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

2on Control Arquitectura de Computadors **Curs 2010-2011 Q2**

Problema 1. (4 puntos)

A pleno rendimiento, una CPU funciona a una frecuencia de 3 GHz y está alimentada a 1,6 V. En modo bajo consumo la CPU funciona a una frecuencia de 1 GHz y está alimentada a 1 V. Hemos medido que el consumo de la CPU en alto rendimiento es de 120W y en modo bajo consumo es de 27,5 W. En estos datos solo se considera la potencia debida a conmutación y la debida a fugas. Tanto la corriente de fugas (I) como la carga capacitiva equivalente (C) son las mismas en ambos modos.

a) **Calcula** la corriente de fugas (I) y la carga capacitiva equivalente (C) de la CPU (usar prefijo más adecuado del SI)

Hemos simulado la ejecución de un programa en esta CPU con un sistema de memoria en donde todos los accesos a memoria tardan 1 ciclo sean aciertos o fallos (denominaremos $\mathbf{CPU_{IDEAL}}$ a esta combinación simulada) y hemos obtenido que el programa se ejecuta en $15x10^9$ ciclos, ejecuta $5x10^9$ instrucciones, realiza $6x10^9$ accesos a memoria y de estos, $500x10^6$ son fallos de cache. Durante la ejecución de un programa la CPU está en modo alto rendimiento.

b) **Calcula** el CPI, el numero de accesos por instrucción, la tasa de fallos y el tiempo de ejecución del programa en la $\mathbf{CPU_{IDEAL}}$.

Queremos integrar esta CPU con una cache unificada (instrucciones+datos) multibanco de mapeo directo organizada en 4 bancos. Esta cache no está segmentada y su tiempo de acceso es de 0,6 ns. Obsérvese que el tiempo de acceso es mayor que el tiempo de ciclo del procesador, por lo que al acceder a cache, el procesador se bloquea durante unos ciclos, y por tanto se produce una pequeña penalización respecto a la $\mathbf{CPU_{IDEAL}}$ (incluso en caso de acierto). En caso de que el acceso sea un fallo de cache, hay una penalización adicional de 20 ciclos más.

c) **Calcula** los ciclos de penalización en caso de acierto y de fallo.

d) **Calcula** el CPI, y el tiempo de ejecución cuando ejecutamos el programa con la cache multibanco.

Nuestra CPU es capaz de continuar ejecutando instrucciones mientras se accede la cache, sin embargo en el apartado d) bloqueamos la CPU en cada acceso para evitar lanzar un segundo acceso a la cache antes de que acabe el acceso anterior. Una posible mejora, que denominaremos control de bloqueos de cache, consiste en no bloquear la CPU en cada acceso, sino solamente si se inicia un acceso antes de que el anterior haya terminado. La CPU no soporta loads no bloqueantes, por lo que en caso de fallo siempre bloquearemos la CPU mientras se trae el bloque del siguiente nivel de la jerarquía (ciclos de penalización adicional). Sabemos que la probabilidad de realizar un acceso es la misma en todos los ciclos y es independiente de lo sucedido en ciclos anteriores. Durante los ciclos que no está bloqueada, la CPU se comporta exactamente igual que en el caso ideal.

e) **Calcula** el tiempo medio entre accesos (en ciclos), la probabilidad de acceder a memoria en un ciclo determinado y la probabilidad de que al realizar un acceso la cache esté ocupada.

f) **Calcula** el CPI, y el tiempo de ejecución cuando ejecutamos el programa en la CPU con control de bloqueos de cache.

En una cache organizada en bancos el acceso a cada banco es independiente, por lo que es posible acceder a un banco aunque otro este ocupado. Una posible mejora, que denominaremos control de bloqueos de banco, consiste en bloquear la CPU solamente en caso de que accedamos a un banco ocupado. En nuestro caso, sabemos que en cada acceso la probabilidad de acceder a cualquiera de los 4 bancos es la misma, y que es independiente de los accesos anteriores. Como en el caso anterior, la CPU no soporta loads no bloqueantes, por lo que en caso de fallo siempre bloquearemos la CPU mientras se trae el bloque del siguiente nivel de la jerarquía (ciclos de penalización adicional).

g) **Calcula** la probabilidad de que al realizar un acceso el banco accedido esté ocupado.

h) **Calcula** el CPI, y el tiempo de ejecución cuando ejecutamos el programa en la CPU con control de bloqueos de banco.

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

2on Control Arquitectura de Computadors **Curs 2010-2011 Q2**

Problema 2. (3 puntos)

Dado el siguiente código en ensamblador:

```
        movl $0, %esi
        movl $0, %ebx
for:    movl v(,%esi,4), %eax
        addl %eax, %ebx
        incl %esi
        cmp $200000000,%esi
fin:    jne for
        movl %ebx, resultado
```

Suponiendo un procesador con memoria perfecta (tiempo de acceso de un ciclo), un IPC (Instrucciones Por Ciclo) de 0,4 y una frecuencia de 1GHz

a) **Calcula** cuanto tiempo tarda en ejecutarse el bucle anterior.

Al procesador del apartado a) le acoplamos un sistema de memoria real con una cache de datos que tiene una tasa de fallos del 6,25%. Suponemos que la cache de instrucciones siempre acierta. Medimos de nuevo el tiempo de ejecución del programa y obtenemos 3,75s.

b) **Calcula** la penalización en ciclos por fallo de la cache que hemos incorporado. Si no se puede calcular contestad "INDEFINIDO".

c) **Deduce** las siguientes características de la cache a partir de la tasa de fallos para el bucle anterior. Si alguna característica no puede averiguarse escribid "INDEFINIDO".

Para intentar mejorar el rendimiento se decide utilizar una cache NON-BLOCKING que puede soportar hasta 16 fallos en vuelo aunque sean al mismo bloque de cache. El procesador ejecuta las instrucciones en orden y se bloquea cuando necesita el dato que ha fallado en cache.

d) ¿Cuantas instrucciones hay entre la que provoca el fallo de cache y la que necesita el dato? ¿cual es el nuevo Texe con la cache NON-BLOCKING?

Para mejorar el rendimiento del programa se decide incorporar al bucle anterior el siguiente código de prefetch software (además de la caché NON-BLOCKING). Para ello se inserta una línea de código con la instrucción "prefetch" que hace que la línea con la dirección indicada se cargue en cache si no estaba en ella.

```
        movl $0, %esi
        movl $0, %ebx
for:    prefetch v+64(,%esi,4) // carga la línea indicada en cache
        movl v(,%esi,4), %eax
        addl %eax, %ebx
        incl %esi
        cmp $200000000,%esi
fin:    jne for
        movl %ebx, resultado
```

e) **Calcula** el tamaño mínimo que debería tener la cache para poder aprovechar el código del bucle anterior?

2 líneas de cache / 128bytes

f) ¿Cuantas instrucciones se ejecutan desde que el procesador tiene en %eax el dato de una línea de cache hasta que necesita en %eax el primer dato de la siguiente línea de cache? ¿Cuantos ciclos tarda en ejecutarlas sabiendo que en esas instrucciones no tiene fallos en cache? ¿Es suficiente para esconder la latencia de un fallo de cache?

g) **Calcula** cual es el nuevo tiempo de ejecución con el código de prefetch.

Para mejorar aún más el programa se decide desenrollar el bucle un factor 2.

h) ¿Cuantas iteraciones tendrá el bucle? ¿Cuantas instrucciones se ejecutarán en cada iteración (suponed que NO usamos instrucciones SIMD)?

i) En este nuevo caso: ¿Cuantas instrucciones se ejecutan desde que el procesador tiene en %eax el dato de una línea de cache hasta que necesita en %eax el primer dato de la siguiente línea de cache? ¿Cuantos ciclos tarda en ejecutarlas sabiendo que en esas instrucciones no tiene fallos en cache? ¿Es suficiente para esconder la latencia de un fallo de cache? ¿Cual será el nuevo tiempo de ejecución en este caso?

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

2on Control Arquitectura de Computadors **Curs 2010-2011 Q2**

Problema 3. (2 puntos)

En el pasado control, programamos en ensamblador x86 la rutina Xprob3 que llamaba a la rutina Exa. En la siguiente figura se muestra el código C de las rutinas Xprob3 y Exa, y parte de las traducción a x86 de la rutina Xprob3:

```
int Exa(int v[], int x) {
  int i;
  i = v[x];
  return v[i];
}
int XProb3(int v[], int *p, int m){
  int i;
  for (i=0; i<1000000; i++)
    v[i] += Exa(v, *p);
  return *p + m;
}
```

```
Xprob3: ...
for:   movl 12(%ebp), %eax
       pushl (%eax)
       pushl %ebx
       call Exa
       addl $8, %esp
       addl %eax, (%ebx, %esi, 4)
       incl %esi
       cmpl $1000000, %esi
       jl for
endfor: ...
```

a) **Traduce** a ensamblador del x86 la subrutina Exa. Dibujad el bloque de activación de la rutina Exa.

Supongamos que en nuestro procesador todas las instrucciones tardan 1 ciclo en ejecutarse. Además, cada acceso a memoria de datos (lectura o escritura) cuesta 1 ciclo adicional.

b) **Completa** la siguiente tabla la siguiente tabla:

	#instrucciones ejecutadas	#accesos a memoria de datos	# ciclos
1 iteración del for (Xprob3) sin contar la rutina Exa	9		
rutina Exa (estimación)	9	7	
1 iteración del for (Xprob3) contando la rutina Exa	18		

Suponiendo que nuestro procesador funciona a 2 GHz.

c) **Calcula** (considerando las 10^6 iteraciones del bucle for) el número total de instrucciones ejecutadas, el número total de accesos a memoria de datos, el CPI, el tiempo de ejecución, los MIPS.

Suponiendo que cada instrucción ocupa 4 bytes y que todos los accesos a memoria de datos son de 4 bytes.

d) **Calcula** el ancho de banda consumido por esta subrutina (instrucciones y datos), considerando las 10^6 iteraciones del bucle for.

Problema 4. (1 punto)

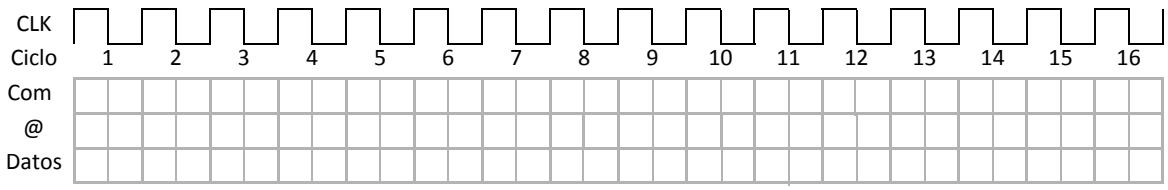
Disponemos de un DIMM de memoria DRAM síncrona (SDRAM) con las siguientes características:

- 8 chips de 1 byte cada uno por DIMM; Latencia de fila: 4 ciclos; Latencia de columna: 3 ciclos; Latencia de precarga: 2 ciclos; Frecuencia de reloj: 200 MHz.

A esta memoria realizamos un acceso en lectura en el que leemos un paquete de 32 bytes. Para indicar la ocupación de los distintos recursos utilizaremos la siguiente nomenclatura:

- **ACT**: comando ACTIVE; **RD**: comando READ; **PRE**: comando PRECHARGE; **@F**: ciclo en que se envía la dirección de fila; **@C**: ciclo en que se envía la dirección de columna; **Di**: ciclo en que se transmite el paquete de datos i (D0, D1, D2, ...)

a) **Rellena** el siguiente cronograma indicando la ocupación de los distintos recursos para una operación de lectura de 32 bytes.



b) **Calcula** el tiempo de ciclo de la memoria (en ns.) y el ancho de banda real suponiendo que somos capaces de iniciar un nuevo acceso a un bloque de 32 bytes tan pronto hemos completado el acceso anterior.