66.20 - Organización De Computadores
Trabajo Práctico 3 - Data Path

Andrew Parlane

21 de junio de 2018

Índice

Índice de figuras

1. Introducción

El objetivo de este trabajo se base en familiarizarse con la microarquitectura y la arquitectura de conjuntos de instrucciones de un CPU MIPS. Para simular el data path se usó el programa *DrMIPS*.

2. Implementación

El programa *DrMIPS* usa dos archivos de *JSON* para especificar el conjunto de instrucciones *(.set)* y el data path *(.cpu)*. Comencé con una microarquitectura monociclo y una de pipeline, y tuve que implementar las siguientes instrucciones:

- J - para la microarquitectura pipeline.
- JR - para los dos CPUs.
- JALR - para los dos CPUs.

2.1. J para pipeline.cpu

La instrucción J (para Jump) tiene opcode 2 (6 bits) y los otros 26 bits están el target. El resultado es: $PC = ((PC + 4) \,\&\, 0xF0000000) \,|\, (target << 2)$

Al principio intenté implementar el data path para ese instrucción en la etapa de *IF*. Desafortunadamente DrMIPS no tiene forma fácil para decodificar una instrucción sin usar un bloque de control, y solo puede haber un bloque de control en el data path. Así tuve que implementarle en la etapa de *ID* y introducir un ciclo de stall.

Figura 1 muestra mi implementación. Hay un distribuidor para obtener los 4 bits más altos de PC+4, y un concatenador para juntarle con el target, desplazado a la izquierda dos bits. Hay un MUX para elegir entre ese y *PC+4*. El mux es controlado por la señal *jump* que viene del unidad de control. Finalmente se introduce un ciclo de stall en IF/ID si hay un branch o un salto.

2.2. JR para unicycle.cpu

La instrucción JR (para Jump Register) tiene opcode 0 y es tipo *R*, con Rt, Rd y SHAMT en ceros, y func 8. El resultado es: PC = Rs.

El desafío principal para ese instrucción estuvo cómo decodificar el función. No se puede hacer lo con el unidad de control normal, hay que usar el unidad de control del ALU. Figura 2 muestra mi implementación. Reemplacé los dos muxes que eligieron el nuevo PC, por uno con cuatro entradas. Se elija cual a usar con dos bits concatinados. El bit más alto es "Jump o JumpRegz el bit más bajo es "Branch o JumpReg".

El unidad de control escribe un uno a la señal RegWrite cuando hay un instrucción con opcode 0. Ese significa que el resultado de la operación del ALU es escrito al registro especificado en Rd. Normalmente esto será un problema porque la instrucción JR no debería cambiar los valores de los registros. Pero para Rd es cero, así escribe al registro R0 que siempre es cero.

2.3. JALR para unicycle.cpu

La instrucción JALR (para Jump And Link Register), es casi el mismo de JR. Las únicas diferencias son que tiene func = 9 y que escribe PC+4 al registro especificado por Rd.

Añadí otra señal "link."l unidad de control del ALU, y Uso para elegir que valor a escribir al banco de registros. Figura 3 muestra mi implementación.

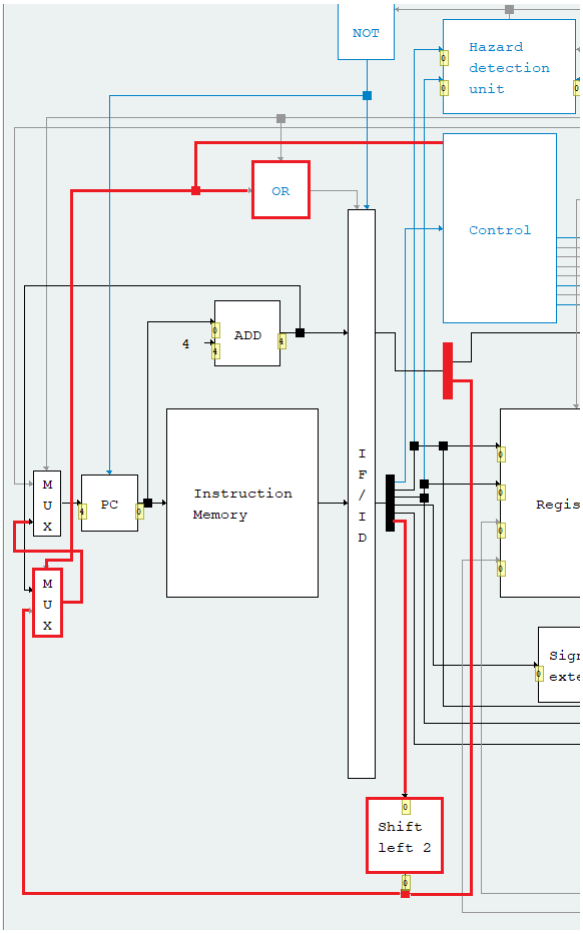

Figura 1: Implementación de la instrución J en la microarquitectura pipeline.cpu

Será posible solo tener una salida extra del unidad de control ALU para JR y JALR en vez de dos, si implementamos JR cómo "JALR Rs, R0".

2.4. JR y JALR para pipeline.cpu

La lógica para esas instrucciones es un poco más complicado en el CPU con un pipeline. Se ejecuta los saltos en la etapa de *EX*. Sería mejor hacerles en la etapa de *ID*, porque solo necesitaría un ciclo de stall en vez de dos.

Figura 4 muestra mi implementación. De nuevo cambio el mux de nuevo PC hasta uno con cuatro entradas, y la nueva entrada toma el valor del registro especificado en Rs durante la etapa de *EX*.

Tengo que flushear el registro ID/EX si hay un branch o un JR/JALR en la etapa de *EX* corriente. Y tengo que flushear el registro IF/ID en el mismo caso y también si hay un salto normal en la etapa de *ID*.

Hice una tabla de lógica para encontrar cómo elegir el nuevo PC. Tenemos tres señales de control:

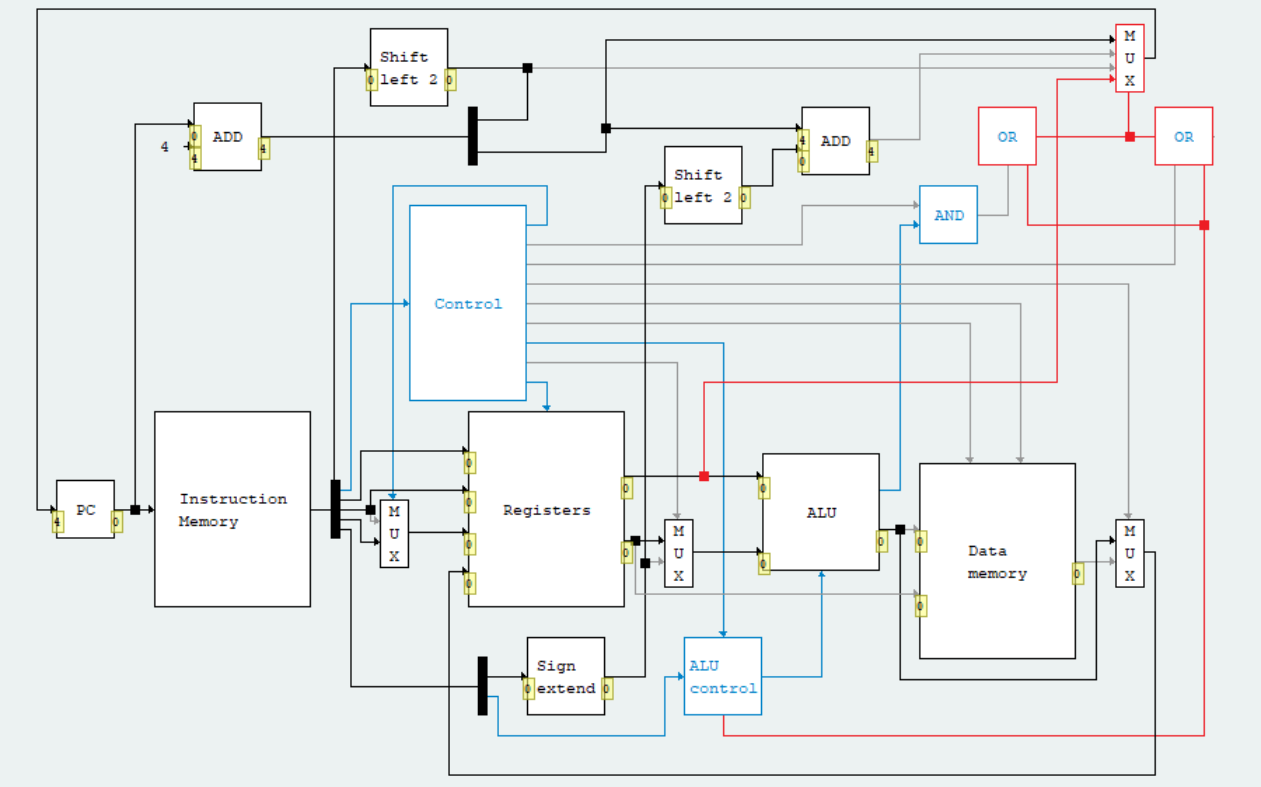

Figura 2: Implementación de la instrución JR en la microarquitectura unicycle.cpu

- J - viene de la etapa *ID*.
- JR - viene de la etapa *EX*.
- B - viene de la etapa *MEM*.

Hay cuatro entradas del MUX:

- 00 - PC+4
- 01 - branch target
- 10 - jump target
- 11 - jump register

Porque las señales de control viene de etapas deferentes, unas tienen que tener preferencia sobre otras. Específicamente B viene de la instrucción más viejo, después JR, después J.

J	JR	B	Mux	Mux1	Mux0
0	0	0	00 PC+4	0	0
0	0	1	01 Branch Target	0	1
0	1	0	11 Jump Register	1	1
0	1	1	01 Branch Target	0	1
1	0	0	10 Jump Target	1	0
1	0	1	01 Branch Target	0	1
1	1	0	11 Jump Register	1	1
1	1	1	01 Branch Target	0	1

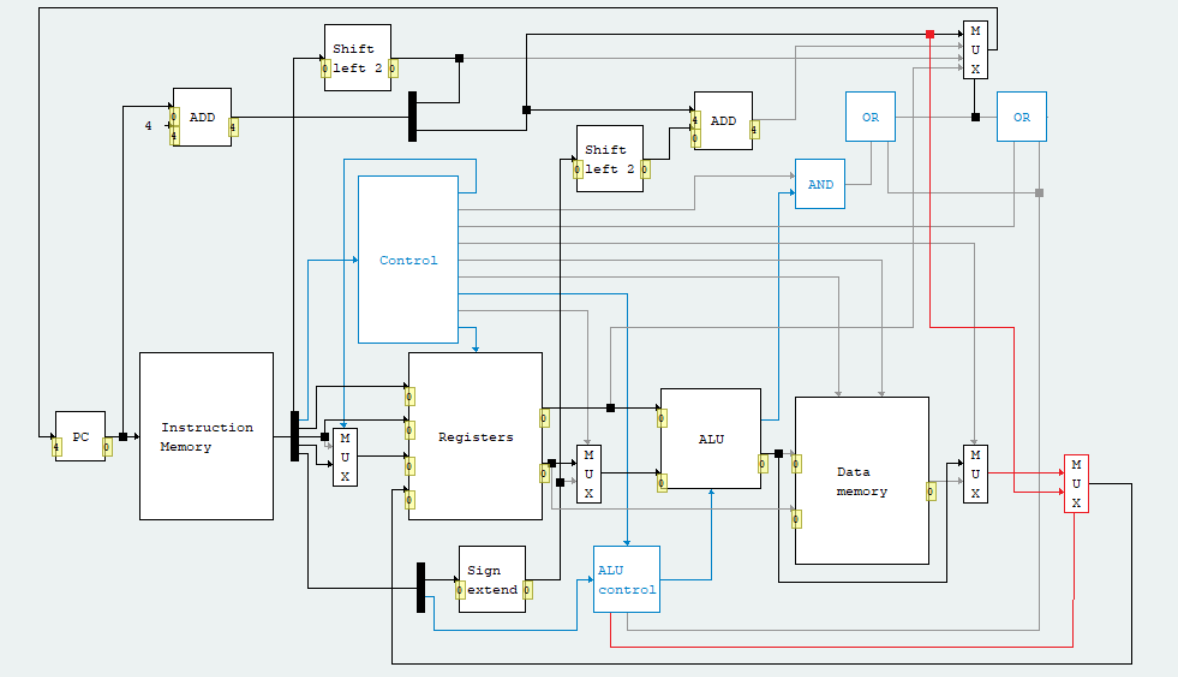

Figura 3: Implementación de la instrución JALR en la microarquitectura unicycle.cpu

La lógica es: Mux0 = JR o B, Mux1 = (J o JR) y !B.

Después tiene que escribir PC+4 al banco de registros en el registro especificado por Rd. Hay un riesgo de data RAW aquí, porque la siguiente instrucción puede usar el registro de destino como fuente, antes de que es escrito. Así usa *forwarding* para pasar ese PC+4 a la etapa de *EX*.

3. Verificación

Escribí una prueba para verificar el comportamiento de mis diseños. Tiene varios combinaciones de operaciones aritméticos, J, JR, JALR y branch.

4. Archivos de diseño y pruebas

Los archivos de diseños y prueba pueden ser encontrado en mi repositorio de github: https://github.com/andrewparlane/orga6620/tree/master/tp3.

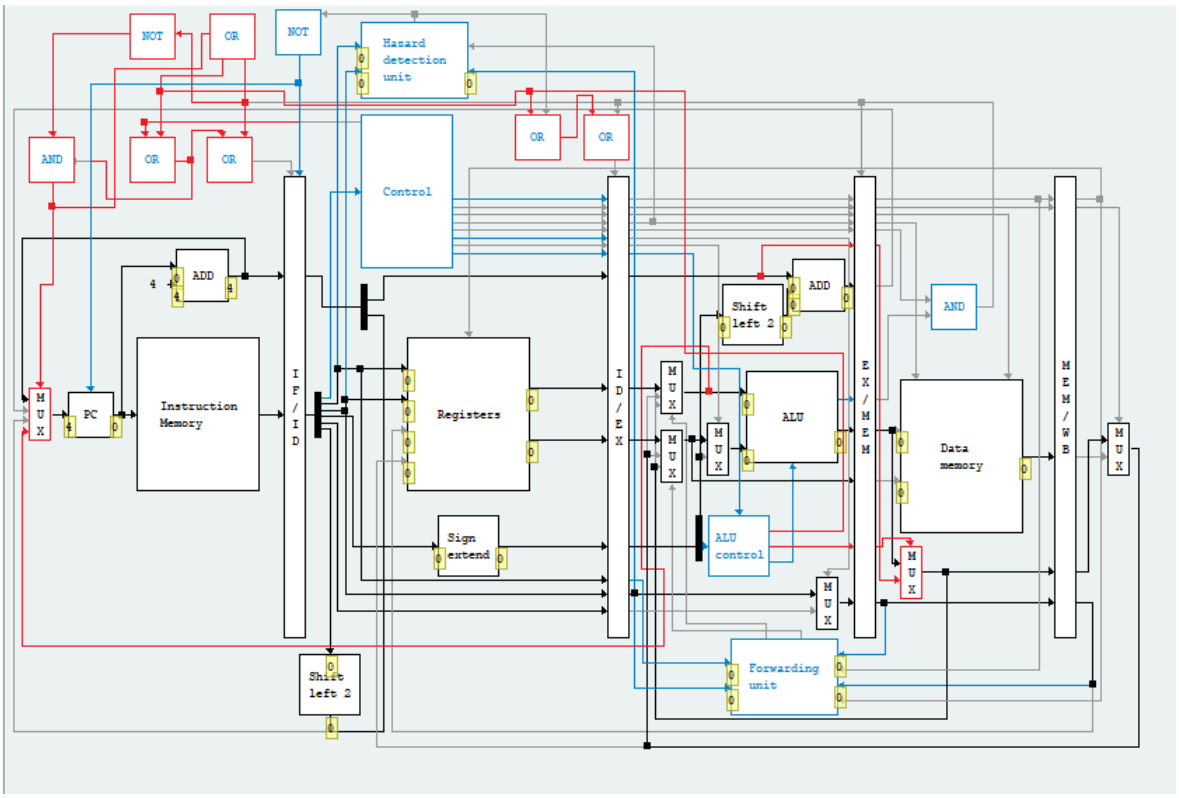

Figura 4: Implementación de la instrución JR y JALR en la microarquitectura pipeline.cpu