**Multiprocesadores: Práctica 4**

***Diseño de un controlador de cache con protocolo VI y escritura retardada***

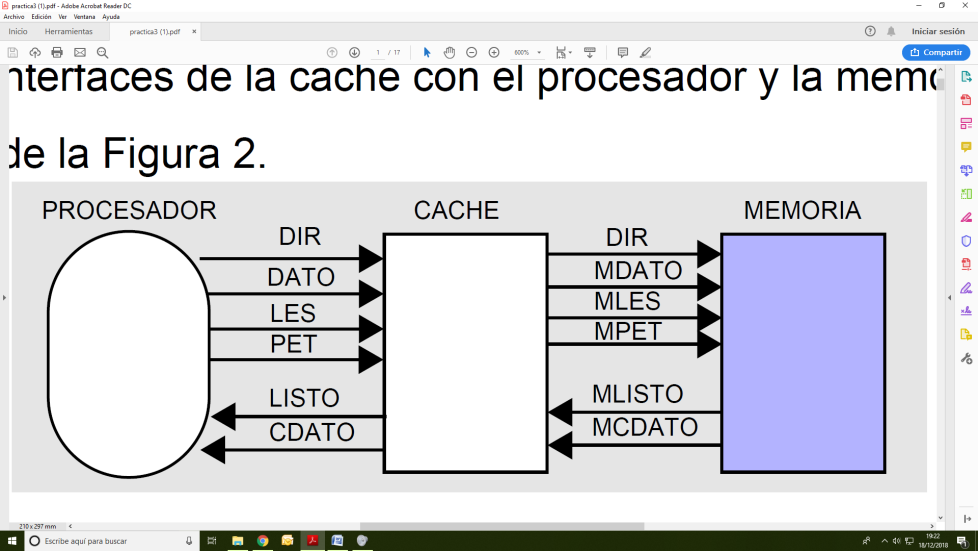
*Objetivo*: En esta memoria se expondrá el diseño de una cache bloqueante, con escritura retardada y asignación de contenedor en caso de fallo de escritura.

El protocolo usado para mantener la coherencia es el denominado VI (Válido/Inválido), que se corresponde a un protocolo de dos estados donde únicamente se guarda información de la validez del bloque en la cache.

En el diseño también se plantea la posibilidad de usar esta cache en un entorno multiprocesador, con lo que se han añadido los elementos necesarios para su correcto funcionamiento en estas condiciones.

Finalmente se han analizado las instrucciones más comúnmente usadas en entornos multiprocesadores para garantizar acceso exclusivo a una posición de memoria: test&set y load linked - store conditional (ll - sc).

**Interfaces de la cache**



*Figura 1. Interfaces entre procesador, cache y memoria*

En la figura 1 se muestran las 2 principales interfaces de la cache para trabajar en un entorno uniprocesador. Las funcionalidades de las diversas señales se definen en la siguiente tabla.

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Procesador - Cache | | | Cache - Memoria | | | |
|  | Dirección | Descripción |  | Dirección | Descripción |
| DIR | Entrada | Dirección del acceso | DIR | Salida | Dirección del acceso |
| DATO | Entrada | Dato a escribir | MDATO | Salida | Dato a escribir |
| LES | Entrada | Tipo de acceso (Lectura / Escritura) | MLES | Salida | Tipo de acceso (Lectura / Escritura) |
| PET | Entrada | Petición de acceso | MPET | Salida | Petición de acceso |
| LISTO | Salida | Cache lista. Indica si el anterior acceso ha acabado | MLISTO | Entrada | Memoria lista. Indica si la petición ha finalizado |
| CDATO | Salida | Dato de salida de la cache | MCDATO | Entrada | Dato de salida de memoria |

*Tabla 1. Señales de las interfaces de la cache. La dirección es referenciada según la cache*

**Organización de la cache**

La cache diseñada corresponde una cache con 16 contenedores y de mapeo directo, con lo que son suficientes 4 bits para diferenciar los contenedores. También se asume que en un ciclo se puede contestar a una petición en caso de acierto.

Se asume que las direcciones de memoria son de 16 bits. La granularidad de todos los accesos, así como el tamaño de bloque, se asumen que son 8 bits. De esta manera se simplifica el diseño. La primera suposición se basa en una memoria de 216 bytes (acceso a byte). En caso que la memoria tenga un tamaño superior, sería necesario dedicar más bits para las direcciones. La segunda petición se añade únicamente por simplicidad, de manera que traer o enviar un bloque se puede realizar en un ciclo. En el caso que esto no sea así, sería necesario disponer de estados intermedios que enviaran el bloque por partes, así como formateadores para poder recomponer el bloque una vez enviado. Por ejemplo, si el bus es de 8 bits y los bloques son de 32, serían necesarios 4 ciclos para enviar un bloque y se requeriría de un formateador para poder recomponer el bloque de 32 bits y escribirlo.

En el diseño propuesto, una cache se divide en tres campos: etiquetas, estado y datos. En el primero se guarda la dirección completa a la cual pertenece el bloque que está en cache. Cabe destacar que, al conocer la posición del contenedor, los últimos 4 bits no son necesarios, con lo que las etiquetas ocuparían 12 bits. El campo estado representa el estado del bloque en cache, es decir, si éste es válido o no. Este campo se implementa únicamente con un bit. Finalmente, el campo datos representa el contenedor donde se guarda el dato.

**Eventos del procesador**

En la tabla 2 se muestran los diferentes eventos del procesador, así como las diferentes acciones del controlador de la cache en respuesta a estas acciones.

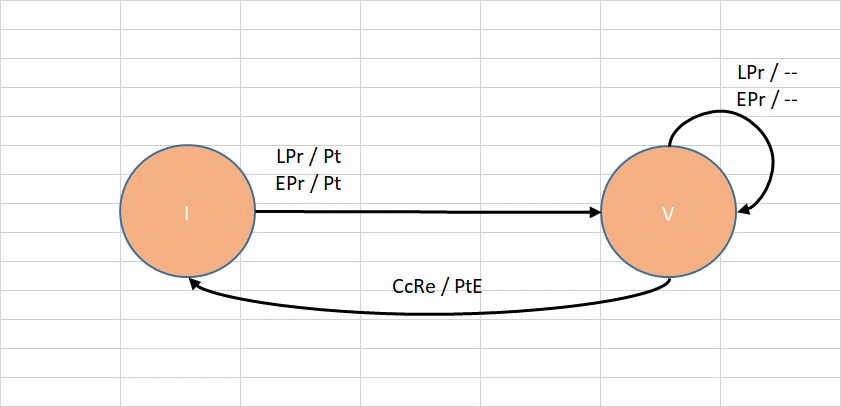
|  |  |
| --- | --- |
| Procesador | Controlador de cache |
| LPr: Petición de lectura | Pt: Petición de bloque |
| EPr: Petición de escritura | PtE: Petición de escritura (Reemplazo de bloque) |

*Tabla 2. Mensajes del procesador y del CC*

**Diagrama de estados y transiciones de un bloque en cache**

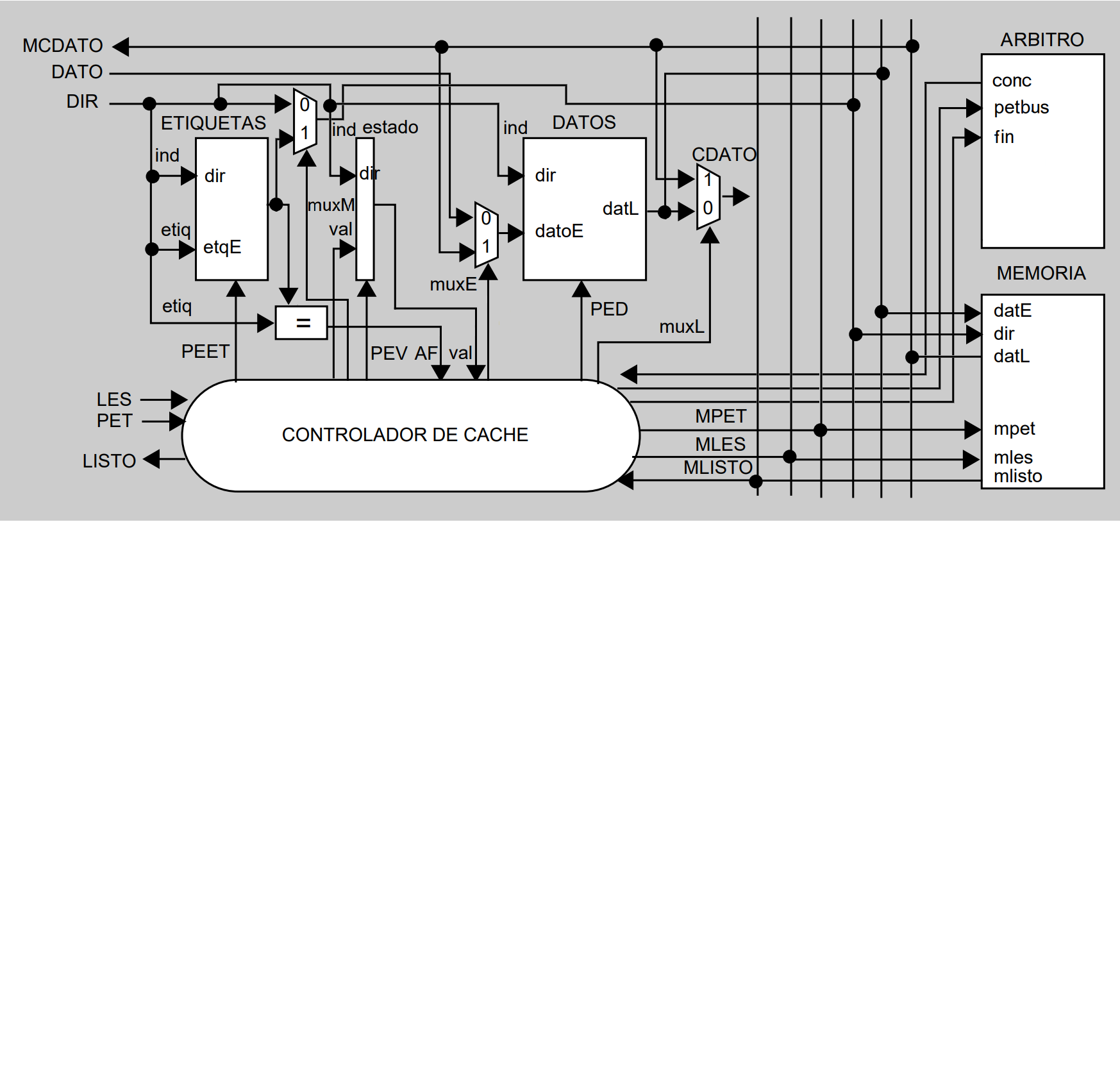
Para un bloque en estado invalido, cualquier acción que produzca fallo en la cache traerá el bloque a la cache, por lo que tanto en fallo de escritura como fallo de lectura este proceso empezará con cambiar el estado del bloque a válido. Tal y como se muestra en la figura 2, un LPr y un EPr generan una Pt (petición de bloque) y ponen el estado del bloque en válido.

Para un bloque en estado válido, cualquier acción de lectura o escritura mantendrá el bloque en estado válido, ya que estos accesos obtendrán un acierto en cache. Es solo en el caso del reemplazo de bloque donde se cambiará el estado del bloque a inválido y se procederá a escribir a memoria el bloque en cuestión.



*Figura 2. Diagrama de estados y transiciones de un bloque en cache*

**Camino de datos y controlador de la cache**



*Figura 3. Camino de datos de la cache*

En la figura 3 se observan todos los elementos antes comentados, así como diversos añadidos que se describen a continuación.

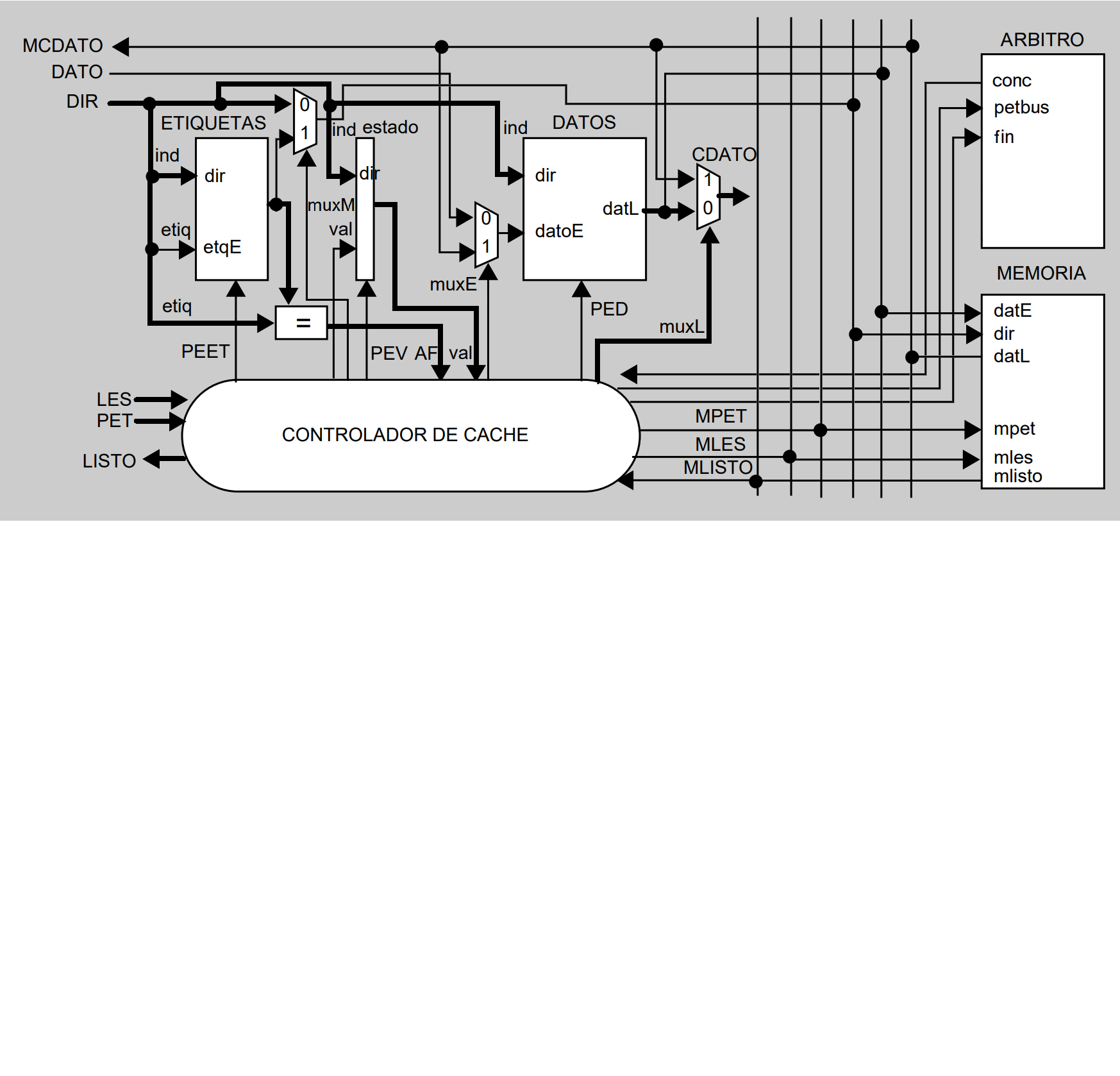
En el camino de datos se distinguen las interfaces comentadas en la figura 1 y en la tabla 1, donde la parte izquierda se corresponde al procesador y la parte derecha a memoria. También se aprecian los 3 campos en los que se divide la cache: etiquetas, estado y datos. En esta figura también se distinguen 3 multiplexores: muxM, muxE y muxL. muxM corresponde al multiplexor que decide qué dirección es la que hay que enviar a memoria en caso que haya que hacer una petición. El segundo (muxE) es el que se encarga de decidir quién puede guardar el dato en cache. Finalmente, muxL permite decidir si es la cache la que emite el dato al procesador o es la memoria. Estos multiplexores deberán ser dirigidos por el controlador de cache según las peticiones que reciba y el estado de la cache.

Por último, se ha añadido un elemento de arbitraje. Este elemento es útil en un entorno multiprocesador y define quién tiene acceso para usar la red de peticiones a memoria. Más adelante se definirá detalladamente este elemento, así como las señales de las que dispone.

**Flujo de información en el camino de datos**

En las siguientes figuras se muestra el flujo de información según los diversos estados de la cache.

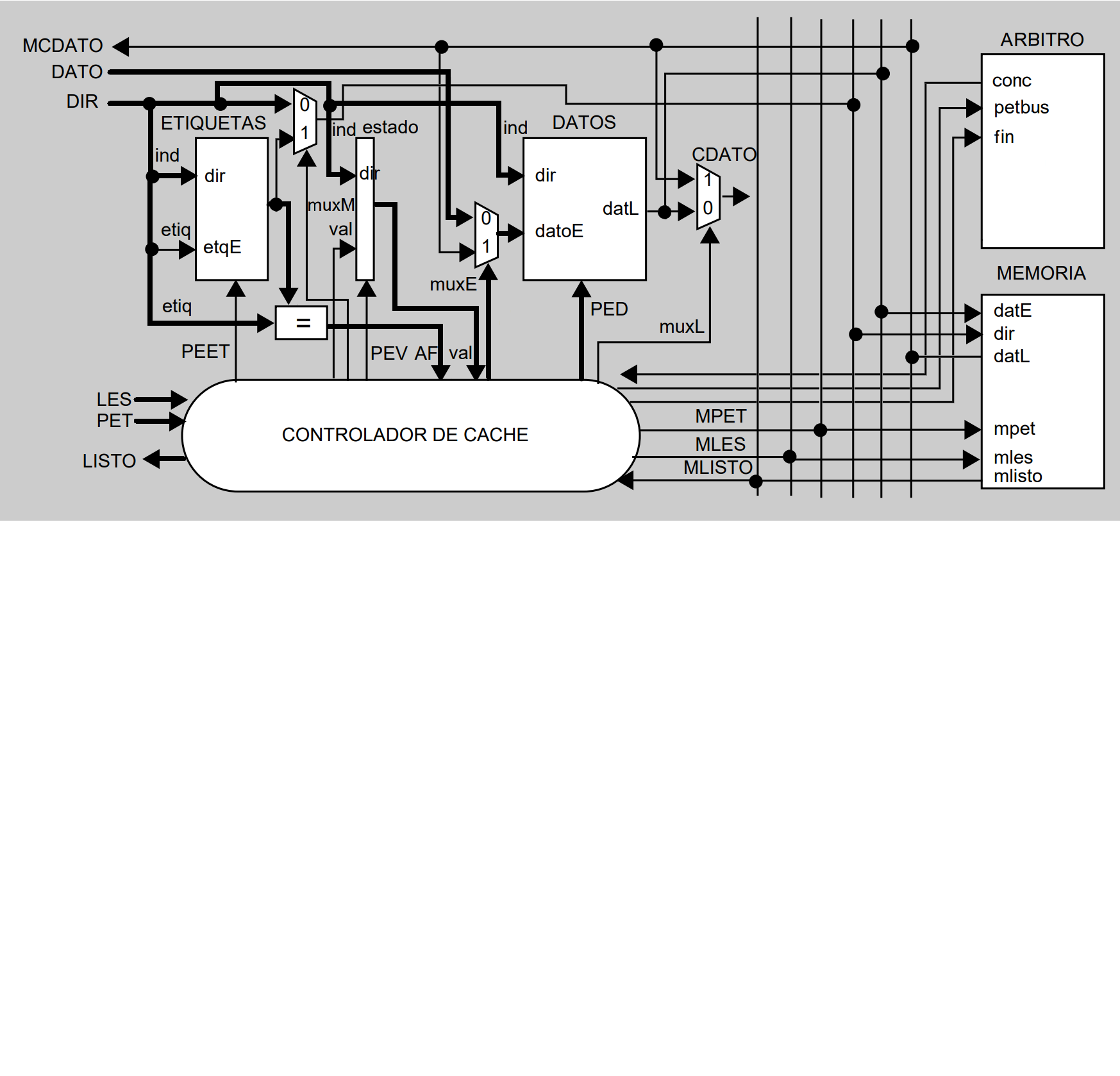
* Acierto en lectura



*Figura 4. Flujo de información en un acierto en lectura*

*Figura 4. Flujo de información en un acierto en lectura*

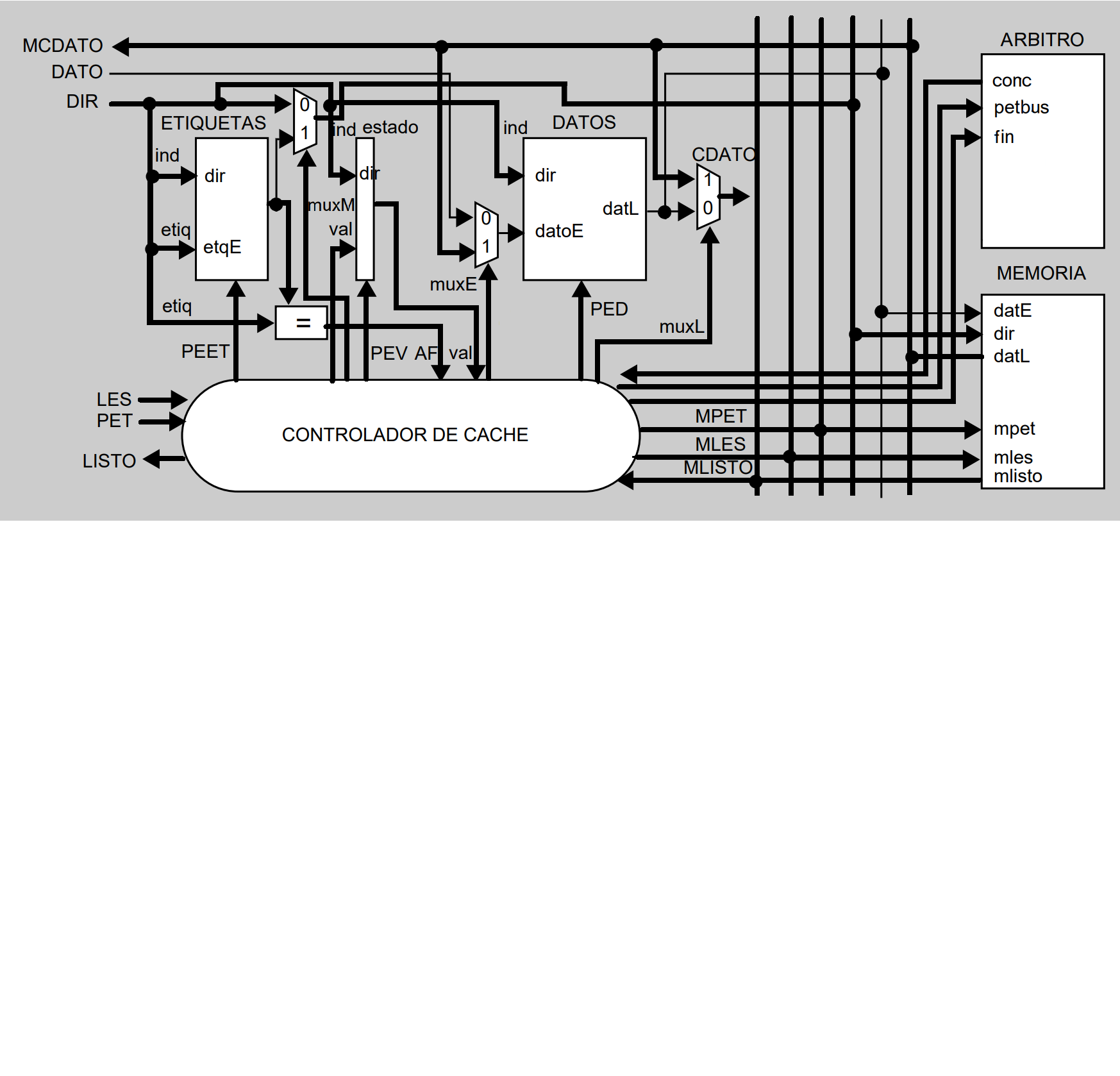
* Acierto en escritura



*Figura 5. Flujo de información en un acierto en escritura*

*Figura 5. Flujo de información en un acierto en escritura*

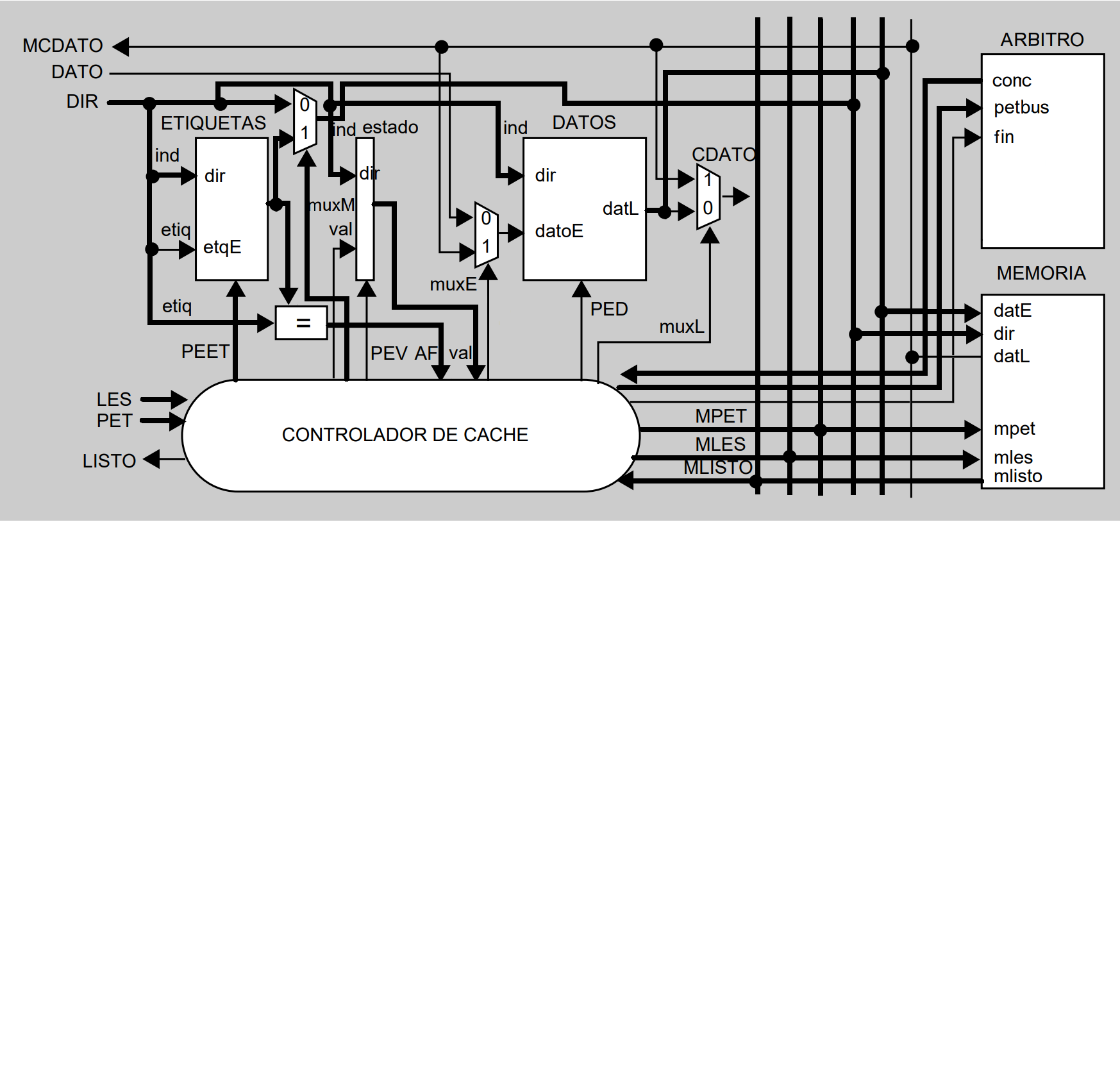
* Fallo en lectura



*Figura 6. Flujo de información en un fallo en lectura*

*Figura 6. Flujo de información en un fallo en lectura*

* Expulsión de bloque



*Figura 7. Flujo de información en una expulsión*

*Figura 7. Flujo de información en una expulsión*

En este diseño, las acciones del controlador pueden ser acumulativas, con lo que solo puede haber estos flujos de información en el camino de datos. Esto es, una acción del controlador puede incluir uno o varios de los flujos anteriormente mencionados (por ejemplo, un reemplazo se puede concebir como una expulsión más un fallo en lectura).

**Controlador de la cache**

**Descripción textual del controlador de cache**

Toda interacción entre el procesador y la cache empieza con la activación de la señal PET. El controlador, en este momento, utiliza los valores de las señales AF y val para determinar si es un acierto o un fallo en cache. En caso de acierto, la petición se puede servir en el mismo ciclo, con lo que se suministra el bloque (en caso de lectura) o se modifica el valor de la cache (en caso de escritura). Por el contrario, si es fallo, el controlador debe efectuar una petición a memoria. Primero, el controlador debe determinar si debe expulsar un bloque o no. Es decir, en caso de fallo, hay que mirar si el bloque que actualmente ocupa ese contenedor es válido o no. Si es válido, hay que proceder a la expulsión. En estas condiciones, el controlador debe efectuar una petición de PtE a memoria y enviar el bloque correspondiente. Para esta petición, el controlador activa la señal MPET y desactiva la señal MLES. También activa el multiplexor muxM para comunicarle a memoria qué dirección es la que se está enviando (la dirección final será la etiqueta que había en el campo de la cache (12 bits) más el identificador de bloque (4 bits)). Memoria responde activando la señal MLISTO cuando la transacción ha finalizado.

Si el bloque no era válido o ya ha sido expulsado, el controlador activa la señal MPET así como la MLES, indicando a memoria que quiere efectuar una lectura de un bloque. En este caso, desactiva el multiplexor muxM, de manera que la dirección es la que ha emitido el procesador. Como en el caso anterior, memoria activa la señal MLISTO cuando ha finalizado la transacción. En este caso, el controlador debe activar muxE, así como PEET, PEV y PED para modificar el estado de la cache a partir de memoria. En caso de lectura, también debe activar muxL. En caso de escritura, el controlador debe esperar un ciclo más para poder escribir el nuevo dato en la cache (a efectos prácticos, un fallo en escritura equivale a un fallo en lectura más un acierto en escritura).

El controlador activa la señal LISTO cuando la petición actual ha finalizado.

**Autómata del controlador de cache**

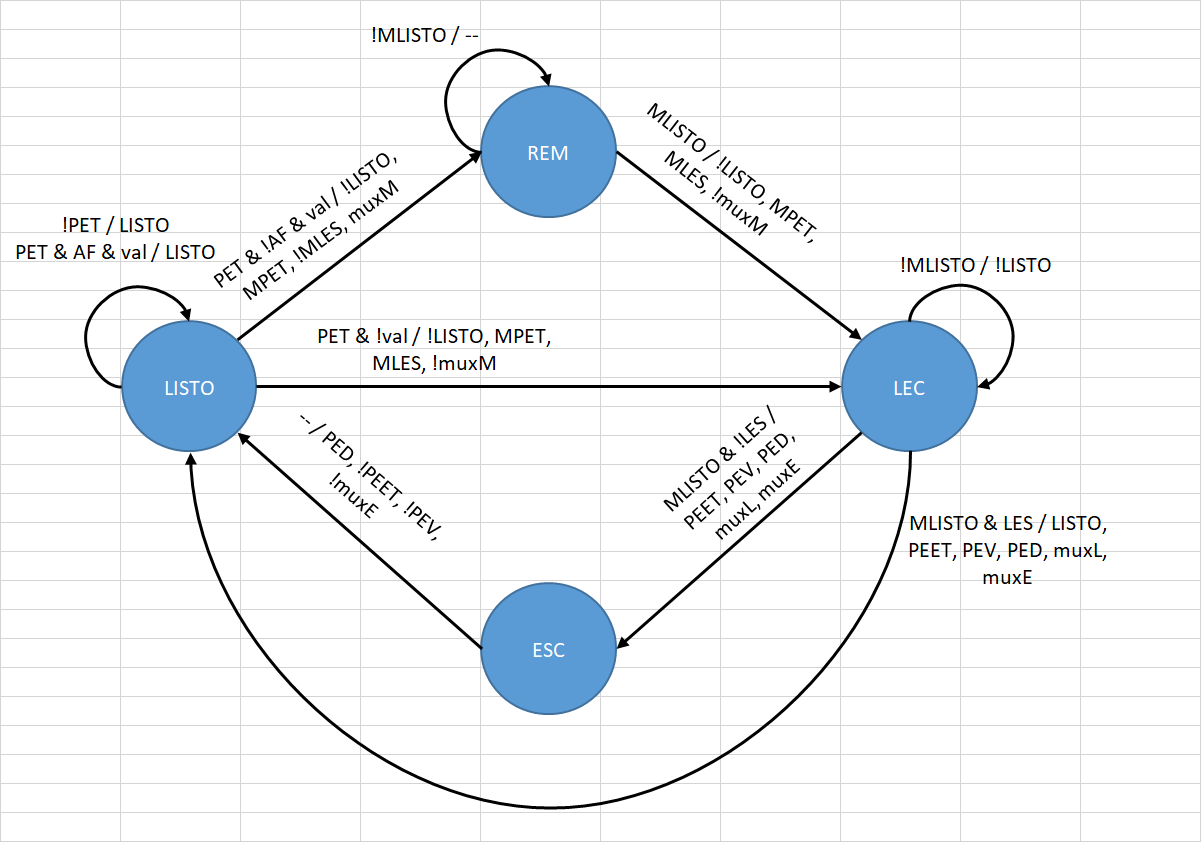
Como podemos ver en la figura 8, el autómata del controlador de cache dispone de 4 estados. El por defecto, LISTO, se encarga de analizar la petición y ver si se trata de un acierto o fallo, así como la validez del bloque en cuestión, y reaccionar según esos criterios. Si el acceso produce un fallo en la cache y el bloque es válido, hay que proceder a un reemplazo (estado REM), mientras que, si se trata de un bloque inválido, hay que traerlo a la cache y asignarle un contenedor (estado LEC).

El estado REM se encarga del reemplazo de bloque. Eso es, enviar el bloque antiguo a memoria para que sea escrito y proceder a la lectura del nuevo bloque. Así pues, este estado se mantiene hasta que el bloque ha sido escrito en memoria, y luego cambia al estado LEC.

El estado LEC, tal y como se ha explicado previamente, se encarga de traer un bloque de memoria a la cache y asignarle un contenedor. Este estado se mantiene hasta finalizar la transacción a memoria, y luego cambia. Si la petición se trataba de una lectura, la transacción ha finalizado, por lo que el estado del CC vuelve a LISTO, mientras que si se trataba de una escritura, cambia al estado ESC.

Finalmente, el estado ESC es el encargado de la escritura. Esto es, permite el paso del bloque proveniente de memoria y cambia el valor de una posición de este bloque al que ha determinado la instrucción de escritura.

En la figura 8 se representan estos cambios de estado, así como las señales que se activan y desactivan en función de la entrada.



*Figura 8. Diagrama de estados del controlador de cache uniprocesador*

**Diseño lógico del controlador de cache**

En las siguientes tablas se muestran las acciones correspondientes a cada estado del controlador de cache. Se asume que solo es necesario activar la petición a memoria durante un ciclo (si esto no es así, debería cambiarse la salida del estado REM para acomodar esta restricción).

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| |  |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | --- | | LISTO | | | | | | | | *PET* | *AF* | *val* | *LISTO* | *MPET* | *MLES* | *muxM* | | 0 | 0 | 0 | 1 | 0 | X | X | | 0 | 0 | 1 | 1 | 0 | X | X | | 0 | 1 | 0 | 1 | 0 | X | X | | 0 | 1 | 1 | 1 | 0 | X | X | | 1 | 0 | 0 | 0 | 1 | 1 | 0 | | 1 | 0 | 1 | 0 | 1 | 0 | 1 | | 1 | 1 | 0 | 0 | 1 | 1 | 0 | | 1 | 1 | 1 | 1 | 0 | X | X | | |  |  |  |  | | --- | --- | --- | --- | | REM | | | | | *MLISTO* | *MPET* | *MLES* | *muxM* | | 0 | 0 | X | X | | 1 | 1 | 1 | 0 | |
| |  |  |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | --- | --- | | LECTURA | | | | | | | | | *MLISTO* | *LES* | *LISTO* | *PEET* | *PEV* | *PED* | *muxL* | *muxE* | | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | | 1 | 0 | 0 | 1 | 1 | 1 | 1 | 1 | | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | | |  |  |  |  |  | | --- | --- | --- | --- | --- | | ESCRITURA | | | | | | *LISTO* | *PED* | *muxE* | *PEET* | *PEV* | | 1 | 1 | 0 | 0 | 0 | |

*Tablas 3, 4, 5, y 6. Diseños lógicos de salida de cada estado*

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| |  |  |  |  | | --- | --- | --- | --- | | LISTO (00) | | | | | *PET* | *AF* | *val* | *P\_EST* | | 0 | 0 | 0 | 00 | | 0 | 0 | 1 | 00 | | 0 | 1 | 0 | 00 | | 0 | 1 | 1 | 00 | | 1 | 0 | 0 | 10 | | 1 | 0 | 1 | 01 | | 1 | 1 | 0 | 10 | | 1 | 1 | 1 | 00 | | |  |  | | --- | --- | | REM (01) | | | *MLISTO* | *P\_EST* | | 0 | 01 | | 1 | 10 | |
| |  |  |  | | --- | --- | --- | | LEC (10) | | | | *MLISTO* | *LES* | *P\_EST* | | 0 | 0 | 10 | | 0 | 1 | 10 | | 1 | 0 | 11 | | 1 | 1 | 00 | | |  | | --- | | ESC (11) | | *P\_EST* | | 00 | |

*Tablas 7, 8, 9 y 10. Diseños lógicos del próximo estado en cada estado*

Tal y como se observa en las tablas 3 - 10, las acciones que debe tomar el CC son distintas según el estado en el que se encuentre. Por ello, para el diseño completo de la lógica de salida y de la lógica de próximo estado se usará un multiplexor de 4 entradas (correspondientes a cada uno de los 4 estados). La salida del multiplexor se controla mediante la señal EST o estado, correspondiente al estado actual.

**Organización del entorno multiprocesador**

Como se ha comentado anteriormente, la cache privada de un procesador es bloqueante, con escritura retardada y asignación de contenedor en caso de fallo. Las interfaces entre la cache y el procesador, así como la cache y la memoria son las que se definen en la figura 1. En un entorno multiprocesador, aun así, hacen falta más interfaces para implementar el protocolo que mantendrá la coherencia. En este caso, se ha diseñado la cache con un protocolo de observación, esto es, el controlador de coherencia mira constantemente las peticiones que circulan por el bus, con lo que sabe si debe actuar o no. Las interfaces que se añaden en este caso son las que se muestran en la siguiente tabla:

|  |  |  |  |
| --- | --- | --- | --- |
| Cache - Bus | | | |
|  | Dirección | Descripción |
| BDIR | Entrada | Dirección del acceso |
| BMPET | Entrada | Petición de acceso |

*Tabla 11. Interface entre la cache y el bus*

Tal y como se observa en la tabla 11, únicamente es necesario conocer si hay petición de acceso y la dirección del mismo.

**Organización de la red**

Se asume que la memoria solo tiene una interfaz para cualquier número de controladores. Por ello, cuando hay más de un controlador en el sistema, hace falta algún mecanismo para decidir quien usa esta interfaz. En el diseño propuesto, se trabaja con un árbitro, que recibe las peticiones de los diversos controladores y asigna la utilización de bus. En estas condiciones, el controlador debe disponer de una interfaz hacia este árbitro. Esta interfaz es la que se ve en la parte superior derecha de la figura 3 y se expone en la siguiente tabla:

|  |  |  |
| --- | --- | --- |
| Cache - Árbitro | | |
|  | Dirección | Descripción | |
| CONC | Entrada | Discriminador de qué controlador tiene acceso al bus | |
| PETBUS | Salida | Petición de acceso | |
| FIN | Salida | Finalización de acceso | |

*Tabla 12. Interface entre la cache y el árbitro*

En este sistema, el controlador pide acceso con la señal PETBUS cuando lo necesita. El árbitro activa la señal CONC cuando da acceso a ese controlador. El controlador activa la señal FIN cuando acaba la transacción. Cabe destacar que el número de señales es directamente proporcional al número de controladores, ya que son necesarias una señal CONC y PETBUS para cada controlador. La señal FIN puede ser única ya que su activación es mutualmente excluyente.

En cuanto a acciones del controlador, respecto a la tabla 2 únicamente se añade una acción: respuesta a una petición de bloque que acierta en cache. En este diseño, se ha decidido que cuando hay una observación que acierta en cache, sea la cache la que responda a la petición en vez de memoria. Para ello se dispone de una señal, denominada MOD que desactiva el suministro de memoria y permite el suministro entre caches (es la cache la que escribe en el bus, en vez de memoria). Esta señal es única ya que su activación es mutualmente excluyente.

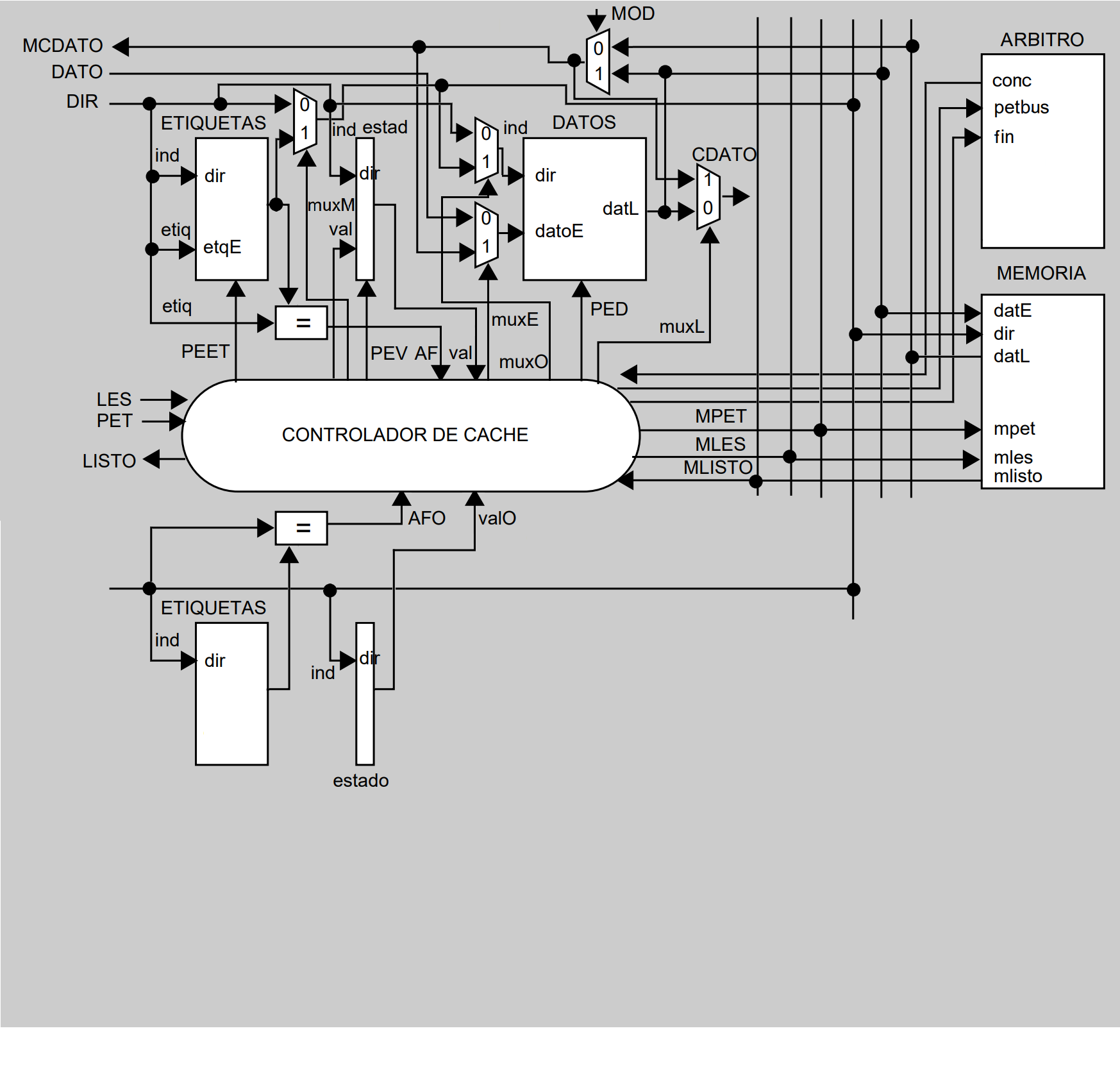
|  |  |  |  |
| --- | --- | --- | --- |
| Cache - Bus | | | |
|  | Dirección | Descripción |
| MOD | Salida | Suministro desde cache |

*Tabla 13. Adición de MOD a la interface entre la cache y el bus*

A efectos lógicos, esta señal se implementa como una OR de la MOD de todas las caches y, por simplicidad, no se dibuja en las siguientes figuras.

**Camino de datos**

En las figuras 3 - 7 se muestra el camino de datos, así como el flujo de información, en un controlador pensado para un entorno uniprocesador. En el caso en que el procesador y el observador deban acceder a la cache a la vez, se observa un riesgo en el camino de datos, ya que el acceso de lectura en los campos de la cache es único y, por lo tanto, es excluyente. Por ello, para permitir paralelismo entre estos dos elementos, en el diseño se han duplicado los campos de etiquetas y estado, de manera que cada uno tiene un puerto, aumentando así el paralelismo. La siguiente figura muestra esta distinción.



*Figura 9. Camino de datos del diseño final*

Tal y como se observa en la figura 8, replicar los campos de etiquetas y estado permite aumentar el paralelismo, ya que tanto el procesador como el observador puede acceder a la vez. Se debe remarcar que en caso que uno de los dos elementos deba efectuar o responder a una petición, se desactiva la señal LISTO, bloqueando el procesamiento de accesos hasta que el actual ha finalizado, con lo que el acceso al campo datos siempre será único. También se debe notar que en la figura 8 se han omitido los elementos de actualización de los elementos replicados por simplicidad, pero en la implementación todos los campos deben actualizarse coherentemente.

**Controlador de cache**

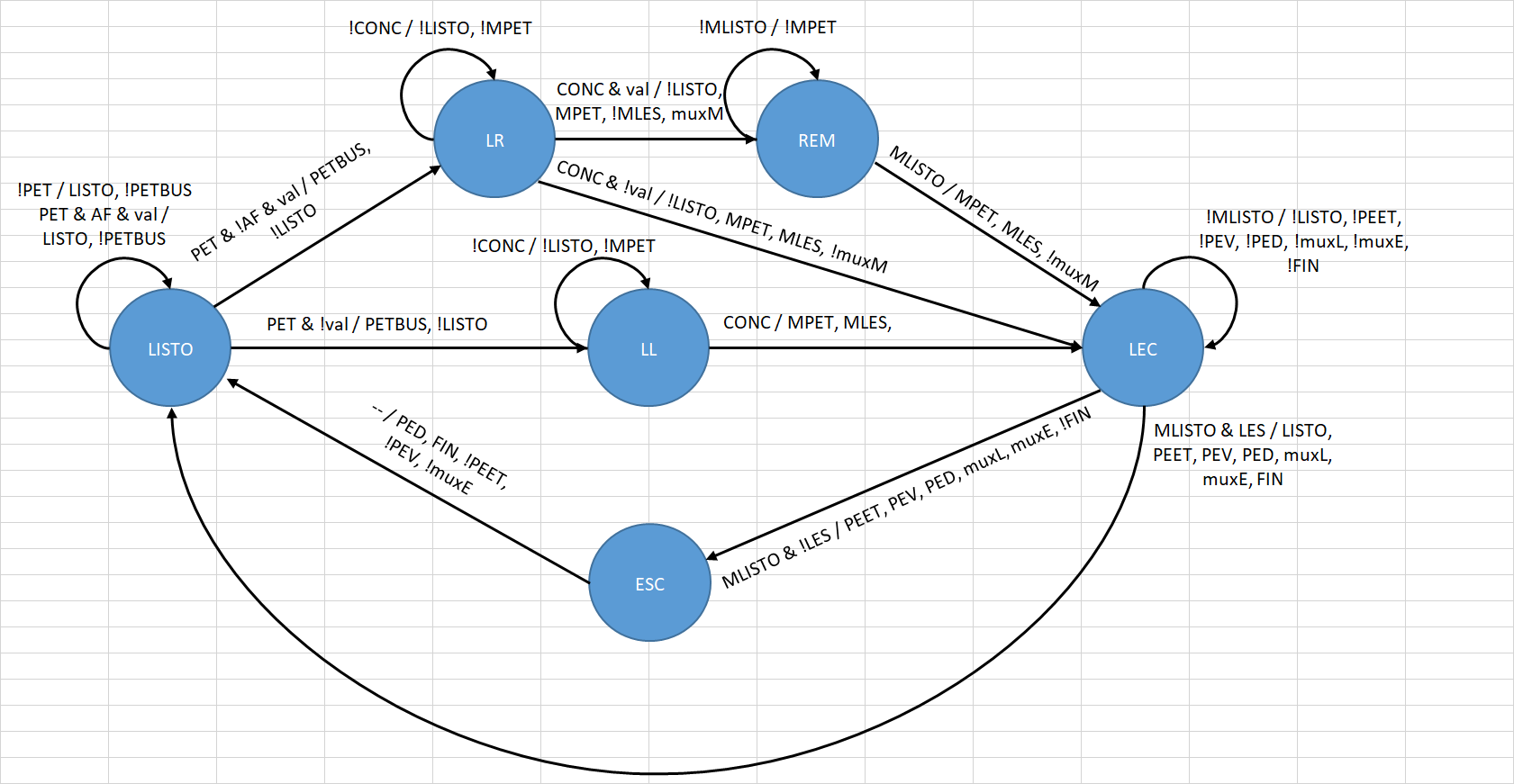
En el diseño final del controlador hay que distinguir dos elementos que cooperan: el agente procesador y el agente observador. El primero se encarga de responder a las peticiones del procesador y el segundo debe observar las peticiones del bus para actuar en caso de que sea necesario. Es necesario disponer de un elemento de comunicación entre los dos agentes, que será quien decida las acciones a tomar. En la red que se ha descrito, la concesión de bus determinará que agente tiene prioridad en caso de conflicto. El agente procesador es el mismo que el descrito para el caso uniprocesador. Las señales compartidas entre ambos agentes, esto es, la señal LISTO, val y PEV, deberán priorizarse según el agente que tenga prioridad.

**Agente procesador**

El agente procesador, tal y como se puede observar en la figura 10, consta de 6 estados, 4 de los cuales ya han sido descritos previamente. Para poder gestionar más de una petición a la vez, se le han añadido 2 estados que controlan las peticiones actuales. Estos estados son LR y LL.

El primero, LR, permite el paso de LISTO a REM una vez se le haya concedido el bus de memoria, ya que necesitará hacer una PtE (petición de escritura) a memoria con el bloque que va a ser reemplazado. Si durante su espera de la concesión de bus el bloque ha sido invalidado, una vez obtenga el bus, este pasará directamente a estado LEC, ya que no es necesario escribir el bloque a memoria si ya ha sido modificado por otro CC.

El segundo, LL, permite el paso de LISTO a LEC una vez se le haya concedido el bus de memoria, ya que necesitará hacer una Pt (petición de lectura) a memoria del bloque en cuestión. Así pues, solo cuándo se le concede el bus cambia de estado a LEC.

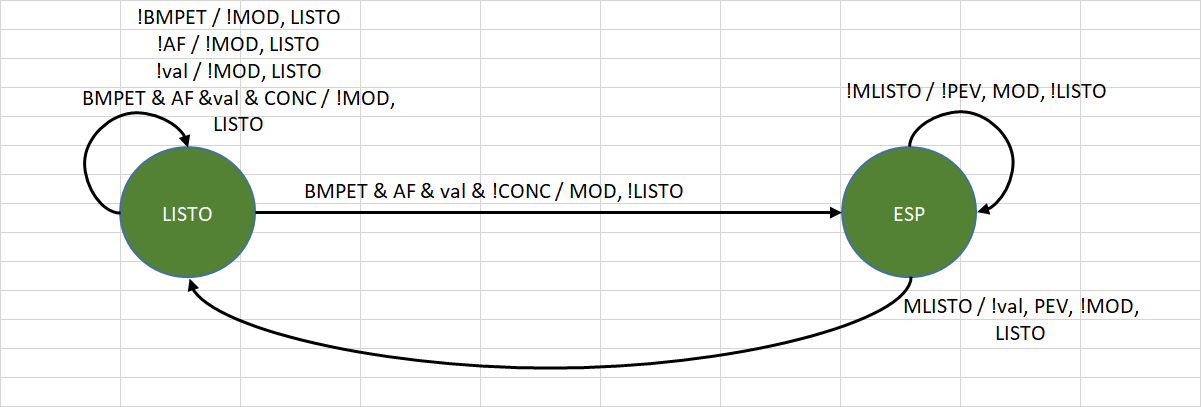


*Figura 10. Diagrama de estados del controlador de cache para el agente procesador*

**Agente observador**

El agente observador consta de 2 estados. Estos estados, representados junto a sus transiciones en la figura 11, son LISTO y ESP.

El estado LISTO, que es el inicial, observa los cambios y peticiones de otros CC, y actúa cuando ve una petición en el bus de memoria que coincide con un bloque válido en su cache. Entonces, al tratarse de un bloque contenido en su cache, lo subministrará directamente a la otra caché e invalidará su bloque después de subministrarlo.



*Figura 11. Diagrama de estados del controlador de cache para el agente observador*

**Descripción textual del autómata de observación**

El autómata de observación solo actúa cuando el controlador no tiene la concesión de bus (CONC desactivada). Cuando se da este caso, el autómata observa por el bus las peticiones que están ocurriendo. Si alguna petición referencia un bloque que tiene almacenado en cache, el autómata desactivará la señal LISTO y accederá al campo de datos de la cache para obtener el bloque que se pide. También activará la señal MOD, dando a entender al controlador que ha efectuado la petición que memoria no tiene el bloque. Por simplicidad de cara a otras peticiones y al diseño, se responde con el bloque cuando memoria activa la señal MLISTO. Se asume que la activación de la señal MLISTO es más lenta que el acceso a cache, de manera que no hay que cambiar el autómata procesador. Cuando se ha servido el bloque, se invalida la copia en la propia cache y se reactiva la señal LISTO.

**Diseño lógico del controlador en un entorno multiprocesador**

Con tal de diseñar las acciones que debe hacer el controlador de cache, se han dispuesto en las siguientes tablas la lógica correspondiente a la salida del controlador, así como la lógica de próximo estado.

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| |  |  |  |  | | --- | --- | --- | --- | | LISTO (000) | | | | | *PET* | *AF* | *val* | *P\_EST* | | 0 | 0 | 0 | 000 | | 0 | 0 | 1 | 000 | | 0 | 1 | 0 | 000 | | 0 | 1 | 1 | 000 | | 1 | 0 | 0 | 011 | | 1 | 0 | 1 | 001 | | 1 | 1 | 0 | 011 | | 1 | 1 | 1 | 000 | | |  |  |  | | --- | --- | --- | | LR (001) | | | | *CONC* | *val* | *P\_EST* | | 0 | 0 | 001 | | 0 | 1 | 001 | | 1 | 0 | 100 | | 1 | 1 | 010 | | |  |  | | --- | --- | | REM (010) | | | *MLISTO* | *P\_EST* | | 0 | 010 | | 1 | 100 | |
| |  |  | | --- | --- | | LL (011) | | | *CONC* | *P\_EST* | | 0 | 011 | | 1 | 100 | | |  |  |  | | --- | --- | --- | | LEC (100) | | | | *MLISTO* | *LES* | *P\_EST* | | 0 | 0 | 100 | | 0 | 1 | 100 | | 1 | 0 | 101 | | 1 | 1 | 000 | | |  | | --- | | ESC (101) | | *P\_EST* | | 000 | |

*Tablas 14, 15, 16, 17, 18 y 19. Lógica del próximo estado en un entorno multiprocesador con asignación de bus*

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| |  |  |  |  |  | | --- | --- | --- | --- | --- | | LISTO | | | | | | *PET* | *AF* | *val* | *LISTO* | *PETBUS* | | 0 | 0 | 0 | 1 | 0 | | 0 | 0 | 1 | 1 | 0 | | 0 | 1 | 0 | 1 | 0 | | 0 | 1 | 1 | 1 | 0 | | 1 | 0 | 0 | 0 | 1 | | 1 | 0 | 1 | 0 | 1 | | 1 | 1 | 0 | 0 | 1 | | 1 | 1 | 1 | 1 | 0 | | |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | | LR | | | | | | | *CONC* | *val* | *LISTO* | *MPET* | *MLES* | *muxM* | | 0 | 0 | 0 | 0 | X | X | | 0 | 1 | 0 | 0 | X | X | | 1 | 0 | 0 | 1 | 1 | 0 | | 1 | 1 | 0 | 1 | 0 | 1 | | |  |  |  |  | | --- | --- | --- | --- | | REM | | | | | *MLISTO* | *MPET* | *MLES* | *muxM* | | 0 | 0 | X | X | | 1 | 1 | 1 | 0 | |
| |  |  |  |  | | --- | --- | --- | --- | | LL | | | | | *CONC* | *LISTO* | *MPET* | *MLES* | | 0 | 0 | 0 | X | | 1 | 0 | 1 | 1 | | |  |  |  |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | --- | --- | --- | | LECTURA | | | | | | | | | | *MLISTO* | *LES* | *LISTO* | *PEET* | *PEV* | *PED* | *muxL* | *muxE* | *FIN* | | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | | 1 | 0 | 0 | 1 | 1 | 1 | 1 | 1 | 0 | | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | | |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | | ESCRITURA | | | | | | | *LISTO* | *PED* | *muxE* | *PEET* | *PEV* | *FIN* | | 1 | 1 | 0 | 0 | 0 | 1 | |

*Tablas 19, 20, 21, 22, 23 y 24. Lógica de salida en un entorno multiprocesador con asignación de bus*

Como en el caso uniprocesador, la salida final se compondrá a partir de un multiplexor. Este multiplexor será de 6 entradas y se controlará a partir de la señal EST o estado.

Finalmente, también hace falta determinar las acciones que tomará el agente observador, con lo que se facilitan las siguientes tablas que determinan la lógica de actuación del autómata.

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| |  |  |  | | --- | --- | --- | | LISTO (0) | | | | *BMPET&AF&val* | *CONC* | *P\_EST\_O* | | 0 | 0 | 0 | | 0 | 1 | 0 | | 1 | 0 | 1 | | 1 | 1 | 0 | | |  |  | | --- | --- | | ESP (1) | | | *MLISTO* | *P\_EST\_O* | | 0 | 1 | | 1 | 0 | |
| |  |  |  |  | | --- | --- | --- | --- | | LISTO | | | | | *BMPET&AF&val* | *CONC* | *MOD* | *LISTO* | | 0 | 0 | 0 | 1 | | 0 | 1 | 0 | 1 | | 1 | 0 | 1 | 0 | | 1 | 1 | 0 | 1 | | |  |  |  |  |  | | --- | --- | --- | --- | --- | | ESP | | | | | | *MLISTO* | *val* | *PEV* | *MOD* | *LISTO* | | 0 | X | 0 | 1 | 0 | | 1 | 0 | 1 | 0 | 1 | |

*Tablas 25, 26, 27 y 28. Lógica de próximo estado (superiores) y salida (inferiores) del autómata observador*

Como ocurría en los casos anteriores, la salida pasará por un multiplexor (en este caso de 2 entradas) que determinará, según el estado, la salida correspondiente.

**Instrucciones**

Para garantizar acceso exclusivo, implementaremos el par de instrucciones load linked y store conditional del siguiente modo:

|  |  |  |
| --- | --- | --- |
| Load linked: LL Rd, X(Rf) | Store Conditional: SC Rfd, X(Rf) | Observación de SC |
| Rdv=M[X+Rfv]  RLDv = X+Rfdv  RLRv = 1 | if(RLDv = X + Rfv and RLRv = 1) then  M[X+Rfv] = Rfdv  Rfdv = 1  else Rfdv = 0  RLRv = 0 | if(RLDv = dirtrans and RLRv = 1) then RLRv = 0 |

*Tabla 29. Implementación del par load linked y store conditional*

Donde el superíndice v indica valor, RLD será el registro donde se almacene la dirección a la que accede un load linked y RLR será el bit de reserva.

Tal y como se observa en la tabla 29, cada instrucción solo requiere de una transacción a memoria. Como ya se ha garantizado durante el diseño del protocolo que estas transacciones son atómicas, la adición de este par de instrucciones es inmediata y no supone ningún problema adicional. Hay que tener en cuenta, aun así, que es el controlador de cache el encargado de gestionar convenientemente los registros RLR y RLD. Cabe destacar que, con tal de diferenciar los diferentes tipos de peticiones, es necesario añadir más interfaces entre procesador y controlador de cache, de manera que las peticiones ya no son solo lectura y escritura, sino que se añaden las instrucciones ll y sc.

Para la implementación de la instrucción Test&set, y utilizando las mismas variables, será implementada como:

|  |
| --- |
| Test&set: TS Rd, X(Rf) |
| Rd = M[X+Rfv]  M[X+Rfv] = 1 |

*Tabla 30. Implementación de la instrucción test&set*

Contrariamente al caso anterior, la tabla 30 muestra como la implementación de la instrucción test&set implica 2 transacciones a memoria. Sin embargo, observando con detenimiento el diseño propuesto, se ve que un fallo en escritura se comporta de manera parecida a un test&set: primero se trae el bloque y luego se escribe. Por ello, solo hace falta modificar el controlador de coherencia para que, en este tipo de instrucciones, devuelva al procesador el dato traído. También es necesario notar que si la instrucción test&set acierta en cache, la operación no es atómica. Por ello, se propone añadir un nuevo tipo de petición (correspondiente a test&set) de manera que el controlador ignore si se acierta o no en cache y siempre falle. En este caso, si la cache disponía del bloque en cuestión, primero lo reemplazará (actualizando el valor en memoria), y luego lo traerá como si se tratara de un fallo en escritura. De esta manera, se consigue que la instrucción siempre sea atómica, independientemente del estado de la cache.

**Análisis de caso de uso**

A partir de la tabla 31 se analizarán las instrucciones anteriormente mencionadas y su comportamiento en un caso de uso real.

|  |  |
| --- | --- |
| Load linked - Store conditional | Test&Set |
| 1. P1 LL llave | 1. P1 TS llave |
| 2. P1 SC llave(1) | 2. P1 store exc(23) |
| 3. P1 store exc(23) | 3. P3 TS llave |
| 4. P1 store llave(0) | 4. P2 TS llave |
| 5. P2 LL llave | 5. P3 TS llave |
| 6. P3 LL llave | 6. P1 store llave(0) |
| 7. P2 SC llave(1) | 7. P2 TS llave |
| 8. P2 LL llave |  |
| 9. P3 SC llave(1) |  |

*Tabla 31. Análisis de caso de uso de las instrucciones ll - sc (izquierda) y test&set (derecha)*

El caso de uso se basa en 3 procesadores intentando obtener acceso a una región exclusiva (variable *exc*) a partir de la variable de exclusividad *llave*. Como se observa en la tabla 31, un procesador obtiene la exclusividad cuando ejecuta con éxito la secuencia de instrucciones LL y SC. Cuando acaba de usar la región, la libera restableciendo la llave. En este momento, los otros procesadores intentan acceder de manera simultánea a esta región. Se observa en las instrucciones 5 - 9 que los procesadores 2 y 3 intentan obtener la exclusividad, pero debido al entrelazado de peticiones no la obtienen nunca. Este efecto se denomina inanición. La más mínima desincronía del entrelazado permitiría a un procesador obtener acceso.

En el caso del test&set, este efecto no se observa: cuando la llave esta libre, el primer procesador que la pide es la que la obtiene. Esto permite evitar el problema de la inanición, haciendo que esta instrucción sea mucho más fiable en este sistema. En las instrucciones 3 - 5 se observa como los otros procesadores compiten por un acceso que aun no ha sido liberado, con lo que repiten la instrucción TS hasta conseguirlo.

**Anexo 1: Descripcion VHDL del controlador de cache**

library ieee;

use ieee.std\_logic\_1164.all;

entity controlador is

generic (retardo: time := 4 ns);

port (

reloj, pcero: in std\_logic;

-- Pcero senal de puesta a cero

LES: in std\_logic; -- Interface con el procesador

PET: in std\_logic;

LISTO: out std\_logic;

AFP: in std\_logic; -- Interface con modulo campos

AFO: in std\_logic; -- Acierto/Fallo observador

PED: out std\_logic; -- y elementos perifericos

PEET: out std\_logic;

PEV: out std\_logic;

valinP: in std\_logic;

valinO: in std\_logic;

valout: out std\_logic;

muxL: out std\_logic;

muxE: out std\_logic;

muxO: out std\_logic;

muxM: out std\_logic; -- Direccion que se envia a memoria

MLES: out std\_logic; -- Interface con memoria

MPET: out std\_logic;

MLISTO: in std\_logic;

BMLES: in std\_logic;

BMPET: in std\_logic;

MOD: out std\_logic; -- Escritura entre caches

CONC: in std\_logic; -- Concesion bus

FIN: out std\_logic; -- Finalizar transaccion

PETBUS: out std\_logic;

VEST: out std\_logic\_vector (3 downto 0);

VESTO: out std\_logic\_vector (1 downto 0)

); -- Observacion externa del estado

end;

architecture comportamiento of controlador is

constant UNO : std\_logic := '1'; -- Constantes para establecer

constant CERO : std\_logic := '0'; -- y comprobar valores

constant lectura : std\_logic := '1';

constant escritura : std\_logic := '0';

constant acierto : std\_logic := '1';

constant fallo : std\_logic := '0';

constant PETICION : std\_logic := '1';

constant NOPETICION : std\_logic := '0';

-- observacion externa del estado

constant VESTLIST: std\_logic\_vector :="1000"; -- Estado LISTO

constant VESTREEM: std\_logic\_vector :="0100"; -- Estado REEM

constant VESTLECT: std\_logic\_vector :="0010"; -- Estado LEC

constant VESTESCR: std\_logic\_vector :="0001"; -- Estado ESC

constant VESTLR: std\_logic\_vector :="0110"; -- Estado LR

constant VESTLL: std\_logic\_vector :="1010"; -- Estado LL

type tipoestado is (LIST, LR, REEM, LL, LECT, ESCR); -- Estados

signal estado, prxestado: tipoestado; -- Registro de estado

-- senales temporales en el proceso de salida, sin sincronizar con la senal de reloj,

-- correspondientes a las PED, PEV y PEET

signal TPED: std\_logic :='1';

signal PEVP: std\_logic :='1';

signal TPEET: std\_logic :='1';

signal LISP: std\_logic:='1';

signal vaP: std\_logic:='1';

-- OBSERVADOR

-- Observacion externa del estado del Obs.

constant VESTDESOB: std\_logic\_vector :="10"; -- Estado DESO

constant VESTESPOB: std\_logic\_vector :="01"; -- Estado ESPO

type tipoestadoO is (DESO, ESPO); -- Estados del automata del observador

signal estado\_O, prxestado\_O: tipoestadoO; -- Registro de estado del observador

-- senales temporales en el proceso de salida del observador, sin sincronizar con la senal de reloj,

-- correspondiente a PEVO

signal PEVO: std\_logic :='1';

-- observador: senal para establecer el estado (listo y val)

signal LISO: std\_logic :='1';

signal vaO: std\_logic :='1';

-- multiplexores. Seleccion entre la salida del procesador y la salida del observador

signal muxva: std\_logic; -- seleccion de la senal val

signal muxEV: std\_logic; -- seleccion de la senal PEV

signal muxLI: std\_logic; -- selec. de la senal LISTO del proc. u obs.

signal TPEV: std\_logic; -- senal de salida del multiplexor muxEV

-- deteccion de flanco ascendente

function flanco\_ascendente (signal reloj: std\_logic) return boolean is

variable flanco: boolean:= FALSE;

begin

flanco := (reloj = '1' and reloj'event);

return (flanco);

end flanco\_ascendente;

begin

-- registro de estado

process (reloj, pcero)

begin

if pcero = UNO then

VEST <= VESTLIST after retardo;

elsif (reloj = UNO and reloj'event) then

estado <= prxestado after retardo;

case prxestado is

when LIST => VEST <= VESTLIST after retardo;

when REEM => VEST <= VESTREEM after retardo;

when LECT => VEST <= VESTLECT after retardo;

when ESCR => VEST <= VESTESCR after retardo;

when LR => VEST <= VESTLL after retardo;

when LL => VEST <= VESTLR after retardo;

end case;

end if;

end process;

-- logica de proximo estado

process(estado, LES, PET, AFP, MLISTO, valinP, CONC, pcero)

begin

prxestado <= estado after retardo;

if pcero = CERO then

case estado is

when LIST =>

if PET = PETICION and AFP = fallo then

if val = UNO then

prxestado <= LR after retardo;

else

prxestado <= LL after retardo;

end if;

end if;

when LR =>

if CONC = UNO then

if valinP = UNO then

prxestado <= REEM after retardo;

else

prxestado <= LECT after retardo;

end if;

end if;

when REEM =>

if MLISTO = UNO then

prxestado <= LECT after retardo;

end if;

when LL =>

if CONC = UNO then

prxestado = LECT after retardo;

end if;

when LECT =>

if MLISTO = UNO then

if LES = escritura then

prxestado <= ESCR after retardo;

else

prxestado <= LIST after retardo;

end if;

end if;

when ESCR =>

prxestado <= LIST after retardo;

end case;

else

prxestado <= LIST after retardo;

end if;

end process;

-- logica de salida

process(estado, LES, PET, AFP, MLISTO, valinP, CONC, pcero)

begin

LISP <= UNO after retardo;

MPET <= CERO after retardo;

TPED <= (PET and AFP and not LES) after retardo;

TPEET <= CERO after retardo;

PEVP <= CERO after retardo;

muxL <= CERO after retardo;

muxE <= CERO after retardo;

MLES <= LES after retardo;

vaP <= CERO after retardo;

FIN <= CERO after retardo;

muxM <= CERO after retardo;

PETBUS <= CERO after retardo;

if (pcero = CERO) then

case estado is

when LIST =>

if PET = PETICION and AFP = fallo then

LISP <= CERO after retardo;

PETBUS <= UNO after retardo;

end if;

when LR =>

LISP <= CERO after retardo;

TPED <= CERO after retardo;

if CONC = UNO then

MPET <= UNO after retardo;

MLES <= not valinP after retardo;

muxM <= valinP after retardo;

end if;

when REEM =>

LISP <= CERO after retardo;

MPET <= MLISTO after retardo;

MLES <= MLISTO after retardo;

TPED <= CERO after retardo;

when LL =>

LISP <= CERO after retardo;

TPED <= CERO after retardo;

if CONC = UNO then

MPET <= UNO after retardo;

MLES <= UNO after retardo;

end if;

when LECT =>

if MLISTO = UNO then

LISP <= LES after retardo;

TPEET <= UNO after retardo;

TPED <= UNO after retardo;

PEVP <= UNO after retardo;

vaP <= UNO after retardo;

muxL <= UNO after retardo;

muxE <= UNO after retardo;

FIN <= LES after retardo;

else

LISP <= CERO after retardo;

TPED <= CERO after retardo;

end if;

when ESCR =>

TPED <= UNO after retardo;

FIN <= UNO after retardo;

end case;

end if;

end process;

-- OBSERVADOR

-- registro de estado del observador

est\_obs: process (reloj, pcero)

begin

if (pcero = UNO) then

estado\_O <= DESO after retardo;

VESTO <= VESTDESOB after retardo;

elsif (flanco\_ascendente(reloj)) then

estado\_O <= prxestado\_O after retardo;

case prxestado\_O is

when DESO => VESTO <= VESTDESOB after retardo;

when ESPO => VESTO <= VESTESPOB after retardo;

end case;

end if;

end process est\_obs;

-- logica de proximo estado del observador

prox\_obs: process(estado\_O, BMPET, AFO, MLISTO)

begin

prxestado\_O <= DESO after retardo;

case estado\_O is

when DESO =>

if (BMPET = PETICION and AFO = acierto) then

prxestado\_O <= ESPO after retardo;

end if;

when ESPO =>

if MLISTO = CERO then

prxestado\_O <= ESPO after retardo;

end if;

end case;

end process prox\_obs;

-- logica de salida del observador

sal\_proc: process(estado\_O, BMPET, AFO, MLISTO)

begin

PEVO <= CERO after retardo;

vaO <= CERO after retardo;

LISO <= UNO after retardo;

MOD <= CERO after retardo;

case estado\_O is

when DESO =>

if (BMPET = PETICION and AFO = acierto) then

LISO <= CERO after retardo;

MOD <= UNO after retardo;

end if;

when ESPO =>

if MLISTO = UNO then

PEVO <= UNO after retardo;

else

LISO <= CERO after retardo

MOD <= UNO after retardo;

end if;

when others =>

LISO <= CERO after retardo;

end case;

end process sal\_proc;

-- seleccion de las senales del procesador o del observador

muxva <= not CONC; -- modelado ideal

muxEV <= not CONC; -- la cache atiendo al procesador u observa

muxLI <= not CONC; -- se utiliza la senal PET como de concesion

LISTO <= LISP after retardo when muxLI = CERO else -- multiplexor muxLI

LISO after retardo;

valout <= vaP after retardo when muxva = CERO else -- multiplexor muxva

vaO after retardo;

TPEV <= PEVP after retardo when muxEV = CERO else -- multiplexor muxEV

PEVO after retardo;

-- actualizacion de cache

PED <= not (TPED and (not reloj)) after retardo;

PEET <= not (TPEET and (not reloj)) after retardo;

PEV <= not (TPEV and (not reloj)) or pcero after retardo;

end;