COGNOMS:

NOM:

IMPORTANTE leer atentamente antes de empezar el examen: Escriba los apellidos y el nombre antes de empezar el examen. Escriba un solo carácter por recuadro, en mayúsculas y lo más claramente posible. Es importante que no haya tachones ni borrones y que cada carácter quede enmarcado dentro de su recuadro sin llegar a tocar los bordes. Use un único cuadro en blanco para separar los apellidos y nombres compuestos si es el caso. No escriba fuera de los recuadros, todo lo que haya fuera de ellos es ignorado.

Problema 1. (3,2 puntos)

Considerar el siguiente fragmento de código escrito en lenguaje C y correspondiente a la forma IJK del producto de matrices C = A * B trabajado en la práctica 8:

```
main()
{
 float A[M][M], B[M][M], C[M][M];
 int i,j,k;
 . . .
  for (i = 0; i < M; i++)
   for (j = 0; j < M; j++)
    for (k = 0; k < M; k++)
     C[i][j] = C[i][j] + A[i][k] * B[k][j];
  . . .
}
```

Suponer que disponemos de una cache de datos de tamaño ilimitado, totalmente asociativa, con líneas de 16 bytes, y que las matrices A, B y C no se encuentran en cache al comienzo del código. El tamaño de página de nuestro sistema es de 16 Kbytes. La constante M toma valores grandes (M > 256).

Responde a las siguientes preguntas:

a) **Indica** si al acceder a las matrices A, B y C la cache aprovecha o no la Localidad Espacial y por qué.

matriz A : SI --> recorrido por Filas

matriz B : NO --> recorrido por Columnas

matriz C : SI --> recorrido por Filas

b) **Indica** para las 3 categorías de fallos de cache (carga, capacidad, conflicto) cuáles aparecen en el código y cuáles no y por qué.

Categoría 1: Carga --> SI, la cache está inicialmente vacía

Categoría 2: Capacidad --> NO, la cache tiene tamaño ilimitado

Categoría 3: Conflicto --> NO, la cache es totalmente asociativa

TODOS los fallos son de Carga.

c) **Calcula** la cantidad de fallos totales en cache que se producen al ejecutar completamente y una única vez el bucle más interno (bucle marcado en **negrita** en el código IJK). Debes dar el resultado en función de la constante M.

Matriz A: M/4

Matriz B: M

Matriz C: 1

Total de fallos = M/4 + M + 1

d) **Calcula** la tasa de fallos en cache que se obtiene al ejecutar completamente el bucle más interno 1 vez y da el resultado en porcentaje.

Total de fallos = M/4 + M + 1

Accesos = 4 * M

Tasa de fallos = (M/4 + M + 1) / (4 * M) = 0.31 ---> m = 31%

e) **Calcula** el total de páginas de Memoria Virtual que se utilizan para ejecutar completamente el bucle más interno 1 vez. Da el resultado para un valor de M = 512.

1 página = 4096 elementos = 8 filas

A: acceso a la Fila i --> 1 página

B: cada 8 filas hay un FP --> M/8 páginas

C: acceso a 1 elemento --> 1 página

Total = 1 + M/8 + 1 páginas = 2 + 64 = 66 páginas

Con el objetivo de reducir los fallos de Carga, a la cache de datos anterior se le añade un mecanismo de prefetch para traer a la cache información antes de que sea solicitada en un futuro cercano. Considerar dos tipos de prefetch: (a) se lanza un prefetch del bloque i+1 cada vez que se accede al bloque i (prefetch OBL - one block lookahead), y (b) al observar una secuencia de accesos a bloque b, b+1*n, entonces se hace prefetch de b+2*n (prefetch con stride).

f) **Calcula** la cantidad de fallos totales en cache que se producen al ejecutar completamente el bucle más interno 1 vez en presencia del prefetch OBL. Debes dar el resultado en función de la constante M.

Matriz A: 1

Matriz B: M

Matriz C: 1

Total de fallos = 1 + M + 1

g) **Calcula** la cantidad de fallos totales en cache que se producen al ejecutar completamente el bucle más interno 1 vez en presencia del prefetch con stride. Debes dar el resultado en función de la constante M.

Matriz A: 2 (hacen falta 2 accesos para poder calcular el stride)

Matriz B: 2

Matriz C: 1

Total de fallos = 5

Suponer que se ejecutan completos los tres bucles del código IJK en un procesador con una cache de tamaño limitado y para un valor de M = 256. Al pasarle al ejecutable la herramienta valgrind observamos que se ejecutan $691x10^6$ instrucciones dinámicas y al instrumentar el código con GetTime ( ) obtenemos un tiempo de ejecución de 57.85 ms. Además, el código IJK se modifica implementando una ordenación distinta de sus bucles y añadiendo una optimización adicional como la que vimos en la práctica 8. El nuevo código se compila con la opción -O3 y se obtiene un ejecutable que se ejecuta 12 veces más rápido.

h) **Calcula** el rendimiento en MFLOPS del código original y del código optimizado. Da el resultado en GFLOPS.

MFLOPSorig = (2 ops FP x M x M x M) / 57.85ms = (33.5×10^6) / 57.85ms = 580 MFLOPS = 0.58 GFLOPS

MFLOPSopt = (2 ops FP x M x M x M) / (57.85/12)ms = 12 x 0.58 = 6.96 GFLOPS

COGNOMS:

NOM:

Problema 2. (3,4 puntos)

Queremos estudiar el efecto de la búsqueda y descodificación de instrucciones en el rendimiento de un procesador segmentado superescalar con ejecución fuera de orden de la familia x86. Para ello, simulamos la ejecución de un programa de prueba P en un simulador parametrizable. Para simplificar el problema, ignoraremos las interferencias que puedan causar los accesos a datos y nos centraremos exclusivamente en los accesos a instrucciones, por lo que supondremos que en los accesos a datos nunca se produce un fallo al acceder a la cache de datos.

En los procesadores segmentados pueden transcurrir decenas de ciclos hasta que se calcula la dirección destino del salto y el procesador decide si el salto se producirá (*taken*) o por el contrario se seguirá ejecutando en secuencia (*not taken*). La alternativa más simple consiste en ejecutar por defecto instrucciones de forma secuencial, con lo que los saltos *not taken* no incurren en ninguna penalización. Sin embargo, los saltos *taken* incurren en una penalización de 10 ciclos. A este procesador le llamaremos PSeq.

Con el simulador hemos obtenido que el 20% de las instrucciones dinámicas de P son saltos, y que el 75% de los saltos son *taken*. También hemos obtenido que, en un procesador ideal donde los saltos *taken* no penalizan, el CPI es de 0,5 ciclos/instrucción.

a) **Calcula** el CPI del procesador PSeq.

CPI = 0,5 c/i + 0,2 saltos/ins * 0,75 taken/salto * 10 ciclos/taken = 2 ciclos/instrucción

Ante la perdida de rendimiento debida a los saltos, se ha decidido incorporar un predictor de saltos al procesador PSeq, con el que hemos obtenido un CPI de 0,9 ciclos/instrucción. En el diseño propuesto, la penalización debida a un salto mal predicho es de 20 ciclos. Los saltos bien predichos no tienen penalización.

b) **Calcula** la tasa de fallos del predictor de saltos.

0,9 c/i = 0,5 i/c + 0,2 saltos/ins * TF fallos/salto * 20 ciclos/fallo
TF = 0,1 fallos/salto

Finalmente, se ha optado por usar un predictor de saltos más sofisticado que usaremos en los siguientes apartados, que tiene una tasa de aciertos del 95% con el que se obtiene un CPI de 0,7 ciclos/instrucción. Hasta ahora no hemos considerado los fallos en la cache de instrucciones. Los procesadores superescalares leen varias instrucciones cada vez que acceden a la cache de instrucciones, por lo que el número de accesos a la cache es menor que el numero de instrucciones ejecutadas. Sabemos que en P se ejecutan $10x10^9$ instrucciones dinámicas y que se realizan $4x10^9$ accesos a la cache de instrucciones, es decir, se realizan sólo 0,4 accesos a la cache por instrucción. Sabemos además que la cache de instrucciones tiene una tasa de fallos de 0,05 fallos/acceso y la penalización es de 100 ciclos/fallo.

c) **Calcula** los ciclos perdidos debidos a los fallos en la cache de instrucciones y el CPI real del procesador.

Ciclos = $4x10^9$ accesos * 0,05 fallos/acceso * 100 ciclos/fallo = $20x10^9$ ciclos
CPI = 0,7 c/i + $20x10^9$ ciclos / $10x10^9$ instrucciones = 2,7 ciclos/instrucción

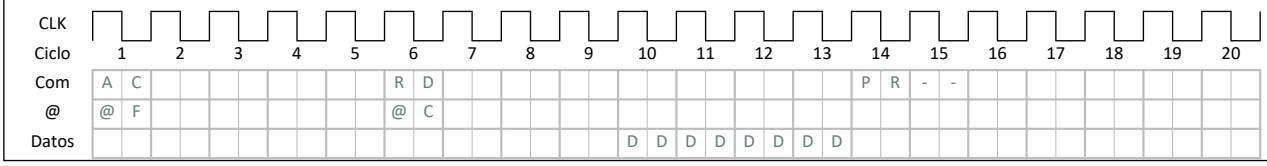
El conjunto procesador-cache está conectado a un sistema de memoria principal[1] mediante un único canal de 64 bits al que se ha conectado un DIMM de 4 Gbytes. Este DIMM tiene 8 chips de memoria DDR-SDRAM (Double Data Rate Synchronous DRAM) de un byte de ancho cada uno. El DIMM esta configurado para leer/escribir ráfagas de 64 bytes. La

1.La CPU funciona a una frecuencia interna mayor que la memoria por lo que un ciclo de memoria corresponde a múltiples ciclos de CPU.

latencia de fila es de 5 ciclos, la latencia de columna es de 4 ciclos y la latencia de precarga es de 2 ciclos.

d) Suponiendo que el comando ACTIVE se lanza en el ciclo 1, **indica** en qué ciclo se lanzará el comando PRECHARGE en una operación de lectura de un bloque de 64 bytes de la memoria DDR-SDRAM. Justifica la respuesta.

En el ciclo 14



Esta DDR funciona a una frecuencia de 1 GHz. La potencia consumida por la DDR depende de su actividad. Durante un acceso a un bloque de 64 bytes se consumen 10 W (desde que se envía el comando ACTIVE hasta que se completa el PRECHARGE), mientras que el resto del tiempo consume sólo 1 W. Sabemos que el programa P ha tardado 9 segundos en ejecutarse y que se han realizado $0{,}2 \times 10^9$ accesos de 64 bytes a la DDR.

e) **Calcula** la potencia media consumida por la memoria DDR durante la ejecución del programa P.

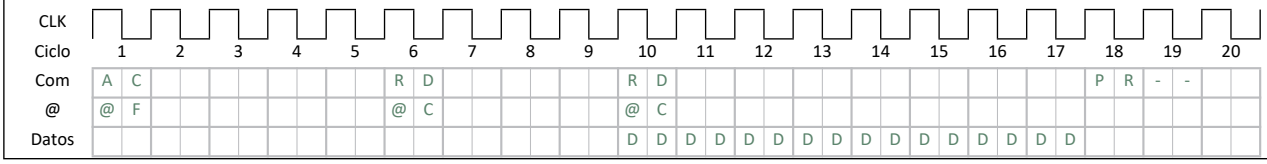
Tiempo a 10 W = $0{,}2 \times 10^9$ accesos * 15 ciclos/acceso / 1×10^9 ciclos/s = 3 segundos ---> Tiempo a 1 W = 6 segundos

Pmedia = (10 W * 3 s + 1 W * 6 s) / 9 s = 4 W

Dado que el rendimiento obtenido hasta ahora es bastante pobre, se ha sugerido hacer prefetch de instrucciones. Se sugiere hacer prefetch en caso de fallo, de forma que en cada fallo se leerán dos bloques de 64 bytes cada uno: el que ha provocado el fallo y el siguiente. En este caso, el controlador de memoria envía los comandos necesarios a la DDR-SDRAM de forma que ambos accesos sean servidos lo más rápidamente posible. Supongamos que los dos bloques están en la misma página de DRAM y que la página no está abierta.

f) Suponiendo que se maximiza el ancho de banda útil del bus y que el comando ACTIVE del primer bloque se lanza en el ciclo 1, **indica** en qué ciclo se lanzará el comando PRECHARGE en una operación de lectura de dos bloques consecutivos de 64 bytes de la memoria DDR-SDRAM. Justifica la respuesta.

En el ciclo 18



Nuestro procesador traduce las instrucciones x86 a microoperaciones RISC (uops). Cada acceso a la cache consume 2,5 nJ (sea acierto o fallo) y cada instrucción x86 gasta 1 nJ en ser descodificada y traducida a uops.

g) **Calcula** la energía consumida por la búsqueda y traducción de las instrucciones x86 durante la ejecución de P.

E = 10×10^9 instr * 1×10^{-9} J/instr + 4×10^9 accesos * $2{,}5 \times 10^{-9}$ J/accesos = 20 J

Se ha decidido introducir una cache de micro-ops donde se guardan las uops que ya han sido traducidas para que en caso de acierto no sea necesario descodificar y traducir de nuevo las instrucciones x86. Con el simulador se ha medido que para ejecutar P se generan 15×10^9 uops dinámicas y se realizan 5×10^9 accesos a la cache de uops. Gracias a esto, el número de accesos a la cache de instrucciones se ha reducido a $0{,}4 \times 10^9$ accesos y el número de instrucciones x86 descodificadas se ha reducido a 1×10^9 instrucciones. Un acceso a la cache de uops consume 1,6 nJ.

h) **Calcula** la energía consumida por el sistema con cache de uops durante la ejecución de P.

E = 1×10^9 instr * 1×10^{-9} J/instr + $0{,}4 \times 10^9$ accesos * $2{,}5 \times 10^{-9}$ J/acceso + 5×10^9 accesos * $1{,}6 \times 10^{-9}$ J/acceso= 10 J

COGNOMS:

NOM:

Problema 3. (3,4 puntos)

Una empresa nos ha contratado para analizar su aplicación y proponer mejoras que reduzcan el tiempo de ejecución. Hemos hecho un estudio sobre la aplicación ejecutándola con un solo procesador y un solo disco (un PC de sobremesa con un procesador single-core), y este nos muestra que la aplicación ejecuta de forma iterativa tres fases bien diferenciadas, tal como se ve en la figura:

```
for (;;) {
  LeerDatos();
  ProcesarDatos();
  EscribirResultados();
}
```

- **Fase 1:** LeerDatos(), únicamente se hacen lecturas desde disco. Ocupa el 10% de toda la ejecución.
- **Fase 2:** ProcesarDatos(), parte intensiva en cálculo que es totalmente paralelizable. Ocupa el 75% de la ejecución.
- **Fase 3:** EscribirResultados(), los datos recién calculados se escriben a disco. Ocupa el 15% de la ejecución.

El tiempo de ejecución de la aplicación es de 100 horas. Queremos estudiar diferentes alternativas que nos permitan reducir este tiempo.

a) **Calcula** el Speedup máximo que podríamos obtener si ejecutáramos este programa en un supercomputador con un número infinito de procesadores y un disco.

S = 1 / (1-0.75) = 4

b) **Calcula** con cuántos procesadores tenemos que ejecutar el programa para obtener un Speedup de 3.

S = 1 / (1-0.75+0.75/s) = 3 --> $s_{fase2} = 9$

Dado que gran parte de la aplicación puede paralelizarse completamente, ofrecemos a la empresa reemplazar el PC anterior por uno nuevo con un procesador multi-core de 12 núcleos.

Además, hemos detectado que el 80% de la Fase 2 se dedica a hacer operaciones matemáticas entre varias matrices en las que cada cálculo es independiente de los demás, así que decidimos transformar el código en esas regiones para utilizar instrucciones SIMD. La arquitectura de los núcleos del procesador nos permite utilizar registros xmm de 128 bits, y el código trabaja con matrices de enteros de 4 bytes.

c) **Calcula** el Speedup de la Fase 2 después de aplicar la transformación en el código y de ejecutarlo en el multi-core de 12 nucleos. **Calcula** el tiempo total de la aplicación después de estas mejoras.

$s_{fase2+SIMD} = 1 / (0.2/12+0.8/(12*4)) = 30$
S = 1 / (1-0.75+0.75/30) = 3.64
T = 100h / 3.64 = 27.5 h

Para mejorar las fases de lectura/escritura de datos necesitaremos un sistema de almacenamiento que nos dé mayor ancho de banda que el actual. Además queremos añadir algún mecanismo de tolerancia a fallos en disco. Valoraremos los sistemas de almacenamiento RAID 10 (con mirror doble), RAID 5 y RAID 6. La aplicación necesita un total de 18 terabytes de almacenamiento.

Para implementar los diversos sistemas RAID disponemos de un único modelo de disco duro de 3TB.

d) **Calcula** cuántos discos duros de 3TB necesitaríamos para cada uno de los niveles de RAID 10 (mirror doble), 5 y 6 si queremos almacenar al menos 18TB útiles

RAID 10 -> 18 TB * 2 mirrors / 3 TB/disco = 12 discos
RAID 5 -> 18 TB / 3 TB/disco + 1 paridad = 7 discos
RAID 6 -> 18 TB / 3 TB/disco + 2 paridad = 8 discos

Después de explicar las diferencias entre los tres niveles de RAID, la empresa se decanta por montar un RAID5 con 10 discos duros de 3TB. Cada disco duro tiene un MTTF de 100.000 horas, y se calcula que, en caso de fallo, el tiempo necesario para reemplazar un disco físico y reconstruir los datos es de 4 horas.

e) **Calcula** el MTTF del sistema de almacenamiento para nuestro RAID 5 con 10 discos físicos.

$MTTF_{RAID5} = 100000^2 / (10*9*4) = 27.8 * 10^6$ horas

Por último, la empresa nos pide que calculemos el Speedup de las dos fases de acceso a disco con RAID 5 respecto al original, teniendo en cuenta que en la Fase 1 se leen 7.2 TB y en la Fase 3 se escriben 10.8 TB. Se considera que todos los accesos a disco son secuenciales.

f) **Calcula** el ancho de banda de lectura en la Fase 1 en nuestro sistema RAID 5 con 10 discos físicos.

AB_{seq} = 7.2 TB / 10 h = 200 MB/s

$AB_{f1} = AB_{lectura} = N * AB_{seq} = 10*200 = 2$ GB/s

g) **Calcula** el ancho de banda de escritura en la Fase 3 en nuestro sistema RAID 5 con 10 discos físicos.

AB_{seq} = 10.8 TB / 15 h = 200 MB/s

$AB_{f3} = AB_{escritura} = (N-1) * AB_{seq} = 1.8$ GB/s

h) **Calcula** el Speedup y el tiempo total de la aplicación respecto al original (1 core + 1 disco) después de aplicar todas las optimizaciones (RAID 5 con 10 discos + 12 cores + SIMD).

S_{f1} = 2000 / 200 = 10

S_{f2} =30

S_{f3} = 1800 / 200 = 9

S = 1 / (0.1/10 + 0.75/30 + 0.15/9) = 19.35

T = 100h / 19.35 = 5.17h