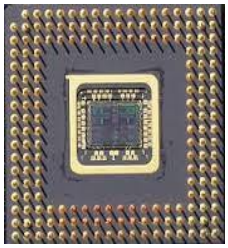
Proyecto de Arquitectura de Computadoras

Curso 2021

S-MIPS



Procesador S-MIPS
Informe

Estudiante: Jordan Plá González
Grupo: C211
Costo: 26

Instruction Cache:

En este proyecto está implementado una memoria caché *direct mapped* con $8 = 2^3$ líneas, por lo que de la *instruction address* (**PC**) empleando solo 16 bits para *address* (después de los 4 bits menos significativos) siendo consecuentes con la memoria RAM implementada, se necesitan 3 bits para indexar en las líneas (*index*) y el resto de bits (13) se emplean para *tag*. Para simular el *offset* en las líneas se emplea 4 *banks* mediante componentes RAM (*Logisim*) de igual modo que se hizo en la RAM ya implementada, para esto se define un *mask* de 4 bits que indica en cual o cuales bancos se desea escribir, usando los bits 2 y 3 de **PC** como selector de un decodificador para establecer dicha *mask*. En la caché se utilizan 8 D flip-flops para simular los bits *valid* que se establece por línea en caché, indicando si estas líneas presentan datos con información válida o no.

Puede darse la situación que se mande a escribir en la RAM algún dato que ocupe el lugar de una instrucción ya almacenada en caché, y para evitar inconsistencias tales como escribir en la RAM sin hacerlo en caché, se aplica política de escritura *write through*, o sea, se actualiza la caché y se escribe en la RAM. Para ello, se introduce la dirección donde se va a escribir, el dato que se va a escribir en la RAM y un bit *check* que es el factor imprescindible ya que es el que le informa si se va a escribir en la RAM o no.

Se comprueba que dicha dirección ya esté en caché (mediante la comparación del *tag* en el *index* que presente la misma) y que, además, la línea donde se va a almacenar dicho dato contenga información válida, de no ser el caso, no hay necesidad de actualizar la caché porque no ha recibido la línea donde se está escribiendo de momento.

Se establece un *control unit* en caché que proporciona el orden de cómo se va a efectuar la lectura de caché (caso de hits) o esta última leer de la RAM (caso de miss). Cuando se lee de la memoria RAM hay que recurrir al tiempo de espera de lectura en la misma, para esto último se emplea un *data path* que establece el decrecimiento del *read time* hasta cero, con la activación o no del *chip select* en el momento oportuno.

Hit: La línea tiene información válida, y el *tag* de la dirección ya está en caché, por tanto, los datos ya están almacenados y no hay que leer de la RAM.
Miss: La línea no tiene información válida, o tiene información válida pero el *tag* de la dirección entrante no es igual al de la línea donde se va a escribir. En estos casos hay que leer de la RAM aprovechando la localidad espacial y luego poder leer de la caché sin problema alguno esperando que ocurran más hits aumentado la eficiencia de esta caché.

Read RAM:

Como bien su nombre indica, este componente está creado específicamente para leer de la RAM algún dato de una dirección dada, en caso de instrucciones tales como *pop* o *lw*. Para su implementación es necesario saber la dirección donde está almacenado el dato en la RAM para establecer el *address* y *mask* que se le pasará a la RAM y ofrezca el dato que interesa. Como la arquitectura es *little endian* cada 1 byte, transformo dicho dato para trabajar con él de manera más cómoda en *big endian*.
Como mismo se hacía en ***Instruction Cache***, al estar leyendo de la RAM, se necesita el *read time* y concluir el tiempo de espera para recibir dicho dato, por lo que de igual modo se realiza un *control unit* y *data path* que proporcionan el orden para leer de memoria y el decremento del *read time* a cero en los distintos ciclos de reloj, obviamente proporcionando el chip select que recibirá la RAM.

Write RAM:

Como bien su nombre indica, este componente está creado específicamente para escribir en la RAM algún dato de una dirección dada, en caso de instrucciones tales como *push* o *sw*. Para su implementación es necesario saber la dirección del cual se podrá establecer el *address* y *mask* que establecerán la línea y el banco respectivamente, donde se va a escribir en la memoria RAM. Como la arquitectura es *little endian* y trabajo con *big endian* para mejor comprensión y claridad, cada 1 byte transformo dicho dato para almacenarlo con la arquitectura correcta y ser consecuente con lo ya hecho. Esto es análogo al ***Read RAM***, pero en este caso es escritura, se necesita el *write time* y concluir el tiempo de espera para almacenar en memoria dicho dato, realizando una vez más un *control unit* y *data path* que proporciona el orden para escribir en memoria y el decremento del *write time* a cero en los distintos ciclos de reloj, obviamente proporcionando el chip select que recibirá la RAM, pero aquí como factor más influyente, ya que le informa a la RAM que se está escribiendo y no leyendo de ella.

Control Unit:

El control unit es el encargado de activar o desactivar las señales correspondientes para que todo mantenga un orden. Establece el *stop* del programa en caso de haberse leído la instrucción *halt,* proporciona un *enable random* en caso de la instrucción *rnd* y demás detalles en el estilo, como activación de *enable read* o *enable write* cuando se vaya a escribir o leer de memoria RAM respectivamente por las instrucciones ya mencionadas con anterioridad, así como la activación del *tty* o el *keyboard* por sus correspondientes instrucciones. Lo mas importante aquí, sin duda alguna, el establecimiento de un patrón, siempre que se haya leído la instrucción o se haya escrito en la memoria pedir la siguiente instrucción. Mi análisis fue trabajar con la instrucción que tengo almacenada, una vez leída en el ***Instruction Finder*** (posteriormente será comentado) pedir la siguiente instrucción a ***Instruction Cache*** para trabajar de manera rápida y efectiva. En fin, es el encargado de mandar a pedir la siguiente instrucción siempre que se cumpla las condiciones pertinentes.

Data Path:

Es quizás la que tenga mayor carga en cuanto a implementación, por lo que para su explicación lo desglosaré y explicaré cada uno de sus subcomponentes.

- Entradas: 4 entradas de 32 bits que representan las salidas de las instrucciones en ***Instruction Cache***, un bit *End* que indica si la caché ya tiene la próxima instrucción almacenada, una entrada *clock* y otra *reset/clear*

  NOTA: estas entradas son de suma importancia para el ***Instruction Finder*** mencionado anteriormente.

  ### *Instruction Finder:*

  Este componente se encarga de tomar una de las 4 instrucciones que le ofrece la caché de instrucciones, seleccionándolo según la máscara que almacena el ***Program Counter*** (***PC***) ya que la dirección fue ofrecida con anterioridad a la caché para buscar las instrucciones de la línea deseada. Como la arquitectura es *little endian* cada 1 byte, transformo dicho dato para trabajar con él de manera más cómoda en *big endian*. Una vez almacenada la instrucción devuelve un bit *done* indicando que ya tiene la instrucción lista y puede pedir la siguiente. Esta instrucción es enviada para el ***Instruction Decoder*** (***ID***)

  ### *Instruction Decoder:*

  Este componente recibe una instrucción en binario y se encarga de compactarlo mediante *Op-Code*, *Func-Code*, *Selector de A*, *Selector de B*, y de activar la señal de la instrucción que representa ya que será enviado al ***Control Unit***. También se encarga junto al ***Control Unit*** de indicar cuando se lee o se escribe en el ***Register File*** y mediante la señal activa según la instrucción que representa indica un *Operador* de 4 bits que recibe el ***ALU*** para efectuar la operación aritmética o lógica correspondiente a dicha señal. Dependiendo de la instrucción que sea se ofrece una constante que indicará un salto en el ***Branch Control*** o una constante como segunda entrada en el ***ALU***. Otro detalle que desarrolla este componente, es la salida del bit *unSigned* que entra en el ***ALU***: si la operación a realizar es sin signo está activado, o con signo está desactivado.

  ### *Branch Control:*

  Este componente recibe la dirección saliente del ***PC*** y un bit de entrada *next* que determine el incremento de esta dirección para establecer la próxima dirección de la siguiente instrucción que se desea hallar, dándole esta nueva dirección a ***PC*** como su entrada. Recibe las señales de salto y los flags salientes del ***ALU*** que determinan si se cumple o no una condición para efectuar un salto u otro. También recibe una entrada *Jump Size* que es el tamaño del salto a efectuar, que mayormente es la constante saliente del ***ID***. Para estas condiciones de salto se aplica la operación resta en el ***ALU***, por eso es mencionado anteriormente que los flags indican si se cumple la condición o no:

  i. beq -> [A] == [B] -> ZF
  ii. bne -> [A] <> [B] -> !ZF
  iii. blez -> [A] <= 0 -> NF || ZF
  iv. bgtz -> [A] > 0 -> !NF && !ZF
  v. bltz -> [A] < 0 -> NF

Register File:

Este componente se encarga de almacenar 32 registros para uso general, con particularidades tales como que *R0* siempre tiene almacenado el valor cero, y *R31* almacena el puntero de la pila. Para disminuir su costo, fue creado un componente *multiplexer* de 5x32 con la ayuda de *buffers controlados*. Su uso es sencillo de comprender, presenta un bit de lectura o escritura que dependiendo de si está activo o no se escribe en el registro *Rx* según el *Destination Address*, el valor entrante *Data Input*. Ofrece dos salidas según los distintos selectores que se le proporcione.

Arithmetic-Logic Unit:

Recibe dos valores de 32 bits y un selector de operador que proporciona el ***ID*** según la instrucción que esté activa. Es el componente encargado de realizar el cálculo ya sea aritmético o lógico, según el selector de operador, el bit *unSigned* del cual se habló anteriormente es el factor condicionante, según la instrucción que esté activa determina si será con signo o sin signo la operación. Presenta los registros *Hi* y *Lo* resultados de aplicar alguna multiplicación o división, y como salida ofrece los mencionados *Hi* y *Lo,* el resultado de la operación entre los dos valores de la entrada, y los flags: NF y ZF, según si el resultado fue negativo o cero respectivamente.

NOTA: se tuvo que realizar las implementaciones del *mulu* y el *div.*

- En el caso del ***div*** el procedimiento realizado fue: como es con signo, en caso de tener el bit 31 en 1 se considera negativo, por lo que se halla el complemento a dos del número para convertirlo a positivo y se efectúa la división normal. Si ambos tienen el mismo signo, el *Lo* es un resultado positivo, sino se le halla complemento a dos al resultado de la división. En el caso del *Hi* que obtiene el *mod*, se toma el resto de dicha división siempre positivo tal y como lo hace Logisim y lo define Python.
- En el caso del ***mulu*** el procedimiento realizado fue: exactamente el mismo que cuando se multiplica dos números decimales, en binario también se aplica el mismo procedimiento, un bit de un número multiplicado por el otro número en su totalidad, aplicar el desplazamiento correspondiente y luego sumar los resultados obtenidos. Pues aquí es igual, pero primero, al ser *unSigned*, se debe convertir el número en positivo por lo que se desplaza los 32 bits hacia la derecha, obteniendo el bit 0 sobrante como un carry y el bit 31 vacío se establece en 0. Esto pasa con ambos números, y posteriormente se multiplican. Pasan dos cosas, que falta multiplicar con el carry de A y el carry de B, por lo que el resultado de la multiplicación que iría para el *Lo* se desplaza dos posiciones hacia la izquierda al igual que el *Hi* obtenido donde los bits 0 y 1 del *Lo* pasan a ser 0, y los bits 30 y 31 del mismo pasan a ser los bits 0 y 1 del *Hi*. Finalmente, si el carry de A era 1 se suma *Lo* con B, los mismo para el caso contrario, y si ambos carrys lo fueron, se le suma 1 al final a *Lo*, y el *carry out* de dicha suma si se efectuaron son sumadas al Hi.

- Otras entradas como el *enable de lectura o escritura* en la memoria RAM, condicionaban la salida *Go Instruction* que manda a la caché de instrucciones a dar la próxima instrucción.
- Se empleó un multiplexer para el Data Input ya que dependiendo de la instrucción puede ser el *Hi* o *Lo* calculado de una multiplicación, una entrada del Keyboard, el resultado de una operación u otras.
- Salidas como Direction o Data, son los respectivos valores que se le propician al ***Write RAM*** o ***Read RAM*** según la instrucción que esté activa y en caso de ser necesario.
- Y el TTYData, que son los 7 bits menos significativos de *A*.

Para probar el proyecto, solo deberá cargar los respectivos *bancos* y al iniciar el primer *tick* del reloj será el motor de arranque para leer en caché las instrucciones de la RAM, no hay que esperar a que se desactive el *reset*, en caso de probar otros casos diferentes, fíjese que el reloj no esté activo mediante un *rising edge* y presione ambos botones *reset*, luego cargue sus bancos y puede probar este nuevo caso.

Este proyecto fue sumamente interesante e importante ya que gracias a él pude comprender mejor como funciona un microprocesador de este tipo (S-MIPS) en su interior, el proceso en que desarrolla las operaciones, como almacena los resultados, y como particiona la memoria entre datos e instrucciones, lo cual fue muy novedoso para mí observar ese proceso. Puedo decir, que gracias a este proyecto comprendí mejor cómo funciona la estructura operacional fundamental de un sistema de computadora, con especial interés en la forma en que la unidad central de proceso (CPU) trabaja internamente y accede a las direcciones de memoria.