2º Grado Informática
Estructura de Computadores
19 Septiembre 2012

Nombre:	
DNI:	**Grupo:**

Examen de Prácticas (4.0p)

1. **Funcionamiento de la Pila.** (1 punto). Responder a las siguientes preguntas sobre la convención de pila usada en Linux x86 (con gcc en modo 32-bit):

A. ¿Cómo modifica la pila la instrucción CALL?
B. ¿Cómo modifica la pila la instrucción LEAVE?
C. ¿Cómo modifica la pila la instrucción RET?
D. ¿Cómo se pasan los argumentos a una función?
E. ¿Cómo se devuelven los valores de retorno a la función invocante?
F. Dibujar una region de pila detallando cómo llamaría un programa a la función

```
printf("%s %d elementos.\n", "Se insertaron", 4);
```

El primer argumento de `printf` (el *string* de formato) está almacenado en `0xbeefbabe` y el segundo *string* en `0xbabecafe`. Dibujar la zona de pila modificada/creada durante esta llamada a función, al menos desde el área de preparación de argumentos hasta el tope de pila alcanzado justo después de ejecutar la instrucción CALL. Como `printf` es una función `libc`, también se puede indicar el siguiente valor que se insertará en la pila.

2. Funciones aritméticas sencillas. (1 punto). A continuación se muestran **tres** pequeñas funciones aritméticas en C con el código ensamblador correspondiente. Cada tramo ensamblador contiene como máximo un error. Si hay error, marcarlo con un círculo y dar una **breve** explicación de por qué está mal en el espacio bajo el código. Si no hay error, decir sencillamente que no lo hay. Notar que el error (si existe) está en la traducción C→ensamblador, no en la lógica o funcionamiento del código C.

```
int twice(int x) {
  return (x + x);
}
08048334 <twice>:
 8048334:       55                      push   %ebp
 8048335:       89 e5                   mov    %esp,%ebp
 8048337:       8b 45 08                mov    0x8(%ebp),%eax
 804833a:       01 c0                   add    %eax,%eax
 804833c:       c3                      ret
 804833d:       c9                      leave
```

```
int second(char *str) {
  return str[1];
}
0804833e <second>:
 804833e:       55                      push   %ebp
 804833f:       89 e5                   mov    %esp,%ebp
 8048341:       8b 45 08                mov    0x8(%ebp),%eax
 8048344:       0f be 40 01             movsbl 0x1(%eax),%eax
 8048348:       c9                      leave
 8048349:       c3                      ret
```

```
int constant() {
  return 2;
}
0804834a <constant>:
 804834a:       55                      push   %ebp
 804834b:       89 e5                   mov    %esp,%ebp
 804834d:       b8 02 00 00 00          mov    0x2,%eax
 8048352:       5d                      pop    %ebp
 8048353:       c3                      ret
```

---

3. Programación mixta C-asm. (1 punto). A continuación se muestra un listado C con compilación condicional, pudiéndose recompilar un bucle *while* o una sentencia *asm inline* según el valor del símbolo ASM. Recordar que el mnemotécnico de Intel `cdq` (*convert double to quad*) equivale a AT&T `cltd`, y que en lenguaje C una expresión de asignación vale el valor asignado y `%` es la operación módulo.

Se ha ocultado parte del código C del bucle *while*. Sabiendo que la sentencia *asm* realiza el mismo cálculo que el bucle *while*:

A. Completar el código del bucle *while* sobre el enunciado

B. Indicar el resultado que se imprimiría en pantalla, debajo del listado

C. Describir brevemente en lenguaje natural (español) qué calcula el programa. Si se conoce el nombre del algoritmo, indicarlo también

```
#include <stdio.h>
#define ASM 1

int main(void){
    int A=21, B=9, r;

#if ! ASM
    while ( _ = _ % _){             // Completar
       _ = _;
       _ = _;
    }
#else

    asm( "movl   %[B],%%ecx  \n"
         "movl   %[A],%%eax  \n"
".bucle:  cdq                \n"    // AT&T cltd
         "idivl       %%ecx  \n"
         "testl %%edx,%%edx  \n"    // while (_=_%_)
         "jz     .fin        \n"
         "movl  %%ecx,%%eax  \n"    // _=_;
         "movl  %%edx,%%ecx  \n"    // _=_;
         "jmp    .bucle      \n"
".fin:                       \n"
         "movl %%eax, %[A]   \n"
         "movl %%ecx, %[B]   \n"
         "movl %%edx, %[r]   \n"

         :[A] "+m" (A),
          [B] "+m" (B),
          [r] "+m" (r)
         :
         :"%eax", "%ecx", "%edx"
    );
#endif

    printf("resultado: %d\n", B);
}
```

```
resultado:
```

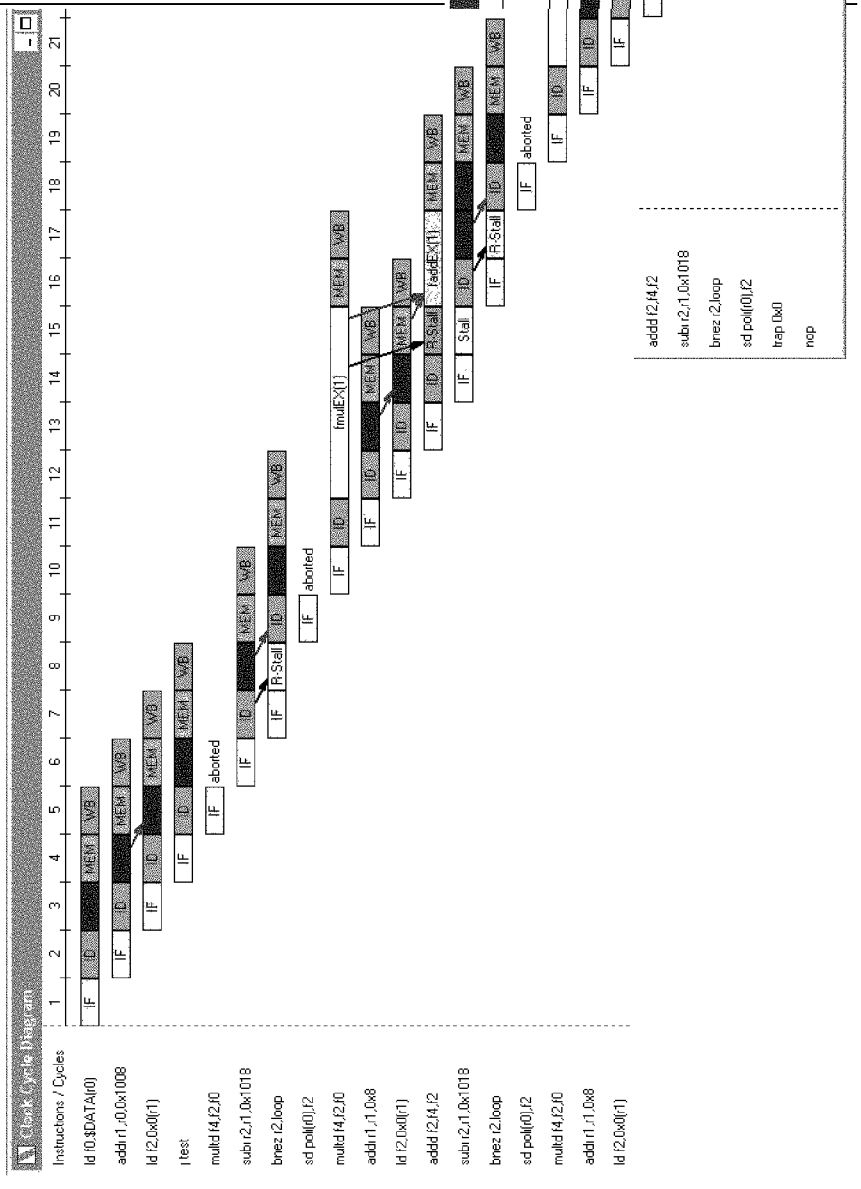
Descripción:

4 . Segmentación de cauce. (1 punto). A continuación se muestra un listado WinDLX de un programa que tarda en ejecutarse 32 ciclos con una unidad de multiplicación de 4 ciclos, como se muestra en la figura más abajo. Por motivos de legibilidad (y de espacio) ha sido necesario poner la figura de forma apaisada.

```
            .data
x:          .double 2
coefs:      .double 3,4,5
poli:       .double 0

            .text
start:
            ld      f0,x
            add     r1,r0,coefs
            ld      f2,(r1)
            j       test
loop:
            multd   f4,f2,f0
            add     r1,r1,8
            ld      f2,(r1)
            addd    f2,f4,f2
test:
            sub     r2,r1,poli-8
            bnez    r2,loop

            sd      poli,f2
            trap     0
;
; 32 ciclos con fMUL=4cyc
```



A. Indicar qué bloqueos presenta el programa: en qué número de ciclo, en qué instrucción, cuántos ciclos dura, por qué motivo, tipo de bloqueo. Si se repite varias veces, basta indicar los #ciclo adicionales en los que se repite

B. Indicar qué resultado calcula el programa: valor concreto y dónde se guarda

C. Describir brevemente en lenguaje natural (español) qué calcula el programa. Si se conoce el nombre del algoritmo, indicarlo también

D. Basta con mover de sitio una instrucción, y además repetirla en otro sitio, para que desaparezcan todas las dependencias. Reescribir el listado, mostrando cómo quedaría tras esa modificación (resaltar las partes modificadas)