AOC2 Proyecto 2 - Jerarquía de memoria de datos

AOC2 Team, EINA, Univ. Zaragoza

Fecha de entrega: 15 mayo

ADVERTENCIA

- Leer este guión atentamente te ahorrará tiempo.
  - Además te ayudará a consolidar o incluso entender aspectos que no aún no tienes claros sobre lo explicado en teoría o problemas.
  - También te servirá de referencia de estilo a la hora de escribir y organizar un informe.
- Comunica cualquier fallo que encuentres y pregunta lo que necesites, pero las cuestiones que estén explícitamente contestadas en este guión las atenderemos con prioridad nula.
- No cambies los nombres de los componentes y señales que se facilitan en los fuentes de partida sin consultarlo con los profesores.
- Todos los materiales del proyecto, como todos los del curso (fuentes, apuntes, problemas, guiones etc) están sujetos a los derechos de propiedad intelectual de la Universidad de Zaragoza. Su divulgación sin permiso o con fines lucrativos tiene consecuencias legales.

Contenidos

1 Resumen

En este proyecto introducimos una jerarquía de memoria algo más compleja que en el proyecto anterior. Sus componentes se comunican entre sí a través de un **bus semi-síncrono**. Esta jerarquía incluye una memoria cache (`MC`), una memoria de datos (`MD`) más lenta que la que teníamos en el proyecto anterior, y una segunda memoria de datos (`MD Scratch`) mucho más rápida que `MD`, cuyos contenidos no se deben guardar en `MC`. El objetivo es diseñar el controlador de la memoria cache que atiende las peticiones del procesador, gestionando las transferencias necesarias a través del bus. Además, debéis detener el procesador (durante las etapas necesarias) cuando la etapa MEM no pueda realizar la operación solicitada en un ciclo de reloj (como sucedía hasta ahora). `MC` incluye contadores para evaluar los factores que pueden limitar el rendimiento de la jerarquía de memoria. **Lee detenidamente este guión incluyendo sus anexos**: ahorrarás tiempo. El procedimiento de presentación y defensa será el mismo que en el primer proyecto. **Lee detenidamente las instrucciones de envío** cuando activemos el recurso correspondiente en Moodle.

2 Sistema a diseñar

En el sistema base que proporcionamos la memoria de datos del MIPS se substituye por un sistema de memoria más complejo. Este sistema incluye la `MC`, la `MD`, la `MD Scratch`, el bus semisíncrono, y un periférico de entrada salida (`IO_Master`) que monitoriza una entrada del sistema y la escribe una y otra vez en la primera palabra de la `MD Scratch`. `IO_Master` actúa como maestro del bus, al igual que el controlador de la `MC`, por ello pueden existir conflictos y es necesario incluir un árbitro que los evite. Para que el reparto del bus sea equitativo el árbitro utiliza una asignación dinámica de prioridades. El interfaz con el MIPS es idéntico al de la memoria del primer proyecto, pero ahora vamos a usar dos señales que antes ignorábamos:

- `Mem_Ready` es la señal que nos avisa de que la etapa MEM va a realizarse en el ciclo actual. Cuando `MC` no pueda realizar la operación solicitada en el ciclo actual bajará la señal `Mem_Ready`, y el procesador deberá detener su ejecución hasta que `Mem_Ready` vuelva a activarse (únicamente responderá a las excepciones, al igual que pasaba cuando se activaba Parar_ID).

- `IO_input` es un puerto de 32 bits de entrada del procesador. En nuestro MIPS es un puerto *mapeado* (localizado) en memoria, i.e. el procesador lo podrá leer mediante un `lw` de una dirección determinada, en este caso la primera palabra de la `MD Scratch`. En otras palabras, utilizamos una dirección de la `MD Scratch` como espacio de E/S del MIPS; podríamos implementar más registros de entrada usando más direcciones de `MD Scratch` para E/S mapeadas. Para vincular (*mapear*)

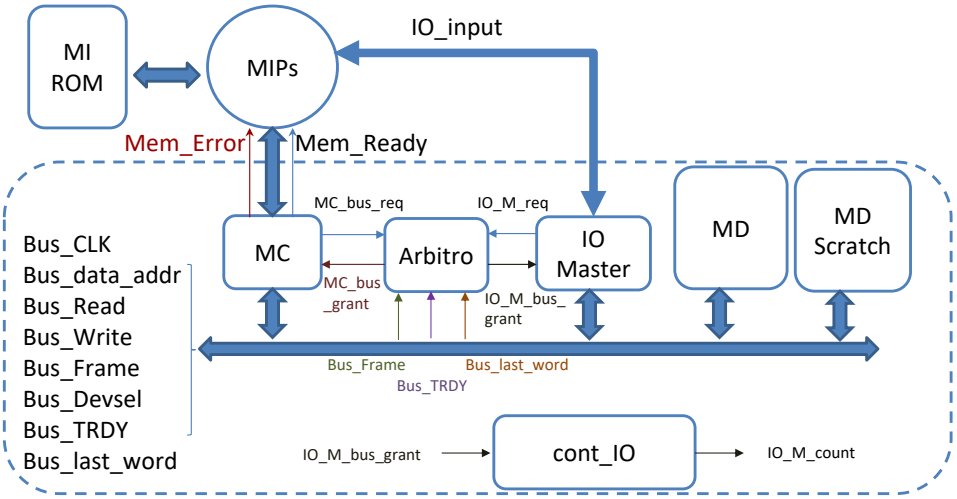

Figure 1: Esquema y señales del bus en el sistema a diseñar.

`IO_input` a la primera palabra de `MD Scratch` utilizamos un periférico que lo lee continuamente (i.e. lo *encuesta*), escribiendo el valor leído en la primera palabra de la `MD Scratch`, que el MIPS puede leer mediante un `lw`. `IO_input` podría usarse para enviar información o una orden. La encuesta por parte del periférico no es un diseño eficiente (pensad una forma mejor, alternativa a la encuesta), pero os va a permitir aprender a gestionar los conflictos del bus interaccionando con el árbitro.

El esquema con las señales del bus se puede ver en la Fig. 1. Describimos los componentes a continuación.

2.1 MIPS

Vamos a usar el mismo procesador que en el proyecto anterior, sustituyendo la memoria de datos que teníamos (eliminadla del proyecto) por un nuevo subsistema con varios módulos y un bus (incluid los nuevos fuentes). Además, aparece un nuevo riesgo estructural que hay que gestionar. Si el MIPS ejecuta la etapa MEM de un `lw` o `sw` y el controlador desactiva `Mem_Ready`, el MIPS tendrá que detener la ejecución de la etapa MEM y anteriores. Podríamos pensar que para en la etapa MEM no afecta a la etapa de WB, sin embargo hay un caso en el que sí que afecta, debido a cómo funciona nuestra unidad de anticipación. Si tenemos una instrucción que provoca una parada en memoria, justo en mitad de dos intrucciones entre las que había anticipación, el dato a anticipar se perderá. **Podéis ver un cronograma explicativo en la wiki del proyecto**. La solución que os proponemos es parar el procesador entero cuando paramos en MEM incluyendo la etapa WB. Escribir varias veces el mismo dato no consume energía, así que no hay ninguna penalización real por repetir la etapa de escritura varios ciclo.

Además hay que incluir `Mem_Ready` a la hora de contar las paradas por riesgos de datos o control. Por ejemplo, Si la UD identifica un load-uso, y justo hay un fallo en `MD` que dura 20 ciclos, el contador no debe contar esos 20 ciclos como parada load-uso, porque el load-uso no ha sido la causa de esa detención. Por tanto, cuando paramos en MEM, no debéis incrementar los contadores de paradas por riesgos de datos o control.

2.2 Bus de memoria

Se trata de un bus semi-síncrono basado libremente en el estándar PCI en el que `MC` e `IO_Master` actúan como maestros y `MD` y `MD Scratch` siempre como servidores. Las **líneas de datos y direcciones son multiplexadas**. El bus soporta ráfagas de tamaño variable. Las transferencias tienen tres fases:

- **Arbitraje** - Antes de comenzar una transferencia hay que solicitar el uso del bus a un árbitro utilizando la señal `Bus_req`. El árbitro contestará activando `Bus_grant`, pero a veces habrá que esperar porque como hay dos maestros puede ser que el bus esté ocupado.
- **Dirección - En el mismo ciclo en que se reciba la señal `Bus_grant` comenzará la transferencia**. Para ello el maestro enviará la dirección (líneas `Bus_addr_data`) y el tipo de operación (`Bus_Rd_Wr` = '0' para lecturas e = '1' para escrituras). **El servidor que identifique que la dirección es suya responderá activando `Bus_DevSel` en ese mismo ciclo**.
  - **Importante:** Si en el ciclo en el que se envía la dirección ningún servidor activa `devsel`, la dirección solicitada se almacenará en un registro (`ADDR_Error_Reg`), y se activará el bit de error `MEM_Error`. La señal `MEM_Error` generará un `Data_abort` en el MIPS. El registro `ADDR_Error_Reg` es accesible desde el MIPS, dirección `x"01000000"`, cuando el procesador lo lea el bit de error se desactivará.
- **Datos - Al ciclo siguiente de que el servidor active la señal `Bus_DevSel` comenzará la fase de datos**. En esta fase, el maestro activará `Bus_frame`, y el servidor avisará si está preparado para realizar la operación que le ha pedido el maestro en el ciclo actual activando la señal `Bus_TRDY` (*target ready*) y recibirá o enviará un dato por ciclo a través de `Bus_Data_Addr` hasta que el maestro desactive la señal `Bus_Frame`. En ese instante el esclavo retirará el dato del bus así que no se debe bajar `Bus_Frame` antes de tiempo). Si en un ciclo dado el esclavo no es capaz de enviar el dato que corresponde, **desactivará la señal `Bus_TRDY` para indicar al maestro que debe esperar**. Los datos se envían /reciben por el bus siempre de forma consecutiva. Por ejemplo, si el maestro pide la palabra 4, el servidor enviará la 4, la 8, la 12 etc. **Cuando el maestro envíe la última palabra activará la señal `last_word`**. Si sólo quiere una palabra, `last_word` se activará desde el principio.

2.3 `MC`

La memoria cache es asociativa con dos vías (s=2 i.e. dos bloques por *set*). Cada vía tiene 4 bloques de 4 palabras. Funciona con **escritura directa (*write-through*)**, la política de fallo en escritura es **fetch on write miss**, y la política de reemplazo es **FIFO**.

2.4 Scratch MD

Es una memoria de datos pequeña y rápida para disponer de datos accesibles con baja latencia (como una cache) pero gestionada exclusivamente mediante los `lw` y `sw` del MIPS: no hay fallos, no se copian bloques desde `MD` ni se actualizan escrituras en `MD`[1]. Conforma un espacio propio de direcciones que denominamos *no cacheable*: la `MC` no almacena los contenidos de la `MD Scratch`. En su lugar si el procesador quiere leer o escribir una palabra, el controlador de `MC` gestionará la lectura o escritura de una palabra en esta memoria a través del bus **pero no se traerá ningún bloque**. Su rango es: `X"10000000"-X"100000FF"`. No soporta modo ráfaga. Es decir, hay que pedir las palabras de una a una.

2.5 Controlador `MC`

Recibe las solicitudes del MIPS y si puede las responde con los datos almacenados en `MC`. En caso contrario realiza las transferencias necesarias con `MD` o la `MD Scratch` a través del bus.

Junto a los materiales del proyecto en Moodle encontrarás un esquema a hoja completa de la organización interna de la `MC`. Os servirá de referencia para diseñar el controlador. **El controlador es una máquina de estados** bastante sencilla. La dificultad consiste en entender bien los protocolos y sus señales.

[1] Las primeras GPGPUs carecían de memoria cache pero incorporaban una scratch de este tipo. En NVIDIA/CUDA se denomina *shared memory* y en OpenCL *scratchpad*). A partir del procesador Fermi de NVIDIA se incorporó un nivel que puede configurarse parte como cache, parte como *scratchpad*, a voluntad del programador. Las *scratchpads* son también habituales en los SoCs de sistemas emprotrados. Nuestra `MD Scratch` está inspirada en esta idea.

El controlador tiene una segunda máquina de estados con sólo dos estados: `Memory_error` y `No_error`, que también debéis gestionar, bien en el mismo proceso que la máquina anterior, porque están relacionadas, bien en otro aparte. Esta máquina se ocupa de generar la señal `Mem_Error` que el MIPS interpretará como `Data_abort`. Cuando detecta que el MIPS pide un dato mal alineado, o el controlador intenta comunicarse por el bus como maestro, y ningún servidor responde, el controlador pasará de estado `No_error` a estado `Memory_error`. En ese estado seguirá funcionando con normalidad, pero con su señal `Mem_Error` activa. Además almacenará la dirección de la palabra que ha generado el error (no almacena el byte) en el registro `Addr_Error_Reg`. El controlador se mantendrá en el estado de error hasta que el MIPS lea el registro `Addr_Error_Reg` que está mapeado en memoria en la dirección `x"01000000"`. Cuando el MIPS lea el registro, el controlador volverá al estado `No_error`.

2.6 Controlador de la MD

Siempre actúa como servidor, trasladando las solicitudes del bus a `MD` si están dentro de su rango (`X"00000000"-X"000001FF"`). Las direcciones fuera de su rango se ignoran.

Temporización: La `MD` que vamos a utilizar es la misma que teníamos en el primer proyecto, pero con retardos adicionales, que son mayores en la primera palabra que en las restantes. Cuando no pueda enviar un dato en el ciclo actual desactivará la señal `Bus_TRDY` para que el controlador sepa que debe esperar.

Modo ráfaga: cuando `Bus_Frame` esté activado la memoria de datos realizará la operación solicitada con la dirección inicial, y después la incrementará internamente.

2.7 Nuevo contador en MIPS

Como hay un nuevo riesgo deberéis **añadir un contador** en el MIPS para saber cuántos ciclos de parada introduce este riesgo. **La salida de ese contador debe llamarse** `paradas_mem`. Importante: no contéis dos veces la misma parada. Si en el mismo ciclo hay varias razones para parar, activad sólo uno de los contadores (pensad cuál sería el correcto).

2.8 Contadores de eventos de MC

La `MC` incluye **tres contadores**[2] :

1. **cont_m** contabiliza los fallos en `MC`. Un fallo se define como un acceso (lw o sw) **a una dirección cacheable** que no está en `MC`. Los accesos a `MD Scratch` o al registro interno de `MC` no se consideran fallos.

2. **cont_w** contabiliza las escrituras realizadas. Debe contar cada vez que el MIPS ordene escribir un dato, es decir una vez por cada store que se ejecute.

3. **cont_Mem_stall** contabiliza los ciclos de retardo que genera nuestro subsistema de memoria. Contamos cada ciclo en el que Mem_ready sea '0'.

`m_count (la cuenta de cont_m)` y `w_count` (la cuenta de `cont_w`) sólo se deben incrementar una vez por cada fallo y escritura. Un fallo de escritura que genere diez ciclos de retardo deberá incrementar una vez `m_count` y `w_count`, y diez veces `Mem_stalls`.

2.9 IO_Master

El sistema incluye un periférico de entrada salida (`IO_Master`) que monitoriza la entrada del sistema `IO_input` y la escribe una y otra vez en la primera palabra de la `MD Scratch`. Este dispositivo actúa como maestro del bus. El bus incluye un contador para ver cuantos ciclos de bus utiliza este dispositivo. Cuanto más eficiente sea nuestra gestión de las transferencias, más ciclos le dejaremos el bus libre al `IO_Master`.

[2]Los controladores reales contienen a menudo más contadores que sirven para evaluar el rendimiento y adaptar los programas a esa jerarquía.

3 Resultados y entregables

3.1 Programas de prueba

En primer lugar, **debéis comprobar que el diseño funciona correctamente para todos los casos posibles** (fallos y aciertos de lectura y escritura, utilizando los cuatro conjuntos y provocando algún reemplazo), y que los contadores de paradas cuentan cuando les corresponde. Para ello debéis definir esos casos en un banco de pruebas (*testbench*). Os proporcionamos ejemplos de *testbench* en los que se prueban algunos casos, pero debéis añadir los vuestros, **describiendo vuestro banco de pruebas en la memoria** y explicando los casos que cubre.

Os recomendamos simular primero el componente `MD_mas_MC` antes de integrar todo en el MIPS. Para ello os proporcionamos un banco de pruebas con bastantes casos preparados (`testbench_MD_mas_MC`). Estas pruebas no hace falta entregarlas, ni comentarlas. Basta con entregar las pruebas realizadas en el MIPS.

3.2 Memoria

En la memoria debéis explicar también brevemente vuestro diseño. Recurrid preferiblemente a fragmentos de código /pseudocódigo y esquemas. Hay que incluir obligatoriamente los elementos que se detallan a continuación.

1. **Diagrama de estados de la unidad de control**, explicando qué se hace en cada estado y qué señales se activan.
2. La **descomposición de la dirección** para el direccionamiento de la `MC` como hemos enseñado en clase (*byte/word – set – tag*)
3. Una **análisis de las latencias de las distintas transferencias en el bus** sin tener en cuenta los retardos debidos al arbitraje. Obtened los siguientes retardos a partir de la simulación:
   - `CrB(MD)`: ciclos para leer un bloque de `MD`, dónde `CrB(MD)` = L ciclos (para la primera palabra) + 3*R ciclos (para las palabras restantes).
   - `CwW(MD)`: ciclos para escribir una palabra en `MD`.
   - `CrW(MDscratch)`: ciclos para leer una palabra de `MD Scratch`.
   - `CwW(MDscratch)`: ciclos para escribir una palabra en `MD Scratch`.
   - `CrW(MDscratch)`: ciclos para leer el registro interno de la `MC`.
4. **Expresión de cálculo de los ciclos efectivos** (ciclos medios por acceso a memoria) a partir de los costes anteriores y los eventos que pueden ocurrir a vuestra `MC`: acierto de lectura o escritura, fallo de lectura o escritura de dirección de `MD`, o de `MD Scratch`, o acceso al registro interno de `MC`. Dado que el arbitraje en el sistema base que os proporcionamos no tarda siempre lo mismo, considerad 1,5 ciclos como valor medio.
5. **Programas de prueba** presentados como en el primer proyecto. Identificad las distintas situaciones de vuestra máquina de estados y **presentar en una tabla dónde y cómo se ha comprobado cada caso**.
6. **Ejemplo de código** en el que se saque partido a esta jerarquía. Calculad el ***speedup*** que aportan la `MC` y la `MD Scratch` con respecto a un sistema similar sólo con la `MD` **de este proyecto, ¡no con la `MD` ideal del proyecto anterior!**. Puede ser uno de vuestros programas de prueba, y se puede hacer de forma teórica, o simulándolo.
7. Cuantificación de horas dedicadas por cada miembro del grupo a cada apartado del proyecto con la **suma total de horas**.
8. Una **autoevaluación** en la que debéis contestar de forma individual a dos preguntas:
   - ¿Crees que has cumplido los objetivos de la asignatura?
   - ¿Qué nota te pondrías si te tuvieses que calificar a ti mismo?

4 Criterios de Evaluación

Recordad que una vez entregados los proyectos a través de Moodle, los comprobamos uno por uno.

Como en el proyecto anterior, **el diseño debe funcionar correctamente para aprobar**. Si detectamos errores y consideramos que son menores, os daremos 24 horas para subsanar.

El diagrama de estados de la memoria debe de ser claro para poder seguirlo. Podéis ampliar el comportamiento de cada estado en el texto, identificando con claridad el número de estado. **En vuestros tests debéis especificar si cada `lw` o `sw` va a ser acierto o fallo, y si se almacena en `MC` indicad el conjunto y la vía**. Si este análisis previo es incorrecto, no pasaremos a comprobar y evaluar el diseño. De igual manera es motivo de suspenso no entender por qué se asigna un valor determinado a una señal en vuestra unidad de control. La respuesta no puede ser del tipo: *porque si pongo '0' en lugar de '1' no nos funciona*. Este aviso no pretende intimidar: el 99% lo vais a saber sin problemas, pero queremos dejar claro que presentar algo sin demostrar que se entiende no sirve para nada.

Valoración de vuestro controlador: El principal factor al evaluarlo será que el **diagrama de estados sea claro y esté bien explicado, y que vuestra descripción y fórmula de los ciclos efectivos coincida con la realidad**. De forma menos crítica, valoraremos que el controlador sea eficiente y genere el mínimo número de ciclos de parada. Os recomendamos partir de un esquema sencillo que os resulte fácil entender y después, si os da tiempo, tratar de mejorar su rendimiento (ver apartado de opciones a continuación).

5 Apartados opcionales del segundo proyecto

- Incluir un ***buffer*** **para las escrituras en** `MD`. Cuando haya que escribir en `MD` se almacena el dato a escribir y su dirección en dos registros y se indica al procesador que puede continuar activando `Mem_Ready`. La `MC` podrá continuar gestionando aciertos mientras la Unidad de Control gestiona la escritura a través del bus. Por supuesto si hay que hacer una segunda transferencia en el bus y no se ha terminado de gestionar la anterior habrá que parar. Explicar casos en que mejore el rendimiento.

- **Avanzado**: Adelantar el envío de la palabra crítica, un mecanismo que referimos a menudo en los vídeos de los temas 5-10. Cuando el controlador gestione un fallo de lectura lo más sencillo es traer el bloque entero y después enviar a la CPU la palabra que ha solicitado. Sin embargo, se puede mejorar el rendimiento entregando al procesador la palabra solicitada tan pronto como llegue. De este modo la CPU puede continuar mientras el controlador termina de traer el resto de palabras del bloque. **Explicar casos en que mejore el rendimiento**.