Memoria del proyecto de Tetris

Este trabajo lo hemos hecho Pablo Asensio Muñoz y Germán Gil Planes.

En esta memoria de proyecto vamos a comentar de manera rápida y concisa las funciones que hemos tenido que crear para el funcionamiento del videojuego Tetris.

Ejercicios de traducción:

- **Imagen_set_pixel:**

  Para traducir este procedimiento hemos usado 8 bytes de memoria en la pila, 4 de ellos para la dirección de retorno y de los 4 restantes usamos 1 para guardar el color de la imagen. Llamamos a la función imagen_pixel_addr directamente ya que los valores guardamos en los registros $a eran justamente los necesarios para esta función y una vez que tenemos la dirección donde va el color pixel la guardamos en esta dirección con sb. Finalmente, sacamos la dirección de retorno y liberamos la memoria en la pila.

(Para evitar ser repetitivo demos por asumido que siempre que tomemos bytes en la pila, los 4 primeros bytes serán para la dirección de retorno, al igual que al finalizar sacaremos el valor en la pila de la dirección de retorno)

- **Imagen_clean:**

  Para traducir este procedimiento hemos usado 20 bytes de memoria de la pila, donde hemos guardado los parámetros que venían con el procedimiento (img, fondo) y además hemos inicializado las variables x e y a 0 que son necesarias para los bucles for. En la etiqueta *for1_imagen_clean* comprobamos la si (y < img → alto), sacando sus respectivos valores de la pila, y en caso de cumplirse entrabamos en el segundo bucle, con etiqueta *for2_imagen_clean,* donde comprobamos si (x < img → ancho) y en caso de cumplirse la condición pusimos en los registros $a correspondientes los valores img, x, y y fondo para llamar al procedimiento *Imagen_set_pixel* , después de esto sumamos le sumamos 1 a la variable x y guardamos su valor. Una vez que se acaba el bucle for2 ponemos x a 0 y sumamos 1 a la variable y, y volvemos a realizar el bucle for1. Finalmente, cuando la condición del bucle for1 no se cumple, saltamos al *fin_for1_imagen_clean* donde liberamos la memoria en la pila
- **Imagen_init:**

  Para traducir este procedimiento hemos usado 4 bytes de memoria de la pila. Hemos guardado los valores ancho ($a1) y alto ($a2) en la dirección de memoria de la imagen correspondientes (ancho en la posición 0 y el alto 4 bytes más adelante) y como la dirección e imagen ya se encontraba en el registro $a0 solamente hemos tenido que mover a $a1 el valor del fondo que se encontraba en $a3 para así poder llamar al procedimiento *imagen_clean.*

- **Imagen_copy:**

  Para traducir este procedimiento hemos usado 24 bytes de memoria de la pila, donde hemos guardado los parámetros que venían con el procedimiento (dst, src), le hemos asignado a los campos ancho y alto de dst los valores del campo y alto de src y hemos guardado los valores en memoria de los campos de src, ya que nos harán falta luego, y además hemos inicializado las variables x e y a 0 que son necesarias para los bucles for. En la etiqueta *for1_imagen_copy* comprobamos si (y < src → alto), sacando sus respectivos valores de la pila, y en caso de cumplirse entramos en el segundo bucle, con etiqueta *for2_imagen_copy*, donde comprobamos si (x < src → ancho) y en caso de cumplirse la condición pusimos en los registros $a correspondientes los valores src, x e y para llamar a la función *Imagen_get_pixel* ,que nos devuelve un valor de un byte que en C sería la variable p. Después de esto ponemos en los registros $a los valores correspondientes sacándolos de la pila y finalmente llamamos al procedimiento *Imagen_set_pixel*, después de esto sumamos le sumamos 1 a la variable x y guardamos su valor. Una vez que se acaba el bucle for2 ponemos x a 0 y sumamos 1 a la variable y, y volvemos a realizar el bucle for1. Finalmente, cuando la condición del bucle for1 no se cumple, saltamos al *fin_for1_imagen_copy* donde liberamos la memoria en la pila.

- **Imagen_dibuja_imagen**

  Esta función recibe como parámetros la dirección de dos imágenes (dst,src) y las coordenadas (x,y) de la primera. Lo que se hace es, recorrer cada píxel de src y lo dibuja en el respectivo píxel de dst, guiándose por su localización a través de (x,y). Por tanto, de eso se trata su implementación: lo primero que se hace es guardar en la pila los valores que llegan de los parámetros y comienza la ejecución de los bucles. Para comprobar los bucles se realiza un bge en ambos casos:

  En el primer caso se comprueba que y, guardado en 24($sp), no sea mayor o igual que src->alto. En el segundo caso, que x, guardado en 20($sp), no sea mayor que src->ancho. El acceso a la dirección de src es fácil ya que lo hemos guardado en 4($sp). Para acceder tanto al ancho como al alto simplemente habría que hacer un lw. En el interior de los bucles se implementa lo que se propone. Solamente hay que sacar los valores que hemos guardado al comienzo en la pila y utilizarlos para mandárselos a imagen_get_pixel e imagen_set_pixel.

  Cabe destacar que imagen_get_pixel devuelve un pixel que se guarda en p mediante un sb. Después, ese pixel se comprobará si es igual a 0($0) con un beq y servirá de condición para la ejecución de imagen_set_pixel. El resto es actualizar las variables locales de los bucles.

- **Imagen_dibuja_imagen_rotada**

  Es similar a la anterior. El único cambio viene en el segundo y tercer parámetros que se pasan a imagen_set_pixel del segundo bucle (son los parámetros que se corresponden con las coordenadas del píxel). Esto es lógico porque se necesita hacer lo mismo que en imagen_dibuja_imagen salvo que en

esta función los píxeles se van dibujando de manera que al final la imagen esté rotada en vez de al derecho, como en la función anterior. Vamos a explicar detalladamente la diferencia:

En la función anterior, se pasaban como parámetros dst_x+x y dst_y+y. En esta se pasan dst_x + src->alto – 1 – y y dst_y + x.

y = 4($sp) ; x = 8($sp) ; dst_x = 28($sp) ; dst_y = 32($sp) ; src->alto = 12($sp)

Con estos datos almacenados en esas posiciones de la pila, los sacamos con un lw y hacemos las operaciones que hemos mencionado. Los movemos a $a1 y $a2 para que puedan ser movidos a imagen_set_pixel.

- **Nueva_pieza_actual**

  Lo primero que se hace es sacar la dirección de pieza_actual y pieza_siguiente para hacer imagen_copy y copiar la pieza siguiente en la pieza siguiente. Se genera entonces una pieza aleatoria y se guarda en 4($sp). Se llama a probar_pieza con esa pieza aleatoria y las coordenadas (8,0). Si devuelve 0, que se comprueba con un beqz, se salta a la etiqueta del fin_if_nueva_pieza_actual y se acabará la partida. Por el contrario, sacaremos de 4($sp) el valor de la pieza aleatoria y lo guardaremos en pieza_siguiente con imagen_copy, pasándole también de parámetros las coordenadas (8,0), que es el punto donde se genera la siguiente pieza.

- **Intentar_movimiento**

  Se reciben como parámetros x e y, así que ellas irán directamente a $a1 y $a2, respectivamente. También se hará un load adress para guardar en $a0 la dirección de pieza_actual. Con estos valores se llama a probar_pieza. Si devuelve 0, el beq lo manda al final de la función devolviendo $v0 = 0 (false). Por el contrario, se pone 1 en $v0 y se actualizan las coordenadas de pieza_actual_x y pieza_actual_y. Para ello, se sacan sus valores con un lw y se le suman x e y, las cuales hemos guardado anteriormente en 4($sp) y 8($sp).

- **Intentar_rotar_pieza_actual**

  Lo primero que se hace es poner la dirección de imagen_auxiliar en $a0 y almacenarlo en 4($sp) ya que lo usaremos a lo largo de la función. Le hacemos un load adress a pieza_actual y con lw se sacan el ancho y el alto. Estos datos, junto con 0 (PIXEL_VACIO) se pasan a imagen_init.

  Después, hay que hacer imagen_dibuja_imagen con pieza_rotada (4($sp)), con la dirección de pieza_actual (load adress) y con dos ceros.

  Lo siguiente será llamar a probar_pieza. Para ello, en $a0 volvemos a poner el valor de 4($sp) y, para $a1 y $a2, hacemos un load word a pieza_actual_x y pieza_actual_y. Si probar_pieza devuelve 0, el beqz nos lleva al final de la función. Si devuelve 1, entonces se hace imagen_copy de la pieza_rotada de

4($sp) en la dirección de pieza_actual (load adress de pieza_actual que se guarda en $a0 y load word de 4($sp) que se guarda en $a1).

- **Bajar_pieza_actual:**

  Antes de empezar con este procedimiento cabe destacar que las funciones que hemos hecho para los ***ejercicios de implementación*** se han implementado dentro de este procedimiento por lo tanto el código en ensamblador no será fiel a su versión en C ya que contendrá más aplicaciones, por ello vamos a comentar directamente el código final ya que sería absurdo comentar dos veces la misma función. Además, los procedimientos que aquí mencionemos y no expliquemos, están explicados con detalle más abajo.

  Comenzamos reservando 16 bytes en la pila en los que guardamos los valores de pieza_actual_x y pieza_actual_y le sumamos 1 al valor de pieza_actual_y y seguidamente llamamos a la función *intentar_movimiento*. En el caso de que no se pueda realizar el movimiento sacamos el valor de puntos y le sumamos 1 ya que si no se puede realizar el movimiento eso nos dice que la pieza ha llegado al final y se tiene que generar una pieza nueva. Además, comprobamos si hemos sumado 20 nuevos puntos en caso afirmativo aumentamos el ritmo de caída de las piezas disminuyendo el tiempo de pausa en un 10%. Continuamos dando a los registros $a los valores de campo, la dirección de pieza_actual, y los valores de pieza_actual_x y pieza_actual_y para llamar al procedimiento *imagen_dibuja_imagen.*

  Ahora hemos implementado un bucle que es necesario para comprobar si hay una línea llena y en ese caso eliminarla que es de la forma for(y =pieza_actual_y; y< pieza_actual_y+ pieza_actual->alto; y++). Para ello hemos implementado una función llamada *comprobar_linea_llena* que devuelve 1 si hay una línea llena en el caso de no haber ninguna línea llena salta a *fin_if2_bajar_pieza_actual* donde se le suma 1 a la variable y, y se vuelve al inicio del bucle. En el caso de haber una línea llena se suman 10 puntos y volvemos a comprobar si hemos sumado 20 nuevos puntos para modificar la velocidad de caída, y además asignamos a $a0 el valor de y para llamar al procedimiento que hemos implementado llamado *eliminar_linea* que como su nombre indica elimina la línea que está llena. Tras esto volveríamos a *fin_if2_bajar_pieza_actual* que ya hemos comentado que ocurre ahí. Finalmente, cuando se acaba el bucle for saltamos a *nueva_pieza_actual* y terminamos liberando la memoria de la pila.

Ejercicios de implementación:

- **Marcador de puntuación:**

  Para esta implementación comenzamos declarando dos strings llamados “strpuntuacion” (contenido del string: "Puntuación: ") y “strpuntos” (contenido del string: 256 bytes para la puntuación). Además, hemos tenido que crear una función llamada *imagen_dibuja_cadena* la cual ahora comentaremos, y hemos

añadido en la etiqueta B10_6 código para hacer posible que se viera por la pantalla.

Vamos a describir la función *imagen_dibuja_cadena:*

Esta función recibe 4 parámetros (pantalla, x, y, string) hemos reservado 20 bytes en la pila para poder manejar todos los datos. Hemos usado un bucle que comprobaba si se había llegado a la marca fin '\0' y en caso contrario nos hemos ayudado de *imagen_set_pixel* para poner en cada pixel su valor correspondiente y luego avanzamos en el eje x y en las posiciones del string, hasta llegar a la marca fin y acaba el bucle.

El código que hemos implementado en B10_6 ha sido el siguiente, después de la llamada a *imagen_dibuja_imagen*:

```
la      $a0, pantalla
li      $a1, 0
li      $a2, 0
la      $a3, strpuntuacion
jal     imagen_dibuja_cadena

lw      $a0, puntos
la      $a1, strpuntos
jal     integer_to_string

la      $a0, pantalla
li      $a1, 12
li      $a2, 0
la      $a3, strpuntos
jal     imagen_dibuja_cadena
```

En el que haciendo uso del procedimiento *integer_to_string* transformamos el valor de la puntuación en una cadena de string que posteriormente ponemos en pantalla gracias al procedimiento que hemos creado (*imagen_dibuja_cadena*).

- **Completando líneas**:

  En el procedimiento *bajar_pieza_actual* hemos usado una función llamada *comprobar_linea_llena*, de la cual vamos a hablar ahora.

  Esta función recibe como parámetro una posición de una línea y comprueba si está llena. Para ello tomamos 16 byte en la pila, definimos una variable x para avanzar horizontalmente, y sacamos el valor del ancho del campo. Hacemos un bucle de la forma for(x=0; x< campo->ancho; x++) en el que ayudándonos de *imagen_get_pixel* vamos comprobando si están los pixeles llenos o no. Si x llega a ser igual que el campo → ancho entonces la función devuelve 1, si en algún pixel tenemos que *imagen_get_pixel* nos devuelve 0 entonces la función devuelve 0. El uso de esta función ese restringe a lo que hemos mencionado en *bajar_pieza_actual*.

- **Ritmo de caída:**

  Para esta implementación hemos empezado declarando ritmo_caida de valor 1000 ya que es el tiempo asignado a la pausa. En el procedimiento *bajar_pieza_actual* ya hemos hablado de la implementación que hemos hecho y como cada 20 puntos disminuimos un 10% el valor de *ritmo_caida*. Únicamente

faltaría ver como se llega a la máxima velocidad de caída. Para ello tenemos que irnos a B23_2 donde hemos sustituido ble $t1, 1000, B23_2 por el siguiente código que determina como tiempo mínimo de espera 300 y este caso sería el mayor ritmo de caída:

```
lw      $t2, ritmo_caida
        blt     $t2, 400, if_menor300
        ble     $t1, $t2, B23_2                 # if (transcurrido < pausa) siguiente iteración
        j       B23_1
if_menor300:
ble     $t1, 300, B23_2         # if (transcurrido < pausa) siguiente iteración
```

Así, conseguiríamos ir aumentando la velocidad de caída y el juego ganaría más emoción.

- **Final de la partida:**

Ya hemos mencionado antes que, en nueva_pieza_actual, se ha modificado el código de manera que ahora se llama a probar_pieza pasándole de parámetros la pieza_aleatoria (la se convertirá en pieza_siguiente) y la coordenada (8,0), que es el punto donde se generan las piezas. Si probar pieza resulta ser 0, entonces finaliza la partida y se muestra el mensaje por pantalla. (acabar_partida = 1 y game_over = 1).

Lo primero de todo es saber que .data ha sido modificado incluyéndole el dibujo del cuadro y el mensaje con varios .ascii (imagen_fin_partida) y también se ha creado una nueva variable (game_over).

Yéndonos a la función de jugar_partida, se puede ver que hay la etiqueta B23_5 introduce unas líneas de código que lo llevan a terminar la partida. Ahí es dónde hacemos nosotros la distinción y comprobamos si game_over es igual a 1 con un beq. En ese caso, en $a0 se guarda la dirección de pantalla, en $a1 la dirección de imagen_fin_partida, $a2=1 y $a3=8. Con esto, se llama a imagen_dibuja_imagen y luego se hace clear_screen. Se vuelve a poner en $a0 la dirección de pantalla y con un imagen_print imprimimos la pantalla con el mensaje de imagen_fin_partida. Finalmente, se espera que el usuario pulse una tecla con read_character.

- **Eliminando líneas:**

A lo largo de la función llamamos a imagen_get_pixel y a imagen_set_pixel en varias ocasiones. Siempre se le pasan como primeros parámetros el campo (load adress a $a0), el valor de x (8($sp)) a $a1 y el valor de y (4($sp)) a $a2. Habrá pequeñas variaciones, que se van a mencionar.

En la implementación de la función hay 3 bucles. Los 2 primeros bucles se utilizan para recorrer de arriba abajo y de derecha a izquierda cada fila de la pantalla, trasladando cada fila a la de abajo. Cómo se recorre cada fila, lo que va a hacer el programa es guardar el píxel de la fila de arriba (imagen_get_pixel con y=y-1. Eso significa que el $a2=$a2-1) y ponerlo en el píxel de la fila de abajo. Para ello, se pasan

a imagen_set_pixel los mismos parámetros que hemos dicho al principio de la explicación, poniendo en $a3 el pixel que ha devuelto imagen_get_pixel. Así, se va recorriendo cada fila de la pantalla y poniendo la fila de arriba en la de abajo, tal y como queremos.

Por último, se eliminará la fila que se ha completado. Para ello, volvemos a hacer un bucle recorriendo de izquierda a derecha esa fila. Píxel por píxel se va haciendo imagen_get_pixel con los parámetros de siempre y al hacer imagen_set_pixel de ese píxel obtenido, le pasamos como parámetro en $a3 un 0, que representa a un píxel vacío.

- **Mostrar siguiente pieza**

Para poder desarrollar esta operación es necesario crear en el .data dos nuevos datos (pieza_siguiente_pantalla y recuadro_pieza_siguiente).

Lo primero que se hace en la línea 675 es copiar mediante un imagen_copy la imagen_pieza_siguiente en recuadro_pieza_siguiente, pasándole como parámetros ambas direcciones sacadas con un load adress a $a0 y $a1.

Después, se hace el imagen_dibuja_imagen para guardar la pieza siguiente en pieza_siguiente_pantalla. Se pone en lo que sería la coordenada (3,2) del recuadro. Para ello, en $a0 se pone la dirección de pieza_siguiente_pantalla, en $a1 la dirección de pieza_siguiente, en $a2 el 3 y en $a3 el 2.

Más adelante, hay que situar pieza_siguiente_pantalla en la pantalla principal mediante un imagen_dibuja_imagen. Se le van a pasar como parámetros los siguientes: en $a0 va a ir la dirección de pantalla, en $a1 irá la dirección de pieza_siguiente_pantalla. En $a2 va a ir lo que sería campo->ancho + 3 y en $a3 irá un 3. De este modo, el recuadro con la pieza siguiente se generará a la derecha del campo 3 píxeles para la derecha y 3 píxeles para abajo.