Instituto Tecnológico de Costa Rica

Área Académica de Ingeniería en Computadores

Curso: CE-4302 Arquitectura de Computadores II

Ing. Luis Barboza Artavia

Proyecto I. Protocolos de coherencia en sistemas multiprocesador

Estudiante:
Francisco J. Murillo Morgan, 2015147156

I Semestre 2020

Listado de requerimientos del sistema

El problema presentado, consiste en la realización de un sistema multiprocesador, el que estará compuesto por dos chips, donde cada chip estará compuesto por:

- Dos núcleos de procesamiento.
- Dos memorias cache L1, cada una asociada a uno de los núcleos de procesamiento.
- Controlador de caché para coherencia de memorias L1.
- Una memoria cache L2

Por último cada uno de los chips contará con una memoria principal compartida.

Núcleos de procesamiento

Para cada uno de los núcleos de procesamiento, se necesita que estos sean capaces de generar 3 tipos de instrucciones distintas:

- Read
- Write
- Calc

Estos deben ser capaces de utilizar una distribución probabilística para la generación de una dirección de memoria a la que se debe aplicar la instrucción.
Los núcleos deben ser capaces de comunicarse con su respectiva memoria cache L1, para el almacenamiento de datos.
Cada núcleo debe ser capaz de ejecutarse independientemente del flujo principal de los demás elementos del sistema, con la finalidad de tener más de un núcleo ejecutándose al mismo tiempo.

Memorias Caché L1

Para las memorias caché L1, estas deben ser capaces de almacenar el estado de coherencia, dirección de memoria, y dato de la dirección de memoria que se encuentra referenciada en la caché.

Este tipo de cache debe permitir lecturas y escrituras del núcleo al que pertenece, así como de su controlador respectivo.

La caché L1 debe ser capaz de cambiar su estado de coherencia dependiendo de la acción que desee realizar su núcleo respectivo.
Por último la caché L1 debe ser capaz de referenciar como máximo dos bloques de memoria principal.

Controlador de memoria cache

Para el controlador de memoria caché L1, este debe ser capaz de leer los bloques de caché de ambas memorias L1 de los núcleos de procesamiento.

El controlador debe ser capaz de cambiar el estado de cada uno de los bloques en L1, para así determinar si estos se encuentran en estado M, S, o I para aplicar el protocolo de coherencia de MSI.

EL controlador debe ser capaz de leer cada bloque en la memoria caché L2, y ser capaz de cargar datos desde memoria principal a la caché L2, así cómo cambiar el estado del bloque, y los procesadores que son dueños de cada bloque, con el fin de poder aplicar el protocolo basado en directorios.

Por último el controlador debe ser capaz de ejecutar el protocolo de monitoreo a implementar, con el fin de mantener la coherencia en todas las caches en cada uno de los chips.

Memoria Caché L2

Para la memoria caché L2, esta debe ser capaz de almacenar directorios, donde se podrá tener un control de cada uno de los bloques de memoria que se encuentran referenciados en L1.

La memoria caché debe ser capaz de almacenar el estado de coherencia del directorio, la dirección en memoria principal a la que se hace referencia, los procesadores que son dueños de cada directorio, así cómo el dato.

La memoria caché L2 debe ser capaz de ser leída y escrita por el controlador de memoria caché.

Memoria Principal

Para la memoria principal de datos, esta debe ser capaz de almacenar su propio estado de coherencia, así cómo saber los chips que son dueños de cada uno de sus bloques, y por último, almacenar un valor para que este sea leído o sobrescrito por los núcleos de procesamiento.

Elaboración de opciones de solución al problema

Propuesta de solución #1

Para realizar la solución al problema planteado, se propone utilizar Python cómo el lenguaje de programación para realizar la implementación. La propuesta de este lenguaje se basa a la naturaleza del problema. Por su naturaleza, se puede realizar una solución más comprensible y estructurada si se utiliza algún lenguaje de programación que permita utilizar el paradigma de orientado a objetos. Además de esto, se necesita que dicho lenguaje posea una implementación sencilla pero eficaz de multiprocesamiento de instrucciones utilizando hilos. Por el dominio que se tiene sobre este lenguaje de programación se sabe que es posible utilizar el paradigma orientado a objetos y además aplicar hilos de instrucciones conforme se vea necesario.

Una vez definido el lenguaje para realizar la implementación, se propone el diagrama de la figura 1, cómo el diagrama de bloques del sistema a implementar.

Cómo se observa en la figura 1, esta solución contiene todos los bloques descritos en la sección anterior, pero incorpora un nuevo controlador llamado Controlador De Memoria General. Este controlador se encargará de leer directamente las cache L2 de cada uno de los chips. Con la información contenida en estas cache, luego este controlador, utilizando un protocolo de monitoreo de datos, se encargará de actualizar todas las caché L1 de los chips, para qué de esta forma los estados de cada bloque en caché se encuentren actualizados y con datos coherentes dentro de cada una de las memorias cache.

Por último esta implementación utilizará una política de reemplazo aleatoria para los bloques de cache en L1, así cómo utilizará el protocolo de escritura write-through con el fin de garantizar que siempre que se realice una lectura de un dato que no se encuentra en caché, todos los procesadore se encuentren leyendo el mismo dato.

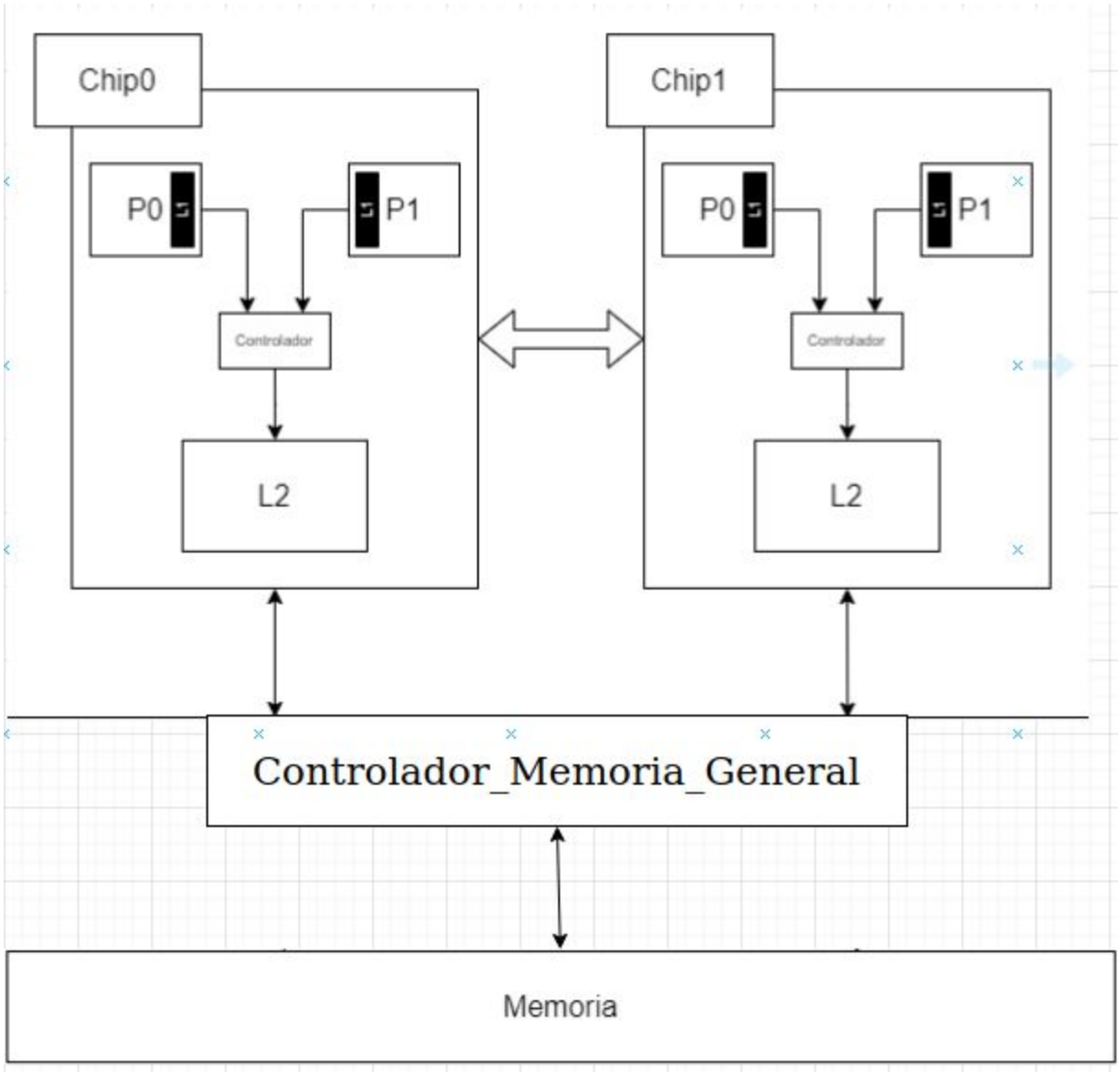

Figura 1.Diagrama de bloques para la propuesta #1

Propuesta de solución #2

Para realizar la solución al problema planteado, se propone utilizar Python cómo el lenguaje de programación para realizar la implementación. La propuesta de este lenguaje se basa a la naturaleza del problema. Por su naturaleza, se puede realizar una solución más comprensible y estructurada si se utiliza algún lenguaje de programación que permita utilizar el paradigma de orientado a objetos. Además de esto, se necesita que dicho lenguaje posea una implementación sencilla pero eficaz de multiprocesamiento de instrucciones utilizando hilos. Por el dominio que se tiene sobre

este lenguaje de programación se sabe que es posible utilizar el paradigma orientado a objetos y además aplicar hilos de instrucciones conforme se vea necesario.

Una vez definido el lenguaje para realizar la implementación, se propone el diagrama de la figura 2, cómo el diagrama de bloques del sistema a implementar.

Para esta segunda solución, se propone sustituir los dos controladores de memoria caché L1 qué tendría cada núcleo, por un controlador de caché unificado. Este controlador de caché al tener acceso completo a las cache L1 de todos los núcleos, facilitará conocer los estados de coherencia de cada bloque que se encuentre cargado en caché. De esta manera también puede realizar la actualización de la memoria caché L2 de cada chip simultáneamente, ya que no necesita hacer una petición al bus para que este le retorne el estado individual de cada bloque de caché L1, ya que al tener una conexión directa con estos, el controlador conocerá todos los datos necesarios para actualizar el directorio en L2.

Por último, esta solución utilizará write-back cómo política de escritura de memoria. Se escogió esta política tomando en cuenta que al realizar modificaciones a los datos, el controlador de caché L1 se encargará de actualizar directamente los bloques afectados ya que este contiene una conexión directa con todos los bloques de cache L1.

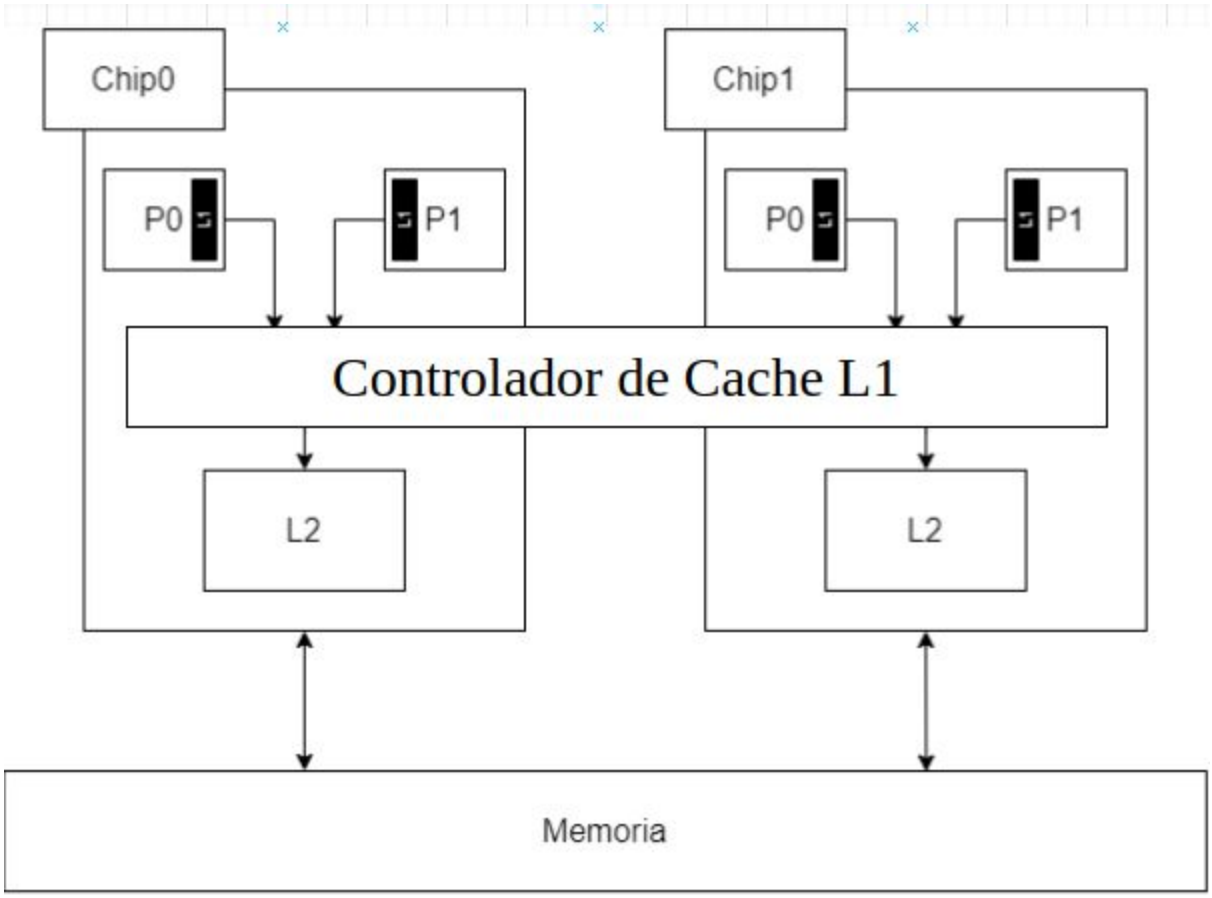

Figura 2.Diagrama de bloques para la propuesta #2

Comparación de las soluciones propuestas

Se puede observar un diagrama de bloques para ambas soluciones propuestas en las figuras 1 y 2 respectivamente. La mayor diferencia se encuentra en la implementación de los controladores de caché y la política de escritura a utilizar. Si realizamos una comparación entre ambas propuestas, la propuesta #2 tendrá menor latencia de comunicación a la hora de realizar la lectura de datos desde memoria. Esto debido a que gracias a su protocolo de monitoreo que utiliza las 4 instancias de caché L1 existentes, en caso de realizar una lectura de un bloque que se encuentre modificado en otra caché, fácilmente se puede leer dicho valor modificado y utilizarlo cómo el valor más reciente en el resto de cachés que lo necesiten. En cambio para la solución 1, el controlador de memoria general, al no tener comunicación directa con las caché L1, se le dificulta conocer qué valor es el más actualizado, por lo que dependerá del write-through cuando algún procesador decida modificar el valor de memoria.

En términos de utilización del bus, la propuesta 2 utilizará más el bus de comunicación entre chips, que la propuesta #1. Dependiendo de la cantidad de veces que el controlador de caché L1 deba utilizar el bus de comunicación afectará el rendimiento del sistema multiprocesador. La implementación de la propuesta #1 carece de este problema, ya que el bus solo se utilizará para realizar la invalidación de los bloques de las otras cachés.

Selección de la propuesta final

Tomando en cuenta los criterios de comparación vistos en la sección anterior, se decide realizar la implementación de la propuesta #2. Esto se debe a que, si bien es cierto qué qué existirá un problema de latencia dependiendo de la cantidad de uso del bus de comunicación, este retraso seguirá siendo menor al generado al tener que leer constantemente de la memoria principal cómo ocurre en la propuesta #1.

Implementación del diseño

La implementación de la propuesta #2, se puede observar en el diagrama de clases que se encuentra en la figura 3. A continuación se dará una descripción de cada una de las clases que componen el sistema, en el que se detallan todos los métodos y atributos de las clases generadas.

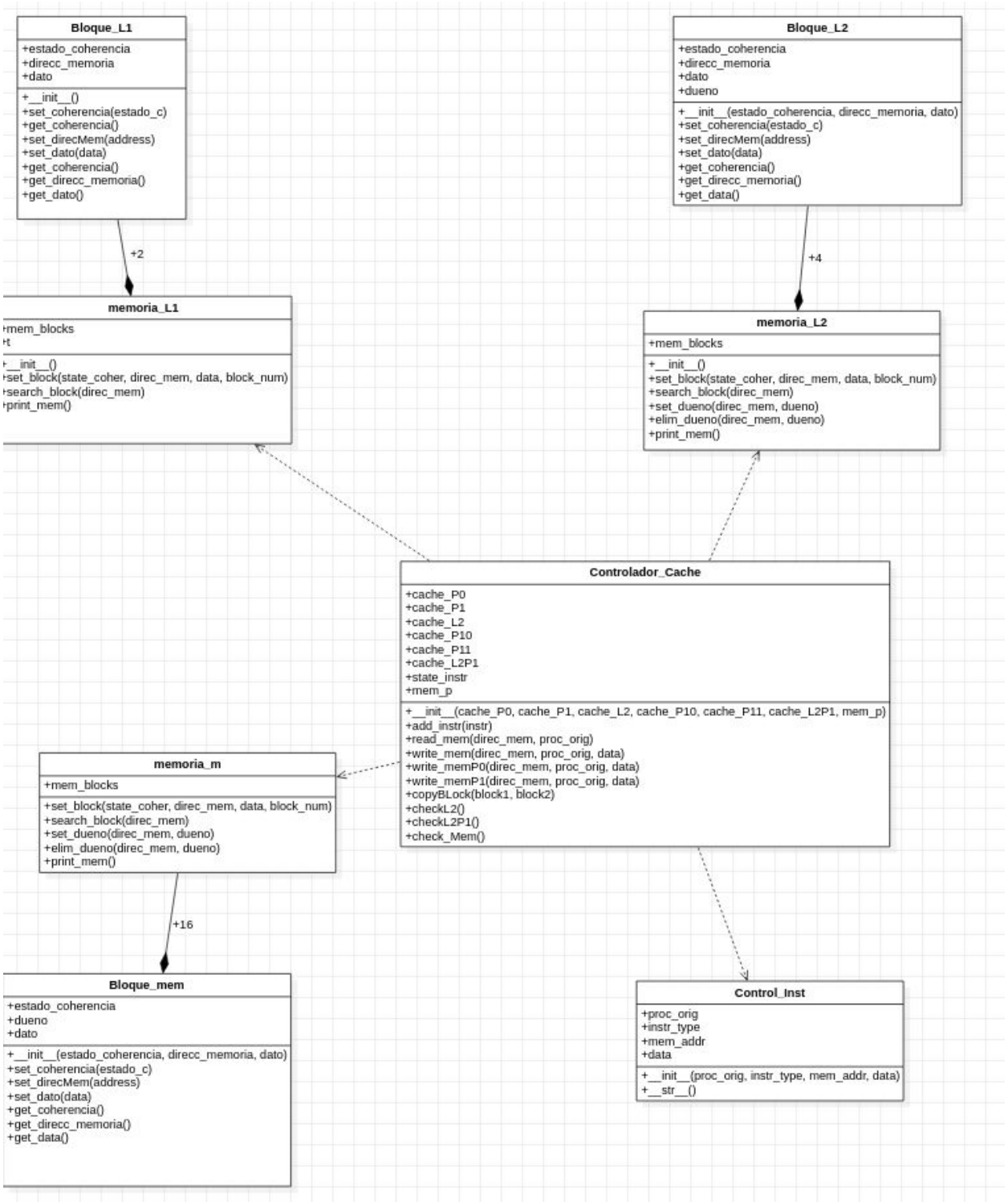

Figura 3. Diagrama de clases de la implementación.

Bloque L1

Esta clase representará un bloque de memoria dentro de la cache L1.

Atributos:

- **estado_coherencia** : Atributo encargado de guardar el estado de coherencia del bloque, ya sea M, S, o I.
- **direcc_memoria**: Atributo encargado de guardar la dirección de memoria principal a la que el bloque está haciendo referencia.
- **dato**: Atributo encargado de guardar el dato que se esta leyendo o escribiendo al bloque de memoria principal. En caso de que se modifique el bloque en cache, este atributo almacenará el valor más reciente del bloque.

Metodos:

- **__init__**(estado_coherencia, direcc_memoria, dato): Método encargado de inicializar la instancia de la clase, asignando a cada uno de los atributos el valor especificado en sus parámetros.
- **set_coherencia**(estado_c): Método encargado de cambiar el estado de coherencia del bloque al estado especificado.
- **set_direcMem**(address) : Método encargado de cambiar la direccion de referencia a memoria del bloque a la dirección especificada.
- **set_dato(data)**: Método encargado de cambiar el dato de referencia a memoria del bloque en caché al valor especificado.
- **get_coherencia()**: Método encargado de retornar el estado actual de coherencia del bloque.
- **get_direcc_memoria()**: Método encargado de retornar la dirección de memoria a la que se hace referencia en el bloque.
- **get_dato()**: Método encargado de retornar el valor almacenado de la dirección de memoria a la que se hace referencia en el bloque.

Memoria_L1:

Esta clase se encarga de realizar la abstracción de una caché L1. Esta clase está compuesta por dos instancias de la clase Bloque_L1.

Atributos:

- **Mem_blocks:** Este atributo será una lista de todos los Bloques_L1 que contendrá la memoria L1.

Métodos:

- **_init__():** Método encargado de inicializar la instancia de la clase.

- **set_block(state_coher, direc_mem, data, block_num):** Método encargado de seleccionar uno de los dos bloques de cache, y actualizar el valor de sus atributos por el valor que se definió en el parámetro.
- **search_block(direc_mem):** Método encargado de buscar en los bloques de caché al bloque que se encuentre haciendo referencia a la dirección de memoria brindada.
- **print_mem():** Método utilizado para imprimir en consola toda la información pertinente de todos los bloques que se encuentran en dicha instancia de la memoria caché.

Bloque L2

Esta clase representará un bloque de memoria dentro de la caché L2.

Atributos:

- **estado_coherencia** : Atributo encargado de guardar el estado de coherencia del bloque, ya sea M, S, o I.
- **direcc_memoria**: Atributo encargado de guardar la dirección de memoria principal a la que el bloque está haciendo referencia.
- **dato**: Atributo encargado de guardar el dato que se está leyendo o escribiendo al bloque de memoria principal. En caso de que se modifique el bloque en caché, este atributo almacenará el valor más reciente del bloque.
- **duenno:** Lista con el identificador de todos los procesadores que son dueños del dato.

Metodos:

- **__init__**(estado_coherencia, direcc_memoria, dato): Método encargado de inicializar la instancia de la clase, asignando a cada uno de los atributos el valor especificado en sus parámetros.
- **set_coherencia**(estado_c): Método encargado de cambiar el estado de coherencia del bloque al estado especificado.
- **set_direcMem**(address) : Método encargado de cambiar la direccion de referencia a memoria del bloque a la dirección especificada.
- **set_dato(data)**: Método encargado de cambiar el dato de referencia a memoria del bloque en caché al valor especificado.
- **get_coherencia()**: Método encargado de retornar el estado actual de coherencia del bloque.
- **get_direcc_memoria()**: Método encargado de retornar la dirección de memoria a la que se hace referencia en el bloque.

- **get_dato()**: Método encargado de retornar el valor almaceneado de la dirección de memoria a la que se hace referencia en el bloque.

Memoria_L2:

Esta clase se encarga de realizar la abstracción de una caché L2. Esta clase está compuesta por cuatro instancias de la clase Bloque_L2.

Atributos:

- **Mem_blocks:** Este atributo será una lista de todos los Bloques_L2 que contendrá la memoria L2.

Métodos:

- **_init__():** Método encargado de inicializar la instancia de la clase.
- **set_block(state_coher, direc_mem, data, block_num):** Método encargado de seleccionar uno de los dos bloques de cache, y actualizar el valor de sus atributos por el valor que se definió en el parámetro.
- **search_block(direc_mem):** Método encargado de buscar en los bloques de caché al bloque que se encuentre haciendo referencia a la dirección de memoria brindada.
- **set_dueno(direc_mem, dueno):** Agrega el identificador especificado a la lista de dueños, del bloque que hace referencia a la direeción de memoria especificada.
- **elim_dueno(direc_mem, dueno):** Elimina el identificador especificado de la lista de dueños, del bloque que hace referencia a la direeción de memoria especificada.
- **print_mem():** Método utilizado para imprimir en consola toda la información pertinente de todos los bloques que se encuentran en dicha instancia de la memoria caché.

Bloque mem

Esta clase representará un bloque de memoria dentro de la memoria principal

Atributos:

- **estado_coherencia** : Atributo encargado de guardar el estado de coherencia del bloque, ya sea M, S, o I.
- **dato**: Atributo encargado de guardar el dato que se está leyendo o escribiendo al bloque de memoria principal. En caso de que se modifique el bloque en caché, este atributo almacenará el valor más reciente del bloque.

- **duenno:** Lista con el identificador de todos los procesadores que son dueños del dato.

Metodos:

- **__init__**(estado_coherencia, direcc_memoria, dato): Método encargado de inicializar la instancia de la clase, asignando a cada uno de los atributos el valor especificado en sus parámetros.
- **set_coherencia**(estado_c): Método encargado de cambiar el estado de coherencia del bloque al estado especificado.
- **set_direcMem**(address) : Método encargado de cambiar la direccion de referencia a memoria del bloque a la dirección especificada.
- **set_dato(data)**: Método encargado de cambiar el dato de referencia a memoria del bloque en caché al valor especificado.
- **get_coherencia()**: Método encargado de retornar el estado actual de coherencia del bloque.
- **get_direcc_memoria()**: Método encargado de retornar la dirección de memoria a la que se hace referencia en el bloque.
- **get_dato()**: Método encargado de retornar el valor almaceneado de la dirección de memoria a la que se hace referencia en el bloque.

Memoria_m:

Esta clase se encarga de realizar la abstracción de la memoria principal. Esta clase está compuesta por dieciséis instancias de la clase Bloque_mem.

Atributos:

- **Mem_blocks:** Este atributo será una lista de todos los Bloques_mem que contendrá la memoria principal.

Métodos:

- **_init__():** Método encargado de inicializar la instancia de la clase.
- **set_block(state_coher, direc_mem, data, block_num):** Método encargado de seleccionar uno de los dos bloques de cache, y actualizar el valor de sus atributos por el valor que se definió en el parámetro.
- **search_block(direc_mem):** Método encargado de buscar en los bloques de caché al bloque que se encuentre haciendo referencia a la dirección de memoria brindada.
- **set_dueno(direc_mem, dueno):** Agrega el identificador especificado a la lista de dueños, del bloque que hace referencia a la dirección de memoria especificada.

- **elim_dueno(direc_mem, dueno):** Elimina el identificador especificado de la lista de dueños, del bloque que hace referencia a la direeción de memoria especificada.
- **print_mem():** Método utilizado para imprimir en consola toda la información pertinente de todos los bloques que se encuentran en dicha instancia de la memoria caché.

Control Instr:

Esta clase es la encargada de representar las instrucciones generadas por cada uno de los núcleos de procesamiento. Cada instancia de esta clase incluirá el procesador que generó la instrucción, la acción a realizar, la dirección de memoria a la cual realizar la acción, y en caso de ser necesario, el dato que se debe escribir en la dirección de memoria seleccionada. Para la generación de instrucciones automáticamente, se utilizó la distribución de probabilidad de Poisson. De esta manera nos aseguramos que las instrucciones no son instrucciones aleatorias, si no que existirá una relación entre ellas y de esta manera se simula de mejor manera el comportamiento real de un sistema multiprocesador.

Atributos:

- **proc_orig** : Atributo encargado de identificar el procesador del cual proviene la instrucción.
- **instr_type**: Atributo encargado de identificar el tipo de instrucción que se debe ejecutar.
- **mem_addr:** Identificador de la dirección de memoria sobre la cual se ejecuta la instrucción.
- **data:** En caso de ser una instrucción de tipo “write” este es el valor que se debe escribir en memoria.

Métodos:

- **__init__(proc_orig, instr_type, mem_addr, data):** Método encargado de inicializar la instancia de la clase asignando a cada uno de sus atributos el valor especificado en los parámetros.
- **def __str__():** Método que retornará un String que nos dará la información de la instrucción almacenada en esta instancia.

Controlador Caché:

Esta clase es la encargada de aplicar el protocolo de monitoreo diseñado para la implementación de este simulador. Cada vez que alguno de los núcleos genera una instrucción de lectura, esta clase es la encargada de leer la instrucción e identificar la direccion de memoria que se desea leer. Al haber identificado dicha direccion, se encarga de revisar en ambas cache L2 y buscar que bloques de caché L1 en cada nucleo hace alguna referencia a la direccion de memoria deseada. En caso que algun bloque de cache L1 tenga una copia del dato, el controlador procede a realizar la lectura del valor de dicho dato directamente de la caché en donde se encontró la copia. Una vez hecho esto, se procede a marcar la direccion de memoria en las caches como un bloque en estado Shared (S). Para el caso en que el controlador no encuentre una referencia a la direccion de memoria, dentro de alguna de las caché L2, el controlador procede a leer el dato directamente de la memoria principal, y procedea escribir dato en el bloque de cache seleccionado, poniendo este en un estado de Modified (M). En el caso que no exista un bloque de memoria L1 disponible para almacenar referenciar el bloque de memoria, el controlador procede a seleccionar un bloque a reemplazar con este nuevo dato, realizar el proceso de escritura en memoria principal (proceso que será descrito más adelante) y a reemplazar el bloque seleccionado con el bloque que se desea referenciar.

Para el caso de una instrucción de escritura en memoria, el controlador primero verificara en ambas caché L2, si alguno de las caches L1 de los núcleos posee una referencia a dicha dirección de memoria. De ser el caso, el controlador procede a poner los bloques compartidos en estado invalido, de esta manera el único bloque que contendrá el dato mas nuevo será el que se acaba de escribir en alguna de las caché L1. Luego de hacer la escritura y la invalidación de los bloques compartidos, el controlador procederá a actualizar los bloques activos y compartidos para las memorias caché L2 en cada uno de los chips.

Atributos:

- **cache_P0** : Instancia de la caché L1 del núcleo 0 del chip 0.
- **cache_P1** : Instancia de la caché L1 del núcleo 0 del chip 1.
- **cache_L2** : Instancia de la caché L2 del chip 0.
- **cache_P10** : Instancia de la caché L1 del núcleo 1 del chip 0.
- **cache_P11** : Instancia de la caché L1 del núcleo 1 del chip 1.
- **cache_L2P1** : Instancia de la caché L2 del chip 1.

Métodos:

- **__init__(cache_P0, cache_P1, cache_L2, cache_P10, cache_P11, cache_L2P1, mem_p):** Método encargado de inicializar la instancia de la clase asignando a cada uno de sus atributos el valor especificado en los parámetros.
- **add_instr(instr):** Añade una instrucción al bus de isntrucciones pendientes que requieran de lectura o escritura de los bloques que no se encuentran ya en caché.
- **read_mem(direc_mem, proc_orig):** Método del protocolo de monitoreo, el cual lee desde cualquier otra caché L1, o desde memoria el dato en la dirección de memoria especificada. Al hacer la lectura toma en cuenta las transiciones entre estados de los otros bloques de caché.
- **write_mem(direc_mem, proc_orig, data):** Método auxiliar que determina si llamar al método write_memP0 en caso de ser una escritura a alguna de las caches L1 del chip 0 o write_memP1 en caso de ser una escritura a alguna de las caches L1 del chip 1.
- **write_memP0(direc_mem, proc_orig, data):** Método encargado de realizar la escritura a algún bloque de memoria caché L1 en el chip 0, invalidando el resto de bloques de caché que compartieran dato con dicho bloque.
- **write_memP1(direc_mem, proc_orig, data):** Método encargado de realizar la escritura a algún bloque de memoria caché L1 en el chip 1, invalidando el resto de bloques de caché que compartieran dato con dicho bloque.
- **copyBLock(block1, block2):** Método encargado de copiar todo el bloque 1 en el bloque 2 especificados en los parámetros.
- **checkL2():** Método encargado de actualizar los valores de la caché L2 en el chip 0 para qué los directorios tengan coherencia con los datos, estado, y procesadores que son dueños de estos.
- **checkL2P1():** Método encargado de actualizar los valores de la caché L2 en el chip 1 para qué los directorios tengan coherencia con los datos, estado, y procesadores que son dueños de estos.
- **check_Mem():** Método encargado de actualizar el estado de cada uno de los bloques de memoria principal, así cómo el identificador de los chips que son dueños de cada bloque.