Práctica 3

Arquitectura de Computadoras
Pipelining

Fernanda Mora, ITAM 5/04/2016

Objetivos

El objetivo de esta práctica es:

- Visualizar, a través del simulador WinMIPS, el funcionamiento de una arquitectura pipeline relativamente simple.
- Identificar el impacto que tienen los saltos y saltos condicionales sobre la ejecución del procesador.

En cada sección se respondieron preguntas y se usaron ejemplos.

Introducción

Pipelining es una técnica de implementación de un procesador en donde muchas instrucciones se traslapan en ejecución. ¿Para qué queremos que se traslapen? El objetivo final de usar Pipelining es disminuir el tiempo de ejecución de un programa, así como también era el objetivo final de diseñar una memoria caché al disminuir el tiempo de acceso a los datos/instrucciones de los programas.

Recordemos la siguiente ecuación que hemos usado todo el curso:

$$t_{ejec_{programa}} = n_{instruc} * ciclos_{instruc} * tiempo_{ciclo} \quad (1)$$

La idea es que el pipelining clásico mejore el $tiempo_{ciclo}$ promedio, pero no mejorará el ciclo de reloj de $ciclos_{instruc} = 1$, ni mejorará $n_{instruc}$.

Recordemos que las instrucciones del set de instrucciones MIPS en general siguen los siguientes cinco pasos (algunas modificaciones pueden ser hechas dependiendo del tipo de instrucción y del diseño del procesador):

1. **F:** Hacer fetch de la instrucción de memoria (de instrucciones).
2. **D:** Leer registros mientras se decodifica la instruccióna ejecutar. Este paso puede dividirse en dos en otras arquitecturas, pero el formato regular de MIPS permite que puedan ocurrir simultáneamente.

3. **E:** Ejecuta la operación: operación entre registros / operacion entre un registro y un inmediato o cálculo una dirección (referencia a memoria sumando base con offset).
4. **M:** Accesar a los operandos en la memoria de datos. Con un *load* se lee de memoria y con un *store* se escribe en memoria.
5. **W:** Escribir el resultado en un registro.

Cualquier programa está compuesto por estas instrucciones y deseamos que al ejecutarse en pipeline el tiempo de ejecución mejore. En general, entre más etapas se tengan de un tamaño similar, más rápido se ejecutará el programa. ¿Qué tanto más rápido? En el límite, cuando todas las etapas son del mismo tamaño (mismo $tiempo_{ciclo}$) y hay suficientes instrucciones por ejecutarse, esta mejora, denominada el $speed\_up$ (qué tanto más rápido es la implementación con pipeline que la implementación secuencial) será igual al número de etapas en el pipeline.

Se pueden tener tres tipos diferentes conflictos al implementar el pipeline:

- Conflictos estructurales: se da cuando el hardware no puede procesar todas las combinaciones posibles de instrucciones simultaneamente. Generalmente se resulven duplicando algún recurso las veces que sean necesarias.
- Conflictos de datos: el mas común es el read after write (WAR) en el cual una instruccion requiere del resultado de una instruccion previa. Algunas veces se pueden resolver usando bypassing.
- Conflictos de flujo de control: los saltos en un programa incurren casi siempre en algun tipo de penalizaci on.Hay varias técnicas de predicción de saltos para reducir dicha penalización.

Desarrollo

Preliminares

Con un editor convencional (por ejemplo, notepad), edite el siguiente programa y guárdelo como prueba1.s

```
1  ; Arquitectura de computadoras
2  ; Programa de demostracion.
3  .data
4  i:  .word32 0
5  j:  .word32 0
6  .text
7      daddi R2,R0,0;
8      daddi r3, R0, 0;
9      daddi r5,R0,10 ;
10 WHIL:   slt R6, R2, R5
```

```
11      beqz R6, ENDW
12      daddi r3, r3, 5
13      sw R3, j(r0)
14      daddi r2,r2,1
15      sw  r2,i(r0)
16      j WHIL
17  ENDW:   nop
18      halt
```

Este programa corresponde al programa en C:

```
1  int main() {
2      int i = 0; int j = 0;
3      while ( i < 10 ) {
4          j = j + 5;
5          i = i + 1;
6      }
7  }
```

Abra una ventana de comandos y compruebe que el programa es sintácticamente correcto mediante la instrucción:

```
1  C:> asm prueba1.s
```

Ejecute el simulador. Desde el menú file, cargue el programa prueba1.s y córralo paso a paso (F7). Resultado en la Figura 1.

Figura 2: Ejecución del programa prueba1.s

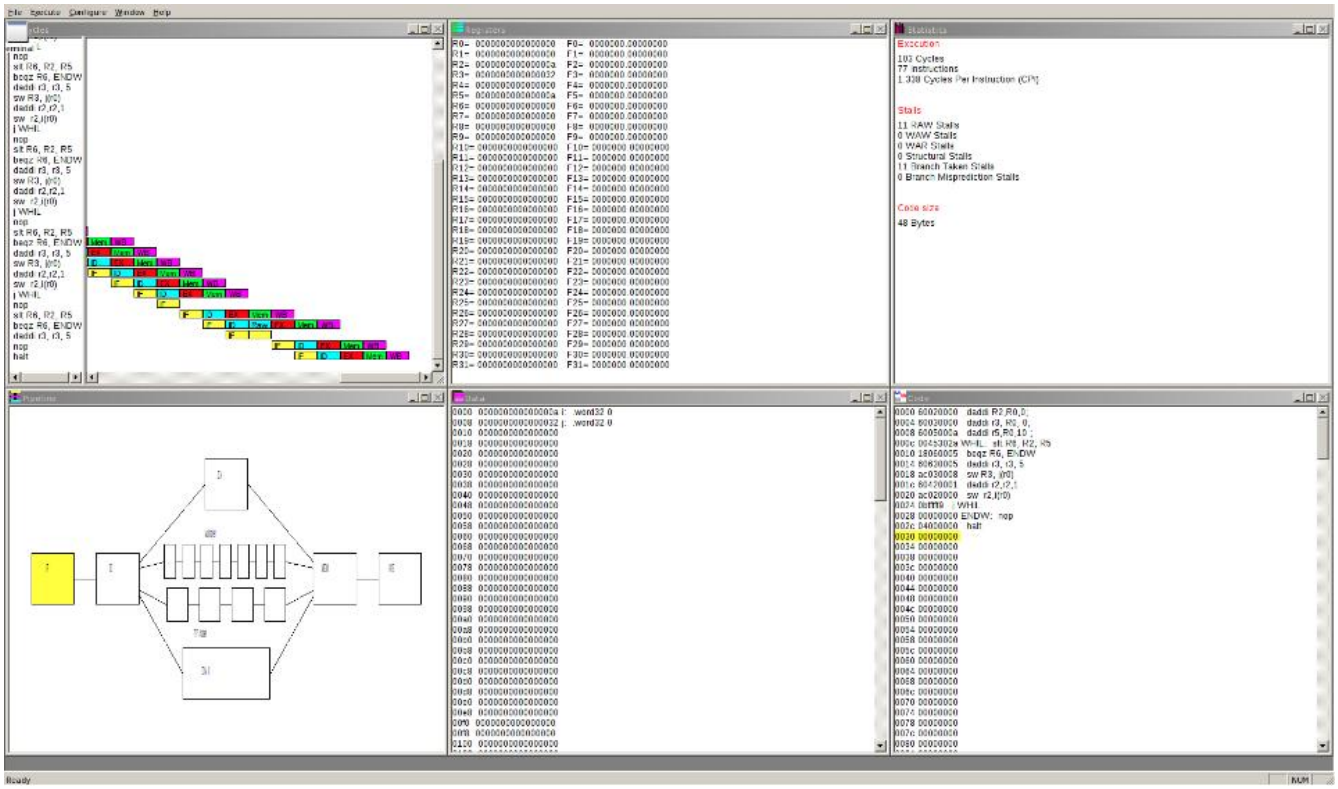

Figura 1: Ejecución de prueba1.s

Primera parte

- *¿Qué registros se utilizan para almacenar las variables i, j?*

  La variable “i” se almacena en el registro R2 y la variable “j” se almacena en el registro R3.

- *¿Para qué se utiliza la instrucción slt R6, R2, R5? ¿Qué ocurre si se intercambian los últimos dos registros de la instrucción?*

  Esta instrucción sirve para guardar el valor booleano de la condición del *while*. Guarda en R6 el valor de cero (falso) en caso de que R2 sea mayor o igual que R5 (i>=10). Si se intercambian R2 y R5 entonces la desigualdad del while cambia de menor a mayor.

- *¿Qué valores tienen i y j al final de la ejecución del programa?*

  Al final de la ejecución el valor de “i” es ‘a en hexadecimal (10 en decimal) y el valor de “j” es ‘32’ en hexadecimal (50 en decimal).

- *Modifique el programa anterior para que las variables y el código se almacenen a partir de las direcciones 100 y 200 de sus respectivos segmentos de datos y código*

  La instrucción .org indica la dirección de inicio en lenguaje ensamblador.

El código está almacenado en las direcciones 0000, 0004, ... ,002c, 0030.

i está almacenada en la localidad 0000 y j en la 0008.

La modificación queda así:

```
1  ; Ejercicio 1 d)
2  .data
3  .org 100
4  i:  .word32 0
5  j:  .word32 0
6  .text
7  .org 200
8      daddi R2,R0,0;
9      daddi r3, R0, 0;
10     daddi r5,R0,10 ;
11 WHIL:   slt R6, R2, R5
12     beqz R6, ENDW
13     daddi r3, r3, 5
14     sw R3, j(r0)
15     daddi r2,r2,1
16     sw  r2,i(r0)
17     j WHIL
18 ENDW:   nop
19     halt
```

Segunda parte

Escriba un programa que almacene las primeras 10 potencias de 2 en un arreglo de 10 elementos. El tamaño de los elementos es de 32 bits.

- *Utilice un algoritmo para calcular las potencias de 2 mediante multiplicación*

  En el simulador winmips64 la instrucción dmul toma 7 ciclos de ejecución. Por lo tanto, la instrucción siguiente sw r5,0(r3) se retrasa por 5 ciclos por conflictos de datos (tipo RAW) y además debe esperar un ciclo más a que el valor del registro se guarde en memoria. Como vimos en clase, este es un conflicto estructural, pues no se puede acceder a memoria dos veces al mismo tiempo. Además se tiene un conflicto por control de flujo. Después de la instrucción de branch entra al ciclo fetch la instrucción nop, por lo que se pierde un ciclo de ejecución más. Por lo tanto, en un total de 10 ciclos hay 50 conflicots de datos (RAW), 10 conflictos estructurales, y 9 conflictos de control de flujo (porque en la última iteración no hay conflicto). Por lo tanto, en total son 69 ciclos debidos a conflictos.

  Finalmente son 57 instrucciones en total: las 5 primeras más 5 por cada una de las 10 iteraciones más las últimas dos del nop y halt. El total de ciclos es de 57 + 69 + 4 (los ciclos de llenado de pipeline)= 130 ciclos.

$$CPI = \frac{130}{57} = 2{,}281.$$

```
 1  .data
 2  i: .word32 0
 3  .text
 4      daddi r1,r0,0;
 5      daddi r2,r0,10;
 6      daddi r3,r0,0;
 7      daddi r4,r0,2;
 8      daddi r5,r0,1;
 9  LOOP: daddi r1,r1,1
10      dmul r5,r5,r4;
11      sw r5,0(r3)
12      daddi r3,r3,8;
13      bne r1,r2,LOOP;
14  ENDW:   nop
15      halt
```

- *Ahora utilice un algoritmo basado en corrimientos a la izquierda*

  Los primeros cuatro ciclos ocurren sin conflictos. La instrucción daddi r1,r1,1 del loop entra a la etapa de fetch en el quinto ciclo de reloj. En el séptimo ciclo entra al fetch la instrucción de corrimiento y se ejecuta daddi r1,r1,1 (incrementa el contador del loop). En el octavo ciclo se ejecuta el corrimiento sin ningun conflicto. Sin embargo después del branch la instrucción nop entra a la etapa de fetch pero como se tiene que tomar el salto entonces se pierde ese ciclo de reloj durante 9 iteraciones (todas menos la última) del algoritmo. En total hay 9 retrasos debidos a conflictos por control de flujo. Como son 57 instrucciones, por lo tanto el número total de ciclos es de 4 + 57 + 9 = 70 y

$$CPI = \frac{70}{57} = 1{,}228.$$

```
 1  .data
 2  i: .word32 0
 3  .text
 4      daddi r1,r0,0;
 5      daddi r2,r0,10;
 6      daddi r3,r0,0;
 7      daddi r4,r0,2;
 8      daddi r5,r0,1;
 9  LOOP: daddi r1,r1,1
10      dsll r5,r5,1;
```

```
11      sw r5,0(r3)
12      daddi r3,r3,8;
13      bne r1,r2,LOOP;
14  ENDW:   nop
15      halt
```

- *Compare los tiempos de ejecución para cada caso. ¿Cuál es la principal fuente de la diferencia en los tiempos de ejecución?*

  Con *dmul* el CPI es de 2,281, mientras que con *dsll* es de 1,228. Asumiendo mismos tiempos de ciclo de reloj, entonces el programa que usa multiplicaciones tarda casi el doble que el que usa corrimientos. La principal fuente de esta diferencia se da porque la multipliación tarda 7 ciclos de reloj, mientras que el corrimiento a la izquierda sólo toma uno.

Tercera Parte: Loop unrolling

Considere el siguiente programa:

```
1  LOOP: lw r10,0(r1);
2      daddi r10,r10,4;
3      sw r10,0(r1);
4      daddi r1,r1,-4;
5      bne r1,r0,LOOP;
```

El cual corresponde al código:

```
1  FOR I := N DOWNTO 1 DO
2      A[I] := A[I] + 4;
3  END
```

Existen tres dependencias de datos (RAW) que no permiten ninguna reordenación del código (por parte del compilador) para evitar las paradas que aparecerán en el pipeline durante su ejecución.

En una primera aproximación se va a desenrrollar el bucle en cuatro copias, quedando de la forma siguiente:

```
1  .text
2      daddi r1,r0,128;
3  LOOP: lw r10,0(r1); Leer elemento vector
4      daddi r10,r10,4 ; Sumarle 4
5      sw r10,0(r1); Escribir valor
6
7      lw r11,-8(r1); 2a copia
8      daddi r11,r11,4;
9      sw r11,-8(r1);
10
11     lw r12,-16(r1); 3a copia
12     daddi r12,r12,4;
13     sw r12,-16(r1);
14
15     lw r13,-24(r1); 4a copia
16     daddi r13,r13,4;
17     sw r13,-24(r1);
18
19     daddi r1,r1,-32; indice
20     bne r1,r0,LOOP ; Fin de vector?
21 ENDW: nop
22     halt
```

Para evitar dependencias de datos se han empleado para cada copia registros distintos al original del bucle. También se han modificado los desplazamientos en las instrucciones de load y store para permitir el acceso a los elementos anteriores al indicado por la variable índice del bucle (registro r1). Así mismo, la actualización de r1 se ha modificado, sustituyendo la constante -4 por -16 con el fin de reflejar el procesamiento de los cuatro elementos del arreglo (cada elemento ocupa cuatro octetos de memoria).

Las dependencias de datos entre cada copia se mantienen (instrucciones lw, daddi y sw), para evitarlas puede reorganizarse el código de la forma siguiente:

```
1  LOOP:   lw r10,0(r1);
2      lw r11,-8(r1);
3      daddi r10,r10,4;
4      daddi r11,r11,4;
5      lw r12,-16(r1);
6      lw r13,-24(r1);
7      daddi r12,r12,4;
8      daddi r13,r13,4;
9      sw r10,0(r1);
```

```
10      sw r11,-8(r1);
11      sw r12,-16(r1);
12      sw r13,-24(r1);
13      daddi r1,r1,-32;
14      bne r1,r0,LOOP;
```

En este código la única dependencia de datos corresponde a las dos últimas instrucciones del bucle (registro r1). Con el desenrrollado que se ha realizado el bucle necesitará ejecutarse únicamente la cuarta parte de veces que el original.

Con el desenrollado que se ha realizado el bucle necesitará ejecutarse únicamente la cuarta parte de veces que el original.

a) Ejecute las tres versiones del código y verifique su ejecución.

El código que se usó fue:

```
1  .text
2       daddi r1,r0,128;
3  LOOP: lw r10,0(r1); Leer elemento
4       daddi r10,r10,4; Sumarle 4
5       sw r10,0(r1); Nuevo valor
6       daddi r1,r1,-8; Indice
7       bne r1,r0,LOOP; Fin del vector?
8  ENDW: nop
9       halt
```

b) Compare las estadísticas obtenidas en cada caso: Ciclos, instrucciones, CPI, riesgos RAW, Riesgos estructurales, Tamaño del código

Conclusiones

El uso de las diferentes técnicas de pipelining puede influir enormemente en el tiempo de ejecución de un programa. Los conflictos estructurales comúnmente se resuelven agregando más unidades de hardware para evitar conflictos de unidades funcionales.

Es importante el orden en el cual se ejecutan las instrucciones de un programa. Por ejemplo, en el caso de sumar una constante a elementos de un arreglo es conveniente agrupar *loads* en registros y las sumas de tal forma que no haya conflictos, y dejar los *stores* para el final. La técnica de loop unrolling es muy simple y efectiva. En el ejemplo se debe hacer en cada iteración del loop el load seguido de la suma y al final es store y estas tres operaciones involucran al mismo registro. Si se “desenrrolla” el loop mediante el uso de varios registros, por ejemplo cuatro, es fácil ver que se resuelven la mayoría de los conflictos de datos. El loop unrolling es una idea sencilla pero muy ingeniosa.

La eficiencia de un programa depende también de la forma en que está escrito. Por ejemplo, si se calculan las potencias del número 2 mediante multiplicación, entonces el programa es considerablemente más lento que si se hiciera mediante corrimientos a la izquierda. Esto se debe a que la operación de multiplicación requiere de un número considerablemente mayor de ciclos de reloj con respecto a la operación de corrimiento a la izquierda.

Por último, Winmips64 es un programa muy hermoso que te permite aprender fácilmente los conceptos más importantes de pipelining. Es interactivo y útil para visualizar el orden en el cual se ejecutan las instrucciones. Lo mejor es que tiene ventanas diferentes para ver el diagrama de los ciclos de ejecución, el contenido de los registros y de la memoria, el código y las estadísticas. El uso de los colores lo hace muy didáctico.

Referencias

[1] Stallings, William. Computer organization and architecture: designing for performance. Pearson Education India, 2000.

[2] Hennessy, John L., and David A. Patterson. Computer architecture: a quantitative approach. Elsevier, 2011.