PRÁCTICA 6: IMAGEN BIT MAP PORTABLE:

ÍNDICE

20.6.1. Programación en C

- Leer el procedimiento de programación en el fichero **LEEME.txt**
- El objetivo es modificar la función principal **main()** del programa original **bitmap_gen_test.c** dando lugar a distintos programas independientes entre sí.
  1. - Compilar y ejecutar el program *bitmap_gen_test.c*
  2. - visualizar la imagen del fichero test.bmp: **$display test.bmp**
  3. - Módulo **cuadrado_128x128.c** :Cambiar las dimensiones de la imagen a 128 pixeles x 128 pixeles definiendo la macro DIMENSION=128 y definiendo para cada pixel un color gris con una intensidad del 50% de su valor máximo.
  4. - Módulo **cuadrados_4.c**: Generar 4 cuadrados, uno dentro de otro simétricamente, donde el cuadrado mayor negro es 512x512 y el resto se reduce 1/8 cada uno. No utilizar ctes en las sentencias de C, utilizar las macros x_coor, y_coor, top para indicar el valor inicial del **for** y la posición máxima (top) de las filas y columnas. Colores de los cuadrados: background (00-00-00)/(FF-00-FF)/(00-FF-FF)/(FF-FF-00)/
  5. - Módulo **bmp_funcion.c**: El bloque de código que realiza el bucle para inicializar los pixeles del cuadrado convertirlo en la función:
     - prototipo: *void pixels_generator(unsigned int x, unsigned int y, unsigned int maximo, RGB_data reg_mem[][top])*
       - x e y son el origen de coordenadas del cuadrado
       - maximo es la coordenada mayor del cuadrado
       - llamada a la función: pixels_generator(xcoor,ycoor,top,buffer);
         - los argumentos xcoor=top/8, ycoor=top/8 y top=512 definirlos mediante macros

Tras leer el procedimiento del fichero, se realizará los cambios en los distintos programas independientes.

En primer lugar, compilamos el programa bit_map_gen_test.c. Para ello, utilizaremos el comando:

- gcc -m32 -o bitmap_gen_test bitmap_gen_test.c

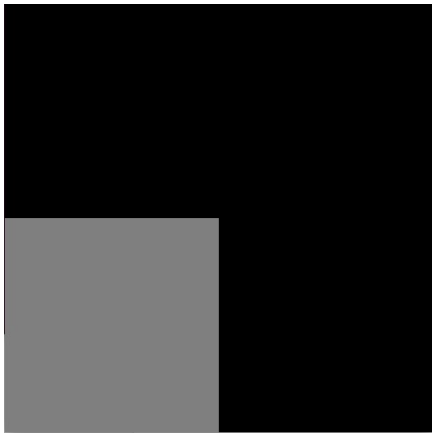
Para ejecutar bitmap_gen_test:

- ./bitmap_gen_test

Por último, para mostrar por pantalla el mapa de bits:

- display test.bmp

```
eduardo@eduardo-Victus-by-HP-Laptop-16-e0xxx:~/Escritorio/Eduardo Ezponda/ESTRUCTURA DE COMPUTADORES/P6/PRUEBA$ gcc -m32 -o bitmap_gen_test bitmap_gen_test.c
eduardo@eduardo-Victus-by-HP-Laptop-16-e0xxx:~/Escritorio/Eduardo Ezponda/ESTRUCTURA DE COMPUTADORES/P6/PRUEBA$ ./bitmap_gen_test
eduardo@eduardo-Victus-by-HP-Laptop-16-e0xxx:~/Escritorio/Eduardo Ezponda/ESTRUCTURA DE COMPUTADORES/P6/PRUEBA$ ls
bitmap_gen_test  bitmap_gen_test.c  test.bmp
eduardo@eduardo-Victus-by-HP-Laptop-16-e0xxx:~/Escritorio/Eduardo Ezponda/ESTRUCTURA DE COMPUTADORES/P6/PRUEBA$ display test.bmp
```



El mapa de bits de dimensiones 512x512 píxeles es de un color grisáceo al utilizar una proporción 0x7f en cada uno de los bytes del rgb_data (blue, red, green). Si todos los bytes tuvieran el valor de 0, el resultado sería el color negro y por eso se muestra en el resto del mapa de bits.

En primer lugar, a través de la acción memset inicializamos cada uno de los píxeles a 0 (negro).

A continuación, realizamos un doble bucle en el que la primera mitad inferior izquierda se modifica el rgb_data para que cada byte tome el valor de 0x7f (gris).

Por último, generaremos el mapa de bits con la acción bmp_generator.

```c
int main(int argc, char **argv)
{
    int i,j;

    RGB_data buffer[512][512]; //defino buffer de tipo rgb data (3 bytes -> green, red and blue) como matriz[512][512]

    memset(buffer, 0, sizeof(buffer)); //inicializo bytes de buffer a 0. El 0 es un pixel negro

    for (i = 0; i < 256; i++) //recorrer desde abajo hasta la derecha subiendo filas desde la más inferior
    {
        for (j = 0; j < 256; j++)
        {
            buffer[i][j].g = buffer[i][j].b = 0x7f; //0x7f = mitad entre 1 y 0 (blanco y negro -> gris)
            buffer[i][j].r = 0x7f;
            //combinados los 3 colores hacen el color blanco
            //intensidad depende de bit
        }
    }

    bmp_generator("./test.bmp", 512, 512, (BYTE*)buffer);   //me sacas un fichero que empiece en buffer con un 512 de altura y ancho

    return EXIT_SUCCESS;
}
```

```c
typedef struct
{
        BYTE    b; //blue, green or read
        BYTE    g;
        BYTE    r;
} RGB_data; // RGB TYPE, plz also make sure the order
```

3. - Módulo cuadrado_128x128.c : Cambiar las dimensiones de la imagen a 128 pixeles x 128 pixeles definiendo la macro DIMENSION=128 y definiendo para cada pixel un color gris con una intensidad del 50% de su valor máximo.

```c
#define DIMENSION 128
```

Para definir la macro DIMENSIÓN, utilizamos la instrucción:

- #define DIMENSION 128

```
for (i = 0; i < DIMENSION/2; i++)
{
    for (j = 0; j < DIMENSION/2; j++)
    {
        buffer[i][j].g = 0x7f;
        buffer[i][j].b = 0x7f;
        buffer[i][j].r = 0x7f;
    }
}
```

Por último, utilizamos la proporción 0x7f para una intensidad del 50% de su máximo para generar un tono grisáceo.

Por los datos del doble bucle, únicamente se modificarán la mitad inferior izquierda. El resto de los píxeles serán negros.

4. - Módulo cuadrados_4.c: Generar 4 cuadrados, uno dentro de otro simétricamente, donde el cuadrado mayor negro es 512x512 y el resto se reduce 1/8 cada uno. No utilizar ctes en las sentencias de C, utilizar las macros x_coor, y_coor, top para indicar el valor inicial del for y la posición máxima (top) de las filas y columnas. Colores de los cuadrados: background (00-00-00)/(FF-00-FF)/(00-FF-FF)/(FF-FF-00)/

```
#define top   512
#define xcoor top/8 // 512/8=64
#define ycoor top/8
```

En primer lugar, definimos las macros a través de la instrucción #define.

A continuación realizaremos los cuatro cuadrados con su respectiva intensidad. El primer cuadrado negro (todos los valores a 0) se realiza con la acción memset.

Luego, se realizarán los tres dobles bucles para cada uno de los colores, y en función de que tamaño del cuadrado le corresponda, el doble bucle tomará unas posiciones o otras.

Además, cada vez los tamaños de los cuadrados son más pequeños, con lo cual, el doble bucle tomará una menor cantidad de posiciones.

PRIMER CUADRADO: (rellena los 512 píxeles (0-511), pero más tarde se modifican)

```
   memset(buffer, 0, sizeof(buffer)); //  inicializa el buffer con el valor cero. Librería libc.
   // Los colores se expresan en formato RGB donde la intensidad de cada color se codifica con un byte
   // 0x00: ausencia de color ; 0xFF: intensidad máxima
   // La ausencia de los tres colores primarios R=G=B=0x00 es el negro
   // R=G=B=0xFF es el blanco
```

SEGUNDO CUADRADO: (rellena 64-447 píxeles)

```
for (i = xcoor; ((xcoor <= i) && (i < (top-xcoor))); i++) //xcorr <= i se podría eliminar ya que no realiza ninguna función
{
    for (j = ycoor; ((ycoor <= j) && (j < (top-ycoor))); j++)
    {
        // intensidad de rojo
        buffer[i][j].r = 0xff;
        // intensidad de verde
        buffer[i][j].g = 0x00;
        // intensidad de azul
        buffer[i][j].b = 0xff;
    }
}
```

TERCER CUADRADO: (rellena 128-383 píxeles)

```
for (i = 2*xcoor; ((2*xcoor <= i) && (i < (top-2*xcoor))); i++)
    {
        for (j = 2*ycoor; ((2*ycoor <= j) && (j < (top-2*ycoor))); j++)
        {
            // intensidad de rojo
            buffer[i][j].r = 0x00;
            // intensidad de verde
            buffer[i][j].g = 0xff;
            // intensidad de azul
            buffer[i][j].b = 0xff;
        }
    }
```

CUARTO CUADRADO: (rellena 192-319 píxeles)

```
for (i = 3*xcoor; ((3*xcoor <= i) && (i < (top-3*xcoor))); i++)
    {
        for (j = 3*ycoor; ((3*ycoor <= j) && (j < (top-3*ycoor))); j++)
        {
            // intensidad de rojo
            buffer[i][j].r = 0xFF;
            // intensidad de verde
            buffer[i][j].g = 0xff;
            // intensidad de azul
            buffer[i][j].b = 0x00;
        }
    }
```

5. - Módulo bmp_funcion.c: El bloque de código que realiza el bucle para inicializar los pixeles del cuadrado convertirlo en la función:

- prototipo: void pixels_generator(unsigned int x, unsigned int y, unsigned int maximo, RGB_data reg_mem[][top])
  - x e y son el origen de coordenadas del cuadrado
  - maximo es la coordenada mayor del cuadrado
  - llamada a la función: pixels_generator(xcoor,ycoor,top,buffer);
    - los argumentos xcoor=top/8, ycoor=top/8 y top=512 definirlos mediante macros

ACCIÓN:

```
void pixels_generator(unsigned int x, unsigned int y, unsigned int maximo, RGB_data reg_mem[][top])
{
 int i,j;

for (i = x; ((x <= i) && (i < (top-x))); i++)
    {
        for (j = y; ((y <= j) && (j < (top-y))); j++)
        {
            // intensidad de rojo
            reg_mem[i][j].r = 0xff;
            // intensidad de verde
            reg_mem[i][j].g = 0x00;
            // intensidad de azul
            reg_mem[i][j].b = 0xff;
        }
    }

}
```

LLAMADA FUNCIÓN PASANDO LOS PARÁMETROS:

```
int main(int argc, char **argv)
{

    RGB_data buffer[top][top]; // Array de pixels

    memset(buffer, 0, sizeof(buffer)); //  inicializa el buffer con el valor cero. Librería libc.
    // Los colores se expresan en formato RGB donde la intensidad de cada color se codifica con un byte
    // 0x00: ausencia de color ; 0xFF: intensidad máxima
    // La ausencia de los tres colores primarios R=G=B=0x00 es el negro
    // R=G=B=0xFF es el blanco


    pixels_generator(xcoor,ycoor,top,buffer);


    bmp_generator("./test.bmp", top, top, (BYTE*)buffer); // casting a buffer. Definido como RGB_data y pasado como BYTE

    return EXIT_SUCCESS;
}
```

DEFINICIÓN DE MACROS:

```
#define top   512
#define xcoor top/8
#define ycoor top/8
```

Apunte:
Se podría modificar la inicialización de los píxeles a negro en vez de con la función memset con la función pixels_generator. Para ello, únicamente se tendría que pasar como argumento la intensidad de los bytes, en este caso 0.

20.6.2. Programación en ASM

1. - Módulo **bmp_as.c**: Implementar la función *void pixels_generator(unsigned int maximo, RGB_data reg_mem[][top])* desarrollando en lenguaje ensamblador la subrutina *pixels_generator* en el nuevo fichero **array_pixel.s**. El fichero en lenguaje ensamblador únicamente contendrá la subrutina.
   - La subrutina implementa el doble bucle.
   - De forma implicita en la propia subrutina consideraremos los argumentos x=y=0.
   - Azul, rojo y verde son las intensidades de todos los pixeles del cuadrado.

LLAMADA FUNCIÓN:

```
pixels_generator(LADO,DIMENSION,buffer);
```

DEFINICIÓN MACROS:

```
#define DIMENSION   1024
//#define ORIGEN_X 0
//#define ORIGEN_Y 0
#define LADO 512
//#define PROPORCION 0x7f
```

ARCHIVO DE OBJETO ARRAY_PIXEL.S:

```
  ## Nuevo frame en la pila
  push %ebp
  mov %esp, %ebp
################################################
### SALVO EL CONTEXTO ANTERIOR A LA SUBRUTINA ###
  push %eax
  push %ebx
  push %ecx
  push %edx
  push %edi
  push %esi

################################################ pixels_generator_2(ORIGEN_X,ORIGEN_Y,LADO,PROPORCION,DIMENSION,buffer);
  ## Capturo el argumento origen_x //en una misma instrucción no se puede acceder dos veces a memoria -> dos instrucciones
  movl $0,origen_x
  ## Capturo el argumento origen_y
  movl $0,origen_y
  ## Capturo el argumento lado
  mov 8(%ebp),%eax
  mov %eax,lado
  ## Capturo el argumento proporcion
  movl $0x7f,proporcion
  ## Capturo el argumento dimension
  mov 12(%ebp),%eax
  mov %eax,dimension
  ## Capturo el argumento buffer
  mov 16(%ebp),%eax
  mov %eax,buffer_ptr   #inicializo el puntero con el argumento
  ## intensidad colores: color INICIAL
  ## color background ubuntu RGB is (48, 10, 36).
  movb proporcion,%al
  movb $0x00,rojo
  movb $0xff,verde
  movb $0x00,azul
  ## Cálculo de los limites: origen_x+lado-1   ;    origen_y+lado-1
  mov origen_x,%eax
  add lado,%eax
  dec %eax
  mov %eax,fin_x
  mov origen_y,%eax
  add lado,%eax
  dec %eax
  mov %eax,fin_y

  ## Bucle de inicialización del array
  mov origen_x,%edi   #inicio Filas
```

```
fila:
  mov origen_y,%esi   #inicio Columnas
columna:

  ## Reset registros aritmética
  xor %eax,%eax
  xor %ebx,%ebx
  xor %ecx,%ecx
  ## Aritmética Fila
  movw %di,%bx    #posición fila
  imul dimension,%ebx
  imul $DIMENSION_Z,%ebx
  ## Aritmética Columna
  movw %si,%cx    #posición columna
  imul $DIMENSION_Z,%ecx
  ## Correspondencia array_pixel -> posición buffer
  xor %eax,%eax
  add %ebx,%eax
  add %ecx,%eax
  mov %eax,%ebx     #EBX contiene el offset del pixel en el buffer

  ## Actualizar colores en el pixel
  xor %ecx,%ecx   #índice color
  movb azul,%al   #intensidad azul
        mov buffer_ptr,%edx
  lea (%edx,%ebx),%edx
  mov %eax,(%edx,%ecx) # dirección efectiva = M[buffer_ptr] + offset + posi_color
  inc %ecx
  movb verde,%al    #intensidad verde
  mov buffer_ptr,%edx
  lea (%edx,%ebx),%edx
  mov %eax,(%edx,%ecx) # dirección efectiva = M[buffer_ptr] + offset + posi_color
  inc %ecx
  movb rojo,%al   #intensidad rojo
  mov buffer_ptr,%edx
  lea (%edx,%ebx),%edx
  mov %eax,(%edx,%ecx) # dirección efectiva = M[buffer_ptr] + offset + posi_color

  ## actualización posición columna
  inc %esi

  ##     control columna
  cmp fin_y,%esi
  jz  col_exit

  ## siguiente columna
```

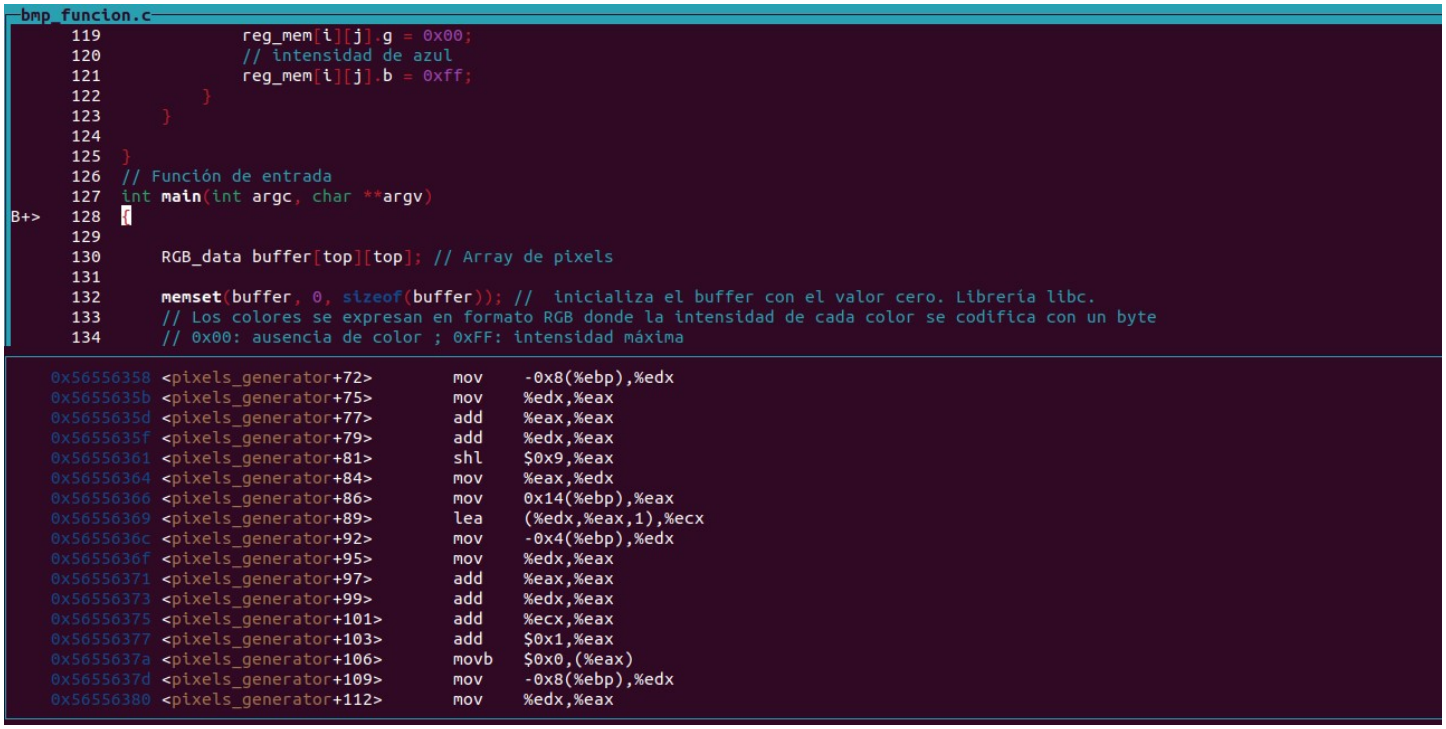
```
  mov buffer_ptr,%edx
  lea (%edx,%ebx),%edx
  mov %eax,(%edx,%ecx) # dirección efectiva = M[buffer_ptr] + offset + posi_color

  ## actualización posición columna
  inc %esi

  ##     control columna
  cmp fin_y,%esi
  jz  col_exit

  ## siguiente columna

  jmp columna

  ## actualización posición fila
col_exit:
  ##siguiente fila
  inc %edi

  cmp fin_x,%edi
  jz fil_exit

  ## siguiente fila
  jmp fila

fil_exit:


################################################
### RESTAURO  EL CONTEXTO ANTERIOR A LA SUBRUTINA ###
  pop %esi
  pop %edi
  pop %edx
  pop %ecx
  pop %ebx
  pop %eax
################################################

  ## Recuperar el antiguo frame
  mov %ebp,%esp
  pop %ebp

  ret

  .end
```

20.6.3. GDB

1. En el programa en **bmp_funcion.c** indicar la posición de la pila donde se salva la dirección de retorno de la subrutina **pixels_generator**, así como el contenido del frame pointer y del stack pointer.
2. Lo mismo que en el apartado anterior con el programa **bmp_as.c** para la subrutina *pixels_generator*

1. En primer lugar, abrimos el terminal y compilamos el programa bmp_funcion.c con gcc -m32 -g -o bmp_funcion bmp_funcion.c . Después, nos introducimos en el gdb. Como estamos en un programa en lenguaje c y queremos ver la dirección de retorno y el contenido de la pila y punteros, utilizamos el comando layout split para ver las instrucciones en ensamblador.

Para ello, deberíamos de haber introducido antes el breakpoint en main y haber hecho “run”. Como se puede observar en la imagen, la mitad inferior sería el resultado de utilizar el comando layout split para mostrar todas las instrucciones en un nivel menor.



Utilizando el comando n (next) avanzaremos una instrucción en el programa.c, así como con el comando s (step) nos adentraremos en la función en el programa.c . Si queremos avanzar una a una las instrucciones en ensamblador, tendremos que utilizar ni (next instruction) y si (step instruction) para introducirnos en la función. Al realizar un next, se realizarán muchas instrucciones en ensamblador a la vez, a diferencia de que al realizar un next instruction, lo más probable es que se necesiten varios para pasar a la siguiente instructión del programa.c .

Con lo cual, realizamos varios “next” hasta la instrucción de la llamada a la función pixels_generator. Como se puede observar en la imagen, se realizan cuatro push correspondientes a los argumentos de la función en el orden inverso al ser la pila una estructura LIFO (Last In First Out). Realizando varios examine antes de la llamada a la función con call, podemos observar a través del stack pointer (esp) el contenido de los argumentos en la pila. El último, es la dirección del buffer.

```
bmp_funcion.c
    132     memset(buffer, 0, sizeof(buffer)); //  inicializa el buffer con el valor cero. Librería libc.
    133     // Los colores se expresan en formato RGB donde la intensidad de cada color se codifica con un byte
    134     // 0x00: ausencia de color ; 0xFF: intensidad máxima
    135     // La ausencia de los tres colores primarios R=G=B=0x00 es el negro
    136     // R=G=B=0xFF es el blanco
    137
    138
>   139     pixels_generator(xcoor,ycoor,top,buffer);
    140
    141
    142     bmp_generator("./test.bmp", top, top, (BYTE*)buffer); // casting a buffer. Definido como RGB_data y pasado como BYTE
    143
    144     return EXIT_SUCCESS;
    145 }
    146
    147
```

Como se puede observar, nos encontramos en la instrucción de la llamada a la función pixels_generator. Por ello, en la parte inferior, se apilan los 4 argumentos de la función.

```
> 0x56556447 <main+103>    push   %eax
  0x56556448 <main+104>    push   $0x200
  0x5655644d <main+109>    push   $0x40
  0x5655644f <main+111>    push   $0x40
```

ARGUMENTOS FUNCIÓN:

PUSH %EAX: eax contiene la dirección del buffer a modificar

PUSH $0x200: contiene el valor de top en hexadecimal → 0x200 = 2*16*16 = 512 en decimal

PUSH $0x40: contiene el valor de ycoor en hexadecimal → 0x40 = 4*16 = 64 en decimal

PUSH $0x40: contiene el valor de xcoor en hexadecimal → 0x40 = 4*16 = 64 en decimal

```
#define top   512
#define xcoor top/8 // 512/8 = 64
#define ycoor top/8
```

Realizando varios next instruction avanzamos hasta la llamada a pixels_generator con call, para mostrar los argumentos de la pila con el comando examine.

```
(gdb) x /xw $esp
0xfff3d010:     0x00000040
(gdb) x /4xw $esp
0xfff3d010:     0x00000040      0x00000040      0x00000200      0xfff3d03c
```

En primer lugar, se muestra la cima de la pila en hexadecimal y tamaño word.
En la parte inferior, se muestran los cuatro argumentos de la función ya introducidos en la pila como se observa con el stack pointer. A continuación se muestran en decimal (d).

```
(gdb) x /4wd $esp
0xfff3d010:     64      64      512     -798660
(gdb)
```

A continuación, introducimos el comando step instruction para llamar a la función.
A través de hacer print (p) $esp, mostramos la dirección de memoria donde está la cima de la pila, y por tanto de la dirección de retorno.

```
(gdb) p /x $esp
$2 = 0xfff3d00c
(gdb)
```

Como se observa, se introduce en la pila la dirección de retorno que en este caso es 0x56556456, que es el contenido del stack pointer. Los siguientes valores de la pila son los argumentos de la función.

```
(gdb) x /xw $esp
0xfff3d00c:     0x56556456
(gdb) x /5xw $esp
0xfff3d00c:     0x56556456      0x00000040      0x00000040      0x00000200
0xfff3d01c:     0xfff3d03c
(gdb)
```

Por último, se apila la antigua dirección del frame pointer (ebp) para tenerla guardada, y se crea el nuevo frame.

```
void pixels_generator(unsigned int x, unsigned int y, unsigned int maximo
{
 int i,j;

for (i = x; ((x <= i) && (i < (top-x))); i++)
    {
        for (j = y; ((y <= j) && (j < (top-y))); j++)
        {
            // intensidad de rojo
            reg_mem[i][j].r = 0xff;

6310 <pixels_generator>          push   %ebp
6311 <pixels_generator+1>        mov    %esp,%ebp
```

Los primer examine se realiza antes de hacer el mov %esp, %ebp en el que el frame pointer empieza a apuntar al stack pointer.

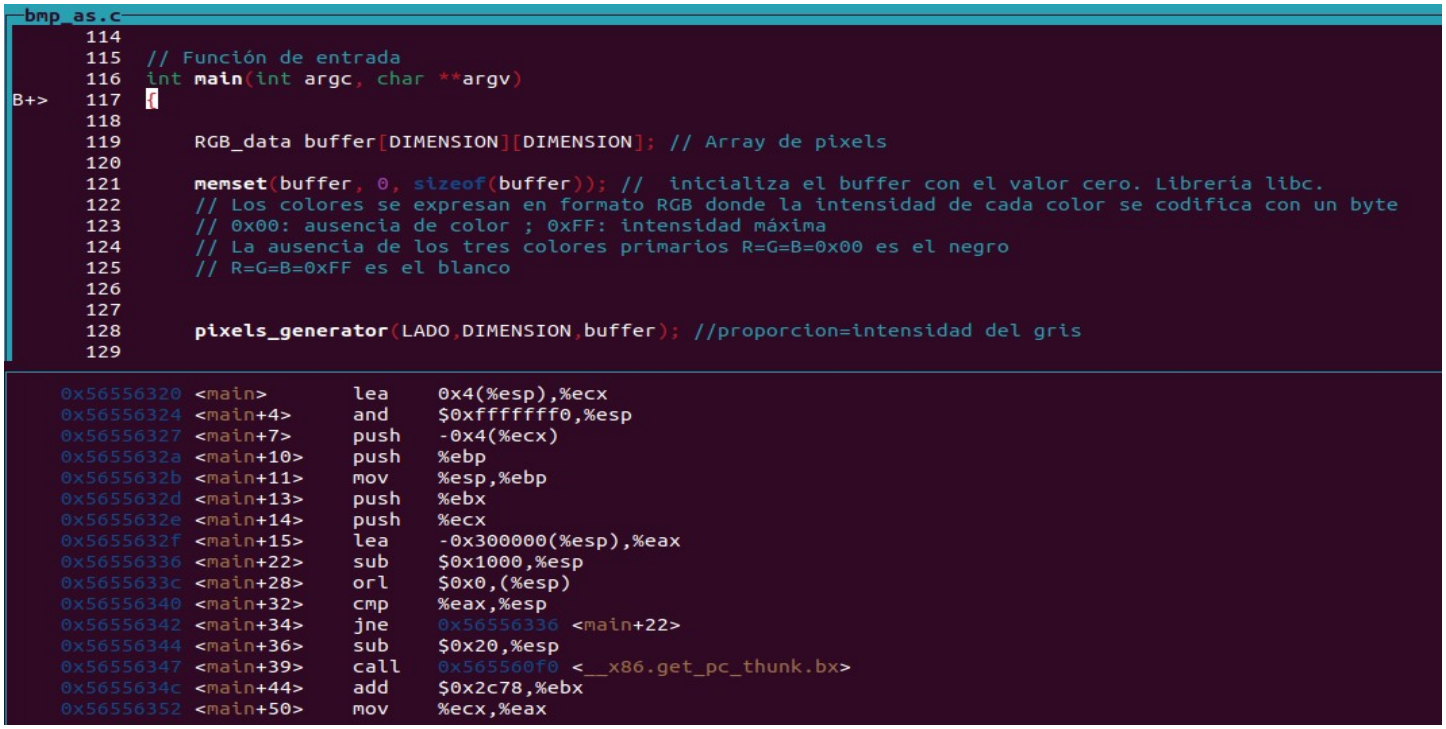
```
(gdb) x /xw $esp
0xfff3d008:     0xffffd048
(gdb) ni
(gdb) x /xw $ebp
0xfff3d008:     0xffffd048
(gdb)
```

Realizamos el mismo procedimiento para el programa bmp_as.c y la subrutina pixels_generator.

1. En primer lugar, abrimos el terminal y compilamos el programa bmp_as.c y el archivo de objeto as - - 32 – gstabs -o pixels_generator.o pixels_generator para después ejecutar compilar con gcc -m32 -g -o bmp_funcion pixels_generator.o bmp_funcion.c .

Después, nos introducimos en el gdb. Como estamos en un programa en lenguaje c y queremos ver la dirección de retorno y el contenido de la pila y punteros, utilizamos el comando layout split para ver las instrucciones en ensamblador.

Para ello, deberíamos de haber introducido antes el breakpoint en main y haber hecho “run”. Como se puede observar en la imagen, la mitad inferior sería el resultado de utilizar el comando layout split para mostrar todas las instrucciones en un nivel menor.



Utilizando el comando n (next) avanzaremos una instrucción en el programa.c, así como con el comando s (step) nos adentraremos en la función en el programa.c . Si queremos avanzar una a una las instrucciones en ensamblador, tendremos que utilizar ni (next instruction) y si (step instruction) para introducirnos en la función. Al realizar un next, se realizarán muchas instrucciones en ensamblador a la vez, a diferencia de que al realizar un next instruction, lo más probable es que se necesiten varios para pasar a la siguiente instrución del programa.c .

Con lo cual, realizamos varios “next” hasta la instrucción de la llamada a la función pixels_generator. Como se puede observar en la imagen, se realizan tres push correspondientes a los argumentos de la función en el orden inverso al ser la pila una estructura LIFO (Last In First Out). Realizando varios examine antes de la llamada a la función con call, podemos observar a través del stack pointer (esp) el contenido de los argumentos en la pila. El último, es la dirección del buffer.

```
bmp_as.c
    121     memset(buffer, 0, sizeof(buffer)); //  inicializa el buffer con el valor cero. Librería libc.
    122     // Los colores se expresan en formato RGB donde la intensidad de cada color se codifica con un byte
    123     // 0x00: ausencia de color ; 0xFF: intensidad máxima
    124     // La ausencia de los tres colores primarios R=G=B=0x00 es el negro
    125     // R=G=B=0xFF es el blanco
    126
    127
 >  128     pixels_generator(LADO,DIMENSION,buffer); //proporcion=intensidad del gris
    129
    130
    131     bmp_generator("./test_as.bmp", DIMENSION, DIMENSION, (BYTE*)buffer); // casting a buffer. Definido como
    132
    133     exit(1);
    134 }
    135
    136
```

Como se puede observar, nos encontramos en la instrucción de la llamada a la función pixels_generator. Por ello, en la parte inferior, se apilan los 3 argumentos de la función.

```
0x5655638a <main+106>    push   %eax
0x5655638b <main+107>    push   $0x400
0x56556390 <main+112>    push   $0x200
0x56556395 <main+117>    call   0x565563c7 <pixels_generator>
```

ARGUMENTOS FUNCIÓN:

PUSH %EAX: eax contiene la dirección del buffer a modificar

PUSH $0x400: contiene el valor de DIMENSION en hexadecimal → 0x400 = 4*16*16 = 1024 en decimal

PUSH $0x200: contiene el valor de LADO en hexadecimal → 0x200 = 2*16*16 = 512 en decimal

call $0x565563c7: llamada a la función pixels_generator con la dirección de retorno

```
#define DIMENSION   1024
#define LADO 512
```

Realizando varios next instruction avanzamos hasta la llamada a pixels_generator con call, para mostrar los argumentos de la pila con el comando examine.

```
(gdb) x /xw $esp
0xffcfd020:     0x00000200
(gdb) x /3xw $esp
0xffcfd020:     0x00000200      0x00000400      0xffcfd04c
```

En primer lugar, se muestra la cima de la pila en hexadecimal y tamaño word.
En la parte inferior, se muestran los tres argumentos de la función ya introducidos en la pila como se observa con el stack pointer. A continuación se muestran en decimal (d).

```
(gdb) x /3dw $esp
0xffcfd020:     512     1024    -3157940
(gdb)
```

A continuación, introducimos el comando step instruction para llamar a la función.
A través de hacer print (p) /x (hexadecimal) $esp, mostramos la dirección de memoria donde se encuentra la cima de la pila y por tanto de la dirección de retorno.

```
(gdb) p /x $esp
$4 = 0xffcfd01c
(gdb)
```

Como se observa, se introduce en la pila la dirección de retorno que en este caso es 0x5655639a, que es el contenido del stack pointer. Los siguientes valores de la pila son los argumentos de la función.

```
(gdb) x /xw $esp
0xffcfd01c:     0x5655639a
(gdb) x /4xw $esp
0xffcfd01c:     0x5655639a      0x00000200      0x00000400      0xffcfd04c
(gdb)
```

Por último, se apila la antigua dirección del frame pointer (ebp) para tenerla guardada, y se crea el nuevo frame.

```
array_pixel.s
     173            pop %esi
     174            pop %edi
     175            pop %edx
     176            pop %ecx
     177            pop %ebx
     178            pop %eax
     179    ################################################
     180
     181            ## Recuperar el antiguo frame
     182            mov %ebp,%esp
     183            pop %ebp
     184
  >  185            ret
     186
     187            .end
     188

   0x565563c7 <pixels_generator>            push   %ebp
>  0x565563c8 <pixels_generator+1>          mov    %esp,%ebp
```

Los primer examine se realiza antes de hacer el mov %esp, %ebp en el que el frame pointer empieza a apuntar al stack pointer.

```
(gdb) x /xw $ebp
0xffffd058:     0x00000000
(gdb) x /xw $esp
0xffcfd018:     0xffffd058
(gdb) ni
(gdb) x /xw $ebp
0xffcfd018:     0xffffd058
(gdb)
```

COMPILACIÓN:

```
gcc -m32  -g -o bitmap_gen_test  bitmap_gen_test.c

gcc -m32 -g -o  bmp_as bmp_as.c

gcc -m32 -g -o bmp_funcion bmp_funcion.c

gcc -m32 -g -o cuadrado_128x128  cuadrado_128x128.c

gcc -m32 -g -o cuadrados_4   cuadrados_4.c

as – - 32 -gstabs -o array_pixel.o array_pixel.s
```

Cabe destacar el uso de -m32 para usar una máquina de 32 bits, y -g para cargar la tabla de símbolos. Además el comando “as – - 32 -gstabs -o array_pixel.o array_pixel.s” crea el archivo de objeto necesario para la ejecución de la función pixel_generator en asm del programa bmp_as.c

CONCLUSIÓN:

Durante ésta práctica se ha introducido la programación de los mapas de bits tanto en lenguaje c como en ASM. Para ello, es necesario la estructura RGB_data que la conforman 3 bits distintos (red, blue, green), y en función de su valor, el píxel tomará un valor o otro.

Con la función memset, se inicializan todos los píxeles del mapa con un valor determinado. Si los 3 valores del RGB_data son 0x00, entonces dará lugar al negro. Sin embargo, si los valores son 0xFF, el resultado será blanco.

Por último, un valor medio como 0x7f dará un tono grisáceo, al tener un valor entre el negro 0x00 y el blanco 0xFF.

Además, con la función bmp_generator se generará el mapa de bits, que una vez compilado y ejecutado el programa, con el comando display se mostrará por pantalla.

Para la programación en ASM de la función pixels_generator, es necesario pasar los argumentos a través de la pila para después capturarlos.

Además, el contenido de la pila tiene que ser igual al final de la función que antes de la llamada, y por ello se guardan los registros y la dirección del bottom pointer en la pila para crear a continuación el nuevo frame.

Para crear el doble bucle, es necesario el uso de dos etiquetas y dos registros que irán aumentando su valor hasta que adquieran el valor del final de la fila o columna.

En el caso de adquiera el valor de la última fila y columna, el recorrido habrá finalizado.

Para calcular el offset de la fila a modificar, es necesario realizar las siguientes instrucciones:

```
movw %di,%bx                #posición fila
imul dimension,%ebx
imul $DIMENSION_Z,%ebx
```

El registro %edi contiene la posición del a fila a modificar. A partir de ello, se le realizará la multiplicación entera con dimensión (número de píxeles) y DIMENSIÓN_Z (3, bytes structura RGB_data (green, red, blue)).

Para calcular el offset de la columna modificar, es necesario realizar las siguientes instrucciones:

```
movw %si,%cx                #posición columna
imul $DIMENSION_Z,%ecx
```

En este caso, el registro %esi contiene la posición de la columna. Una vez obtenidos ambos valores, su suma dará lugar al offset.

Finalmente, para el cálculo de la dirección efectiva en cada byte:

```
movb verde,%al              #intensidad verde
mov buffer_ptr,%edx
lea (%edx,%ebx),%edx
mov %eax,(%edx,%ecx) # dirección efectiva = M[buffer_ptr] + offset + posi_color
inc %ecx
```

El registro ebx contiene el offset.
El registro edx contiene M[buffer_ptr].
El registro %al contiene posi_color.

Por último, es necesario incrementar ecx para que pase al siguiente byte del struct RGB_data. El orden de declaración de la estructura RGB_data tiene que concordar con el valor de ecx en este caso.

Para finalizar el programa, se restablecen los valores de los registros y se recupera el antiguo frame debido a que la dirección de ebp estaba apilada anteriormente en la pila.