2º Grado Informática
Estructura de Computadores
3 de septiembre de 2015

Nombre:	
DNI:	**Grupo:**

Test de Prácticas (4.0p)

Todas las preguntas son de elección simple sobre 4 alternativas.
Cada respuesta vale 4/20 si es correcta, 0 si está en blanco o claramente tachada, -4/60 si es errónea.
Anotar las respuestas (a, b, c o d) en la siguiente tabla.

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20

1. En contraposición a un ejecutable Linux ELF, un fichero objeto (obtenido con gcc -c)

a. no contiene tablas de símbolos, que sólo se añaden al ejecutable tras enlazar
b. ubica el código (y los datos) a partir de la posición 0x0, las direcciones definitivas sólo se calculan tras enlazar
c. contiene tablas de símbolos, pero sólo para los símbolos locales; p.ej., sólo al enlazar un programa que llame a printf se añadirá una entrada a la tabla con la dirección de printf
d. no reserva espacio para las direcciones de objetos desconocidos; p.ej., sólo al enlazar un CALL a printf se insertan los 4B de la dirección de printf entre el codop de CALL y el de la siguiente instrucción máquina

2. Tras ejecutar las tres instrucciones que se muestran desensambladas a continuación, el registro EAX toma el valor

```
08048074 <_start>:
8048074: be 74 80 04 08 mov $_start, %esi
8048079: 46             inc  %esi
804807a: 8b 06          mov (%esi), %eax
```

a. 0x08048074
b. 0x08048075
c. 0x08048079
d. 0x0804807a

3. Si %edx contiene 0xf000 y %ecx contiene 0x0100, el direccionamiento 0x80(%ecx,%edx,2) se refiere a la posición

a. 0xf182
b. 0xf280
c. 0x1e180
d. Ninguna de las respuestas anteriores es correcta

4. ¿Cuál de las siguientes secuencias de instrucciones multiplica %eax por 10?

a. leal(%eax,%eax,4), %eax
   sall $2, %eax
b. imull $0x10, %eax
c. addl %eax, %eax
   shll $5, %eax
d. Varias o ninguna de las respuestas anteriores son correctas, no se puede marcar una y sólo una

5. ¿Cuál de las siguientes secuencias de instrucciones calcula a=b-a, suponiendo que %eax es a y %ebx es b?

a. subl %eax, %ebx
b. subl %ebx, %eax
c. notl %eax
   addl %ebx, %eax
d. Varias o ninguna de las respuestas anteriores son correctas, no se puede marcar una y sólo una

6. Para desplazar %eax a la derecha un número variable de posiciones ≤ 32, indicado en %ebx, se puede hacer

a. sar %ebx, %eax

b. sar %bl, %eax

c. mov %ebx, %ecx
sar %cl, %eax

d. No se puede, sar sólo admite un número fijo de posiciones (debe ser un valor inmediato)

---

7. Indicar cuál es la dirección de salto (en qué dirección empieza la subrutina <main>) para esta instrucción call

```
0804854e: e8 3d 06 00 00 call <main>
08048553: 50             pushl %eax
```

a. 08048b90

b. 08048b8b

c. 450a854e

d. No se puede deducir de la información proporcionada, faltan datos

---

8. El ajuste de marco de pila que gcc (Linux/IA-32) prepara para todas las funciones consiste en las instrucciones

a. movl %ebp, %esp
pushl %ebp

b. movl %esp, %ebp
pushl %esp

c. pushl %ebp
movl %esp, %ebp

d. pushl %esp
movl %ebp, %esp

---

9. Al traducir de lenguaje C a ensamblador, gcc en máquinas Linux/IA-32 almacena (reserva espacio para) una variable "var" local a una función "fun" en una dirección de memoria referenciable (en lenguaje ensamblador) como

a. var (el nombre de la variable representa su posición de memoria)

b. -k(%ebp), siendo k un número constante positivo relativamente pequeño

c. k(%ebp), siendo k un número constante positivo relativamente pequeño

d. k(%esp), siendo k un número constante positivo relativamente pequeño

---

10. En x86_64 se pueden referenciar los registros

a. %rax, %eax, %ax, %ah, %al

b. %rsi, %esi, %si, %sih, %sil

c. %r8, %r8d, %r8w, %r8l

d. %r12q, %r12d, %r12w, %r12l

---

11. Comparando las convenciones de llamada de gcc Linux IA-32 con x86_64 respecto a registros

a. En IA-32 %ebx es salva-invocante, pero en x86_64 %rbx es salva-invocado

b. En IA-32 %ecx es salva-invocante, y en x86_64 %rcx es salva-invocante también

c. En IA-32 %esi es salva-invocado, y en x86_64 %rsi es salva-invocado también

d. En IA-32 %ebp es especial (marco de pila), y en x86_64 %rbp también

---

12. Si declaramos **`int val[5]={1,5,2,1,3};`** entonces

a. &val[2] es de tipo int* y vale lo mismo que val+8

b. val+4 es de tipo int* y se cumple que *(val+4)==5

c. val+1 es de tipo int y vale 2

d. val[5] es de tipo int y vale 3

---

13. Al traducir la sentencia C **`r->i = val;`** gcc genera el código ASM **`movl %edx, 12(%eax)`**. Se deduce que

a. r es un puntero que apunta a la posición de memoria 12

b. el desplazamiento de i en *r es 12

c. i es un entero que vale 12

d. val es un entero que vale 12

---

14. Una función C llamada **`get_el()`** genera el siguiente código ensamblador. Se puede adivinar que

```
movl 8(%ebp), %eax
leal (%eax,%eax,4), %eax
addl 12(%ebp), %eax
movl var(,%eax,4), %eax
```

a. var es un array multi-nivel (punteros a enteros) de cuatro filas
b. var es un array multi-nivel pero no se pueden adivinar las dimensiones
c. var es un array bidimensional de enteros, no se pueden adivinar dimensiones
d. var es un array bidimensional de enteros, con cinco columnas

---

15. En la práctica "suma" se pide sumar una lista de 32 enteros **con** signo de 32bits en una plataforma de 32bits sin perder precisión, esto es, evitando *overflow*. ¿Cuál es el menor valor positivo que repetido en toda la lista causaría *overflow* con 32bits?

a. 0x0400 0000
b. 0x0800 0000
c. 0x4000 0000
d. 0x8000 0000

---

16. En la práctica "suma" se pide sumar una lista de 32 enteros **con** signo de 32bits en una plataforma de 32bits sin perder precisión, esto es, evitando *overflow*. ¿Cuál es el mayor valor negativo (menor en valor absoluto) que repetido en toda la lista causaría *overflow* con 32bits?

a. 0xffff ffff
b. 0xfc00 0000
c. 0xfbff ffff
d. 0xf000 0000

---

17. La práctica "*parity*" debía calcular la suma de paridades impar (XOR de todos los bits) de los elementos de un array. La siguiente función contiene un único error (en realidad se han editado 2 líneas de código sobre la versión correcta) pero produce resultado **correcto** cuando se usa como test el array

```
int   parity4(unsigned* array,
                int len){
  int i;
  unsigned x;
  int val=0, result=0;

  for (i=0; i<len; i++){
    x = array[i];
    asm("\n"
"ini:                   \n\t"
    "xor %[x], %[v]   \n\t"
    "shr %[x]         \n\t"
    "jnz ini          \n\t"
  : [v]"+r" (val)
  : [x] "r" (x)
    );
    result += val & 0x1;
  }
  return result;
}
```

a. array={0, 1, 2, 3}
b. array={1, 2, 4, 8}
c. array={1, 16, 256, 1024}
d. array={5, 4, 3, 2}

---

18. En la práctica de la cache, el código de "line.cc" incluye la sentencia

```
for (unsigned long long line=1;
line<=LINE; line<<=1) { … }
```

¿Qué objetivo tiene la expresión line<<=1?

a. salir del bucle si el tamaño de línea se volviera menor o igual que 1 para algún elemento del vector
b. duplicar el tamaño del salto en los accesos al vector respecto a la iteración anterior
c. volver al principio del vector cuando el índice exceda la longitud del vector
d. sacar un uno (1) por el stream line

---

19. En la práctica de la cache, el código de "size.cc" accede al vector saltando de 64 en 64. ¿Por qué?

a. Porque cada elemento del vector ocupa 64 bytes
b. Para recorrer el vector más rápidamente
c. Porque el tamaño de cache L1 de todos los procesadores actuales es de 64KB
d. Para anular los aciertos por localidad espacial, esto es, que sólo pueda haber aciertos por localidad temporal

---

20. En el programa "size.cc" de la práctica de la cache, si el primer escalón pasa de tiempo=2 para un tamaño de vector

menor que 32KB a tiempo=8 para un tamaño mayor que 32KB, podemos asegurar que:

a. La cache L1 es cuatro veces más rápida que la cache L2.
b. La cache L1 es seis veces más rápida que la cache L2.
c. La cache L1 es al menos cuatro veces más de rápida que la cache L2.
d. La cache L1 es como mucho cuatro veces más rápida que la cache L2.