Organización de Computadoras - TP 3

Martín Cohen, Padron: 100812 martincohen98@gmail.com

1er Cuatrimestre 2020

Resumen

En este trabajo practico se implementaron instrucciones para una CPU de arquitectura MIPS, ya sea modificando configuraciones o el datapath en si, con el objetivo de familiarizarse con la arquitectura.

1 Introducción

El trabajo practico tiene como objetivo familiarizarse con la arquitectura MIPS. Para esto se utilizo el programa DrMIPS[1], en el que se realizaron modificaciones y simulaciones varias.

El trabajo consistió en implementar varias instrucciones para dos tipos de CPU distintos. Una de los CPU es uniciclo, en el que las instrucciones se ejecutan una a la vez. La otra CPU es una implementación con pipeline en la las instrucciones tienen varias etapas. Dividir las instrucciones en etapas hace que se pueda ejecutar mas de una instrucción a la vez, ya que distintas etapas de las instrucciones utilizan distintas partes de la CPU.

Se pidio que se implementen las distintas instrucciones:

- Un andi, en el que se compara un registro con un valor inmediato. Esta instrucción se implemento para las CPU uniciclo y con pipeline.
- Un jump en el que se salta a un valor en el que se suman los valores de dos registros y luego se mueve ese valor al PC (multiplicado por 4 para que caiga en el inicio de una instrucción). Tambien implementado para ambas CPU.
- Un load implementando direccionamiento escalado. Esta solo se pidio ser implementada para la arquitectura con piepline.

Para implementar estas instrucciones fue necesario modificar los sets de instrucciones y hasta los datapaths de las CPU. Estos se hizo modificando unos archivos con formato JSON en el que se detalla toda la implementacion de los CPU.

Los sets de instrucciones están en archivos con terminación .set, mientras que los CPU están en los archivos de terminación .cpu. Los archivos CPU especifican el archivo de instrucciones que utilizan. Para realizar el trabajo se utilizaron como base los archivos default del programa para ambos los CPU y sets de instrucciones.

Para el testeo de las instrucciones creadas se programaron funciones simples de Assembly MIPS en las que se utilizan las instrucciones comprobando que funcionen correctamente.

[1] https://brunonova.github.io/drmips/

2 Desarrollo de las instrucciones

Para ver los datapaths finales utilizados se puede abrir el programa DrMIPS con los archivos encontrados en el repositorio adjunto.

2.1 And inmediato

La implementación del and inmediato fue idéntica para ambas CPU, pero llevo a comprender el funcionamiento del programa y de la CPU que emula.

Para la implementación de esta instrucción solo fue necesario modificar el archivo .set, ya que no fue necesario agregar ningún componente extra ni conectar cables extra. Sin embargo,no fue suficiente solo crear la instrucción en la lista de instrucciones disponibles. Fue necesario modificar la configuración de la unidad de control y del controlador de la ALU.

En primer lugar, se tuvo que utilizar un nuevo opcode para poder levantar un set de flags diferente a los que es posible en el archivo .set default. Esto es debido a que, al ser una instrucción de tipo I (con un inmediato), no se tiene el campo func que tienen las instrucciones de tipo R. Esto significa que se tiene que explicitar la operación que debe realizar la ALU solo con el opcode.

En el opcode implementado se especifico a la ALU que debía utilizar el inmediato y que realiza la operación and sin necesidad de utilizar el campo func de las instrucciones R. Esto fue posible sin modificar el cableado ya que con los dos bits que se utilizan para comunicar la operación al controlador de la ALU en el campo ALUOp uno de los valores estaba inutilizado.

Como el cableado no fue modificado, realizar el cambio en los archivos .set que utilizan ambas CPU fue suficiente para hacerlos funcionar.

2.2 Jump con suma

Para esta instrucción las implementaciones para las distintas CPU fueron diferentes.

2.2.1 CPU uniciclo

Para esta implementación fue necesario que el set de instrucciones no tuviera la instrucción j, ya que esta iba a llevar el mismo nombre. Por esta razón, fue posible eliminar la parte del circuito relacionada con el jump que ya existía en el default (o utilizar la versión de la CPU default sin jump). En mi caso utilice la versión default eliminando todo menos el multiplexor final que lleva a la entrada del PC.

Luego de esto agregue una salida extra al fork en la salida de la ALU y la conecte a un Shift Left de dos bits para alinear el resultado de la adición a las instrucciones. La salida del shift fue conectada al multiplexor preexistente.

Fue necesario también cambiar los flags utilizados para el opcode del jump, ya que ahora se le debe indicar a la ALU la necesidad de hacer la suma. El tipo de instrucción cambio de ser de tipo J, ya que las instrucciones de tipo

J no permiten registros como parámetros. Por esta razón la transforme a una operación tipo R.

2.2.2 CPU con pipeline

Esta implementación fue la más compleja, ya que implico agregar varios componentes y cables.

El primer paso fue preparar la instrucción, esto significa modificar el archivo .set de la misma manera que se modifico el uniciclo. Sin embargo, para el caso del pipeline, la CPU por defecto no incluía la posibilidad de realizar un jump. Teniendo en cuenta esto, el resultado es un archivo .set casi idéntico al de la implementación uniciclo.

La gran diferencia fue cuando fue necesario modificar el datapath. En principio fue necesario agregar una salida extra en la unidad de control. Esta salida es el flag de jump que no se encontraba en el default. Este flag lo hice pasar por los registros de pipeline ID/EX y EX/MEM antes de tener alguna función, ya que antes de realizar el jump los datos tienen que pasar por la ALU para realizar la suma.

La gran cantidad de modificaciones aparece en la etapa de memoria, donde ya se tiene el resultado de la ALU. La principal motivación que me llevo a mi implementación fue intentar utilizar la mayor parte del cableado de los branch, ya que estos y el jump tienen la similitud de que modifican el PC y requieren flushear los registros de pipeline.

En esta etapa cree un fork para el flag nuevo flag jump para utilizarlo en dos otros nuevos componentes.

Una de las salidas fue hacia un multiplexor en el que se utilizo como selector. Las entradas de este multiplexor son la salida de la ALU, viniendo de una salida extra de un fork preexistente pasada por un shift left de dos bits, y la dirección target para las instrucciones branch saliendo del registro EX/MEM. La salida yendo directamente al multiplexor que modifica el PC. Anteriormente esta entrada del multiplexor venia directamente del target de los branch, ya que era lo único que podía modificar al PC. El nuevo multiplexo tiene como función llevar el resultado de la suma a ese multiplexor que modifica el PC en vez del target del branch cuando se enciende el flag de jump de la unidad de control.

La otra salida del fork del flag jump fue hacia un or lógico. La otra entrada del or lógico es la salida del and lógico que define si se realiza un branch. Esto significa que si hay un branch tomado o un jump la salida del or va a estar encendida. Ambas de esas condiciones significan que se modifica el PC. Afortunadamente el default ya incluye un cable que lleva a un camino que flushea todos los registros de pipeline anteriores y termina en el selector que modifica el PC. Este es el camino que se usaba cuando había un branch tomado. Conectando la salida del or logico a este camino hace que cuando haya un jump o un branch tomado se flusheen los registros de pipeline y se modifique el PC. Con esto y la modificación anterior, se logra que cuando hay un jump se flusheen todos los registros de pipeline anteriores y se modifique el PC con el resultado de la suma de la ALU.

De esta manera no fue necesario modificar el cableado de la hazard detection unit y asegurar de que no so genere ninguna inconsistencia con los registros.

2.2.3 Load con direccionamiento escalado

Para esta instrucción, fue necesario utilizar otra CPU default, ya que se necesitaba una ALU extendida debido a que para realizar parte de la operación se necesita multiplicar.

Para esta instrucción no fue posible implementarla como una instrucción nativa. Esto se debe principalmente a que se necesita utilizar dos veces la ALU, y al ser esto necesario no se puede implementar como una instrucción normal. Otra de las barreras fue que el tipo de dato que utiliza DrMIPS para la carga y guardado de datos no puede ser separado entre registro e inmediato, entonces realizar operaciones con solo uno de los componentes no fue posible. Por estas razones esta fue implementada como pseudoinstrucción.

Al ser este el caso la implementación fue más simple de lo esperado. Por el hecho de no poder separar el inmediato del registro para las instrucciones que interactúan con la memoria, el inmediato y el registro se toman como parámetros separados.

Teniendo eso en cuenta, lo primero que se hace es cargar el inmediato a un registro para poder realizar la multiplicación, ya que las instrucciones de multiplicación solo toman registros. Luego se realiza la multiplicación y se toma el valor que queda en el registro LO. Finalmente se realiza la suma final y la carga del dato de memoria.

Al ser una pseudoinstrucción y solo utilizar instrucciones preexistentes no hay posibilidad de generar hazards.

3 Casos de Prueba

Para probar todas las instrucciones se programo una función simple en Assembly que prueba la funcionalidad.

Para el and con inmediato se realizan una serie de ands lógicos para comprobar que solo los bits que coinciden quedan en el resultado.

En el caso del jump con suma se hace un jump con una instrucción que modifica un registro luego de la instrucción de jump y uno en la instrucción antes de la que se salta. Finalmente en la instrucción a la que se salta se modifica otro registro para probar que se salte exitosamente. El objetivo es que solo se modifique el registro al que cae el salto y no los otros. Al ocurrir esto significa que el jump funciona correctamente.

Para probar el load simplemente se carga un dato directamente a una posición de memoria y luego se busca utilizando el nuevo load con todos sus parámetros distintos de 1 en el caso de la multiplicación, y a 0 en el caso del registro que suma. Si se carga correctamente el mismo dato entonces el load funciona correctamente.

4 Conclusiones

Para poder realizar este trabajo exitosamente fue sumamente necesario la comprensión del funcionamiento de las distintas CPU de arquitectura MIPS. De no haber sido este el caso la implementación de las instrucciones habría sido imposible. Durante el trabajo fue necesario agregar, remover y modificar casi todas las partes del CPU. Que no solo me llevo a amaestrar el funcionamiento del simulador DrMIPS, sino que también entender como se modela una CPU real para una computadora real de arquitectura MIPS.

Fue necesario también comprender en profundidad los componentes que conforman la CPU, ya que hubo que ingeniar cuales eran necesarios para lograr formar las instrucciones pedidas. Al tener que buscar que componente seria el más adecuado para realizar lo que necesito que la CPU haga me llevo a un entendimiento de los componentes mucho mayor de lo que esperaba inicialmente.

Es necesario aclarar que el trabajo en su totalidad fue una tarea desafiante, ya que realizar una modificación al diseño de un procesador es una tarea distinta a lo que acostumbra en el área de la informática. Eso me llevo a tener que utilizar una forma distinta para encarar el problema necesitando utilizar herramientas nuevas. Sin embargo, eso es lo que hizo que fuera una experiencia más enriquecedora.