Trabajo Práctico 2: MIPS Datapath

66.20 Organización de las Computadoras

Nicolás Calvo, *Padrón Nro. 78.914*
nicolas.g.calvo@gmail.com
Celeste Maldonado, *Padrón Nro. 85.630*
maldonado.celeste@gmail.com
Matias Acosta, *Padrón Nro. 88.590*
matiasja@gmail.com

2do. Cuatrimestre de 2011

Facultad de Ingeniería, Universidad de Buenos Aires

Índice

1. Introducción **1**

2. Desarrollo **2**

3. Conclusión **14**

1. Introducción

En este trabajo, se busca el objetivo de familiarizarse con el camino de datos de la arquitectura MIPS y la implementación del pipeline de cinco etapas.

Para la realización del trabajo práctico se implementaron segmentos de código Assembly MIPS que fueron analizados posteriormente mediante a herramienta de simulación WinDLX .

La herramienta WinDLX permite simular una arquitectura MIPS 32 con pipeline de cinco etapas, pudiendo ver los valores de cada registro dentro del archivo de registros, la ejecución de instrucciones dentro de cada etapa del pipeline y el acceso a estadísticas de la ejecución, como la cantidad de ciclos que requirió la ejecución o la cantidad y tipos de ciclos de stall.

Los resultados se analizaron con la opción de forwarding activada y sin activar. Forwarding es la técnica utilizada para reducir la cantidad de ciclos de stall producidos ante un riesgo Read after Write (RAW), la cual consiste en adelantar el resultado de la etapa de ejecución de la instrucción que escribe el registro a ser leído en una instrucción próxima, de manera que las instrucciones siguientes que requieran del contenido actualizado de dicho registro puedan acceder al mismo antes de haber sido escrito en la etapa de Write-Back.

2. Desarrollo

1. Realizar las siguientes operaciones:

```
A=4
B=5
C=7
D=3
X=B+C
Y=D-C
Z=(((((A+C)-B)*C)-X)/Y)
A=X*Y*3
```

   Donde las variables A,B,C,D están en memoria. La variable Z es de tipo double. Determinar el CPI para el conjunto de las operaciones, cantidad y tipo de Stalls.

   Resolución:

2. Realizar un reordenamiento del código assembly implementado en el ítem anterior que disminuya el CPI.

Resolución:

3. Implementar el siguiente segmento de código en Assembly MIPS:

```
int a = 0;
int i = 0;

for(i = 0; i < 10; ++i){
    if( i % 2 ){
        a += i;
    }

    if( a > 11){
        break;
    }
}
```

Determinar el CPI para el conjunto de las operaciones, cantidad y tipo de Stalls. Verificar la posibilidad de utilizar branch delay slot para realizar una mejora en el tiempo de ejecución.

Resolución:

Ejecución del programa sin forwarding

Se ejecutan 129 ciclos y 64 instrucciones , con 2 instrucciones en el pipeline al finalizar.

$$CPI = 129 ciclos / 64 instrucciones = 2{,}02$$

Se contaron un total de 64 stalls, divididos en las siguientes categorías:

- 10 stalls de control (7.75
- 2 stalls correspondientes a la instruccion trap (1.55
- 52 stalls RAW (Read After Write) (40.31

Ejecución del programa con forwarding:

Se ejecutan 98 ciclos y 64 instrucciones , con 2 instrucciones en el pipeline al finalizar.

$$CPI = 98 ciclos / 64 instrucciones = 1,53$$

Se logró un $SpeedUp = 1{,}32$

Se contaron un total de 33 stalls, divididos en las siguientes categorías:

- 10 stalls de control (10.20
- 2 stalls correspondientes a la instruccion trap (2.04 % de todos los ciclos)
- 21 stalls RAW (Read After Write) (21.43 % de todos los ciclos), de los cuales los 21 corresponden a stalls de branch, es decir, que dichos branches tienen por argumentos registros escritos en la instruccion que los precede.

Para el codigo presentado Branch Delay Slot no podrÃa usarse para lograr una mejora significativa en tiempo de ejecución debido a las dependencias de los branches respecto a los argumentos de las instrucciones que se ejecutan antes y despues de ellos.

Por ejemplo, para el segmento:

```
andi r3,r2,#1
bnez r3,Modulo
add r1,r1,r2
Modulo: sgt r8,r1,r6
```

La instruccion andi se usa en el branch, por lo que no puede moverse al delay slot y sgt requiere de un registro que es modificado en caso de no tomar el branch, por lo que mover esta instruccion al delay slot modificarÃ¡ la logica del programa.

4. Realizar un reordenamiento del código assembly implementado en el ítem anterior que disminuya el CPI.

   Resolución:

   Se reordenó el codigo moviendo a la instrucción

```
andi r3,r2,#1
```

   de la siguiente forma:

   Codigo original:

```
sge r8,r2,r5
bnez r8,Fin
andi r3,r2,#1
bnez r3,Modulo
add r1,r1,r2
```

   Codigo reordenado:

```
sge r8,r2,r5
andi r3,r2,#1
bnez r8,Fin
bnez r3,Modulo
add r1,r1,r2
```

   De esta forma se espera reducir los ciclos de stall RAW para que bnez r3, Modulo tenga disponible el valor del registro r3. Algo similar sucedería para bnez r8,Fin respecto a sge r8,r2,r5.

   Se obtuvieron los siguientes resultados:

   Ejecucion sin forwarding:

   Se ejecutan 108 ciclos y 64 instrucciones , con 2 instrucciones en el pipeline al finalizar.

$$CPI = 108 ciclos / 64 instrucciones = 1{,}69$$

Se contaron un total de 43 stalls, divididos en las siguientes categorías:

- 10 stalls de control (9.26
- 2 stalls correspondientes a la instruccion trap (1.85
- 31 stalls RAW (Read After Write) (28.70

SpeedUp respecto al codigo $sinreordenamiento = 1{,}2$

Ejecución con forwarding:

9 Se ejecutan 84 ciclos y 64 instrucciones , con 2 instrucciones en el pipeline al finalizar.

$$CPI = 84ciclos/64instrucciones = 1{,}31$$

SpeedUp respecto al codigo$sinreordenamiento = 1{,}17$

Se contaron un total de 19 stalls, divididos en las siguientes categorías:

- 10 stalls de control (11.09
- stalls correspondientes a la instruccion trap (2.38
- stalls RAW (Read After Write) (8.33

5. Implementar un segmento de código que con la adición de instrucciones de tipo nop mejore su desempeno. Si no fuera posible, explicar los motivos.

   Resolución:

   Riesgos Estructurales

   Los riesgos reducen el rendimiento de la velocidad ideal lograda por la segmentación. Hay tres clases de riesgos:

   1. Riesgos estructurales surgen de conflictos de los recursos, cuando el hardware no puede soportar todas las combinaciones posibles de instrucciones en ejecuciones solapadas simultáneamente.

   2. Riesgos por dependencias de datos surgen cuando una instrucción depende de los resultados de una instrucción anterior, de forma que, ambas, podrían llegar a ejecutarse de forma solapada.

   3. Riesgos de control surgen de la segmentación de los saltos y otras instrucciones que cambian el PC.

   Riesgos por dependencias de datos:

   El problema de la dependencia de datos entre una instrucción I1 y otra I2 que sigue, puede prevenirse retrasando la ejecución de I2 un número de K de etapas hasta que desaparezca el problema de que I2 lea un operando que I1 no ha escrito todavía.

   Este retraso puede conseguirse insertando un número K de instrucciones entre I1 e I2. Esto significa que el compilador tiene que reordenar el programa para encontrar K instrucciones que puedan ejecutarse después de I1 y antes de I2 sin que ello varíe la estructura lógica del problema.

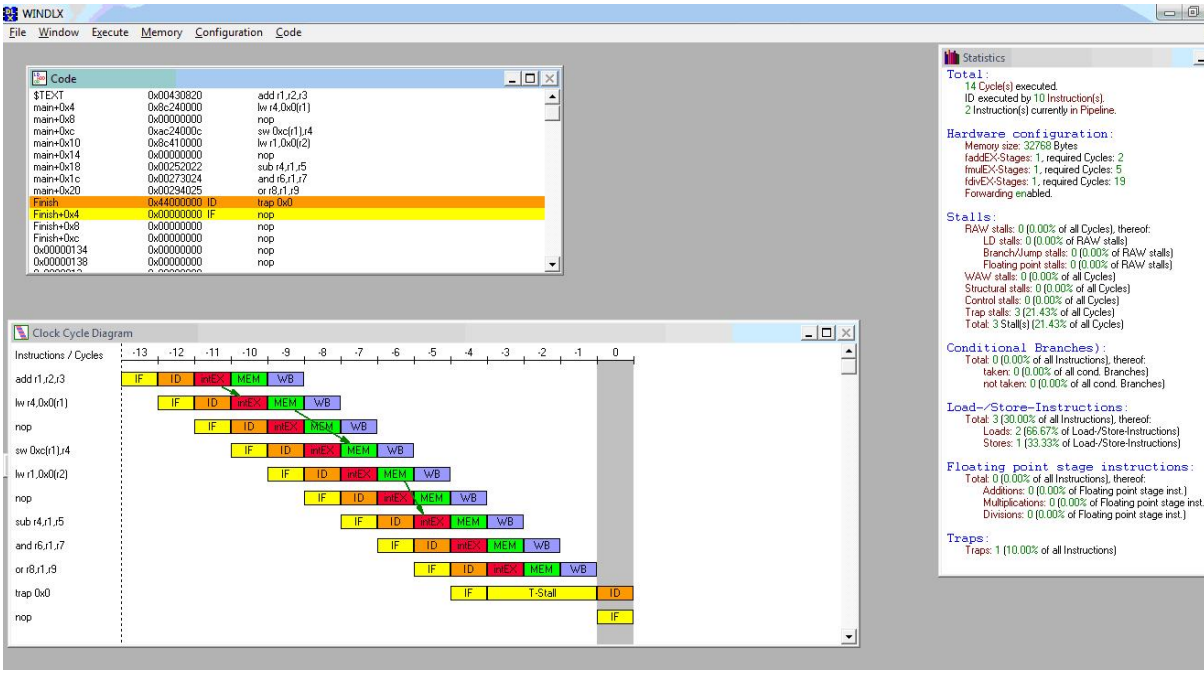
   Si el compilador no puede reorganizar el código para encontrar esas K instrucciones antes mencionadas, sin modificar la lógica del programa, debe insertar operaciones NOP (no operación) entre I1 e I2.

Si se trata de una instrucción de salto, hasta que no llega a la etapa de ejecución no se establece en el PC la dirección de la siguiente instrucción a ejecutar, por lo que la etapa alimentación de instrucción no sabe por donde seguir alimentando instrucciones. La etapa de alimentación no puede extraer la siguiente instrucción a una bifurcación hasta que esta última no finalice su etapa de ejecución, ésta tendría que esperar hasta que se alimente la siguiente instrucción y vaya avanzando por todos las etapas anteriores, que se habrán quedado vacías. A estas etapas vacías se las llama huecos de retardo (delay slots).

Nos encontramos con la misma solución, en la cual debemos reorganizar el código para rellenar los huecos de retardo con instrucciones útiles. Si no es posible reordenar el código sin afectar la semántica, se debe insertar operaciones NOP en los huecos de retardo.

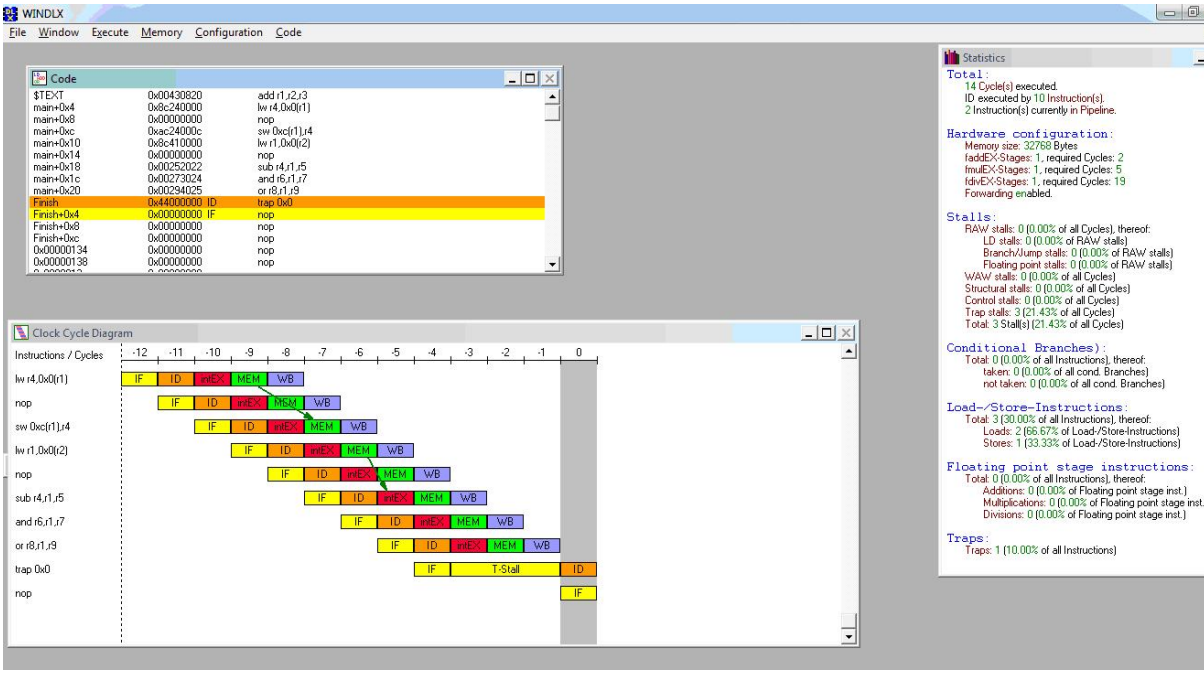
A continuación mostraremos un segmento de código con riesgo RAW:

```
main:
                add             r1,r2,r3
                lw              r4,0(r1)        ;load into r4 using r1
                sw              12(r1),r4       ;store r4 using r1
                lw              r1,0(r2)
                sub             r4,r1,r5
                and             r6,r1,r7
                or              r8,r1,r9
Finish:
                trap            0
```



Ahora, utilizaremos nop para evitar los riesgos producidos.

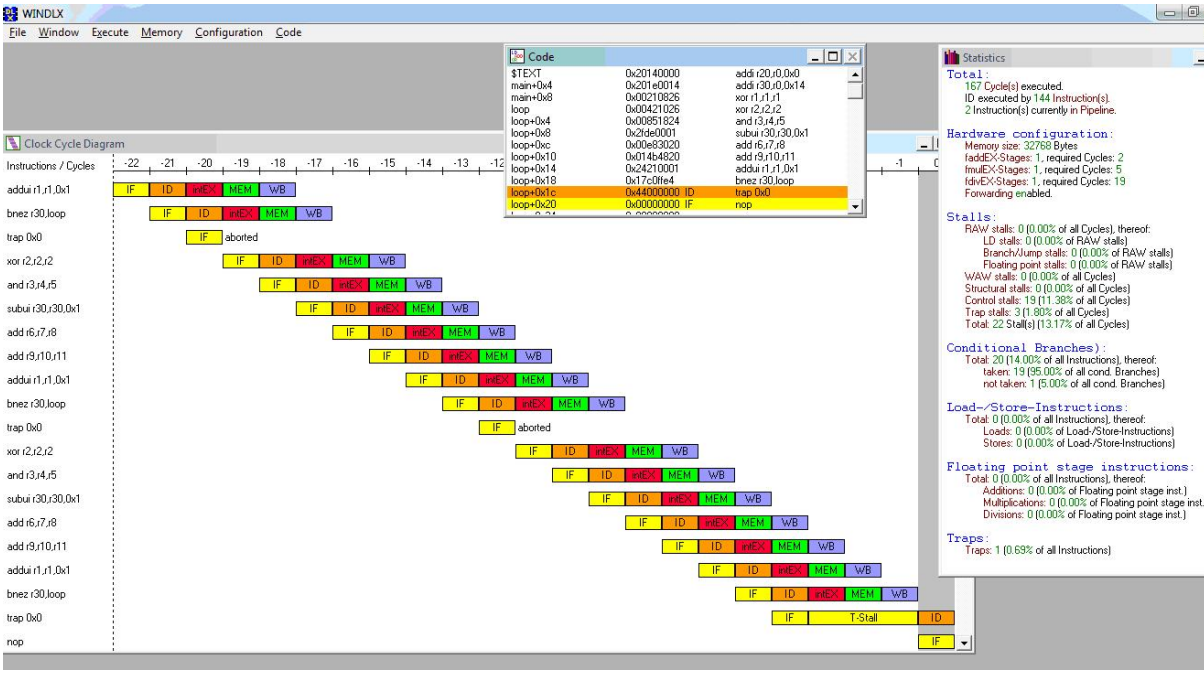
```
main:
                add             r1,r2,r3
                lw              r4,0(r1)        ;load into r4 using r1
nop
                sw              12(r1),r4       ;store r4 using r1
                lw              r1,0(r2)
nop
                sub             r4,r1,r5
                and             r6,r1,r7
                or              r8,r1,r9
Finish:
                trap            0
```



Como se puede apreciar, no hay beneficio en cuanto a la cantidad de ciclos necesarios para ejecutar el programa.

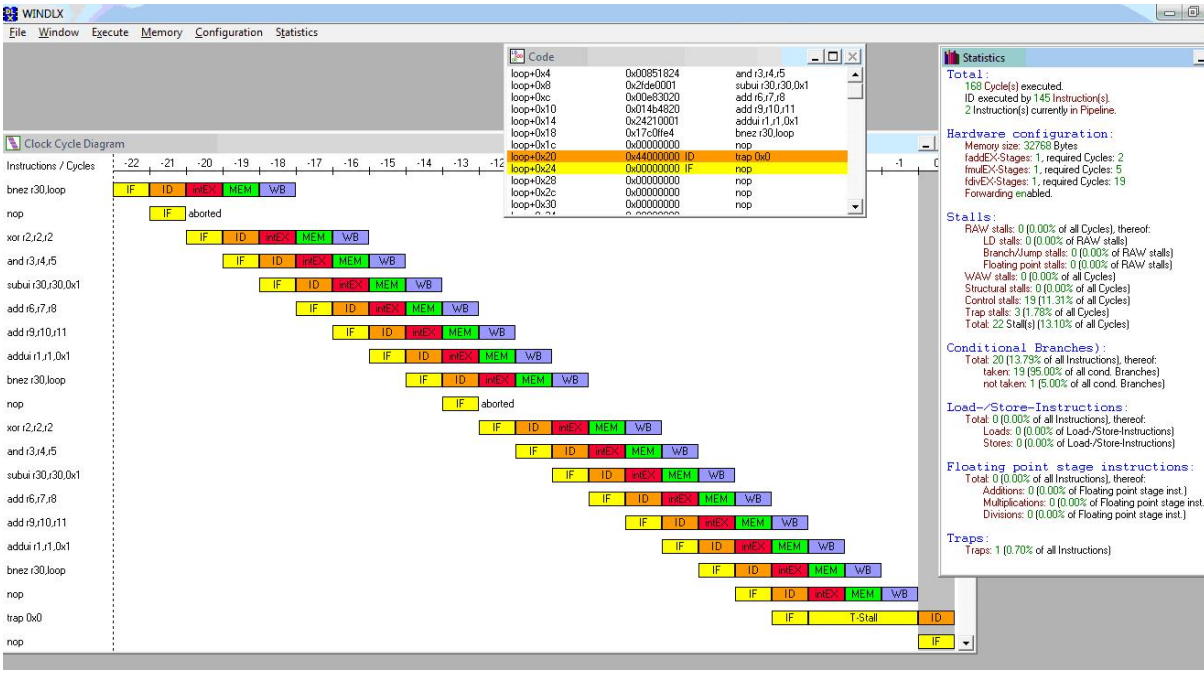
Ahora veremos un riesgo por salto condicional.

```
.data 0
dato:  .word 1
.text
main:
addi r20, r0, dato
addi r30, r0, 20
xor r1, r1, r1
loop:  xor r2, r2, r2
and r3, r4, r5
subui r30, r30, 1
add r6, r7, r8
add r9, r10, r11
addui r1, r1, 1
bnez r30, loop
trap #0
```



Ahora utilizando nop:

```
.data 0
dato:  .word 1
.text
main:
addi r20, r0, dato
addi r30, r0, 20
xor r1, r1, r1
loop:  xor r2, r2, r2
and r3, r4, r5
subui r30, r30, 1
add r6, r7, r8
add r9, r10, r11
addui r1, r1, 1
bnez r30, loop
nop
trap #0
```



Se puede ver que el segmento de código se ha ejecutado en 168 ciclos, uno más que sin utilizar el comando NOP.

3. Conclusión

Como conclusión, podemos decir que el comando NOP, solo sirve para evitar riesgos pero no para acelerar el procesamiento del código.