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

1er Control Arquitectura de Computadors **Curs 2010-2011 Q2**

Problema 1. (4 puntos)

Un computador está formado por los componentes mostrados en la tabla siguiente. La tabla también muestra el numero de componentes de cada tipo y el tiempo medio hasta fallo (MTTF) de cada componente.

Componente	Fuente alimentación	CPU	Ventilador CPU	Placa base	DIMMs	Disco duro	SSD *
Nº	1	1	1	1	4	1	1
MTTF (horas)	125.000	1.000.000	100.000	200.000	1.000.000	100.000	500.000

* SSD: Solid State Disc (disco de estado solido)

El tiempo medio para reemplazar un componente que ha fallado (*mean time to repair*) es de 10 horas.

a) **Calcula** el tiempo medio hasta fallos del sistema (MTTF).

MTTF = 1/(1/125000+1/1000000+1/100000+1/200000+4/1000000+1/100000+1/500000) = **25000 horas**

b) **Calcula** el tiempo medio entre fallos (MTBF).

MTBF = MTTF + MTTR = **25010 horas**

c) **Calcula** la disponibilidad del sistema.

25000 h / 25010 h * 100 = **99,96%**

La CPU de este sistema tiene una superficie de 200 mm^2 y se fabrica en una oblea de silicio con una superficie útil de 64.000 mm^2. El coste energético de la oblea y el proceso de impresión y verificación de los dados es de 25.600 MJoules. Durante este proceso el factor de yield es del 80%. El coste de empaquetado y test final de las CPUs es de 25 MJoules por dado y el yield final de las CPUs después del test final es del 87,5%.

d) **Calcula** el coste energético de un dado (antes del empaquetado y testeo final).

64.000 mm^2 / 200 mm^2/dado = 320 dados
320 dados x 0,8 dados buenos/dado = 256 dados buenos
25.600 MJ/oblea / 256 dados/oblea = **100 MJoules / dado**

e) **Calcula** el coste energético final de una CPU.

coste por oblea con empaquetado y testeo = 256 dados x (100 MJ/dado + 25 MJ/dado) = 32.000 MJ/oblea
CPUs funcionales = 256 dados * 0,875 = 224 CPUs
coste por CPU = 32.000 MJ/oblea / 224 CPUs/oblea = **142,9 Mjoules/CPU**

En este sistema tenemos instalado el entorno usado en el laboratorio de AC y hemos medido que un programa se ha ejecutado en 2 segundos usando $6x10^9$ ciclos y ha ejecutado $4,8x10^9$ instrucciones

f) **Calcula** el CPI del programa y la frecuencia de la CPU (usa el prefijo del sistema internacional más adecuado).

CPI = $4,8x10^9$ instrucciones / $6x10^9$ ciclos = **1,25 c/i**
Frec = $6x10^9$ ciclos / 2 segundos = $3x10^9$ ciclos/segundo = **3 GHz**

El tiempo de ejecución calculado anteriormente se corresponde al tiempo de CPU (usuario + sistema). Usando el comando "time" de linux hemos obtenido que el tiempo de CPU representa solo el 20% del tiempo total del programa (wall time). El 80% restante es tiempo de entrada/salida (accesos al disco duro concretamente). Cada acceso al disco duro tarda 8 milisegundos, mientras que si los datos estuviesen en el disco SSD cada acceso tardaría 10 microsegundos.

g) **Calcula** la ganancia en la parte de entrada salida si los datos del programa estuviesen en el SSD en lugar de el disco duro.

Ganancia = $8x10^{-3}$ segundos /acceso / $10x10^{-6}$ segundos /acceso = **800**

h) **Calcula** la ganancia total en el programa a partir de la ganancia en entrada salida.

Ganancia = 1/((1-fm)+fm/gm) = 1/((1-0,8)+0,8/800) = **4,975**

A pleno rendimiento la CPU tiene una carga capacitiva equivalente de 16 nF (nanoFaradios), funciona a un voltaje de 1,25 V y una frecuencia de 2GHz. Se ha determinado que esta CPU tiene una corriente de fugas de 8 A.

i) **Calcula** la potencia media debida a fugas, la debida a conmutación y la total cuando la CPU esta a pleno rendimiento.

P fugas = I*V = 8 A * 1,25 V = 10W
Pconmutacion = $C*V^2*f$ = $16x10^{-9}$ F * $(1,25\ V)^2$ * $2x10^9$ Hz = 50 W
Ptotal = 10 W + 50 W = **60 W**

Las CPUs actuales, cuando no están a plena carga, reducen el voltaje y la frecuencia para ahorrar energía. En modo bajo consumo nuestra CPU consume tan solo 20 W. Sabemos que nuestro sistema está 4 horas diarias en modo alto rendimiento, 10 horas en modo bajo consumo y el resto esta totalmente apagado (consumo 0 W).

j) **Calcula** la energía que ahorramos cada día gracias a la reducción de frecuencia y voltaje de la CPU (usa el prefijo del sistema internacional más adecuado).

El ahorro solo se produce en los periodos de bajo rendimiento

Ahorro = tiempo * reducción potencia = 10 h * 3600 s/h * (60 W - 20 W) = $1,44x10^6$ Joules = **1,44 MJoules**

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

1er Control Arquitectura de Computadors **Curs 2010-2011 Q2**

Problema 2. (3 puntos)

Dado el siguiente código escrito en C:

```
typedef struct {
  int a;
  char b;
  char c;
  double d;
} s1;

typedef struct {
  short e[5];
  s1 f;
} s2;

short F(s1 *alto, int bola, char *cola);
int examina(s1 uno, char dos, s2 *tres){
  char vl1;
  int vl2;
  ...
}
```

a) **Dibuja** como quedarían almacenadas en memoria las estructuras s1 y s2, indicando claramente los deplazamientos respecto al inicio y el tamaño de todos los campos.

s1 (16 bytes)

4	a	+0
1	b	+4
1	c	+5
2	---	
8	d	+8

s2 (28 bytes)

	e[0]	+0
10	. .	
	e[4]	
2	---	
16	f	+12

b) **Dibuja** el bloque de activación de la función examina, indicando claramente los desplazamientos relativos al registro EBP necesarios para acceder a los parámetros y a las variables locales.

1	vl1	-8
3	---	
4	vl2	-4
4	ebp old	<--%ebp
4	ret	
16	uno	8
1	dos	24
3	---	
4	tres	28

c) **Traduce** la siguiente sentencia a ensamblador del x86, suponiendo que está dentro de la función examina:

```
vl1=dos+uno.b;
```

```
movb 24(%ebp),%al
addb 12(%ebp),%al
movb %al,-8(%ebp)
```

d) **Traduce** la siguiente sentencia a ensamblador del x86, suponiendo que está dentro de la función examina:

```
tres->e[1]=F(&uno, vl2, &uno.c);
```

```
leal 13(%ebp), %eax
pushl %eax
pushl -4(%ebp)
leal 8(%ebp), %eax
pushl %eax
call F
addl $12, %esp
movl 28(%ebp), %ecx
movw %ax, 2(%ecx)
```

e) **Traduce** la siguiente sentencia a ensamblador del x86, suponiendo que está dentro de la función examina:

```
if (vl2 > 0)
    vl2 = tres->f.a;
```

```
cmpl $0,-4(%ebp)
jle fin
movl 28(%ebp),%ecx
movl 12(%ecx),%ecx
movl %ecx,-4(%ebp)
fin:
```

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

1er Control Arquitectura de Computadors **Curs 2010-2011 Q2**

Problema 3. (3 puntos)

Dado el siguiente código escrito en C:

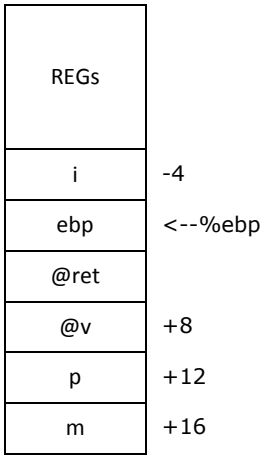
```
int Exa(int v[], int x);
int XProb3(int v[], int *p, int m){
  int i;
  for (i=0; i<1000000; i++)
    v[i] += Exa(v, *p);
  return *p + m;
}
```

a) **Dibuja** el bloque de activación de de la subrutina Xprob3.

b) **Traduce** a ensamblador del x86 la subrutina Xprob3.

```
Xprob3: pushl %ebp
        movl %esp,%ebp
        subl $4, %esp
        pushl %esi
        pushl %ebx
        movl 8(%ebp),%ebx
        xorl %esi,%esi
for:    cmpl $1000000, %esi
        jge endfor
        movl 12(%ebp), %eax
        pushl (%eax)
        pushl %ebx
        call Exa
        addl $8, %esp
        addl %eax, (%ebx, %esi, 4)
        incl %esi
        jmp for
endfor: movl 12(%ebp), %eax
        movl (%eax), %eax
        addl 16(%ebp), %eax
        popl %ebx
        popl %esi
        movl %ebp, %esp
        popl %ebp
        ret
```

Bloque activación



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)

$(1{,}6\ V)^2 * 3x10^9\ Hz * \mathbf{C} + 1{,}6\ V * \mathbf{I} = 120\ W$ (ecuación alto rendimiento)
$(1\ V)^2 * 1x10^9\ Hz * \mathbf{C} + 1\ V * \mathbf{I} = 27{,}5\ W$ (ecuación bajo consumo)

resolvemos sistema de 2 ecuaciones lineales con 2 incognitas .....

C = 12,5 nF
I = 15 A

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 **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 **CPU_{IDEAL}**.

CPI = $15x10^9$ ciclos / $5x10^9$ instrucciones = **3 c/i**
api = $6x10^9$ accesos / $5x10^9$ instrucciones = **1,2 a/i**
m = $500x10^6$ fallos / $6x10^9$ accesos = 0,083 f/a = **8,33%**
Texe = $15x10^9$ ciclos / $3x10^9$ Hz = **5 s**

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 **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.

penalización acierto = $\lceil 0{,}6x10^{-9}\ s * 3\ x10^9\ Hz \rceil$ - 1 ciclos = 1 ciclos
penalización fallo = 1+20 = 21 ciclos

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

ciclos = $15x10^9$ ciclos + 1 ciclos/acceso * $6x10^9$ accesos + 20 ciclos/fallo * $500x10^6$ fallos = $31x10^9$ ciclos
CPI = $31x10^9$ ciclos / $5x10^9$ instrucciones = **6,2 c/i**
Texe = $31x10^9$ ciclos / $3x10^9$ Hz = **10,33 s**

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.

tiempo medio entre acccessos = $15x10^9$ ciclos / $6x10^9$ accesos = **2,5 ciclos**
probabilidad acceso en un ciclo = 1/tiempo medio = 1/2,5 = **0,4**
la cache sólo puede estar ocupada durante un ciclo (por el acceso anterior)
probabilidad de acceso con cache ocupada = **0,4**

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

ciclos = $15x10^9$ ciclos + 1 ciclos/acceso * $6x10^9$ accesos * 0,4 + 20 ciclos/fallo * $500x10^6$ fallos = $27,4x10^9$ ciclos
CPI = $27,4x10^9$ ciclos / $5x10^9$ instrucciones = **5,48 c/i**
Texe = $27,4x10^9$ ciclos / $3x10^9$ Hz = **9,13 s**

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.

probabilidad de acceso con cache ocupada = **0,4**
probabilidad de acceder mismo banco = **1/4**

probabilidad de acceder a banco ocupado = 1/4*0,4 = **0,1**

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

ciclos = $15x10^9$ ciclos + 1 ciclos/acceso * $6x10^9$ accesos * 0,1 + 20 ciclos/fallo * $500x10^6$ fallos = $25,6x10^9$ ciclos
CPI = $25,6x10^9$ ciclos / $5x10^9$ instrucciones = **5,12 c/i**
Texe = $25,6x10^9$ ciclos / $3x10^9$ Hz = **8,53 s**

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.

CPI = 1/0,4 = 2,5 ciclos/instrucción

Texe= 1000000000 ins * 2,5 ciclos/ins * $1x10^{-9}$ s / ciclo = 2,5 s

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".

Tmem = 3,75 s - 2,5 s = 1,25 s

Tpf = 1,25 s * $1x10^{9}$ ciclos/s / ($2x10^{8}$ acc.mem.* 0,0625 fallos/acc.mem.) = 100 ciclos/fallo

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".

Tamaño Línea: 64 bytes

Tamaño Cache: INDEFINIDO

Asociatividad: 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?

0 instrucciones, el dato lo consume la instrucción siguiente

El mismo que antes (3,75s) ya que necesitamos el dato después de hacer el load y el procesador no puede seguir ejecutando.

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 / 128 bytes

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?

1 línea = 16 elementos

6 instrucciones por elemento => 96 instrucciones => 240 ciclos

Sí

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

CPI = 2,5 (de a)

Texe= (1200000000 ins * 2,5 ciclos/ins) * $1x10^{-9}$ s / ciclo = 3 s

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)?

200000000 iteraciones orig. / 2 = 100000000 iteraciones

Cada iteración se ejecutarán 1 ins. de prefetch + 2*2 de "trabajo" + 3 de control = 8 ins.

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?

Con 64 instrucciones por línea también se tolera la latencia de un fallo de memoria así que:

Texe= 100000000 iteraciones * 8 ins./iteración * 2,5 ciclos/ins. * $1x10^{-9}$ s/ciclo= 2s

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.

```
Exa: pushl %ebp
     movl %esp,%ebp
     movl 8(%ebp),%edx
     movl 12(%ebp),%ecx
     movl (%edx,%ecx,4),%eax
     movl (%edx,%eax,4),%eax
     movl %ebp, %esp
     popl %ebp
     ret
```

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.

Instrucciones ejecutadas: $18 \cdot 10^6$

Accesos a Memoria: $14 \cdot 10^6$

Ciclos totales: $32 \cdot 10^6$

CPI = 32 / 18 = 1,78 ciclos por instrucción

Tiempo de ejecución = $(32 \cdot 10^6) \cdot (1/2 \cdot 10^9) = 16 \cdot 10^{-3}$ s = 16 ms

MIPS = $18 \cdot 10^6 / 16 \cdot 10^{-3} \cdot 10^6 = 1125$ 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.

Instrucciones ejecutadas: $18 \cdot 10^6$

Accesos a Memoria: $14 \cdot 10^6$

Bytes leidos: $128 \cdot 10^6$

Ancho de banda: $128 \cdot 10^6 / 16 \cdot 10^{-3} = 8 \cdot 10^9$ B/s = 8 GB/s

Problema 4. (1 punto)

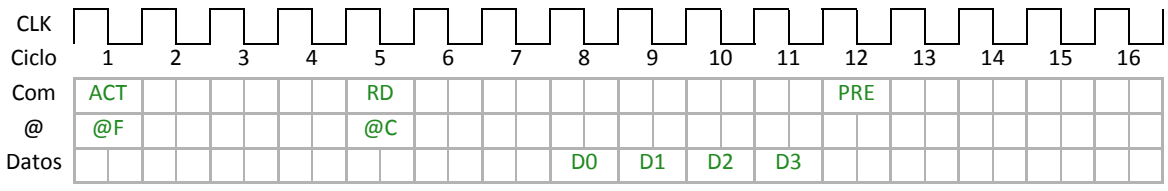
Disponemos de un DIMM de memoria DRAM síncrona (SDRAM) con las siguientes características:

- 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.

Tciclo = 13 ciclos · (1 / 200 · 10^6 ciclos/s) = **65 ns**

ANCHO de BANDA = (32 bytes / 13 ciclos) · 200 · 10^6 ciclos / s = **492 MB/s**

3er Control Arquitectura de Computadors — Curs 2010-2011 Q2

Problema 1. (1 punto).

Dado el siguiente código escrito en C:

```
typedef struct {
  short a;
  char b;
  char c;
  char d;
  short e;
} s1;

char *examen(s1 paruno, s2 *pardos){
char vl1;
   ...
}
```

```
typedef struct {
  char e[5];
  s1 f;
} s2;
```

a) **Dibuja** como quedarían almacenadas en memoria las estructuras s1 y s2, indicando claramente los desplazamientos respecto al inicio y el tamaño de todos los campos.

b) **Dibuja** el bloque de activación de la función examen, indicando claramente los desplazamientos relativos al registro EBP necesarios para acceder a los parámetros y a las variables locales.

Problema 2. (3 puntos)

Disponemos de una CPU que funciona a 3 GHz. En esta CPU hemos simulado la ejecución de un programa 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 $15x10^9$ ciclos, ejecuta $5x10^9$ instrucciones, realiza $6x10^9$ accesos a memoria y de estos, $500x10^6$ son fallos de cache.

a) **Calcula** el numero medio de ciclos entre accesos, la probabilidad de acceder a memoria en un ciclo determinado, el CPI, el tiempo de ejecución del programa en la **CPU_{IDEAL}**.

Queremos integrar esta CPU con una cache unificada (datos + instrucciones) de mapeo directo. 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. En estas condiciones, todos los accesos a memoria se ven penalizados en 1 ciclo (aciertos y fallos). En caso de que el acceso sea un fallo de cache, hay una penalización adicional de 30 ciclos más.

b) **Calcula** los ciclos totales, el CPI, y el tiempo de ejecución cuando ejecutamos el programa con la cache directa.

Esta CPU tiene 2 modos de funcionamiento: modo de alto rendimiento, la CPU funciona a una frecuencia de 3 GHz con un consumo de 70 W; y el modo de bajo consumo en el que la CPU funciona a 1 GHz. En ambos modos la potencia de fugas es 10W. Suponiendo que C y V tienen el mismo valor en ambos modos de funcionamiento.

c) **Calcula** la potencia de conmutación en el modo de alto rendimiento. **Calcula** la potencia total en el modo de bajo consumo.

d) **Calcula** la energía consumida por el procesador con cache unificada ejecutando el programa de prueba, suponiendo que la CPU está en el modo de alto rendimiento.

Con el objetivo de reducir el consumo, se ha modificado el control del procesador para que cuando se produce un fallo de cache, durante los 30 ciclos de penalización (a 3 GHz) el procesador pasa al modo de bajo consumo. El tiempo de penalización por fallo en cache depende del nivel superior de la jerarquía y en este caso es constante e independiente de la frecuencia de reloj de la CPU.

e) **Calcula** el tiempo que la CPU estará en modo bajo consumo (tiempo perdido por fallos de cache).

f) **Calcula** la energía consumida por el procesador ejecutando el programa de prueba, suponiendo que la CPU pasa al modo de bajo consumo mientras se resuelve un fallo de cache. Calcula la potencia media en este último caso.

A partir de aquí supondremos que la CPU funciona siempre a 3 GHz.

Nuestra CPU es capaz de continuar ejecutando instrucciones mientras se accede la cache. Sin embargo en el apartado anterior, bloqueamos la CPU en cada acceso a cache, para evitar lanzar un segundo acceso a la cache antes de que acabe el 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.

g) **Calcula** la probabilidad de que al realizar un acceso a memoria la cache esté ocupada.

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

Otra mejora que podemos añadir, es organizar la cache en 4 bancos. En una cache organizada en bancos el acceso a cada banco es independiente, por lo que es posible acceder a un banco aunque otro esté ocupado. La mejora 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).

i) **Calcula** los ciclos totales, el CPI, y el tiempo de ejecución cuando ejecutamos el programa en la CPU con la cache multibanco.

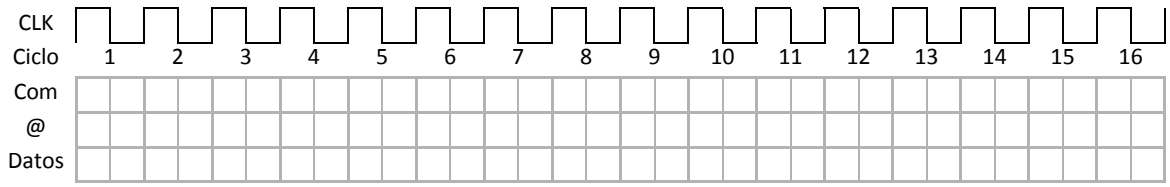
Problema 3. (3 puntos)

Disponemos de un DIMM de memoria **DDR** (Double Data Rate) con 8 chips de 1 byte cada uno por DIMM, la latencia de fila es de 4 ciclos, la de columna es de 3 ciclos y la de precarga es de 2 ciclos. El bus de esta memoria funciona a una frecuencia de reloj de 800 MHz. Los accesos se hacen en ráfagas de 4 paquetes de 8 bytes, es decir se transmiten 32 bytes de datos en cada acceso.

Para indicar la ocupación de los distintos recursos utilizaremos la siguiente nomenclatura:

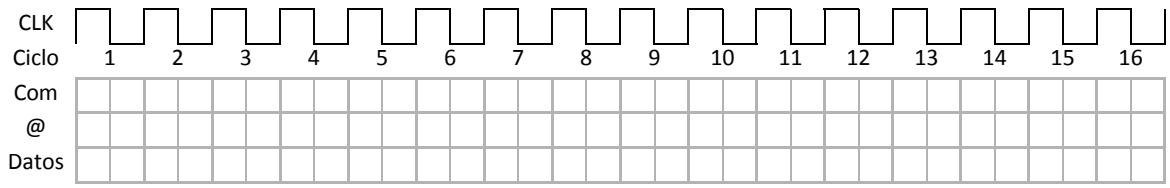
- **ACT**: comando ACTIVE (abrir página)
- **RD**: comando READ
- **WR**: comando WRITE
- **PRE**: comando PRECHARGE (cerrar página)
- **@F**: ciclo en que se envía la dirección de fila
- **@C**: ciclo en que se envía la dirección de columna
- **D**: transmisión de un paquete de 8 bytes

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



Este DIMM lo conectamos a una CPU integrada con un controlador de memoria y un primer nivel de cache con caches de instrucciones y datos separadas, ambas con bloques de 64 bytes. Cuando el controlador de memoria de este procesador desea leer un bloque de memoria de 64 bytes tiene que abrir página, realizar dos lecturas de 32 bytes y cerrar página.

b) **Rellena** el siguiente cronograma indicando la ocupación de los distintos recursos para leer un bloque de memoria de 64 bytes de forma que se haga lo más rápidamente posible.



Esta CPU es un procesador RISC segmentado. En esta CPU ejecutamos el siguiente código, en donde **v** y **c** son variables de punto flotante en doble precisión (8bytes).

```
for (i=0; i<N; i++)
   v[i] = v[i] * c;
```

El compilador ha alineado el vector **v** a un múltiplo del tamaño de bloque y ha traducido el cuerpo del bucle al siguiente código máquina RISC:

```
// valor inicial de los registros: rf1 = variable c; r1  = @ inicio de v; r2  = N
loop: loadf  rf2 <- M[R1]
      mulf   rf3 <- rf2 * rf1
      storef M[r1] <- rf3
      add    r1 <- r1 + 8
      sub    r2 <- r2 - 1
      jnz    r2, loop
```

Se ha calculado que, en ausencia de fallos de cache, el bucle se ejecuta con un CPI de 1,5 ciclos/instrucción. Dado que se trata de un código muy pequeño, que cabe perfectamente en la cache de instrucciones, ignoraremos el efecto de los fallos de instrucciones y nos centraremos en los accesos a datos. La cache de datos es de mapeo directo, tiene una capacidad de 64Kbytes, un tamaño de bloque de 64 bytes y sigue una política de escritura copy back.

c) **Explica** brevemente porque en este fragmento de código no importa si la cache es write allocate o write NO allocate.

Ejecutamos el bucle anterior con N = 8192 (8*1024)

d) **Calcula** cuantos fallos de cache (de datos) se producen, cuantos bloques leemos de memoria principal y cuantos bloques escribimos.

La CPU y las caches funcionan a una frecuencia de 1,6 GHz (el doble del bus) por lo que 1 ciclo de bus se corresponde a 2 ciclos de CPU. Teniendo en cuenta el tiempo de acceso a la DDR más los tiempos de transmisión de los datos y el tiempo de arbitraje, la penalización en caso de fallo es de 25 ciclos de CPU si el bloque a reemplazar tiene el *dirty bit* == 0, mientras que si el bloque a reemplazar tiene el *dirty bit* == 1 la penalización en caso de fallo es de 50 ciclos.

e) **Calcula** el CPI del bucle con N = 8192

En caso que ejecutásemos el bucle con N = 16384 (16*1024)

f) **Calcula** cuantos fallos de cache (de datos) se producen, cuantos bloques leemos de memoria principal y cuantos bloques escribimos.

g) **Calcula** el CPI de la ejecución con N = 16384

Una posible mejora consiste en añadir un mecanismo de *prefetch hardware* de forma que cuando un bloque de cache se accede por primera vez se genera un prefecth del siguiente bloque de memoria. Este bloque que se prebusca puede que reemplace a otro bloque de la cache, por lo que nuevamente se pueden dar los casos de *dirty bit* == 0 y *dirty bit* == 1. El tiempo que tarda el prefetch en ambos casos es el mismo que las respectivas penalizaciones por fallo (25 y 50 ciclos respectivamente).

h) ¿Cuantas instrucciones se ejecutan desde que el procesador accede por 1a vez el 1er dato de un bloque de cache hasta que accede el 1er dato del siguiente bloque 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 prefecth?

i) **Calcula** el CPI del bucle con N = 16384 con el mecanismo de prefecth descrito anteriormente.

Problema 4. (2 puntos)

Una base de datos ocupa 3,6 Tbytes de información. Como medida de seguridad se realizan regularmente backups de la base de datos. Para ello se dispone de un sistema de backup basado en cinta con un ancho de banda efectivo de 50 Mbytes/s. En caso que se produzca una perdida de los datos podremos recuperar el último backup. Dado que el resto del sistema soporta anchos de banda mucho mayores, la cinta de backup es el cuello de botella que limita el tiempo de recuperar el backup.

a) **Calcula** el tiempo (en horas) que se tarda en recuperar un backup de la base de datos completa.

El sistema de disco para almacenar la base de datos esta formado por 20 discos de 500 Gbytes de capacidad. Cada disco ofrece un ancho de banda efectivo de 100 Mbytes/s y tiene un tiempo medio entre fallos ($MTTF_{disco}$) de 100.000 horas. Por cuestiones de rendimiento, el sistema de disco se ha organizado como un RAID 0. Suponemos que disponemos de suficientes peticiones de acceso para saturar el ancho de banda de los discos.

b) **Calcula** el ancho de banda efectivo del sistema de disco.

c) **Calcula** el MTTF del sistema de disco.

En caso de que falle un disco en un RAID 0 hay que reemplazar el disco que ha fallado y recuperar el backup completo de la base de datos, hasta que no se haya recuperado toda la base de datos el sistema no estará disponible. Nuestro sistema de disco dispone de un disco adicional de forma que la sustitución del disco fallido por el nuevo sea inmediata y se pueda empezar a recuperar el backup tan pronto se detecta el fallo.

d) **Calcula** la disponibilidad del sistema de disco con la organización RAID 0

Dado que esta disponibilidad se considera insuficiente, se ha decidido evaluar la posibilidad de organizar los discos mediante un RAID 5. En nuestra base de datos se producen accesos de forma aleatoria, aunque se dispone de suficientes accesos para saturar el ancho de banda de todos los discos.

e) **Calcula** el ancho de banda efectivo del sistema si todos los accesos son lecturas.
Calcula el ancho de banda efectivo del sistema si todos los accesos son escrituras.

Sabemos que en nuestra base de datos se producen un 20% de escrituras.

f) **Calcula** el % de ancho de banda que se perderia respecto al RAID 0

En el sistema RAID 5 en caso de que falle un disco se puede recuperar ese disco ($MTTR_{disco}$) en 2 horas. En caso que falle el sistema de disco nuevamente hay que recuperar la base de datos completa del backup en cinta.

g) **Calcula** el MTTF del sistema.

h) **Calcula** la disponibilidad del sistema.

Problema 5. Problema de instrucciones (1 punto)

Los actuales procesadores CISC de la familia x86 traducen internamente las instrucciones de lenguaje máquina x86 a microoperaciones (que denominaremos uops). A continuación se presenta un código ensamblador y su correspondiente traducción a uops::

```
  movl $0, %ecx
loop:
  cmpl $1000000, %ecx
  jge fin
  movl x, %eax
  imull V(,ecx,4), %eax
  addl %eax, suma
  incl %ecx
  jmp loop
fin:
```

```
  movl %ecx <- 0
loop:
  cmpl %ecx - $1000000
  jge fin
  load %eax <- M[x]
  load %r1 <- M[%ecx*4+V]
  imull %eax <- %r1*%eax
  load %r1 <- M[suma]
  addl %r1 <- %r1 + %eax
  store M[suma] <- %r1
  addl %ecx <- %ecx + $1
  jmp loop
fin:
```

Sabemos que cada uop se codifica con 6 bytes. Además este procesador, que funciona a 2 GHz, incorpora una cache de uops (de nivel 0) que permite evitar la traducción reiterada del mismo código. En el código anterior se ha medido el ancho de banda de la cache de uops y se ha visto que este es de 30 Gbytes/s.

a) **Calcula** cual es el UPC (uops por ciclo) al que se ejecuta este código.

b) **Calcula** cual es el CPI (Ciclos Por Instrucción) al que se ejecuta el código x86 anterior.

c) **Calcula** el tiempo de ejecución del programa.

d) Con los datos de que dispones, ¿es posible calcular el ancho de banda efectivo de la cache de datos del procesador? Si la respuesta es sí, **calcula** el ancho de banda.

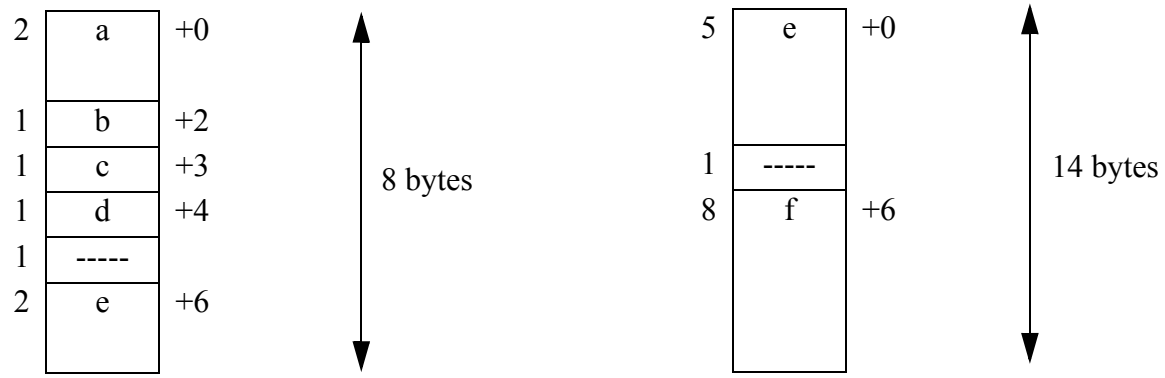
e) Con los datos de que dispones, ¿es posible calcular el ancho de banda efectivo de la cache de instrucciones (no la de uops) del procesador? Si la respuesta es sí, **calcula** el ancho de banda.

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

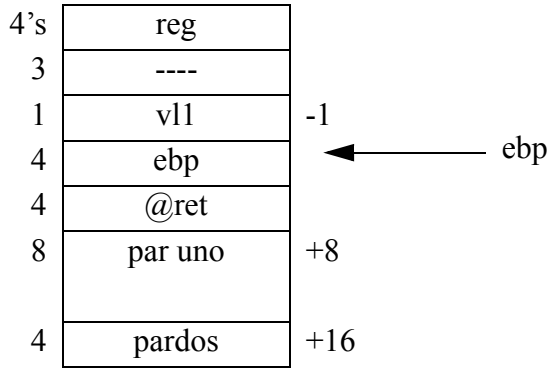
3er Control Arquitectura de Computadors **Curs 2010-2011 Q2**

Problema 1. (1 punto).

a) **Dibuja** como quedarían almacenadas en memoria las estructuras s1 y s2, indicando claramente los desplazamientos respecto al inicio y el tamaño de todos los campos.



b) **Dibuja** el bloque de activación de la función examen, indicando claramente los desplazamientos relativos al registro EBP necesarios para acceder a los parámetros y a las variables locales.



Problema 2. (3 puntos)

a) **Calcula** el numero medio de ciclos entre accesos, la probabilidad de acceder a memoria en un ciclo determinado, el CPI, el tiempo de ejecución del programa en la **CPU_{IDEAL}**.

tiempo medio entre acccessos = $15x10^9$ ciclos / $6x10^9$ accesos = **2,5 ciclos**
probabilidad acceso en un ciclo = 1/tiempo medio = 1/2,5 = **0,4**
CPI = $15x10^9$ ciclos / $5x10^9$ instrucciones = **3 c/i**
Texe = $15x10^9$ ciclos / $3x10^9$ Hz = **5 s**

b) **Calcula** los ciclos totales, el CPI, y el tiempo de ejecución cuando ejecutamos el programa con la cache directa.

ciclos = $15x10^9$ ciclos + 1 ciclos/acceso * $6x10^9$ accesos + 30 ciclos/fallo * $500x10^6$ fallos = $36x10^9$ ciclos
CPI = $36x10^9$ ciclos / $5x10^9$ instrucciones = **7,2 c/i**
Texe = $36x10^9$ ciclos / $3x10^9$ Hz = **12 s**

c) **Calcula** la potencia de conmutación en el modo de alto rendimiento. **Calcula** la potencia total en el modo de bajo consumo.

Potencia conmutación (AR) = 70W - 10W = 60W
Potencia conmutación = $C \cdot V^2 \cdot f$ -> Potencia conmutación (BC) = 60W / 3 = 20W

Potencia media (BC) = 20 W + 10 W = 30W

d) **Calcula** la energía consumida por el procesador con cache unificada ejecutando el programa de prueba, suponiendo que la CPU está en el modo de alto rendimiento.

Energía (AR) = 12 s · 70 W = 840 J

e) **Calcula** el tiempo que la CPU estará en modo bajo consumo (tiempo perdido por fallos de cache).

ciclos(fallo cache) = 30 ciclos/fallo * $500x10^6$ fallos = $15 \cdot 10^9$ ciclos
Tiempo = $15 \cdot 10^9$ ciclos / $3x10^9$ Hz = 5 s

f) **Calcula** la energía consumida por el procesador ejecutando el programa de prueba, suponiendo que la CPU pasa al modo de bajo consumo mientras se resuelve un fallo de cache. Calcula la potencia media en este último caso.

ciclos(fallo cache) = 10 ciclos/fallo * $500x10^6$ fallos = $5 \cdot 10^9$ ciclos
Tiempo = $5 \cdot 10^9$ ciclos / $1x10^9$ Hz = 5 s
ciclos(resto) = $21 \cdot 10^9$ ciclos
Tiempo = $21 \cdot 10^9$ ciclos / $3x10^9$ Hz = 7 s
Energía (BC) = 7·70 + 5·30 = 640 J
Potencia media (BC) = 640J/12s = 53,33 W

g) **Calcula** la probabilidad de que al realizar un acceso a memoria la cache esté ocupada.

la cache sólo puede estar ocupada durante un ciclo (por el acceso anterior)
probabilidad de acceso con cache ocupada = **0,4**

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

ciclos = $15x10^9$ ciclos + 1 ciclos/acceso * $6x10^9$ accesos * 0,4 + 30 ciclos/fallo * $500x10^6$ fallos = $32,4x10^9$ ciclos
CPI = $32,4x10^9$ ciclos / $5x10^9$ instrucciones = **6,48 c/i**
Texe = $32,4x10^9$ ciclos / $3x10^9$ Hz = **10,8 s**

i) **Calcula** los ciclos totales, el CPI, y el tiempo de ejecución cuando ejecutamos el programa en la CPU con la cache multibanco.

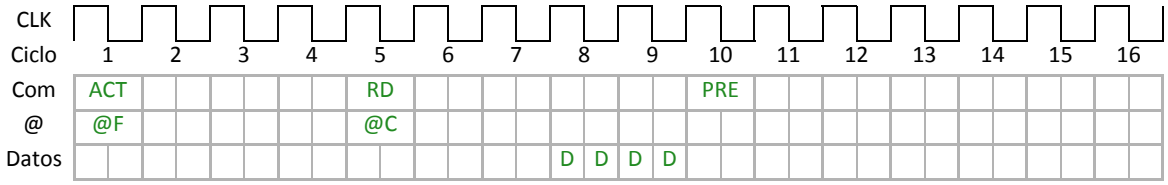
ciclos = $15x10^9$ ciclos + 1 ciclos/acceso * $6x10^9$ accesos * 0,1 + 30 ciclos/fallo * $500x10^6$ fallos = $30,6x10^9$ ciclos
CPI = 30,,$6x10^9$ ciclos / $5x10^9$ instrucciones = **6,12 c/i**
Texe = $30,6x10^9$ ciclos / $3x10^9$ Hz = **10,2 s**

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

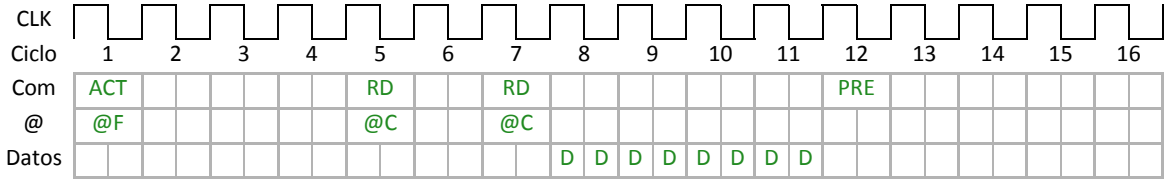
3er Control Arquitectura de Computadors **Curs 2010-2011 Q2**

Problema 3. (3 puntos)

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



b) **Rellena** el siguiente cronograma indicando la ocupación de los distintos recursos para leer un bloque de memoria de 64 bytes de forma que se haga lo más rápidamente posible.



c) **Explica** brevemente porque en este fragmento de código no importa si la cache es write allocate o write NO allocate.

No hay fallos de escritura (siempre se lee antes)

d) **Calcula** cuantos fallos de cache (de datos) se producen, cuantos bloques leemos de memoria principal y cuantos bloques escribimos.

8*1024 iteraciones / 8 elementos por bloque

1024 fallos
1024 bloques leidos
0 bloques escritos (no hay reemplazos)

e) **Calcula** el CPI del bucle con N = 8192

CiclosMem = 1024 fallos * 25 ciclos/fallo = 25600 ciclos
Instrucciones = 6 instrucciones/iteración * 8192 iteraciones = 49152 instrucciones
CPImem = 25600 ciclos / 49152 instrucciones = 0,52 ciclos/instrucción
CPI = CPIideal+CPImem = 1,5 ciclos/instrucción + 0,52 ciclos/instrucción = **2,02 ciclos/instrucción**

f) **Calcula** cuantos fallos de cache (de datos) se producen, cuantos bloques leemos de memoria principal y cuantos bloques escribimos.

2048 fallos

2048 bloques leidos

1024 bloques escritos (el vector es el doble de la cache y reemplazamos los 1024 primeros)

g) **Calcula** el CPI de la ejecución con N = 16384

CiclosMem = 1024 fallos * 25 ciclos/fallo + 1024 fallos * 50 ciclos/fallo= 76800 ciclos
Instrucciones = 6 instrucciones/iteración * 16384 iteraciones = 98304 instrucciones
CPImem = 76800 ciclos / 98304 instrucciones = 0,78 ciclos/instrucción
CPI = CPIideal+CPImem = 1,5 ciclos/instrucción + 0,78 ciclos/instrucción = **2,28 ciclos/instrucción**

h) ¿Cuantas instrucciones se ejecutan desde que el procesador accede por 1a vez el 1er dato de un bloque de cache hasta que accede el 1er dato del siguiente bloque 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 prefecth?

Instrucciones = 6 instrucciones/iteración * 8 iteraciones = 48 instrucciones
Ciclos = 48 instrucciones * 1,5 ciclos/instrucción = 72 ciclos
SI, en ambos casos (25 y 50 ciclos respectivamente)

i) **Calcula** el CPI del bucle con N = 16384 con el mecanismo de prefecth descrito anteriormente.

Se esconde la latencia => CPI ideal
CPI = 1,5 ciclos/instrucción

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

3er Control Arquitectura de Computadors **Curs 2010-2011 Q2**

Problema 4. (2 puntos)

a) **Calcula** el tiempo (en horas) que se tarda en recuperar un backup de la base de datos completa.

Tiempo = (3,6x10^{12} bytes / 50x10^{6} bytes/s)/3600s/h = 20 horas

b) **Calcula** el ancho de banda efectivo del sistema de disco.

Ancho Banda = 20 discos * 100 Mb/s/disco = 2 Gbytes/s

c) **Calcula** el MTTF del sistema de disco.

$MTTF_{raid0}$ = 100000 horas/disco / 20 discos = 5000 horas

d) **Calcula** la disponibilidad del sistema de disco con la organización RAID 0

Disponibilidad = MTTF/(MTTF+MTTR) = 5000h / 5020h = 0,996

e) **Calcula** el ancho de banda efectivo del sistema si todos los accesos son lecturas.
Calcula el ancho de banda efectivo del sistema si todos los accesos son escrituras.

En RAID5 podemos leer en parallelo de todos los discos
AnchoBandaLectura = 20 discos * 100 Mb/s/disco = 2 Gbytes/s

En RAID5 una escritura aleatoria requiere 4 accesos a disco
AnchoBandaEscritura = 20 discos * 100 Mb/s/disco / 4 = 500 Mbytes/s

f) **Calcula** el % de ancho de banda que se perderia respecto al RAID 0

Ancho de Banda medio = 0,8 * 2x10^{9} bytes/s + 0,2 * 500x10^{6} bytes/s = 1,7 Gbytes/s

Perdemos = 2 Gbytes/s - 1,7 Gbytes/s = 0,3Gbytes/s

0,3Gbytes/s / 2Gbytes/s = 0,15 => **15%**

g) **Calcula** el MTTF del sistema.

MTTF = $MTTFd^{2}$/(N*(N-1)*MTTRd) = (100000 horas/disco)^{2} / (20 discos * 19 discos * 2 horas) = 13,16x10^{6} horas

h) **Calcula** la disponibilidad del sistema.

Disponibilidad = 13157895/(13157895 + 20) = 0,9999985

Problema 5. (1 punto)

a) **Calcula** cual es el UPC (uops por ciclo) al que se ejecuta este código.

Uop/s = 30 Gbytes/s / 6 bytes/uop = $5x10^9$ uops/s
UPC = $5x10^9$ uops/s /$2x10^9$ ciclos/s = 2,5 uops/ciclo

b) **Calcula** cual es el CPI (Ciclos Por Instrucción) al que se ejecuta el código x86 anterior.

7 instr/10 uops = 0,7 instrucciones /uop
IPC = 0,7 instr/uop * 2,5 uops/ciclo = 1,75 instr/ciclo
CPI = 1/IPC = 0,571 ciclos/instrucción

c) **Calcula** el tiempo de ejecución del programa

uops dinámicas = 10 uops/iteracion * 10^6 iteraciones = $10x10^6$ uops
Texe = $10x10^6$ uops / $5x10^9$ uops/s = 2 ms

d) Con los datos de que dispones, ¿es posible calcular el ancho de banda efectivo de la cache de datos del procesador? Si la respuesta es sí, **calcula** el ancho de banda.

Bytes por Uop = 4 accesos (3 load + 1 store) por iteracion * 4 bytes/acceso / 10 uops/iteración = 1,6 bytes /uop
AnchoBanda Datos = 1,6bytes/uop * $5x10^9$ uops/s = 8 Gbytes/s

e) Con los datos de que dispones, ¿es posible calcular el ancho de banda efectivo de la cache de instrucciones (no la de uops) del procesador? Si la respuesta es sí, **calcula** el ancho de banda.

NO, no sabemos el tamaño de las instrucciones