



## Graduado en Ingeniería Informática

Universidad Politécnica de Madrid

Escuela Técnica Superior de  
Ingenieros Informáticos

### Trabajo fin de Grado

Un Modelo simplificado del procesador 68000 en VHDL

Autor: Salvador González García

Tutor: Mariano Hermida de la Rica

MADRID, ENERO 2017



# Índice

RESUMEN .....	6
ABSTRACT .....	7
1. INTRODUCCIÓN .....	8
2. TRABAJOS PREVIOS .....	9
2.1 VHDL y ModelSim .....	9
2.2 PicoComputador .....	9
2.2.1 Instrucciones.....	9
2.2.2 Arquitectura .....	10
2.3 Low Risk Computer.....	11
2.3.1 Arquitectura .....	11
2.3.2 Instrucciones.....	12
2.3.3 Estructura de los procesos .....	13
3. DESARROLLO DEL MODELO .....	15
3.1 Diseño de las instrucciones.....	15
3.1.1 MOVE .....	15
3.1.2 ADD .....	15
3.1.3 CMP .....	15
3.1.4 AND .....	16
3.1.5 OR .....	16
3.1.6 SUB .....	16
3.1.7 RTS.....	17
3.1.8 RTE .....	17
3.1.9 LINK .....	17
3.1.10 UNLK.....	17
3.1.11 BSR .....	17
3.1.12 BRA.....	18
3.1.13 STOP .....	18
3.1.14 BCC.....	18
3.1.15 NOP.....	19
3.2 Diseño y arquitectura del 68000 .....	19
3.2.1 Interrupciones .....	20
3.2.2 Direccionamientos .....	20

3.3	Módulos implementados .....	21
3.3.1	ALU .....	21
3.3.2	Módulo auxiliar .....	22
3.3.3	Buffer triestado .....	23
3.3.4	Registro de estado.....	23
3.3.5	Unidad de control .....	23
3.3.6	CPU2017 .....	51
3.3.7	Memoria .....	51
3.3.8	Registro de instrucción .....	52
3.3.9	Registro de dirección de memoria .....	53
3.3.10	Contador de programa .....	53
3.3.11	Registro auxiliar .....	53
3.3.12	Banco de registros .....	54
3.3.13	Test .....	55
4.	VERIFICACIÓN DEL MODELO .....	56
4.1	Move.txt.....	56
4.2	Add.txt .....	57
4.3	Bcc.txt .....	58
4.4	Bra.txt .....	60
4.5	Link.txt.....	61
4.6	Sub.txt .....	63
5.	ENSAMBLADOR .....	67
6.	CONCLUSIÓN .....	68
7.	BIBLIOGRAFÍA.....	69



## **RESUMEN**

Se realiza un modelo simplificado del procesador 68000 en el lenguaje de descripción de hardware VHDL. Para ello, se diseña una arquitectura simplificada del procesador que permita realizar una implementación del mismo y que pueda ejecutar las instrucciones de su juego de instrucciones. Todos los subsistemas están simplificados respecto de la documentación disponible de Motorola. Entre los subsistemas estudiados e implementados se encuentran la unidad de control, la ALU, el banco de registros, el contador de programa, el registro de estado, así como una memoria externa sencilla y otros módulos auxiliares que sirven para simular el comportamiento de las señales internas del procesador con el programa ModelSim. Para testear el funcionamiento del procesador, se realizará un ensamblador sencillo capaz de convertir en código máquina programas escritos en el lenguaje ensamblador del 68000.

## **ABSTRACT**

A simplified model for the Motorola 68000 processor is made, coded in the hardware description language VHDL. A simplified architecture for the processor is designed in order to make an implementation of the 68000, as well as an instruction set architecture capable of executing instructions sequentially. All subsystems are simplified versions of the original processor, based on Motorola reference documentation. Control unit, register bank, program counter, state register, as well as a simple external memory and other auxiliary modules are among the subsystems examined and implemented. Internal processor signals are simulated using ModelSim. A simple assembler is made for executing programs written in assembly language for the 68000 processor.

## **1. INTRODUCCIÓN**

El objetivo general de este trabajo es realizar un modelo simplificado del procesador 68000 en VHDL. Para ello, es necesario conocer previamente la estructura de los computadores, así como el lenguaje VHDL que permite describir circuitos digitales. Además, se necesitará leer, estudiar y analizar la documentación de los procesadores elegidos poder entender el funcionamiento de sus unidades de control. Para la realización del trabajo, se utilizarán como ejemplos unidades de control de procesadores más sencillos, para luego continuar con el procesador 68000, el cual se describirá con un juego de instrucciones más pequeño y se realizará un ensamblador para el mismo.

El 68000 es un procesador CISC introducido en el año 1979 diseñado y comercializado por Motorola Semiconductor Products Sector. Fue muy popular comercialmente y se sigue usando en la actualidad. Tiene bus de datos interno de 32 bits y registros de 32 bits, pero con ALU de 16 bits y buses externos de 16 bits [1][2][3]. El compilador de VHDL que se utilizará para realizar el modelo será ModelSim de Mentor Graphics, con capacidad para simular diseños FPGA.

El procesador 68000 cuenta con diversos modos de direccionamiento y un juego de instrucciones muy amplio, lo que hace que sea muy ortogonal. Toda la información relacionada con este procesador viene en el manual de usuario y el manual de referencia del programador, incluyendo la sintaxis y el diseño de todas las instrucciones del juego de instrucciones de la arquitectura.

Se realizará un diseño básico y esquemático de la arquitectura del procesador 68000 para crear el juego de instrucciones (ISA) del computador. Cada instrucción en ensamblador realiza una serie de acciones que se deben corresponder con el diseño propuesto de la máquina. Estas instrucciones, a la hora de implementarlas en VHDL, seguirán los pasos que se han pensado previamente sobre la arquitectura. El procesador 68000, al ser una máquina de tipo CISC, tiene un ISA muy complejo y no es posible realizar todas las instrucciones en el tiempo que dura la realización de este trabajo, por lo que se implementará un juego reducido de instrucciones, concretamente de la unidad de enteros, dejando para posibles futuros trabajos las instrucciones de coma flotante o de supervisor.

## 2. TRABAJOS PREVIOS

Para entender el funcionamiento de una CPU se estudian dos ejemplos didácticos más simples. El PicoComputador es una máquina ya implementada y se estudia cómo está diseñada. El LowRisk Computer se implementa para comprobar que se ha entendido lo estudiado hasta ahora, además de servir como base para realizar la implementación del procesador 68000, adaptando a las características de esta máquina.

### 2.1 VHDL y ModelSim

Antes de empezar a estudiar el PicoComputador y el LowRisk Computer se repasa el lenguaje VHDL para ser capaz de leer y programar con cierta soltura, y se aprende a utilizar el compilador ModelSim para poder compilar y simular los procesadores estudiados y diseñados. Estos dos elementos son las herramientas básicas para realizar este trabajo.

### 2.2 PicoComputador

El PicoComputador [4] es una máquina muy sencilla de una sola dirección que cuenta con un registro acumulador que se usa en las instrucciones con 2 operandos, en el que un operando es dicho registro acumulador y el otro operando está situado siempre en memoria. El resultado se deja siempre en el registro acumulador. El bus de datos es de 16 bits, direccionable a nivel de palabra, por lo que no se puede acceder a bytes individuales. El juego de instrucciones es reducido, con 8 instrucciones. El formato de las instrucciones siempre será el mismo, en este caso, 3 bits para el código de operación y 13 bits restantes para indicar la posición de memoria o el dato inmediato. Son 13 bits de campo de dirección, por lo que  $2^{13} = 8192$  que equivalen a 8K palabras diferentes y el bus de direcciones será también de 13 bits.

#### 2.2.1 Instrucciones

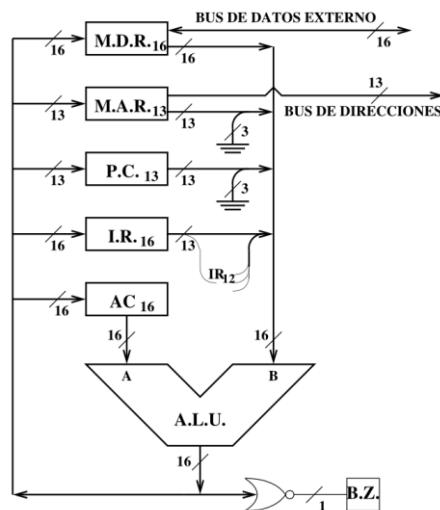
Las 8 instrucciones estudiadas son:

- LDA Dirección. Carga en el registro acumulador el contenido de la dirección de memoria indicada en la instrucción.
- STA Dirección. Almacena el contenido del registro acumulador en la zona de memoria indicada por la dirección que viene en la instrucción.

- SUM Dirección. Realiza una suma en complemento a 2 almacenando el resultado en el registro acumulador y utilizando los operandos de la dirección de memoria proporcionada y el registro acumulador.
- NOR Dirección. Realiza una operación lógica NOR bit a bit con el operando que está en memoria y el dato que está en el registro acumulador, guardando el resultado en dicho registro.
- JNZ Dirección. Realiza un salto condicional, si el resultado de la última operación no fue cero, a la dirección indicada en la instrucción.
- JZ Dirección. Realiza un salto condicional, si el resultado de la última operación fue cero, a la dirección indicada en la instrucción. Si se combina esta instrucción con JZ se puede hacer un salto incodicional.
- LDI Operando. Carga en el registro acumulador el operando inmediato que viene en los 13 bits de operando de la instrucción.
- SUMI Operando. Realiza una suma utilizando el operando que viene en los 13 bits de la instrucción con el contenido del registro acumulador, almacenando el resultado en dicho registro.

### 2.2.2 Arquitectura

La arquitectura del PicoComputador se puede ver en la siguiente figura:



El PicoComputador dispone de cinco registros, el biestable de cero y la ALU. Los registros son el Registro de datos de memoria, Registro de direcciones de memoria, PC, registro de direcciones y el Registro Acumulador. Cada elemento del computador tiene su módulo en VHDL. Los módulos VHDL son alu, alu\_MC, biestable\_Z, memoria, picoComputador, registro\_AC, registro\_IR, registro\_MAR, registro\_MDR, registro\_PC, test y unidadControl.

El registro de direcciones de memoria y el contador de programa solo trabajan con direcciones, por lo que solo necesitan tener 13 bits, mientras que el registro de datos de memoria y el registro acumulador tienen ambos 16 bits porque manejan datos. El registro de instrucción, al ser las instrucciones de 16 bits, también es de 16 bits.

La ALU tiene 2 entradas. Una de ellas está siempre conectada al registro acumulador, la otra puede venir de los otros 4 registros. La forma de ordenar que se active cierto registro como entrada para la ALU es activando la puerta triestado que tiene cada registro. La salida de la ALU puede guardar el resultado de la operación en cualquiera de los cinco registros, activando la señal de carga del registro en cuestión.

El módulo que implementa la unidad de control tiene las señales que controlan el comportamiento del computador a la hora de ejecutar instrucciones. A la hora de implementar este módulo hay que tener en cuenta los diferentes tipos de señales que se pueden presentar:

- Las señales de carga de los registros permiten que los registros puedan recoger la información presente en el bus de salida de la ALU.
- Las señales que activan los registros para volcar su contenido en el bus.
- Señales de control de la ALU, que indican que deben activar la ALU o cambiar la operación que se va a realizar en la ALU.
- Señales de control de la memoria, que indican si se va a realizar una lectura, una escritura, o no se va a realizar nada.

Para evitar los conflictos en los buses, es necesario tener diferentes estados que permitan realizar las operaciones necesarias sin solapar la información que pasa por los buses.

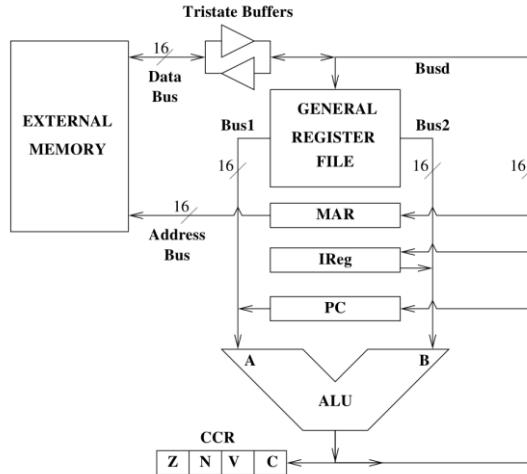
## 2.3 Low Risk Computer

Habiendo entendido fundamentalmente cómo funciona el PicoComputador, se pasa a implementar la unidad de control de una máquina más compleja, el LowRisk Computer, de tipo RISC. Cuenta con un banco de 8 registros de propósito general, arquitectura de tipo Load/Store, palabras y buses de 16 bits y direccionamiento a nivel de palabra. En las instrucciones, de 16 bits, puede haber 3 operandos con registros: 2 registros fuente y 1 registro destino [5].

### 2.3.1 Arquitectura

El banco de registros tiene una entrada, que será el busd, y 2 salidas, una para cada registro fuente, que serán el bus1 y el bus2 y que están conectados a las entradas A y B de la ALU. El Registro R0 está cableado a 0 siempre.

La arquitectura de la máquina se puede ver en la siguiente figura:



Elementos que se pueden observar son el contador de programa, el registro de instrucción, el registro de dirección de memoria, y el Registro condition code (CCR). Este último registro lo que hace es almacenar las características de los resultados realizados por la ALU. Los posibles valores que almacena el registro CCR son si el resultado de la última operación ha sido cero, negativo, desbordamiento o si hay acarreo.

La información que va hacia la memoria o viene de la memoria pasa por puertas triestado que deben ser activadas (solamente una de las 2) para dejar pasar la información por el busd o el Data bus.

Las instrucciones son de 16 bits y están formadas por: 3 bits de código de operación, 3 bits de registro de destino, 3 bits de registro fuente1, 3 bits de registro fuente2 y 4 bits de extensión de código de operación. Las instrucciones tienen su código de operación, pero se puede ampliar su rango de funciones mediante esta extensión del código de operación.

### 2.3.2 Instrucciones

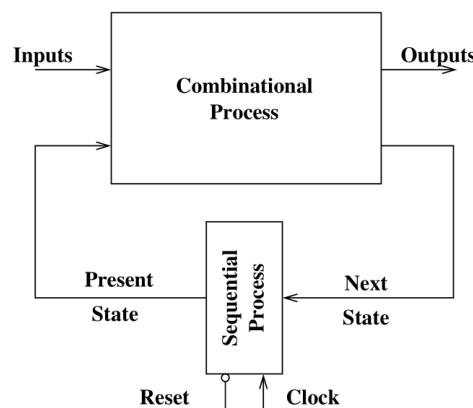
Las instrucciones de esta máquina son:

- LOAD. Código de operación: 001. Carga el contenido de la posición de memoria según la suma de los 2 registros fuente
- STORE. Código de operación: 010. Almacena en la dirección de memoria compuesta por la suma de los 2 registros fuente.
- ADD. Código de operación: 011. Suma los 2 registros fuente y los almacena en el registro destino. Con el código de extensión la operación puede cambiar. En concreto, invertir el segundo operando, activar el CCR en la operación para que cambie este registro, usar el acarreo para la operación de la ALU o complementar el acarreo.

- OPLOG. Código de operación: 100. Realiza una operación bit a bit entre los 2 registros fuente y almacena el resultado en el registro destino. El tipo de operación según la extensión del código de operación puede ser AND, OR, NOR o XOR.
- JMPC. Código de operación: 101. Realiza un salto condicional, guardando previamente el contador de programa en el registro destino indicado. El nuevo PC es la suma de los 2 registros fuente. Por supuesto, el salto se realiza si se cumple la condición de salto (indicada en la extensión de la instrucción). Comprueba el registro CCR para ver si se cumple la condición.
- BRANCH. Código de operación: 110. Hace un salto sumando al PC un desplazamiento indicado en los bits 12 al 4 de la instrucción. La condición de salto viene indicada, al igual que en JMPC, en la extensión de la instrucción (bits 3 a 0).
- OPI. Código de operación: 111. Guarda el operando inmediato en un registro destino. Esta instrucción es importante porque es necesaria para que otras instrucciones usen registros con datos. Según la extensión, esta instrucción puede cargar en el registro en los bits inferiores llenando de 0 la parte superior, o bien cargar en los bits inferiores extendiendo el signo a la parte superior, o llenando con ceros el byte inferior, o bien sumar extendiendo el signo del operando inmediato.
- STOP. Código de operación: 000. Esta instrucción simplemente detiene la máquina, pasando al estado ‘halted’.

### 2.3.3 Estructura de los procesos

La estructura general de esta máquina tiene 2 procesos. Un proceso secuencial que tiene las señales clock, que se activa en el flanco de subida, y reset, que se activa a nivel bajo, y que sirve para actualizar al estado futuro de la máquina. El otro proceso es combinacional y genera todas las señales de control y la señal de estado futuro.



La unidad de control tiene 8 estados posibles, que son Initial, ReadInst, Execute, ReadData, WriteData, GoTo, Halted y Error. El estado Initial se usa como fase de fetch para empezar a obtener la instrucción. El estado ReadInst se utiliza para leer la instrucción

de memoria y llevarla al registro de instrucción. El estado Execute ejecuta la instrucción como tal, leída de memoria. Los estados ReadData y WriteData se usan para leer o escribir datos en memoria, según pida la instrucción. El estado GoTo se usa en los saltos, ya sea Branch o Jump. El estado Halted detiene la máquina y no sale nunca de ese estado, hasta que se reinicie la máquina. Finalmente, el estado Error denota que ha habido un error y, al igual que Halted, no transicionará a otros estados.

Los estados son necesarios para evitar conflictos en los buses y registros, debido a la temporización concurrente de las señales y sus retardos. Por ejemplo, el busd requiere de ser usado varias veces al ejecutar una instrucción que lea un dato de memoria, pues hay que acceder a la memoria para obtener la instrucción y para obtener el dato, es decir, 2 accesos de lectura.

Utilizando esta información que proporciona la documentación de la máquina se implementa la unidad de control del Low Risk Computer. A su vez, esta implementación de la unidad de control será la base para realizar la del procesador 68000, adaptando el diseño del Low Risk a las especificaciones y características del 6800

### 3. DESARROLLO DEL MODELO

#### 3.1 Diseño de las instrucciones

Para el diseño de las instrucciones, se ha elegido un subconjunto de la unidad de enteros como simplificación del modelo.

A continuación, se describe el formato, estructura y uso de cada código de operación.

##### 3.1.1 MOVE

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	SIZE		REGISTER	DESTINATION	MODE		MODE	MODE	SOURCE		REGISTER			

Operación: Fuente → Destino

Donde size puede ser 01, 11 y 10. Sin embargo, al solo haber un tamaño de instrucción, los 3 modos son equivalentes.

El operando destino y el operando fuente tienen la misma estructura. 3 bits para elegir el registro, y 3 bits para elegir el modo de direccionamiento.

##### 3.1.2 ADD

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	1	REGISTER	OPMODE		MODE	EFFECTIVE ADDRESS		REGISTER					

Operación: Fuente + Destino → Destino

Por simplificar el modelo, los dos modos de direccionamiento de los operandos fuente y destino siguen la misma estructura que la instrucción MOVE.

El operando destino y el operando fuente tienen la misma estructura. 3 bits para elegir el registro, y 3 bits para elegir el modo de direccionamiento.

##### 3.1.3 CMP

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	1	1	REGISTER	OPMODE		MODE	EFFECTIVE ADDRESS		REGISTER					

Operación: Destino – Fuente → cc

Por simplificar el modelo, los dos modos de direccionamiento de los operandos fuente y destino siguen la misma estructura que la instrucción MOVE.

El operando destino y el operando fuente tienen la misma estructura. 3 bits para elegir el registro, y 3 bits para elegir el modo de direccionamiento.

### 3.1.4 AND

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	0	REGISTER		OPMODE						EFFECTIVE ADDRESS			
												MODE		REGISTER	

Fuente AND Destino → Destino

Por simplificar el modelo, los dos modos de direccionamiento de los operandos fuente y destino siguen la misma estructura que la instrucción MOVE.

El operando destino y el operando fuente tienen la misma estructura. 3 bits para elegir el registro, y 3 bits para elegir el modo de direccionamiento.

### 3.1.5 OR

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	0	0	REGISTER		OPMODE						EFFECTIVE ADDRESS			
												MODE		REGISTER	

Fuente OR Destino → Destino

Por simplificar el modelo, los dos modos de direccionamiento de los operandos fuente y destino siguen la misma estructura que la instrucción MOVE.

El operando destino y el operando fuente tienen la misma estructura. 3 bits para elegir el registro, y 3 bits para elegir el modo de direccionamiento.

### 3.1.6 SUB

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	0	1	REGISTER		OPMODE						EFFECTIVE ADDRESS			
												MODE		REGISTER	

Operación: Destino – Fuente → Destino

Por simplificar el modelo, los dos modos de direccionamiento de los operandos fuente y destino siguen la misma estructura que la instrucción MOVE.

El operando destino y el operando fuente tienen la misma estructura. 3 bits para elegir el registro, y 3 bits para elegir el modo de direccionamiento.

Para realizar la resta, se utiliza la operación de suma en la ALU, pero proporcionando un acarreo inicial de 1.

### 3.1.7 RTS

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	1	1	1	0	0	1	1	1	0	1	0	1

Operación:  $(SP) \rightarrow PC; SP + 1 \rightarrow SP$

Retorna de subrutina restaurando el PC de la pila.

### 3.1.8 RTE

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	1	1	1	0	0	1	1	1	0	0	1	1

Operación:  $(SP) \rightarrow SR; SP + 1 \rightarrow SP; (SP) \rightarrow PC; SP + 1 \rightarrow SP;$

Retorna de la rutina de tratamiento de interrupción, restaura el registro de estado y el PC.

### 3.1.9 LINK

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	1	1	1	0	0	1	0	1	0	1	0	REGISTER

WORD DISPLACEMENT

Operación:  $SP - 1 \rightarrow SP; An \rightarrow (SP); SP \rightarrow AN; SP + dn \rightarrow SP$

Crea el marco de pila en el registro de dirección indicado en los bits 2 a 0, con espacio para variables según el desplazamiento, indicado en la segunda palabra de la instrucción.

### 3.1.10 UNLK

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	1	1	1	0	0	1	0	1	1	0	REGISTER	

Operación:  $An \rightarrow SP; (SP) \rightarrow An; SP + 1 \rightarrow SP$

Deshace el puntero de marco de pila del registro de dirección elegido en los bits 2 a 0.

### 3.1.11 BSR

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	0	0	0	0	1								8-BIT DISPLACEMENT
16-BIT DISPLACEMENT IF 8-BIT DISPLACEMENT = \$00															
32-BIT DISPLACEMENT IF 8-BIT DISPLACEMENT = \$FF															

Operación: SP - 1 → SP; PC → (SP); PC + dn → PC

Realiza un salto a subrutina con retorno, salvaguardando el PC. Si el desplazamiento es de 8 bits, entonces va en la misma palabra que la instrucción. Si los 8 bits inferiores son todos 0 o todos 1, entonces se considera que el desplazamiento va en la próxima palabra.

### 3.1.12 BRA

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
8-BIT DISPLACEMENT															
16-BIT DISPLACEMENT IF 8-BIT DISPLACEMENT = \$00															
32-BIT DISPLACEMENT IF 8-BIT DISPLACEMENT = \$FF															

Operación: PC + dn → PC

Realiza un salto incondicional. Si el desplazamiento es de 8 bits, entonces va en la misma palabra que la instrucción. Si los 8 bits inferiores son todos 0 o todos 1, entonces se considera que el desplazamiento va en la próxima palabra.

### 3.1.13 STOP

0000000000000000

Operación: detiene el procesador.

La instrucción STOP no realiza estas acciones en este procesador, pero se ha implementado para detener el programa en la simulación de ModelSim,

### 3.1.14 BCC

Mnemonic	Condition
CC(HI)	Carry Clear
CS(LO)	Carry Set
EQ	Equal
GE	Greater or Equal
GT	Greater Than
HI	High
LE	Less or Equal

Mnemonic	Condition
LS	Low or Same
LT	Less Than
MI	Minus
NE	Not Equal
PL	Plus
VC	Overflow Clear
VS	Overflow Set

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
CONDITION															
8-BIT DISPLACEMENT															
16-BIT DISPLACEMENT IF 8-BIT DISPLACEMENT = \$00															
32-BIT DISPLACEMENT IF 8-BIT DISPLACEMENT = \$FF															

Operación: Si condición TRUE

Then  $PC + dn \rightarrow PC$

Realiza un salto si se cumple la condición. Los códigos de condición vienen en la tabla superior.

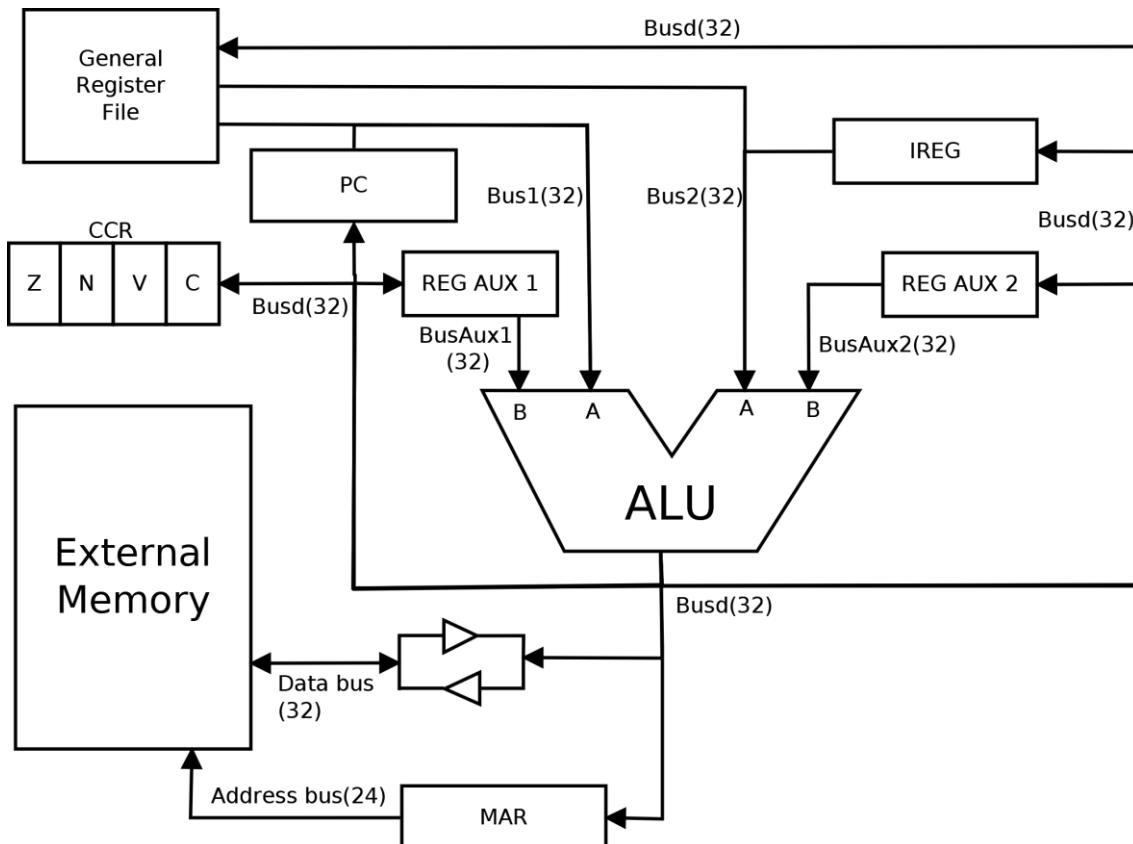
### 3.1.15 NOP

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	1	1	1	0	0	1	1	1	0	0	0	1

Operación: ninguna

## 3.2 Diseño y arquitectura del 68000

El procesador 68000 tiene una gran complejidad interna, y en un solo trabajo difícilmente se podría modelar el procesador en su totalidad. Por ello, la arquitectura del sistema se ha simplificado de tal forma que las instrucciones elegidas y sus funciones sean equivalentes. La arquitectura del sistema es la siguiente:



Los módulos de la arquitectura son:

- External Memory (memoria externa). La memoria no pertenece al procesador.
- ALU (Unidad aritmético-lógica).
- Bufs3state (Buffer triestado).
- Condition Code Register (Registro de estado).
- Control Unit (Unidad de control).
- Instruction Register (Registro de instrucción).
- MAR (Registro de dirección de memoria).
- PC Unit (Contador de programa).
- RegAux (Registros auxiliares).
- Register File (Registros de propósito general).

### **3.2.1 Interrupciones**

El procesador 68000 cuenta con un sistema de prioridad de interrupciones de 7 niveles. Así mismo, el registro de estado cuenta con 3 bits para enmascarar las interrupciones.

En este trabajo las interrupciones se han simplificado, con una única señal de interrupción en la unidad de control y sin máscara en el registro de estado. Una vez llega una interrupción activando la señal Intr con valor ‘1’, el procesador termina de ejecutar la instrucción actual y una vez que termina, en el estado Initial, comprueba si la señal de interrupción está activada. Si está activada, comienza el proceso para tratar la interrupción. Salvaguarda el contador de programa y el registro de estado en la pila y modifica el PC para ejecutar la rutina de tratamiento de interrupción, cuya dirección es siempre la dirección 32 decimal. (En los ficheros de programa, la línea 32 pasará a ser la rutina de tratamiento de interrupción). Una vez ejecutadas todas las instrucciones de la rutina de tratamiento de interrupción, se procede a ejecutar la instrucción RTE, que sirve para retornar de una interrupción, restaurando el PC (cambiando la operación de PC a “cargar PC”) y el registro de estado mediante la señal LoadSR, que estaban guardados en la pila. [6]

### **3.2.2 Direccionamientos**

El procesador cuenta con diversos modos de direccionamiento para los operandos fuente y destino.

Los direccionamientos implementados en este procesador para los operandos fuente son:

- Direccionamiento directo a registro de datos.
  - Código de direccionamiento: 000

- Direccionamiento directo a registro de dirección
  - Código de direccionamiento: 001
- Direccionamiento indirecto a registro
  - Código de direccionamiento: 010
- Direccionamiento indirecto a registro con postincremento
  - Código de direccionamiento: 011
- Direccionamiento indirecto a registro con predecremento
  - Código de direccionamiento: 100
- Direccionamiento inmediato
  - Código de direccionamiento: 111
- Direccionamiento relativo a registro base
  - Código de direccionamiento: 101

Los direccionamientos implementados en este procesador para los operandos destino son:

- Direccionamiento directo a registro de datos.
  - Código de direccionamiento: 000
- Direccionamiento directo a registro de dirección
  - Código de direccionamiento: 001
- Direccionamiento indirecto a registro
  - Código de direccionamiento: 010
- Direccionamiento indirecto a registro con postincremento
  - Código de direccionamiento: 011
- Direccionamiento indirecto a registro con predecremento
  - Código de direccionamiento: 100
- Direccionamiento relativo a registro base
  - Código de direccionamiento: 101

El direccionamiento inmediato no está disponible en operandos destino. La información sobre los direccionamientos disponibles de cada instrucción está disponible en la documentación del manual del programador del procesador [3].

### **3.3 Módulos implementados**

En esta sección se describen los módulos implementados en este trabajo. La mayoría de módulos están basados en el trabajo hecho en el Low Risk Computer pero modificados. La unidad de control y todos los estados han sido creados desde cero.

#### **3.3.1 ALU**

Fichero: ALU.vhd

La ALU está basada en la ALU del Low Risk Computer. Cuenta con 2 señales de selección de entrada de la ALU, para elegir entre el banco de registro o los registros auxiliares. AluInputASel con valor 0 elige de entrada el bus1, conectado al banco de registros. AluInputBSel con valor 0 elige de entrada a la ALU el bus2, conectado al banco de registros. Con valor 1 elige de entrada el busAux1, si se trata AluInputASel o AluInputBSel.

Una vez elegidas los buses de entrada (bus1/bus2 o busAux1 busAux2), se le hace un xor a la segunda entrada de la ALU, se asigna el carry que viene de la unidad de control y se comprueba el valor de ALUcontrol para saber qué operación de la ALU realizar.

La principal simplificación de la ALU del procesador 68000 es que es de 32 bits, en lugar de 16 bits como se describe en el manual. Además, al simplificar los tamaños de media palabra, palabra y doble palabra, y con direccionamiento a nivel de palabra, solamente hay una operación de incremento y una operación de decremento, sumando o restando 1.

A continuación, se explican las operaciones de la ALU que se usan en este procesador:

**PASS\_A:** Se asigna a la salida de la ALU la primera entrada sin modificar, de tal forma que “pasa” por la ALU. Pone el registro de estado sin actualizar a 0000.

**PASS\_B:** Se asigna a la salida de la ALU la segunda entrada sin modificar, de tal forma que “pasa” por la ALU. Pone el registro de estado sin actualizar a 0000.

**INCR\_WORD:** Incrementa la palabra en 1. Esta operación es la misma que sumar, pero en este caso la segunda entrada de la ALU tiene como dato un 1. La suma que se realiza es en binario. Asigna el resultado a la salida.

**DECR\_WORD:** Decrementa la palabra en 1. Esta operación es la misma que restar, pero en este caso la segunda entrada de la ALU tiene como dato un 1. La resta que se realiza es en binario. Asigna el resultado a la salida.

**ALU\_ADD:** Realiza una suma o una resta binaria en complemento a dos, dependiendo del acarreo inicial y modificando el registro de estado sin actualizar y lo asigna a la salida de la ALU.

**ALU\_AND:** Esta operación realiza un *and* bit a bit y lo asigna a la salida de la ALU.

**ALU\_OR:** Esta operación realiza un *and* bit a bit y lo asigna a la salida de la ALU.

### 3.3.2 Módulo auxiliar

Fichero: Auxiliary.vhd

El módulo auxiliar contiene una serie de constantes que facilitan el proceso de desarrollo de la CPU, así como una mayor legibilidad del código. Algunas de estas constantes y tipos son las operaciones de la ALU, el banco de registros, la constante de alta impedancia, constantes de todo a 1 y las operaciones sobre el PC.

### 3.3.3 Buffer triestado

Fichero: Bufs3state.vhd

El buffer triestado permite el paso de datos de la memoria a la CPU y de la CPU a la memoria. El paso se permite mediante las señales de control DriveCPU y DriveMem.

Si está activada la señal DriveCPU a 1, asigna el busd al bus de datos, si es 0, pone el busd en alta impedancia.

Si está activada la señal DriveMem a 1, asigna el contenido del Busd en el bus de datos. Si es 0, pone el bus de datos en alta impedancia.

### 3.3.4 Registro de estado

Fichero: CondCodeRegister.vhd

El registro de estado está conectado al Busd y cuenta con 4 bits de estado:

- Z: determina si es el resultado ha sido cero
- N: determina si el resultado ha sido negativo.
- V: determina si ha habido overflow/desbordamiento.
- C: determina si ha habido acarreo en la operación.

El registro de estado se actualiza desde la ALU cada vez que se activa a ‘1’ la señal LoadCCR en el flanco de subida de reloj. En el caso de querer restaurar el registro de estado (al volver de una interrupción), se activa a ‘1’ la señal LoadSR en el flanco de subida de reloj y los primeros 4 bits del busd se asignan al registro de estado ZNVC.

### 3.3.5 Unidad de control

Fichero: Control.vhd

La unidad de control es el módulo responsable de dirigir el estado y el funcionamiento del procesador. Todas las señales de control están situadas en este módulo. Se proporciona una breve descripción para cada señal interna de la unidad de control.

A continuación, se describen todas las señales usadas en la unidad de control. Para que estas señales estén integradas en todo el procesador, hay que hacer el port mapping en el módulo de CPU2017, asignando las señales a sus respectivos módulos para que los módulos que usen estas señales tengan acceso a ellas.

Señales de la unidad de control:

**AluInputASel:** Esta señal está conectada a la ALU. Sirve para elegir si la entrada A de la ALU está conectada al registro auxiliar1 mediante el busAux1 o al banco de registros mediante el bus 1. Actúa como un multiplexor de selector de entrada. Una vez asignada a la señal el valor ‘0’ (banco) o ‘1’ (auxiliar), el módulo de la ALU actúa en consecuencia cambiando la entrada.

**AluInputBSel:** Esta señal está conectada a la ALU. Sirve para elegir si la entrada B de la ALU está conectada al registro auxiliar2 mediante el busAux2 o al banco de registros mediante el bus 2. Actúa como un multiplexor de selector de entrada.

**ALUcontrol:** Esta señal está conectada a la ALU. Sirve para elegir la operación de la ALU que se va a llevar a cabo. Estas operaciones están definidas en el módulo de la ALU. Los nombres asignados a las constantes de las operaciones están definidos en el módulo auxiliar.

**Cin:** Esta señal está conectada a la ALU. Es el carry de entrada que recibe la ALU. En general, será ‘0’ para la suma y ‘1’ para la resta. Se deberá asignar su valor en un estado de la unidad de control al realizar una operación de la ALU.

**EnableALU:** Activa la salida de la ALU. Está conectada al busd, que es un bus compartido. Si se activa con valor ‘1’, el valor del busd pasa a ser la salida que haya asignado la ALU. Si está desactivada, el valor de busd pasa a estar en alta impedancia (Todos los bits a Z).

**LoadAux1:** Esta señal pertenece al módulo de registro auxiliar. Cuando esta señal está activa, carga el contenido del busd en el registro auxiliar1. El registro auxiliar1 tiene como entrada el busd y como salida el busAux1. Este registro se usa como almacenamiento temporal para calcular los direccionamientos de los operandos de la instrucción que se está ejecutando.

**LoadAux2:** Esta señal pertenece al módulo de registro auxiliar. Cuando esta señal está activa, carga el contenido del busd en el registro auxiliar1. El registro auxiliar2 tiene como entrada el busd y como salida el busAux2. Este registro se usa como almacenamiento temporal para calcular los direccionamientos de los operandos de la instrucción que se está ejecutando.

**EnableAux1:** Esta señal pertenece al módulo de registro auxiliar. Cuando esta señal está activa, vuela el contenido del registro auxiliar 1 en el busAux1, el cual está conectado a la primera entrada B de la ALU. Para poder usar el contenido de este registro en la ALU

hay que seleccionar esta entrada con la señal AluInputASel. Cuando está desactivada, el busAux1 está en alta impedancia (todos los bits a ‘Z’).

**EnableAux2:** Esta señal pertenece al módulo de registro auxiliar. Cuando esta señal está activa, vuelca el contenido del registro auxiliar 2 en el busAux2, el cual está conectado a la segunda entrada B de la ALU. Para poder usar el contenido de este registro en la ALU hay que seleccionar esta entrada con la señal AluInputBSel. Cuando está desactivada, el busAux2 está en alta impedancia (todos los bits a ‘Z’).

**RS1sel:** Esta señal pertenece al banco de registros. Sirve para seleccionar qué registro fuente se va a usar en la **primera** entrada A de la ALU. Los valores posibles van de “0000” a “1111”. Los registros de “0000” a “0111” son los registros de datos. Los registros de “1000” a “1111” son los registros de dirección. Hacen falta 2 señales más, además de esta, para volcar el contenido de un registro a la ALU, que son EnableRS1 activado a ‘1’ y AluInputASel a ‘0’ para seleccionar la primera entrada de la ALU con el banco de registros.

**RS2sel:** Esta señal pertenece al banco de registros. Sirve para seleccionar qué registro fuente se va a usar en la **segunda** entrada A de la ALU. Los valores posibles van de “0000” a “1111”. Los registros de “0000” a “0111” son los registros de datos. Los registros de “1000” a “1111” son los registros de dirección. Hacen falta 2 señales más, además de esta, para volcar el contenido de un registro a la ALU, que son EnableRS2 activado a ‘1’ y AluInputBSel a ‘0’ para seleccionar la primera entrada de la ALU con el banco de registros.

**RDSel:** Esta señal pertenece al banco de registros. sirve para seleccionar qué registro destino se va a usar para cargar en el banco de registros. Los valores posibles van de “0000” a “1111”. Los registros de “0000” a “0111” son los registros de datos. Los registros de “1000” a “1111” son los registros de dirección. Hace1 falta 1 señal más, además de esta, para activar la escritura en el banco de registro, que es LoadRD activado a ‘1’.

**EnableRS1:** Esta señal pertenece al banco de registros. Cuando esta señal está activa, vuelca el contenido del registro del banco de registros, seleccionado con la señal RS1Sel, en el bus1, el cual está conectado a la **primera** entrada A de la ALU. Para poder usar el contenido de este registro en la ALU hay que seleccionar esta entrada con la señal AluInputASel con valor ‘0’, además de seleccionar el registro con RS1Sel. Cuando está desactivada EnableRS1, el bus1 está en alta impedancia (todos los bits a ‘Z’).

**EnableRS2:** Esta señal pertenece al banco de registros. Cuando esta señal está activa, vuelca el contenido del registro del banco de registros, seleccionado con la señal RS2Sel, en el bus2, el cual está conectado a la **segunda** entrada A de la ALU. Para poder usar el contenido de este registro en la ALU hay que seleccionar esta entrada con la señal AluInputBSel con valor ‘0’, además de seleccionar el registro con RS2Sel. Cuando está desactivada EnableRS2, el bus2 está en alta impedancia (todos los bits a ‘Z’).

**LoadRD:** Esta señal pertenece al banco de registros. Cuando esta señal está activa con valor ‘1’, carga el contenido del busd en el banco de registros. Concretamente, en el **registro elegido por RDSel**. El banco de registros tiene como entrada el busd y como salidas el bus1 y el bus2, conectados a la primera y segunda entrada A de la ALU

mediante AluInputASel con valor ‘0’ y AluInputBSel con valor ‘0’, además del registro concreto seleccionado con RS1Sel y/o RS2Sel. Este banco de registros cuenta con registros de propósito general, incluyendo los registros A6 y A7 (marco de pila y pila).

**PCop:** Esta señal pertenece al módulo de contador de programa (PC). Sirve para elegir la operación que se va a realizar en el PC. Los valores pueden ir de “00” a “11”. Significando, en orden de menor a mayor, que la operación a realizar por el PC en el próximo estado o ciclo será: No hacer nada (“00”), Rutina de tratamiento de interrupción (“01”), Incrementar el PC de forma normal ( $PC \leftarrow PC + 1$ ) (“01”), Cargar al PC la dirección que se encuentre en el busd (“11”), equivalente a como si hubiese una señal LoadPC con valor ‘1’. Cada constante tiene asignada un nombre como etiqueta en el módulo auxiliar, para facilitar la modificación (o la no modificación) del PC, por lo que es preferible usar estas constantes a usar el valor numérico.

**EnablePC:** Esta señal pertenece al módulo de contador de programa (PC). Cuando esta señal está activa, vuelca el contenido del contador de programa en el bus1, el cual está conectado a la **primera** entrada A de la ALU. Para poder usar el contenido de este registro en la ALU hay que seleccionar esta entrada con la señal AluInputASel con valor ‘0’. Cuando está desactivada EnablePC, el bus1 está en alta impedancia (todos los bits a ‘Z’).

**ZNVC:** Esta señal pertenece al Condition Code Register (CCR). Se trata del registro de estado. La Z es 1 si ha sido cero. La N es 1 si el resultado ha sido Negativo. V es 1 en caso de overflow/desbordamiento y C es 1 en caso de acarreo/Carry. El registro ZNVC **no se modifica en la unidad de control**, para ello tenemos otra señal que es LoadSR. Desde la unidad de control solo se puede acceder a este registro para consultar su valor directamente, **nada de volcarlo en busd**, para eso estaría en este caso la señal EnableSR. Por ejemplo, si queremos consultar si el valor de la última operación ha sido cero, basta con poner ZNVC(3) en la expresión que nos haga falta consultar el registro de estado.

**Intr:** Esta señal pertenece a la unidad de control. Si su valor es ‘1’ **indica que hay una interrupción**. La única forma de que esta señal sea 1 es forzando desde ModelSim que su valor sea ‘1’. Por defecto, al iniciar la ejecución del procesador, su valor es siempre 0. Al estar la señal de interrupción activa, la próxima instrucción a ejecutar será aplazada para tratar la interrupción, yendo directamente a la rutina de tratamiento de interrupción, definida en la dirección de memoria 32 decimal.

**LoadCCR:** Esta señal pertenece al Condition Code Register (CCR). Modifica el registro de estado con la última operación de la ALU ejecutada. Esta señal se usa en la unidad de control y se activa cuando su valor es 1.

**LoadSR:** Esta señal pertenece al Condition Code Register (CCR) y se usa desde la unidad de control. Carga el registro de estado ZNVC con el contenido del busd **para restaurar el registro** de estado después de tratar una interrupción. Se usa en la instrucción RTE.

**EnableSR:** Esta señal pertenece al Condition Code Register (CCR) y se usa desde la unidad de control. Sirve para volcar el contenido del registro de estado ZNVC al busd con el fin de guardar el estado de la máquina en la pila al ocurrir una interrupción.

**YesJump:** Esta señal pertenece al módulo CPU2017. Su valor está activo ‘1’ cuando se cumple la condición de salto, y ‘0’ cuando la condición de salto no se cumple y no hay

que saltar en el PC. La condición se comprueba en el módulo CPU2017, comprobando el valor del registro de estado para cada tipo de condición. Si una condición se cumple, la señal YesJump pasa a ser activa.

**Instrucción:** Esta señal pertenece al Registro de Instrucción. Contiene el valor del registro de instrucción. Útil para saber qué instrucción se debe ejecutar en la unidad de control. Es posible acceder a partes de este registro. Por ejemplo, si deseamos comprobar cuáles son los primeros 4 bits de la instrucción, escribiríamos Instrucción(15 downto 12).

**LoadIReg:** Esta señal pertenece al Registro de Instrucción. Carga en el registro de instrucción el contenido del busd. Se usa en el estado de leer la siguiente instrucción (ReadInst).

**EnableDisp:** Esta señal pertenece al Registro de Instrucción. Hay veces que necesitamos usar parte del contenido de una instrucción. Por ejemplo, en muchas instrucciones de salto tienen el desplazamiento para modificar el PC en la propia instrucción, por lo que se vuelcan los 8 bits inferiores del registro de instrucción en el busd para modificar en la ALU o para modificar el PC.

**DriveMem:** Esta señal pertenece al buffer triestado. Deja pasar el contenido del busd al bus de datos, para poder escribir en memoria. Se activa cuando la señal vale 1. Se utiliza cada vez que se quiera realizar una escritura en memoria.

**DriveCPU:** Esta señal pertenece al buffer triestado. Deja pasar el contenido del bus de datos al busd, para poder leer de memoria. Se activa cuando la señal vale 1. Se utiliza cada vez que se quiera realizar lectura de memoria.

**ReadMem:** Esta señal pertenece a la memoria externa. Cuando está activa en ‘1’ se indica que se quiere realizar una lectura de memoria.

**WriteMem:** Esta señal pertenece a la memoria externa. Cuando está activa en ‘1’ se indica que se quiere realizar una escritura en memoria.

**Mhold:** Esta señal pertenece a la memoria externa. Cuando su valor es ‘0’ significa que el acceso a memoria no se ha completado y sigue procesándose. Cuando su valor es ‘1’ quiere decir que el acceso a memoria de lectura o escritura se ha completado.

**PrintMem:** Esta es una señal auxiliar de la memoria externa. Imprime el contenido de la memoria cuando su valor es 1 en un fichero denominado MEMORYDUMP.txt. Esta señal se activa cuando el procesador llega al estado Halted.

**LoadMAR:** Esta señal pertenece al registro de dirección de memoria (MAR). Sirve para cargar una dirección de 24 bits en el registro de dirección para acceder a memoria y realizar una lectura o escritura. No hace falta una señal de EnableMAR, ya que siempre que haya una dirección cargada en este registro será la elegida por la memoria para acceder a ella.

**Reset:** La señal de reset se activa en nivel bajo.

**Clock:** La señal de reloj se activa en el flanco de subida.

Una vez explicada todas las señales internas de la unidad de control, se explica su comportamiento y estructura de estados. La unidad de control funciona como una máquina de estados finitos determinista. Cada estado representa un ciclo de reloj en la ejecución de la máquina. La necesidad de transicionar entre estados está en que no puede haber conflictos entre recursos. Si un bus está ocupado por una señal, por este bus no puede ir al mismo tiempo otra señal, por lo que es necesario esperar a que se complete el ciclo para ceder el bus a la señal que está esperando a volcar su contenido. De los estados presentes y futuros se encarga el circuito secuencial, actualizando el estado en cada ciclo. Si se activa la señal de reset, el estado pasa a ser el estado inicial (Initial).

El procesador empieza a funcionar en el estado Inicial, comprueba si hay una interrupción, y si no la hay, procede de forma normal, cargando el valor del PC en el registro de dirección de memoria traer de memoria la instrucción a ejecutar. Además, se actualiza el PC incrementando su valor. Si hubiese una interrupción, es en este estado donde empieza a tratarse, pasando a ejecutar la rutina de tratamiento de interrupción, salvaguardando el PC y el registro de estado.

Una vez en el estado de leer la instrucción de memoria (ReadInst), activa las señales de leer de memoria (ReadMem y DriveCPU) y activa la señal de carga del registro de instrucción. El acceso a memoria para realizar la lectura no es inmediato, requiere de varios ciclos de reloj, por lo que hasta que la señal mhold no pasa a ser 1, se vuelve reiteradamente a este estado hasta que se completa el acceso a memoria, es decir, mhold pasa a valer 1.

Una vez leída la instrucción, pasa a ejecutarse. Aquí puede haber muchas posibilidades. Lo normal es que sea una instrucción con 2 operandos fuente y destino, como puede ser una suma ADD o un MOVE. Pero puede que sea una instrucción de salto (BRA, BCC, BSR), o de creación de marco de pila (LINK) o de parada del procesador (Stop). En estos últimos casos, hay estados especiales para ellos como se puede ver en el diagrama de estados. Si se tiene una instrucción con dos operandos. Se elige qué tipo direccionamiento tiene el operando fuente. El operando fuente se almacena en el registro auxiliar 2. Si hiciese falta acceder a memoria para tener el dato del operando, se transicionaría a otros estados (estados B, que representan la obtención del operando fuente).

Una vez obtenido el operando fuente en el registro auxiliar 2, se pasa a obtener el operando destino (estados A, que representan la obtención del operando destino). De nuevo, dependiendo del tipo de direccionamiento, se elige un estado u otro para obtener el dato, pudiendo requerir de varios accesos a memoria. El dato del operando destino se almacena en el registro auxiliar 1. La ejecución continúa en el estado C1.

En el estado C1 se realiza la operación de la ALU, utilizando como entradas los buses auxiliares conectados a los registros auxiliares. Se actualiza el registro de estado y se guarda el resultado de la operación de la ALU en un registro del banco de registros

indicado por el operando destino, o en la memoria realizando una escritura en la dirección indicada por el operando destino.

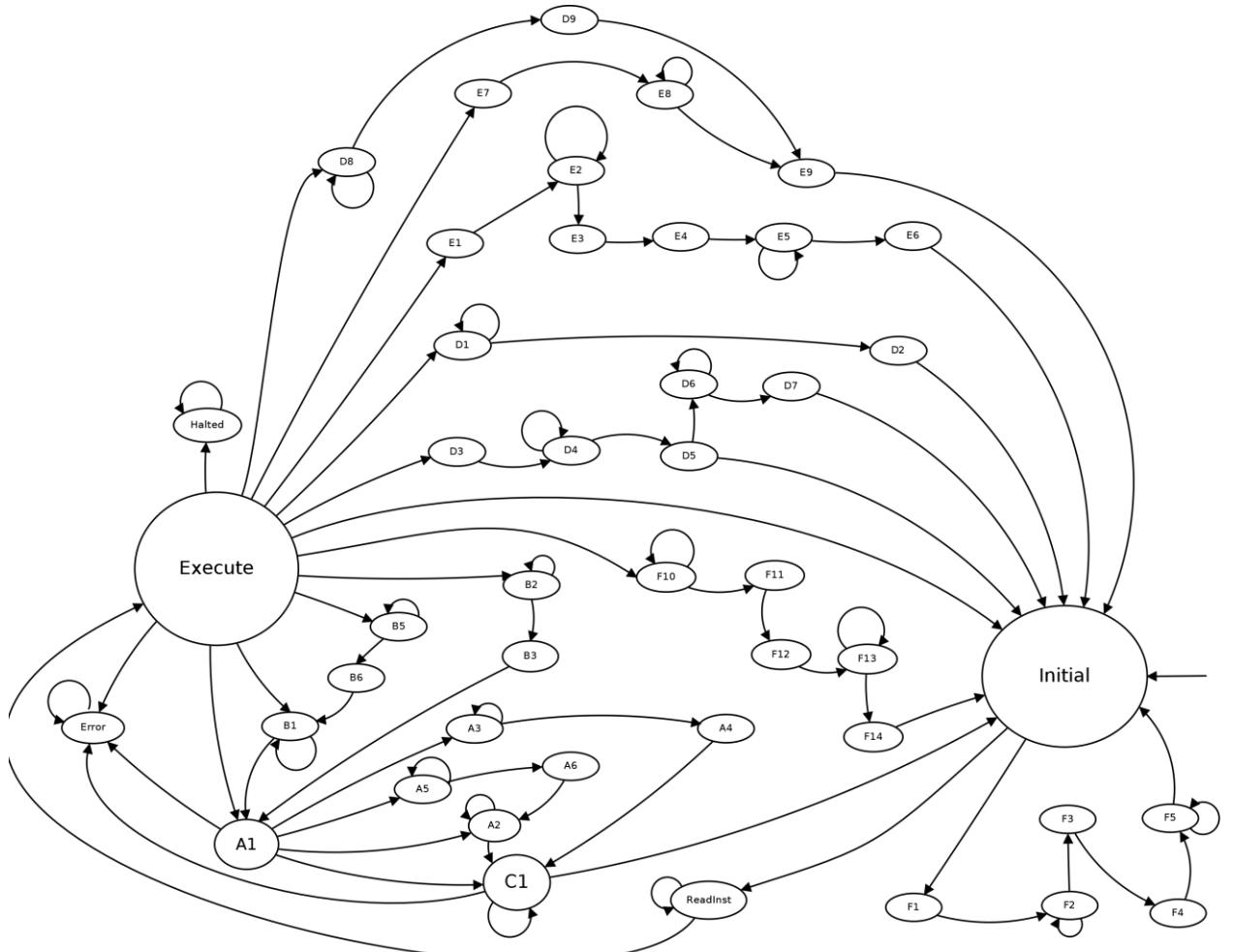
El resto de estados son casos particulares para cada instrucción, empezando su ejecución en el estado Execute. En el diagrama de estados se puede ver de forma superficial todas las posibilidades de estados. Además, se explica con varios ejemplos por qué estados pasa la ejecución de una instrucción y qué elementos realiza.

Una vez terminada la ejecución de una instrucción, es necesario que las señales de volcado en buses (Enable) y de carga de registros (Load) cambien su valor a 0, porque en caso contrario podría producir conflictos en los buses y registros de estados futuros.

Lo que no hay que modificar son los selectores (de registro, registro de instrucción...), pues es posible se deban utilizar en estados futuros, y no afecta a la ejecución de otras instrucciones con conflictos.

Los estados de la unidad de control son:

Initial, ReadInst, Execute, Halted, Error, B1, B2, B3, B4, B5, B6, A1, A2, A3, A4, A5, A6, C1, D1, D2, D3, D4, D5, D6, D7, D8, D9, E1, E2, E3, E4, E5, E6, E7, E8, E9, F1, F2, F3, F4, F5, F6, F7, F8, F10, F11, F12, F13, F14.

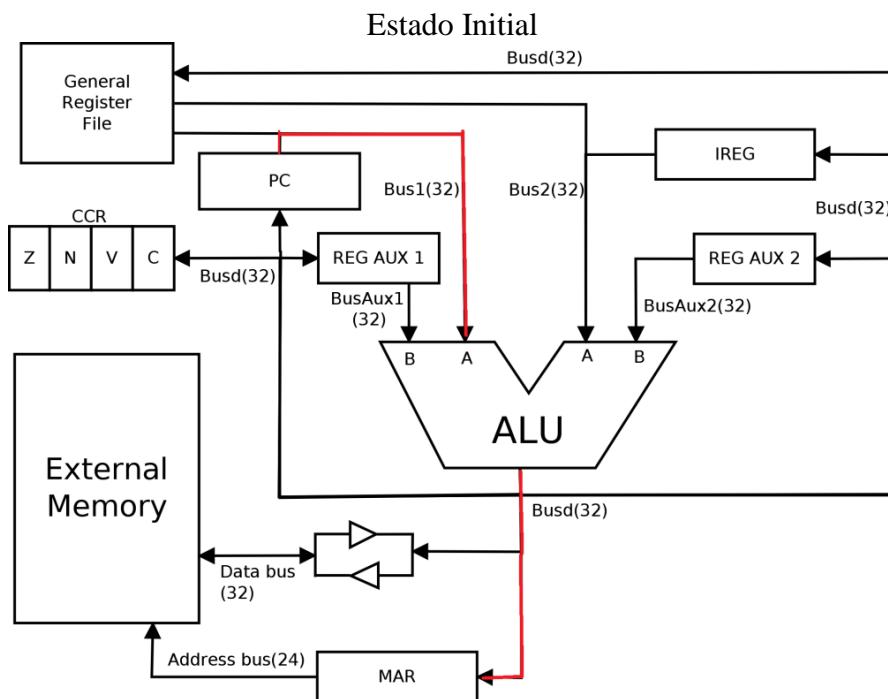


A continuación, se muestran diversos ejemplos de instrucciones con un análisis detallado de cada estado.

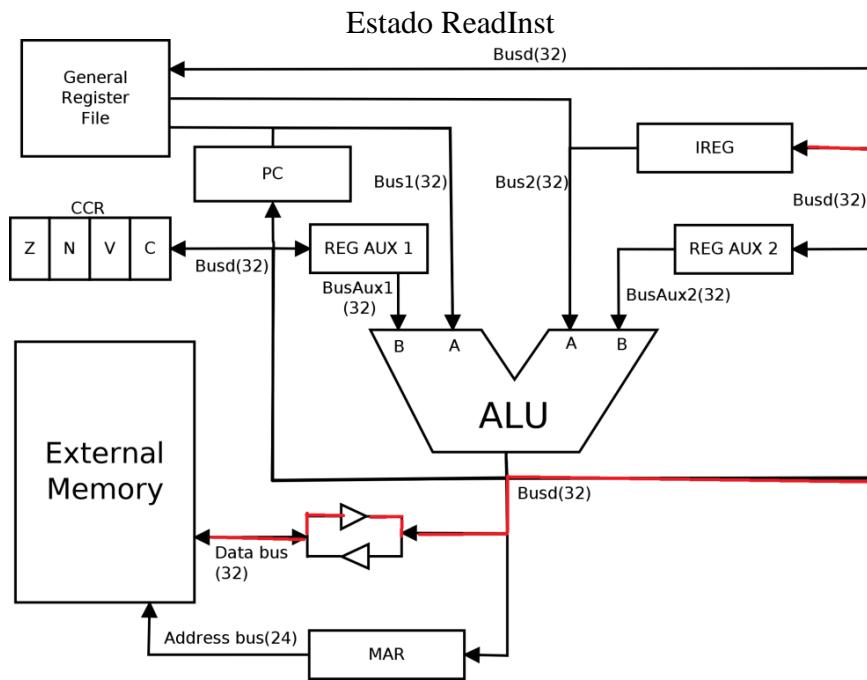
### 3.3.5.1 MOVE #100,D1

Esta instrucción transfiere el número 100 decimal al registro de datos D1.

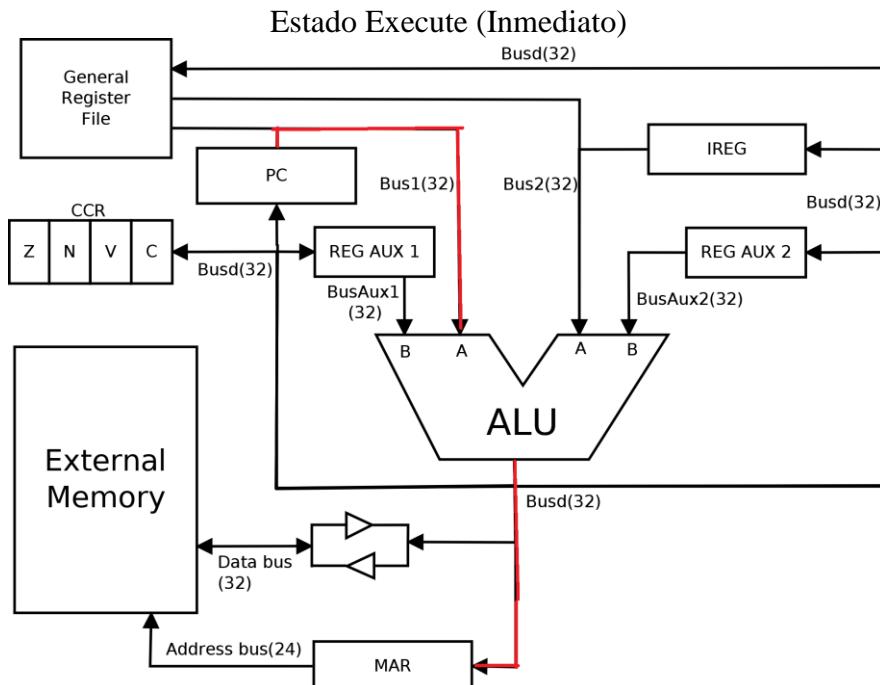
Estados:



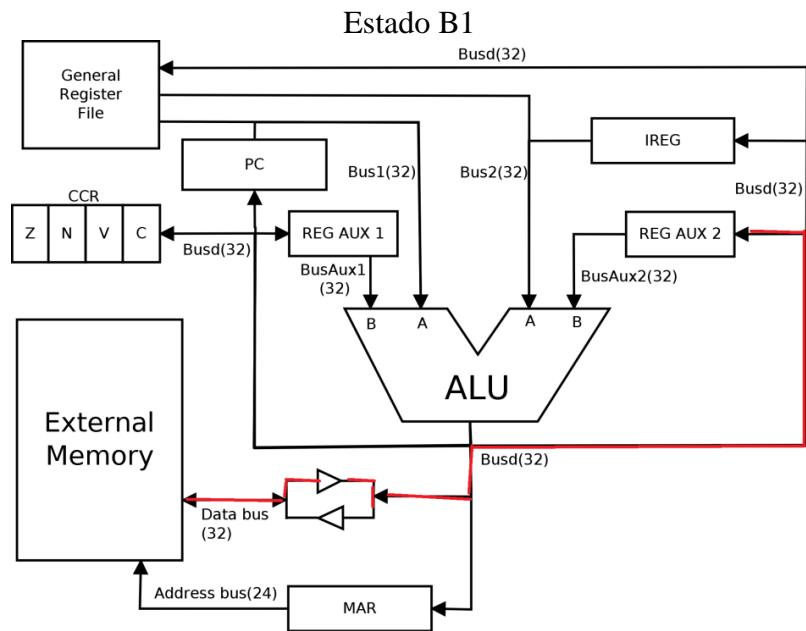
En este estado se carga la dirección del PC en el registro de memoria para leer la instrucción



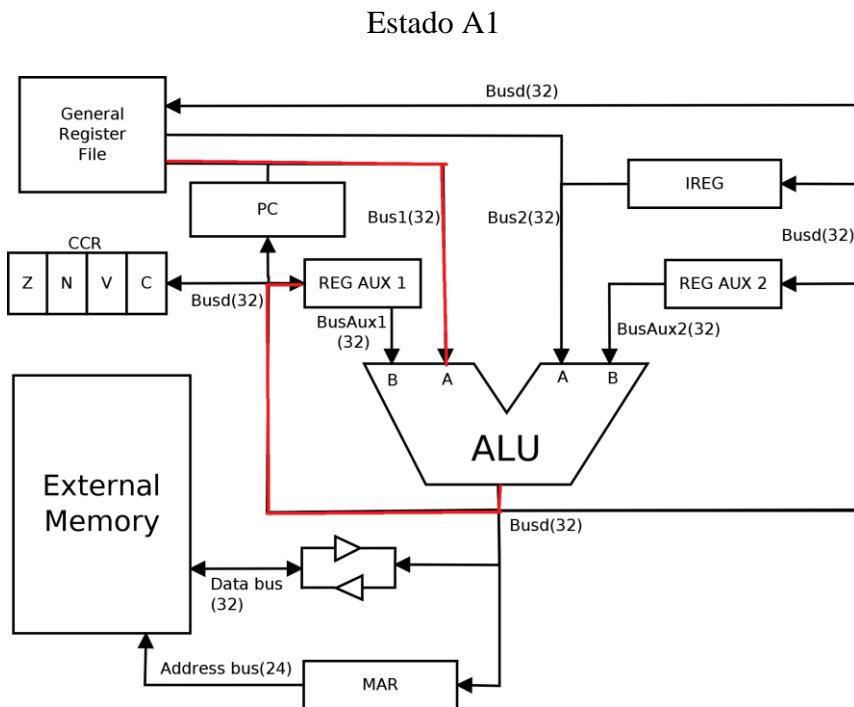
Lee la instrucción de memoria, abre el buffer triestado y carga la instrucción en el registro de instrucción.



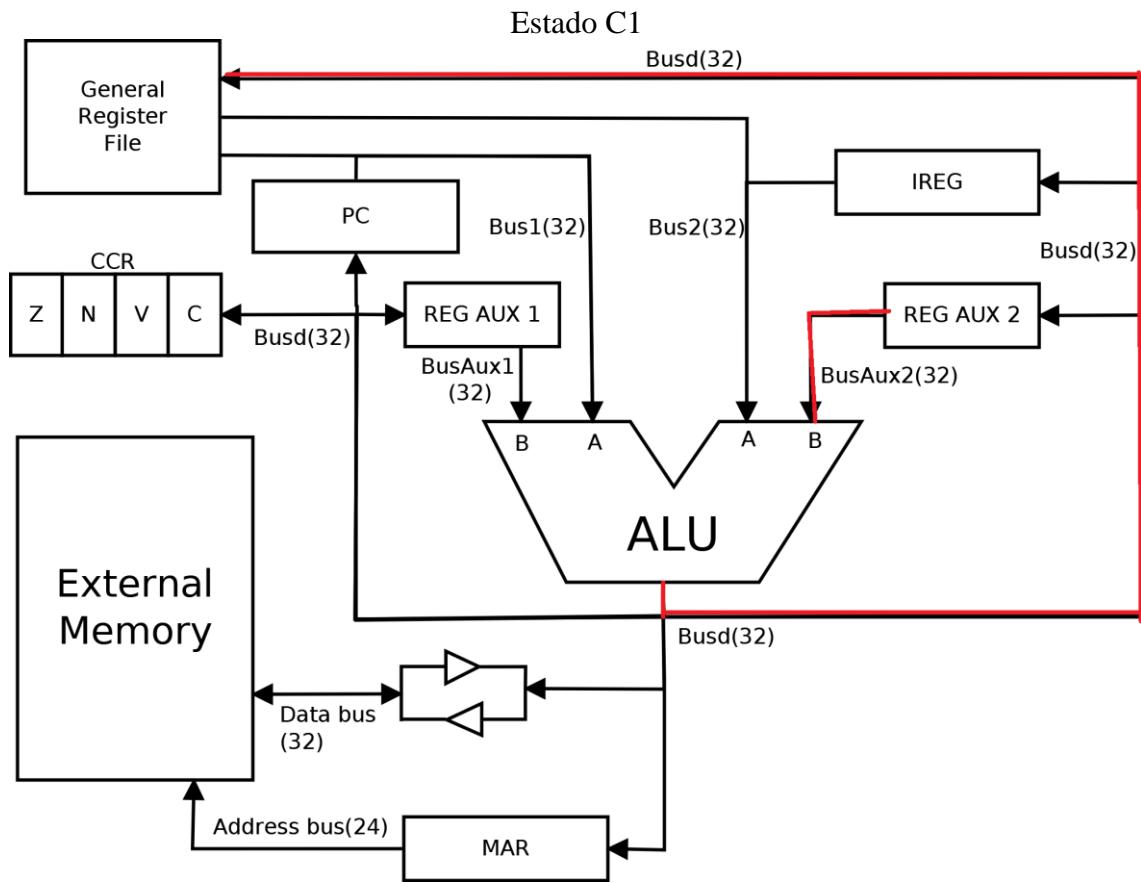
El operando B es el de tipo inmediato. Procede a recoger la segunda palabra de memoria cargando en el PC la siguiente dirección.



Carga el operando inmediato fuente en el registro auxiliar 2.



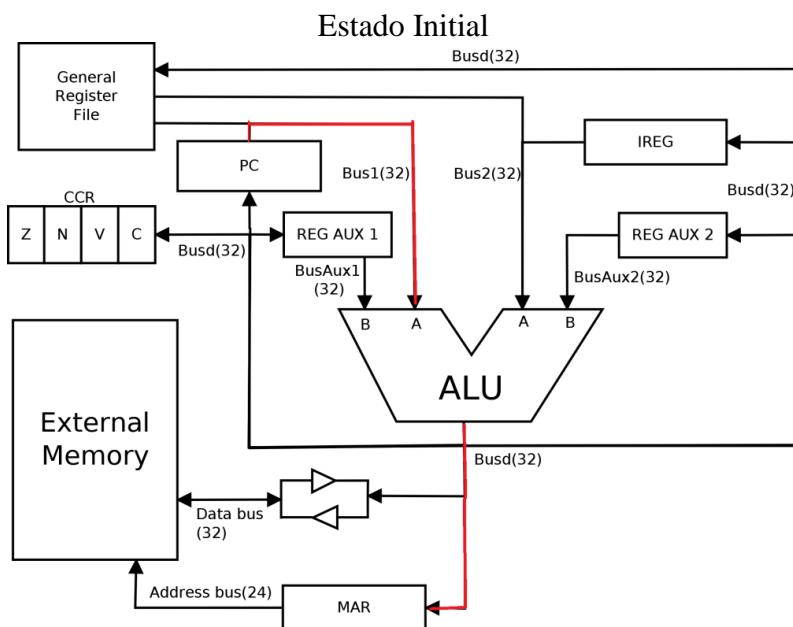
Carga el operando destino A en el registro auxiliar 1. En este caso no tiene importancia, puesto que vamos a sustituir el contenido del registro D1.



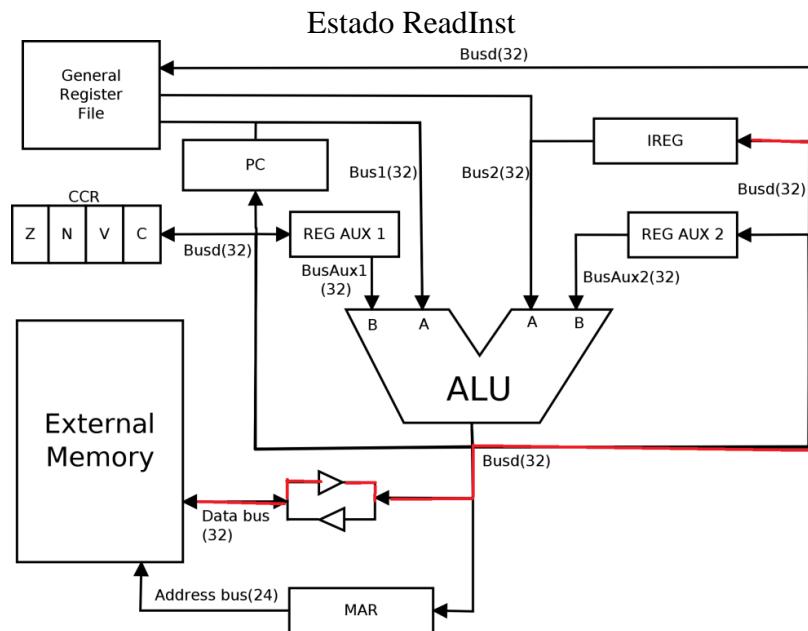
Finalmente carga el dato inmediato almacenado en Registro auxiliar 2 en el registro D1.

### 3.3.5.2 ADD (A1),(A2)+

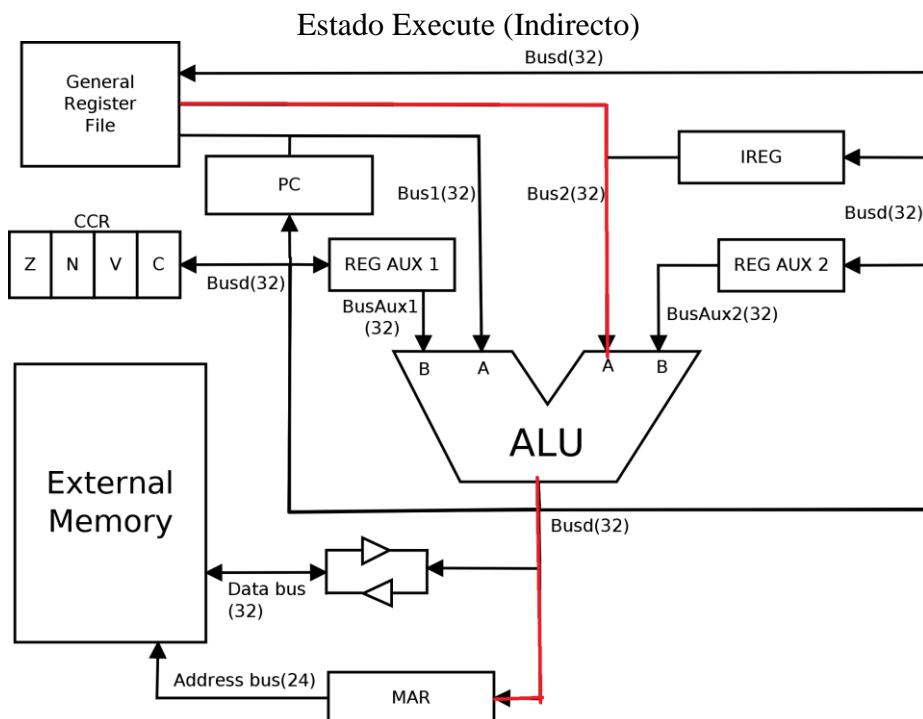
Estados:



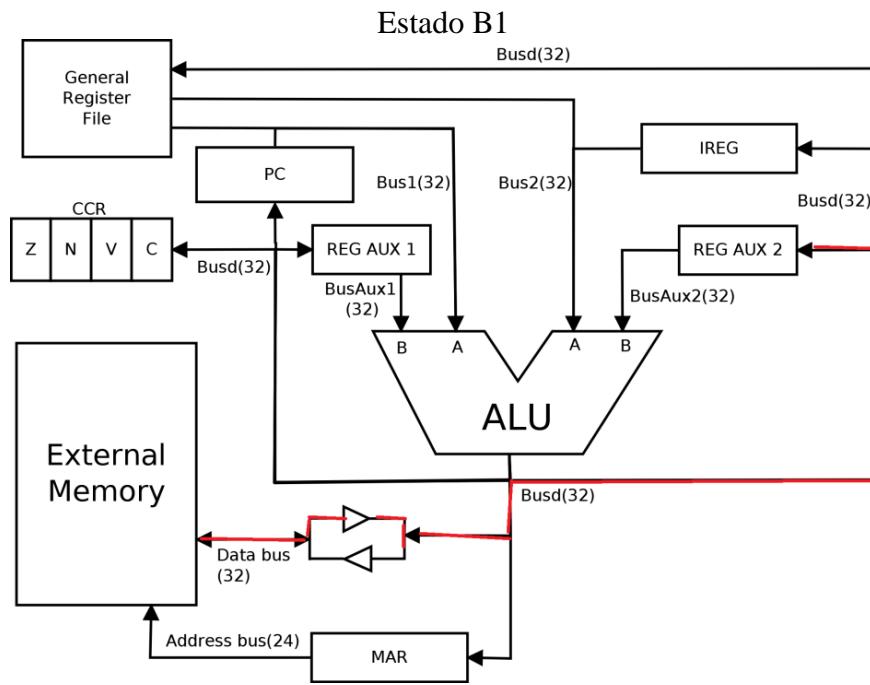
En este estado se carga la dirección del PC en el registro de memoria para leer la instrucción.



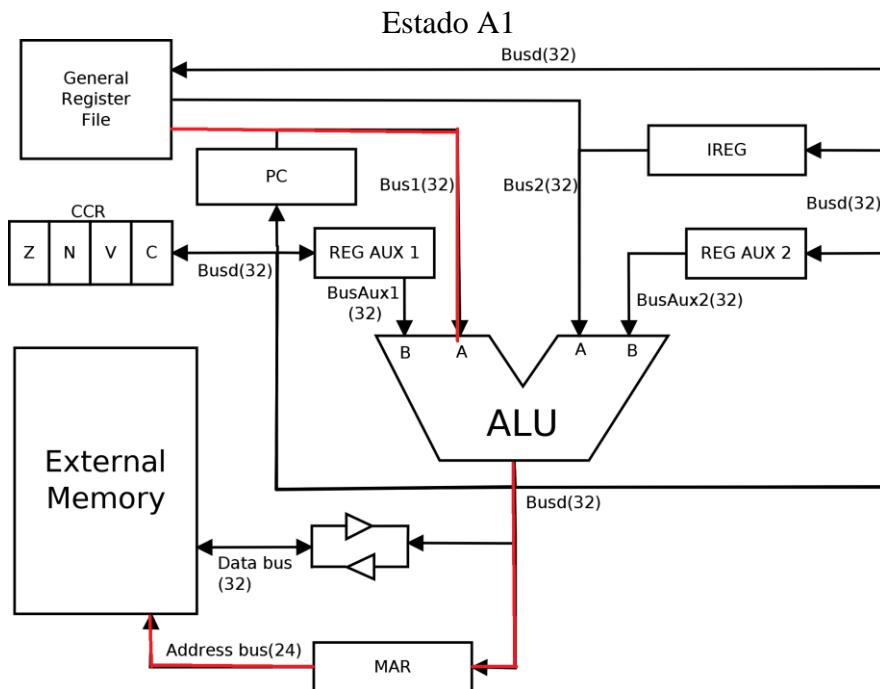
Lee la instrucción de memoria, abre el buffer triestado y carga la instrucción en el registro de instrucción.



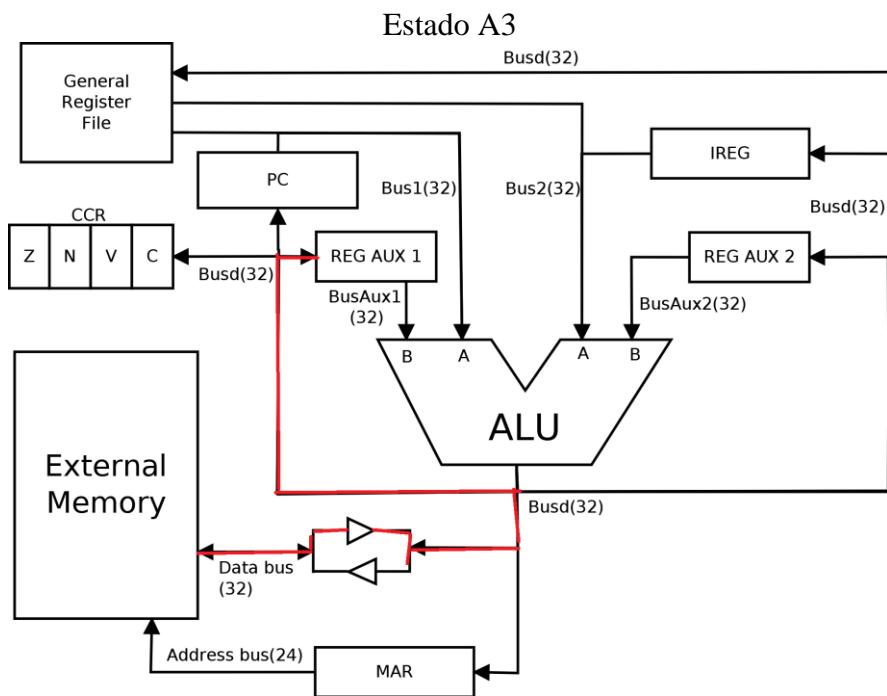
El operando Fuente es un indirecto a registro. Hay que acceder a memoria para obtener el dato de memoria.



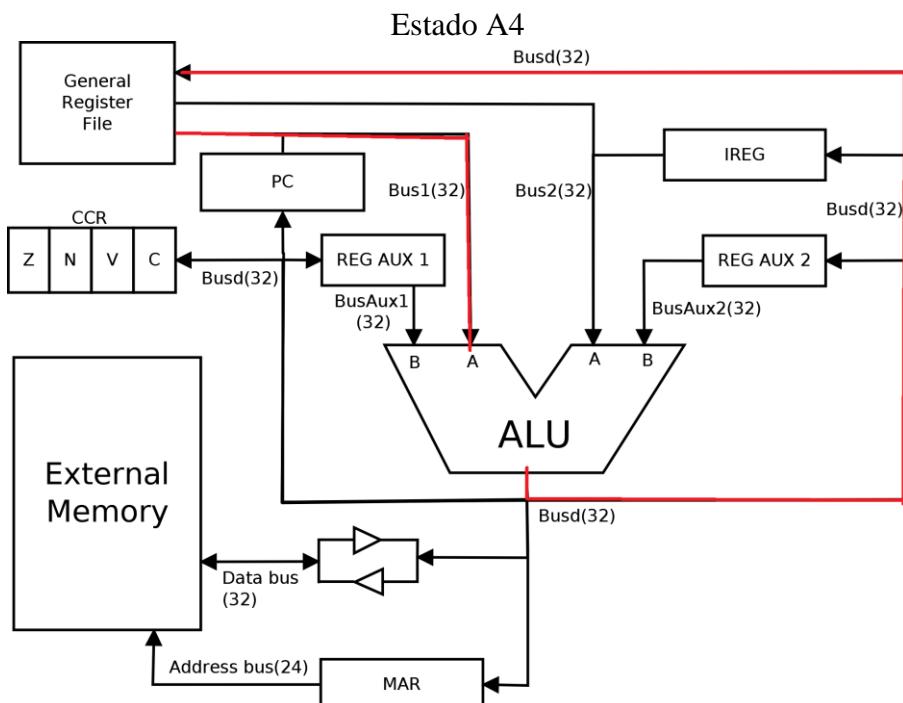
Obtiene el dato del operando fuente de memoria, abre la puerta triestado y lo carga en registro auxiliar 2.



El operando destino también es un indirecto a registro, pero con postincremento. Accede de nuevo a memoria para obtener el dato, mediante la dirección que hay en el registro de dirección del operando destino.

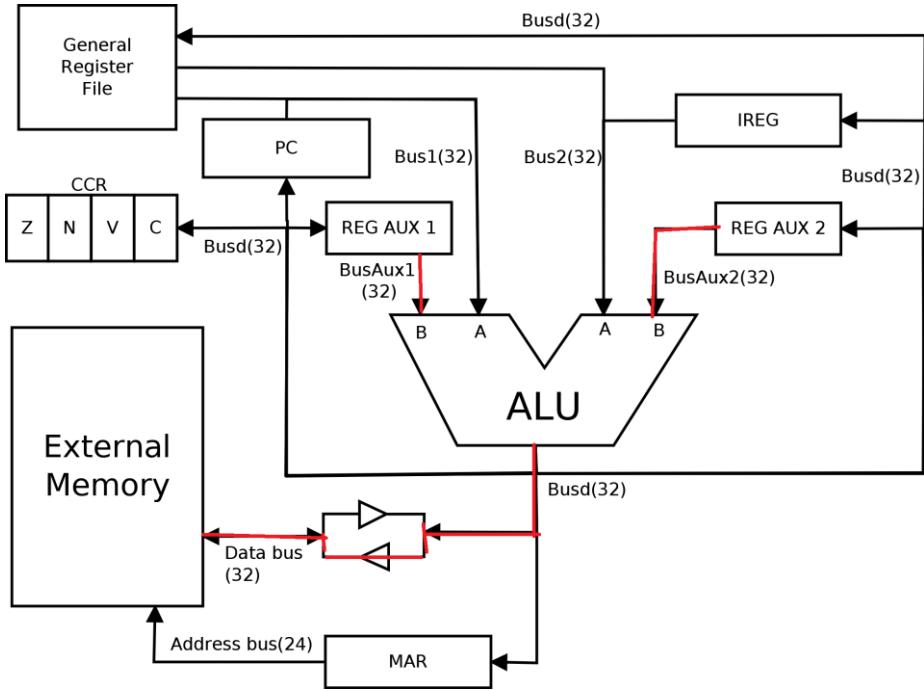


Obtiene el dato del operando destino y lo carga en el registro auxiliar 2.



Realiza el postincremento del operando destino.

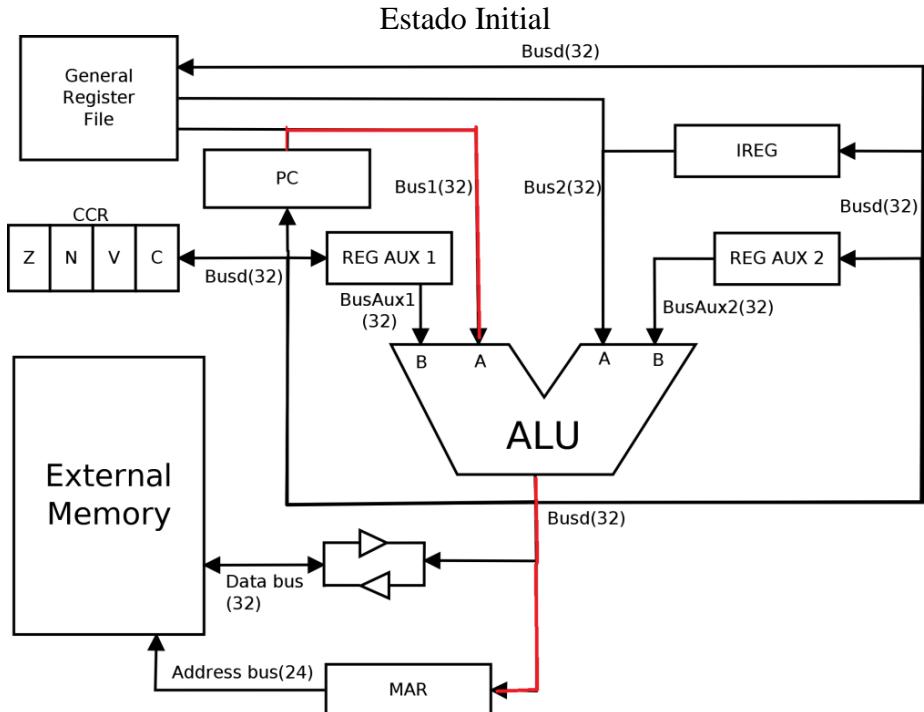
**Estado C1**



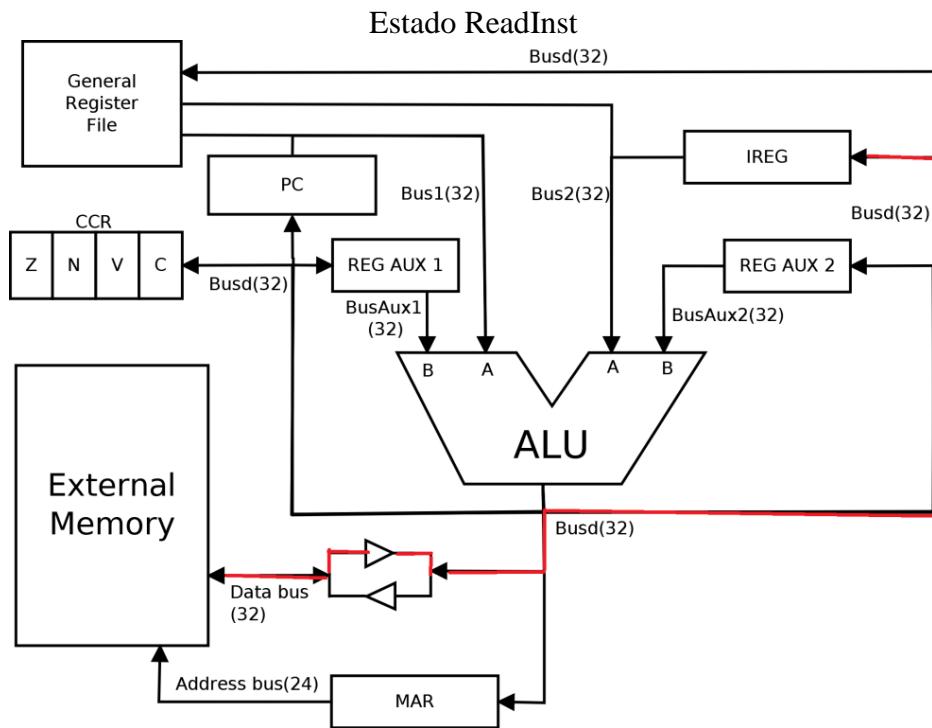
Finalmente realiza la suma, activando los 2 registros auxiliares y cargando en memoria, en la dirección del operando destino.

### 3.3.5.3 RTS

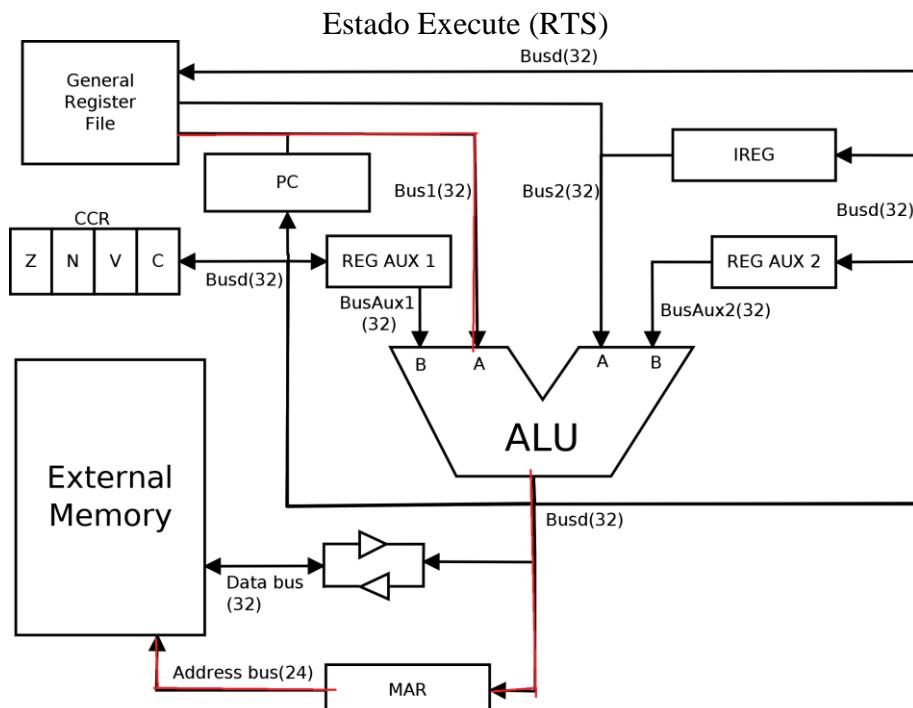
Estados:



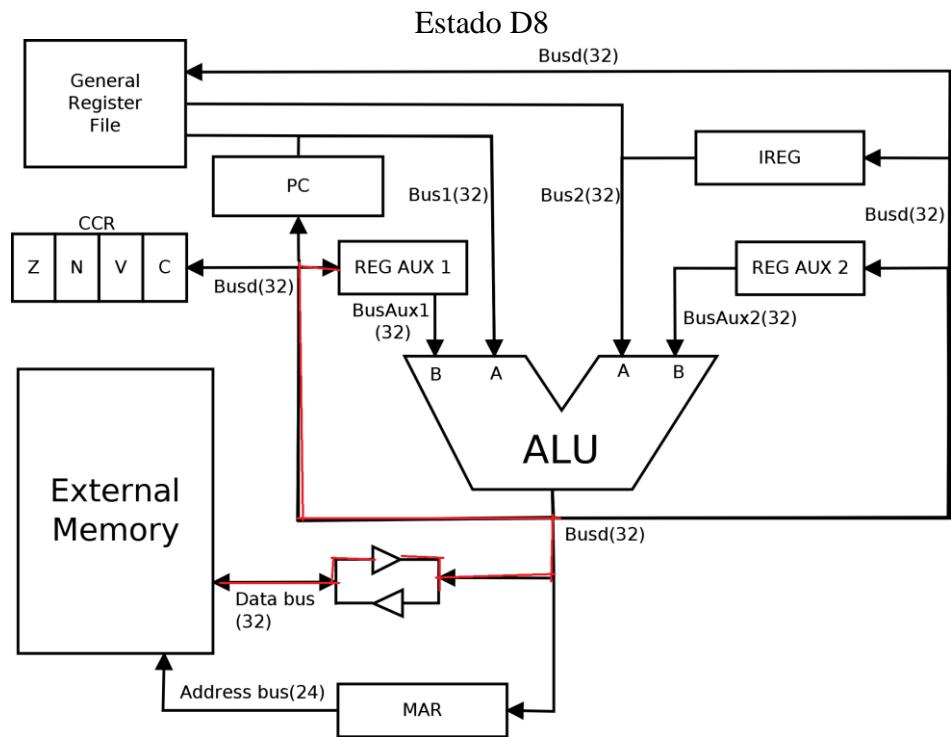
En este estado se carga la dirección del PC en el registro de memoria para leer la instrucción.



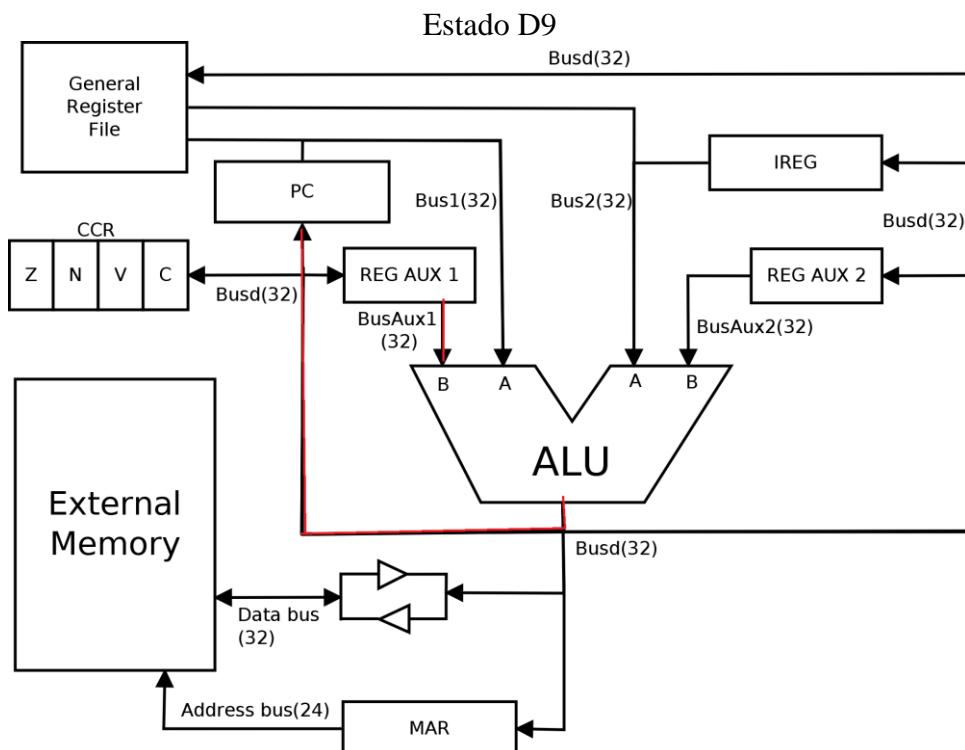
Lee la instrucción de memoria, abre el buffer triestado y carga la instrucción en el registro de instrucción.



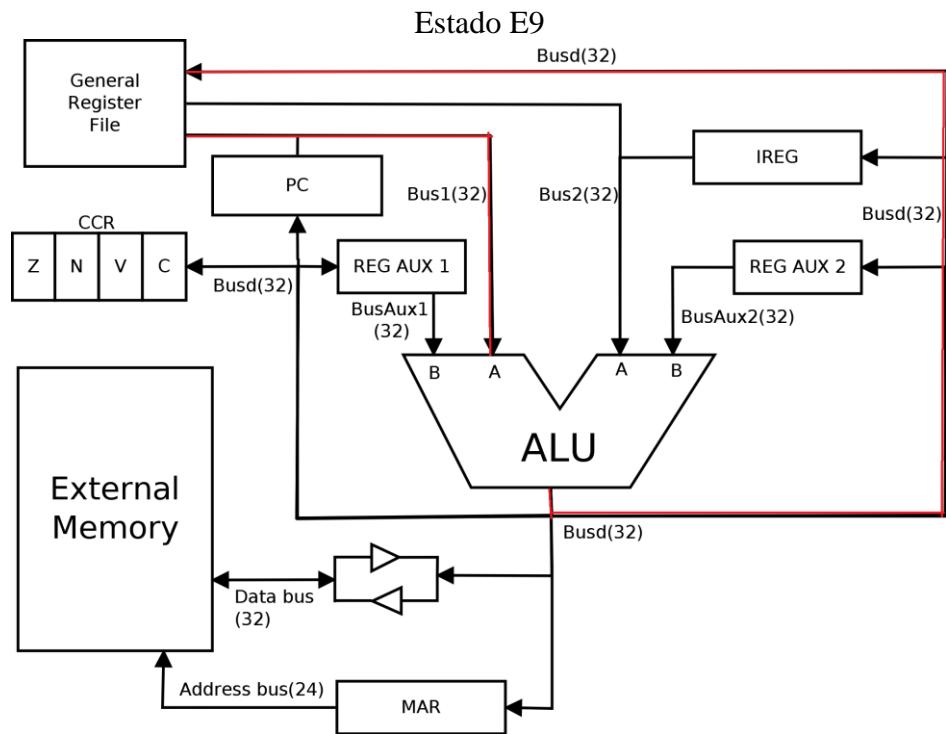
Accede a la dirección que apunta el puntero de pila, porque dentro está el PC a restaurar.



Guarda el PC que se va a restaurar en el Registro auxiliar 1.



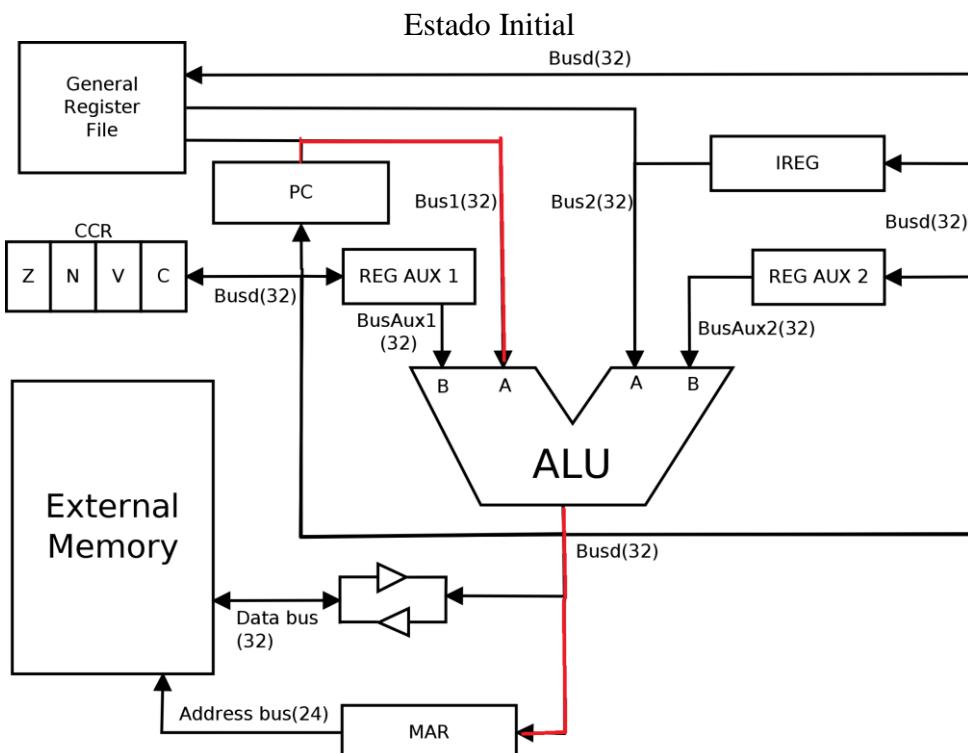
Restaura el PC antiguo cargándolo con la operación de PC de cargar PC.



Incrementa el puntero de pila en 1.

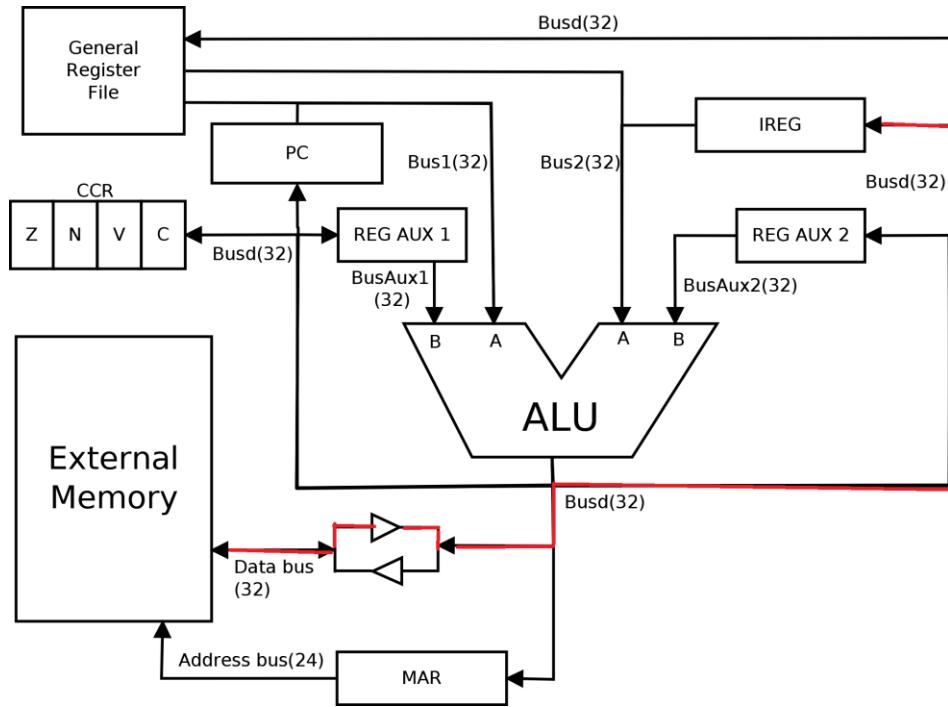
### 3.3.5.4 RTE

Estados:



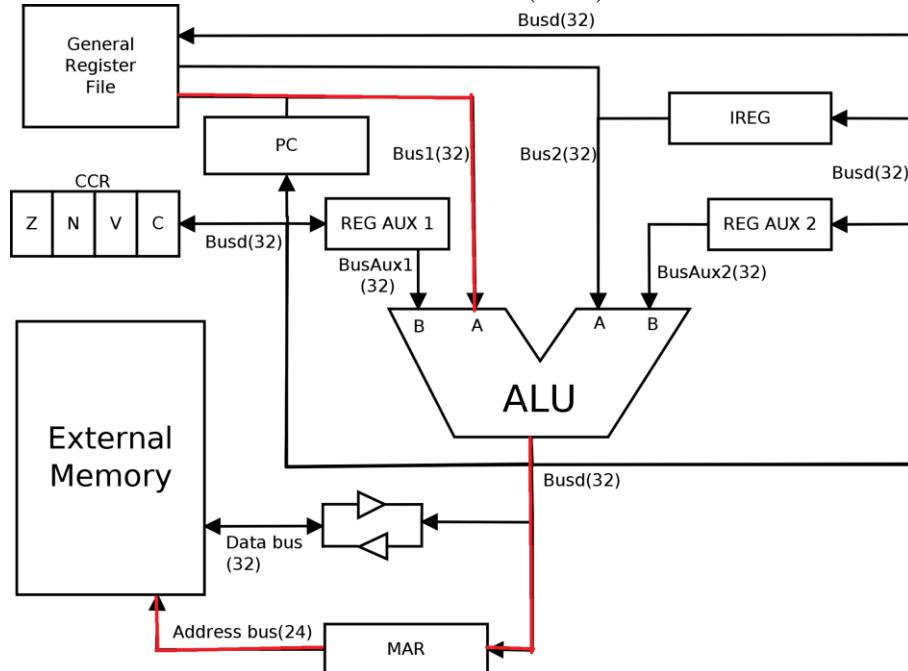
En este estado se carga la dirección del PC en el registro de memoria para leer la instrucción.

### Estado ReadInst

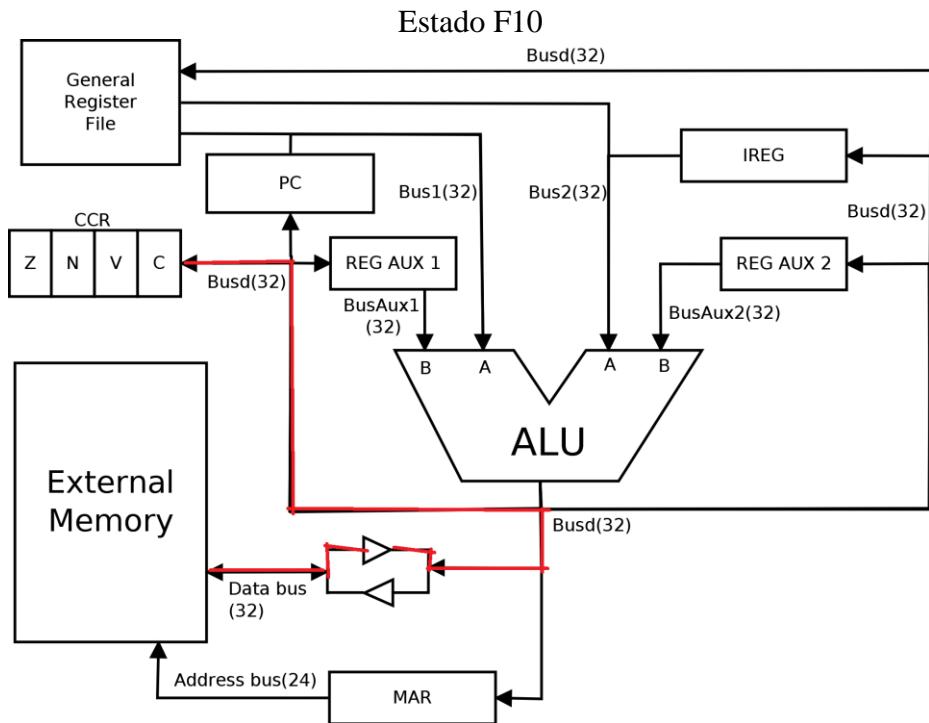


Lee la instrucción de memoria, abre el buffer triestado y carga la instrucción en el registro de instrucción.

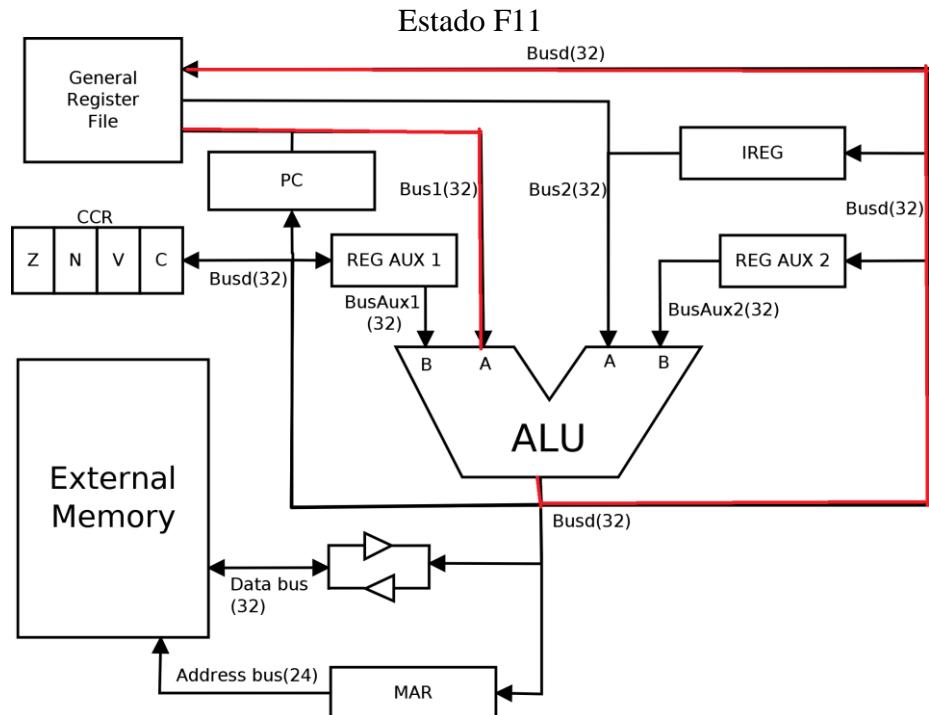
### Estado Execute (RTE)



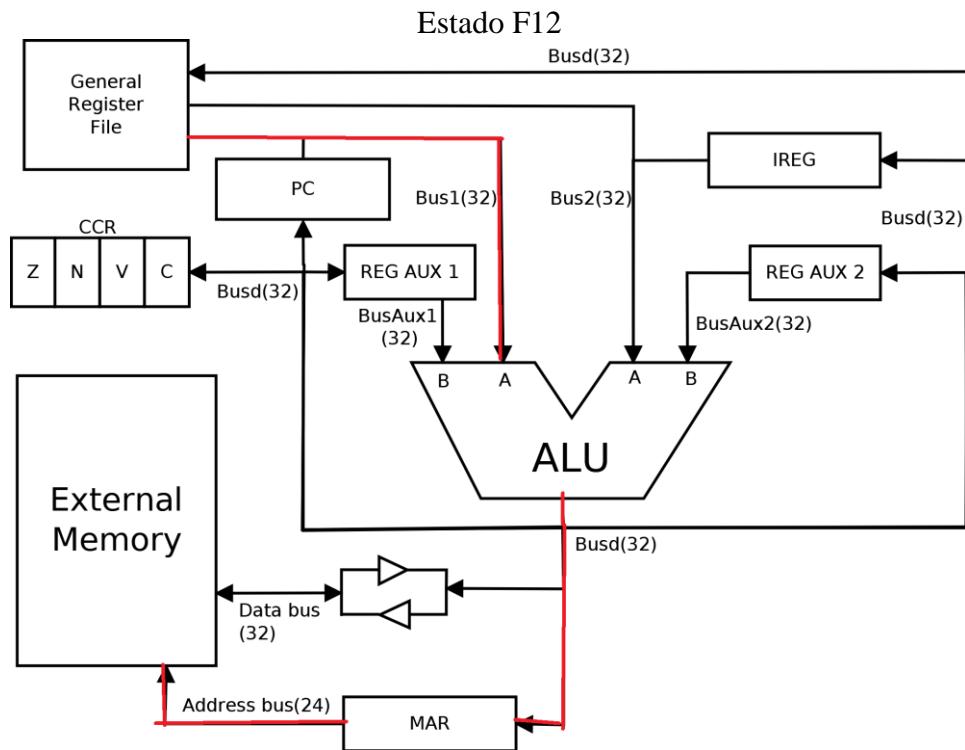
Carga la dirección del puntero de pila para poder restaurar el registro de estado.



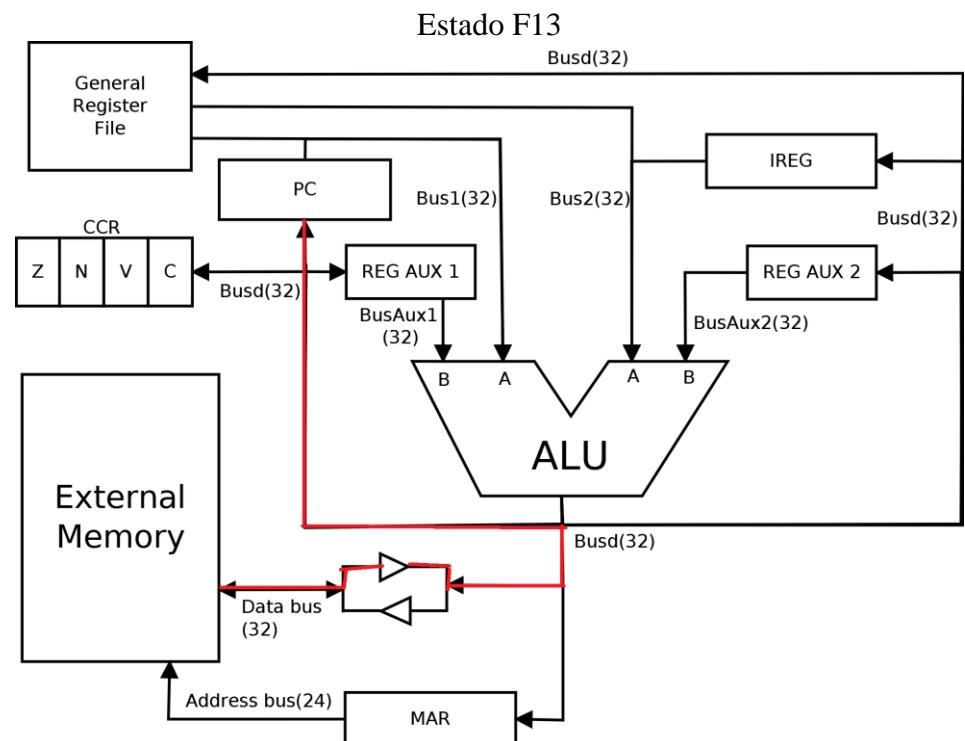
Restaura el registro de estado que estaba salvaguardado en pila.



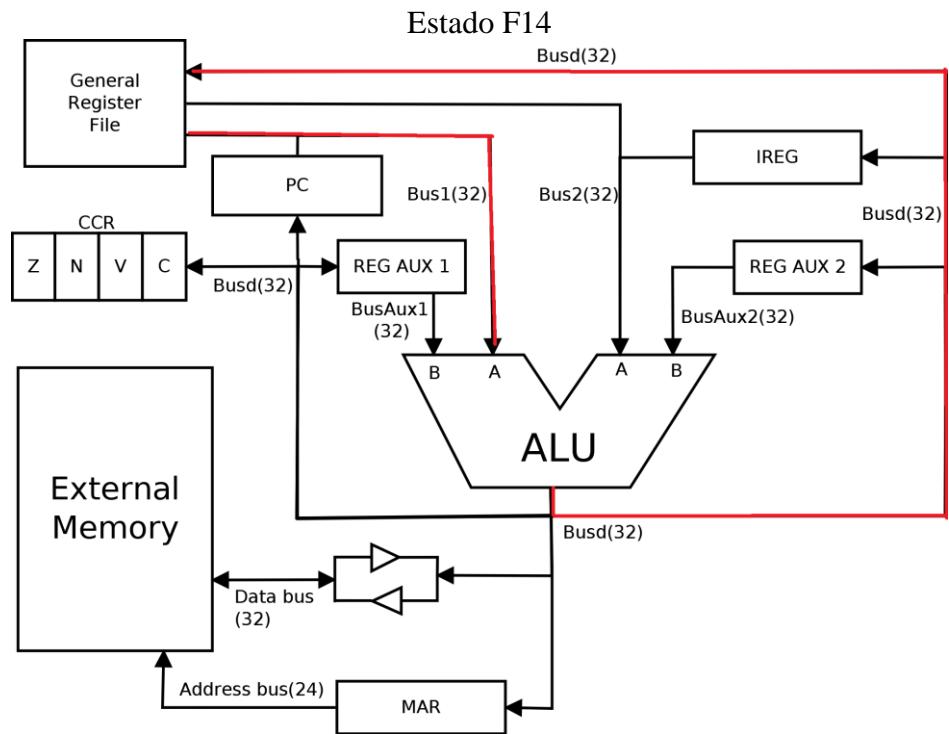
Incrementa el puntero de pila en 1.



Carga el puntero de pila en el registro de dirección para acceder al PC salvaguardado y poder restaurarlo.



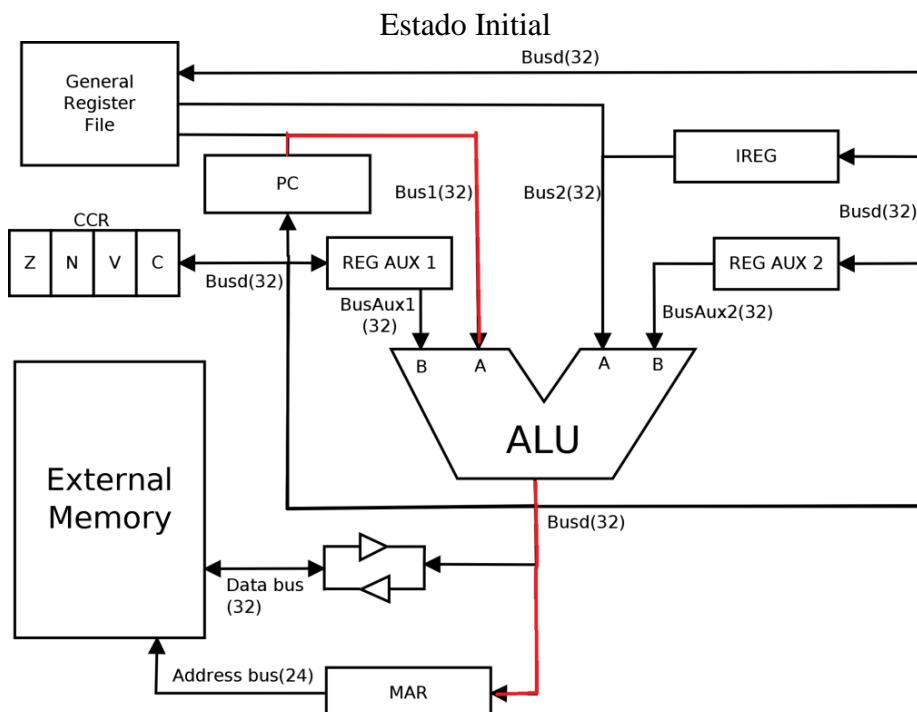
Restaura el PC salvaguardado en pila.



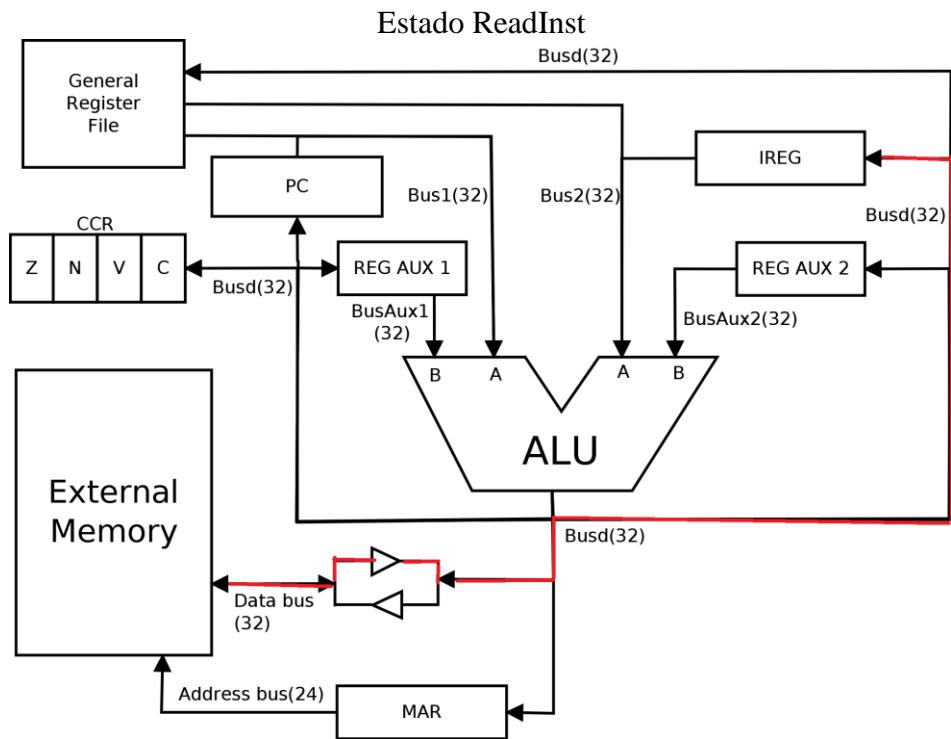
Incrementa en 1 el puntero de pila.

### 3.3.5.5 LINK A6,-2

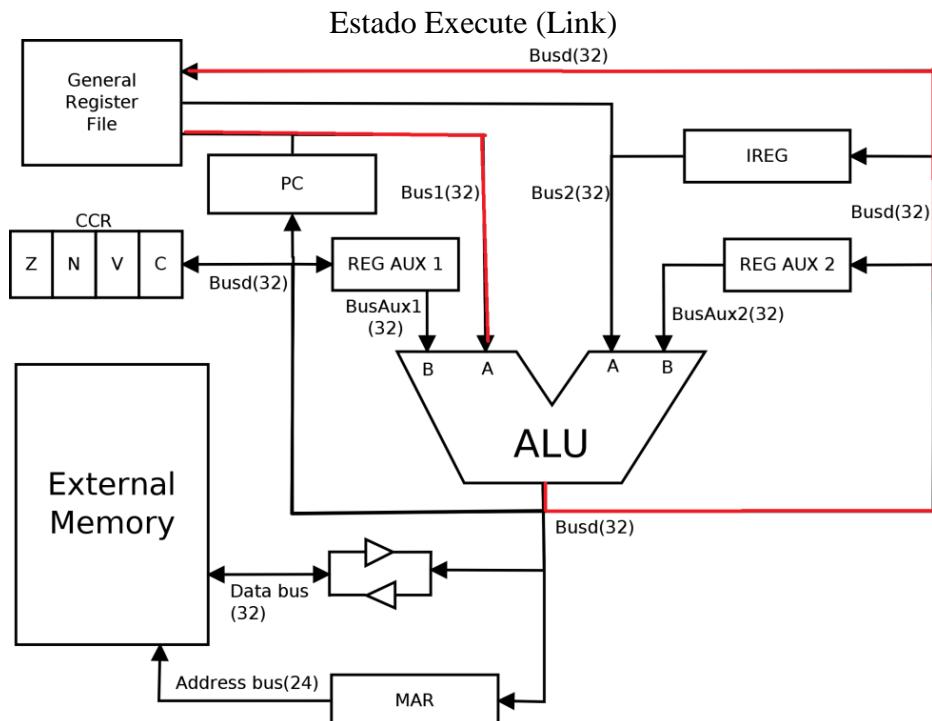
Estados:



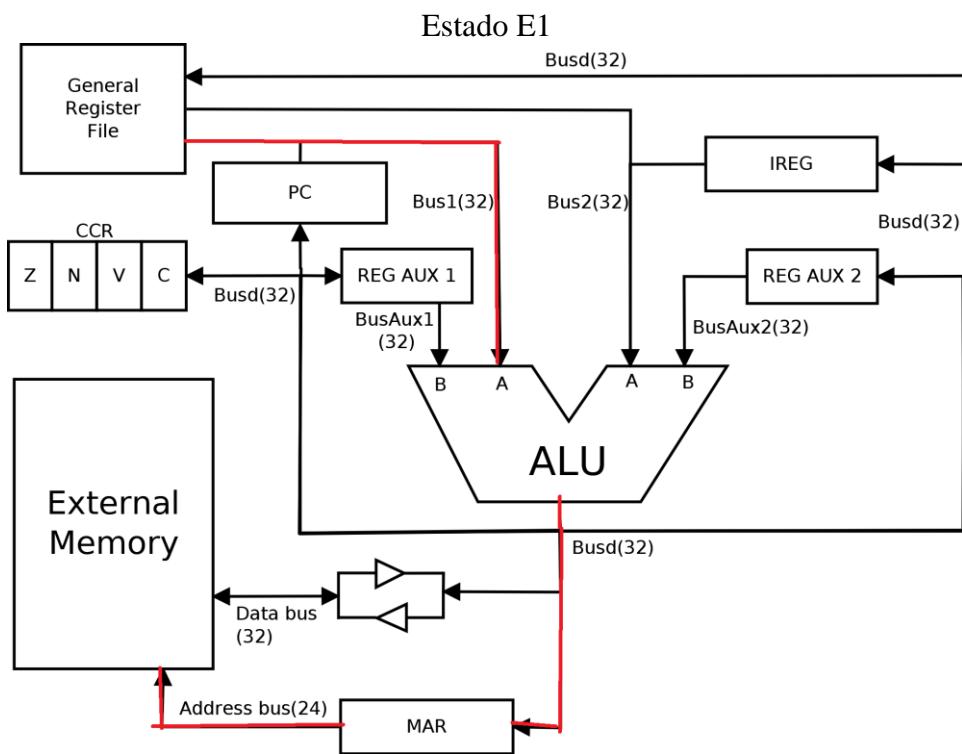
En este estado se carga la dirección del PC en el registro de memoria para leer la instrucción.



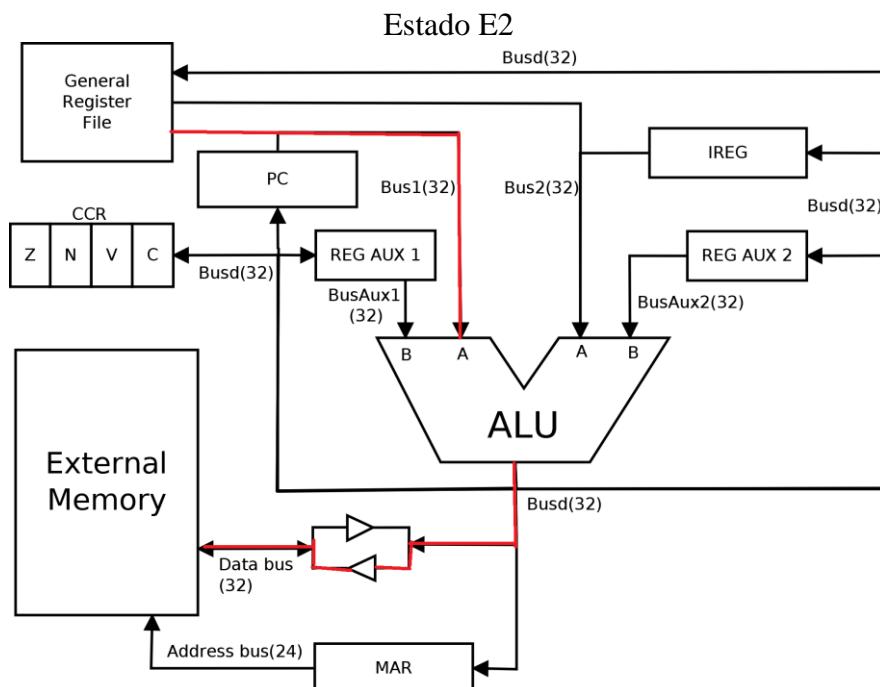
Lee la instrucción de memoria, abre el buffer triestado y carga la instrucción en el registro de instrucción.



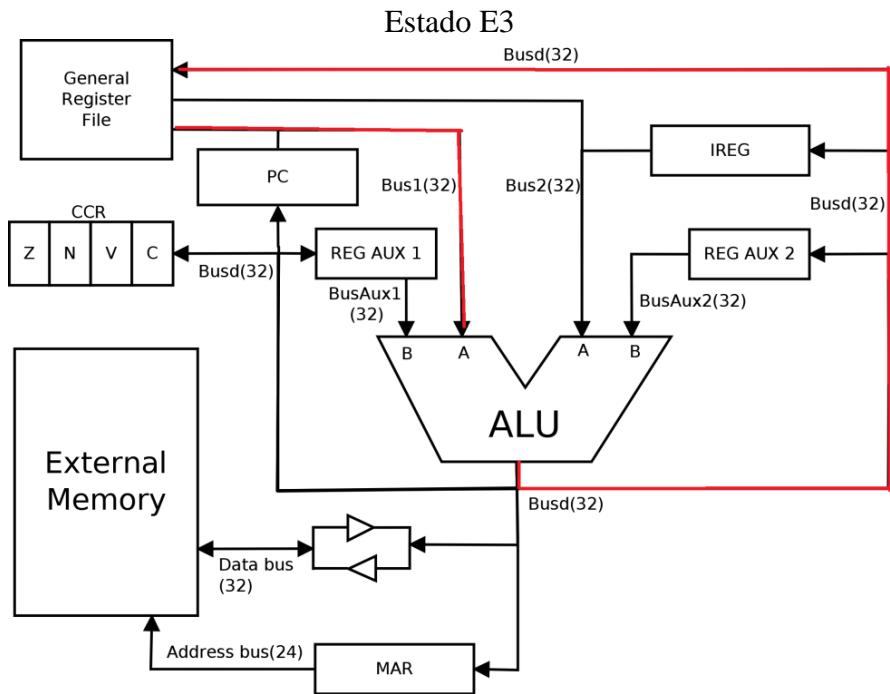
Decrementa en 1 el puntero de pila en la ALU.



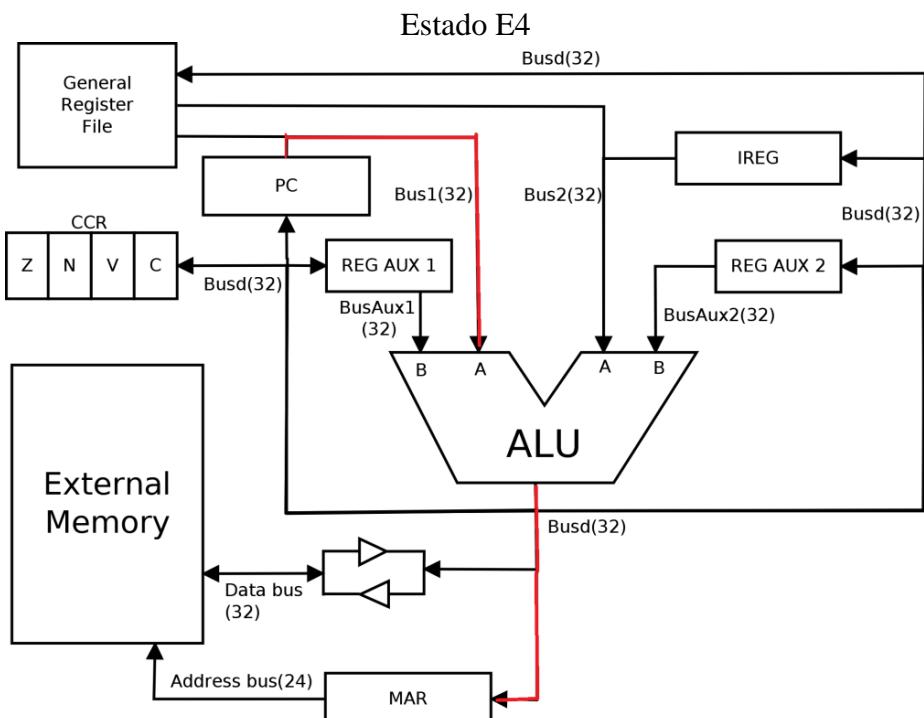
Carga en el registro de dirección de memoria el puntero de pila para poder escribir en él en el próximo estado.



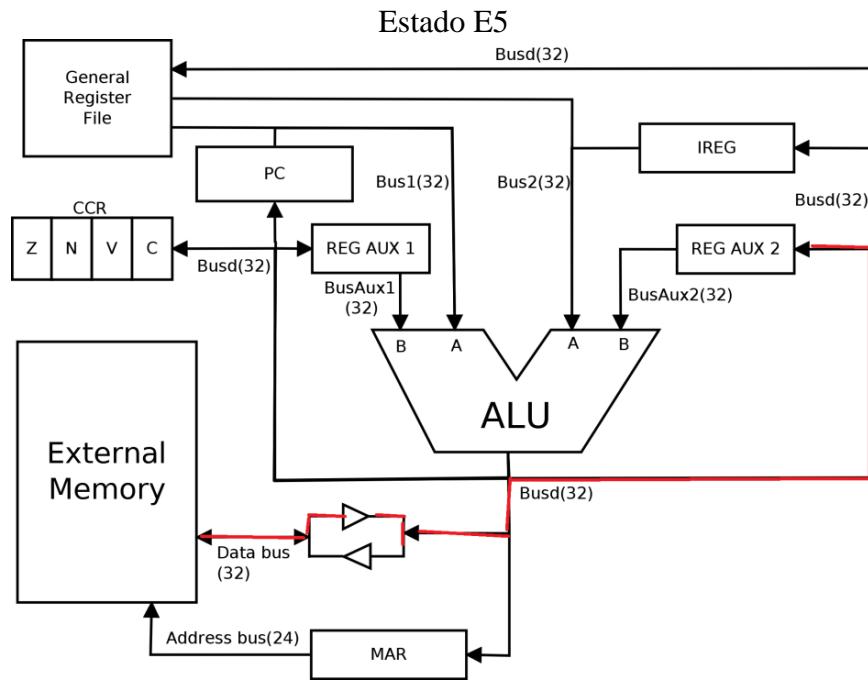
Escribe el contenido del registro de dirección de la instrucción en los bits 2 a 0 (registro A6) en la pila.



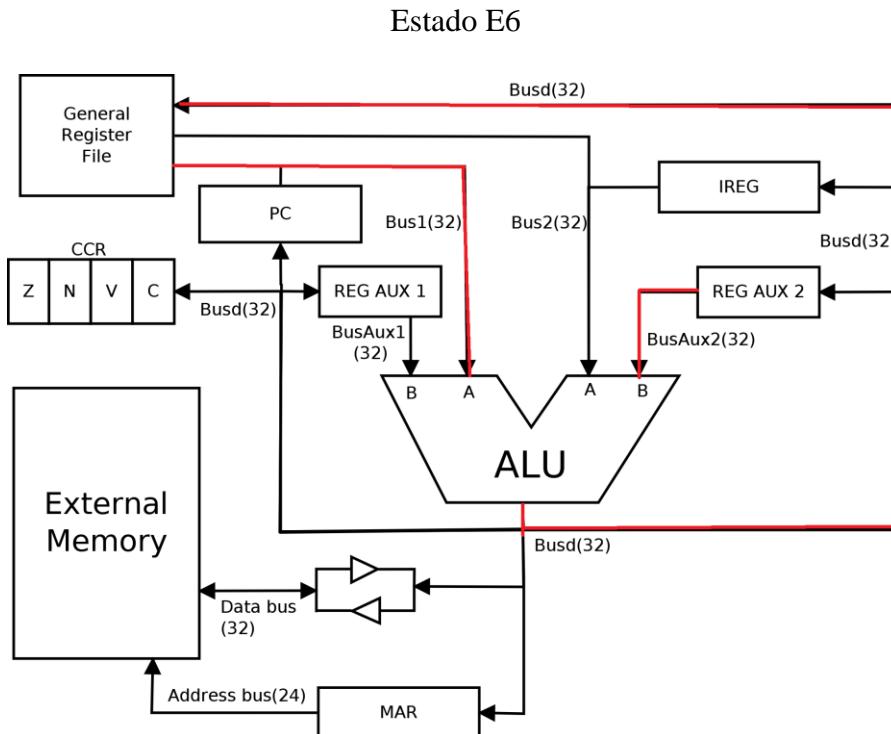
Carga el contenido del puntero de pila en el registro A6.



A continuación, tenemos que leer el desplazamiento -2 que está en memoria, en la siguiente palabra de la instrucción



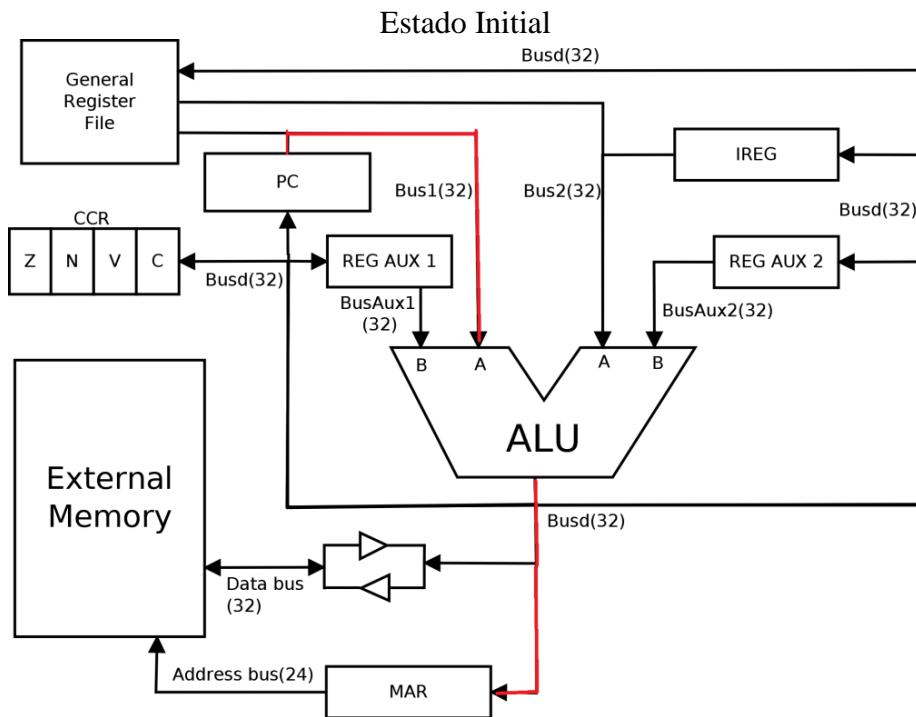
Obtiene el desplazamiento y lo guarda en el registro auxiliar 2.



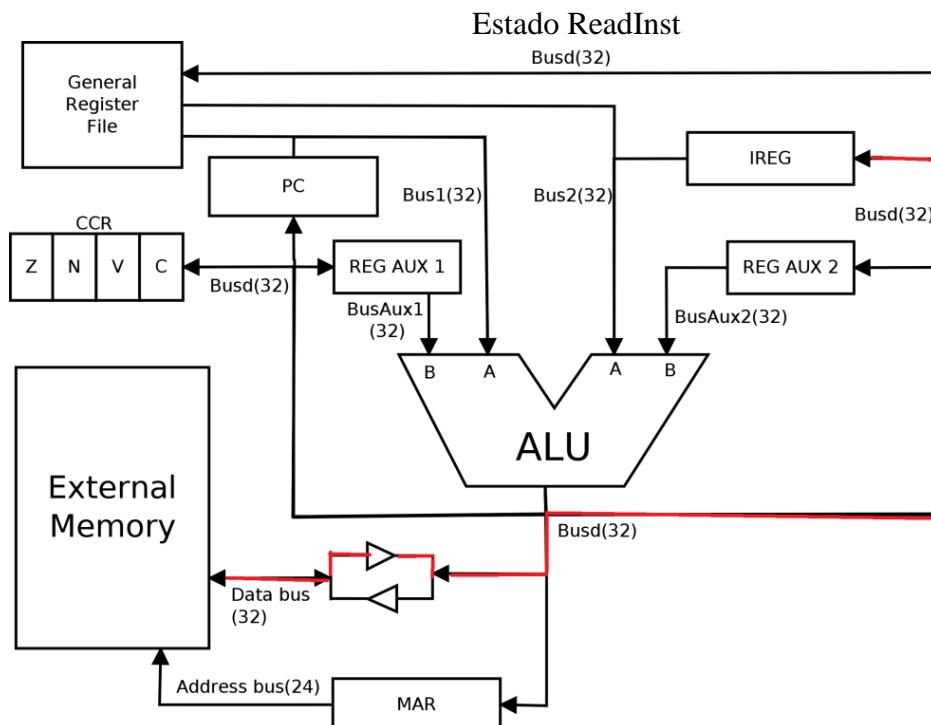
Suma el puntero de pila con el desplazamiento y lo guarda en el registro puntero de pila.

### 3.3.5.6 BEQ #5

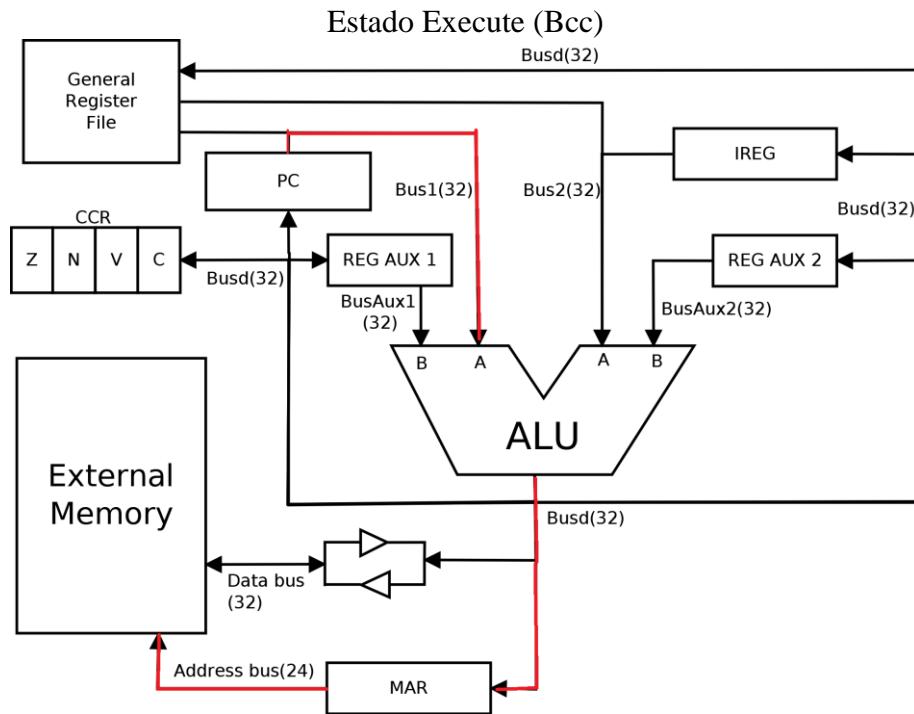
Estados:



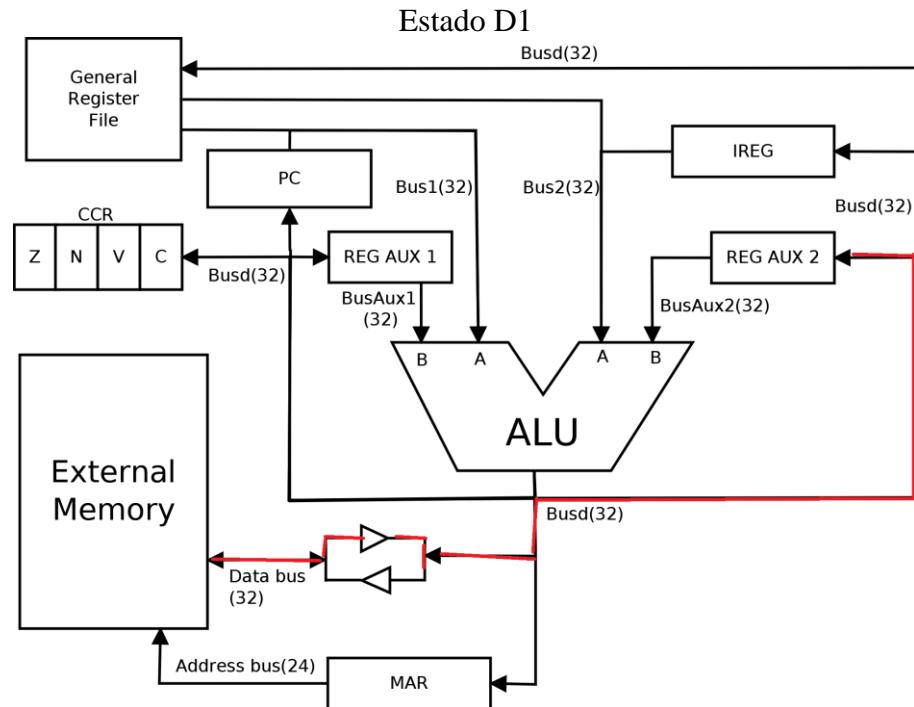
En este estado se carga la dirección del PC en el registro de memoria para leer la instrucción.



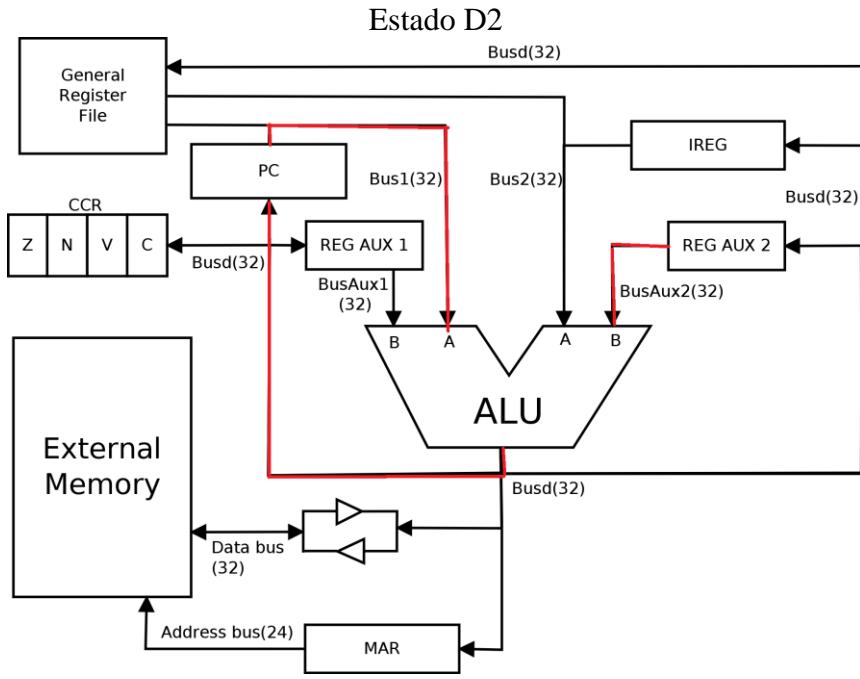
Lee la instrucción de memoria, abre el buffer triestado y carga la instrucción en el registro de instrucción.



Comprueba que la señal YesJump se está activa en '1'. Supongamos que está activa. Eso quiere decir que se ha cumplido la condición de salto determinada por los bits 12 a 9. Se dispone a leer la próxima palabra (que es el desplazamiento del salto) incrementando el PC.



Obtiene el desplazamiento de memoria y lo guarda en el registro auxiliar 2.



Finalmente, suma el contenido del PC y el desplazamiento almacenado en el registro auxiliar 2. El resultado de la operación en la ALU lo carga en el PC con la operación de carga del PC.

### 3.3.6 CPU2017

Fichero: CPU2017.vhd

Este módulo integra y conecta todos los elementos de la CPU, haciendo port mapping de todas las señales de cada módulo. Es aquí donde se comprueba las condiciones de salto para activar la señal YesJump a ‘1’. La comprobación se realiza mediante el registro de estado. Para cada condición de salto hay un código que representa esa condición. El código de condición que se comprueba es el que viene en la instrucción de salto condicional, entre los bits 11 y 8 de la instrucción. Estos códigos de condición están documentados en el manual del programador [3], en la instrucción BCC.

### 3.3.7 Memoria

Fichero: Memory.vhd

La memoria no pertenece al procesador, pero es necesaria para llevar a cabo la ejecución de las instrucciones y poder realizar la simulación en ModelSim.

La memoria está basada casi en su totalidad en la del Low Risk Computer, cambiando para el procesador 68000 varios parámetros para hacerla compatible.

Los buses de dirección externos del procesador 68000 son de 24 bits e internamente de 32 bits. Sin embargo, no es posible direccionar de forma efectiva más de 24 bits, por lo que ha optado en dejarlo en 24 bits, pudiendo direccionar 16 MB de memoria.

En la documentación del 68000 los datos están estructurados en memoria en palabras de 16 bits. Para simplificar el modelo, y dado que los buses de datos internos son de 32 bits, se ha optado por estructurar la memoria en palabras de 32 bits.

Además, otra simplificación que se ha realizado es tener solo 1 tamaño de palabra de 32 bits para los datos, por lo que no es posible elegir entre tamaño byte, palabra (16 bits en la documentación) o doble palabra (32 bits en la documentación) en los códigos de operación de las instrucciones. Por tanto, aunque las instrucciones sean de 16 bits, se almacenan en palabras completas de 32 bits, llenando con ceros la parte superior de la instrucción (bits 31 a 16) en el fichero de instrucciones.

La memoria tiene la señal ReadMem, para activar el acceso de lectura de memoria, y la señal WriteMem, para activar el acceso de escritura en memoria. Además, cuenta con la señal Dtack, señal externa del procesador 68000 según la documentación de Motorola en su sección 3 del manual de usuario [2].

Al comienzo de la simulación la memoria se inicializa. Cada dirección, que llega del bus de dirección a través del registro de direcciones de memoria, se comprueba para ver si es una dirección válida. Asimismo, cuando la CPU llega al estado Halted, la señal PrintMem se activa y genera (o hace append a) un fichero de texto de salida con el contenido de la memoria en ese instante de tiempo, llamado MEMORYDUMP.txt

### 3.3.8 Registro de instrucción

Fichero: IReg.vhd

El registro de instrucción es de 16 bits. Almacena la instrucción a ejecutar en la fase de ejecución en el estado Execute. Está conectado al bus2 de salida, y al busd como entrada. Se carga el registro con los primeros 16 bits del busd al activar con ‘1’ la señal LoadIReg en el flanco de subid. La salida (con la señal EnableDisp activa en ‘1’) sirve para utilizar el desplazamiento que tienen algunas instrucciones en sus últimos 8 bits (para las instrucciones de salto con desplazamiento en la propia instrucción).

### **3.3.9 Registro de dirección de memoria**

Fichero: MARreg.vhd

El registro de dirección de memoria, también llamado MAR (Memory Address Register) almacena la dirección de memoria a la que se va a acceder. El registro está conectado en su entrada por el busd y en su salida por el bus de direcciones addr.

La carga del registro de dirección de memoria se realiza mediante la señal LoadMar cuando está activa en ‘1’ en el flanco de subida de reloj. No hay una señal de EnableMAR, puesto que el bus de direcciones siempre está cableado a memoria desde el registro. Las direcciones deben ser de 24 bits.

### **3.3.10 Contador de programa**

Fichero: PCunit.vhd

El contador de programa, también llamado PC (program counter), es un registro de propósito específico y su función es almacenar la dirección de la instrucción que se va a ejecutar. Está conectado en su entrada por el busd, y en su salida por el bus1. Con la señal EnablePC activa en ‘1’, vuelca el contenido del PC en el bus1. Las direcciones tienen que ser de 24 bits, puesto que solo se cargan los primeros 24 bits del busd en el PC.

Volcar el contenido del PC en el bus1 puede utilizarse para cargar la dirección en el registro de dirección de memoria y obtener un dato o una instrucción, o bien puede utilizarse para sumar o restar el PC con un desplazamiento en la ALU y así dar un salto.

La señal PCop representa la operación que se va a realizar en el PC. Las operaciones posibles son 00, 01, 10, 11.

00. El PC no hace nada. Cuando cambia de estado, el pc no debe cambiar, a menos que se indique lo contrario.
01. RTI. Carga en el PC la dirección de la rutina de tratamiento de interrupción (siempre en la dirección 32 decimal de memoria)
10. Incrementa en 1 la dirección del PC.
11. Equivalente a LoadPC. Carga la dirección del busd en el PC.

### **3.3.11 Registro auxiliar**

Fichero: RegAux.vhd

Este módulo contiene 2 registros auxiliares de 32 bits, registro auxiliar 1 y registro auxiliar 2, conectados por su salida directamente a la ALU mediante el busAux1 y el busAux2. La entrada de estos registros está conectada al busd.

Cuenta con las señales LoadAux1 y LoadAux2 activas en nivel alto ('1') para cargar el contenido del busd en los registros auxiliares en el flanco de subida de reloj.

Las señales EnableAux1 y EnableAux2, activas en nivel alto ('1'), vuelcan el contenido de los registros en los buses BusAux1 y BusAux2. Para poder usar estos registros en la ALU, es necesario cambiar la primera y la segunda entrada de la ALU con las señales AluInputASel con valor '1' para el registro auxiliar 1, y AluInputBSel con valor '1' para el registro auxiliar 2.

Estos registros auxiliares sirven como almacenamiento temporal debido a los diferentes tipos de direccionamiento con los que cuenta este procesador. Al obtener el operando source (estados B del diagrama), se almacena en el registro auxiliar 2, y al obtener el operando destino, se almacena en el registro auxiliar 1. Cuando hay que mantener un dato durante más de un ciclo, recurrimos a estos registros para poder usarlos más adelante.

### 3.3.12 Banco de registros

Fichero: RegisterFile.vhd

Este módulo es el banco de registros de propósito general. Cuenta con 16 registros de 32 bits, divididos en 2 tipos, los registros de datos y los registros de dirección. La codificación de cada registro es la siguiente:

- 0000: D0
- 0001: D1
- 0010: D2
- 0011: D3
- 0100: D4
- 0101: D5
- 0110: D6
- 0111: D7
- 1000: A0
- 1001: A1
- 1010: A2
- 1011: A3
- 1100: A4
- 1101: A5
- 1110: A6. Utilizado como puntero de marco de pila.
- 1111: A7. Utilizado como puntero de pila.

El banco de registros tiene 1 entrada conectada al busd. Para activar la carga de un registro hay que seleccionar cuál es el registro destino con RD<sub>Sel</sub> y activar la señal de carga LoadRD activa en ‘1’ en flanco de subida.

El banco de registros tiene 2 salidas, conectadas al bus1 y bus2.

La señal RS1Sel selecciona el registro fuente que se volcará en el bus1. Para volcar el registro fuente por el bus1 hace falta activar en nivel alto (‘1’) la señal EnableRS1. Además hay que seleccionar la entrada de la ALU conectada a bus1 en la señal AluInputASel con valor 0.

La señal RS2Sel selecciona el registro fuente que se volcará en el bus2. Para volcar el registro fuente por el bus2 hace falta activar en nivel alto (‘1’) la señal EnableRS2. Además hay que seleccionar la entrada de la ALU conectada a bus2 en la señal AluInputBSel con valor 0.

### 3.3.13 Test

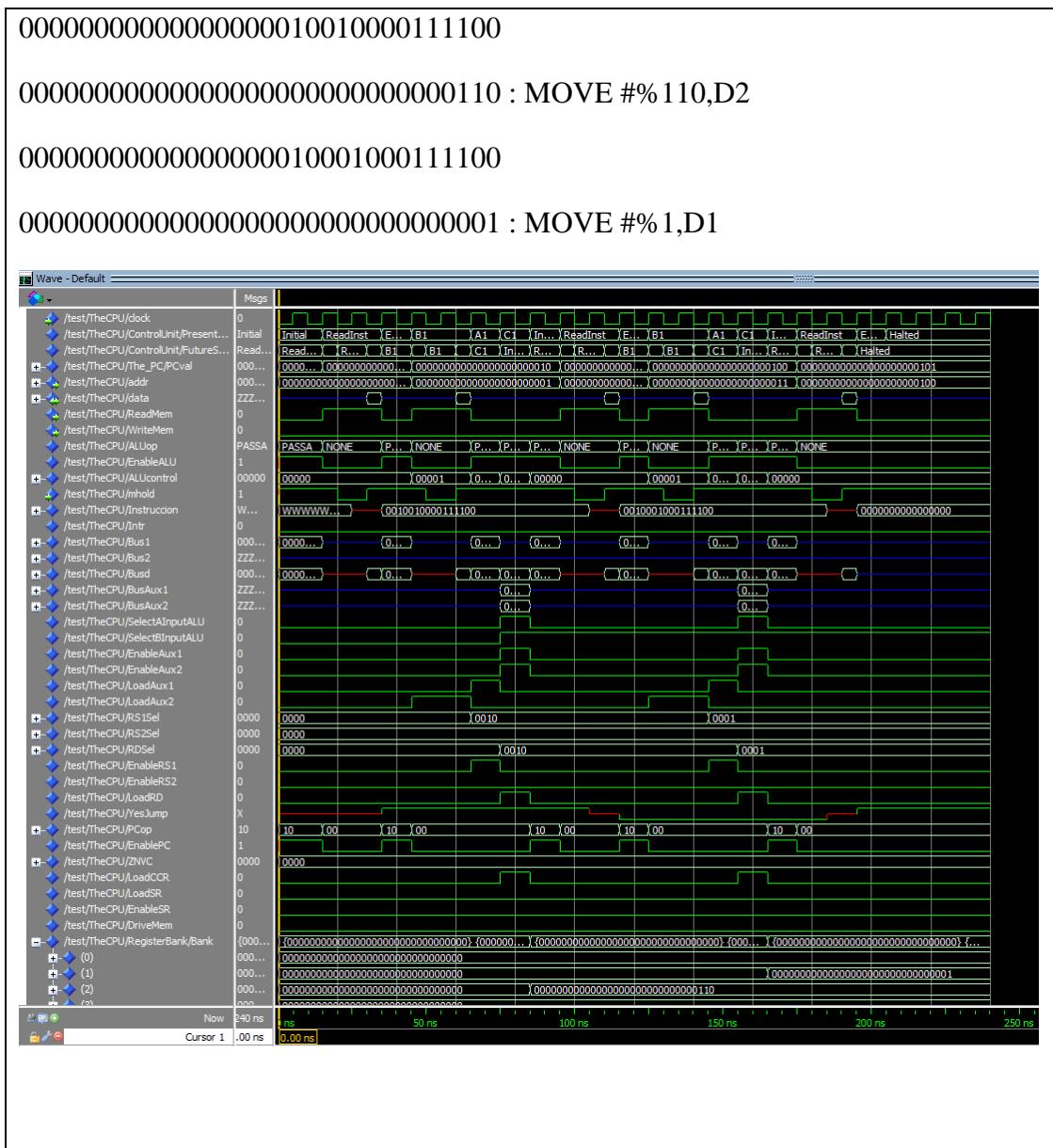
Fichero: Test.vhd

Este es el módulo que sirve para poder simular el procesador en ModelSim. Se hace un port mapping del procesador completo y del sistema de memoria y se asignan e inicializan las señales de la memoria, de reloj y de reset.

## 4. VERIFICACIÓN DEL MODELO

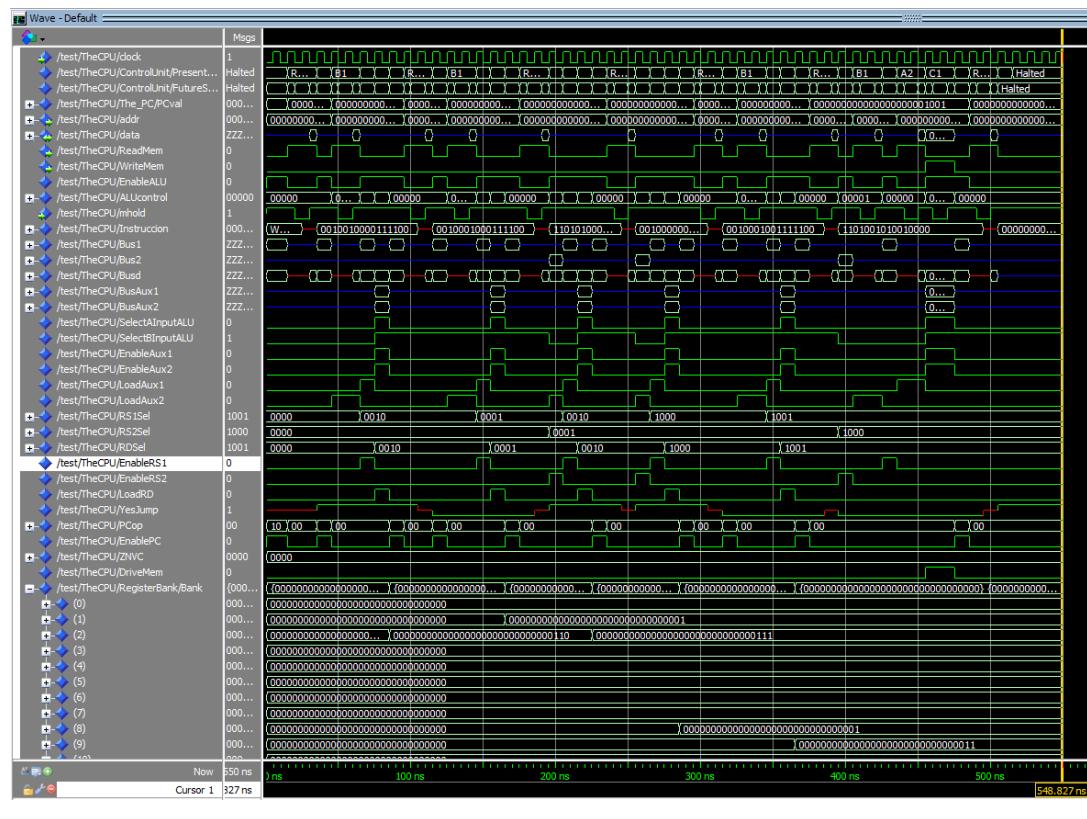
A continuación, se muestran diversos ficheros de test para verificar el modelo mediante ModelSim. Los ficheros y capturas de pantalla de alta resolución se adjuntan con la memoria

## 4.1 Move.txt



Descripción breve: El fichero move.txt tiene 2 ejemplos de la instrucción MOVE. La primera instrucción copia el número 6 decimal al registro D2. El número 6 es la segunda palabra de la instrucción. En la segunda instrucción transfiere un 1 al registro de datos D1.

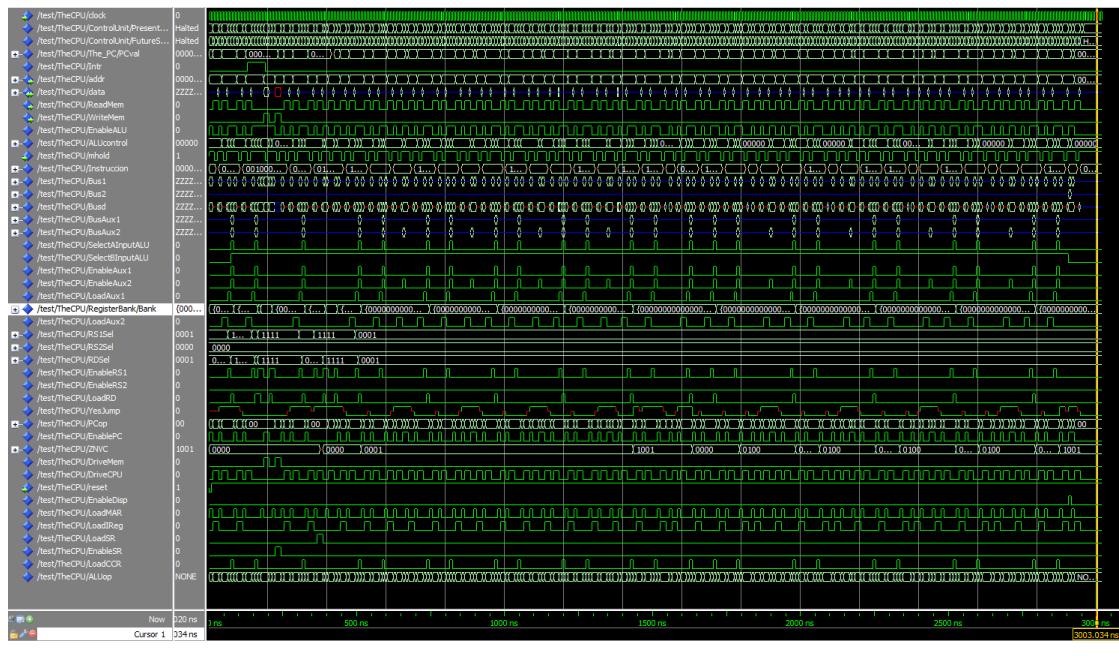
## 4.2 Add.txt

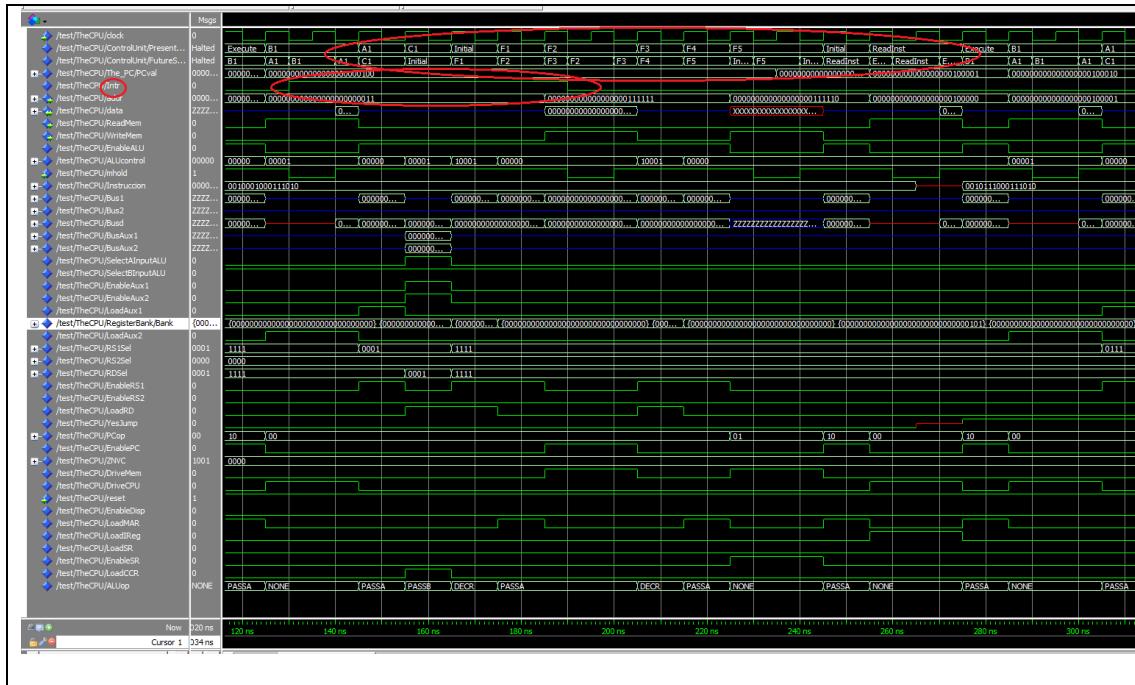


Descripción breve: En este fichero de test se comprueba si funciona la suma. Se asignan a 2 registros los valores binarios 1 y 110. Su suma da 111. Luego, mueve el resultado D1

al registro de dirección A0 y mueve un 11 a A1. Luego suma el contenido de las direcciones de memoria apuntadas por A0 y A1.

4.3 Bcc.txt



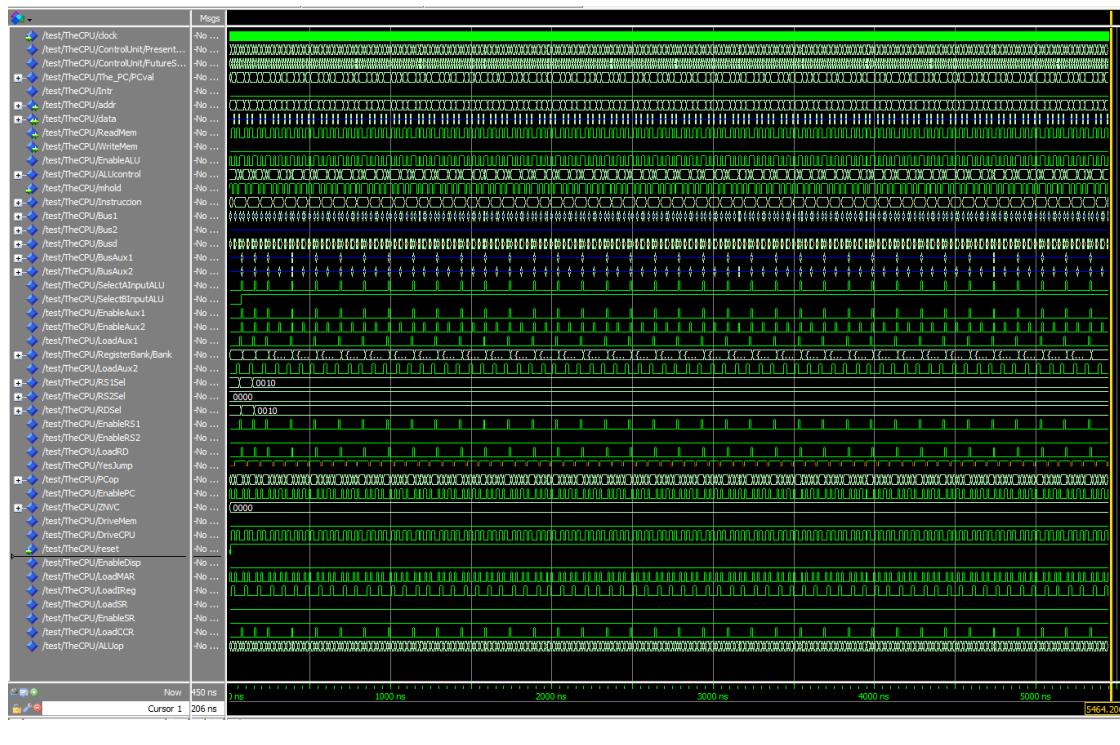


Descripción breve: En este fichero de prueba se comprueba que funcionan los saltos con condición, probando en este caso las instrucciones BNE (salta si no es igual) y BEQ (salta si es igual). Se ejecutan bucles gracias a la instrucción de salto incondicional BRA. Los saltos con condición permiten que el programa pueda terminar.

La segunda imagen es la misma que la primera, pero con un aumento. Se puede observar cómo alrededor de los 130 ns la señal de interrupción (Intr) se activa. La ejecución de la instrucción termina normal y entonces, cuando llega el estado inicial, transiciona a los estados de interrupción (F). Cuando termina de tratarse la interrupción, la ejecución vuelve a reanudarse por la instrucción que iba, terminando finalmente en el estado de Halted.

## 4.4 Bra.txt

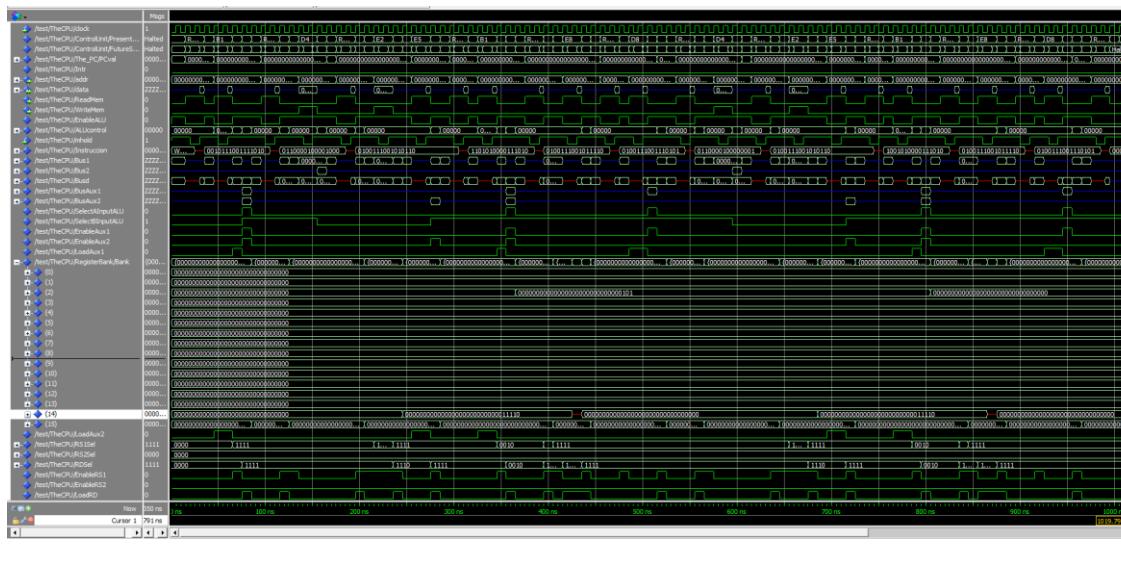
## Contenido del fichero:



Descripción breve: En este caso se demuestra la ejecución de la instrucción BRA (salto incondicional). Se realiza un salto incondicional hacia atrás (-8 posiciones de memoria), por lo tanto, entra en un bucle infinito, ya que siempre se encuentra con la instrucción BRA y no consigue llegar a la instrucción de STOP.

## 4.5 Link.txt

## Contenido del fichero:

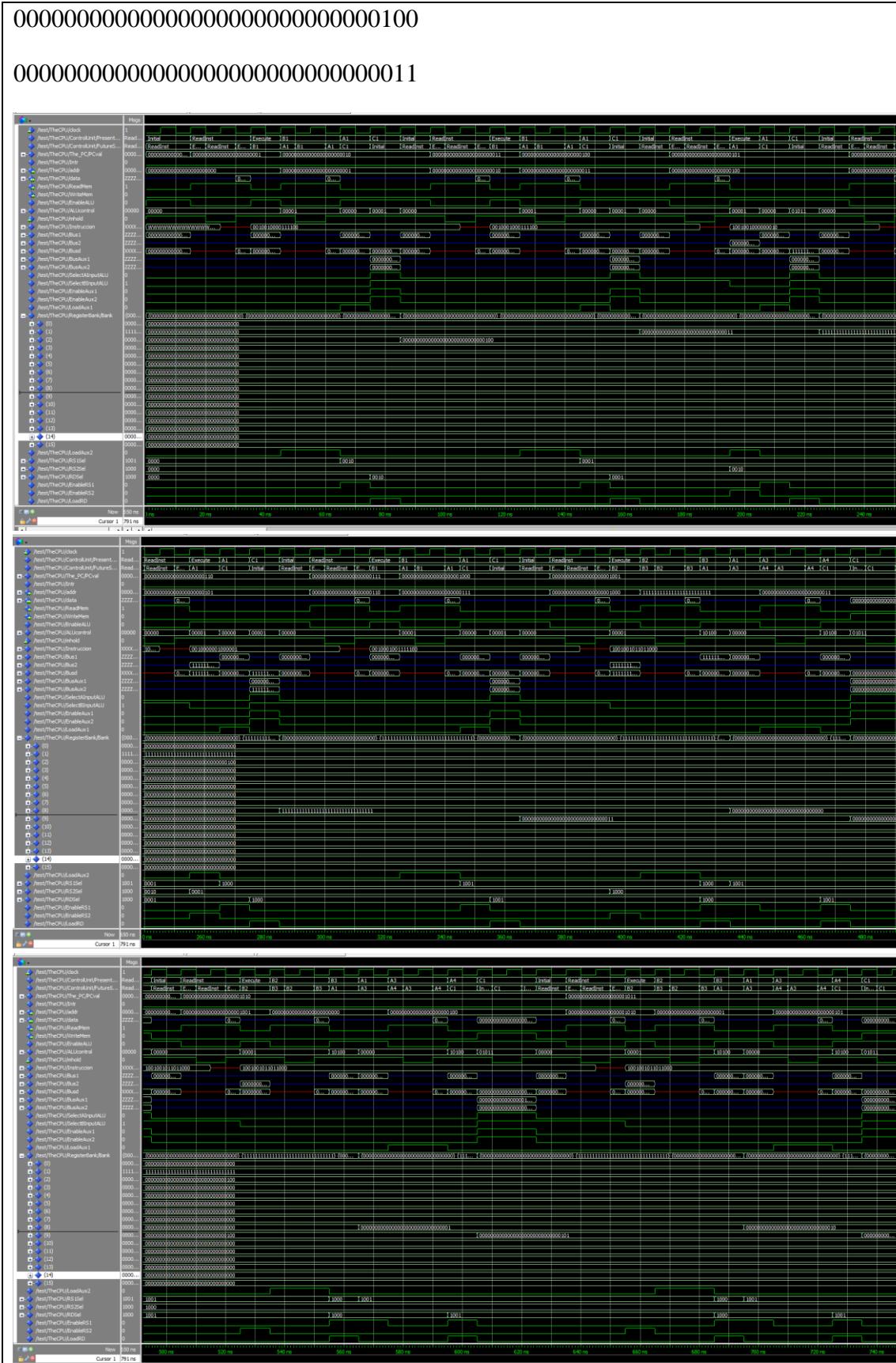


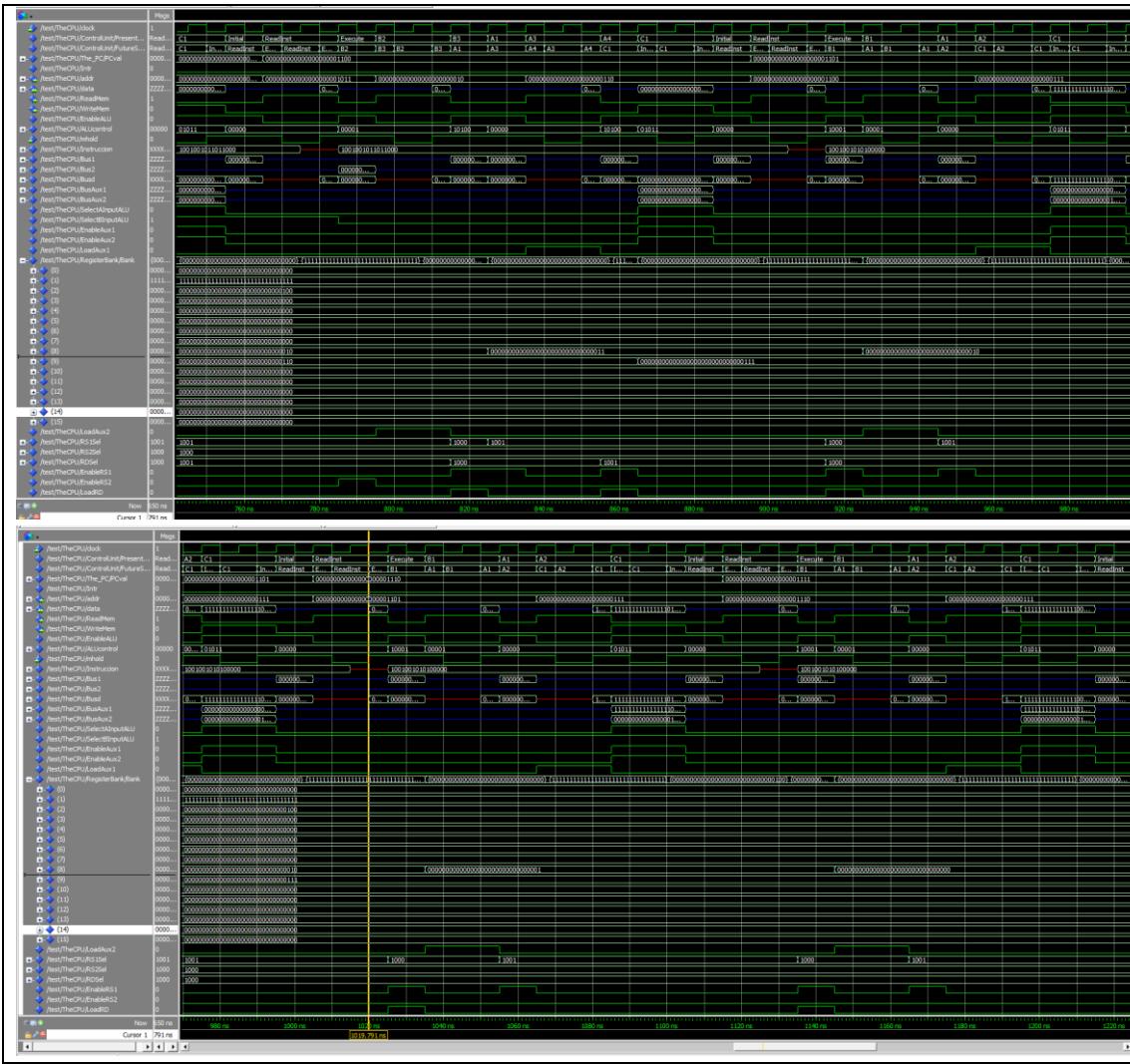
Descripción breve: En este fichero de test se prueba la instrucción LINK, UNLK, BSR y RTS. El programa comienza definiendo el puntero de pila. Salta con retorno a la subrutina 1, esta subrutina crea un marco de pila en A6 con espacio para 2 variables, realiza 2 sumas en D1 y retorna de la subrutina con RTS. A continuación entra en la subrutina 2, crea un

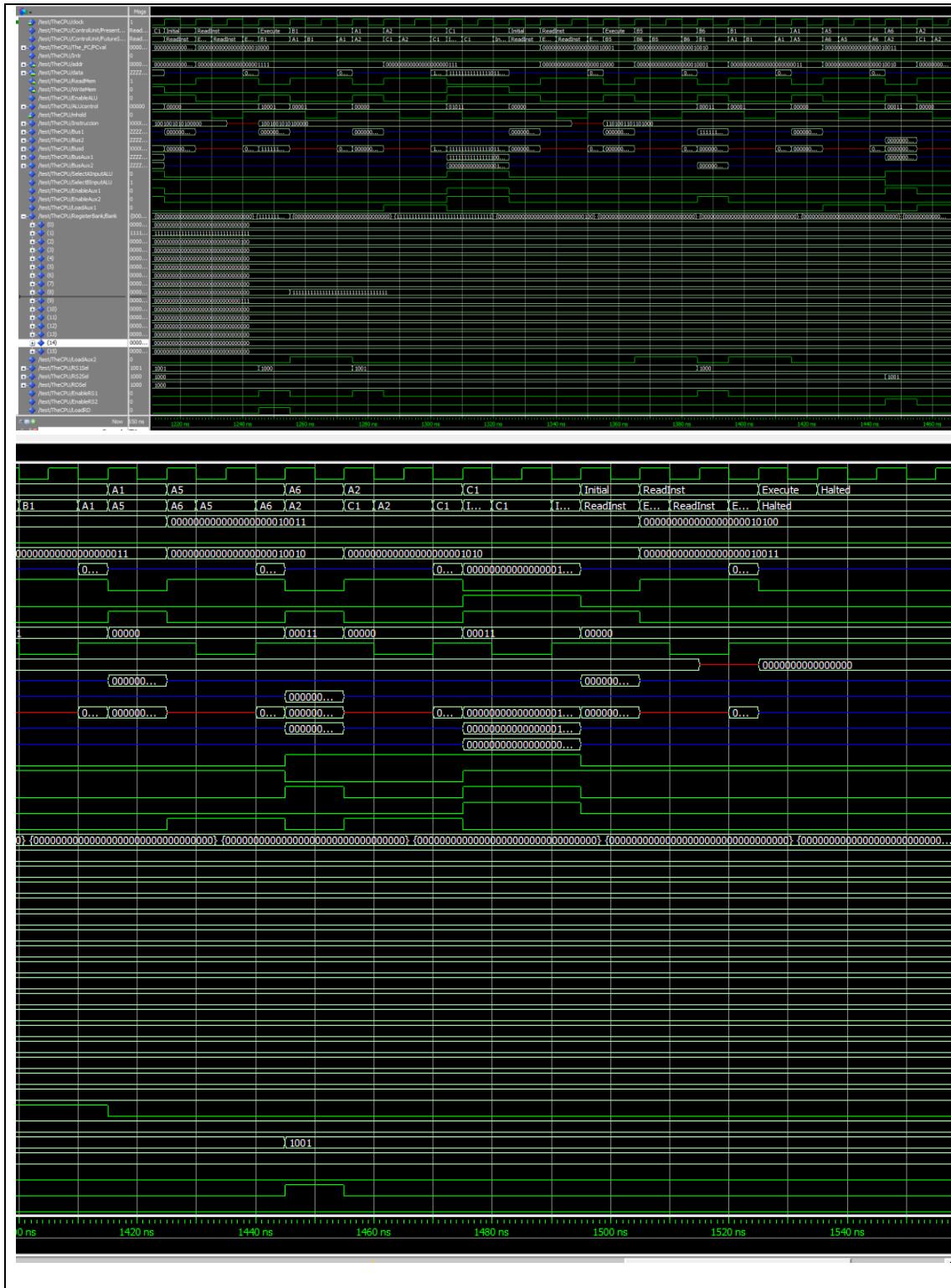
nuevo marco de pila con espacio para 2 variables, resta 2 veces el mismo registro D1, destruye el marco de pila y retorna a la subrutina. Finalmente, para la ejecución con STOP.

## 4.6 Sub.txt

## Contenido del fichero:







Descripción breve: Este es un programa sencillo para probar que funcionan las restas y los direccionamientos indirectos a registro de dirección con predecremento y postincremento.

## 5. ENSAMBLADOR

Para comprobar que el sistema realizado funciona correctamente, se ejecutan en código máquina programas con instrucciones pertenecientes al juego de instrucciones de la arquitectura del 68000. Para escribir rápidamente estos programas se utiliza un ensamblador, que permite traducir programas escritos en lenguaje ensamblador a lenguaje máquina.

El ensamblador realizado en este trabajo está programado en el lenguaje Perl. Cuenta con todos los códigos de operación y registros implementados en el modelo simplificado, guardados en tablas hash o diccionarios de clave-valor. Para reconocer las instrucciones en lenguaje ensamblador, se utilizan expresiones regulares. La documentación de referencia del lenguaje Perl explica cómo funcionan las expresiones regulares utilizadas en el código del ensamblador.

El programa empieza abriendo los ficheros de texto con los que va a trabajar. Empieza un bucle que analiza línea por línea todo el fichero. En cada línea, analiza si la instrucción en lenguaje ensamblador es reconocida mediante las expresiones regulares. Cada expresión regular puede reconocer uno o más códigos de operación. Si alguna expresión regular reconoce un código de operación, asigna las expresiones reconocidas y guardadas en variables string, para convertirlas en binario según sea registro fuente, registro destino, modo fuente, modo destino o código de operación. La comprobación de errores es muy primitiva, únicamente comprobando si el número de bits de cada campo de la instrucción en código máquina es el que debe ser. Una vez se obtienen todos los datos y se convierten a binario, se imprime el string en el fichero que se genera como resultado. El programa continúa con las siguientes líneas.

Para traducir los números en base decimal usados en las instrucciones en lenguaje ensamblador, se utiliza una función que los convierte a complemento a dos, pasando como parámetros de la función el signo y el valor del número. [7].

El funcionamiento del ensamblador es:

```
perl asm68k.pl test
```

Donde *test* es el fichero que se traducirá a lenguaje máquina. Se generan 2 ficheros. En *out.txt* se genera la salida del programa con las instrucciones traducidas a código máquina junto con comentarios con las instrucciones originales en lenguaje ensamblador. En *ERRORS.txt* se muestran los errores al traducir del fichero *test*.

## 6. CONCLUSIÓN

Uno de los puntos fuertes del sistema realizado es su diseño modular. Se pueden ir añadiendo más funcionalidades fácilmente con la arquitectura presente, o incluso modificar la propia arquitectura. Para añadir nuevas instrucciones simplemente habría que modificar la unidad de control. Si en el futuro se quisiera implementar otro tipo de instrucciones, como por ejemplo las de coma flotante, habría que añadir otro subsistema de coma flotante y modificar acordemente la unidad de control.

Es recomendable implementar cuanto antes el ensamblador, antes incluso de hacer la implementación, una vez se tiene pensado el formato de las instrucciones. Tener el ensamblador hecho antes de la implementación hace que se ahorre mucho tiempo de implementación y de comprobación de errores, pues mucho tiempo se dedica escribir o comprobar código máquina si no hay un ensamblador que realice esta operación automáticamente, además de que realizar un ensamblador es relativamente rápido. Escribir las instrucciones en código máquina no tiene nada de malo, el problema es que hay propensión a cometer errores a la hora de escribir las instrucciones. En mi caso, dejé la realización del ensamblador para el final y eso me hizo perder tiempo.

Es destacable lo que puede llegar a hacer el procesador con tan pocas instrucciones, solamente las más básicas y fundamentales, por lo que es un proyecto con mucho potencial de futuro. Ahora mismo sería capaz de ejecutar algunas subrutinas del proyecto de entrada/salida de la asignatura Arquitectura de Computadores.

Realizar este proyecto deja entrever cómo de complejo debe de ser diseñar una arquitectura y un procesador partiendo de cero. Los actuales procesadores comerciales tienen ciclos de desarrollo muy largos, que pueden conllevar varios años.

Los próximos cambios que se pueden hacer a este procesador son, el resto de instrucciones de la unidad de enteros. Por ejemplo, las instrucciones de bit clear y bit test/set, swap, los desplazamientos de bits a izquierda y derecha, o las operaciones de multiplicación y división. Otro campo en el que se puede avanzar es en completar la vectorización de interrupciones, implementando la máscara de interrupciones en el registro de estado, así como la prioridad de interrupciones de siete niveles. Más adelante se puede realizar la unidad de coma flotante o las instrucciones del supervisor, así como simular el resto de señales que vienen en la sección de señales de la documentación del procesador [2].

## **7. BIBLIOGRAFÍA**

- [1] Proyecto de Arquitectura de Computadores. Sistemas de Entrada/Salida. UPM. DATSI. Antonio Pérez Ambite, Santiago Rodríguez de la Fuente. Curso 2015/2016.
- [2] Motorola M68000 8-/16-/32-Bit Microprocessors User's Manual. Ninth Edition. Freescale Semiconductor, Inc. 1993.
- [3] Motorola M68000 Family Programmer's Reference Manual. Motorola. 1992.
- [4] Apuntes de CPU y el PicoComputador. UPM. DATSI. Sistemas Digitales. 20 de abril de 2013.
- [5] Low Risk Computer. The Control Unit of a CPU. UPM. DATSI. Digital Systems. April 30, 2017.
- [6] FPDP-11. UPM. DATSI. 20 de noviembre de 2017.
- [7] An Assembler for the LowRisk CPU. UPM. DATSI. 21 de mayo de 2017.