Mars

- Proyecto Programación: Tetris

Grupo1.3

- Marcelo Orenes Vera

Índice

Introducción:

-El entorno de trabajo:

La realización de las prácticas se ha desarrollado con el simulador Mars versión 4.1. Este programa realiza emulaciones del procesador MIPS basándose en el lenguaje ensamblador correspondiente a ese ISA, introducido por el usuario. Podemos introducir gran variedad de instrucciones, así como ajustar parámetros de ellas. Cuenta además con la posibilidad de ver en tiempo de ejecución el valor de cada uno de los 32 registros con los que cuenta su banco.

Sin duda su punto más importante no se trata de la posibilidad escribir en lenguaje ensamblador, si no de poder simular su funcionamiento para poder comprobar que no hay errores sin necesidad de compilarlo de forma externa al programa. La función de simulación cuenta con un modo de reproducción continuo y otro paso a paso (Step by Step) que lo hacemos funcionar para ver como avanza el programa a cada instrucción.

Este último será el método utilizado para reproducir nuestro programa, ya que nos permite ver por dónde va ejecutándose el programa y comprobar de una manera muy visual los saltos a subrutinas y los regresos.

Se nos da un esqueleto del fichero en ensamblador, un fichero en C con las funciones básicas ya implementadas, del que podremos servirnos de ayuda. Así mismo tenemos un pdf con las especificaciones y requerimientos de cada una de las funciones.

-Objetivos:

Esta práctica es el proyecto de un Tetris, del que nos dan un esqueleto que tenemos que terminar las funciones básicas requeridas para aprobar e implementar unas funciones avanzadas propuestas. Este el programa final que se pide, pero el objetivo en sí del proyecto es desarrollar nuestro conocimiento en ensamblador MIPS32 de modo que podamos desenvolvernos sin problemas al programar en este lenguaje.

-Desarrollo:

En esta documentación detallaremos todo el trabajo realizado desde que empezamos con el proyecto del Tetris hasta su finalización.

Los pasos son los seguidos en el pdf del proyecto, ayudándonos del Tetris.c cuando lo necesitemos. Así mismo comprobaremos que el resultado es el esperado a través del visor de registros en tiempo de ejecución y la simulación final.

2.Funciones Básicas:

```
1.  imagen_set_pixel:          # ($a0, $a1, $a2,$a3) = (img, x, y,color)
2.  addiu   $sp, $sp, -8
3.  sw      $ra, 0($sp)        # guardamos $ra porque haremos un jal
4.  sw      $s3, 4($sp)
5.  move    $s3,$a3
6.  jal     imagen_pixel_addr  # (img, x, y,char) ya en ($a0, $a1, $a2)
7.  sb      $s3, 0($v0)                # lee el pixel a devolver
8.  lw      $s3, 4($sp)
9.  lw      $ra, 0($sp)
10. addiu   $sp, $sp, 8
11. jr $ra
```

Esta función es muy parecida a imagen_get_pixel pero con la salvedad de que aquí una vez que tienes la dirección del pixel que necesitas, en vez de obtener el pixel que hay, lo estableces con un carácter que te han pasado como parámetro de $a3 y que hemos guardado en $s3 para preservarlo entre llamadas.

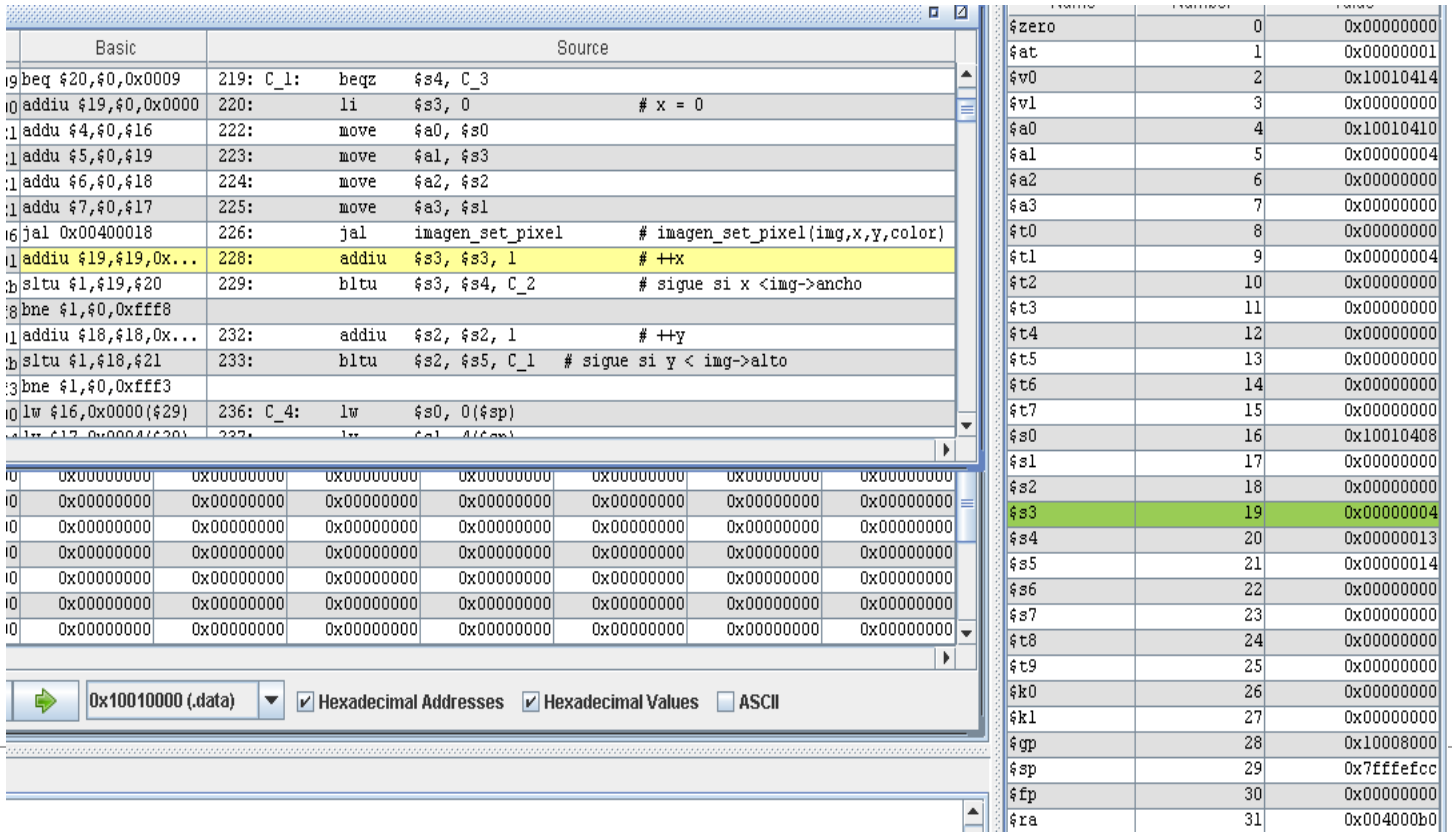
```
1.  imagen_clean:              # ($a0, $a1) = (img,color)
2.  addiu   $sp, $sp, -28
3.  sw      $ra, 24($sp)
4.  sw      $s5, 20($sp)
5.  sw      $s4, 16($sp)
6.  sw      $s3, 12($sp)
7.  sw      $s2, 8($sp)
8.  sw      $s1, 4($sp)
9.  sw      $s0, 0($sp)
10. move    $s1, $a1           # color
11. move    $s0, $a0           #img     # for (int y = 0; y<img->ancho; ++y) {
12. lw      $s5, 4($s0)        #img->alto
13. beqz    $s5, C_4
14. li  $s2, 0                 # y = 0   #  for (int x = 0; x < img->alto; ++x) {
15. lw      $s4, 0($s0)        #img->ancho
16. C_1:    beqz    $s4, C_3
17. li  $s3, 0                 # x = 0
18. C_2:
19. move    $a0, $s0
20. move    $a1, $s3
```

```
21. move    $a2, $s2
22. move    $a3, $s1
23. jal     imagen_set_pixel      # imagen_set_pixel(img,x,y,color)
24. addiu   $s3, $s3, 1           # ++x
25. bltu    $s3, $s4, C_2         # sigue si x <img->ancho
26. C_3:                          #  } // for x
27. addiu   $s2, $s2, 1           # ++y
28. bltu    $s2, $s5, C_1         # sigue si y < img->alto
29. C_4:                          # } // for y
30. lw      $s0, 0($sp)
31. lw      $s1, 4($sp)
32. lw      $s2, 8($sp)
33. lw      $s3, 12($sp)
34. lw      $s4, 16($sp)
35. lw      $s5, 20($sp)
36. lw      $ra, 24($sp)
37. addiu   $sp, $sp, 28
38. jr $ra
```

En esta función tenemos dos bucles for, uno externo (coordenada y) y otro interno (coordenada x), que se van recorriendo y va estableciendo para todos los pixeles de la imagen un determinado carácter, que se le pasa por parámetro. Aquí se guardan una serie de valores en la pila para no perderlos al estar constantemente llamando a imagen_set_pixel.

De estos valores que se quieren preservar los más importantes serían las variables de control de ambos bucles, pues si éstas se sobreescribiesen en el bucle podría pasar cualquier cosa. De hecho la primera vez que implementé esta función me equivoqué al hacer la instrucción de incrementar la variable de control de la coordenada y (almacenada en $s2) y el simulador nunca terminaba. Finalmente la solución la encontré después de estar mirando el banco de registros y avanzar instrucción por instrucción hasta que me di cuenta del fallo.

Tanto para esta, como para el resto de funciones la comprobación de que todo funcionaba bien se hace de



manera muy sencilla con el visor de banco de registros.

```
1.  imagen_init:                # ($a0, $a1, $a2,$a3) = (img,ancho,alto,color)
2.  addiu   $sp, $sp, -4
3.  sw      $ra, 0($sp)         # guardamos $ra porque haremos un jal
4.  move    $s3,$a3
5.  sw      $a1,0($a0)
6.  sw      $a2,4($a0)
7.  move $a1,$a3
8.  jal     imagen_clean        # (img,color) ya en ($a0, $a1)
9.  lw      $ra, 0($sp)
10. addiu   $sp, $sp, 4
11. jr $ra
```

Esta función inicializa una imagen, con un tamaño y un carácter, que se le pasan por parámetros. Para ello establece el alto y el ancho especificados como resolución de la imagen, a partir de su dirección de comienzo, ya que el ancho está almacenado en sus primeros 4 bytes y el alto en los 4 siguiente. Además llama a la subrutina imagen_clean para establecer el "color" de fondo de la imagen.

```
1.  imagen_copy:                # ($a0, $a1) = (dst,src)
2.  addiu   $sp, $sp, -28
3.  sw      $ra, 24($sp)
4.  sw      $s5, 20($sp)
5.  sw      $s4, 16($sp)
6.  sw      $s3, 12($sp)
7.  sw      $s2, 8($sp)
8.  sw      $s1, 4($sp)
9.  sw      $s0, 0($sp)
10. move    $s1, $a1        #src
11. move    $s0, $a0        #dst
12. lw      $s2,0($s1)      #src->ancho
13. sw      $s2,0($s0)      #dst->ancho=src->ancho
14. lw      $s3,4($s1)      #src->alto
15. sw      $s3,4($s0)      #dst->alto=src->alto
16.                         # for (int y = 0; y<img->alto; ++y) {
17. beqz    $s3, C_13
18. li  $s4, 0              # y = 0
19.                         #   for (int x = 0; x < img->ancho; ++x) {
20. C_6:    beqz    $s2, C_11
21. li  $s5, 0              # x = 0
22. C_8:    move    $a0, $s1
23. move    $a1, $s5
24. move    $a2, $s4
25. jal     imagen_get_pixel    # imagen_get_pixel(src,x,y)
26. move    $a0, $s0
27. move    $a1, $s5
```

```
28. move    $a2, $s4
29. move    $a3, $v0
30. jal     imagen_set_pixel    # imagen_set_pixel(dst,x,y,color)
31. addiu   $s5, $s5, 1         # ++x
32. bltu    $s5, $s2, C_8       # sigue si x <img->ancho
33. #  } // for j
34. C_11:
35. addiu   $s4, $s4, 1         # ++y
36. bltu    $s4, $s3, C_6   # sigue si y < img->alto
37. # } // for i
38. C_13:   lw      $s0, 0($sp)
39. lw      $s1, 4($sp)
40. lw      $s2, 8($sp)
41. lw      $s3, 12($sp)
42. lw      $s4, 16($sp)
43. lw      $s5, 20($sp)
44. lw      $ra, 24($sp)
45. addiu   $sp, $sp, 28
46. jr $ra
```

Para esta subrutina tenemos de nuevo dos bucles for que van recorriendo las coordenadas x e y de la imagen. Además inicia la imagen destino aportándole el ancho y alto de la imagen de la que se copia, por tanto no tiene porqué estar preinicializada. El bucle interno de la x, va recorriendo horizontalmente la imagen, obteniendo el carácter de la imagen origen y estableciéndolo en la de destino. Cuando llega al ancho de la imagen se incrementa en 1 la coordenada vertical y vuelve a recorrerse horizontalmente, así hasta que se llegue al alto de la imagen.

Comprobando con el visor de banco de registros podemos ver cómo van aumentando las variables de control de los bucles tanto interno como externo, hasta llegar al final.

La siguiente función implementada es muy similar a la anterior, con la diferencia de que la imagen destino debe estar inicializada previamente y su tamaño debe ser mayor que la de origen. De otra manera se saldría de la imagen destino, ya que se va recorriendo la estructura de origen de principio a fin y se imprime en otra con un desplazamiento relativo. Esta configuración también debe de ser la adecuada para que no se salga del rango, porque no se realiza comprobación alguna.

En resumen este código dibuja una imagen en otra (como ya se puede intuir por el título) con un desplazamiento determinado desde por su entrada.

```
1. imagen_dibuja_imagen:        # ($a0, $a1,$a2,$a3) = (dst, src, dst_x, dst_y)
2. addiu   $sp, $sp, -36
3. sw      $ra, 32($sp)
4. sw      $s7, 28($sp)
5. sw      $s6, 24($sp)
6. sw      $s5, 20($sp)
7. sw      $s4, 16($sp)
8. sw      $s3, 12($sp)
```

```
9.  sw      $s2, 8($sp)
10. sw      $s1, 4($sp)
11. sw      $s0, 0($sp)
12. move    $s1, $a1            #src
13. move    $s0, $a0            #dst
14. lw      $s2,0($s1)          #src->ancho
15. lw      $s3,4($s1)          #src->alto
16. move    $s6,$a2             #dst_x
17. move    $s7,$a3             #dst_y
18. beqz    $s3, C_25           # for (int y = 0; y<src->alto; ++y) {
19. li      $s4, 0              # y = 0
20. C_21:   beqz    $s2, C_25   #   for (int x = 0; x < src->ancho; ++x) {
21. li      $s5, 0              # x = 0
22. C_22:   move    $a0, $s1
23. move    $a1, $s5
24. move    $a2, $s4
25. jal     imagen_get_pixel    # imagen_get_pixel(src,x,y)
26. beqz    $v0,C_23
27. move    $a0, $s0
28. add     $a1, $s5,$s6
29. add     $a2, $s4,$s7
30. move    $a3, $v0
31. jal     imagen_set_pixel    # imagen_set_pixel(dst,x,y,color)
32. C_23:   addiu   $s5, $s5, 1 # ++x
33. bltu    $s5, $s2, C_22      # sigue si x src->ancho  #   } // for x
34. addiu   $s4, $s4, 1         # ++y
35. bltu    $s4, $s3, C_21      # sigue si y < src->alto   # } // for y
36. C_25:   lw    $s0, 0($sp)
37. lw      $s1, 4($sp)
38. lw      $s2, 8($sp)
39. lw      $s3, 12($sp)
40. lw      $s4, 16($sp)
41. lw      $s5, 20($sp)
42. sw      $s6, 24($sp)
43. sw      $s7, 28($sp)
44. lw      $ra, 32($sp)
45. addiu   $sp, $sp, 36
46. jr $ra
```

Aquí se recorren dos subrutinas temporales al igual que en imagen_copy. Sin embargo, además de las diferencias que ya he mencionado al principio de la función, aquí no se copia el pixel si es vacío, es decir, si el contenido es '/0' no se traslada a la imagen destino. De este modo no machaca el contenido que hubiese en ese lugar en la imagen destino, que podía tener o no contenido inicialmente.

Esto se consigue con la condición de la línea 26, en la que si imagen_get_pixel devuelve '/0' sigue recorriendo el bucle.

Además como ya he dicho, al establecer el carácter en la imagen destino lo hace con un desplazamiento relativo, esto lo hacemos con las instrucciones de las líneas 28 y 29, ya que antes de establecer el carácter, se le suma a la coordenada x e y el traslado requerido.

Dichos sumandos se le han pasado como parámetros junto a otros, y se han preservado entre llamadas gracias a la pila, ya que los hemos almacenado en registros que no se modifican entre llamadas, por convenio de ABI en MIPS, que hemos estado cumpliendo al implementar el resto de funciones.

Imagen_dibuja_imagen_rotada es la misma función que la anterior salvo que a la hora de establecer los pixeles en la imgen destino el desplazamiento relativo es diferente y ya nos viene dado por los apuntes del proyecto del Tetris. Simplemente mostraré el trozo de código que es diferente en ésta función y que corresponde al traslado (dest_x+alto - 1 -y, dest_y+x).

```
1.  Imagen_dibuja_imagen_rotada:
2.  . . .
3.  move    $a0, $s0
4.  subi    $t3,$s3,1           #Alto-1
5.  sub     $t3,$t3,$s4         #Alto-1-y
6.  add     $a1, $t3,$s6        # dest_x+alto - 1 -y
7.  add     $a2, $s5,$s7        # dest_y+x
8.  move    $a3, $v0
9.  jal     imagen_set_pixel    # imagen_set_pixel(dst,x,y,color)
```

```
1.  nueva_pieza_actual:         # (void) = (void)
2.  addiu   $sp, $sp, -4
3.  sw      $ra, 0($sp)
4.  jal     pieza_aleatoria
5.  move    $a1,$v0
6.  la      $a0,pieza_actual
7.  jal     imagen_copy
8.  lw      $t1,campo
9.  div     $t1,$t1,2
10. sw      $t1, pieza_actual_x
11. sw      $zero, pieza_actual_y
12. lw      $ra, 0($sp)
13. addiu   $sp, $sp, 4
14. jr $ra
```

Para nueva_pieza_actual se llama a pieza_aleatoria, que devuelve la dirección de memoria de una imagen, posteriormente se llama a imagen_copy con este parámetro como origen y con pieza_actual como destino. Además se establece la coordenada de pieza_actual_y como 0 y en pieza_actual_x se ponde el cociente de la división entre del ancho del campo por 2. Es decir la pieza siguiente aparecerá con la forma de pieza_aleatoria y por el centro del campo, por arriba. Posteriormente esta función será modificada para permitir previsualizar la siguiente pieza que va a aparecer. Esto vendrá reflejado en el apartado de funciones avanzadas.

```
1.  intentar_rotar_pieza_actual:
2.  addiu     $sp, $sp, -4
3.  sw        $ra, 0($sp)
4.  la        $a0,imagen_auxiliar
5.  la        $t1,pieza_actual
6.  lw        $a1,4($t1)
7.  lw        $a2,0($t1)
8.  li        $a3,0
9.  jal       imagen_init
10. la        $a0,imagen_auxiliar
11. la        $a1,pieza_actual
12. li        $a2,0
13. li        $a3,0
14. jal       imagen_dibuja_imagen_rotada
15. la        $a0,imagen_auxiliar
16. lw        $a1,pieza_actual_x
17. lw        $a2,pieza_actual_y
18. jal       probar_pieza
19. beqz      $v0,C_61
20. la        $a0,pieza_actual
21. la        $a1,imagen_auxiliar
22. jal       imagen_copy
23. C_61:     lw      $ra, 0($sp)
24. addiu     $sp, $sp, 4
25. jr $ra
```

Esta función realiza una serie de llamadas a funciones ya creadas, por lo que no conlleva más complejidad que pasarle los parámetros adecuados a las funciones adecuadas.

Para empezar se inicializa imagen_auxiliar con el tamaño de pieza_actual, de esta forma obtiene su ancho y su alto. A continuación se dibuja la pieza_actual rotada en la auxiliar, gracias a la llamada a imagen_dibuja_imagen_rotada. Una vez que tenemos en imagen_auxiliar la pieza rotada se comprueba si esa pieza se puede colocar en el campo, es decir, no choca con las paredes o con otra pieza ya colocada. Esto lo hacemos pasando la imagen auxiliar y sus coordenadas a probar_pieza. En caso afirmativo se copia la imagen_auxiliar como pieza_actual y ya tenemos la pieza rotada. Esta última condición la conseguimos con el salto condicional de la línea 19, en la que en caso negativo sale de la función y la pieza_actual sigue tal cual entró.

```
1.  intentar_movimiento:          # ($a0, $a1) = ( x, y)
2.  addiu     $sp, $sp, -12
3.  sw        $ra,8($sp)
4.  sw        $s1, 4($sp)
5.  sw        $s0, 0($sp)
6.  move $s0,$a0
```

```
7.  move $s1,$a1
8.  move $a2,$a1
9.  move $a1,$a0
10. la $a0,pieza_actual
11. jal     probar_pieza
12. beqz $v0,C_41
13. sw      $s0,pieza_actual_x
14. sw      $s1,pieza_actual_y
15. C_41:   lw      $s0, 0($sp)
16. lw      $s1, 4($sp)
17. lw      $ra, 8($sp)
18. addiu   $sp, $sp, 12
19. jr $ra
```

Para hacer la función intentar_movimiento, que recibe los parámetros x e y, que pretende conocer si la pieza actual puede colocarse en esa coordenada, la implementación es sencilla, ya que tenemos la función probar_pieza ya creada. Simplemente llamaremos a la función probar_pieza con los parámetros pieza_actual, x e y. Esto nos devolverá un valor de verdad, 1 ó 0. Si es afirmativo se guardan las coordenadas x e y como las coordenadas de pieza_actual.

```
1.  bajar_pieza_actual:
    . . .
2.  C_48: li    $t1,1
3.  C_49: lw    $a0,pieza_actual_x
4.  lw  $a1,pieza_actual_y
5.  add     $a1,$a1,$t1
6.  jal intentar_movimiento         #algoritmo basico de bajar_pieza_actual
7.  bnez $v0,C_51
8.  la  $a0,campo
9.  la  $a1,pieza_actual
10. lw  $a2,pieza_actual_x
11. lw  $a3,pieza_actual_y
12. jal imagen_dibuja_imagen
13. jal nueva_pieza_actual
```

Bajar_pieza_actual es una función básica, aunque posteriormente le he añadido más funcionalidades y llamadas que aquí no aparecen. Aquí simplemente llama a intentar_movimiento con los parámetros de pieza_actual_x y pieza_actual_y +1. De modo que si la respuesta es falsa, quiere decir que no puede colocar la pieza en esa coordenada y por tanto ha chocado con otra pieza o con el fondo, por tanto se dibujará la pieza actual en el campo y le llamará a nueva_pieza_actual.

3.Funciones avanzadas:

En la misma función de bajar_pieza_actual se añaden otras funcionalidades cuando la respuesta de intentar_movimiento resulta 0 (la pieza choca). Aquí podemos ver el código y mientras tanto explicaré lo que hacen:

Bajar_pieza_actual:

. . .

```
1.  jal  intentar_movimiento          #algoritmo basico de bajar_pieza_actual
2.  bnez $v0,C_51                     #Si la pieza no choca salta al final
3.  la   $a0,campo
4.  la   $a1,pieza_actual
5.  lw   $a2,pieza_actual_x
6.  lw   $a3,pieza_actual_y
7.  jal  imagen_dibuja_imagen         #Función básica ya comentada
8.  lw   $t1,puntuacion               #cuando llega una pieza al final se suma un punto
9.  addi      $t1,$t1,1               #Aquí se implementa parte de la función marcador
10. sw   $t1,puntuacion
11. jal  borrar_linea_llena           #Llama a la función que comprueba si hay líneas llenas.
```

Para comprobar si la partida se ha terminado solo hay que comprobar si la pieza que actualmente está saliendo no puede llegar a colocarse porque ya se ha alcanzado la cima, para ello compruebo si en el pixel donde salen las piezas está ocupado. Con salen por el centro de la primera fila, obtengo el pixel de dicha posición y si está ocupado entonces se termina la partida. **MENSAJE FIN**

```
12. C_53: la     $a0,campo            #comprueba si se ha llegado a la cima del campo.
13. lw   $a1,0($a0)
14. div  $a1,$a1,2
15. li   $a2,0
16. jal  imagen_get_pixel             #Obtiene el pixel de la primera línea, en el centro.
17. beqz      $v0,C_50                #Si el pixel está vacío salta a la siguiente función.
18. li   $t2,1
19. sb   $t2,acabar_partida           #Si el pixel está lleno se acaba la partida.
20. j    C_51
21. C_50: jal    nueva_pieza_actual   #aparece una nueva pieza
```

Para implementar la opción de que cada 50 puntos la velocidad aumente un 10%, lo que hacemos es que el retardo disminuya un 10%. Para ello se carga la puntuación actual y se le hace una división entera por 50. Si el cociente es diferente del que fue en la división anterior es que se ha cumplido la franja de 50 puntos y por tanto se carga el valor de tiempo_pausa, se multiplica por 9, se divide por 10 y se vuelve a almacenar.

```
22. lw   $t1,puntuacion               #cada 50 puntos la velocidad aumenta un 10%
23. li   $t2,50
24. div  $t1,$t2
25. mflo      $t1
```

```
lw  $t4,tiempo_pausa_cont
beq $t1,$t4,C_51
lw  $t3,tiempo_pausa
mul     $t3,$t3,9
div $t1,$t3,10
sw  $t1,tiempo_pausa
C_51:   lw      $ra, 0($sp)
addiu $sp, $sp, 4
jr  $ra
```

Quizá la función avanzada más difícil de todas sea la de comprobar si hay líneas completas y si es así, eliminarlas, para ello creamos dos funciones más:

```
1. comprobar_linea_llena: #($a0)=(línea a comprobar)
2. addiu    $sp, $sp, -20
3. sw       $ra, 16($sp)
4. sw       $s3,12($sp)
5. sw       $s2, 8($sp)
6. sw       $s1, 4($sp)
7. sw       $s0, 0($sp)
8. la $s0,campo
9. lw       $s1,0($s0)      #campo->ancho
10. move    $s2,$a0
11. li  $s3,0           #x=0
12. C_80:   beq $s3,$s1,C_81
13. move $a0,$s0
14. move $a1,$s3
15. move $a2,$s2
16. jal imagen_get_pixel
17. addi    $s3,$s3,1
18. bnez    $v0,C_80
19. C_81:   lw      $s0, 0($sp)
20. lw      $s1, 4($sp)
21. lw      $s2, 8($sp)
22. lw      $s3,12($sp)
23. lw      $ra, 16($sp)
24. addiu $sp, $sp, 20
25. jr $ra
```

Esta función comprueba si la línea que le llega como parámetro está llena o no, para ello la recorre de izquierda a derecha obteniendo el pixel de cada coordenada con un while. En cuanto un pixel esté vacío se sale del bucle y devuelve 0, en el caso de que llegue al final y no haya ningún pixel vacío devuelve el valor del último pixel comprobado (distinto de 0).

Esto lo usaremos para la siguiente función, que recorre todas las líneas de arriba abajo en busca de una línea llena. En cuanto la encuentra salta a un procedimiento para mover todo el campo que está por encima de ella una línea más abajo. Una vez que la elimina sigue realizando comprobaciones hasta llegar a la base.

```
26. borrar_linea_llena:
27. addiu     $sp, $sp, -24
28. sw        $ra,20($sp)
29. sw        $s4, 16($sp)
30. sw        $s3, 12($sp)
31. sw        $s2, 8($sp)
32. sw        $s1, 4($sp)
33. sw        $s0, 0($sp)
34. li  $s1,0                          #y=0 para comprobar la línea (pasarla como parámetro)
35. la $s0,campo
36. lw        $s4,0($s0)               #campo->ancho
37. C_52a:    lw $t0,4($s0)            #campo->alto
38. beq       $s1,$t0,C_52e            #Si las comprobaciones llegan a la base termina la función.
39. move      $a0,$s1
40. jal       comprobar_linea_llena    #si la hay líneas llenas devuelve un valor distinto de 0
41. beqz      $v0,C_52d
42. lw        $t1,puntuacion           #si la hay líneas llenas suma 10 puntos por cada una
43. addi      $t1,$t1,10               #Aquí se implementa parte de la función marcador
44. sw        $t1,puntuacion
45. move      $s2,$s1                  #y=lineal llena. Para realizar el desplazamiento.
46. C_52b:    beqz $s2,C_52d           #salta si hemos llegado a la cima del campo.
47. li  $s3,0                          #x=0 para realizar el desplazamiento
48. C_52c:    move    $a0, $s0
49. move      $a1, $s3
50. subi      $a2, $s2,1               #Se obtiene el pixel de la fila superior.
51. jal       imagen_get_pixel         # imagen_get_pixel(src,x,y)
52. move      $a0, $s0
53. move      $a1, $s3
54. move      $a2, $s2
55. move      $a3, $v0                 #Colocamos el mismo pixel en la fila actual
56. jal       imagen_set_pixel         # imagen_set_pixel(dst,x,y,color)
57. addiu     $s3, $s3, 1              # ++x
58. bltu      $s3, $s4, C_52c          # salta si x<src->ancho
59. subiu     $s2, $s2, 1              # --y (ascendemos a la fila superior)
60. j         C_52b                    #Salta para seguir copiando filas
61. C_52d:    addi     $s1,$s1,1       #Baja una fila para comprobarla
62. j         C_52a                    #Salta para seguir comprobando filas
63. C_52e:    lw       $s0, 0($sp)     #Desapilado y final de la función.
64. lw        $s1, 4($sp)
65. lw        $s4, 16($sp)
    . . .
```

En esta función básicamente recorremos de arriba abajo las filas del campo, comprobando si están llenas. Si se encuentra con una llena copia en dicha fila la fila inmediatamente superior, y después asciende una fila más, haciendo con ésta lo mismo hasta llegar a la cima. Una vez que se ha completado todo el desplazamiento necesario sigue comprobando si hay mas filas llenas hasta que llega a la base, si por el camino se encuentra alguna más repite el proceso de desplazamiento. Además añade 10 puntos al marcador cada vez que elimina una línea.

```
1.  Actualizar_pantalla:
2.  . . .
3.  lw      $a0,puntuacion
4.  la      $a1,buffer2
5.  jal     integer_to_string          #Traslada la puntuación a un buffer
6.  move    $a0,$s0
7.  la      $a1,mensaje_puntuacion
8.  li      $a2,0
9.  li      $a3,0
10. jal     imagen_dibuja_imagen       #imprime el mensaje previo a la puntuación
11. move    $a0,$s0
12. lw      $t5,mensaje_puntuacion     #mensaje puntuación->ancho
13. move    $a1,$t5
14. li      $a2,0
15. la      $a3,buffer2
16. jal     imagen_dibuja_cadena       #imprime la puntuación a continuación del mensaje previo.
```

Esta la parte de la implementación del marcador que afecta a la función actualiza_pantalla, aquí se dibuja el marcador en la línea 0 de la pantalla ($s0). Para poder dibujarlo en la pantalla, previamente hay que crear dos nuevas funciones, integer_to_string, que traslada la puntuación de un entero a una cadena y la función imagen_dibuja_cadena que permite imprimir una cadena en una imagen, como por ejemplo, pantalla.

```
1.  integer_to_string:      # ($a0, $a1) = (n, buffer2)
2.  move    $t7,$a1
3.  la      $t0, buffer     # char *p = buffer
4.                          # for (int i = n; i > 0; i = i /10) {
5.  move    $t1, $a0        # int i = n
6.  li      $t4,10
7.  B0_13:
8.  blez    $t1, B0_17      # si i <= 0 salta el bucle
9.  div     $t1, $t4        # i / 10
10. mflo    $t1             # i = i / 10
11. mfhi    $t2             # d = i % 10
12. addiu   $t2, $t2, '0'   # d + '0'
13. sb      $t2, 0($t0)     # *p = $t2
14. addiu   $t6,$t6,1
15. addiu   $t0, $t0, 1     # ++p
16. j       B0_13           # sigue el bucle
17. B0_17:
18. sb      $zero, 0($t0)   # *p = '\0'
19. move    $t1, $zero      # int i = 0
20. B0_14:
21. beq     $t1,$t6,B0_18   # si i >= num salta el bucle
22. subiu   $t0,$t0,1
23. lb      $t2, 0($t0)
24. sb      $t2, 0($t7)     # *p = $t2
25. addiu   $t7, $t7, 1     # ++p
```

```
26. addiu   $t1, $t1, 1          # i++
27. j       B0_14                # sigue el bucle
28. B0_18:
29. Sb      $zero, 0($t7)        # *p = '\0
30. B0_110:
31. jr      $ra
```

Esta función ya la habíamos implementado para la primera entrega de prácticas de ensamblador, aquí simplemente se traslada un numero pasado por parámetro a un buffer haciendo divisiones sobre la base decimal, quedando dicho numero pero del revés, posteriormente se almacena en otro buffer en posición correcta. Este último buffer es el de la dirección que se le pasa por parámetro.

A continuación se llama a la función imagen_dibuja_cadena que tiene por parámetros la imagen donde se va a dibujar, las coordenadas y el buffer donde está guardado el número a escribir.

```
32. imagen_dibuja_cadena: #{$a0,$a1,$a2,$a3}={img,x,y,buffer2}
33. addiu   $sp, $sp, -20
34. sw      $ra, 16($sp)
35. sw      $s3, 12($sp)
36. sw      $s2, 8($sp)
37. sw      $s1, 4($sp)
38. sw      $s0, 0($sp)
39. move    $s0, $a0             #dst
40. move    $s3, $a3             #buffer2
41. move    $s1,$a1              #dst_x
42. move    $s2,$a2              #dst_y
43. C_71:
44. lbu     $a3,0($s3)
45. beqz    $a3, C_74            # salta si es el final de la cadena
46. move    $a2,$s2
47. move    $a1,$s1
48. move    $a0,$s0
49. jal     imagen_set_pixel     # imagen_set_pixel(dst,x,y,color)
50. addiu   $s3, $s3, 1          # ++i
51. addiu   $s1, $s1, 1          # ++x
52. j       C_71
53. C_74:
54. lw      $s0, 0($sp)
55. lw      $s1, 4($sp)
56. lw      $s2, 8($sp)
57. lw      $s3, 12($sp)
58. lw      $ra, 16($sp)
59. addiu   $sp, $sp, 20
60. jr      $ra
```

Esta función se parece mucho a imagen_dibuja_imagen pero con la salvedad de que aquí solo se recorre la coordenada x, no la y. Se recorre la cadena de entrada con un bucle while que termina cuando se llega al final de la cadena /0. Mientras tanto se va escribiendo en la coordenada y que se ha establecido y la x va avanzando conforme avanza la cadena.

4.Funciones opcionales:

Para que podamos ver la pieza que aparecerá al caer la pieza actual hay que crear una nueva función y añadir un par de cosas en actualizar_pantalla y nueva_pieza_actual.

```
1.  nueva_pieza_siguiente:
2.  addiu   $sp, $sp, -4
3.  sw      $ra, 0($sp)
4.  jal     pieza_aleatoria         # $v0=pieza_aleatoria()
5.  move    $a1,$v0
6.  la      $a0,pieza_siguiente
7.  jal     imagen_copy
8.  lw      $ra, 0($sp)
9.  addiu   $sp, $sp, 4
10. jr              $ra
```

Esta función llama a pieza_aleatoria y almacena el resultado en pieza_siguiente, que es una imagen auxiliar para no sobreescribir la pieza _actual y poder mostrar las dos a la vez en pantalla.

```
11. nueva_pieza_actual:     # (void) = (void)
12. addiu   $sp, $sp, -4
13. sw      $ra, 0($sp)
14. la      $a1,pieza_siguiente
15. la      $a0,pieza_actual
16. jal     imagen_copy
17. lw      $t1,campo
18. div     $t1,$t1,2
19. sw      $t1,    pieza_actual_x
20. sw      $zero,pieza_actual_y
21. jal     nueva_pieza_siguiente
22. lw      $ra, 0($sp)
23. addiu   $sp, $sp, 4
24. jr      $ra
```

Ahora nueva_pieza_actual ya no llama a pieza_aleatoria, si no que copia el contenido de pieza_siguiente. Finalmente llama a nueva_piez_siguiente para tenerla almacenada para poder mostrarla también en pantalla y tenerla disponible para el siguiente ciclo.

```
1.  Actualizar_pantalla:
2.  . . .
3.  move    $a0,$s0
4.  la      $a1,pieza_siguiente
5.  lw      $t6,campo               #$t6=campo->ancho
6.  addi    $s6,$t6,3               #$s6=campo->ancho+3
7.  move    $a2,$s6                 #$s6 se pasa como parámetro dst_x
8.  li      $a3,4                   #como parámetro dst_y se pasa 4 (se imprime a altura 4)
9.  jal     imagen_dibuja_imagen    #imprime la imagen siguiente
10. move    $a0,$s0
```

```
11. la      $a1,recuadro_pieza_sig      #
12. subi    $a2,$s6,1                   #Como dst_x se pasa 1 posición menos que la pieza
13. li      $a3,3                       # Como dst_y se pasa 1 posición menos que la pieza
14. jal     imagen_dibuja_imagen        #imprime el recuadro de la imagen siguiente
```

En actualizar_pantalla se añade la impresión de la pieza actual a una altura concreta de 4 y a 3 posiciones del final del campo. El recuadro también se imprime en una coordenada relativa a esta de forma que encajen la pieza y el recuadro. Para todo ello nos hemos servido de la función ya definida que es imagen_dibuja_imagen, ya que tanto la pieza como el recuadro están definidas como tales.

Otra de las funciones opcionales es la de modificar el menú principal para que haya una opción de configuración del juego, se puede elegir el ancho y el alto de la pantalla, dentro de un determinado rango posible, además de tres niveles de dificultad (Principiante, Veterano y Chuck Norris).

Para implementar todo ello se ha modificado el main, y la función principal de jugar_partida. Además se ha creado una pequeña función a la que llamaremos para leer enteros.

```
1.  read_integer:
2.  li      $v0, 5
3.  syscall
4.  jr      $ra

5.  main:
6.  . . .
7.  jal     print_string        # print_string("Tetris\n\n 1 - Jugar\n 2 - Salir\n 3 -
                                        Configuracion\n\nElige una opcion:\n")
8.  jal     read_character      # char opc = read_character()
9.  beq     $v0,'3',B23_3
10. beq     $v0, '2', B23_1     # if (opc == '2') salir
11. bne     $v0, '1', B23_5     # if (opc != '1') mostrar error
12. B23_4: jal      jugar_partida   # jugar_partida()
13.         la      $a0,campo
14. . . .
```

Al comenzar el programa aparece el menú principal que vemos en negrita, si se elige la opción 3, la rutina salta a B23_3, cuya implementación vemos a continuación.

```
15. B23_3:
16. jal     clear_screen        # clear_screen()
17. la      $a0, str003         #Elige ancho
18. jal     print_string        # print_string
19. jal     read_integer        # char opc = read_integer()
20. move    $a1,$v0
21. la      $a0, str004         #Elige alto
22. jal     print_string        # print_string
23. jal     read_integer        # char opc = read_integer()
```

```
move    $a2,$v0
blt     $a1,17,B23_5        #Si ancho<17 salta un mensaje de error
mul     $t2,$a1,$a2         #Si la ancho x alto >1024 salta un mensaje de error
bgt     $t2,1024,B23_5
la      $a0, str005         #Elige nivel de dificultad
jal     print_string        # print_string
jal     read_integer        # char opc = read_integer()
move    $a0,$v0
blt     $a0,1,B23_5
bgt     $a0,3,B23_5
li      $v0,0
j       B23_4               #Jugar_partida()
```

Si el nivel de dificultad elegido es inferior a 1 o superior a 3, o las dimensiones de pantalla elegidas superan la memoria asignada salta a una etiqueta (B23_5) donde se imprime un mensaje de error:

("\Opción incorrecta. Pulse cualquier tecla para seguir.\")

El mismo mensaje de error que en el menú principal cuando se pulsaba un número que no estaba asignado. Finalmente se añade un detalle que es poner $v0 a 0 para que cuando se salte a jugar_partida saber si hemos llegado hasta allí desde el menú de configuración o desde la predefinida.

```
jugar_partida:
addiu   $sp, $sp, -12
sw      $ra, 8($sp)
sw      $s1, 4($sp)
sw      $s0, 0($sp)
```

Si se llega desde la opción de menú 1 define el ancho y el alto automáticamente, por el contrario si venimos de la 3 establece la configuración que se le ha pasado como parámetro.

```
beqz    $v0,B22_3           #Aquí es donde se distingue de que opción se ha llegado.
li      $a1, 19             #campo ancho=19
li      $a2, 20             #campo alto=20
j       B22_31
B22_3:                      #Aquí distinguimos el nivel de dificultad.
```

Si se ha pulsado la opción de configuración la rutina pasa por aquí. Primero distinguimos 3 niveles de dificultad, que se basan en reducir el tiempo de pausa cuanto más difícil. Si la opción de dificultad es principiante se salta directamente al comienzo y mantiene el tiempo_pausa como el predefinido (1000).

```
li      $t3,1
beq     $a0,$t3,B22_31
li      $t3,3
beq     $a0, $t3, B22_33        # if (opc == '3') saltara B22_33, tiempo_pausa=250
li      $t1,500
j       B22_323
B22_33: li      $t1,250
B22_323: sw $t1,tiempo_pausa
```

```
19. B22_31:
20. la      $a0, campo
21. li      $a3, 0
22. addi    $s0,$a1,7
23. addi    $s1,$a2,2
24. jal     imagen_init                 # Si menú=1, imagen_init(campo, 19, 20, PIXEL_VACIO)
```

El ancho y el alto del campo se establecen o bien por parámetro o por inicialización, dependiendo del la opción de menú escogida. Sin embargo la resolución de la pantalla se establece de forma relativa al campo, 2 píxeles más de alto y 7 más de ancho, de manera predefinida.

```
25. la      $a0, pantalla
26. move    $a1, $s0                    #Pantalla ancho
27. move    $a2, $s1                    #Pantalla alto
28. li      $a3, 32
29. jal     imagen_init                 #Si menú=1,  imagen_init(pantalla, 26, 22, ' ')
30. jal     nueva_pieza_siguiente       #Hay que añadir esta llamada para que funcione la
                                        función anterior de previsualizar la pieza siguiente.
31. jal     nueva_pieza_actual          # nueva_pieza_actual()
32. sb      $zero, acabar_partida       # acabar_partida = false
33. jal     get_time                    # get_time()
34. move    $s0, $v0                    # Hora antes = get_time()
35. jal     actualizar_pantalla         # actualizar_pantalla()
36. j       B22_2                       #Sigue con la función jugar_partida
37. . . .
```

Bonus. Teclas ocultas:

```
1.  procesar_entrada.opciones:
2.  .byte    'x'
3.  .space   3
4.  .word    tecla_salir
5.  .byte    'p'
6.  .space   3
7.  .word    tecla_pausa
8.  .byte    'm'
9.  .space   3
10. .word    tecla_secreta
11. .byte    'a'
12. .space   3
13. .word    tecla_izquierda
```

Mi objetivo era añadir un par de funciones con teclas sin usar, con la tecla "p" pretendo tener la opción de pausar el juego y la tecla "m" es una tecla secreta que te permite ascender en el campo por si el jugador se encuentra en un apuro y quiere ascender para poder colocar bien su pieza, la existencia de este último comando solo lo deben conocer unos pocos afortunados, para el resto solo tienen las teclas usuales y el pause.

Para implementar esto primero he añadido las referencias a estas teclas en procesar_entrada.opciones, a continuación hay que crear dos etiquetas con la implementación de lo que sucederá al pulsar dicha tecla.

```
1.  tecla_pausa:
2.  lw          $t1,tiempo_pausa
3.  li $t2,1000000000
4.  blt         $t1,$t2,C_90
5.  lw          $t3,tiempo_pausa_guardar
6.  sw          $t3,tiempo_pausa
7.  j   C_91
8.  C_90:sw     $t1,tiempo_pausa_guardar
9.  sw          $t2,tiempo_pausa
10. C_91:       jr      $ra
```

Al pulsar la tecla_pausa se carga el contenido del tiempo de pausa actual (la velocidad con la que bajan las piezas del Tetris), si ese tiempo es muy grande (1000000000) es que nos encontrábamos en pausa, por tanto se va a reanudar el juego, para ello se carga tiempo_pausa_guardado que es el retardo actual de descenso de las piezas, y lo almacena en tiempo_pausa. Por el contrario, si al pulsar la tecla el tiempo era un número "natural" lo que se hace es pausar el juego, para ello se le asigna un tiempo de retardo enorme, para que no avancen. Previamente se había guardado el tiempo_pausa para rescatarlo cuando se reanude el juego, ya que dependiendo del nivel de dificultad elegido, habrá un retardo u otro.

Por otro lado para la tecla_secreta simplemente se llama al procedimiento bajar_pieza_actual con un parámetro de entrada negativo, de esta manera en vez de descender, asciende. Previamente se ha modificado dicha función de la original, que no tenía ningún parámetro.

```
11. tecla_secreta:
12. addiu     $sp, $sp, -4
13. sw        $ra, 0($sp)
14. li $a0,-2
15. jal       bajar_pieza_actual     # bajar_pieza_actual()
16. lw        $ra, 0($sp)
17. addiu     $sp, $sp, 4
18. jr $ra
```

Por último hay que hacer una pequeña modificación en procesar_entrada:

```
1.  la      $s3, procesar_entrada.opciones
2.  li      $s4, 56                 # sizeof(opciones) // == 7 * sizeof(opciones[0]) == 7 * 8
3.  B21_1:  addu    $t1, $s3, $s1   # procesar_entrada.opciones + i*8
```

Mientras que antes el tamaño de las opciones era e 40 bytes (5 teclas), ahora al haber dos teclas más hay que darle algo más de tamaño, 56 bytes en total.

Bibliografía:

No se han consultado bibliografías externas al material docente de la asignatura, se han seguido las pautas de la guía de prácticas y de ahí hemos sacado la información necesaria para realizar nuestros ejercicios. En cierta medida se puede decir que nuestra bibliografía se basa en los apuntes utilizados en clase y en las explicaciones del profesor en las sesiones de laboratorio.

Para la realización de la documentación nos hemos servido de varias herramientas ofimáticas. La realización de los circuitos está implementada en el simulador Mars.

Conclusión:

Con este juego ya hemos puesto a prueba los conocimientos aprendidos en clase sobre lenguaje ensamblador MIPS32 y también nos hemos desenvuelto en el simulador Mars.

Durante el desarrollo del mismo se han afianzado los manejos de los bucles, saltos y condiciones entre otros, de este lenguaje, así como los convenios que se han tenido que seguir. Pero lo aprendido adquirido va más allá del MIPS32, al menos por mi parte, he adquirido más destreza a la hora de pensar al programar: tener que depurar fallos difíciles de encontrar, programar subrutinas específicas para un funcionamiento, además de conocer una manera de implementar capas de una imagen sobre otra y en sí el cómo implementar la estructura necesaria para un programa de este tipo. Sin olvidar el hábito adquirido de dedicación y trabajo para hacer un proyecto de programación.

Finalmente con estas prácticas vemos que también se puede programar a bajo nivel sin mayor dificultad, lo que puede servirnos en un futuro para depurar algún programa o para hacer algún algoritmo con un alto nivel de eficiencia.