DEPARTAMENTO DE CIENCIAS DE LA COMPUTACIÓN
ESCUELA DE INGENIERÍA
PONTIFICIA UNIVERSIDAD CATÓLICA DE CHILE

IIC2343 - Arquitectura de Computadores (II/2025)

Etapa 2 del Proyecto

Descripción

Su trabajo en esta etapa se realizará tanto en el componente `Basys3` como la `CPU`.

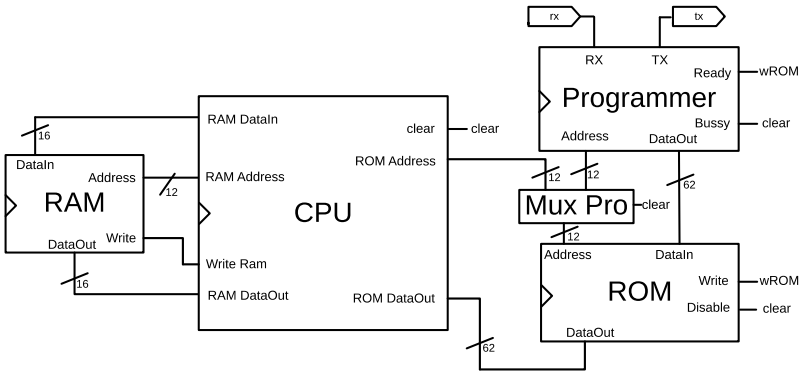

Figura 1: Diagrama parcial del computador básico de proyecto dentro del componente Basys3.

Para completar su computador básico, deberá incorporar la funcionalidad de **subrutinas**, junto con las funciones de entrada y salida de dispositivos I/O **mapeados a memoria**. La conectividad con la placa cambiará respecto a la entrega pasada. Ahora, existirán registros encargados de la conectividad con los dispositivos, por lo que toda conexión de interruptores, botones y LEDs debe ser eliminada del archivo `Basys3`. Los cambios a la arquitectura se pueden observar en el diagrama.

Hardware

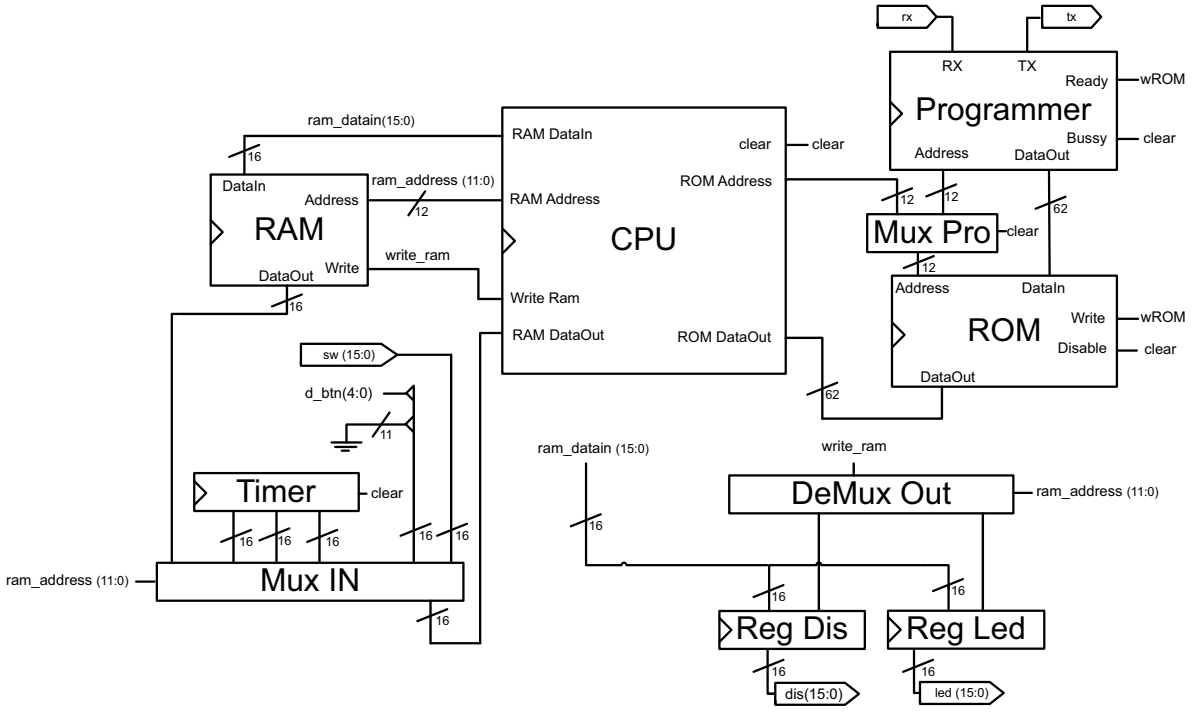

Figura 2: Diagrama parcial del computador básico de proyecto con entradas y salidas mapeadas a memoria dentro del componente `Basys3`.

Para esta entrega, la conectividad con la placa cambiará respecto a la entrega pasada. Ahora, existirán registros encargados de la conectividad con los dispositivos, por lo que toda conexión de interruptores, botones y LEDs debe ser eliminada del archivo `Basys3`.

Las entradas y salidas se mapearán a las siguientes direcciones en la memoria de datos:

Dirección	Nombre	Tipo
0	LEDs	Salida
1	Switches	Entrada
2	Displays	Salida
3	Botones	Entrada
4	Segundos	Entrada
5	MSegundos	Entrada
6	uSegundos	Entrada

Para la entrada de los botones, la señal **`d_btn`** deberá ser extendida hasta 16 bits agregando ceros por su extremo más significativo, *i.e.* `"00..0" & d_btn`.

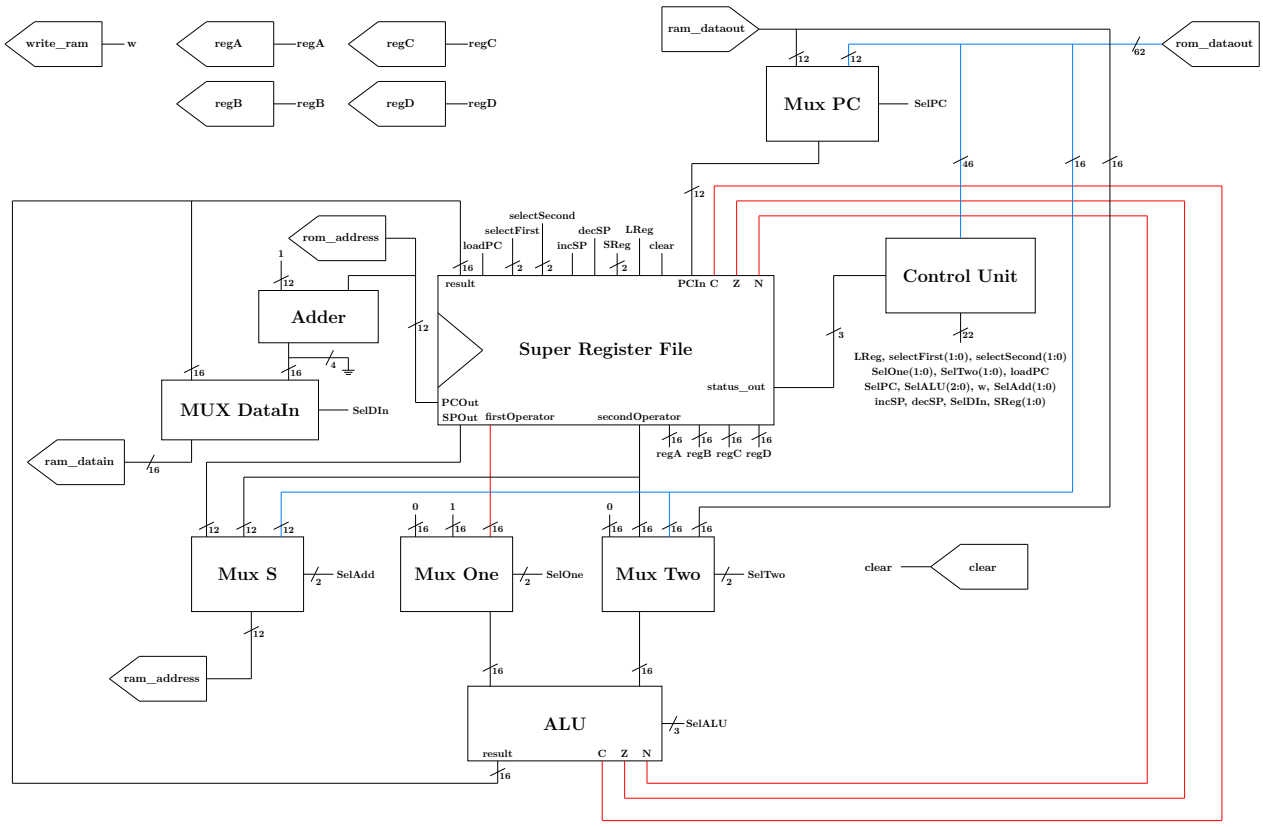

Figura 3: Diagrama interno de la `CPU` del computador básico del proyecto.

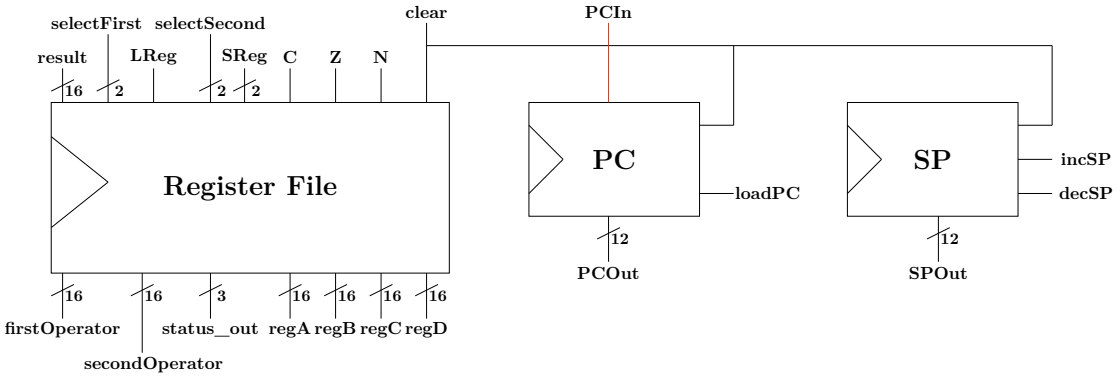

Figura 4: Diagrama interno del componente `Super Register File`.

Componentes a agregar

Dentro del componente `Super Register File`:

- Un **registro contador** `SP` para la dirección del puntero al *stack* de al menos 12 *bits*.

Dentro del componente `CPU`:

- Un **multiplexor** `MUX PC` para la carga de `PC` que seleccione una de dos entradas, cada una de 12 *bits*.

- Un **multiplexor** `MUX DataIn` para la entrada de datos de la `RAM` que seleccione una de dos entradas, cada una de 16 *bits*.
- Un **sumador** `Adder` que incrementa el valor de 12 bits del `PC` en una unidad y entrega un resultado en 12 *bits*, al que se le concatenan 4 *bits* de valor 0 en la posición más significativa del bus de salida.

Dentro del componente `Basys3`:

- Un **componente** `Timer` que entrega 3 enteros de 16 *bits*: segundos, milisegundos y microsegundos. Este fue entregado junto con el proyecto base.
- Un **registro** `Dis` de 16 *bits* para conservar el estado de los *displays* de 8 segmentos.
- Un **registro** `Led` de 16 *bits* para conservar el estado de los LEDs.
- Un **multiplexor** `In` que permite el mapeo de los dispositivos de entrada a la memoria de datos.
- Un **demultiplexor** `Out` que permite el mapeo de los dispositivos de salida a la memoria de datos.

Software

En esta etapa, usará un ensamblador para traducir y transferir distintos programas en lenguaje de *Assembly* a la ROM de su computador y, de ese modo, poder ejecutarlos. Escriban los programas que estime convenientes para probar su arquitectura. Al final del enunciado, se encuentran algunos ejemplos. No está de más mencionar que estos son, como se señaló, **ejemplos** y no son *tests*. Debido a esto, el funcionamiento correcto de estos ejemplos no asegura que su computador sea 100 % funcional.

En la siguiente tabla, se describe cómo las palabras en la ROM son codificadas para representar las instrucciones. El ensamblador entregado se encargará de leer las instrucciones en *Assembly*, codificarlas y enviarlas al computador programable. En caso de que ninguno de los bits de la palabra de la ROM esté activo para la selección de un multiplexor, estos deben seleccionar una de sus señales de entrada de forma arbitraria a través de `with others` en Vivado. Se muestra en la sección ejemplos varios programas que puede usar para probar la arquitectura.

Importante: Deben considerar que en los ejemplos ocurre una de dos cosas: (1) No se definen variables y se utilizan las direcciones de entrada y salida de forma explícita en el código; o (2) se definen variables de forma que sus *labels* se asocien directamente a las direcciones de los dispositivos correspondientes. Si usted define variables en posiciones asociadas a direcciones mapeadas a dispositivos sin esa intención, su código **no funcionará correctamente** ya que no leerá ni escribirá el contenido en la memoria de datos.

Bits	Acción
0	1 cuando la operación de la ALU es Sumar
1	1 cuando la operación de la ALU es Restar
2	1 cuando la operación de la ALU es `AND`
3	1 cuando la operación de la ALU es `OR`
4	1 cuando la operación de la ALU es `XOR`
5	1 cuando la operación de la ALU es `NOT`
6	1 cuando la operación de la ALU es *shift right*
7	1 cuando la operación de la ALU es *shift left*
8	1 cuando se carga un registro
9	1 cuando el primer operando es `A`
10	1 cuando el primer operando es `B`
11	1 cuando el primer operando es `C`
12	1 cuando el primer operando es `D`
13	1 cuando el segundo operando es `A`
14	1 cuando el segundo operando es `B`
15	1 cuando el segundo operando es `C`
16	1 cuando el segundo operando es `D`
17	1 cuando se carga el registro `A`
18	1 cuando se carga el registro `B`
19	1 cuando se carga el registro `C`
20	1 cuando se carga el registro `D`
21	1 cuando en el `MuxOne` se selecciona el primer operando de `Super Register File`
22	1 cuando en el `MuxOne` se selecciona el Uno
23	1 cuando en el `MuxOne` se selecciona el Cero
24	1 cuando en el `MuxTwo` se selecciona el segundo operando de `Super Register File`
25	1 cuando en el `MuxTwo` se selecciona el Cero
26	1 cuando en el `MuxTwo` se selecciona el Literal
27	1 cuando en el `MuxTwo` se selecciona la salida de la RAM
28	1 cuando el `MuxDataIn` se selecciona la salida de la ALU
29	1 cuando el `MuxDataIn` se selecciona `PC+1`
30	1 cuando se habilita la escritura en la RAM
31	1 cuando en el `MuxS` se selecciona el Literal
32	1 cuando en el `MuxS` se selecciona el segundo operando de `Super Register File`
33	1 cuando en el `MuxS` se selecciona `SP`
34	1 cuando se incrementa el `SP`
35	1 cuando se decrementa el `SP`
36	1 cuando hay un salto incondicional
37	1 cuando hay un salto por Igualdad
38	1 cuando hay un salto por Desigualdad
39	1 cuando hay un salto por ser Mayor
40	1 cuando hay un salto por ser Mayor o Igual
41	1 cuando hay un salto por ser Menor
42	1 cuando hay un salto por ser Menor o Igual
43	1 cuando hay un salto por un Acarreo
44	1 cuando en el `MuxPC` se selecciona el Literal
45	1 cuando en el `MuxPC` se selecciona la salida de la RAM
61 al 46	Valor del Literal de 16 bits

Tabla 1: Tabla de palabras de control.

Presentación

Se aceptarán envíos de *commits* hasta **el martes 21 de octubre a las 13:30 horas**, momento en el que se recolectarán todos los repositorios. **Se tendrá que presentar en los computadores del laboratorio en no más de 10 minutos** el día de laboratorio correspondiente a su sección durante esa misma semana. Esto se llevará a cabo con la descarga del comprimido de su repositorio que el equipo docente le proveerá. El día de presentación deberá mostrar y cargar con el ensamblador los *tests* para verificar el funcionamiento correcto de su arquitectura. Nuevamente, tendrán que explicar cuáles fueron las decisiones de diseño que realizaron y subir al repositorio de su grupo en *GitHub* su proyecto en Vivado.

Puntaje

La nota será calculada utilizando 4 *tests*. Cada uno de estos debe correr de manera consecutiva, es decir, si su grupo no logra pasar el *test* 1, no se probará el *test* 2). Esto se debe a que el *test* `i+1` hará uso de las instrucciones del *test* `i`. Si no se logra pasar un *test* en su totalidad, no obstante, se probarán *sub-tests* de este, en donde se asignará puntaje parcial según su ejecución.

Los *tests*, junto con sus puntajes, son los siguientes:

- ***Test* 1: Correctitud de entrega 1 completa (1.5 puntos)**
- ***Test* 2: Subrutinas y manejo de memoria *stack* (1.5 punto)**
- ***Test* 3: Manejo de entradas y salidas con interruptores, *leds* (1.5 punto)**
- ***Test* 4: Manejo total de todo tipo de entrada y salida (1.5 punto)**

Cálculo de nota entrega: $PuntajeObtenido + 1$

Es importante mencionar que el uso de *process* en componentes que no requieren sincronía significará **la evaluación de toda la entrega con nota mínima.**

Recuperación de puntaje y requisitos

Durante las semanas previas a la presentación de la entrega, se realizarán salas de ayuda en el horario de laboratorio para que trabaje en el proyecto y resuelva dudas con sus ayudantes. Si su grupo termina con un 100 % de asistencia en estas instancias, se le permitirá **recuperar hasta la mitad del puntaje descontado en la evaluación**. Para que la asistencia sea considerada, **al menos una persona** del grupo debe estar presente **durante los dos bloques de laboratorio** que correspondan a su sección. **Solo se aceptará asistencia al laboratorio de su sección**, es decir, cualquier persona que intente asistir a un laboratorio que no le corresponde no será admitida y su asistencia no será registrada.

La recuperación del puntaje, en caso de cumplir con los criterios de asistencia, se realizará durante el próximo laboratorio que corresponda a su sección (semana del 27 de octubre). En esta, debe mostrar que arregló **todos** los errores de su entrega original. El código de la recuperación se presentará el día de su laboratorio correspondiente, donde deberá incluir los arreglos de esta entrega, pero **nada posterior a ella**.

Assembly

MOV	R1,R2	Guarda el valor de R2 en R1
	R1,Lit	Guarda un literal Lit en R1
	R1,(Dir)	Guarda el valor de Mem[Dir] en R1
	(Dir),R1	Guarda el valor de R1 en Mem[Dir]
	R1,(R2)	Guarda el valor de Mem[R1] en R2
	(R2),R1	Guarda el valor de R1 en Mem[R2]
	(R2),Lit	Guarda un literal Lit en Mem[R2]
ADD SUB AND OR XOR	R1,R2	Guarda el resultado de R1 op R2 en R1
	R1,Lit	Guarda el resultado de R1 op Lit en R1
	R1,(Dir)	Guarda el resultado de R1 op Mem[Dir] en R1
	R1,(R2)	Guarda el resultado de R1 op Mem[R2] en R1
NOT SHL SHR	R1	Guarda el resultado de op R1 en R1
	(Dir),R1	Guarda el resultado de op R1 en Mem[Dir]
	(R2),R1	Guarda el resultado de op R1 en Mem[R2]
INC	R1	Incrementa el valor de R1 en una unidad
	(Dir)	Incrementa el valor de Mem[Dir] en una unidad
	(R1)	Incrementa el valor de Mem[R1] en una unidad
DEC	R1	Decrementa el valor de R1 en una unidad
CMP	R1,R2	Ejecuta la instrucción SUB R1,R1 sin actualizar el valor en ningún registro
	R1,Lit	Ejecuta la instrucción SUB R1,Lit sin actualizar el valor en ningún registro
	R1,(Dir)	Ejecuta la instrucción SUB R1,(Dir) sin actualizar el valor en ningún registro
	R1,(R2)	Ejecuta la instrucción SUB R1,(R2) sin actualizar el valor en ningún registro
JMP	Ins	Carga la dirección de la instrucción Ins en PC
JEQ	Ins	Carga la dirección de la instrucción Ins en PC si en Status se cumple Z = 1
JNE	Ins	Carga la dirección de la instrucción Ins en PC si en Status se cumple Z = 0
JGT	Ins	Carga la dirección de la instrucción Ins en PC si en Status se cumple N = 0 y Z = 0
JGE	Ins	Carga la dirección de la instrucción Ins en PC si en Status se cumple N = 0
JLT	Ins	Carga la dirección de la instrucción Ins en PC si en Status se cumple N = 1
JLE	Ins	Carga la dirección de la instrucción Ins en PC si en Status se cumple N = 1 o Z = 1
JCR	Ins	Carga la dirección de la instrucción Ins en PC si en Status se cumple C = 1
NOP		No hace cambios
PUSH	R1	Guarda el valor de R1 en Mem[SP] y decrementa SP en una unidad
POP	R1	Incrementa SP en una unidad y en el siguiente ciclo guarda el valor de Mem[SP] en R1
CALL	Ins	Guarda PC+1 en Mem[SP], carga la dirección de la instrucción Ins en PC y decrementa SP en una unidad
RET		Incrementa SP en una unidad y en el siguiente ciclo carga el valor de Mem[SP] en PC

Ejemplos

- Programa 13:

```
DATA:

CODE:

MOV B,AAAAh     // |
MOV (0),B       // | Mostrar "1010101010101010" en los Leds

MOV C,ABCDh     // |
MOV (2),C       // | Mostrar "ABCD" en el Display


end:
JMP end         //   No Hacer Nada Más
```

- Programa 14:

```
DATA:

CODE:                   // Sumar inputs

MOV A,(1)               // Guardar switches
MOV D,(3)               // Guardar botones
ADD A,D                 // Sumar inputs
MOV (2),A               // Mostrar en el Display

end:
 JMP end
```

- Programa 15:

```
DATA:

CODE:           // Pseudo Reloj | Velocidad de clock a "full"

loop:

 MOV A,(4)      // |
 MOV (2),A      // | Mostrar Segundos en el Display

 MOV B,(5)      // |
 MOV (0),B      // | Y Milisegundos en los Leds

JMP loop        //   Repetir
```

- Programa 16:

```
DATA:

led  0
sw   0
dis  0
btn  0
sec  0
msec 0
usec 0

arr  0

CODE:                          // Sumar switches | Velocidad de clock a "full"

MOV B, arr                     // Puntero a B
PUSH B                         // Guardar puntero

input:
 CALL std_io_btn_wait          // Esperar cambio en botones
 MOV A,(sw)                    // Switches a A
 MOV (B),A                     // A a arr[i]
 INC B                         // Incrementar puntero
 CMP A,0                       // Si Switches != 0
 JNE input                     // Siguiente input

POP B                          // Recuperar puntero
MOV A,0                        // Resultado = 0

sumar:
 PUSH A                        // Guardar resultado
 MOV A,(B)                     // arr[i] a A
 CMP A,0                       // Si arr[i] == 0
 JEQ sumar_fin                 // Terminar
 POP A                         // Recuperar resultado
 ADD A,(B)                     // Resultado + arr[i]
 INC B                         // Puntero++
 JMP sumar                     // Siguiente
 sumar_fin:
 POP A                         // Recuperar Resultado

end:
 MOV (dis),A                   // Mostrar en el Display
 JMP end

////////////////Libreria std_io/////////////////////////////////////////
                                                                      //
std_io_btn_wait:               // * en A, * en B                      //
 PUSH B                        // Guarda B                            //
 MOV A,(btn)                   // Estado actual                       //
 std_io_btn_wait_press_lp:                                            //
  MOV B,(btn)                  // Nuevo estado                        //
  CMP A,B                      // Si ==                               //
  JEQ std_io_btn_wait_press_lp // Continuar                          //
 XOR B,A                       // Bits cambiados                      //
 std_io_btn_wait_release_lp:                                          //
  MOV A,(btn)                  // Nuevo estado                        //
  AND A,B                      // Bits aún cambiados                  //
  CMP A,0                      // SI != 0                             //
  JNE std_io_btn_wait_release_lp// Continuar                         //
 MOV A,B                       // Bits cambiados a A                  //
 POP B                         // Recupera B                          //
RET                            // Retorna Bit(s) en A                 //
                                                                      //
//////////////////////////////////////////////////////////////////////
```

- Programa 17:

```
DATA:

led  0
sw   0
dis  0
btn  0
sec  0
msec 0
usec 0


odd 0           // Es impar
last 0          // Fue impar

CODE:           // Parpadeo Inverso | Velocidad de clock a "full"

loop:

 MOV A,(sec)    // |
 AND A,1        // |
 MOV (odd),A    // | Segundo es Impar ?

 MOV B,(sw)     //   Leer Switches

 MOV A,0        // |
 NOT A          // |
 ADD A,(odd)    // |
 XOR A,B        // | Si fue Par, Invertir Switches

 MOV (led),A    //   Resultado a Leds

 MOV A,(last)   // |
 CMP A,(odd)    // |
 JEQ loop       // | Si Hubo Cambio

 MOV B,1        // |
 XOR (last)     // | Invierte Variable

 MOV A,(btn)    // |
 MOV (dis),A    // | Y Envia Botones al Display

JMP loop        //   Repetir
```