2011/
2012

Simulación de procesadores superescalares con memoria real

Práctica 2

Alberto Manuel Mireles Suárez
David Guillermo Morales Sáez

ULPGC
Arquitectura de Computadores
2011/2012

Índice

Ejercicio 1

1. ¿Qué evento significativo ocurre en el ciclo 7? ¿Cómo es posible que la instrucción 0 entre tan pronto en el pipeline? Si es necesario, revisar la descripción de la etapa IF en la sección 3.

En el ciclo 7 se carga la primera instrucción del procedimiento VectProd, que es una instrucción adds. Es posible que la instrucción 0 entre tan pronto ya que la latencia de salto es de un único ciclo y consideramos que la predicción de salto es siempre correcta.

2. Explica la causa del ciclo de penalización que aparece en el ciclo 13. ¿Qué nuevo símbolo aparece y cuál es su significado? Para los que contesten que el problema es una dependencia RAW, sin más, fijarse que entre las instrucciones 14 y 15 también hay dependencia RAW y sin embargo no se produce penalización.

En la etapa ID del ciclo 13 marcamos la instrucción número 4 con un símbolo de exclamación, indicando que existe una dependencia RAW (Read After Write). Esto no sucede en la dependencia que existe entre las instrucciones 14 y 15 ya que se produce un adelantamiento de los datos de la etapa MEM a la EX en el ciclo 5.

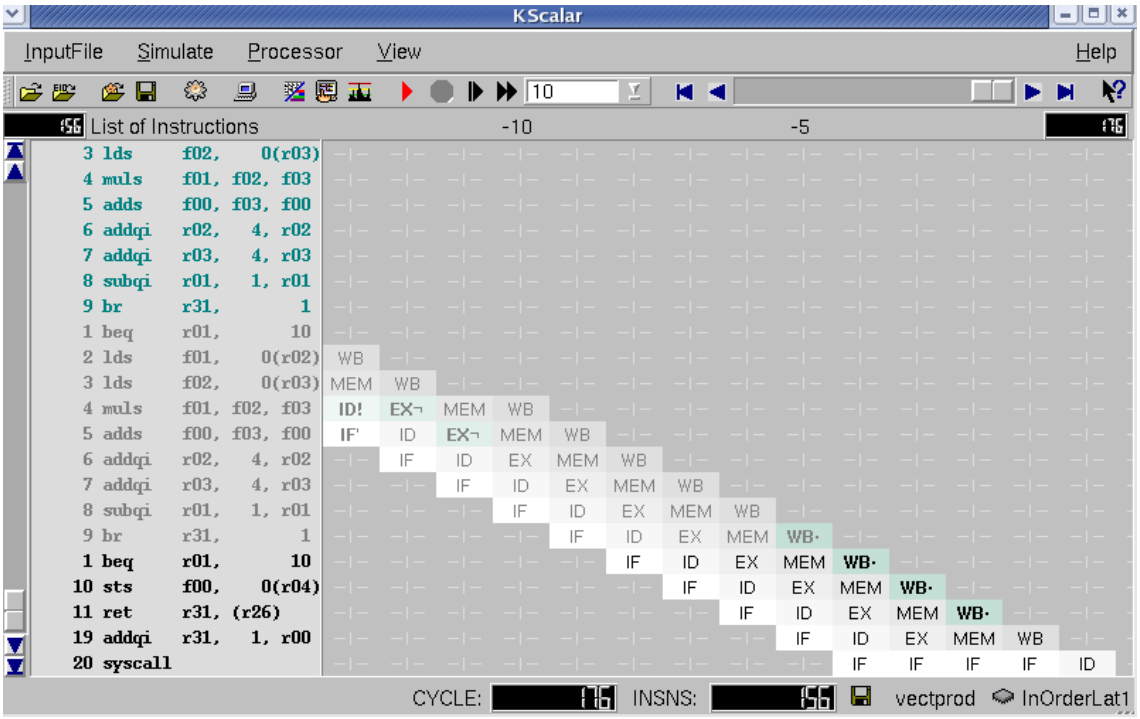
3. Encuentra el resto de símbolos que aparecen en las columnas que representan las etapas del pipeline y lo que significan.

El resto de símbolos que aparecen son:

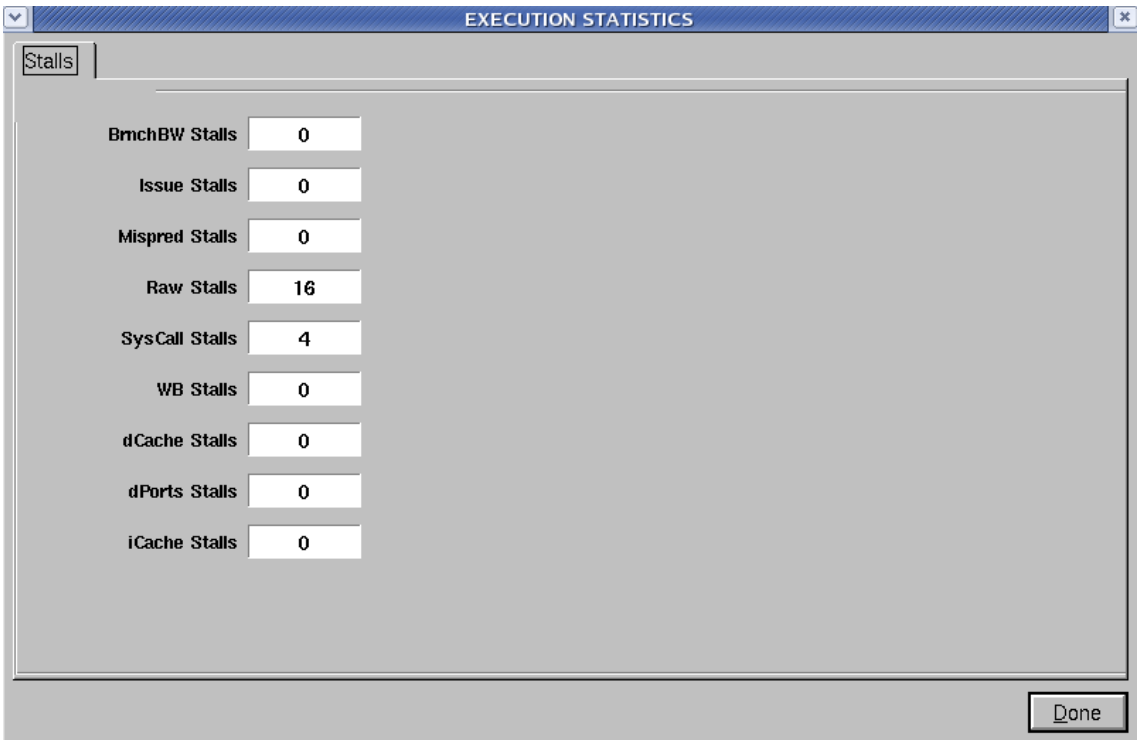
- · : Indica que no se escribe nada en la etapa WB
- ‘ : Indica que la instrucción se retiene debido a la paralización del cauce
- *: Indica que la instrucción se detiene debido otra instrucción está accediendo a los registros.
- @: Indica que el dato no se encuentra en la caché L1 y se ha de buscar en la caché L2.

4. ¿Qué puede ocurrir con la instrucción *syscall?* ¿Cuántos ciclos de penalización provoca?

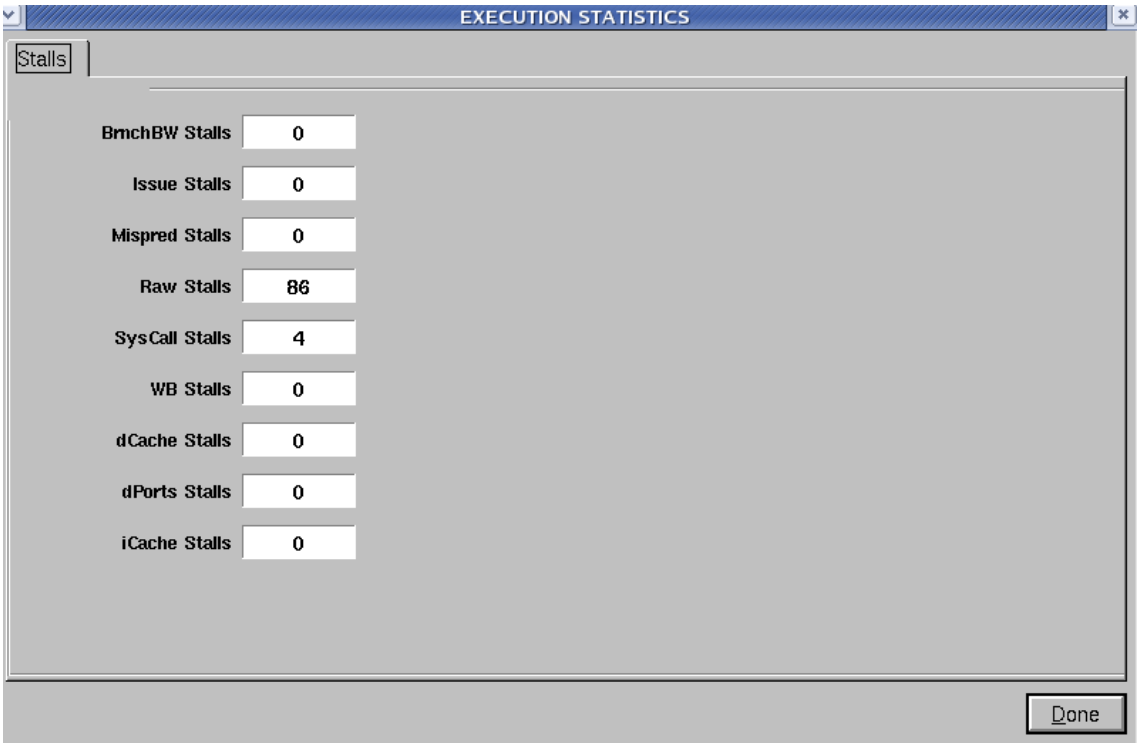
El syscall vacía el pipeline, por lo que provoca 4 ciclos de penalización tal y como podemos observar en el siguiente diagrama.



176 ciclos



246 ciclos



1. ¿Qué porcentaje de pérdida de rendimiento se produce? Para contestar, encontrar el número de ciclos de penalización añadidos respecto a la ejecución con caminos de realimentación.

Para calcular el porcentaje de pérdida, primero calcularemos el rendimiento de cada ejecución de manera independiente. Para calcular el rendimiento individual, hemos de estudiar el tiempo que tarda ejecutarse cada uno, y dado que vamos a comparar ambos la característica diferenciadora serán los ciclos de ejecución, por lo que obviaremos el tiempo de ciclo.

$$Rendimiento_X = \frac{1}{Ciclos} = \frac{1}{176} \approx 0.0057$$

$$Rendimiento_Y = \frac{1}{Ciclos} = \frac{1}{246} \approx 0.0041$$

$$\Delta Rendimiento = \frac{Rendimiento_Y - Rendimiento_X}{Rendimiento_X} = \frac{-0.0016}{0.0057} \approx -0.28$$

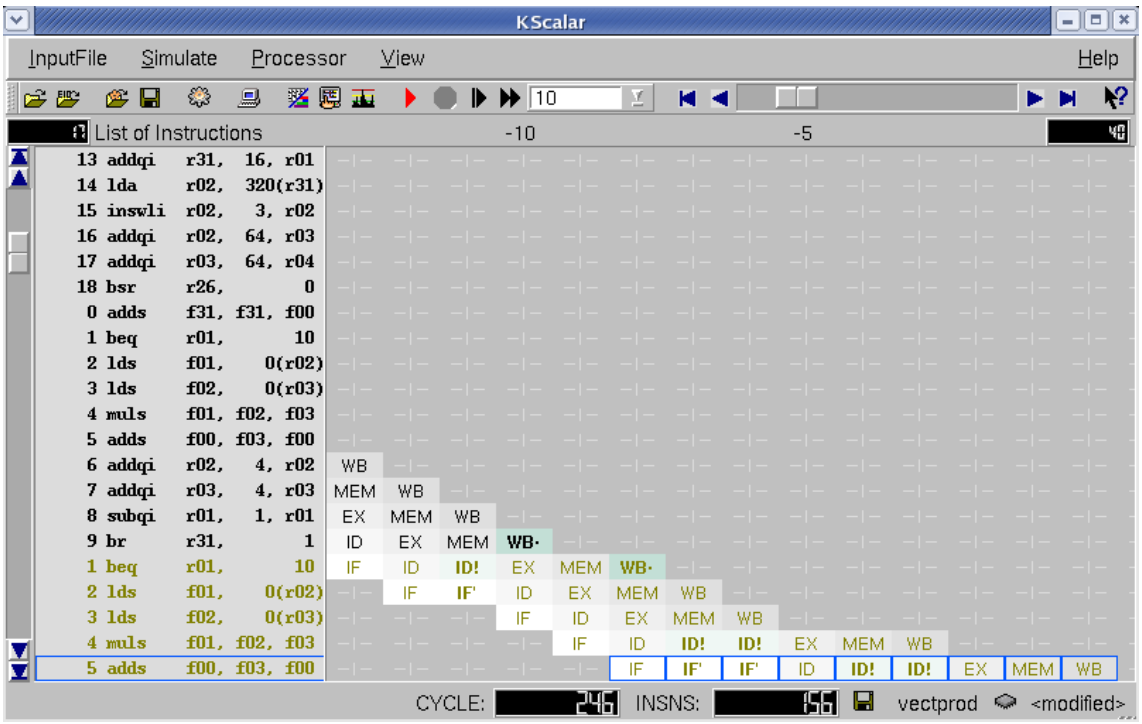
Como podemos comprobar tenemos una pérdida de rendimiento de un 28 %.

2. Mirando las estadísticas (o mirando el diagrama de ciclos con cuidado) se puede observar que la clase de penalización que aumenta es la de tipo RAW. Argumenta las causas.

El incremento de las penalizaciones por dependencias RAW se produce principalmente por la ausencia del adelantamiento de los datos ya calculados. Por tanto, siempre se producirán penalizaciones de al menos 2 ciclos cuando se dé este caso.

3. ¿Dónde (entre qué instrucciones) se produce el aumento de este tipo de penalización? Mirando con detalle el diagrama de ciclos de la ejecución completa, identificar todas las parejas de instrucciones que generan una penalización y el número de ciclos de penalización que producen. Fijarse en que no todas las parejas " problemáticas" provocan el mismo número de ciclos de penalización ¿de qué depende este número de ciclos?

Las penalizaciones por dependencias RAW se producen dentro del bucle en el cálculo del producto escalar en las instrucciones de salto, multiplicación y suma. En el caso de la instrucción de salto, sólo hay un ciclo de penalización, mientras que en las instrucciones de multiplicación y suma se producen dos.



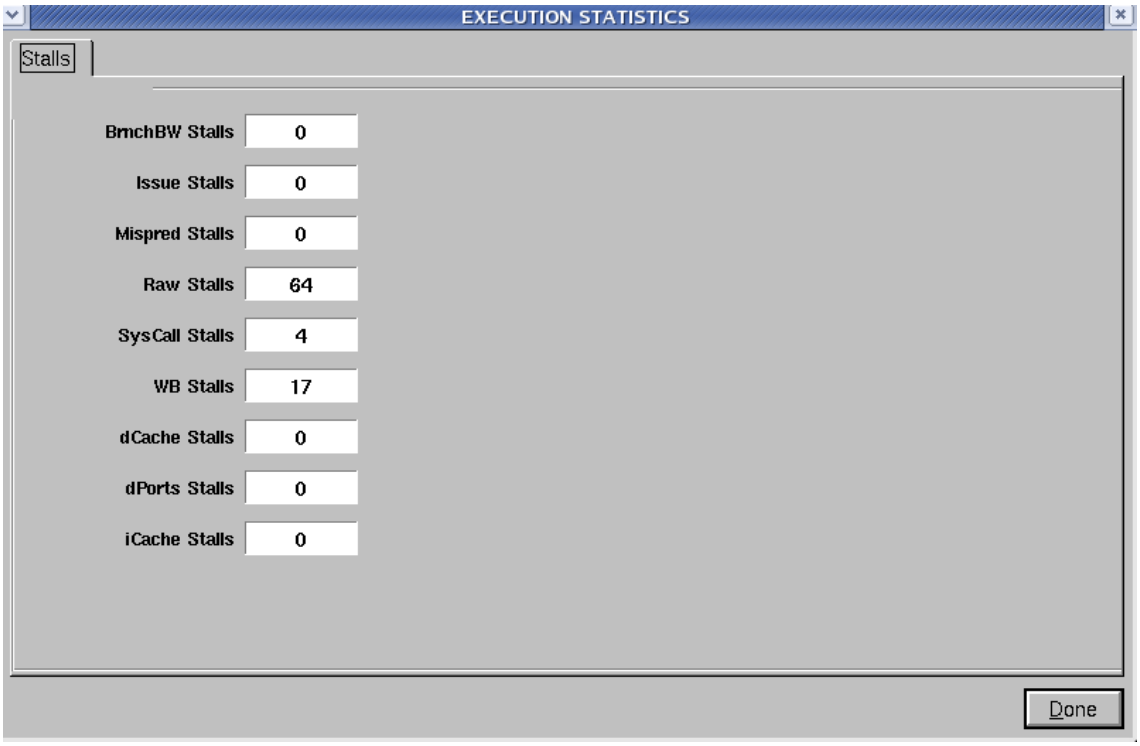
1. ¿Qué porcentaje de pérdida de rendimiento se produce?

$$Rendimiento_X = \frac{1}{Ciclos} = \frac{1}{176} \approx 0.0057$$

$$Rendimiento_Y = \frac{1}{Ciclos} = \frac{1}{241} \approx 0.0041$$

$$\Delta Rendimiento = \frac{Rendimiento_Y - Rendimiento_X}{Rendimiento_X} = \frac{-0.0016}{0.0057} \approx -0.28$$

Como podemos comprobar tenemos una pérdida de rendimiento de un 28 %. Es decir, no se produce ningún cambio significativo respecto a la configuración del ejercicio anterior.



2. ¿Qué tipo de penalización aumenta más? Mirando con detalle el diagrama de ciclos de la ejecución completa, identificar todas las parejas de instrucciones que generan una penalización y el número de ciclos de penalización que producen.

Las penalizaciones "WB Stalls", que ocurren por la tardía escritura en los registros, son los que más han aumentado, aumentando en 60. Las parejas que generan penalizaciones son las siguientes:

lds f02, 0(r03) -> muls f01, f02, f03: genera 2 ciclos de penalización

muls f01, f02, f03 -> adds f00, f03, f00: genera 2 ciclos de penalización

addqi r02, 4, r02 -> subqi r01, 1, r01: genera un ciclo de penalización.

3. ¿Qué tipo de penalización nueva aparece? ¿A qué es debida? Abstenerse de recordar el problema de las dependencias WAW: el método que se describe en las etapas ID y WB evita que este tipo de problemas genere ningún ciclo de penalización. El nuevo tipo de penalización también empieza por WB, pero su causa es otra muy diferente. Podréis entender el problema si usáis la ayuda en línea.

El nuevo tipo de penalización es la WB Stall y aparece debido a la ejecución fuera de orden, es decir, debido a su escritura tras la ejecución de otras instrucciones posteriores.

Ejercicio 2

1.

vecprod.eio	Tiempo de Riesgo	Ciclo de Ejecución	Instrucción que genera el dato	Instrucción que usa el dato	Ciclos de penalización
Ideal con Anticipacion	81	241	lds f02, 0(r03)	muls f01, f02, f03	81
Ideal sin Anticipacion	151	311	lds f02, 0(r03) // muls f01, f02, f03 // subqi r01, 1, r01	muls f01, f02, f03 // adds f00, f03, f00 // beq r01, 10	151
Branch	81	279	lds f02, 0(r03) // muls f01, f02, f03	muls f01, f02, f03 // adds f00, f03, f00	123
BTB	81	249	muls f01, 02, 03	adds f00, f03, f00	93
SuperScalar	114	173	lds f01, 0(r02) // lds f02, 0(r03) // muls f01, f02, f03	muls f01, f02, f03 // adds f00, f03, f00	17
Dcache	81	256	lds f02, 0(r03) // muls f01, f02, f03	muls f01, f02, f03 // adds f00, f03, f00	100

2.

Tras ejecutar el programa vecprod.eio con las distintas configuraciones, se pueden justificar los ciclos de penalización dividiéndolos en pérdidas por riesgo de datos o por disponibilidad de las etapas de segmentación. Para mostrarlo, tomaremos como ejemplo la última configuración, “DCache”.

Con esta configuración, se ejecuta el programa en 256 ciclos, 100 más de los debidos. Esto se debe a que realizan 15 iteraciones, dentro de las cuales se pierden 6 ciclos por dependencias de datos (90 ciclos). Además, perdemos 10 ciclos por la búsqueda de datos en la memoria secundaria, sumando el total 100 ciclos de pérdida.

3.

vecprod.eio	Ciclos	CPI	Speed-Up
Ideal con Anticipacion	241	1,54	1
Ideal sin Anticipacion	311	1,99	0,77
Branch	279	1,78	0,87
BTB	249	1,6	0,97
SuperScalar	173	1,11	1,39
Dcache	256	1,64	0,94

4.

La configuración que mayor Speed-up genera es la Superescalar, ya que disponemos del doble de unidades funcionales, adquiriendo y finalizando el doble de instrucciones por ciclo. El procesador Superescalar no consigue llegar a una mejora de 2 debido a las dependencias de datos, que impiden la generación del doble de datos por ciclo.

5.

Las instrucciones que generan penalizaciones por el fallo en la predicción de salto son:

· sts f00, 0(r04)
· ret r31, (r26)

De los 119 ciclos de penalización, perdemos 38 ciclos por culpa de los fallos de predicción de salto, además, perdemos 5 ciclos por iteración (75), además de los 10 ciclos de pérdida finales.

6.

Se pierden sólo 7 ciclos de pérdida ya que se guardan los últimos saltos y sólo falla en la primera y en la última iteración.

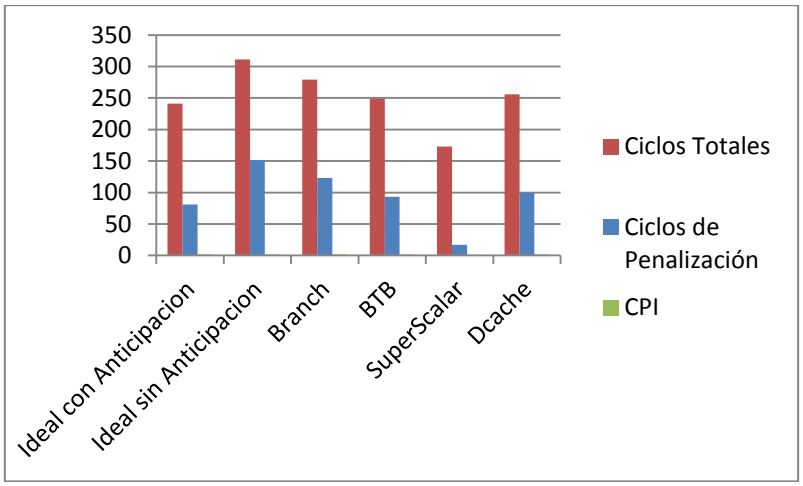
7.

Las instrucciones que producen penalizaciones por fallo de acceso a la memoria caché L1 de datos son:

· lds f02, 0(r03)
· sts f00, 0(r04)

Cada una de estas instrucciones penaliza con 5 ciclos (10), además de perder 6 ciclos por iteración (90), sumando ambos 100 ciclos de penalización.

8.



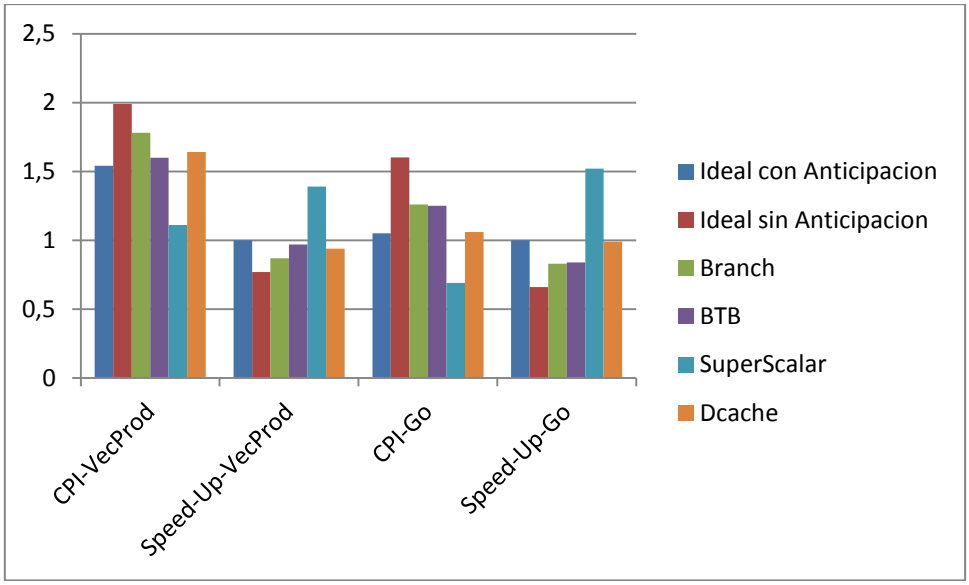
8

vecprod.eio	Ciclos	Ciclos de penalizacion
Ideal con Anticipacion	241	1,54
Ideal sin Anticipacion	311	1,99
Branch	279	1,78
BTB	249	1,6
SuperScalar	173	1,11
Dcache	256	1,64

9.

go.eio	Ciclos	CPI	Speed-Up
Ideal con Anticipacion	2,4E+07	1,05	1
Ideal sin Anticipacion	3,7E+07	1,601	0,66
Branch	2,5E+07	1,26	0,83
BTB	2,8E+07	1,25	0,84
SuperScalar	1,5E+07	0,69	1,52
Dcache	2,7E+07	1,06	0,99

10.



11.

gcc.eio	Ciclos	CPI-GCC	Speed-Up-GCC
Ideal con Anticipacion	2,5E+07	1,12	1
Ideal sin Anticipacion	3,7E+07	1,84	0,61
Branch	2,8E+07	1,36	0,82
BTB	3E+07	1,34	1,84
SuperScalar	1,7E+07	0,82	1,67
Dcache	3E+07	1,14	0,98

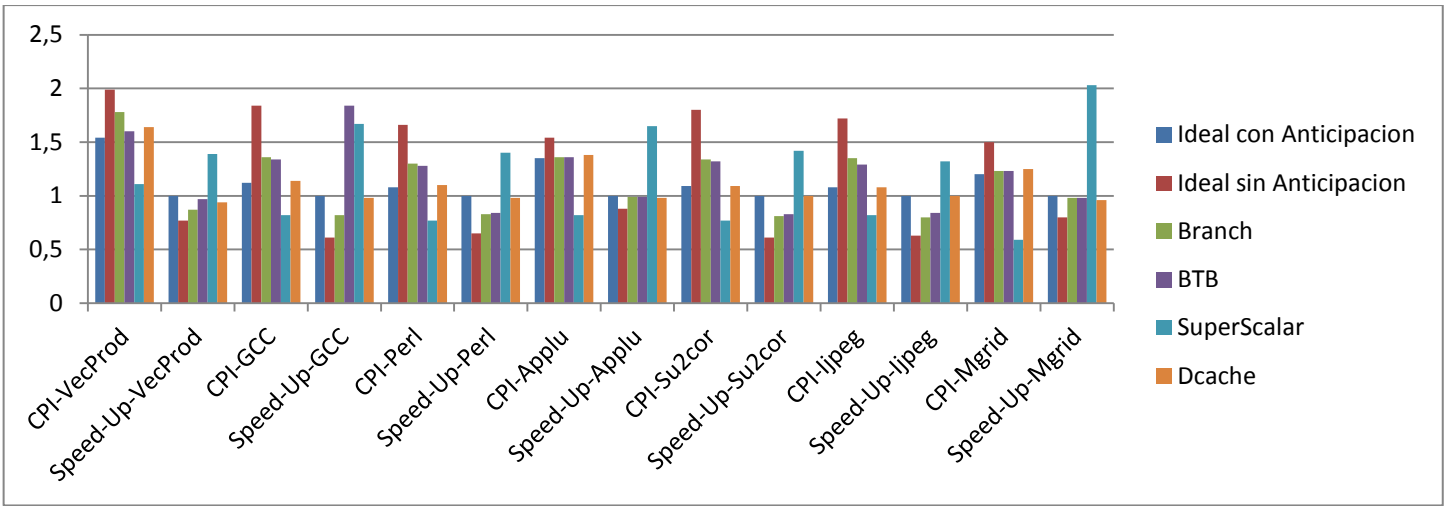
perl.eio	Ciclos	CPI-Perl	Speed-Up-Perl
Ideal con Anticipacion	2,7E+07	1,08	1
Ideal sin Anticipacion	3,5E+07	1,66	0,65
Branch	2,8E+07	1,3	0,83
BTB	2,8E+07	1,28	0,84
SuperScalar	1,6E+07	0,77	1,4
Dcache	2,5E+07	1,1	0,98

applu.eio	Ciclos	CPI-Applu	Speed-Up-Applu
Ideal con Anticipacion	2,9E+07	1,35	1
Ideal sin Anticipacion	3,4E+07	1,54	0,88
Branch	2,8E+07	1,36	0,99
BTB	2,8E+07	1,36	0,99
SuperScalar	1,7E+07	0,82	1,65
Dcache	2,8E+07	1,38	0,98

su2cor.eio	Ciclos	CPI-Su2cor	Speed-Up-Su2cor
Ideal con Anticipacion	2,7E+07	1,09	1
Ideal sin Anticipacion	3,7E+07	1,8	0,61
Branch	2,8E+07	1,34	0,81
BTB	2,8E+07	1,32	0,83
SuperScalar	1,6E+07	0,77	1,42
Dcache	2,5E+07	1,09	1

ijpeg.eio	Ciclos	CPI-Ijpeg	Speed-Up-Ijpeg
Ideal con Anticipacion	2,7E+07	1,08	1
Ideal sin Anticipacion	3,7E+07	1,72	0,63
Branch	2,8E+07	1,35	0,8
BTB	2,8E+07	1,29	0,84
SuperScalar	1,7E+07	0,82	1,32
Dcache	2,5E+07	1,08	1

mgrid.eio	Ciclos	CPI-Mgrid	Speed-Up-Mgrid
Ideal con Anticipacion	2,7E+07	1,2	1
Ideal sin Anticipacion	3,7E+07	1,5	0,8
Branch	2,8E+07	1,23	0,98
BTB	2,8E+07	1,23	0,98
SuperScalar	1,5E+07	0,59	2,03
Dcache	2,6E+07	1,25	0,96



Ejercicio 3

1.

El programa se ejecuta en 170 ciclos (156 instrucciones)

170-(7-1) -156 = 8 ciclos de penalización

adds f31, f31, f0 instrucción p. flotante (EX 4 ciclos) 3 penalizaciones
adds f00, f03, f00 3 penalizaciones IQ (RAW)
adds f0, f03, f0 1 penalización IQ (RAW)
adds f0, f03, f0 1 penalización IQ (RAW)

2. Ganancia

En orden tarda 141 ciclos, por lo que hay una ganancia de 1,42 con respecto a la ejecución fuera de orden.

3.

	InOrder	OutofOrder
matprod-reorder	47963 ciclos (39522 instr.)	39521 ciclos (39511 instr.)
vectprod-reorder	193 ciclos (156 instr.)	166 ciclos (156 instr.)

Al usar la ejecución fuera de orden en un código reordenado se aprovecha la falta de penalizaciones entre instrucciones para aumentar el número de instrucciones "independientes".

matprod ganancia -> 0.82
vectprod ganancia -> 0.86

Las ganancias en ambos programas (fuera-de-orden/en-orden) son muy similares, por lo que podemos argumentar que son casi igual de útiles.

4.

	InOrder	OutofOrder
matprod-unroll	33883 ciclos (27223 instr.)	27233 ciclos (27223 instr.)
vectprod-unroll	138 ciclos (108 instr.)	118 ciclos (108 instr.)

Se observa que en ambos programas, al emplear el desenrrollamiento de bucles se ha reducido el tiempo de ejecución. Con respecto al CPI se observa que este aumenta cuando aplicamos esta técnica.

5.

	IQ size =1	IQ size =2
matprod-unroll	29789 ciclos (27223 instr.)	27223 ciclos (27223 instr.)
vectprod-unroll	124 ciclos (108 instr.)	118 ciclos (108 instr.)

6.

Predicción de salto perfecto

muls f01, f02, f03 EXEC (3 penalizaciones)
adds f00, f03, f00 ID->RAW x3
subqi r01, 1, r01 ID -> WB hazard

RAW Stalls: 64
Syscall Stalls: 4
WB Stats: 17

Salto no tomado (inroderBpred.cnf):

Mispred Stalls: 38
RAW Stalls: 64
Syscall Stalls: 4
WB Stats: 17

En cada iteración del bucle, al instrucción 9 genera 2 ciclos de penalización, que con la configuración inOrderBpred son inevitables ya que hasta que no se resuelve el salto en la etapa de ejecución se siguen introduciendo instrucciones, aunque sea un salto incondicional, ya que así lo especifica la configuración

7.

7.1. La primera vez que se ejecuta la instrucción 9 se produce una penalización ya que la BTB no tiene información sobre este salto, por ello, la primera vez falla, mientras que el resto de veces acertará salvo que se trate de la última iteración.

7.2. No se produce penalización ya que al activar la opción RAS Active (Return Adress Stack) se crea una pila de 2 entradas donde almacenar las direcciones de retorno al realizar una llamada.

7.3. La penalización que más se produce es la instrucción 9 (el salto) ya que a pesar de tratarse de un salto incondicional, la configuración OutOrderBpred.cnf siempre predice el salto como no tomado. A su vez, antes de entrar en el bucle se produce una penalización al realizar otro salto incondicional (instrucción 18). Al salir del bucle tenemos otra penalización generada por la instrucción 1, que se trata de un salto condicional que debería haberse tomado.

8.

8.1.

Instrucción 13: 5 ciclos
Instrucción 16: 5 ciclos
Instrucción 0: 5 ciclos
Instrucciones 2 y 3: 10 ciclos en la primera iteración del bucle.
Instrucciones 2 y 3: 10 ciclos en la novena iteración del bucle.
Instrucción 10: 5 ciclos, después de terminar el bucle.

Lo que suma un total de **40 ciclos de penalización** por fallos en caché de datos e instrucciones, lo que se confirma con la ejecución del comando de estadísticas:

iCaché hits : 168
cache miss rate 1.785714
icache misses: 3
cache hits: 28
dcache miss rate: 15.151515
dcache misses: 5

8.2. Todas las penalizaciones son de 5 ciclos porque es la latencia de la cache L2. Esto es así porque la cache está definida como una cache perfecta, si esto no fuera así, el número de ciclos de penalización se multiplicaría. Al aumentar el tamaño de línea, el número de bloques que se traen en cada acceso a cache, tardando más en hacer el próximo acceso a la memoria.

8.3. El problema es mayor en un procesador con ejecución en orden, ya que al producirse un fallo se bloquea el cauce por completo.

8.4. Se podría intentar localizar los datos de los vectores de forma que estén juntos a la hora de “traerlos” de la memoria principal, reduciendo el número de fallos en la caché de datos.

Ejercicio 4

1.

InOrder2 -> 110 ciclos -> 108 instrucciones
InOrder -> 138 ciclos -> 108 instrucciones
Ganancia: 138/110 = 1,25

2.

Si suponemos que no hay ningún tipo de penalización, entrarían dos instrucciones por ciclo, por lo que se ejecutaría el problema en tantos ciclos como la mitad de instrucciones mas los ciclos de llenado de pipeline:

108/2+4 = 58 ciclos

Para ver el número de ciclos de penalización que hemos tenido, restamos los ciclos totales con los ciclos iniciales:

110-108/2-4 = 52 ciclos

3.

CPI = CICLOS/NINSTR = 110/108 = 1.019
CPI_ideal = CICLOS_ideal/NINSTR = 58/108 = 0.54

4.

adds f31, f31, f00: 1 ciclo (llenado de pila)

lds f01, 0(r02), lds f02, 0(r03): 1 ciclo
lds f02, 0(r03), lfs f05, 4(r02): 1 ciclos
lds f02, 0(r03), muls f01, f02, f03: 1 ciclo
lfs f06, 4(r03), muls f05, f06, f07: 1 ciclo
muls f01, f02, f03, adds f00, f03, f00: 2 ciclos
adds f00, f03, f00, adds f00, f07, f00: 3 ciclos

Todos ellos se repiten 8 veces (8 iteraciones cada uno).

Al final, también tienen penalización en la etapa ID:
adds f00, f07, f00, sts f00, 0(r04): 3 ciclos

el total de ciclos de penalización es:
9*8 + 3 = 75
75+58 = 133

Si bien podemos ver claramente que los ciclos no coinciden con el número de ciclos que ha tardado, hemos revisado analizado el pipeline y la ejecución del programa y no conseguimos ver cómo podemos conseguir que éstos cuadren. Es posible que esto se deba a que, si bien no somos capaces de verlo claramente, es posible que algunas penalizaciones se vean compensadas gracias a su ejecución en paralelo, es decir, a la ejecución simultánea de dos instrucciones.

5.

Con la configuración monoescalar (InOrder.cnf), la instrucción 16 no penaliza, ya que tiene activado el adelantamiento de datos y se ejecutan como instrucciones independientes, es decir, sin paralelismo; mientras que con la configuración superescalar (InOrder2.cnf) estas dos instrucciones son lanzadas a ejecutar de manera simultánea, por lo que la instrucción 16 ha de esperar a que se resuelva el valor de r02.

6.

	InOrder4.cnf	InOrder8.cnf
Vectprod-unroll.eio	102 ciclos/0.94	101 ciclos/0.94
Matprod-unroll.eio	24103 ciclos/0.88	24087 ciclos/0.88

7.

OutOrder2 -> 90 ciclos -> 108 instrucciones
OutOrder -> 118 ciclos -> 108 instrucciones

Ganancia = 118/90 = 1,31

8.

adds f31, f31, f00 -> 1 ciclo
lfs f01, 0(r02) -> 1 ciclo
lfs f02, 0(f03) -> 1 ciclo
lfs f05, 4(r02) -> 1 ciclo
muls f01, f02, f03 -> 2 ciclos
lds f06, 4(r03) -> 1 ciclo
subqi r01, 2, r01 -> 2 ciclos
muls f05, f06, f07 -> 2 ciclos
adds f00, f03, f00 -> 3 ciclos
addqi r02, 8, r02 -> 1 ciclo
addqi r03, 8, r03 -> 1 ciclo
adds f00, f07, f00 -> 5 ciclos
bne r01, 2 -> 1 ciclo

22 ciclos *8 = 176

sts f00, 0(r04) -> 7 ciclos
ret r31, (r26) -> 3 ciclos
addqi r31, 1, r00 -> 1 ciclo
addqi r02, 64, r03 -> 1 ciclo
addqi r03, 64, r04 -> 1 ciclo
bsr r26, 0 -> 1 ciclo

Total de ciclos de penalización: 176+14 = 190 ciclos

Como se puede ver fácilmente, esto no es viable para el número de ciclos en los que se ejecuta el programa (90 ciclos). Esto se debe a que, en un ciclo, no sólo puede finalizar una instrucción, si no que pueden terminar varias instrucciones, llegando a salir 4 instrucciones a la vez.

9.

	OutOrder4.cnf	OutOrder8.cnf
Vecprod-unroll.eio	81 ciclos/0.75	80 ciclos /0.74

	OutOrder2.cnf	OutOrder4.cnf	OutOrder8.cnf
Matprod-unroll.eio	16963 ciclos/0.62	16943 ciclos/0.62	12860 ciclos/0.47

10.

El nuevo cuello de botella está en el tamaño de IQ, es decir, las instrucciones van llenando la tabla hasta el punto de llenarla y, por muchas instrucciones que se lancen en paralelo, el tamaño no cambiará (respecto al inicial), por lo que se formará un cuello de botella.

11.

	InOrder.cnf	OutOrder.cnf	OutOrder8.cnf
vectprod-unroll	138 ciclos/1.28	118 ciclos/1.09	80 ciclos/0.74
vectprod-par	154 ciclos/1.43	122 ciclos /1.13	52 ciclos/0.48
matprod-unroll	33883 ciclos/1.24	27233 ciclos/1	12860 ciclos/0.47
matprod-par	37979 ciclos/1.39	27237 ciclos/1	8498 ciclos/0.31

12.

Se puede comprobar que, exceptuando en el caso de OutOrder8, el código optimizado se ejecuta en un número mayor de ciclos.

13.

Se ha añadido una instrucción que suma el valor de f03 y el valor de f07 y lo guarda en f07, sustituyendo una suma similar, pero con el valor de f03 y el de f0. Esto permite trabajar con menos registros (los accesos a f0 son menores). Para hacer esto, se ha utilizado la propiedad asociativa de la suma (a+(b+c) = (a+c)+b).

14.

a)

InOrder -> 1804 ciclos -> 1 ciclo de penalización por iteración

cmpult r03, r00, r04

OutOrder8-> 521 ciclos (ideal 193) -> 10 ciclos por iteración, pero dado que en cada ciclo entran 8 instrucciones, esto se ve paliado

ldl r00, 0(r02) -> 1 ciclo
ldl r03, 0(r02) -> 1 ciclo
cmpult r03, r00, r04 -> 3 ciclos
cmovne r04, r03, r00 -> 4 ciclos
bgt r01, 1 -> 1 ciclo

b)

En la versión InOrder, se podría mejorar su ejecución mediante el desenrrollamiento simbólico de bucles, es decir, partiendo de un código prólogo, en cada iteración se almacene el valor calculado en la iteración anterior, se calcula el

nuevo valor a partir de los valores leídos anteriormente y se carga el valor de la siguiente iteración. De esta manera, se reducen las dependencias RAW.

En la versión OutOrder, se podría intentar llevar a cabo un reordenamiento de instrucciones, buscando la reducción de dependencias de las instrucciones de salto.

c)

Si presuponemos que los cálculos han de ser similares a los ejemplos dados en clase y en el libro de la asignatura, el Speed-Up en la primera versión sería de 1.8 veces, por lo que el número de ciclos sería, aproximadamente, 1003 ciclos.

Si presuponemos que hayamos conseguido que no se ejecuten instrucciones posteriores al salto antes de haber calculado su resolución, se ejecutaría en 265 ciclos, con un Speed-Up de 1.97.

d)

Es importante conocer el procesador sobre el que se va a ejecutar el programa, ya que debemos conocer el impacto de las penalizaciones sobre la ejecución y las medidas que toma el procesador para evitarlas. Además, a la hora de optimizar el código, se ha de tener en cuenta la arquitectura del procesador para explotar el paralelismo de instrucciones, siempre y cuando se posea esta característica.