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.

Problema 1. (2,5 puntos)

Tenemos una aplicación de la que hemos programado dos versiones, una **secuencial (VS)** y una **paralela (VP)**. Se ha ejecutado **VS** en un sistema con una única CPU que funciona a una frecuencia de 2 GHz. Este programa ejecuta $50x10^9$ instrucciones, realiza $20x10^9$ operaciones de punto flotante y tarda $80x10^9$ ciclos.

a) **Calcula** los MIPS y MFLOPS de **VS** en dicho sistema.

Texe = Ciclos / F= $80x10^9$ ciclos / 2GHz = 40 seg

MIPS = Instrucciones / Texe*10^6 = $50x10^9$ instrucciones / 40*10^6 = **1250 MIPS**

MFLOPS = OpsPF / Texe*10^6 = $20x10^9$ instrucciones / 40*10^6 = **500 MFLOPS**

Sabemos que la version **VP** tiene una parte de su código que es perfectamente paralelizable (speed-up = numero de CPUS).

b) **Calcula** el porcentaje (%) de ese código que deberíamos paralelizar para conseguir un speed-up de 2 en un multiprocesador con 5 CPUs.

Ley de amdahl:
2 = 1/(1-p + p/5) ---> p = 0.625 ---> **62.5%**

Este sistema multiprocesador esta formado por los siguientes componentes:

	Numero	MTTF (horas)	Consumo	Fase Secuencial	Fase Paralela
CPUs	5	1.000.000	50 W	1	5
Placa Base	1	1.000.000	40 W	1	1
Fuente de alimentación	1	250.000	0 W	1	1
Discos duros	4	200.000	5 W	4	0
DIMMs de memoria	20	1.000.000	2,5 W	4	20

c) **Calcula** el tiempo medio hasta fallo del sistema suponiendo que todos los componentes están activos.

MTTF = 1/(5/1e6+1/1e6+1/2.5e5+4/2e5+20/1e6) = **20.000 Horas**

Al ejecutar **VP** en el sistema multiprocesador el tiempo de ejecución se ha reducido a 10 segundos y observamos que la fase secuencial representa el 25% del tiempo de **VP** y la paralela el resto del tiempo. Para reducir el consumo se apagan los componentes que no se usan, la tabla anterior muestra también el consumo de cada componente y cuantos se usan en cada fase.

d) **Calcula** la energía consumida al ejecutar **VP**.

Pseq = 1*50+1*40+4*5+4*2,5 = 120 W
Ppar = 5*50+1*40+0*5+20*2,5 = 340 W

E = (0,25*Pseq+0,75*Ppar)*T = (0,25*120+0,75*340)*10 = **2850 Joules**

COGNOMS:

NOM:

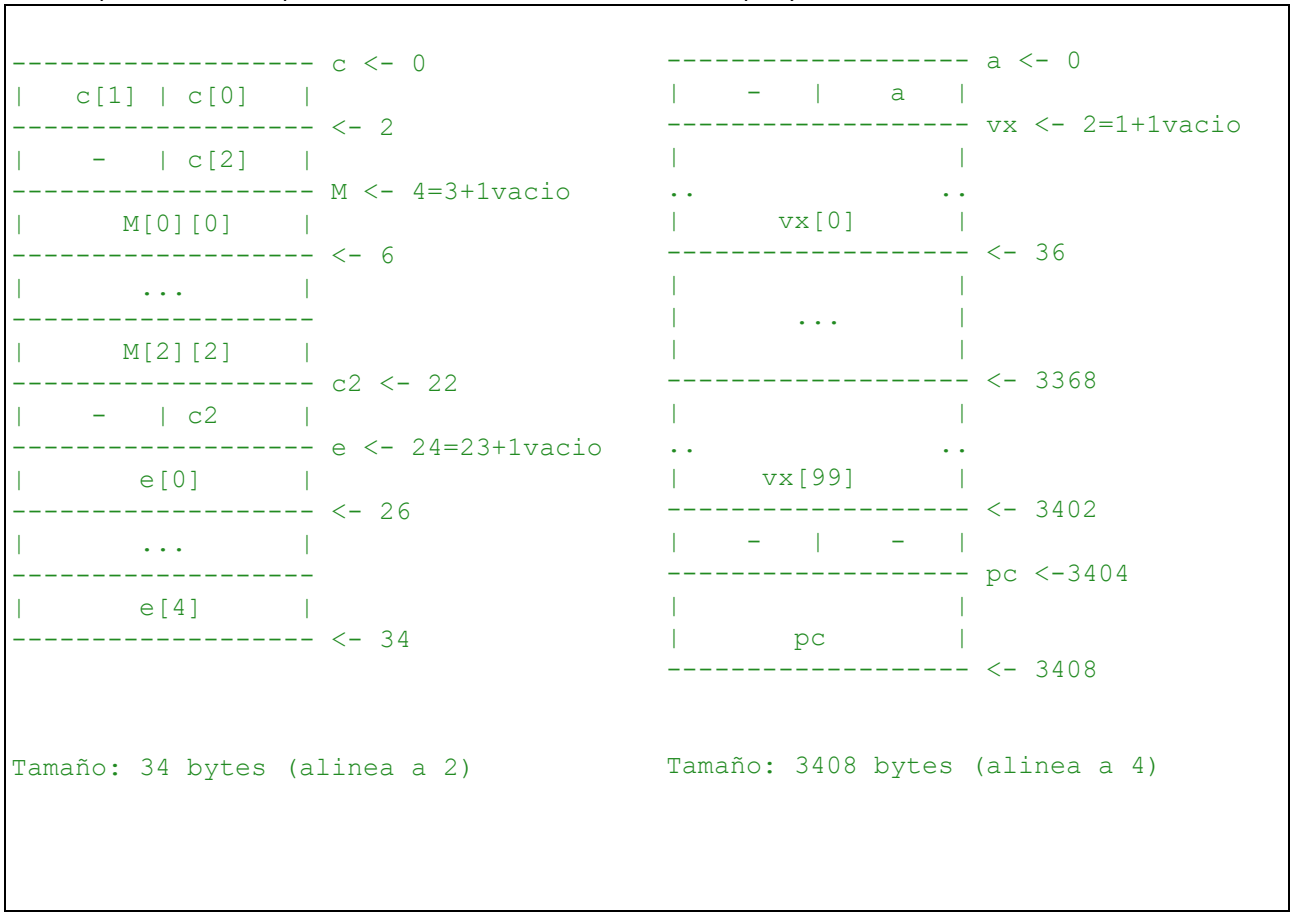
Problema 2. (2.5 puntos)

Dado el siguiente código escrito en C, que compilamos para un sistema linux de 32 bits:

```
typedef struct
{
  char c[3];
  short int M[3][3];
  char c2;
  short int e[5];
} sa;
```

```
typedef struct
{
  char a;
  sa vx[100];
  char *pc;
} sb;
```

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



b) **Escribe** UNA ÚNICA INSTRUCCIÓN que permita mover **k.vx[37].M[2][1]** al registro **%ax**, siendo **k** una variable de tipo **sb** cuya dirección está almacenada en el registro **%ecx**. Indica claramente la expresión aritmética utilizada para el cálculo de la dirección.

La expresión aritmética para calcular la dirección del operando es: @k+2+(37*34)+4+(2*3+1)*2 = %ecx+1278

y la instrucción es: **movw 1278(%ecx), %ax**

COGNOMS:

NOM:

Problema 3. (2.5 puntos)

Dado el siguiente código escrito en C:

```
int examen(short a, char b, int v[10], short M[4][4]) {
  int ii;
  char cc;
  short aa;
  short *matriz;
  int *vector;
      ...
  return v[9];
}
```

a) **Dibuja** el bloque de activación de la rutina examen, indicando claramente los desplazamientos respecto a **%ebp** y el tamaño de todos los campos.

```
     --------------------  ii<- ebp-16
4    |        ii          |
     --------------------  cc<- ebp-12
4    |    aa   | - | cc   | aa<- ebp-10
     ---------------------- *matriz<- ebp-8
4    |     *matriz        |
     ---------------------- *vector<- ebp-4
4    |     *vector        |
     ----------------------   <- ebp
4    |     ebp old        |
     ----------------------
4    |        ret         |
     ---------------------- <- ebp+8
4    |  --  -- |    a     |
     ---------------------- <- ebp+12
4    |  --  --  --  |  b |
     ---------------------- <- ebp+16
4    |        @v          |
     ---------------------- <- ebp+20
4    |        @M          |
     ---------------------- <- ebp+24
```

b) **Traduce** a ensamblador del x86 la instrucción `return v[ii]` USANDO EL MÍNIMO NÚMERO DE INSTRUCCIONES y suponiendo que la subrutina ha usado los registros %eax, %ebx, %ecx y %edx (esto no afecta al apartado anterior).

```
movl 16(%ebp), %ecx       ; %ecx <- @V
movl -16(%ebp), %eax      ; %ecx <- ii
movl (%ecx,%eax,4), %eax  ; %eax <- V[ii]
popl %ebx                 ; restaurar registros. Sólo es necesario guardar ebx
movl %ebp, %esp           ; deshacer enlace dinámico
popl %ebp
ret
```

COGNOMS:

NOM:

Problema 4. (2.5 puntos)

Tenemos un procesador de muy bajo rendimiento que puede usar memoria virtual. Sabemos que el CPIideal (asumiendo que todos los accesos a memoria tardasen 1 ciclo) de un programa es de 7 ciclos. El programa tarda 1.6s en ejecutarse, ejecuta 1x10^9 instrucciones y realiza un total de 1.3x10^9 accesos a memoria. La memoria principal de dicho procesador tiene un tiempo de acceso real de 10 ciclos y suponemos que no hay fallos de página.

a) **Calcula** cuantos accesos a datos y cuantos accesos a instrucciones realiza el programa. **Indica** también cual es el CPI real del programa si no tenemos en cuenta la memoria virtual (no hay accesos a la tabla de páginas).

1x10^9 accesos a instrucciones

0.3x10^9 accesos a datos

CPI = 7 ciclos + 9 ciclos de penalización de ins + 9*0.3 ciclos de penalización de datos = 18.7 c/i

b) **Calcula** el CPI del mismo programa si usáramos memoria virtual mediante paginación sin TLB.

1 acceso a memoria implicaría 2 accesos, 1 a la tabla de páginas y otro acceso al dato/instrucción en si. 2 accesos son 20 ciclos así que la penalización sería de 19 ciclos por acceso

CPI = 7 ciclos + 19 ciclos de penalización de ins + 19*0.3 ciclos de penalización de datos = 31.7 c/i

c) **Calcula** el CPI real del sistema con paginación si usáramos un TLB con un 99% de aciertos. El TLB se accede siempre y tiene 1 ciclo de latencia..

CPI = 7ciclos + 10 ciclos (penalización acierto TLB ins) * 0.99 + 20 ciclos (penalización fallo TLB ins) * 0,01 + 0,3 * (10 ciclos (penalización acierto TLB datos) * 0.99 + 20 ciclos (penalización fallo TLB datos) * 0,01) = 20.13 c/i

d) **Calcula** el tamaño de la tabla de páginas de un solo nivel de 1 programa si el sistema tuviera 16 bits de direcciones lógicas, páginas de 8 KB y 15 bits de direcciones físicas.

8 entradas * (2 bits PPN por entrada + 1 bit presencia + 1 bit modificación) = 32 bits

Finalmente, al procesador anterior le hemos incorporado un TLB que asumiremos que acierta siempre y una cache unificada 3-asociativa con lineas de 16 bytes. En dicho sistema ejecutamos la siguiente función en alto nivel compilada sin optimizaciones:

```
int sumarvector(int v[]) {
   int i, suma;
   for (i=0; i<50000;i++)
      suma += v[i];
     return suma;
```

e) Sabiendo que al ejecutar la función tanto las instrucciones como las variables locales están almacenadas en la cache unificada, ¿**cuantos** fallos tendrá al ejecutarse la función anterior? **Razona** el resultado.

Como la cache es de 3 vías sabemos que variables locales, v[] y las instrucciones no se interferirán entre ellas. Solo tendremos un fallo de v[i] por cada 4 accesos (también puede haber uno extra si v[] no está alineado) -> 12500 o 12501 fallos.

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.

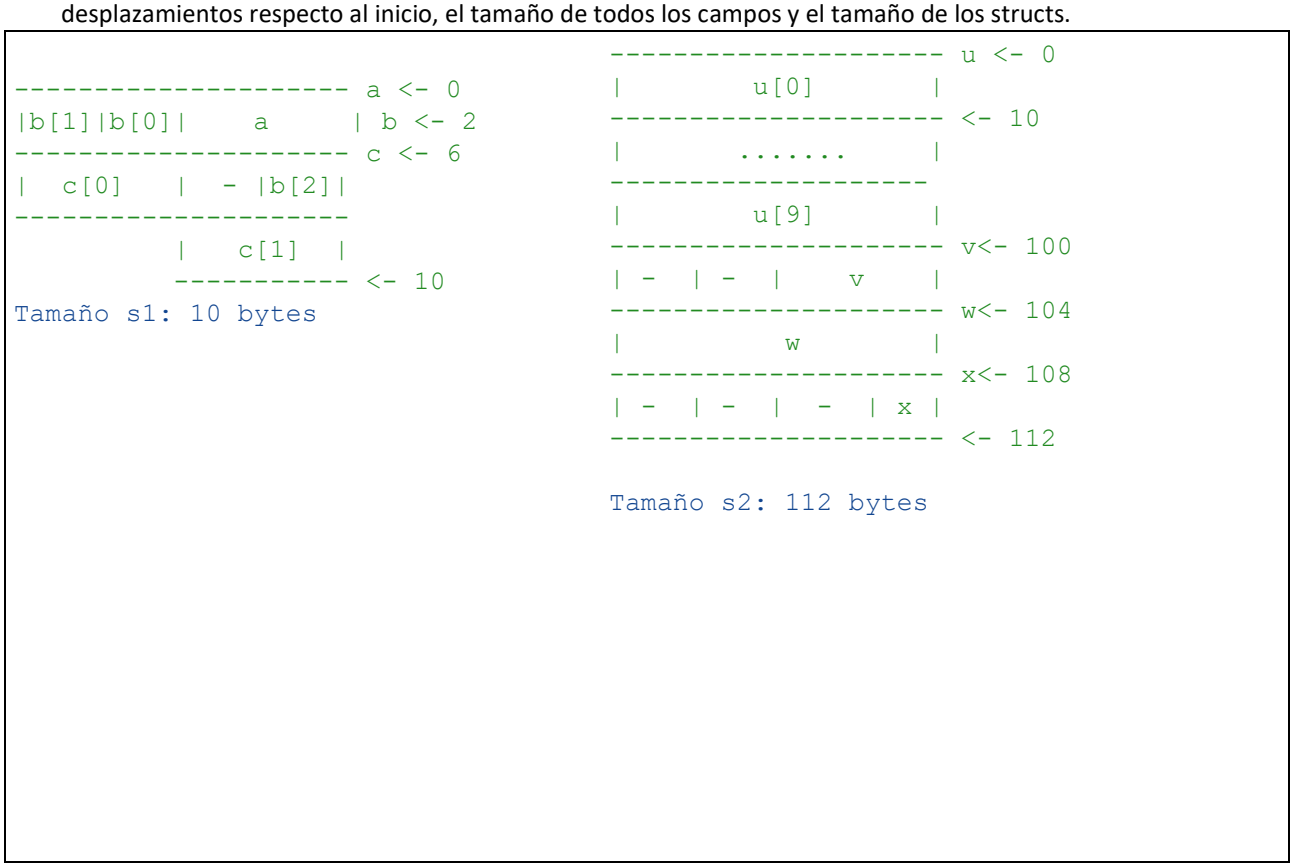
Problema 1. (5 puntos)

Dado el siguiente código escrito en C, que compilamos para un sistema linux de 32 bits:

```
typedef struct {
  short a;
  char b[3];
  short c[2];
  } s1;
```

```
typedef struct {
  s1 u[10];
  short v;
  int w;
  char x;
  } s2;
```

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



b) **Escribe** UNA ÚNICA INSTRUCCIÓN que permita mover **x.u[8].c[0]** a los 16 bits de menor peso del registro **%edx**, siendo **x** una variable de tipo **s2** cuya dirección está almacenada en el registro **%ecx**.
Indica claramente la expresión aritmética utilizada para el cálculo de la dirección.

La expresión aritmética para calcular la dirección del operando es: @x+10*8+6, por lo tanto %ecx+86

La instrucción es: **movw 86(%ecx), %dx**

Dado el siguiente código escrito en C, que compilamos para un sistema linux de 32 bits:

```
int examen(char a, char b[3][3], short c) {
  short x;
  char y[3][3];
  short z;
  int w;
  . . .
  w=examen(y[2][2],y,z);
  . . .
}
```

c) **Dibuja** el bloque de activación de la rutina examen, indicando claramente los desplazamientos respecto a **%ebp** y el tamaño de todos los campos.

```
examen
----------------------------- x<- ebp-20
|y[0,1] y[0,0]|      x      | y<- ebp-18
----------------------------- <- ebp-16
|y[1,2] y[1,1] y[1,0] y[0,2]|          |
----------------------------- <- ebp-12
|  -    | y[2,2]y[2,1]y[2,0] |
----------------------------- z<- ebp-8
|  --     --    |    z        |
----------------------------- w<- ebp-4
|             w               |
-------------------------------
|          ebp                | <- ebp
-------------------------------
|          RET                |
-------------------------------
|  --   --   --        |  a    | a <- ebp+8
-------------------------------
|             @b              | @b <- ebp+12
-------------------------------
|  --   --      |     c       | c <- ebp+16
----------------------------- <- ebp+20
```

d) **Traduce** a ensamblador x86 la instrucción `w=examen(y[2][2],y,z);` que se encuentra en el interior de la subrutina, usando el mínimo número de instrucciones.

```
pushl -8(%ebp)
leal -18(%ebp), %eax
pushl %eax
pushl -10(%ebp)
call examen
addl $12, %esp
movl %eax, -4(%ebp)
```

COGNOMS:

NOM:

Problema 2. (5 puntos)

En un ordenador en el que tenemos instalado el entorno usado en el laboratorio de AC, hemos medido que un programa de 1000 instrucciones ensamblador se ha ejecutado en 2 segundos usando $6x10^9$ ciclos y ha ejecutado $4,8x10^9$ instrucciones y $2,4x10^8$ operaciones de coma flotante que debido a la falta de hardware específico ejecutan 10 instrucciones cada una.

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

CPI = 6x10^9 ciclos / 4,8x10^9 instrucciones = 1,25 c/i

Frec = 6x10^9 ciclos / 2 segundos = 3x10^9 ciclos/segundo = 3 GHz

b) **Calcula** los MIPS y MFLOPS a los que se ejecuta el programa.

MIPS=4,8x10^9 instrucciones/ 2 segundos / 10^6 = 2400 MIPS

MFLOPS= 2,4x10^8 operaciones / 2 segundos / 10^6 = 120 MFLOPS

El tiempo de ejecución usado en el primer apartado 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 del sistema tarda 8 milisegundos, mientras que si los datos estuviesen en un disco SSD cada acceso tardaría 10 microsegundos.

c) **Calcula** la ganancia total en el programa que se obtendría con el cambio de tipo de disco.

Ganancia Disco = 8x10-3 s/a / 10x10-6 s/a = 800

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

A pleno rendimiento, la 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 0,8 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 25 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.

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

$P = Pc + Pf = V^2*F*C + I_f*V$

(1,6 V)^2 * 3x10^9 Hz * C + 1,6 V * I = 120 W (alto rendimiento)
(1 V)^2 * 0,8x10^9 Hz * C + 1 V * I = 25 W (bajo consumo)

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

C = 12,5 nF I = 15 A

Pare el resto del problema tendremos en cuenta solo la fase de calculo del programa, es decir solo tiempo de CPU (usuario + sistema).

e) **Calcula** la ganancia en energía que tendría el sistema si ejecutara el programa en el modo de bajo consumo en vez del modo de alto rendimiento suponiendo que el CPI medio no varía.

N y CPI no varian , el tiempo de Bc sera proporcional a la variación de frecuencia
Tiempo Bc = 2s * 3GHz/0,8Gz = 7,5 s

Ear = 120W * 2s = 240 J
Ebc = 25W * 7,5 s = 187,5

G = 240/187,5 = 1,28

Este procesador tiene direcciones físicas de 32 bits, una cache de datos de primer nivel (L1) 4-asociativa con tamaño de bloque 64 bytes y política de escritura *Copy Back + Write Allocate*. Las etiquetas (TAGS) de la cache son de 18 bits.

f) **Calcula** el numero de bloques (líneas) de la cache.

64 bytes/bloque -> 6 bits de offset (byte)
32 bits (@) - 18 bits (TAG) - 6 bits (offset) = 8 bits (conjunto)
256 conjuntos * 4 (asociatividad) = 1024 bloques

El procesador dispone ademas de un TLB que se accede en paralelo a L1.

g) **Calcula** el tamaño mínimo que pueden tener las páginas de memoria virtual para que sea posible el acceso paralelo a cache y TLB.

256 bloques / vía * 64 bytes / bloque -> 16Kbytes /vía
Tamaño Página >= Tamaño vía
Página >= 16K bytes

Para la fase de cálculo el tiempo medio de acceso a memoria (Tma) es de 1,3 ciclos. En caso de acierto en L1 el tiempo de acceso es de un ciclo. En caso de fallo hay una penalización (Tpf) de 10 ciclos adicionales si el bloque reemplazado tiene el *dirty bit* D=0 y de 20 ciclos si el bloque reemplazado tiene D=1. Sabemos que en media el 50% de los bloques tiene D=1. La influencia de los fallos de TLB y de los fallos de página es despreciable.

h) **Calcula** la tasa de fallos de la cache L1.

tpf = 0,5*10 + 0,5*20 = 15
Tma = Tsa + m*tpf
1,3 = 1 + m*15 - > m = 2%

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.

Problema 1. (2,5 puntos)

Un programa secuencial (PS) ejecuta $4x10^9$ instrucciones dinámicas que realizan $3x10^9$ operaciones de punto flotante. El programa PS tarda en ejecutarse $10x10^9$ ciclos en un procesador C que funciona a una frecuencia de 5 GHz.

a) **Calcula** el CPI y el tiempo de ejecución en segundos (Texec) del programa PS.

CPI = $10x10^9$ ciclos / $4x10^9$ instrucciones = **2,5 ciclos / instrucción**

Texe = Ciclos / F= $10x10^9$ ciclos / 5GHz = **2 seg**

El procesador C tiene una corriente de fuga de 20 A, se alimenta a un voltaje de 2 V y tiene una carga capacitiva equivalente de 5 nF. El consumo debido a cortocircuito es despreciable.

b) **Calcula** la energía consumida al ejecutar el programa PS en el procesador C.

P = Pfuga + Pconmut = $I*V + C*V^2*F$ = 20 A * 2 V + 5 nF * $(2\ V)^2$ * 5 GHz = 140 W

E = P*t = 140W * 2 s = **280 Joules**

Hemos programado una variante paralela de PS (llamada PP) que tiene un 90% del código paralelo y un 10% secuencial.

c) **Calcula** el número (N) de procesadores idénticos a C que son necesarios para conseguir un speed-up de 5 en el programa PP respecto PS.

Ley de amdahl:
5 = 1/(0,1 + 0,9/N) ---> **N = 9**

d) **Calcula** los MFLOPS de PP en dicho sistema paralelo (N procesadores C).

TexeP = Texe/5 = 0,4s

MFLOPS = OpsPF / TexeP*10^6 = $3x10^9$ opsPF / 0,4*10^6 = **7500 MFLOPS**

Una CPU para móvil dispone de 1 procesador A (alto rendimiento) y 4 procesadores B (bajo consumo). Cada procesador B tiene un 75% del rendimiento de A, pero consume solo un 30% de la potencia. La CPU está limitada por la potencia disipada por lo que simultáneamente solo puede funcionar o un procesador A o los 4 procesadores B, pero nunca todos a la vez. PS se ejecuta exclusivamente sobre el procesador A. Mientras que para PP, la parte secuencial se ejecuta en el procesador A y la parte paralela en los 4 procesadores B.

e) **Calcula** la ganancia en energía de PP respecto PS en esta CPU.

Parte secuencial: GEs = 1
Parte paralela: Tb = 1/(0,75*4) Ta; Pb = 0,3*4 Pa -> Eb = (0,30*4)/(0,75*4) Ea = 0,4 Ea -> GEp = 2,5
G (energia) = 1/(0,1 + 0,9/2,5) = **2,17**

COGNOMS:

NOM:

Problema 2. (2.5 puntos)

Dado el siguiente código escrito en C que compilamos para un sistema linux de 32 bits:

```
typedef struct {
 short int v1[7];
 int g;
 char c;
 short int v2[4];
} sa;
```

```
typedef struct {
 sa tabla[50];
 short int *h;
} sb;
```

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

sa

```
----------------  <---- 0
|    v1[0]     |
----------------  <---- 2
|       ...    |
----------------
|    v1[6]     |
----------------  <---- 14 + 2 vacio = 16
|      g       |
----------------  <---- 20
|      c       |
----------------  <---- 21 + 1 vacio = 22
|    v2[0]     |
----------------  <---- 24
|       ...    |
----------------
|    v2[3]     |
----------------  <---- 30 + 2 vacio = 32
```

Tamaño de sa = 32 bytes

sb

```
----------------  <---- 0
|   tabla[0]   |
----------------  <---- 32
|       ...    |
----------------
|   tabla[49]  |
----------------  <---- 1600
|      *h      |
----------------  <---- 1604
```

Tamaño de sb = 1604 bytes

b) **Escribe** UNA ÚNICA INSTRUCCIÓN en ensamblador que traduzca la instrucción en C: **y.tabla[36].c = 'A'**, siendo **y** una variable de tipo **sb** cuya dirección está almacenada en el registro **%ebx**. **Indica** claramente la expresión aritmética utilizada para el cálculo de la dirección.

La expresión aritmética para calcular la dirección es @ini_y + 32*36 + 20 = @ini_y + 1172, por lo tanto la expresión es: %ebx + 1172

La instrucción es: **movb $'A', 1172(%ebx)**

COGNOMS:

NOM:

Problema 3. (2.5 puntos)

Dado el siguiente código escrito en C, que compilamos para un sistema linux de 32 bits:

```
int funcion(char x, short int y[3]) {
  int a;
  int *b;
  short int c[3];
  char d;
  . . .
  a = funcion(x, c) + *b;
  . . .
}
```

a) **Dibuja** el bloque de activación de la rutina funcion, indicando claramente los desplazamientos respecto a **%ebp** y el tamaño de todos los campos.

```
funcion
----------------------------- <- ebp-16
|              a              |
----------------------------- <- ebp-12
|             *b              |
----------------------------- <- ebp-8
|    c[1]     |    c[0]       |
-----------------------------
| --- |   d   |    c[2]       | <- d en ebp-2
----------------------------- <- ebp
|             EBP             |
-----------------------------
|             RET             |
----------------------------- <- ebp+8
| ---------------- |   x      |
----------------------------- <- ebp+12
|             *y              |
-----------------------------
```

b) **Traduce** a ensamblador x86 la sentencia `a = funcion(x,c) + *b;` que se encuentra en el interior de la subrutina, sabiendo que el valor de la expresión `*b` se encuentra en el registro **%edx** justo antes de la llamada. Se valorará usar el mínimo número de instrucciones y el mínimo número de accesos a memoria.

```
movl  %edx, %ebx          #  salvamos *b en un registro reguro
leal  -8(%ebp), %eax
pushl %eax                # c
pushl 8(%ebp)             # x
call  funcion
addl  $8, %esp
addl  %ebx, %eax          # valor retorno + *b
movl  %eax, -16(%ebp)     # a <- funcion2(x,c) + *b
```

Otras alternativas podrían ser guardar y restaurar %edx en la pila, o volver a leer *b.

COGNOMS:

NOM:

Problema 4. (2.5 puntos)

Disponemos de un sistema de cómputo que utiliza memoria virtual sin memoria cache ni TLB. El sistema funciona a 1GHz, tiene un CPI ideal (suponiendo que cada acceso a memoria principal se resuelve en 1 ciclo) de 8 ciclos y supondremos una tabla de páginas uninivel que es capaz de resolver todas las traducciones en un solo acceso. Supondremos también que no hay fallos de página.

Supongamos el siguiente código que se ejecuta en 0,424s. Las etiquetas "V" y "bucle" están alineadas a 1MB:

```
        movl $1000000, %ecx
        movl $0, %eax
bucle:  addl V(,%ecx,4), %eax
        decl %ecx
        jnz bucle
```

a) **Calcula** el número de accesos físicos a memoria principal producidos al ejecutar el bucle. ¿**Cuál** es el tiempo de acceso a la memoria principal?

Instrucciones ejecutadas = 3 * 1000000 = 3000000
Accesos a memoria = 4 (3 ins + 1 datos) * 2 (1 tabla páginas + 1 @física) * 1000000 = 8000000

Ciclos programa = 3000000 * 8 + 8000000 * (ta-1) = 424000000 = 24 000000 + 50*8 000000
-> ta -1 = 50 -> ta = 51 ciclos

Vista la ineficiencia del sistema decidimos incorporar al procesador un TLB y una memoria cache. La memoria cache que pensamos usar tiene líneas de 32 bytes. Tanto el TLB como la cache son unificados para instrucciones y datos.

b) Teniendo en cuenta que el TLB es completamente asociativo con reemplazo LRU, ¿**cuál** es el tamaño mínimo de TLB que nos permitiría tener el máximo número de aciertos en el bucle anterior?

2 entradas, 1 para datos y otra para instrucciones

Finalmente decidimos utilizar 2 caches (y 2 TLBs) separados de instrucciones y datos. Cada TLB tiene 8 entradas y cada cache 128 lineas. La cache de instrucciones es de acceso directo y la de datos 4 asociativa. Nuestro sistema con TLB y cache tiene un CPI ideal de 6 ciclos. Las páginas del sistema son de 4 KB.

c) ¿**Cuántos** fallos de cache dará en total el bucle anterior?

1 instrucciones y 1000000 / 8 de datos -> 125001

d) ¿**Cuántos** fallos de TLB dará en total el bucle anterior?

1 de instrucciones y 1000000 / 1024 -> 1 + 977 = 978

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.

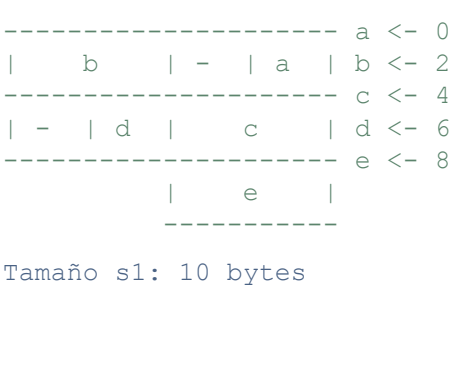
Problema 1. (5 puntos)

Dado el siguiente código escrito en C, que compilamos para un sistema linux de 32 bits:

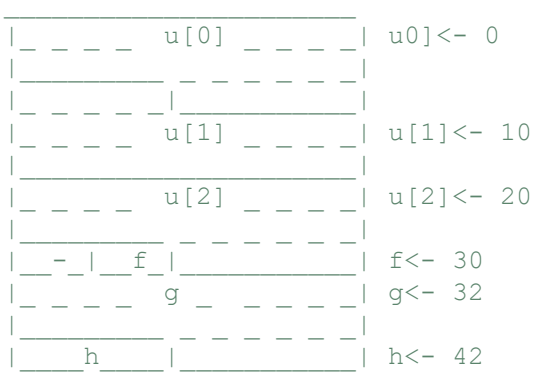
```
typedef struct {
 char a;
 short b;
 short c;
 char d;
 short e;
} s1;
```

```
typedef struct {
 s1 u[3];
 char f;
 s1 g;
 short h;
} s2;
```

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



Tamaño s1: 10 bytes



Tamaño s2: 44 bytes

b) **Escribe** UNA ÚNICA INSTRUCCIÓN que permita mover **z.u[2].c** al registro **%ax**, siendo **z** una variable de tipo **s2** cuya dirección está almacenada en el registro **%ebx**. **Indica** claramente la expresión aritmética utilizada para el cálculo de la dirección.

La expresión aritmética para calcular la dirección del operando es @ini_z + 10*2 + 4 = @ini_z + 24, por lo tanto %ebx + 24

La instrucción es: **movw 24(%ebx), %ax**

Dado el siguiente código escrito en C:

```
int examen(short a, char b, int v[10], short M[4][4]) {
  int ii;
  short aa;
  short *matriz;
  int *vector;
      ...
  return v[ii];
}
```

c) **Dibuja** el bloque de activación de la rutina examen, indicando claramente los desplazamientos respecto a **%ebp** y el tamaño de todos los campos.

```
      --------------------  ii<- ebp-16
  4   |         ii        |
      --------------------  aa<- ebp-12
  4   |    --   |   aa    |
      ---------------------- *matriz<- ebp-8
  4   |     *matriz       |
      ---------------------- *vector<- ebp-4
  4   |     *vector       |
      ----------------------   <- ebp
  4   |     ebp old       |
      ----------------------
  4   |        ret        |
      ---------------------- <- ebp+8
  4   |  --  -- |    a    |
      ---------------------- <- ebp+12
  4   |  --  --  --  | b |
      ---------------------- <- ebp+16
  4   |        @v         |
      ---------------------- <- ebp+20
  4   |        @M         |
      ---------------------- <- ebp+24
```

d) **Traduce** a ensamblador del x86 la instrucción `return v[ii]` USANDO EL MÍNIMO NÚMERO DE INSTRUCCIONES y suponiendo que la subrutina ha usado los registros %eax, %ebx, %ecx y %edx.

```
movl 16(%ebp), %ecx      ; %ecx <- @V
movl -16(%ebp), %eax     ; %ecx <- ii
movl (%ecx,%eax,4), %eax ; %eax <- V[ii]
popl %ebx                ; restaurar registros. Sólo es necesario guardar ebx
movl %ebp, %esp          ; deshacer enlace dinámico
popl %ebp
ret
```

COGNOMS:

NOM:

Problema 2. (5 puntos)

Un sistema de vigilancia analiza imágenes, provenientes de una cámara, cada T milisegundos, por lo que el procesado de cada imagen debe realizarse en un tiempo igual o inferior a T. Se ha calculado que un procesador P funcionando a una frecuencia F es suficiente para procesar una imagen en T milisegundos. Un análisis de rendimiento y consumo de energía muestra que la rutina HFR (Human Face Recognition) consume el 60% del tiempo y el 80% de la energía necesarios para procesar una imagen.

Se ha decidido incorporar un acelerador hardware (AH) al sistema. AH realiza el procesado HFR vía hardware pero consumiendo la cuarta parte de la energía respecto la ejecución de la rutina en P. El resto del código, que llamaremos PNA (parte no acelerada) se sigue ejecutando en P y por tanto no se beneficia del acelerador.

a) **Calcula** la ganancia en energía del sistema con acelerador (P+AH) respecto al sistema P.

Ley de amdahl:
G = 1/(0,2 + 0,8/4) ---> **G = 2,5**

El acelerador AH, ademas de consumir menos energía, también realiza el procesado HFR en menos tiempo. Sabemos que el tiempo de ejecución de PNA en P es proporcional al tiempo de ciclo. Dado que no hay ningún beneficio en procesar una imagen en menos de T milisegundos, podemos reducir la frecuencia de P de forma que la parte no acelerada tarde más (siempre que el conjunto no supere T milisegundos).

b) **Calcula** el speedup mínimo que debería producir AH respecto la rutina HFR para que podamos reducir la frecuencia de P a la mitad sin que el tiempo de procesado de una imagen sea superior a T.

Ley de amdahl generalizada:
El speedup global es 1 (tenemos que tardar lo mismo que sin acelerador).
La parte no acelerada tarda el doble (mitad de frecuencia)
1 = 1/(0,4/0,5 + 0,6/x) ---> **X = 3**

El procesador P funciona a un frecuencia de 1 GHz. La parte no acelerada (PNA) que corre sobre P ejecuta 6 millones de instrucciones, realiza 3 millones de operaciones de punto flotante y tarda 9 millones de ciclos.

c) **Calcula** el CPI y el tiempo de ejecución en milisegundos (Texec) de PNA en P.

CPI = $9x10^6$ ciclos / $6x10^6$ instrucciones = **1,5 ciclos / instrucción**

Texe = Ciclos / F= $9x10^6$ ciclos / 1GHz = **9 ms**

d) **Calcula** los MIPS y MFLOPS de PNA en dicho sistema.

MIPS = Instrucciones / Texe*10^6 = $6x10^6$ instrucciones / $9*10^3$ = **667 MIPS**

MFLOPS = OpsPF / Texe*10^6 = $3x10^6$ opsPF / $9*10^3$ = **333 MFLOPS**

El procesador P dispone de una cache con una política de escritura *Copy Back* y *Write Allocate*. En caso de acierto en la cache el tiempo de acceso es de 1 ciclo. En caso de fallo, hay una penalización de 60 ciclos para reemplazar un bloque NO modificado y de 120 ciclos para reemplazar un bloque modificado.

PNA realiza 8 millones de accesos a memoria y tiene una tasa de fallos (miss) del 10%. Sabemos que el 15% de los accesos son escrituras y que la probabilidad de que un bloque haya sido modificado en cache es del 20%.

e) **Calcula** el tiempo medio de acceso a memoria (Tma) en ciclos de la parte no acelerada (PNA).

Tma = Tsa+m*(Pm*Tpfm + Pn*Tpfn) = 1 + 0,10*(0,20*120+0,8*60) = **8,2 ciclos/acceso**

Leer o escribir un bloque en memoria principal consume 100 nanoJoules (nJ).

f) **Calcula** el consumo total de energía de la memoria principal causada por los fallos de cache.

fallo que reemplaza bloque NO modificado genera 1 acceso a MP (lectura bloque)
fallo que reemplaza bloque modificado genera 2 accesos a MP (escritura bloque + lectura bloque)
AccesosMP = accesos * m * (Pn*1 + Pm *2) = $8x10^6$ * 0,1* (0,8*1 + 0,2*2) = $0,96x10^6$accesos a MP

E = $0,96x10^6$a * 100 nJ = **96 mJoules**

El procesador P genera direcciones lógicas de 36 bits y direcciones físicas de 24 bits. La jerarquía completa de memoria está compuesta por un TLB, la memoria cache y la memoria principal. El TLB tiene 4 entradas y es completamente asociativo. La cache tiene un tamaño de 64 Kbytes, líneas de 64 bytes y es 4-asociativa. El tamaño de página del sistema es de 4 KBytes. El TLB se accede antes que la cache, por lo que la cache se indexa con direcciones físicas.

g) **Calcula** el número de líneas, vías y conjuntos que tiene la cache. **Especifica claramente** cómo has realizado los cálculos.

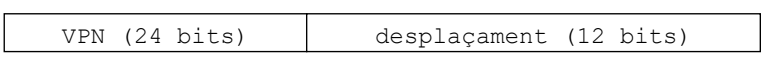
64*1024 bytes / 64 bytes/linea = **1024 líneas**
4-asociativa -> **4 vias**
1024 lineas / 4 lineas/conjunto = **256 conjuntos**

El procesador lanza un acceso a la dirección lógica 0xEFABCD012 y sabemos que el contenido del TLB es:

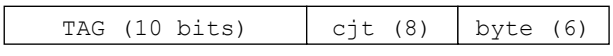
VPN	PPN
0xFABCD0	0xA00
0xBCD012	0xB01
0xEFABCD	0xC02
0xABCD01	0xD03

h) **Indica** a qué dirección física se accede, en qué conjunto de la cache se encuentra el dato y cuál es la etiqueta guardada en memoria cache. **Justifica** la respuesta.

VPN = 0xEFABCD -> PPN = 0xC02 **@Fisica = 0xC02012**



Conjunto = 0x80 TAG = 0x300



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.

Problema 1. (2,5 puntos)

Un programa P ejecuta $5x10^9$ instrucciones dinámicas y realiza $3x10^9$ operaciones de punto flotante. El programa P tarda en ejecutarse $8x10^9$ ciclos en un procesador C que funciona a una frecuencia de 4 GHz.

a) **Calcula** el CPI y el tiempo de ejecución en segundos (Texec) del programa P.

CPI = $8x10^9$ ciclos / $5x10^9$ instrucciones = **1,6 ciclos / instrucción**

Texe = Ciclos / F= $8x10^9$ ciclos / 4GHz = **2 seg**

El procesador C tiene una corriente de fuga de 10 A, se alimenta a un voltaje de 2 V y tiene una carga capacitiva equivalente de 5 nF. El consumo debido a cortocircuito es despreciable.

b) **Calcula** la energía consumida al ejecutar el programa P en el procesador C.

P = Pfuga + Pconmut = $I*V + C*V^2*F$ = 10 A * 2 V + 5 nF * $(2\ V)^2$ * 4 GHz = 100 W

E = P*t = 100W * 2 s = **200 Joules**

El programa P tiene un 80% del código que es perfectamente paralelizable.

c) **Calcula** el número (N) de procesadores idénticos a C que son necesarios para conseguir un speed-up de 4 en el programa P.

Ley de amdahl:
4 = 1/(0,2 + 0,8/N) ---> **N = 16**

d) **Calcula** los MFLOPS de P en dicho sistema paralelo (N procesadores C).

TexeP = Texe/4 = 0,5s

MFLOPS = OpsPF / TexeP*10^6 = $3x10^9$ opsPF / 0,5*10^6 = **6000 MFLOPS**

En un sistema paralelo con 25 procesadores idénticos a C, cada procesador tiene un MTTF de 10 millones de horas. El resto del sistema (sin contar las CPUs) tiene un MTTF de 100.000 horas.

e) **Calcula** el tiempo medio hasta fallo (MTTF) del sistema completo.

MTTF = 1/(25/10.000.000+1/100.000) = **80.000 horas**

COGNOMS:

NOM:

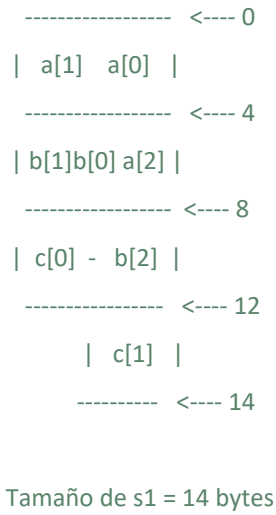
Problema 2. (2.5 puntos)

Dado el siguiente código escrito en C que compilamos para un sistema linux de 32 bits:

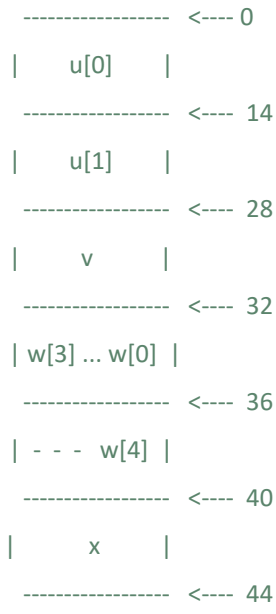
```
typedef struct {
 short a[3];
 char b[3];
 short c[2];
} s1;
```

```
typedef struct {
 s1 u[2];
 int v;
 char w[5];
 char *x;
} s2;
```

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



Tamaño de s1 = 14 bytes



Tamaño de s2 = 44 bytes

b) **Escribe** UNA ÚNICA INSTRUCCIÓN que permita mover **z.u[1].c[0]** al registro **%ax**, siendo **z** una variable de tipo **s2** cuya dirección está almacenada en el registro **%ebx**. **Indica** claramente la expresión aritmética utilizada para el cálculo de la dirección.

La expresión aritmética para calcular la dirección del operando es @ini_z + 14*1 + 2*3 + 1*3 + 1 = @ini_z + 24, por lo tanto %ebx + 24

La instrucción es: **movw 24(%ebx), %ax**

COGNOMS:

NOM:

Problema 3. (2.5 puntos)

Dado el siguiente código escrito en C:

```
int examen(short int a, short int b, short int v[10], int M[10][10]) {
  int ii;
  short int aa, bb;
  short int vector[10];
  int *matriz;
  ...
  return M[ii][ii];
}
```

a) **Dibuja** el bloque de activación de la rutina examen, indicando claramente los desplazamientos respecto a **%ebp** y el tamaño de todos los campos.

```
     --------------------  i<- ebp-32
4    |          ii        |
     --------------------  aa<- ebp-28
4    |    bb    |   aa    | bb<- ebp-26
     ---------------------- <- ebp-24
20   |      vector        |
     ---------------------- <- ebp-4
4    |     *matriz        |
     ----------------------   <- ebp
4    |     ebp old        |
     ----------------------
4    |        ret         |
     ---------------------- <- ebp+8
4    |    --    |    a    |
     ---------------------- <- ebp+12
4    |    --    |    b    |
     ---------------------- <- ebp+16
4    |        @v          |
     ---------------------- <- ebp+20
4    |        @M          |
     ---------------------- <- ebp+24
```

b) **Traduce** a ensamblador del x86 la instrucción `return M[ii][ii]` suponiendo que no hay que restaurar el valor de ningún registro.

```
imul $10, -32(%ebp), %eax; se puede optimizar multiplicando por 11
                         ;(10*ii + ii = 11*ii): imul $11, %eax, %eax
addl -32(%ebp), %eax     ; no es necesaria si se ha multiplicado por 11
                         ; en la instrucción anterior
movl 20(%ebp), %ecx      ; ecx <- @M
movl (%ecx,%eax,4), %eax ; eax <- M[ii][ii]
                         ; deshacer enlace dinámico
movl %ebp, %esp
popl %ebp
ret
```

COGNOMS:

NOM:

Problema 4. (2.5 puntos)

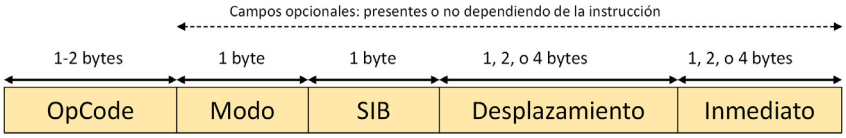
Cuando se va a ejecutar la siguiente instrucción escrita en ensamblador del x86:

```
movl $337, -40(%ebp, %esi, 4)
```

El contenido de los registros del procesador (en hexadecimal) es:

```
eax: 0x00000005     ebx: 0x00000004     ecx: 0x00000001     edx: 0x00000002
esp: 0x80800f30     ebp: 0x80800f48     esi: 0x00000002     edi: 0x00001001
eip: 0x808000fe     eflags: 0x286
```

Sabemos que la codificación de instrucciones sigue el siguiente esquema tal y como se explica en clase:



Además sabemos que el procesador tiene una cache de datos de 32Kbytes (copy back+write allocate), y otra de instrucciones de 16Kbytes, ambas con lineas de 32 bytes y asociatividad 2. Sus páginas son de 4Kbytes y dispone de un TLB de 4 entradas. Asumimos que tanto el TLB como las caches están vacíos.

a) ¿**Cuantos** bytes ocuparía la instrucción anterior si sabemos que el OpCode de movl solo ocupa 1 byte?

1 opcode + 1 Modo + 1 SIB + 1 Desplazamiento + 2 Inmediato = 6 bytes

b) ¿**Cuantos** bloques de memoria se leen para ejecutar el movl? **Escribe** la dirección de dichos bloques.

2 instrucciones (0x808000e0 y 0x80800100) + 1 datos (0x80800f20)

c) ¿**Cuantos** bloques de memoria se escriben para ejecutar el movl? **Escribe** la dirección de dichos bloques.

Ninguno, es copy back + write allocate

d) ¿**Cuantos** fallos de TLB tendremos al ejecutar la instrucción anterior? **Indica** a que páginas es el fallo

1 fallo de TLB, instrucciones (0x80800) y datos están en la misma página

Suponiendo el siguiente código:

```
        leal N-1(), %esi              <- Esta instrucción ocupa 2 bytes
bucle:  movl $337, -40(%ebp, %esi, 4) <- Instrucción a evaluar
        decl %esi
        jns bucle                     <- El bucle hace N iteraciones
```

e) ¿**Cuantos** fallos de cache provoca el movl en función del valor de N? (suponemos de nuevo que las caches están vacías y los registros tienen valor indicado anteriormente antes de ejecutar el código).

1 + N/8 + 1 si N no es múltiplo de 8: 1 para las instrucciones + N/8 para los datos pero aumenta en 1 fallo extra si N no es múltiplo de 8 por alineamiento.

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.

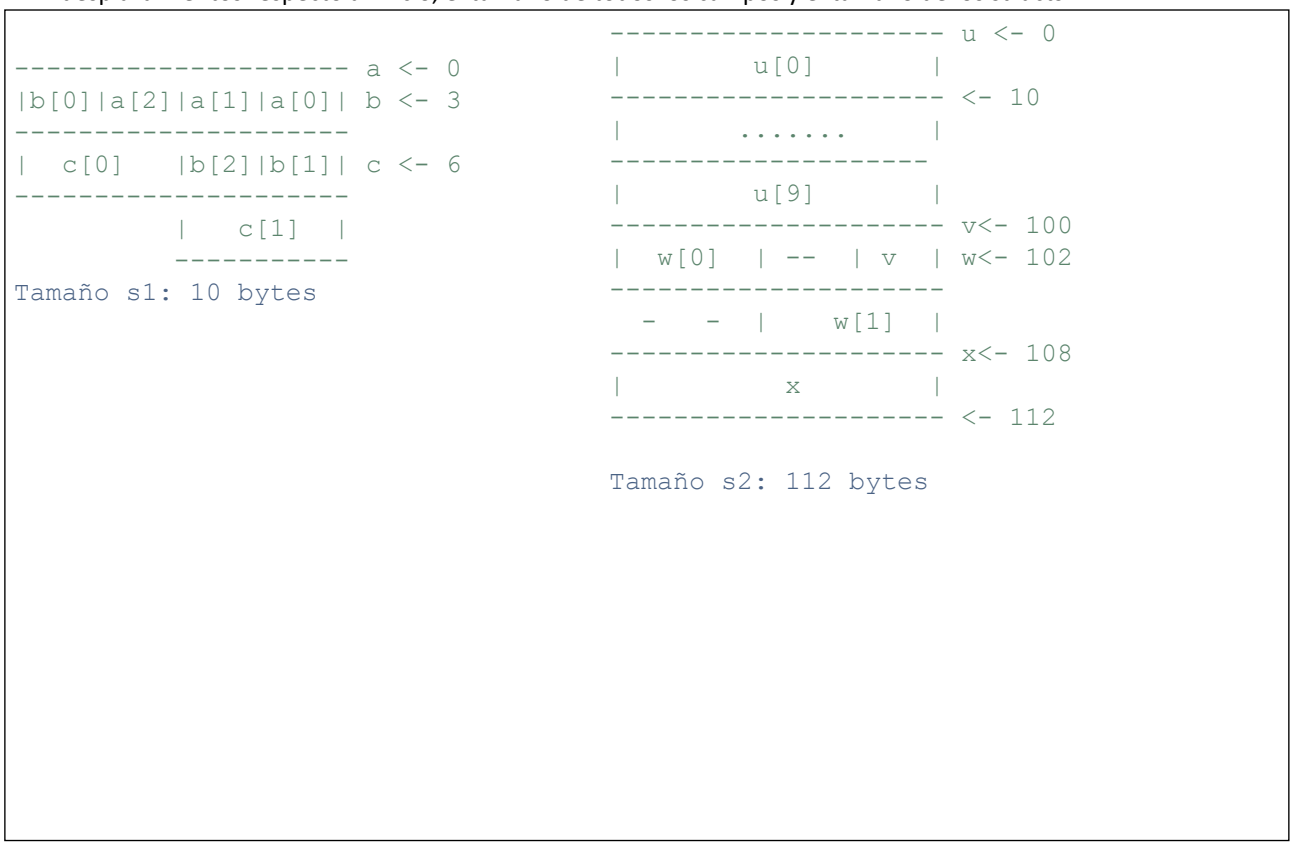
Problema 1. (5 puntos)

Dado el siguiente código escrito en C, que compilamos para un sistema linux de 32 bits:

```
typedef struct {
  char a[3];
  char b[3];
  short c[2];
  } s1;
```

```
typedef struct {
  s1 u[10];
  char v;
  short w[2];
  int x;
  } s2;
```

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



b) **Escribe** UNA ÚNICA INSTRUCCIÓN que permita mover **x.u[5].b[1]** al registro **%dh**, siendo **x** una variable de tipo **s2** cuya dirección está almacenada en el registro **%ecx**.
Indica claramente la expresión aritmética utilizada para el cálculo de la dirección.

La expresión aritmética para calcular la dirección del operando es: @x+10*5+4, por lo tanto %ecx+54

La instrucción es: **movb 54(%ecx), %dh**

Dado el siguiente código escrito en C, que compilamos para un sistema linux de 32 bits:

```
int examen(char b[2][3], char c, short d) {
  char y[2][3];
  short z;
  short w;
  int x;
  . . .
  x=examen(y,y[0][1],w);
  . . .
}
```

c) **Dibuja** el bloque de activación de la rutina examen, indicando claramente los desplazamientos respecto a **%ebp** y el tamaño de todos los campos.

```
examen
---------------------------- y<- ebp-16
|y[1,0] y[0,2] y[0,1] y[0,0]|          |
---------------------------- <- ebp-12
|      z      | y[1,2]y[1,1]  | z<- ebp-10
---------------------------- w<- ebp-8
|  --     --    |     w        |
---------------------------- x<- ebp-4
|           x                 |
-----------------------------
|          ebp                | <- ebp
-----------------------------
|          RET                |
-----------------------------
|         @b                  | @b <- ebp+8
-----------------------------
|  --   --   --         |  c   | c <- ebp+12
-----------------------------
|  --   --      |      d       | d <- ebp+16
-----------------------------
```

d) **Traduce** a ensamblador x86 la instrucción `x=examen(y,y[0][1],w);` que se encuentra en el interior de la subrutina, usando el mínimo número de instrucciones.

```
pushl -8(%ebp)
pushl -15(%ebp)
leal -16(%ebp), %eax
pushl %eax
call examen
addl $12, %esp
movl %eax, -4(%ebp)
```

COGNOMS:

NOM:

Problema 2. (5 puntos)

Un programa P tiene un 90% del código que es perfectamente paralelizable.

a) **Calcula** el numero mínimo de procesadores para conseguir un speed-up de 5 en el programa P.

Ley de amdahl:
5 = 1/(0,1 + 0,9/x) ---> **X = 9**

Cada CPU funciona a un frecuencia de 3 GHz. Se ha ejecutado el programa P secuencial en un simulador con una única CPU ideal donde todos los accesos a memoria tardan un ciclo. Dicho programa ejecuta $6x10^9$ instrucciones, realiza $3x10^9$ operaciones de punto flotante y tarda $9x10^9$ ciclos.

b) **Calcula** el CPI ideal (CPI_{ID}) y el tiempo de ejecución en segundos (Texec) del programa P en este sistema ideal.

CPI_{IDEAL} = $9x10^9$ ciclos / $6x10^9$ instrucciones = **1,5 ciclos / instrucción**

Texe = Ciclos / F= $9x10^9$ ciclos / 3GHz = **3 seg**

c) **Calcula** los MIPS y MFLOPS en dicho sistema ideal.

MIPS = Instrucciones / Texe*10^6 = $6x10^9$ instrucciones / 3*10^6 = **2000 MIPS**

MFLOPS = OpsPF / Texe*10^6 = $3x10^9$ opsPF / 3*10^6 = **1000 MFLOPS**

La implementación (CPU_R) de dicha CPU dispone de una cache con una política de escritura *Copy Back* y *Write Allocate*. En caso de acierto en la cache el tiempo de acceso es de 1 ciclo. En caso de fallo, el tiempo de penalización es de 75 ciclos para reemplazar un bloque NO modificado y de 150 ciclos para reemplazar un bloque modificado.

El programa P realiza $9,6x10^9$ accesos a memoria, con una tasa de fallos (miss) del 10%. Sabemos que el 25% de los accesos son escrituras y que la probabilidad de que un bloque haya sido modificado en cache es del 20%.

d) **Calcula** el tiempo medio de acceso a memoria (Tma) para el programa P en CPU_R.

Tma = Tsa+m*(Pm*Tpfm + Pn*Tpfn) = 1 + 0,10*(0,20*150+0,8*75) = **10 ciclos/acceso**

e) **Calcula** el CPI del programa P en la CPU_R.

nr = $9,6x10^9$ accesos / $6x10^9$ instrucciones = 1,6 a/i
CPI = CPIid + nr*(Tma-1) = 1,5 c/i + 1,6 a/i * 9 c/a = 15,9 c/i

Alternativa: CPI = CPIid + CPImem = 1,5 + 1,6*0,10*(0,20*150+0,80*75) = 15,9 c/i

Cada acceso a memoria principal consume 100 nanoJoules (nJ).

f) **Calcula** el consumo total de energía de la memoria principal causada por los fallos de cache.

fallo que reemplaza bloque NO modificado genera 1 acceso a MP (lectura bloque)
fallo que reemplaza bloque modificado genera 2 accesos a MP (escritura bloque + lectura bloque)
AccesosMP = accesos * m * (Pn*1 + Pm *2) = $9{,}6x10^9$ * 0,1* (0,8*1 + 0,2*2) = $1{,}152x10^9$accesos a MP

E = $1{,}152x10^9$a * 100 nJ = **115,2 Joules**

Dicha CPU genera direcciones lógicas de 36 bits y direcciones físicas de 24 bits. La jerarquía completa de memoria está compuesta por un TLB (al que se accede ANTES de acceder a la cache), la memoria cache y la memoria principal. El TLB tiene 4 entradas y es completamente asociativo. La cache tiene un tamaño de 64 Kbytes, líneas de 64 bytes y es 4-asociativa. El tamaño de página del sistema es de 4 KBytes

g) **Calcula** el número de líneas, vías y conjuntos que tiene la cache. **Especifica claramente** cómo has realizado los cálculos.

64*1024 bytes / 64 bytes/linea = 1024 líneas
4-asociativa -> 4 vias
1024 lineas / 4 lineas/conjunto = 256 conjuntos.

La CPU lanza un acceso a la dirección lógica 0xEFABCD012 y sabemos que el contenido del TLB es:

VPN	PPN
0xFABCD0	0xA00
0xEFABCD	0xB01
0xABCD01	0xC02
0xBCD012	0xD03

h) **Indica** a qué dirección física se accede, en qué conjunto de la cache se encuentra el dato y cuál es la etiqueta guardada en memoria cache. **Justifica** la respuesta.

VPN = 0xEFABCD -> PPN = 0xB01 **@Fisica = 0xB01012**

VPN (24 bits)	desplaçament (12 bits)

Conjunto = 0x40 TAG = 0x2C0

TAG (10 bits)	cjt (8)	byte (6)

i) **Indica** el tamaño máximo que puede tener la cache para que sea posible acceder a la cache y al TLB en paralelo, suponiendo que se mantiene el tamaño de línea y el grado de asociatividad, y que se mantienen también el resto de parámetros de la jerarquía de memoria. **Justifica** la respuesta.

El tamaño máximo de la cache viene delimitado por los bits de byte + los bits de conjunto. Estos bits pueden direccionar como máximo una página del sistema (4 KBytes --> 12 bits).

Bits de byte = 6 -> número máximo de bits de conjunto = 6.

Tamaño máximo de cache = 64 conjuntos * 4 lineas/conjunto * 64 bytes/linea = 16 KBytes.