66.20 Organización de Computadoras: Trabajo Práctico 2

Eirik Harald Lund, *Padrón Nro. 103081*,
eirikharald@hotmail.com

Grupo Nro. ? - 1er. Cuatrimestre de 2018
66.20 Organización de Computadoras
Facultad de Ingeniería, Universidad de Buenos Aires

28/06/2018

Resumen

En este trabajo práctico, se iba a implementar unas funciones en el Datapath de una máquina. Las funciones eran saltos de tipos diferentes. Para hacerlo, había que poner los componentes para decidir si hay un salto y para cambiar el PC al valor correspondiente. En la parte con el procesador multicycle, se tenía que también verificar que no se causaran hazards.

1. Documentación

Se usaba un programa que se llama DrMips para mostrar y simular el funcionamiento de un prosesador MIPS32. Modificando archivos de JSON, se podía cambiar y añadir componentes y cables. Junto con un archivo del conjunto de instrucciones, se tiene un simulador.

El primer paso era implementar la instrucción Jump en un procesador con pipeline. Hay que elegir en cuál etapa del pipeline se va a implementarlo. Elegí la etapa de memoria. Como es una etapa al final y hay que *flush* o vaciar todas las anteriores, no es la opción más eficiente. No obstante, es más fácil de implementar, porque el funcionamiento de Branch ya está y la elección entre ellos no depende de qué etapa viene la señal. Para hacer el Jump, hay que tomar el valor inmediato de la instrucción, *left-shift* 2 bits y concatenar los 4 bits más significativos de PC+4. Con éste, se hace un

MUX con el valor objetivo de Branch, decidiendo usando la señal de Jump de Control. Es decir, se obtene el valor objetivo del Jump y luego decide si se va a usar éste, el valor de Branch, o PC+4. Para flush, se usa la señal de Jump también.

Para el segundo paso, se implementa Jump Register para un procesador unicycle. La diferencia principal entre Jump Register y Jump es de dónde se obtiene el valor objetivo. En lugar de hacer los pasos el párafo arriba, se toma el valor directamente de un registro. Como el procesador es unicycle, no se necesita flush. Una señal del ALUControl da si se toma un salto o no.

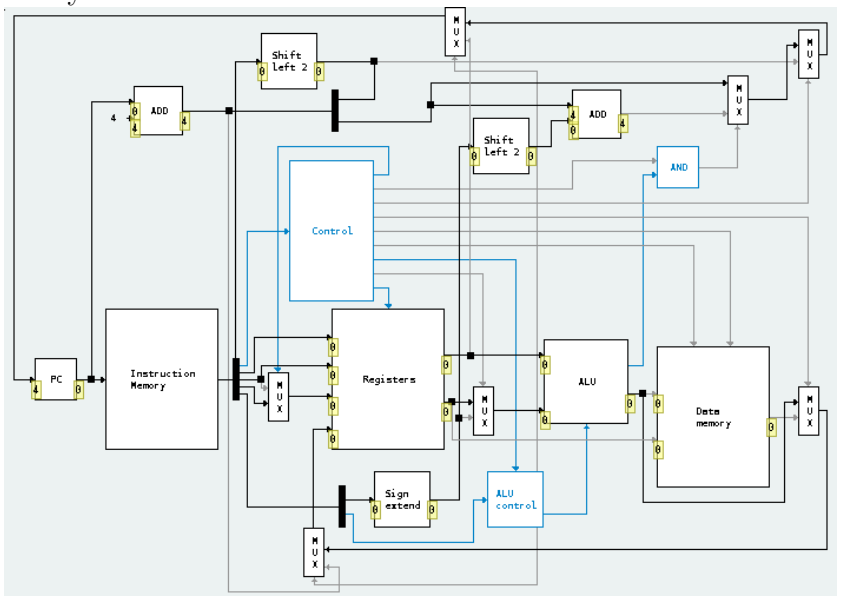
En el tercer paso, se agrega el mismo funcionamiento como en el último, siguiendo el mismo procedimiento del primero. Con este diseño, no se producen hazards, porque se toma la decisión en el paso casi final y se vacían los registros anteriores. Se pierden muchos ciclos, pero se evitan los hazards en una manera bastante fácil.

El cuarto paso se implementa agregando componentes para dirigir PC+4 a la salida final, para que se lo escriba al registro objetivo. Implica que hay que poner un MUX que toma PC+4 y la salida que ya está. El MUX se controla con una señal del ALUControl como en el segundo paso.

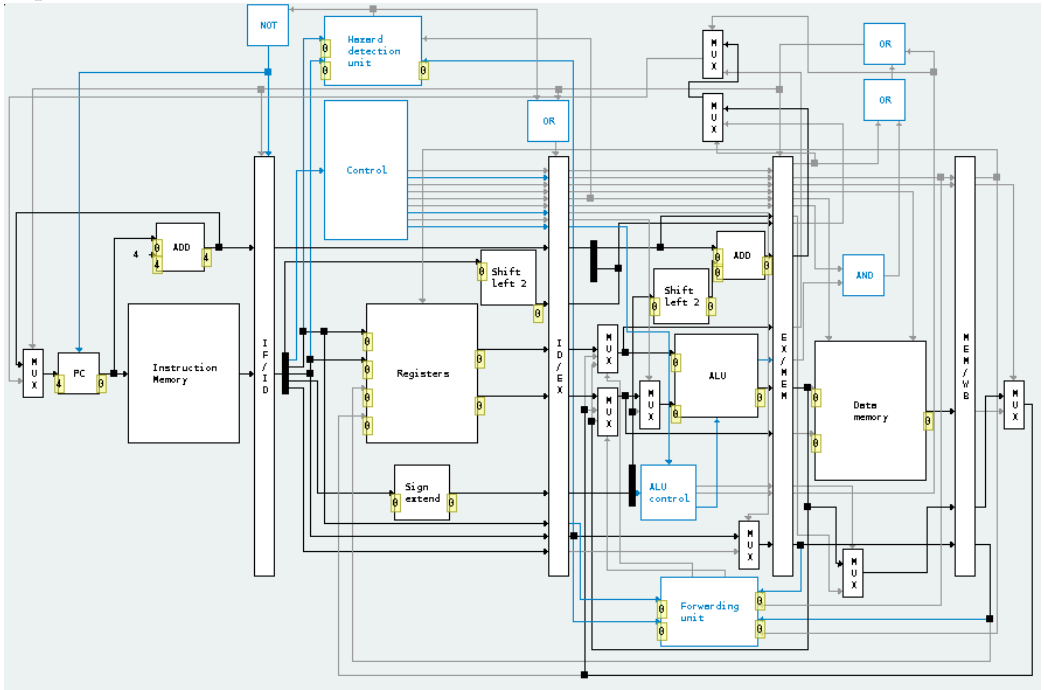
El último, quinto paso es implementar Jump and Link Register para el pipeline también. Primero, se agregan los mismos componentes como arriba. Luego, el desafío consiste en dirigir los valores a la etapa donde sean necesarios. No se cambia la parte que maneja los hazards, por lo tanto no causa más problemas.

2. Diseños

Unicycle:



Pipeline:



3. Corridas de prueba

Para saber si los comandos de saltos eran correctamente implementados, se hicieron unas corridas de prueba, con series de instrucciones Assembly que causarían problemas en el caso que no.

3.1. Prueba 1 - Jump Pipeline

La primera prueba simplemente verifica que la instrucción Jump funciona en el Pipeline:

```
0 j label
4 li $t2, 64
8 label:
12 li $t1, 1
```

0 es el valor predeterminado. El registro $t1$ termina con el valor 1 y $t2$ con 0, como es de esperar.

3.2. Prueba 2 - Jump Register Unicycle

Ahora verificamos que Jump Register funciona bien en el Unicycle:

```
0 li $t1, 12
4 jr $t1
8 li $t2, 64
12 li $t3, 1
```

$t1 = 12, t2 = 0, t3 = 1$

0 es el valor predeterminado, así que todos los registros tienen los valores esperados.

3.3. Prueba 3 - Jump Register Pipeline

Hacemos lo mismo con el Pipeline, y obtenemos el mismo resultado.

3.4. Prueba 4 - JALR Unicycle

```
0 li $t1, 12
4 jalr $t1, $t2
8 li $t2, 64
12 nop
```

$t1 = 12, t2 = 8$

PC = 8 representa la instrucción que viene después de la JALR, prueba exitosa.

3.5. Prueba 5 - JALR Pipeline

Con la misma prueba como arriba, terminamos con los mismos valores, prueba exitosa.

3.6. Prueba 6 - Hazards en Pipeline

Ésta contiene comandos de los dos saltos implementados; es decir Jump Register y Jump and link Register. También incluye una instrucción de Branch, para verificar que no se rompió. La idea con la prueba es ver que pasa cuando haya saltos que depende de valores que se calculan justo antes de ejecutarlos.

```
0  li $t1, 12
4  jalr $t1, $t2
8  b end
12 jr $t2
16 li $t4, 64
20 end: li $t3, 1
```

$t1 = 12, t2 = 8, t3 = 1, t4 = 0$

El valor de t2 viene de JALR y el de t3 del final. t4 contiene 0, así que se sabe que la instrucción 16 no se ejecutó. Como se explicó arriba, el diseño no produce hazards porque se vacían todos los registros intermedios. Sin embargo, si hubiera branch prediction, por ejemplo con un Branch Target Buffer, la predicción podría causar problemas, si es suficientemente rápida para extraer la instrucción siguiente.

4. Conclusioń

Este trabajo práctico te hace pensar en a dónde van las señales y valores durante la ejecución de un programa. En el caso del unicycle, se puede seguir el recorrido de una instrucción durante un ciclo, sabiendo que no hay interferencia de otras. Con el pipeline, hay que tener en cuenta que se guarda cada valor durante cada etapa. Si se ponen mal los cables, se puede terminar con un valor de otra istrucción. El equilibrio entre facilidad de implementación y desempeño es un problema muy importante. En este caso, elegí la facilidad, a costa del desempeño. Por otra parte, se resuelve el problema de hazard al

mismo tiempo, que también facilita el desarrollo. En un caso real, sería más sensato poner esfuerzo para llegar a una solución más eficiente.