Práctica 3: Simulador MIPS

Luis Jafet Ramos Castillo y Walter Martínez Santana

1. Objetivos

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.

2. Introducción

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

```
.data
i:      .word32 0
j:      .word32 0
.text
        daddi R2,R0,0;
        daddi r3, R0, 0;
        daddi r5,R0,10 ;
WHIL:   slt R6, R2, R5
        beqz R6, ENDW
        daddi r3, r3, 5
        sw R3, j(r0)
        daddi r2,r2,1
        sw      r2,i(r0)
        j WHIL
ENDW:   nop
        halt
```

Este programa corresponde al siguiente programa en C

```
int main(){
    int i= 0; int j= 0;
    while(i < 10){
        j = j + 5;
        i = i +1;
    }
}
```

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

```
asm prueba1.s
```

Ejecute el simulador. Desde el menú file, cargue el programa prueba1.s y córralo paso a paso (F7)

```
C:\Users\Walter\Downloads\winmips64>asm prueba1.s
Pass 1 completed with 0 errors
                ; Arquitectura de computadoras
                ; Programa de demostraci├n.
00000000        .data
00000000 00000000 i:    .word32 0
00000008 00000000 j:    .word32 0
00000000        .text
00000000 60020000       daddi R2,R0,0;
00000004 60030000       daddi r3, R0, 0;
00000008 6005000a       daddi r5,R0,10 ;
0000000c 0045302a WHIL: slt R6, R2, R5
00000010 18060005       beqz R6, ENDW
00000014 60630005       daddi r3, r3, 5
00000018 ac030008       sw R3, j(r0)
0000001c 60420001       daddi r2,r2,1
00000020 ac020000       sw      r2,i(r0)
00000024 0bffffff       j WHIL
00000028 00000000 ENDW: nop
0000002c 04000000       halt

Pass 2 completed with 0 errors
Code Symbol Table
               WHIL = 0000000c
               ENDW = 00000028
Data Symbol Table
                 i = 00000000
                 j = 00000008
```

Figura 1: Ejecución del simulador

3. Primera parte

a) ¿Qué registros se utilizan para almacenar las variables i, j?

Se utilizan los registros R2 y R3 respectivamente.

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

Para comparar los valores en los registros R2 y R5, es decir, se hace una comparación lógica y se almacena el resultado de la comparación en el registro R6. La operación lógica que realiza es la de "si es menor que". En el manual, se describe de la siguiente manera *slt reg,reg,reg - set if less than*. Si se intercambian los registros R2 y R5, se evalúa la condición lógica "menor que" y el resultado es 0, es decir, no se cumple, y por lo tanto, no entra al ciclo.

c) ¿Qué valores tienen i y j al final de la ejecución del programa?

En el simulador $i = a$ y $j = 32$ que son números hexadecimales que equivalen a $i = 10$ y $j = 50$ en decimal.

d) 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.

Se uso la instrucción del manual *.org<n>- start address.*

```
0000  0000000000000000
0008  0000000000000000
0010  0000000000000000
0018  0000000000000000
0020  0000000000000000
0028  0000000000000000
0030  0000000000000000
0038  0000000000000000
0040  0000000000000000
0048  0000000000000000
0050  0000000000000000
0058  0000000000000000
0060  0000000000000000
0068  0000000000000000 i:  .word32 0
0070  0000000000000000 j:  .word32 0
0078  0000000000000000
0080  0000000000000000
0088  0000000000000000
0090  0000000000000000
0098  0000000000000000
00a0  0000000000000000
00a8  0000000000000000
00b0  0000000000000000
00b8  0000000000000000
```

Figura 2: Datos

```
009c 00000000
00a0 00000000
00a4 00000000
00a8 00000000
00ac 00000000
00b0 00000000
00b4 00000000
00b8 00000000
00bc 00000000
00c0 00000000
00c4 00000000
00c8 60020000   daddi R2,R0,0;
00cc 60030000   daddi r3, R0, 0;
00d0 6005000a   daddi r5,R0,10 ;
00d4 0045302a WHIL:  slt R6, R2, R5
00d8 18060005   beqz R6, ENDW
00dc 60630005   daddi r3, r3, 5
00e0 ac030070   sw R3, j(r0)
00e4 60420001   daddi r2,r2,1
00e8 ac020068   sw  r2,i(r0)
00ec 0bfffff9   j WHIL
00f0 00000000 ENDW:  nop
00f4 04000000   halt
00f8 00000000
```

Figura 3: Código

```
.data
.org 100
i:   .word32 0
j:   .word32 0
.text
.org 200
     daddi R2,R0,0;
     daddi r3, R0, 0;
     daddi r5,R0,10 ;
WHIL:       slt R6, R2, R5
     beqz R6, ENDW
     daddi r3, r3, 5
     sw R3, j(r0)
     daddi r2,r2,1
     sw       r2,i(r0)
     j WHIL
ENDW:       nop
     halt
```

4. 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.

a) Utilice un algoritmo para calcular las potencias de 2 mediante multiplicación.

```
.data
j: .word32 0
potencia2: .word32 0,0,0,0,0,0,0,0,0,0
.text
daddi r1, r0, 1; total
daddi r2, r0, 0; index
daddi r3, r0, 2; potencia
daddi r4, r0, 10; size
daddi r5, r0, 0; counter

WHIL: slt R6, R5, R4
beqz R6, ENDW
dmul r1, r1, r3
sw r1, potencia2(r2)
daddi r2, r2, 4
daddi r5, r5, 1
sw r5, j(r0)
j WHIL
ENDW: nop
halt
```

b) Ahora utilice un algoritmo basado en corrimientos a la izquierda.

```
.data
j: .word32 0
potencia2: .word32 0,0,0,0,0,0,0,0,0,0
.text
daddi r1, r0, 1; total
daddi r2, r0, 0; index
daddi r4, r0, 10; size
daddi r5, r0, 0; counter

WHIL: slt R6, R5, R4
beqz R6, ENDW
dsll r1, r1, 1
sw r1, potencia2(r2)
daddi r2, r2, 4
daddi r5, r5, 1
sw r5, j(r0)
j WHIL
ENDW: nop
Halt
```

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

El principal problema radica en que cuando se lleva a cabo la operación de *shift*, ésta solo requiere un ciclo en la etapa de ejecución (EX), mientras que en la multiplicación, el pipeline pasa por cada uno de los 7 bloques de *multiplier*. Por lo tanto, la multiplicación es 7 veces más tardada que el corrimiento.

A continuación se muestran las estadísticas de ambas.

```
Execution
175 Cycles
89 Instructions
1.966 Cycles Per Instruction (CPI)

Stalls
61 RAW Stalls
0 WAW Stalls
0 WAR Stalls
10 Structural Stalls
11 Branch Taken Stalls
0 Branch Misprediction Stalls

Code size
60 Bytes
```

Figura 4: Multiplicación

```
Execution
115 Cycles
89 Instructions
1.292 Cycles Per Instruction (CPI)

Stalls
11 RAW Stalls
0 WAW Stalls
0 WAR Stalls
0 Structural Stalls
11 Branch Taken Stalls
0 Branch Misprediction Stalls

Code size
60 Bytes
```

Figura 5: Corrimiento

5. Tercera parte - Loop unrolling

Considere el siguiente programa.

```
LOOP:   lw r10,0(r1); Leer un elemento de un vector
        daddi r10,r10,4 ; Sumar 4 al elemento
        sw r10,0(r1); Escribir el nuevo valor
        daddi r1,r1,-4 ; Actualizar la var indice
        bne r1,r0,LOOP ; Fin de vector?
```

El cual corresponde al siguiente código.

```
FOR I := N DOWNTO 1 DO
A[I] := A[I]+4;
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 desenrollar el bucle en cuatro copias, quedando de la siguiente forma.

```
LOOP:   lw r10,0(r1); Leer elemento vector
        daddi r10,r10,4 ; Sumar 4 al elemento
        sw r10,0(r1); Escribir nuevo valor

        lw r11,-4(r1); 2nda copia
        daddi r11,r11,4 ;
        sw r11,-4(r1);

        lw r12,-8(r1); 3era copia
        daddi r12,r11,4 ;
        sw r12,-8(r1);

        lw r13,-12(r1); 4ta copia
        daddi r13,r11,4 ;
        sw r13,-12(r1);

        daddi r1,r1,-16 ; Actualizar indice
        bne r1,r0,LOOP ; Fin de vector?
```

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).

Asimismo, 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 4 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 siguiente forma.

```
LOOP:   lw  r10,0(r1);
        lw  r11,-4(r1);
        daddi  r10,r10,4  ;
        daddi  r11,r11,4  ;
        lw  r12,-8(r1);
        lw  r13,-12(r1);
        daddi  r12,r11,4;
        daddi  r13,r11,4;
        sw  r10,0(r1);
        sw  r11,-4(r1);
        sw  r12,-8(r1);
        sw  r13,-12(r1);
        daddi  r1,r1,-16;
        bne  r1,r0,LOOP  ;
```

En este código la única dependencia de datos corresponde a las dos últimas instrucciones del bucle (registro r1).

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.

```
Execution
300 Cycles
185 Instructions
1.622 Cycles Per Instruction (CPI)

Stalls
75 RAW Stalls
0 WAW Stalls
0 WAR Stalls
0 Structural Stalls
37 Branch Taken Stalls
0 Branch Misprediction Stalls

Code size
20 Bytes
```

Figura 6: Caso 1

```
Execution
300 Cycles
208 Instructions
1.442 Cycles Per Instruction (CPI)

Stalls
75 RAW Stalls
0 WAW Stalls
0 WAR Stalls
0 Structural Stalls
15 Branch Taken Stalls
0 Branch Misprediction Stalls

Code size
56 Bytes
```

Figura 7: Caso 2

```
Execution
300 Cycles
260 Instructions
1.154 Cycles Per Instruction (CPI)

Stalls
18 RAW Stalls
0 WAW Stalls
0 WAR Stalls
0 Structural Stalls
18 Branch Taken Stalls
0 Branch Misprediction Stalls

Code size
56 Bytes
```

Figura 8: Caso 3

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

	Ciclos	Instr	CPI	RAW	WAW	WAR	Struct	Branch	Código
Caso 1	300	185	1.622	75	0	0	0	37	20
Caso 2	300	208	1.442	75	0	0	0	15	56
Caso 3	300	260	1.154	18	0	0	0	18	56

Tabla 1: Estadísticas

El desempeño que se obtuvo al eliminar dependencias y ordenar el código fue mejor, basta sólo ver la reducción de RAW en más de 75 %, el aumento de las instrucciones en el mismo número de ciclos en más de 40 % y una mejora en CPI del caso 3, de manera que es 1.41 veces más rápido que el caso 1.

6. Conclusión

Se pudo visualizar el funcionamiento de una arquitectura pipeline e indentificar el impato que tienen los saltos y saltos condicionales sobre la ejecución del procesador.