# Proyecto Optativo 2: Diseño de un controlador de bus

Fernando Aliaga Ramón - 610610

## Diseño: diagrama de estados

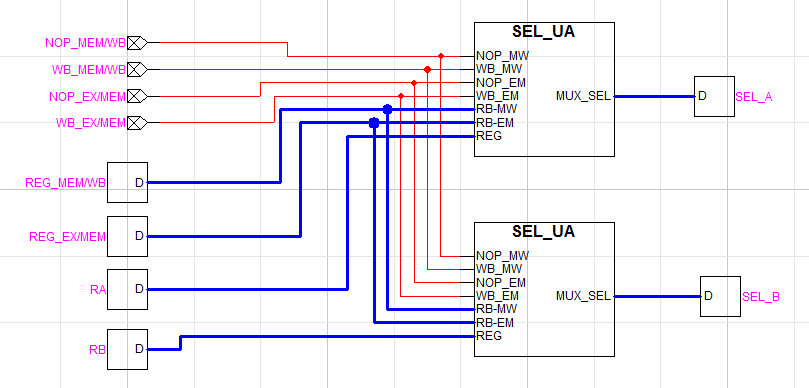
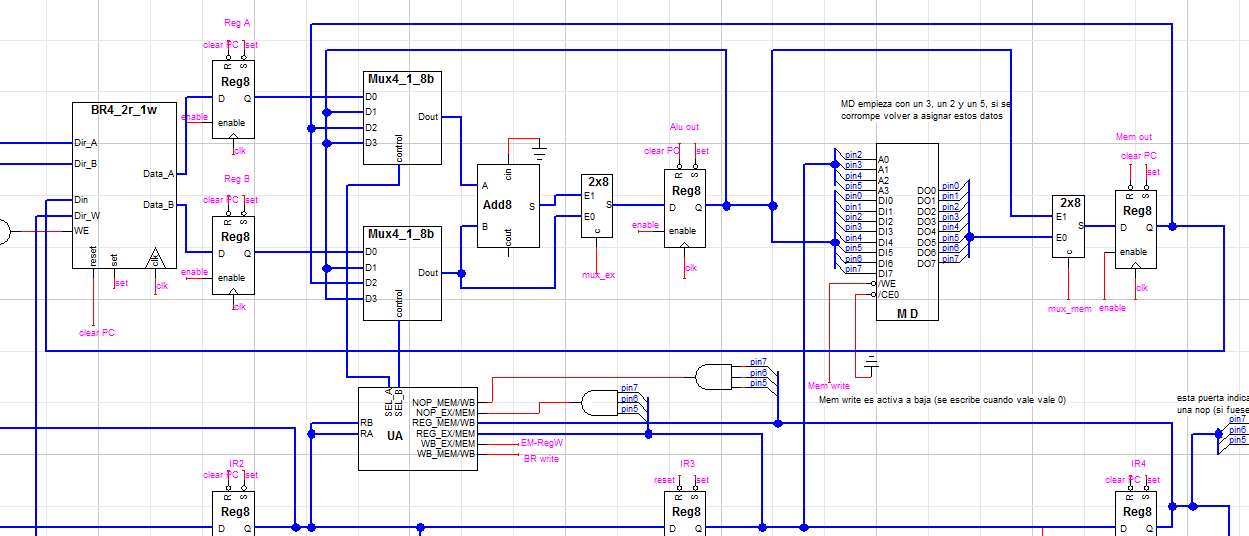
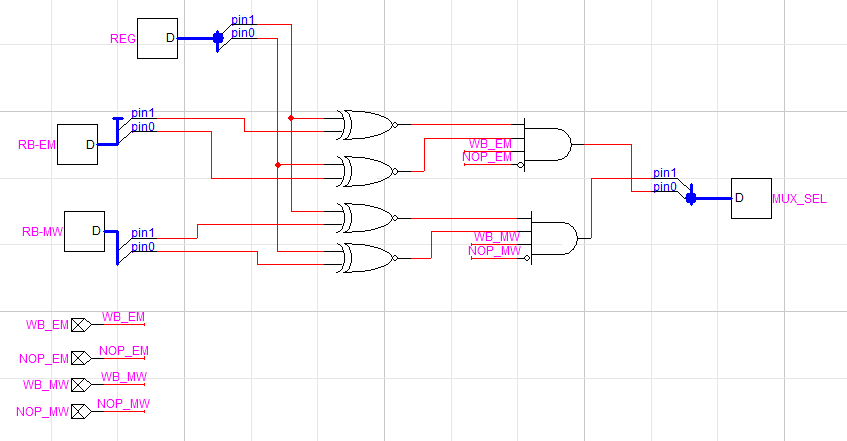
El controlador de Bus debe gestionar dos tipos de operaciones: lectura y escritura. Los ciclos son los siguientes:

Estado 0  
Lectura instrucción

Estado 1  
Primer ciclo RAM

Estado 0  
Soltar BUS

Estado 0  
Segundo ciclo lectura

**Procesador**

**SEL\_UA**

**Unidad de Anticipación**

Viendo el diseño podemos eliminar ciertos riesgos de datos:

## Operación Load

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Op | Ciclo 1 | Ciclo 2 | Ciclo 3 | Ciclo 4 | Ciclo 5 | Ciclo 6 | Ciclo 7 | Ciclo 8 | Ciclo 9 |
| LD | IF | ID | EX | **MEM** | BW |  |  |  |  |
| LD |  | IF | ID | EX | MEM | BW |  |  |  |
| ST |  |  | IF | ID | **EX** | MEM | BW |  |  |
| JMP |  | IF | ID | EX | MEM | BW |  |  |  |
| ADD |  |  | IF | ID | **EX** | MEM | BW |  |  |

## Operación Add

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Op | Ciclo 1 | Ciclo 2 | Ciclo 3 | Ciclo 4 | Ciclo 5 | Ciclo 6 | Ciclo 7 | Ciclo 8 | Ciclo 9 |
| ADD | IF | ID | **EX** | MEM | BW |  |  |  |  |
| LD |  | IF | ID | **EX** | MEM | BW |  |  |  |
| ST |  | IF | ID | **EX** | MEM | BW |  |  |  |
| JMP |  | IF | ID | EX | MEM | BW |  |  |  |
| ADD |  | IF | ID | **EX** | MEM | BW |  |  |  |

Al añadir nuevos componentes que operan en la etapa EX, deberemos realizar un nuevo estudio del camino crítico que marcará el tiempo del reloj.

TEX = 1ns(Reg8) + 1ns(and3&xnor) + 1ns(and4) + 4ns(mux) + 16ns(Adder) + 2ns(mux) +1ns(TsetupReg8) = 26ns

Por lo tanto, deberemos redefinir el tiempo del reloj a ciclos de 26ns para el correcto funcionamiento del procesador.

## Paso 3: Detención del segmentado

La anticipación de operandos nos permitía eliminar muchos riesgos de datos, sin embargo existía riesgo tras una operación de tipo Load si queríamos leer el registro que primero escribiría la operación Load. Para solucionar estos riesgos solo era posible a nivel de código, incluyendo una operación entre medio, ya sea una *nop* u otra instrucción. Para abstraer de esa tarea al programador del código se crea la *Unidad de Detención*, que se encarga de parar la ejecución hasta que la instrucción con riesgo obtenga sus operandos.

Dicha *Unidad* tendrá varias entradas y una única salida, la cual indica si vale 1 que hay que detener la ejecución y 0 si no hay que detener nada:

|  |
| --- |
| Datos de entrada |
| RA (2bits) |
| RB (2bits) |
| REG\_ID/EX (2bits) |
| OP\_ID/EX (2bit) |
| BRWrite\_ID/EX (1bit) |

Con estos datos de entrada podemos definir una función lógica que devuelva una correcta señal de *STOP*:

Si(BRWrite=1 & OP=00 & RA=REGID/EX ) o (BRWrite=1 & OP=00 & RB=REGID/EX)   
entonces STOP=1

Sino STOP = 0

La señal *STOP* indica al procesador que debe detenerse, lo cual significa a nivel hardware:

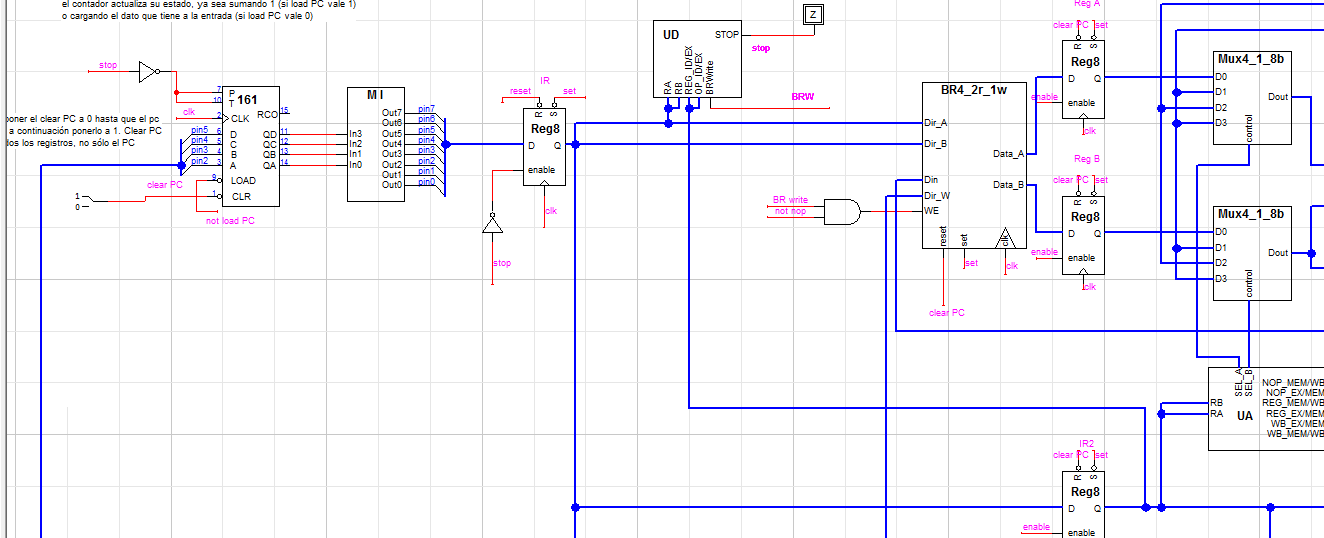
1. Los valores P y T del PC pasan a valer 0.
2. El valor *Enable* del Reg8 IR pasa a valer 0.

|  |  |  |  |
| --- | --- | --- | --- |
| MUX | LPC | MW | BRW |
| X | X | 0 | 0 |

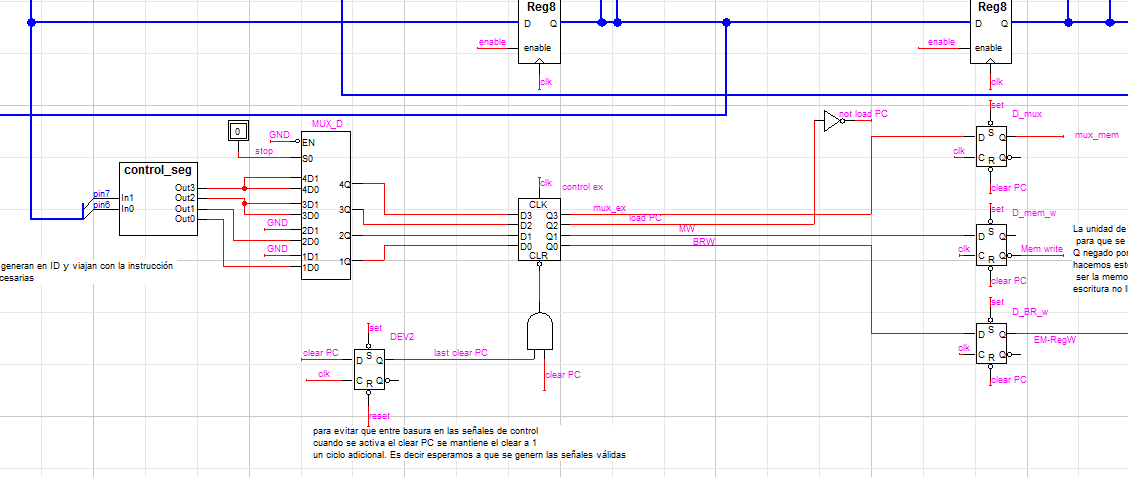
1. El valor del *Mux* que controla las señales de control pasa a valer 0, cargando las siguientes señales:

Esto hace que la misma instrucción viajará por el procesador sin escribir ni en memoria ni en el banco de registros.

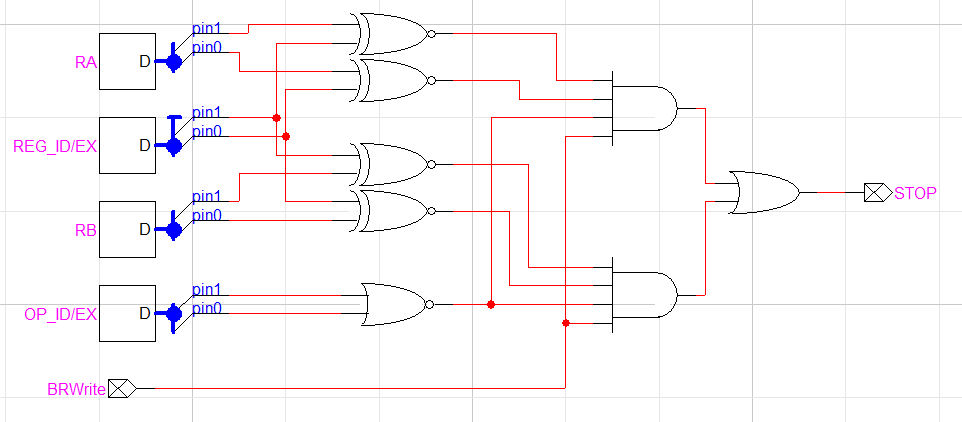
La implementación de la Unidad de Detención ha sido la siguiente:



**Procesador**



**Señal de Control**



**Unidad de Detención**

La ventaja que ofrece la detención del segmentado es la eliminación de riesgos de datos en la operación Load:

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Op | Ciclo 1 | Ciclo 2 | Ciclo 3 | Ciclo 4 | Ciclo 5 | Ciclo 6 | Ciclo 7 | Ciclo 8 | Ciclo 9 |
| LD | IF | ID | EX | MEM | BW |  |  |  |  |
| LD |  | IF | ID | EX | MEM | BW |  |  |  |
| ST |  | IF | ID | **STOP** | EX | MEM | BW |  |  |
| JMP |  | IF | ID | EX | MEM | BW |  |  |  |
| ADD |  | IF | ID | **STOP** | EX | MEM | BW |  |  |

## Comprobación funcional y temporal

Para comprobar el correcto funcionamiento de las nuevas características del procesador hemos realizado un código que lo compruebe. El código es el siguiente:

Memoria inicial:  
[#0] = 01  
[#1] = 02

|  |  |  |
| --- | --- | --- |
| INSTRUCCIONES | CÓDIGO | OPERACIÓN |
| LD R0,[#0] | 00 | R0 = 01 |
| LD R1,[#1] | 05 | R1 = 02 |
| ADD R0,R1 | C4 | R0 = 01+02=03 |
| ST R0,[#2] | 48 | [#2] = 03 |
| LD R2,[#2] | 0A | R2 = 03 |
| ADD R2,R0 | C2 | R2 = 03+03 =06 |
| ADD R2,R0 | C2 | R2 = 06+03 = 09 |
| LD R2,[#2] | 0A | R2 = 03 |
| ST R2,[#3] | 4E | [#3] =03 |
| NOP | E0 | NOP |
| ADD R0,R0 | C0 | R0 = 03+03 = 06 |
| JMP [#0] | 80 | SALTO A 0 |
| NOP | E0 | NOP |
| ST R0,[#4] | 50 | [#4] = 06 |

Memoria final:  
[#0] = 01  
[#1] = 02  
[#2] = 03  
[#3] = 03  
[#4] = 06

Hemos realizado otro código para comprobar el tiempo del reloj, centrándonos en la etapa EX:

Memoria inicial:  
[#1] = 01  
[#2] = FF

|  |  |  |
| --- | --- | --- |
| INSTRUCCIONES | CÓDIGO | OPERACIÓN |
| LD R1,[#1] | 05 | R1 = 01 |
| LD R2,[#2] | 0A | R2 = FF |
| ADD R1,R2 | C9 | R1 = 01+FF=00 |
| ADD R1,R2 | C9 | R1 = 00+FF=FF |
| ST R1,[#3] | 4D | [#3] = FF |

Memoria inicial:  
[#1] = 01  
[#2] = FF  
[#3] = FF

# Speedup

Mirando el código de la práctica 2 se pueden realizar modificaciones en su orden:

|  |  |
| --- | --- |
| CÓDIGO PASO 3 | **CÓDIGO CORTOCIRCUITO** |
| LDI R0,[#0] (0) | LDI R0,[#0] (0) |
| LDI R1,[#1] (4) | LDI R1,[#1] (4) |
| LDI R2,[#2] (5) | LDI R2,[#2] (5) |
| NOP | ADD RO,RO (1) |
| ADD RO,RO (1) | ADD R2,R1 (6) |
| NOP | ADD RO,RO (2) |
| ADD R2,R1 (6) | JMP #0 (8) |
| NOP | STI R2,[#4] (7) |
| ADD RO,RO (2) | STI R0,[#3] (3) |
| NOP | X |
| JMP #0 (8) | X |
| STI R2,[#4] (7) | X |
| STI R0,[#3] (3) | X |

Si reordenamos así el código, podremos ayudarnos de los cortocircuitos. Sin embargo, la unidad de detención no es usada en esta modificación.

Speedup = = = = 1,006