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

2on Control Arquitectura de Computadors **Curs 2011-2012 Q1**

Problema 1. (3 puntos)

Dada la siguiente función escrita en C:

```
int F(int *p, int x) {
  int aux;
  aux = *p;
  *p = x;
  return aux >> 3;
}
```

a) **Dibuja** el bloque de activación de la rutina F y traducidla a ensamblador x86.

Dado el siguiente código en ensamblador:

```
      movl $V,%ebx
      xorl %esi,%esi
for:  cmpl $2000000000,%esi
      jge fin
      pushl %esi
      pushl %ebx
      call S
      addl %8,%esp
      addl $58, $eax
      addl %eax, (%ebx,%esi,4)
      incl %esi
      jmp for
end:
```

```
S:  pushl %ebp
    movl %esp,%ebp
    movl 8(%ebp),%edx
    movl 12(%ebp),%ecx
    movl (%edx,%ecx,4),%eax
    andl $0x0F,%ecx
    imul %ecx,%eax
    movl %ebp,%esp
    popl %ebp
    ret
```

Este código, que realiza $2 \cdot 10^9$ iteraciones, se ha lanzado en un procesador que funciona a 2 GHz y ha tardado 15 segundos en ejecutarse. Para todo el resto del problema, las instrucciones de inicialización y de final de bucle pueden despreciarse.

b) **Calcula** el número de instrucciones ejecutadas y el CPI de este código.

c) Si no consideramos los fallos en la cache de instrucciones, en la cache de datos se producen $250 \cdot 10^6$ fallos y cada fallo penaliza 10 ciclos, **calcula** el CPI ideal de este código.

d) Suponiendo que todas las instrucciones ocupan 4 bytes, **calcula** el ancho de banda con la cache de instrucciones.

e) **Calcula** el ancho de banda con la cache de datos. Para calcular cuántos accesos a memoria de datos se realizan en cada iteración, marcad con una cruz las instrucciones que acceden a memoria de datos en el código.

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

2on Control Arquitectura de Computadors **Curs 2011-2012 Q1**

Problema 2. (4 puntos)

Dado el siguiente código escrito en ensamblador del x86:

```
      movl $0, %ebx
      movl $0, %esi
for:
      cmpl $512*1000, %esi
      jge end

 (a)  movl (%ebx, %esi, 4), %eax
 (b)  addl %eax, 8*1024(%ebx, %esi, 4) // Dos accesos a memoria
 (c)  movl %eax, 16*1024(%ebx, %esi, 4)

      incl %esi
      jmp for
end:
```

Sabemos que se ejecuta en un sistema con memoria cache y memoria virtual. Sabemos que la memoria virtual utiliza páginas de tamaño 4KB y que disponemos de un TLB de 4 entradas (reemplazo LRU). Además sabemos que la memoria cache de datos (únicos accesos a memoria que contemplaremos en este problema) es Write Through + Write No Allocate, de 2 vías (reemplazo LRU), con 64 conjuntos y 16 bytes por bloque. Responde a las siguientes preguntas:

a) **Calcula** para cada uno de los accesos (etiquetas a, b, c) el conjunto y vía de la memoria cache al que se accede en cada una de las 9 primeras iteraciones del bucle.

Conjunto:

	0	1	2	3	4	5	6	7	8	9
a										
b										
c										

Vía (poner una X si no se puede determinar):

	0	1	2	3	4	5	6	7	8	9
a										
b										
c										

b) **Calcula** la cantidad de aciertos y de fallos de cache, en todo el código.

c) Para cada uno de los accesos indicados (etiquetas a, b, c), **indica** a qué página de la memoria virtual se accede en cada una de las siguientes iteraciones del bucle.

	0	1*512	2*512	3*512	4*512	5*512	6*512	7*512	8*512	9*512
a										
b										
c										

d) **Calcula** la cantidad de aciertos y de fallos de TLB, en todo el código..

El programa anterior tiene un CPI ideal de 2 funcionando en un procesador a 2 GHz. Sabemos que un fallo de cache tiene una penalización de 10 ciclos y un fallo de TLB (asumiremos que no hay fallos de página) tiene una penalización de 1000 ciclos.

e) **Calcula** el tiempo de ejecución del código anterior.

Sabemos que todas las direcciones de memoria son de 32 bits (tanto en las direcciones lógicas como en las físicas).

f) **Calcula** el tamaño en bits de la memoria de datos de la cache, el de la memoria de etiquetas de la cache y también el de todo el TLB (en ambos casos podéis prescindir de los bits de estado).

Sabemos que la corriente de fugas de la memoria RAM estática es de 3 uA (microAmperios) por bit y que está alimentada a 1,2 V. La energía consumida durante un acceso a la memoria de etiquetas es de 5 nJ (nanoJoules) por vía y la consumida durante un acceso a la memoria de datos es de 25 nJ por vía (en esta memoria siempre se accede en paralelo a ambas vías).

g) **Calcula** la potencia media estática (debida a fugas) de la cache.

h) **Calculad** la potencia media dinámica (debida aconmutación) consumida por la cache durante la ejecución del programa. Para simplificar asumiremos que todos los accesos consumen lo mismo sean acierto o fallo. La energía extra consumida al acceder a memoria principal en caso de fallo está fuera de los objetivos de este problema.

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

2on Control Arquitectura de Computadors **Curs 2011-2012 Q1**

Problema 3. (3 puntos)

Hemos simulado la ejecución de un programa (para simplificar el problema suponemos que solo realiza lecturas) en una CPU con un sistema de memoria en donde todos los accesos a memoria tardan 1 ciclo sean aciertos o fallos (denominaremos **CPU_{IDEAL}** a esta combinación simulada) y hemos obtenido que el programa se ejecuta en $16x10^6$ ciclos y realiza $5x10^6$ accesos a memoria.

Para funcionar a la frecuencia deseada, la cache de primer nivel (L1), que contiene instrucciones y datos, debe tener un tiempo de acceso de 3 ciclos. Supongamos de momento que nunca tenemos fallos en la L1 (a esta combinación la denominamos **CPU_1**).

a) **Indica** los ciclos de penalización de un acceso a memoria respecto a **CPU_{IDEAL}** y **calcula** los ciclos que tarda en ejecutarse el programa en la **CPU_1**.

La cache de primer nivel (L1) tiene, para nuestro programa, una tasa de fallos del 10% y cada fallo añade una penalización adicional de 20 ciclos para ser servido del segundo nivel de cache (L2). De momento supondremos que L2 no tiene fallos (esta combinación la denominamos **CPU_2**).

b) **Calcula** el número de accesos a L2 y los ciclos que tarda en ejecutarse el programa en la **CPU_2**.

Se denomina **tasa de fallos local** a la relación entre los fallos que se producen en una cache (L2 en este caso) y los accesos que se realizan a dicha cache. Para nuestro programa, la cache L2 tiene una tasa de fallos local del 20%. Habitualmente, en una jerarquía de memoria, cada nivel contiene un subconjunto de los datos del nivel superior. En nuestro caso L2 incluye los datos de L1, este tipo de cache (L2) se denomina **cache inclusiva** (incluye los datos de L1). Cuando hay un fallo en L1 y L2 se trae el nuevo bloque de memoria principal a L2 y L2 sirve el bloque a L1. Cada fallo en L2 tiene una penalización adicional, respecto la **CPU_2**, de 100 ciclos. Denominaremos **$CPU_{inclusiva}$** a la organización con la jerarquía de memoria completa en donde L2 es inclusiva.

c) **Calcula** el número de accesos a memoria principal y los ciclos que tarda en ejecutarse el programa en la **$CPU_{inclusiva}$**.

Una alternativa a la cache L2 inclusiva es lo que se denomina **cache exclusiva** (utilizada por algunos procesadores AMD). En una cache L2 exclusiva, los datos que hay en L1 no están repetidos en L2. La L2 exclusiva funciona de forma similar a la cache de víctimas vista en problemas (problema 9 del tema 3), con la salvedad de que no tiene por que ser totalmente asociativa y su tamaño es mucho mayor que el de L1. En caso de fallo en L1 miramos en L2 y, si también es fallo en L2, el bloque se trae directamente de memoria principal a L1. El bloque expulsado de L1 se guarda en L2. En caso que sea acierto en L2 el bloque se trae de L2 y el bloque expulsado de L1 se guarda en L2.

La cache exclusiva tiene la ventaja de que la penalización en caso de fallar en L1 y L2 es ligeramente inferior a la suma de penalizaciones causadas por servir el fallo de L2 (apartado c) y servir el fallo de L1 (apartado b). En nuestro caso un acceso que sea fallo de L1 y L2 tiene una penalización de 110 ciclos. Sin embargo tiene el inconveniente de que los accesos que son fallo en L1 pero acierto en L2 deben leer un bloque de L2 (el bloque pedido) y escribir otro bloque en L2 (el bloque expulsado de L1). Por lo que la penalización es mayor, en nuestro caso es de 30 ciclos (en lugar de los 20 ciclos de la L2 inclusiva). A esta organización la denominaremos **$CPU_{exclusiva}$**. De momento supondremos que L2 sigue teniendo una tasa de fallos local del 20%.

d) **Calcula** los ciclos que tarda en ejecutarse el programa en la **$CPU_{exclusiva}$**.

Otra ventaja de la cache exclusiva es que, para los mismos tamaños de L1 y de L2, puede contener más datos que la inclusiva (los datos de L1 no están repetidos en L2), por lo que la tasa de fallos suele ser inferior.

e) **Calcula** la tasa de fallos local que debería tener la L2 exclusiva para que nuestro programa tarde los mismos ciclos que con la inclusiva.

Finalmente, se ha organizado la L2 exclusiva en 4 bancos. Si el bloque expulsado de L1 va a un banco de L2 distinto del banco que contiene el bloque que se desea leer la penalización, en caso de acierto en L2, es de solo 20 ciclos (como el caso de la cache inclusiva). Si ambos bloques se encuentran en el mismo banco entonces la penalización sigue siendo de 30 ciclos. En caso de fallo en L1 y L2 la penalización por traer el bloque de memoria principal a L1 sigue siendo de 110 ciclos. Esta organización la denominamos **CPU_{bancos}**. En nuestro programa, la tasa de fallos local de L2 sigue siendo del 20% y los accesos a bloques se distribuyen de forma equiprobable entre los bancos, siendo además independiente el banco en que se encuentra el bloque accedido del banco al que va el bloque reemplazado.

f) **Calcula** los ciclos que tarda en ejecutarse el programa en la **CPU_{bancos}**.
Pista: calcular la probabilidad de que los 2 bloques vayan al mismo banco