2º Grado Informática
Estructura de Computadores
23 de enero de 2017

Nombre:	
DNI:	**Grupo:**

Test de Prácticas (4.0p)

Todas las preguntas son de elección simple sobre 4 alternativas.
Cada respuesta vale 0.2p si es correcta, 0 si está en blanco o claramente tachada, -0.0$\widehat{6}$p si es errónea.
Anotar las respuestas (a, b, c ó 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. ¿Qué hace gcc -O0?

a. Compilar sin optimización
b. Compilar .c→.o (fuente C a objeto)
c. Compilar .s→.o (fuente ASM a objeto)
d. Compilar .c→.s (C→ASM sin generar objeto)

2. ¿Qué modificador (switch) de gcc hace falta para compilar una aplicación de 32 bits en un sistema de 64 bits?

a. -m32
b. -m64
c. -march32
d. -march64

3. La etiqueta del punto de entrada a un programa ensamblador en el entorno de las prácticas 1 a 4 (GNU/as Linux x86) es:

a. _main
b. .L0
c. _start
d. _init

4. La siguiente línea en la sección de datos de un programa en ensamblador de IA32

```
result: .int 0,0
```

a. Reserva espacio para un único entero, inicializado a 0,0
b. Reserva espacio para un entero, inicializado a 0, seguido de un dato de tamaño indefinido, también inicializado a 0
c. Reserva espacio para dos enteros, inicializados ambos a 0
d. Reserva espacio para un único entero, inicializado a 0, en la posición de memoria 0

5. El volcado mostrado abajo se ha obtenido con el comando...

```
00000000 <main>:
 0: 55              push %ebp
 1: 89 e5           mov  %esp,%ebp
 3: 83 e4 f0        and  $-16,%esp
 6: 83 ec 10        sub  $0x10,%esp
 9: c7 44 24 04 03 movl $3, 4(%esp)
 e: 00 00 00
11: c7 04 24 01 00 movl $0x1,(%esp)
16: 00 00
18: e8 fc ff ff ff call <main+0x19>
1d: c9             leave
1e: c3             ret
```

a. objdump -h p
b. objdump -d p
c. objdump -d p2.o
d. objdump -t p2.o

6. En la práctica "media" se desea invocar desde lenguaje ensamblador la función printf() de libC. Eso implica que este programa, como todo programa que use libC,

a. es obligatorio que contenga main (y entonces es más cómodo usar gcc para enlazar)
b. es obligatorio enlazarlo usando gcc (y entonces es más cómodo que contenga main)
c. es ventajoso para ensamblarlo que contenga main, y entonces es conveniente enlazarlo usando gcc (aunque ambas cosas son opcionales)

d. es ventajoso para enlazarlo usar gcc, y entonces es conveniente que contenga main (aunque ambas cosas son opcionales)

---

7. En la práctica "media" se pide sumar una lista de 32 enteros SIN signo de 32 bits en una plataforma de 32 bits sin perder precisión, esto es, evitando perder acarreos. Un estudiante entrega la siguiente versión

```
# $lista en EBX, longlista en ECX
suma:
   mov $0, %eax
   mov $0, %edx
   mov $0, %esi
bucle:
   add (%ebx,%edx,4), %eax
   jnc  seguir
   inc  %edx
seguir:
   inc  %esi
   cmp  %esi,%ecx
   jne bucle
   ret
```

Esta función presenta una única diferencia frente a la solución recomendada en clase, relativa al indexado en el array.

Esta función suma:

a. Produce siempre el resultado correcto
b. Fallaría con **lista: .int 1,1,1,1, 1,1,1,1, ...**
c. Fallaría con **lista: .int 1,2,3,4, 1,2,3,4, ...**
d. No es correcta pero el error no se manifiesta en los ejemplos propuestos, o se manifiesta en ambos

---

8. En la práctica "media" se pide sumar una lista de 32 enteros SIN signo de 32 bits en una plataforma de 32 bits sin perder precisión, esto es, evitando perder acarreos. De entre los siguientes, ¿cuál es el mínimo valor entero que repetido en toda la lista causaría acarreo con 32 bits (sin signo)? Se usa notación decimal y espacios como separadores de millares/millones/etc.

a. 10 000 000
b. 100 000 000
c. 1 000 000 000
d. 10 000 000 000

---

9. En la práctica "media" se pide sumar una lista de 32 enteros CON signo de 32 bits en una plataforma de 32 bits sin perder precisión, esto es, evitando desbordamiento. De entre los siguientes, ¿cuál es el valor negativo más pequeño (en valor absoluto) que repetido en toda la lista causaría desbordamiento con 32 bits (en complemento a 2)? Se usa notación decimal y espacios como separadores de millares/millones/etc.

a. -10 000 000
b. -100 000 000
c. -1 000 000 000
d. -10 000 000 000

---

10. ¿Cuál es el popcount (peso Hamming, nº de bits activados) del número 19?

a. 2
b. 3
c. 4
d. 5

---

11. La práctica "popcount" debía calcular la suma de bits (peso Hamming) de los elementos de un array. Un estudiante entrega la siguiente versión de popcount3:

```
int popcount3(unsigned* array,
              int len){
    int i, res = 0;
    unsigned x;
    for( i = 0; i < len; i++ ) {
      x = array[i];
      asm("ini3:        \n"
       "shr %[x]        \n"
       "adc $0, %[r]    \n"
       "add $0, %[x]    \n"
       "jne ini3        \n"
      : [r] "+r" (res)
      : [x] "r" (x) );
    }
    return res;
}
```

Esta función sólo tiene una diferencia con la versión "oficial" recomendada en clase. En concreto, una instrucción máquina en la sección **asm()** es distinta.

Esta función popcount3:

a. produce siempre el resultado correcto
b. fallaría con **array={0,1,2,3}**
c. fallaría con **array={1,2,4,8}**
d. no es correcta pero el error no se manifiesta en los ejemplos propuestos, o se manifiesta en ambos

---

12. La práctica "popcount" debía calcular la suma de bits (peso Hamming) de los elementos de un array. Un estudiante entrega la siguiente versión de popcount3:

```
int popcount3(int* array, int len){
    long val = 0;
    int i;
    unsigned x;
    for (i=0; i<len; i++){
      x= array[i];
      do{
          val += x & 0x1;
          x >>= 1;
      } while (x);
      val += (val >> 16);
      val += (val >> 8);
    }
    return val & 0xFF;
}
```

Esta función es una mezcla inexplicada de las versiones "oficiales" de **popcount2** y **popcount4**, incluyendo diferencias en 2 tipos de datos, la ausencia de la variable **res** y la diferente posición de la máscara **0xFF**.

Esta función popcount3:

a. produce siempre el resultado correcto
b. fallaría con **array={0,1,2,3}**
c. fallaría con **array={1,2,4,8}**
d. no es correcta pero el error no se manifiesta en los ejemplos propuestos, o se manifiesta en ambos

---

13. La práctica "popcount" debía calcular la suma de bits (peso Hamming) de los elementos de un array. Un estudiante entrega la siguiente versión de popcount4:

```
int popcount4(unsigned* array,
              int len){
  int i,j;
  unsigned x;
  int result = 0;
  for(i=0;i<len;i++){
    x=array[i];
    for(j=0;j<8;j++){
      result += x & 0x01010101;
      x>>=1;
    }
  }
  result += (result >> 16);
  result += (result >> 8);
  return result & 0xFF;
}
```

Esta función presenta varias diferencias con la versión "oficial" recomendada en clase, incluyendo la ausencia de una variable auxiliar **val** y la diferente posición de los desplazamientos >> y máscara **0xFF**.

Esta función popcount4:

a. produce siempre el resultado correcto
b. fallaría con **array={0,1,2,3}**
c. fallaría con **array={1,2,4,8}**
d. no es correcta pero el error no se manifiesta en los ejemplos propuestos, o se manifiesta en ambos

---

14. La práctica "parity" debía calcular la suma de paridades impar (XOR de todos los bits) de los elementos de un array. Un estudiante entrega la siguiente versión de parity5:

```
int parity5(unsigned * array,
            int len){
  int i,j, result = 0;
  unsigned x;
  for(i = 0; i<len; i++){
    x=array[i];
    for(j=1; j<8*sizeof(int); j*=2)
      x ^= x >> j;
    result += x & 0x1;
  }
  return result;
}
```

Esta función sólo se diferencia de la versión "oficial" recomendada en clase, en las condiciones del bucle **for** interno.

Esta función parity5:

a. Produce siempre el resultado correcto
b. Fallaría con **array={0,1,2,3}**
c. Fallaría con **array={1,2,4,8}**
d. No es correcta pero el error no se manifiesta en los ejemplos propuestos, o se manifiesta en ambos

---

15. La función **gettimeofday()** en la práctica de la "bomba digital" se utiliza para

a. Para comparar las duraciones de las distintas soluciones del programa
b. Para imprimir la fecha y hora
c. Para cifrar la contraseña en función de la hora actual
d. Para cronometrar lo que tarda el usuario en introducir la contraseña

---

16. Un fragmento de una "bomba" desensamblada es:

```
0x0804873f: call 0x8048504 <scanf>
0x08048744: mov 0x24(%esp),%edx
0x08048748: mov 0x804a044,%eax
0x0804874d: cmp %eax,%edx
0x0804874f: je 0x8048756 <main+230>
0x08048751: call 0x8048604 <boom>
0x08048756: ...
```

La contraseña/clave en este caso es...

a. el string almacenado a partir de la posición de memoria 0x804a044
b. el string almacenado a partir de la posición de memoria 0x24(%esp)
c. el entero almacenado a partir de la posición de memoria 0x804a044
d. el entero 0x804a044

---

17. Una de las "bombas" utiliza el siguiente bucle para cifrar la cadena con la contraseña introducida por el usuario:

```
80485bb: rolb $0x4,(%eax)
80485be: add  $0x1,%eax
80485c1: cmp  %edx,%eax
80485c3: jne 80485bb <encrypt+0x20>
```

La intrucción **rolb** rota el byte destino hacia la izquierda tantos bits como indica el operando fuente. Si inicialmente eax apunta a la cadena del usuario, que se compara con otra cadena "\x16\x26\x27\x16\x36\x16\x46\x16\x26\x27\x16", almacenada en el código, la contraseña es:

a. "\x61\x62\x72\x61\x63\x61\x64\x61\x62\x72\x61" ("abracadabra")
b. "\x61\x72\x62\x61\x64\x61\x63\x61\x72\x62\x61" ("arbadacarba")
c. "\x63\x61\x64\x61\x62\x72\x61\x61\x62\x72\x61" ("cadabraabra")
d. "\x61\x62\x72\x61\x61\x62\x72\x61\x63\x61\x64" ("abraabracad")

---

18. Una de las "bombas" utiliza el siguiente código para cifrar la clave numérica introducida por el usuario y ahora almacenada en eax:

```
804870d:    xor    $0xfff,%eax
8048712:    mov    $0x2,%ecx
8048717:    cltd
8048718:    idiv   %ecx
804871a:    cmp    %eax,0x804a034
```

Si el entero almacenado a partir de 0x804a034 es 0x7ff, la clave numérica puede ser:

a. 0x10094F97 (269045655)
b. 0xfff (2047)
c. 0x7ff (4095)
d. 1

---

19. En el programa line.cc de la práctica 5, si para cada tamaño de línea (line) recorremos una única vez el vector, la gráfica resultante es decreciente porque:

a. Cada vez que line aumenta al doble, el número de aciertos por localidad temporal aumenta, porque ya habíamos accedido a cada posición i del vector cuando lo recorrimos en el punto anterior del eje X.
b. Cada vez que line aumenta al doble, el número de aciertos por localidad espacial aumenta, porque ya habíamos accedido a cada posición i-1 del vector cuando lo recorrimos en el punto anterior del eje X.
c. Cada vez que line aumenta al doble, se accede con éxito a más posiciones del vector en niveles de la jerarquía de memoria más rápidos.
d. Cada vez que line aumenta al doble, realizamos la mitad de accesos al vector que para el valor anterior.

---

20. ¿Cuál de las siguientes afirmaciones sobre el programa size.cc de la práctica 5 es cierta?

a. La diferencia de velocidades entre L2 y L3 es mayor que la diferencia de velocidades entre L1 y L2.
b. Si continuáramos multiplicando por 2 el tamaño del vector en el eje X obteniendo más puntos de la gráfica, esta continuaría horizontal para cualquier valor más allá de 64 MB.
c. La gráfica tiene escalones hacia arriba porque en cada punto del eje X accedemos al mismo número de elementos del vector y el número de aciertos por localidad temporal disminuye bruscamente en ciertos puntos al aumentar el tamaño del vector.
d. La gráfica tiene tramos horizontales porque el hecho de realizar la mitad de accesos al vector en cada punto de un tramo horizontal respecto al anterior punto de ese mismo tramo horizontal es compensado por el número de fallos creciente en ese mismo tramo horizontal.