Cognoms: . . . . Nom: . . . .

2on Control Arquitectura de Computadors **Curs 2011-2012 Q2**

Problema 1. (2 puntos)

Los dispositivos que no disponen de hardware específico de coma flotante deben realizar las operaciones de coma flotante por software. La siguiente función escrita en C suma a su parámetro res todos los elementos del parámetro vectorfloats mediante llamadas a la rutina sumfloat que como su nombre indica realizan por software la suma en coma flotante de sus dos parámetros dejando el resultado en el primer parámetro:

```
void sumavectorescalar(float *res, float vectorfloats[], int N)
{
    int i;

    for (i=0;i<N; i++)
       sumfloat(res, vectorfloats[i]);
}
```

a) Dibujad el bloque de activación de la rutina sumavectorescalar y traducidla a ensamblador x86. (1 punto)

El código anterior se ejecuta en un procesador que funciona a una frecuencia de 1 GHz y tarda 7 s cuando N vale 10^8. Sabemos que la rutina sumfloat contiene 40 instrucciones estáticas que en media generan 55 instrucciones dinámicas en cada llamada a la misma.

b) Calculad cuantas instrucciones se ejecutan en cada iteración del bucle de sumavectorescalar, el CPI al que se ejecuta el programa y los MIPS y MFLOPS del sistema. (0,5 puntos)

Si añadimos al procesador anterior una unidad aritmética de coma flotante podríamos substituir la llamada a la rutina sumfloat por la operación en alto nivel: "res+=vectorfloats[i];". En este caso el bucle anterior tan solo tardaría 0,5 s en ejecutarse pero el procesador incrementaría su gasto en potencia media en un 10%.

Calculad cuál sería el porcentaje de ganancia en MFLOPS/W respecto al caso anterior. (0,5 puntos)

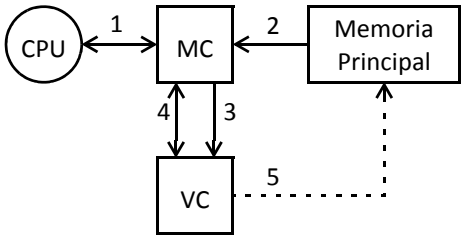
Cognoms: . . . . . Nom: . . . . . .

2on Control Arquitectura de Computadors **Curs 2011-2012 Q2**

Problema 2. (4 puntos)

Una Victim Cache (VC) es una cache totalmente asociativa, muy pequeña (p.e. 4-6 bloques) que trabaja en paralelo con la Memoria Cache (MC), que normalmente es directa. La VC almacena bloques que han sido expulsados de la MC. Aunque la VC es completamente asociativa, su tiempo de acceso es muy similar o inferior al de la MC porque es muy pequeña.

La figura ilustra el comportamiento de un sistema de memoria con una VC. Cuando se produce un acceso a memoria se pueden producir 3 situaciones:



- **Acierto en MC**. Se realiza el acceso en MC (1).
- **Fallo en MC y Fallo en VC**. Se trae el bloque correspondiente de Memoria Principal a MC (2) y el bloque expulsado de MC se almacena en la VC (3). Se realiza el acceso en MC (1).
- **Fallo en MC y Acierto en VC**. El bloque pedido está en la VC. El bloque expulsado de MC se intercambia con el bloque solicitado de la VC (4). Se realiza el acceso en MC (1).

Por último hay que considerar los bloques modificados en una cache copy back. Dado que cuando se expulsa un bloque de MC no desaparece del sistema, sino que se copia en la VC, sólo será necesario actualizar la Memoria Principal, cuando un bloque modificado sea expulsado definitivamente de la VC (5).

Vamos a considerar dos implementaciones de la memoria cache de datos:

- Una memoria cache directa con 16 bloques de tamaño 16B, con copy back + write allocate.
- Una memoria cache directa con 16 bloques de tamaño 16B + una victim cache con reemplazo FIFO de 3 bloques de capacidad, con copy back + write allocate.

Sabemos que el contenido inicial de la cache (sólo se muestran los tags) es el siguiente (el asterístico indica que ese bloque ha sido modificado previamente):

BLOQUE	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**A**	**B**	**C**	**D**	**E**	**F**
TAG	FF	FF	AC*	00	00	EC	00	00	80*	AA	AA	04*	20	11	20	12*

a) Considerando exclusivamente la MC directa, **rellena** la siguiente tabla indicando para cada acceso: TAG, bloque de MC, acierto o fallo, bytes leídos de MP, bytes escritos en MP y tag del bloque reemplazado.

dirección (hex)	**TAG**	**bloque MC**	**¿acierto o fallo?**	**bytes leídos de MP**	**bytes escritos en MP**	**TAG del bloque reemplazado**
ESCR B220						
LECT FF51						
LECT AC22						
LECT FF53						
ESCR B224						
LECT 11D5						
LECT AC56						
ESCR B227						
LECT AC28						
LECT EC59						

Sabemos que el contenido inicial de la victim cache (nótese que se muestran el número de bloque de MP) es el siguiente (el asterístico indica que ese bloque ha sido modificado previamente):

1E4	20A*	12E

El bloque 1E4 es el que lleva más tiempo en la VC, el bloque 12E es el que lleva menos tiempo en la VC

b) Conociendo el contenido inicial de la VC y partiendo del mismo contenido inicial de la MC del anterior apartado, **rellena** la siguiente tabla indicando para cada acceso: TAG, bloque de MC, acierto en MC o acierto en VC o fallo en las 2, bytes leídos de MP, bytes escritos en MP y el número de bloque de MP expulsado de la VC.

dirección (hex)	TAG	bloque MC	¿acierto MC o acierto VC o fallo?	bytes leídos de MP	bytes escritos en MP	#bloque de MP expulsado de la VC
ESCR B220						
LECT FF51						
LECT AC22						
LECT FF53						
ESCR B224						
LECT 11D5						
LECT AC56						
ESCR B227						
LECT AC28						
LECT EC59						

Para evaluar estas propuestas hemos utilizado un conjunto de programas de test en el que se han ejecutado $100 \cdot 10^9$ instrucciones en un procesador ideal (todos los accesos a memoria tardan un ciclo), con un coste de $140 \cdot 10^9$ ciclos. Se han realizado $25 \cdot 10^9$ accesos a memoria de datos, de los cuales un 10% son fallos en la cache directa.

c) Sabiendo que el tiempo de ejecución real es de 80s y que el procesador funciona a 3GHz, **calcula** el coste de penalización por fallo.

d) Sabiendo que la tasa de fallos del sistema de cache directa + victim cache tiene una tasa de fallos del 2,5%, **calcula** el tiempo de ejecución del conjunto de programas de test.

Sabiendo que cada fallo de cache consume 200 nJ, que un acceso a la cache directa consume 5 nJ y que un acceso al sistema de cache directa + victim cache consume 15nJ.

e) **Calcula** la energía consumida por cada una de las implementaciones del sistema de memoria.

f) **Calcula** la potencia media de cada una de las implementaciones del sistema de memoria.

Cognoms: . . . . . Nom: . . . . . .

2on Control Arquitectura de Computadors **Curs 2011-2012 Q2**

Problema 3. (4 puntos)

Dado el siguiente código en C:

```
int V[80000];
...
int s=0;
for (i=0; i<80000; i=i+8)
  s = s + V[i];
```

El compilador ha generado el siguiente código ensamblador:

```
        /* %edi contiene la dirección base de V */
        movl $0, %ecx              /* s */
        movl $0, %esi              /* i */
loop:   movl (%edi,%esi, 4), %eax  /* load V[i] */
        addl $8, %esi
        addl %eax, %ecx
        cmpl $80000, %esi
        jl   loop
```

Dicho código se ejecuta en un procesador que tiene una cache de datos (L1) con bloques de 32 bytes (ignoraremos los accesos a instrucciones), cuyos fallos son servidos por un segundo nivel de cache (L2). Para simplificar el problema asumimos que siempre se produce acierto en L2. Sabemos además que la dirección del vector V está alineada con el inicio de un bloque de memoria.

a) **Calcula** cuantos fallos en la cache de datos producirá dicho código suponiendo que esta está inicialmente vacia.

En dicho procesador todas las instrucciones tardan un ciclo excepto si se produce un fallo en la cache de datos. En caso de fallo en la cache de datos, la instrucción requiere **6 ciclos adicionales** de ejecución. Este procesador soporta l**oads no bloqueantes** (la cache L1 es no bloqueante) por lo que, aunque una instrucción tenga un fallo de cache, el procesador sigue ejecutando instrucciones hasta que una de ellas requiera el dato producido por el load. Suponemos que el procesador tiene un número suficiente de MSHRs para almacenar los fallos pendientes necesarios. En el código anterior hemos mostrado en negrita el registro producido por el load y la instrucción que lo consume. La siguiente tabla muestra el cronograma de la ejecución de las instrucciones de la primera iteración del bucle:

Ciclo	01	02	03	04	05	06	07	08	09	10	11	12	13	14
movl (%edi,%esi, 4), **%eax**	X	F	F	F	F	F	F							
addl $8, %esi		X												
addl **%eax**, %ecx			-	-	-	-	-	X						
cmpl $80000, %esi									X					
jl loop										X				

En dicho cronograma se indica la ejecución de las instrucciones con una “X”, los ciclos esperando por un fallo de la cache de datos con una “F” y los ciclos en que una instrucción espera que un operando esté disponible con “-”. El dato del load no se escribe en %eax hasta el ciclo 7, por lo que la instrucción que lo usa debe esperar hasta el ciclo 8. Como podemos ver, los loads no bloqueantes no son muy efectivos en éste código, dado que solo hemos podido esconder uno de los ciclos de penalización debido al fallo.

b) **Calcula** el CPI (ciclos por instrucción) del bucle anterior y el número total de ciclos que tardaríamos en ejecutar el bucle.

Un programador experimentado ha mejorado el código para que los loads no bloqueantes sean más efectivos:

```
         movl $0, %ecx              /* s */
         movl $0, %esi              /* i */
         movl (%edi,%esi, 4), %eax  /* load V[0] */
loop:    addl %eax, %ecx
         movl 32(%edi,%esi, 4), %eax /* load V[i+8] */
         addl $8, %esi
         cmpl $80000, %esi
         jl   loop
```

En este código, el primer load se realiza antes de entrar en el bucle. Una vez dentro del bucle, en cada iteración se realiza el load que será usado por la siguiente iteración del bucle, por lo que ahora, entre el load y la instrucción que consume el dato se ejecutan otras 3 instrucciones independientes.

c) **Rellena** la siguiente tabla con el cronograma correspondiente a la ejecución de las 3 primeras iteraciones del bucle. Suponed que V[0] ya se encuentra en el registro %eax al iniciar la primera iteración del bucle.

Ciclo	01	02	03	04	05	06	07	08	09	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30
addl **%eax**, %ecx																														
movl 32(%edi,%esi, 4), **%eax**																														
addl $8, %esi																														
cmpl $80000, %esi																														
jl loop																														
addl **%eax**, %ecx																														
movl 32(%edi,%esi, 4), **%eax**																														
addl $8, %esi																														
cmpl $80000, %esi																														
jl loop																														
addl **%eax**, %ecx																														
movl 32(%edi,%esi, 4), **%eax**																														
addl $8, %esi																														
cmpl $80000, %esi																														
jl loop																														

d) **Calcula** el CPI (ciclos por instrucción) del bucle anterior y el speed-up del bucle respecto el bucle original (compilador).

En un intento por mejorar aún más el rendimiento, nuestro experto programador ha realizado una versión del mismo en que se ha desenrollado el bucle, realizando dos iteraciones del bucle en alto nivel por cada iteración en ensamblador:

```
         movl $0, %ecx               /* s */
         movl $0, %esi               /* i */
         movl (%edi,%esi, 4), %eax   /* load V[0] */
         movl 32(%edi,%esi, 4), %ebx /* load V[8] */
loop:    addl %eax, %ecx
         movl 64(%edi,%esi, 4), %eax /* load V[i+16] */
         addl %ebx, %ecx
         movl 96(%edi,%esi, 4), %ebx /* load V[i+24] */
         addl $16, %esi
         cmpl $80000, %esi
         jl   loop
```

Cognoms: . . . . . Nom: . . . . . .

2on Control Arquitectura de Computadors **Curs 2011-2012 Q2**

Problema 3 (cont)

Como se puede observar, en el bucle se realiza un nuevo load antes de que se ejecute la instrucción que consume el resultado del primer load, por lo que es posible que haya un segundo load pendiente antes de que haya acabado el anterior. Dado que la jerarquía de memoria solo sirve un fallo a la vez, el servicio del segundo fallo se bloqueará hasta que se haya completado el anterior. La siguiente tabla muestra un ejemplo de cronograma con 2 loads seguidos que provocan fallo de cache, seguidos de una instrucción independiente.

Ciclo	01	02	03	04	05	06	07	08	09	10	11	12	13	14	15	16	17	18	19	20
movl (%edi,%esi, 4), **%eax**	X	F	F	F	F	F	F													
movl 64(%edi,%esi, 4), **%ebx**		X	..	..	..	..	..	F	F	F	F	F	F							
cmpl $80000, %esi			X																	

En dicho cronograma se indica la ejecución de las instrucciones con una "X", los ciclos esperando por un fallo de la cache de datos con una "F" y los ciclos en que un fallo espera a que se haya servido otro fallo anterior con "..".

e) **Rellena** la siguiente tabla con el cronograma correspondiente a la ejecución de las 3 primeras iteraciones del bucle ensamblador (6 primeras del código C). Suponed que V[0] y V[8] ya se encuentran en los registros %eax y %ebx respectivamente al iniciar la primera iteración del bucle. Por razones de espacio (`%edi,%esi, 4`) se ha abreviado como (`. .`).

Ciclo	01	02	03	04	05	06	07	08	09	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32	33	34	35	36	37	38	39	40
addl **%eax**, %ecx																																								
movl 64(...),**%eax**																																								
addl **%ebx**, %ecx																																								
movl 96(...),**%ebx**																																								
addl $16, %esi																																								
cmpl $80000,%esi																																								
jl loop																																								
addl **%eax**, %ecx																																								
movl 64(...),**%eax**																																								
addl **%ebx**, %ecx																																								
movl 96(...),**%ebx**																																								
addl $16, %esi																																								
cmpl $80000, %esi																																								
jl loop																																								
addl **%eax**, %ecx																																								
movl 64(...),**%eax**																																								
addl **%ebx**, %ecx																																								
movl 96(...),**%ebx**																																								
addl $16, %esi																																								
cmpl $80000, %esi																																								
jl loop																																								

f) **Calcula** el CPI (ciclos por instrucción) del bucle anterior y el speed-up del bucle respecto el bucle original (compilador).

El tiempo de servir los fallos se ha convertido en el cuello de botella de este código, ya que no podemos solapar el servicio de un fallo con el servicio del siguiente. Para ello, un arquitecto de computadores ha propuesto una organización del segundo nivel de cache organizada en 2 bancos. En esta organización, los bancos de la cache L2 están entrelazados a nivel de bloque, de forma que bloques de memoria consecutivos se almacenan en bancos distintos de L2. El bus entre L1 y L2 tiene el ancho de un bloque de L1 y un bloque se transfiere durante el último ciclo, por lo que de los 6 ciclos que dura el servicio de un fallo de L1, el bus está libre durante los cinco primeros.

g) **Rellena** la siguiente tabla con el cronograma correspondiente a la ejecución de las 3 primeras iteraciones del bucle ensamblador (6 primeras del código C) ejecutado con una L2 organizada en bancos. Suponed que V[0] y V[8] ya se encuentran en los registros %eax y %ebx respectivamente al iniciar la primera iteración del bucle. Por razones de espacio `(%edi,%esi, 4)` se ha abreviado como `(...)`.

Ciclo	01	02	03	04	05	06	07	08	09	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32	33	34	35	36
addl **%eax**, %ecx																																				
movl 64(...),**%eax**																																				
addl **%ebx**, %ecx																																				
movl 96(...),**%ebx**																																				
addl $16, %esi																																				
cmpl $80000,%esi																																				
jl loop																																				
addl **%eax**, %ecx																																				
movl 64(...),**%eax**																																				
addl **%ebx**, %ecx																																				
movl 96(...),**%ebx**																																				
addl $16, %esi																																				
cmpl $80000, %esi																																				
jl loop																																				
addl **%eax**, %ecx																																				
movl 64(...),**%eax**																																				
addl **%ebx**, %ecx																																				
movl 96(...),**%ebx**																																				
addl $16, %esi																																				
cmpl $80000, %esi																																				
jl loop																																				

h) **Calcular** el CPI (ciclos por instrucción) del bucle anterior y el speed-up respecto el bucle original (compilador).