)
3. Se deberá incluir la instanciación del controlador en el archivo placa.v
4. Los accesos desde el programa asm deben realizarse utilizando el número de dispositivo asignado.
5. En caso que el dispositivo se comuniqué con la placa , (por ejemplo, VGA port, memoria externa, ethernet, etc) se deberá incluir las entradas y salidas en la interface del módulo placa, y se deberán incluir en el UCF del proyecto las señales correspondientes.
6. **IMPORTANTE: El controlador deberá registrar sus entradas y/o salidas.**

Tercera Parte: Ejemplos de Dispositivos y Programas de Uso

Ejemplo: UART

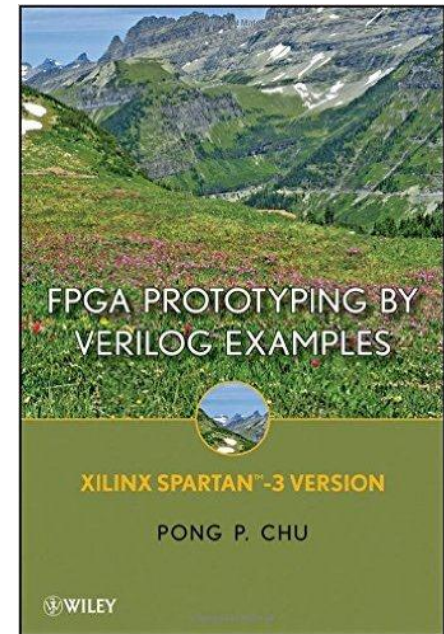
```
ControladorUART CntrlUART  
(  
  .we(device3in[18]),  
  .reg_sel({device3in[17:16]}),  
  .in(device3in[7:0]),  
  .out(device3out[7:0]),  
  .cs(device3cs), .clk(clk),  
  .reset(reset)  
);
```

rx se encuentra en device3out[0]
tx se encuentra en device3in[0]

- La uart está mapeada en la dirección 3
- Tiene 4 registros accesibles al programador:
 - 00: registro de transmisión
 - 01: registro de recepción
 - 10: flag de cola de transmisión llena
 - 11: flag de cola de recepción vacía
- Para usar la UART es necesario vaciar las colas, esto se hace escribiendo en la dirección 01.

Controlador UART

- El módulo UART se adaptó del incluido en el libro de Pong P. CHU.
- Las principales modificaciones realizadas fueron:
 - Se adaptó el módulo generador del baud rate para funcionar con el reloj de nexys3 100Mhz
 - Se reemplazó la cola de entrada y de salida (FIFOs) por los IP cores de Xilinx
 - Se modificó la señal de reset de forma de poder ser reseteado (en decir, vaciadas sus colas) desde el software. Esto se realiza escribiendo en un registro que es de solo lectura.
 - La uart esta configurada a : 19200 baudios, 1 bit stop, 8 bits data, flow control:none

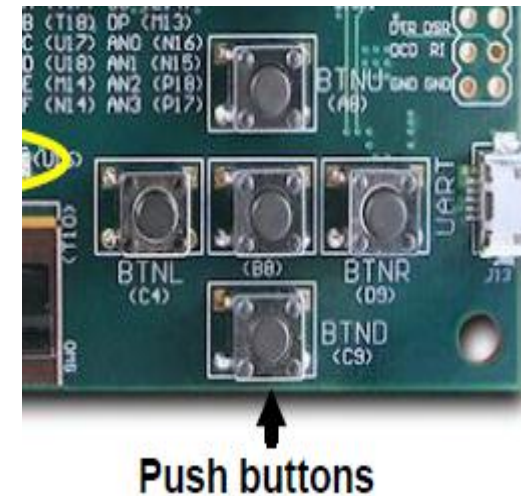


Programa: ejemplo de uso de la UART

```
OUT 5, @0 ;; inicializar la UART
Polling1:IN 7, a ;; polling sobre el registro de cola vacía de recepción
CMP a, @1 ;;
BEQ Polling1 ;; si está vacía seguir esperando
IN 5, b;; leer dato de recepción
ADD @1, b;; sumarle uno
OUT 0, b;; enviarlo a los leds
OUT 4, b;; transmitirlo
CMP a, a ;; forzar salto
BEQ Loop1;; volver a empezar
```

Controlador de Botones

- Provee una interface para los cuatro botones de nexys3.
- El central se reserva para reset
- Cada botón está asociado a una dirección de registro, tal como está especificado en el Mapeo de Direcciones por Defecto.

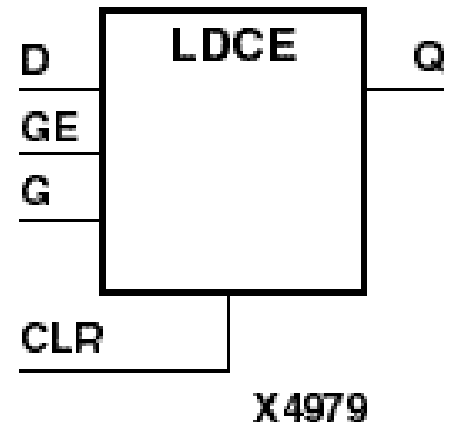


Controlador de Botones: Uso

- Para poder leer un botón, es decir, para saber si se ha apretado el botón, se debe hacer polling sobre la dirección del registro correspondiente.
- Una vez leído, se debe forzar el valor del botón a cero para evitar que la misma señal vuelva a ser leída.
- El controlador de botón es útil para sincronizar entradas, por ejemplo de los switches.

Controlador del Botón: implementación

- Dado que los botones son dispositivos mecánicos, al apretarlos se genera un rebote que es necesario filtrar para evitar que la señal se interprete como múltiples eventos. Esto se realiza con un módulo anti-rebote, también obtenido del libro de Pong Chu, referenciado anteriormente.
- Se utilizó un Latch LDCE (Data Latch with Asynchronous Clear and Gate Enable) :
 - La señal filtrada, en el caso de que el programa esté esperando el evento (que el dispositivo esté seleccionado) se utiliza como señal de Gate Input (G) del LDCE
 - La señal CLR, se activa cuando el programa escribe en el dispositivo, poniendo el latch a cero
 - La señal de Dato (D) es siempre 1, y se refleja en la salida Q solo cuando el dispositivo está seleccionado (señal Gate Enable) y la señal Gate Input pasa de 1 a 0 (indicando que se ha soltado el botón)



Controlador de Botón: código verilog

```
module ControladorBoton(  
    input clk, reset,  
    input btn_in,  
    input cs, we,  
    output out  
);  
  
    wire tick, db_level, G, clear;  
  
    assign clear=(we)& cs;  
    assign G= (~we)& (~db_level)& tick;  
  
    debounce db (.clk(clk), .reset(reset),.sw(btn_in),.db_level(db_level),.db_tick(tick));  
  
    LDCE LDCE_inst (  
        .Q(out),    // Data output  
        .CLR(clear), // Asynchronous clear/reset input  
        .D(1'b1),   // Data input  
        .G(G),       // Gate input  
        .GE(cs)      // Gate enable input  
    );
```

Controlador de Botones: código

```
module ControladorBotones(  
    input clk, reset, we, cs,  
    input [1:0] reg_sel,  
    input [15:0] in,  
    output reg [15:0] out  
);  
  
    assign btn_3 = in[3];  
    assign btn_2 = in[2];  
    assign btn_1 = in[1];  
    assign btn_0 = in[0];  
  
    ControladorBoton2 b3 (.clk(clk), .reset(reset), .cs(cs), .we(we), .btn_in(btn_3), .out(btn3_out));  
    ControladorBoton2 b2 (.clk(clk), .reset(reset), .cs(cs), .we(we), .btn_in(btn_2), .out(btn2_out));  
    ControladorBoton2 b1 (.clk(clk), .reset(reset), .cs(cs), .we(we), .btn_in(btn_1), .out(btn1_out));  
    ControladorBoton2 b0 (.clk(clk), .reset(reset), .cs(cs), .we(we), .btn_in(btn_0), .out(btn0_out));  
  
    always@(posedge clk)  
        if (cs & !we)  
            case (reg_sel)  
                2'b00: out <= {15'b0, btn0_out};  
                2'b01: out <= {15'b0, btn1_out};  
                2'b10: out <= {15'b0, btn2_out};  
                2'b11: out <= {15'b0, btn3_out};  
            endcase  
        else  
            out <= 16'd0;  
    endmodule
```

Controlador de Botones, programa ejemplo

Este programa demuestra el correcto funcionamiento del botón que no pierde ni repite eventos

```
mov @1,I ; contador de iteraciones
polling1: IN 20, a; polling del boton Norte
CMP a, @0
BEQ polling1
IN 24, b; entrada de switches
OUT 28, b; salida a los leds
OUT 20, @1 ; poner a cero el flag antes de volver a leer
polling2: IN 20, b; siguiente polling a boton Norte
CMP b, @0
BEQ polling2
OUT 20, b; poner a 0 antes de volver a leer
OUT 28, @7; mandar una constante a los leds
INC I;
OUT 16, I; sacar el contador de iteraciones a los 7-segmentos
JMP polling1
```


Controlador : Timer

- El timer tiene las siguientes funcionalidades:
 - Permite esperar una cantidad de tiempo
 - Permite contar el tiempo transcurrido de un grupo de instrucciones de un programa
- Formado por cuatro registros:
 - ROUNDS_reg: especifica un número entre 0 y 65536.
 - Se escribe un valor para esperar tiempo. En la funcionalidad contar tiempo, se puede leer.
 - PRESCALER_reg: especifica un número entre 0 y 65536
 - Se lo programa para establecer hasta cuanto hay que contar para lograr un round. También se puede leer.
 - START_reg:
 - Se debe poner un 1 para iniciar el conteo en las dos funcionalidades
 - DONE_reg:
 - Se consulta en el primera modalidad para saber si ha terminado la cuenta.

Controlador Timer: Ejemplo de espera

- Para esperar un segundo, podemos programar los registros de la siguiente manera:
 - Si la frecuencia del reloj es de 100 Mhz, un ciclo de reloj dura 10nseg.
 - Un segundo son 10^9 nanosegundos, o sea debemos esperar 10^8 ciclos
 - Usando la calculadora de tiempos del timer (timer-calc.html) los valores mas aproximados a 1 segundo se obtienen haciendo:
 - Round= 1526
 - Prescaler=65530

Controlador Timer: Ejemplo de espera de un segundo

- Programa que actualiza un contador que se visualiza en los siete segmentos cada segundo
- Los valores son aproximados en función de :
 - La precisión del timer (+/- 3000 nseg)
 - La cantidad de tiempo que se utilizan para ejecutar las instrucciones de puesta en marcha (6) y polling (3). Si contamos las 3 últimas instrucciones para actualizar el contador, obtenemos un total de 12 instrucciones.
- Cada instrucción tarda en media 5 ciclos: 50 nseg. Es decir que podemos considerar que la ejecución de las instrucciones agrega aprox. 500 nseg.
- Observación: podemos, si queremos hacer las cuentas exactas

```
MOV @1,a
OUT 16, a
iterar: MOV @1526, rounds
MOV @65535, prescaler
OUT 8, rounds
OUT 9, prescaler
MOV @1, start
OUT 10, start
polling: IN 11, done
CMP @0,done
BEQ polling
INC a
OUT 16,a
JMP iterar
```

Controlador de Timer: contar tiempo de ejecución

- Segunda funcionalidad del timer: contar tiempo de ejecución
- Puede ser interesante usarlo cuando la subrutina incluye una espera activa (polling)

```
MOV @0,i
OUT 16,i
; inicializa contador
MOV @65535, rounds
MOV @65535, prescaler
OUT 8, rounds
OUT 9, prescaler
MOV @1, start
OUT 10, start
tardar: ADD @1, i
CMP @100, i
BEQ medir
JMP tardar
medir:
OUT 10, @0 ;; detener el contador
IN 8, rounds
IN 9, prescaler
OUT 16, prescaler
OUT 28, rounds
final: JMP final
```

Timer

- El bucle *tardar* se ejecuta 99 veces con BEQ no tomado y una vez con BEQ tomado.
- Una iteración con BEQ no tomado tarda 19 ciclos según conteo de instrucciones siguiente:
 - ADD 5ciclos
 - CMP 5ciclos
 - BEQ no tomado 2 ciclos
 - JMP (CMP y BEQ) 7 ciclos
- Una iteración con BEQ tomado tarda 12 ciclos
 - ADD 5 ciclos
 - CMP 5 ciclos
 - BEQ tomado 2 ciclos
- Total de ciclos = $99 \cdot 19 + 12 = 1893$
- Total de ciclos medidos = 1896
- Conclusión: es necesario restar 3 ciclos que corresponden a los ciclos necesarios para detener el conteo (OUT 10, @0)

```
tardar:  
ADD @1, i  
CMP @100, i  
BEQ medir  
JMP tardar  
medir:
```

Módulo Shifter

- Dado que la MS no soporta repertorio de instrucciones extenso (solo ¡4!) todas las operaciones complejas tales como multiplicaciones, divisiones, e incluso restas deben realizarse por hardware
- El módulo Shifter es un ejemplo. Como cualquier módulo controlador tiene una dirección asignada y 4 registros visibles al programador.

Módulo Shifter: Programación

- Registros:
 - Registro 0 (w): Dato
 - Registro 1(w): Cantidad de Desplazamiento
 - Registro 2(r): Resultado shifteado izquierda
 - Registro 3(r): Resultado shifteado derecha
- Dado que la implementación es combinacional, la inmediata lectura del resultado es posible.

Módulo Shifter: Programación

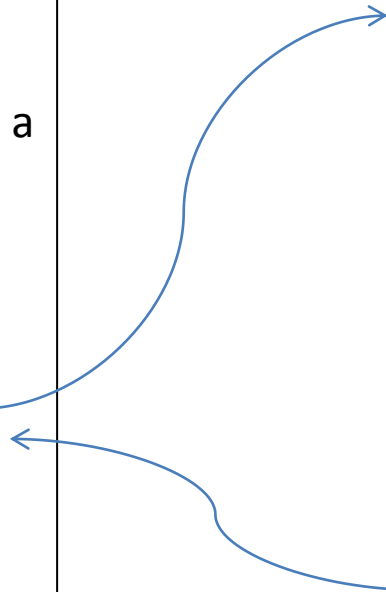
- Usa el timer para esperar un tiempo
- El shifter está en la dirección 1

```
MOV @1, i
OUT 28, i
MOV @1, k
iterar: MOV k, a
OUT 5, a ;; times, lo vario de 1 a 15
OUT 4, i ;; dato que es siempre 1
IN 6, j ;; obtengo dato shift left
OUT 16, j
MOV @1000, rounds
MOV @65535, prescaler
OUT 8, rounds
OUT 9, prescaler
MOV @1, start
OUT 10, start
polling: IN 11, done
CMP @0, done
BEQ polling
ADD @1, k
CMP @16, k
BEQ final
JMP iterar
final: JMP final
```


Módulo Shifter con Subrutina

```
MOV @1, i
OUT 28, i
MOV @1, k
iterar: MOV k, a
OUT 5, a
OUT 4, i
IN 6, j
OUT 16, j
CALL esperar
ADD @1, k
CMP @16, k
BEQ final
JMP iterar
final: JMP final
```

```
esperar:
MOV @1000, rounds
MOV @65535, prescaler
OUT 8, rounds
OUT 9, prescaler
MOV @1, start
OUT 10, start
polling: IN 11, done
CMP @0, done
BEQ polling
ret
```



ALM: Módulo Aritmético Lógico

- Implementa distintas operaciones que trabajan sobre dos operandos de 16 bits.
- Idea: el procesador permanece sencillo , sólo se instancia la unidad externa ALM cuando es necesario.
- Exceptuando el divisor y multiplicador todas las operaciones están implementadas con lógica combinacional, por lo que los resultados son inmediatos.
- Interface: Cuatro registros de 16 bits:
 - 0: D0 (w) operando fuente 0
 - 1: D1 (w) operando fuente 1
 - 2: OP (w) operación
 - 3: OUT (r) resultado
- Los registros de escritura mantienen sus valores hasta ser explícitamente sobrescritos.

ALM: operaciones bit a bit

Código (OP)	Descripción	Resultado (OUT)
0	AND	$D0 \& D1$
1	OR	$D0 \mid D1$
2	NOT	$\sim D0$
3	NAND	$\sim(D0 \& D1)$
4	XOR	$D0 \wedge D1$
5	NOR	$\sim(D0 \mid D1)$
6	XNOR	$\sim(D0 \wedge D1)$
7	BITSEL	$D0[D1]$
8	MERGE_H	$\{D1[15:8], D0[15:8]\}$
9	MERGE_L	$\{D1[7:0], D0[7:0]\}$
10	MERGE_HL	$\{D1[15:8], D0[7:0]\}$
11	MERGE_LH	$\{D1[7:0], D0[15:8]\}$
12	SWAP_HL	$\{D0[7:0], D0[15:8]\}$

ALM: operaciones aritméticas

Código (OP)	Descripción	Resultado (OUT)
13	MULT	$D0[7:0] * D1[7:0]$
14	DIV	$D0 / D1$
15	DIV_REM	$D0 \% D1$

ALM: Multiplicador

Multiplier

Documents View

Resource Estimates

Resource Estimates

LUT6s	72
XtremeDSP slices	0
BRAMS	0

Additional Information

Please note that the LUT resource estimate does not include SRLs.

Resource counts may not reflect true post-map resource usage when a custom output width is used and the output product MSB is less than full-precision MSB

Multiplier

xilinx.com:ip:mult_gen:11.2

Output Product Range

Output product width (max, min) = (15, 0)

Use Custom Output Width

☐ Use Custom Output Width

Output MSB Range: 0..127

Output LSB Range: 0..15

Output Rounding

☐ Use Symmetric Rounding

Pipelining and Control Signals

Pipeline Stages Optimum pipeline stages: 3

☐ Clock Enable

☐ Synchronous Clear

SCLR/CE Priority

☒ SCLR overrides CE

☐ CE overrides SCLR

IP Symbol Resource Estimates

Datasheet

< Back Page 3 of 3 Next > Generate Cancel Help

ALM: Ejemplo de multiplicación

```
polling1: IN 20, a;; esperar botón
CMP a, @0
BEQ polling1
IN 24, b ; primer operando entra por switches
OUT 28,b
OUT 20, @1 ; poner a cero el flag antes de volver a leer
polling2: IN 20, a;; esperar botón
CMP a, @0
BEQ polling2
IN 24, c ; segundo operando entra por switches
OUT 28,c
OUT 20, @1 ; poner a cero el flag antes de volver a leer
multiplica:
OUT 0,b
OUT 1,c
OUT 2,@13
IN 3,resultado
aqui: OUT 16, resultado
JMP aqui
```

Reportes: Síntesis

Macro Statistics

# RAMs	: 7
128x16-bit single-port RAM	: 1
16x7-bit single-port Read Only RAM	: 4
4x4-bit single-port Read Only RAM	: 1
8x8-bit single-port Read Only RAM	: 1
# Adders/Subtractors	: 15
16-bit adder	: 3
18-bit adder	: 1
21-bit subtractor	: 4
3-bit adder	: 2
4-bit adder	: 2
7-bit adder	: 1
7-bit addsub	: 1
9-bit adder	: 1
# Registers	: 36
1-bit register	: 5
16-bit register	: 10
18-bit register	: 1
21-bit register	: 4
3-bit register	: 2
4-bit register	: 6
5-bit register	: 1
7-bit register	: 2
8-bit register	: 4
9-bit register	: 1

# Comparators	: 3
16-bit comparator equal	: 2
16-bit comparator not equal	: 1
# Multiplexers	: 71
1-bit 2-to-1 multiplexer	: 19
16-bit 2-to-1 multiplexer	: 5
16-bit 3-to-1 multiplexer	: 1
16-bit 32-to-1 multiplexer	: 1
16-bit 4-to-1 multiplexer	: 4
16-bit 8-to-1 multiplexer	: 1
21-bit 2-to-1 multiplexer	: 16
21-bit 4-to-1 multiplexer	: 4
3-bit 2-to-1 multiplexer	: 2
4-bit 2-to-1 multiplexer	: 12
4-bit 4-to-1 multiplexer	: 2
7-bit 4-to-1 multiplexer	: 1
8-bit 2-to-1 multiplexer	: 1
8-bit 4-to-1 multiplexer	: 1
9-bit 2-to-1 multiplexer	: 1
# FSMs	: 7
# Xors	: 2
16-bit xor2	: 2

Reportes: Area ocupada

Selected Device : 6slx16csg324-2

Slice Logic Utilization:

Number of Slice Registers:	447 out of 18224	2%
Number of Slice LUTs:	728 out of 9112	7%
Number used as Logic:	680 out of 9112	7%
Number used as Memory:	48 out of 2176	2%
Number used as RAM:	48	

Reportes: Tiempo

Timing constraint: Default period analysis for Clock 'clk'
Clock period: 7.870ns (frequency: 127.065MHz)
Total number of paths / destination ports: 36172 / 1021

Delay: **7.870ns (Levels of Logic = 10)**
Source: MS/C/estado_FSM_FFd2 (FF)
Destination: TimerModule/running (FF)
Source Clock: clk rising
Destination Clock: clk rising

Last But Not Least
presentamos:

Tokuwaga: El ensamblador

Tokuwaga: El ensamblador

- No quiere decir La Máquina Sencilla en japonés.
- Es el nombre de un guerrero que se llama Tokugawa y que pacificó Japón allá por los años 1600...
- Tokugawa



Características Generales:

Ensamblador de la máquina sencilla

- Soporta las 8 instrucciones básicas: ADD, CMP, BEQ, MOV, IN, OUT, CALL, RET
- Soporta pseudoinstrucciones:
 - JMP (salto incondicional), DEC, INC, LEA (Load Effective Address), SUB (resta)
- Soporta directivas de almacenaje de datos
 - DW
- Soporta constantes
- Informa tamaño memoria ocupada
- Tiene un desensamblador para verificación de código
- Tiene un simulador
- Ver manual de Tokuwaga

MS no soporta modo inmediato

- La MS no contiene modo de direccionamiento inmediato, es decir, no soporta constantes
- Esto obligaba, antes de disponer de un ensamblador, a tener que almacenar las constantes en posiciones de memoria, y referenciarlas con esa dirección.
- Lo mismo sucedía con las etiquetas, había que llevar la cuenta de las direcciones
- Este trabajo manual generaba muchísimos errores

Ejemplo pre-ensamblador

Esto escribíamos

```
1110000011000110
0110000001000110
1100000000000000
1110000001000010
1111000101000010
1110000111000110
0110000001000110
11000000000000101
1110000001000011
1111000101000011
1010000001000100
1010000001000101
0110001011000011
1100000000010010
0010000101000100
0010000011000101
0110000001000000
1100000000001100
1111001001000100
0110000001000000
1100000000001100
```

IN	1, flag
CMP	cero, flag
BEQ	0
IN	0,a
OUT	2,a
IN	3,flag
CMP	cero, flag
BEQ	5
IN	0,b
OUT	2,b
MOV	cero,c
MOV	cero,i
CMP	i,b
BEQ	18
ADD	uno,c
ADD	uno,i
CMP	cero,cero
BEQ	12
OUT	4, c
CMP	cero,cero
BEQ	18

Mapa de Instrucciones

polling_a	0
polling_b	5
fin	18
bucle	12

Mapa de Datos

cero	64	1000000
uno	65	1000001
a	66	1000010
b	67	1000011
c	68	1000100
i	69	1000101
Flag	70	1000110

Mapa de I/O

switches	0
boton1	1
Boton2	3
leds	2
7-seg	4

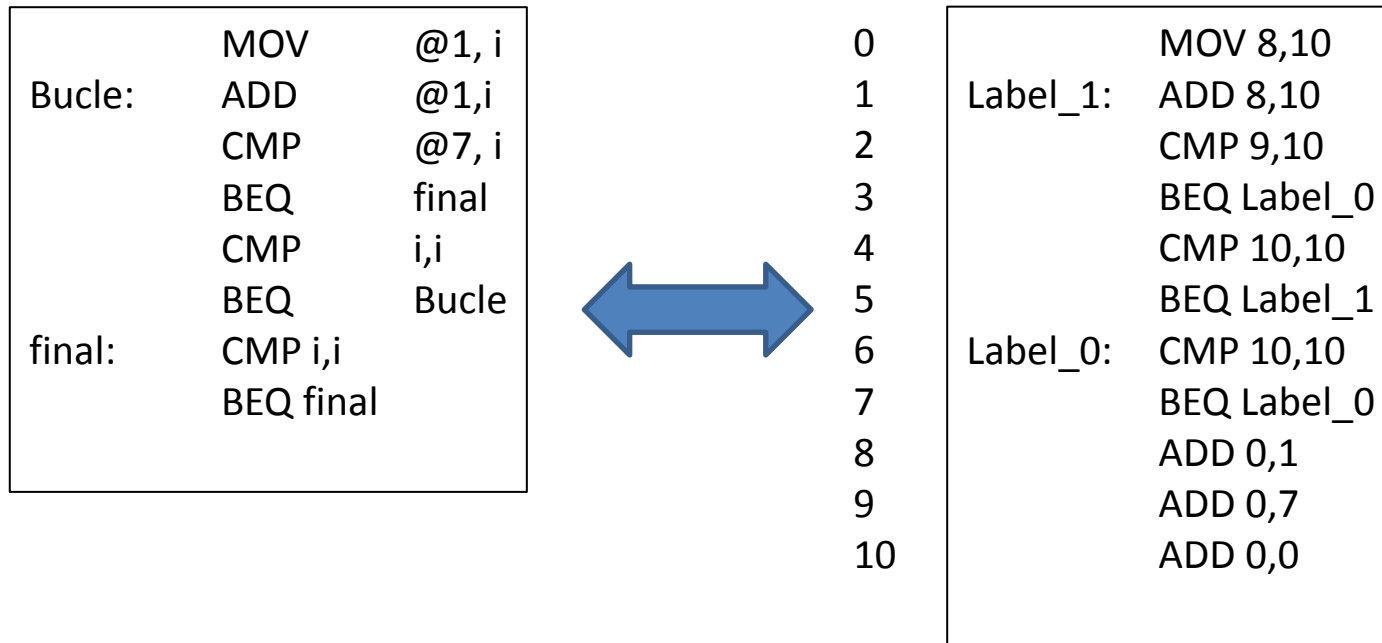
Ejemplo post ensamblador usando constantes y etiquetas

IN	dir1, flag
CMP	cero, flag
BEQ	0
IN	dir0,a
OUT	dir2,a
IN	dir3,flag
CMP	cero, flag
BEQ	5
IN	dir0,b
OUT	dir2,b
MOV	cero,c
MOV	cero,i
CMP	i,b
BEQ	18
ADD	uno,c
ADD	uno,i
CMP	cero,cero
BEQ	12
OUT	dir4, c
CMP	cero,cero
BEQ	18

Polling1:	IN	1, flag
	CMP	@0, flag
	BEQ	Polling1
	IN	0,a
	OUT	2,a
Polling2:	IN	3,flag
	CMP	@0, flag
	BEQ	Polling2
	IN	0,b
	OUT	2,b
	MOV	@0,c
	MOV	@0,i
Bucle:	CMP	i,b
	BEQ	Final
	ADD	@1,c
	ADD	@1,i
	CMP	@0,@0
	BEQ	Bucle
Final:	OUT	4, c
	CMP	@0,@0
	BEQ	Final

- El ensamblador nos resuelve el problema de asignar direcciones a constantes y armar la tabla de etiquetas.

Constante y Etiquetas



Pseudoinstrucciones: JMP, INC

```
Polling1:  IN      1, flag
           CMP     @0, flag
           BEQ     Polling1
           IN      0,a
           OUT     2,a
Polling2:  IN      3,flag
           CMP     @0, flag
           BEQ     Polling2
           IN      0,b
           OUT     2,b
           MOV     @0,c
           MOV     @0,i
Bucle:    CMP     i,b
           BEQ     Final
           ADD     @1,c
           ADD     @1,i
           CMP     @0,@0
           BEQ     Bucle
Final:    OUT     4, c
CMP       @0,@0
BEQ       Final
```



- Cambiamos la manera de escribir pero el código generado por el ensamblador usa sólo las 4 instrucciones verdaderas

```
Polling1:  IN      1, flag
           CMP     @0, flag
           BEQ     Polling1
           IN      0,a
           OUT     2,a
Polling2:  IN      3,flag
           CMP     @0, flag
           BEQ     Polling2
           IN      0,b
           OUT     2,b
           MOV     @0,c
           MOV     @0,i
Bucle:    CMP     i,b
           BEQ     Final
           INC     c
           INC     i
           JMP     bucle
Final:    OUT     4, c
           JMP     Final
```

Ejemplo: pseudoinstrucción SUB

- Algunas pseudoinstrucciones son verdaderos programas:

```
MOV @7,a  
MOV @9, b  
SUB a,b  
OUT LED,b  
final:JMP final
```

desensamblador

```
0  MOV 14,17  
1  MOV 15,18  
2  MOV 18,16  
3  Label_1: CMP 12,16  
4  BEQ Label_0  
5  ADD 13,16  
6  ADD 13,17  
7  CMP 0,0  
8  BEQ Label_1  
9  Label_0: OUT 28,18  
10 Label_2: CMP 0,0  
11 BEQ Label_2  
12 ADD 0,0  
13 OUT 31,127  
14 ADD 0,7  
15 ADD 0,9  
16 ADD 0,0  
17 ADD 0,0  
18 ADD 0,0
```

- La instrucción ADD 0,7 corresponde al dato 7 en la posición de memoria 14

Ejemplo: Pseudoinstrucción Shift

- Las pseudo-instrucciones SHIFTL y SHIFTR se ensamblan generando las operaciones de escritura y lectura de los registros del Controlador de Shifter

```
MOV @7,a  
SHIFTL a,@2  
OUT LED,a  
final:JMP final
```

0	MOV 9,11
1	OUT 5,8
2	OUT 4,11
3	IN 6,11
4	OUT 28,11
5	Label_1:CMP 0,0
6	BEQ Label_1
7	ADD 0,0
8	ADD 0,2
9	ADD 0,7
10	ADD 0,0
11	ADD 0,0

¡Punteros!

- Si realmente no puede vivir sin punteros...¡Tokuwaga se los da! (Piense bien antes de usarlos, ocupa mucho código implementarlos, y justamente no es memoria lo que sobra)
- Nueva instrucción LEA (Load Effective Address)
- Nuevo modo de direccionamiento para la instrucción MOV= indirección []

Punteros: LEA

MOV @6,i

LEA Data, dir1 ; obtiene dirección del vector

MOV i, [dir1]; escribe valor 6 en vector[0]

OUT 16, dir1 ; escribe el valor del puntero en los leds

MOV [dir1],a ; lee contenido del vector[0]

OUT 28,a ; lo muestra en los 7 segmentos

final: JMP final

Data: DW 10

LEA:

- crea una posición de memoria para la dirección de Data (dirección 21)
- Esa posición de memoria es la dirección , cuyo contenido es 21, es la dirección 31 en el ejemplo
- dir1 es la dirección 29
- Luego se reemplaza LEA 21, 29 por MOV 31, 29
- O sea, en la dirección 29 estará el puntero a Data

0	MOV 25,30
1	MOV 31,29
2	MOV 33,8
3	OUT 4,26
4	OUT 5,8
5	IN 6,8
6	ADD 24,8
7	ADD 29,8
8	MOV 30,0
9	OUT 16,29
10	MOV 29,17
11	OUT 4,26
12	OUT 5,17
13	IN 6,17
14	ADD 24,17
15	MOV 32,27
16	ADD 27,17
17	MOV 29,0
18	OUT 28,28
19	Label_0: CMP 0,0
20	BEQ Label_0
21	ADD 0,10
22	ADD 0,0
23	ADD 0,1
24	MOV 0,0
25	ADD 0,6
26	ADD 0,7
27	ADD 0,0
28	ADD 0,0
29	ADD 0,0
30	ADD 0,0
31	ADD 0,21
32	ADD 0,28
33	ADD 0,30

Punteros: Indirección

```
MOV @6,i
LEA Data, dir1 ; obtiene dirección del vector
MOV i, [dir1]; escribe valor 6 en vector[0]
OUT 16, dir1 ; escribe el valor del puntero en los leds
MOV [dir1],a ; lee contenido del vector[0]
OUT 28,a ; lo muestra en los 7 segmentos
final: JMP final
Data:      DW 10
```

Código automodificable

Indirección MOV A,[B]

```
LEA A,X
OUT PUERTO_0_SHIFTER,@7
OUT PUERTO_1_SHIFTER,X
IN PUERTO_2_SHIFTER,X
ADD @32768,X
ADD B,X
X:    MOV A,0
```

```
0  MOV 25,30
1  MOV 31,29
2  MOV 33,8
3  OUT 4,26
4  OUT 5,8
5  IN 6,8
6  ADD 24,8
7  ADD 29,8
8  MOV 30,0
9  OUT 16,29
10 MOV 29,17
11 OUT 4,26
12 OUT 5,17
13 IN 6,17
14 ADD 24,17
15 MOV 32,27
16 ADD 27,17
17 MOV 29,0
18 OUT 28,28
19 Label_0: CMP 0,0
20 BEQ Label_0
21 ADD 0,10
22 ADD 0,0
23 ADD 0,1
24 MOV 0,0
25 ADD 0,6
26 ADD 0,7
27 ADD 0,0
28 ADD 0,0
29 ADD 0,0
30 ADD 0,0
31 ADD 0,21
32 ADD 0,28
33 ADD 0,30
```

Punteros: Indirección

```
MOV @6,i
LEA Data, dir1 ; obtiene dirección del vector
MOV i, [dir1]; escribe valor 6 en vector[0]
OUT 16, dir1 ; escribe el valor del puntero en los leds
MOV [dir1],a ; lee contenido del vector[0]
OUT 28,a ; lo muestra en los 7 segmentos
final: JMP final
Data:      DW 10
```

Código automodificable

```
Indirección: MOV [A],B
OUT PUERTO_0_SHIFTER,@7
OUT PUERTO_1_SHIFTER,A
IN PUERTO_2_SHIFTER,X
ADD @32768,X
LEA B,var
ADD var,X
X: MOV A,0
```

```
0  MOV 25,30
1  MOV 31,29
2  MOV 33,8
3  OUT 4,26
4  OUT 5,8
5  IN 6,8
6  ADD 24,8
7  ADD 29,8
8  MOV 30,0
9  OUT 16,29
10 MOV 29,17
11 OUT 4,26
12 OUT 5,17
13 IN 6,17
14 ADD 24,17
15 MOV 32,27
16 ADD 27,17
17 MOV 29,0
18 OUT 28,28
19 Label_0: CMP 0,0
20 BEQ Label_0
21 ADD 0,10
22 ADD 0,0
23 ADD 0,1
24 MOV 0,0
25 ADD 0,6
26 ADD 0,7
27 ADD 0,0
28 ADD 0,0
29 ADD 0,0
30 ADD 0,0
31 ADD 0,21
32 ADD 0,28
33 ADD 0,30
```

Herramientas para Programar

- Ensamblador: ver manual
- Desensamblador
- Simulador



Vista del Plan Ahead

The screenshot displays the PlanAhead 14.6 software interface. The top menu bar includes File, Edit, Flow, Tools, Window, Layout, View, and Help. The title bar shows the project path: MS_final - [C:/Users/Patricia/Materia FPGA/SASE2016/sase_project-master_14_07/Proyecto_MS/MS_final.ppr] - PlanAhead 14.6.

The left sidebar contains the Flow Navigator and Project Manager. The Project Manager is expanded, showing a hierarchy of sources. The 'Sources' pane lists the project structure, including 'Placa (Placa.v)' and its sub-components like 'MS - MaquinaSencilla', 'C - UC', 'P - UP', 'A - Alu', 'Mem - RAM', 'L - LogicalIO', and various control modules. The 'Hierarchy' pane shows the selected source, 'Placa (Placa.v)', and its properties.

The 'Source Node Properties' pane for 'Placa (Placa.v)' shows the following details:

- Module: Placa
- Location: C:/Users/Patricia/Materia FPGA/SASE2016/sase_
- Type: Verilog
- Library: work
- Size: 3.5 KB
- Modified: Wednesday, 07/12/16 04:39:59 PM

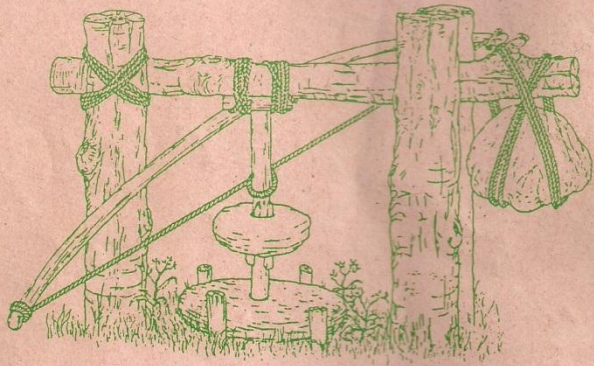
The main editor window displays the Verilog code for the 'Placa' module. The code defines the module's inputs, outputs, and internal logic, including a 'module Placa' declaration and a 'module MaquinaSencilla' instantiation.

```
22
23 module Placa(
24     input clk,
25     input reset,
26     input [7:0] sw,
27     input [3:0] btn,
28     output [7:0] Led, sseg,
29     output [3:0] an,
30     input rx,
31     output tx
32 );
33
34 wire [15:0] import;
35 wire [4:0] dirport;
36 wire [15:0] output;
37 wire we;
38 MaquinaSencilla MS(.clk(clk), .reset(reset), .import(import), .dirport(dirport), .outpc
39
40 wire [18:0] device0in, device1in, device2in, device3in, device4in, device5in, device
41 wire [15:0] device0out, device1out, device2out, device3out, device4out, device5out, dev
42 LogicalIO L(.dev_sel(dirport[4:2]), .reg_sel(dirport[1:0]), .we(we), .data_out(output),
43     .device0in(device0out), .device0out(device0in), .device0cs(device0cs),
44     .device1in(device1out), .device1out(device1in), .device1cs(device1cs),
45     .device2in(device2out), .device2out(device2in), .device2cs(device2cs),
46     .device3in(device3out), .device3out(device3in), .device3cs(device3cs),
47     .device4in(device4out), .device4out(device4in), .device4cs(device4cs),
48     .device5in(device5out), .device5out(device5in), .device5cs(device5cs),
49     .device6in(device6out), .device6out(device6in), .device6cs(device6cs),
50     .device7in(device7out), .device7out(device7in), .device7cs(device7cs)
51 );
```

The bottom status bar shows the zoom level (23:1), the current mode (Insert), and the language (Verilog).

La máquina sencilla...

- Continuará... porque es un soporte perfecto para aprender a programar hardware y para entender los conceptos mas elementales, pero no por ello comprendidos por el alumno, en cualquier nivel de la carrera en que se encuentre, de un sistema computador.



cpda
fib
upc

La Máquina Sencilla

1988-2016