Práctica 1

Proyecto Hardware 2019-2020

Eduardo Ruiz Cordón 764539
Martín Gascón Laste 764429

26-10-2019

Índice

1. Resumen

Para esta práctica de la asignatura Proyecto Hardware se nos pedía crear una biblioteca para controlar el timer 2 y una serie de funciones en ARM basadas en la función de mayor coste computacional del juego reversi facilitado por el profesorado para después realizar un pequeño estudio sobre ellas con el timer 2.

Hemos creado una biblioteca para controlar el timer 2 de la placa, permitiendo así iniciar el cronómetro, leer un tiempo y parar el cronómetro obteniendo el tiempo transcurrido desde que se inició. El cronómetro tendrá la precisión de microsegundos ya que será la unidad de tiempo que nos devolverá el cronómetro.

Previamente a la creación de la biblioteca del timer desarrollamos una versión implementada en ensamblador de la función patron_volteo desde la cual se llama a la subrutina implementada en C ficha_valida (ARM-C). Una vez que hemos optimizado el código ensamblador de la implementación ARM - C hemos diseñado una nueva versión de la función en la que también ficha_valida está implementada en código ensamblador (ARM-ARM).

Tras la optimización del código ARM-ARM diseñamos una función para comprobar la corrección de nuestras implementaciones, esta función consiste en realizar una serie de jugadas de prueba sobre un tablero que hemos diseñado para poder realizar un número representativo de jugadas diferentes. En total son 8 jugadas desde las cuales se buscará el patrón en todas las direcciones posibles y las comparará con las otras versiones de modo que si alguna de nuestras versiones es diferente a la implementación original (C - C) asumiremos que alguna de ellas es incorrecta.

Con las implementaciones funcionando ya correctamente lo siguiente fue medir los tiempos de ejecución por lo que creamos una función dedicada a la medición de tiempos donde se miden el tiempo en microsegundos que tarda en ejecutar patron_volteo en todas las direcciones posibles desde la posición (2, 3), para ello hemos utilizado el cronómetro Timer2.

Por último, calculamos el número de instrucciones y tamaño en bytes de las diferentes implementaciones usando la herramienta disassembly del debuger de eclipse, que nos muestra las direcciones de PC así que realizando dos una resta supimos el número de instrucciones de patrón volteo (lo mismo para ficha_valida). Puesto que las instrucciones son de 32 bits el tamaño del programa era igual al número de instrucciones por 32

2. Introducción

Para la realización de la práctica 1 de la asignatura Proyecto hardware se nos ha pedido investigar y diseñar una serie de funcionalidades para la placa S3CEV40 a partir del código del juego del reversi que se nos ha facilitado. Con la realización de esta práctica se busca que aprendamos a interactuar con una placa real, a gestionar la entrada y salida con dispositivos básicos, aprender a usar los periféricos de la placa, buscar información necesaria para la realización de un proyecto y gestionar el tiempo para su correcto desarrollo.

3. Objetivos

El objetivo de la práctica es integrar el código del reversi en la placa, desarrollando diferentes versiones para estudiar el impacto de las diferentes optimizaciones en la compilación sobre el tiempo de ejecución, tamaño y número de instrucciones mediante la modificación de la función más costosa en cómputo que en nuestro proyecto será patron_volteo. Además, buscamos realizar una verificación automatizada de la corrección de las funciones diseñadas e implementadas.
También con el objetivo de entender el funcionamiento de los periféricos de entrada se busca diseñar un cronómetro preciso que nos permita realizar las mediciones de rendimiento necesarias.

4. Metodología

4.1. Jerarquía de ficheros

La práctica se compone de los siguientes ficheros y subdirectorios:

- common: fuentes de la placa base.
- Debug: Carpeta generada por el compilador
- Fuentes del juego facilitados por los profesores (reversi8_2019.c)
- Timers (timer.c, timer2.c y timer.h)
- Fuentes en ensamblador (patron_volteo_arm_arm.asm y patron_volteo_arm_C.asm)
- main.c

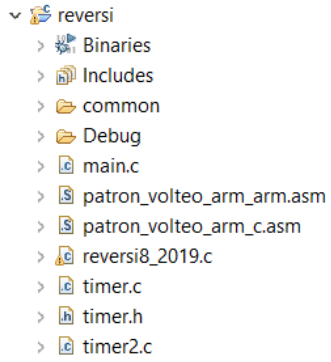


Figura 1

4.2. Estudio Proyecto inicial

Se han estudiado los ficheros facilitados por los profesores en el apartado anterior, para ello tuvimos que ejecutar paso a paso en alto nivel y en ensamblador el código suministrado. Al hacer la ejecución paso a paso en alto nivel fue necesario hacer uso de breakpoints a lo largo del código para poder observar el contenido de los registros y de la memoria y otras herramientas de debug.

4.3. Diseño de código en ensamblador

El siguiente paso que tuvimos que realizar fue el diseño de la función ya diseñada en reversi8_2019.c patron_volteo y reescribirla en una nueva versión en ensamblador y otra en ensamblador incluyendo la ficha válida también reescrita en ensamblador.

4.3.1 ARM-C

En primer lugar, se carga la dirección de la cima de la pila en el registro IP para poder acceder a los parámetros una vez se haya salvado el contexto. Después se guardan en la pila los registros que se van a modificar, guardando también el contenido de r0 y r1 [Marco_pila 1] debido a que posteriormente vamos a necesitar estos registros para pasarlos como parámetro a la función ficha_valida.
Ahora mismo los registros contienen los valores de las siguientes variables:

r0 → dirección a tablero
r1 → dirección a longitud
r2 → FA
r3 → CA

r0
r1
r4 a r7
lr
parámetros de llamada

Marco de pila 1

Como se puede observar faltan por guardar en registros algunos de los valores pasados como parámetro, para ello se realiza un load múltiple usando la dirección guardada en el registro ip:

r0 → dirección a tablero
r1 → dirección a longitud
r2 → FA
r3 → CA
r4 → SF
r5 → SC
r6 → color

Las siguientes acciones que se realizan son actualizar la fila y la columna que pasaremos como parámetro a ficha_valida y guardar sus valores en la zona de variables locales de la pila con un store múltiple, actualizando así el marco de pila [Marco_pila 2].

r0 → dirección a tablero
r1 → FA
r2 → CA
r3 → dirección a posicion_valida
r4 → SF
r5 → SC
r6 → color

Lo siguiente que se hace es cargar los parámetros que pasaremos a la subrutina ficha_valida, cargando en r1, r2 y r3 los valores de FA, CA y la posición en memoria reservada para posicion_valida respectivamente.

r2
r3
r0
r1
r4 a r7
lr
parámetros de llamada

Marco de pila 2

A continuación, se ejecuta el salto a ficha_valida, que mientras el retorno de esta no sea afirmativo y el color de la ficha a analizar sea diferente del valor devuelto por ficha_valida se realizarán las siguientes acciones [Figura 2]:

- Restaurar los valores de r2 y r3 guardados en la pila, actualizando el marco de pila [Marco_pila 3].
- Actualizar la fila y columna sumándoles SF y SC.
- Restaurar los valores iniciales de los registros r0 y r1 y actualizar el marco de pila [Marco_pila 4].
- Actualizar el valor de la longitud, para ello cargamos en r7 el contenido la dirección de memoria guardada en r1(ahora dirección de longitud), se incrementa en 1 y se guarda el contenido de r7 en memoria.
- Guardar en la pila r0 y r1 actualizando el marco de pila [Marco_pila 5].
- Guardar en la pila r2 y r3 actualizando el marco de pila [Marco_pila 6].
- Cargar en r1, r2 y r3 los parámetros que pasaremos a la subrutina ficha_valida de la misma forma que se ha explicado anteriormente, saltar a ficha_valida y al terminar volver al inicio del bucle (etiqueta “bucle:”).

r0
r1
r4 a r7
lr
parámetros de llamada

Marco de pila 3

r4 a r7
lr
parámetros de llamada

Marco de pila 5

r0
r1
r4 a r7
lr
parámetros de llamada

Marco de pila 4

r2
r3
r0
r1
r4 a r7
lr
parámetros de llamada

Marco de pila 6

```
bucle:
    ldr r3, =espacio                @
    ldr r3, [r3]                    @   Cargamos el contenido de posicion valida

    cmp r3, #1                      @   si la posicion no es valida
    bne salir_bucle                 @

    cmp r0, r6                      @   si el color es igual
    beq salir_bucle                 @
    ldmia sp!, {r2,r3}

    add r2, r2, r4                  @FA = FA + SF
    add r3, r3, r5                  @CA = CA + SC

    ldmia sp!, {r0,r1}              @   Recuperamos tablero y longitud

    ldr r7, [r1, #0]                @   |
    add r7, r7, #1                  @   |   longitud ++
    str r7, [r1, #0]                @   |

    stmdb sp!, {r0,r1}

    stmfd sp!, {r2, r3}
    mov r1, r2                      @   |   cargo los parametros a pasar a ficha
    mov r2, r3                      @   |
    ldr r3, =espacio

    bl  ficha_valida                @   |
    b bucle
```

Figura 2

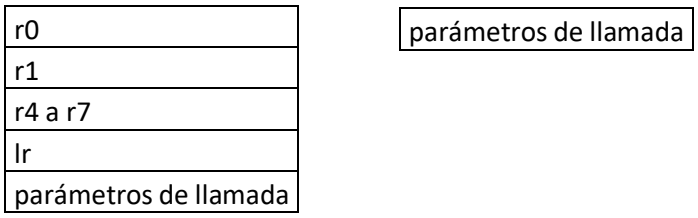
Una vez se sale del bucle se pasará a comprobar si ha habido patrón o no, para ello se realizará las siguientes comprobaciones:

- Si la posición es válida para ello se comprobará si posicion_valida es 1.
- Si el valor devuelto por la función ficha_valida es el mismo que la variable color. (antes de esto se restauran los registros r2 y r3 guardados en pila que corresponden a FA y CA [Marco_pila 7]).
- Se recupera los valores de r0 y r1 de la pila sin actualizar el marco de pila, se carga en r7 el valor de la longitud y se comprueba si es mayor que 0.

Si la primera condición no se cumple se saltará a la etiqueta "no_patron_carga_pila:" donde se restauran los valores de r2 y r3 actualizando el marco de pila [Marco_pila 7] y ejecutara las mismas instrucciones que en el caso de no haber patrón. Las acciones que se realizan en el caso de haber patrón y en el caso de no haberlo son muy similares ya que:

- En ambas se restaura el contexto actualizando el marco de pila [Marco_pila 8].
- En ambas se salta a la instrucción siguiente de la llamada a patron_volteo_arm_c.

Sin embargo, si hay patrón se pone a 1 el registro r0, pero si no lo hay se pone a 0 para indicar a su salida si ha habido patrón o no.



Marco de pila 8 *Marco de pila 7*

Las principales optimizaciones que se han realizado han sido las siguientes:

- Uso de stores y load múltiple para guardar u obtener valores de la pila con esto nos ahorramos mucho tiempo ya que los accesos a memoria son costosos y aquí hacemos 1 para varios valores.
- Guardamos en la pila los registros r0, r1, r2 y r3 para poder tener más registros libres para posibles cambios que se quieran hacer.
- Se usa el valor devuelto por la función ficha_valida en r0 devolviendo a este su valor correspondiente después de usarlo para eliminar una instrucción mov para guardarlo.

4.3.2 ARM-ARM

Los primeros pasos que seguimos son los mismo que en patron_volteo_arm_c, es decir, guardar la dirección de los parámetros en la pila en ip, salvar el contexto actualizando el marco de pila [Marco_pila 9], recuperar los valores de los parámetros de r4 a r6 con un load múltiple y actualizar FA y CA sumando SF y SC respectivamente. Ahora en patron_volteo_arm_c se prepararía el contenido de los registros para pasar los parámetros a ficha_valida pero debido a que en esta versión está incluida en nuestro código no se hacen esos pasos, directamente se pasa a realizar las acciones que haría ficha_valida.
ficha_valida al iniciar hace las siguientes comprobaciones:

- Si FA(r2) es menor que la dimensión del tablero (8).
- Si FA(r2) es mayor o igual que 0.

- Si CA(r3) es menor que la dimensión del tablero (8).
- Si CA(r3) es mayor o igual que 0.
- Si el valor de la posición (FA, CA) en el tablero es distinto de CASILLA_VACIA (0).

Marco_pila 9
r0
r1
r4 a r10
lr
parámetros de llamada

Marco de pila 9

Si se cumplen todas las condiciones anteriores se pondrá a 1 el contenido de la dirección de memoria reservado para posicion_valida. Si hay alguna de las condiciones que no se cumplen se guardará 0 en la dirección de memoria de posicion_valida y se carga 0 en r7 indicando así que la casilla está vacía.

Ahora se pasará a realizar las comprobaciones del bucle las cuales son las mismas que en patron_volteo_arm_c, es decir, comprobar que la posición es válida (posicion_valida == 1) y comprobar que el valor de la posición del tablero es distinto que la variable color (r7 != r6). Si se cumplen esas condiciones se modificará el valor de FA y CA sumando SF y SC respectivamente y se incrementará 1 la longitud [Figura 3], ahora se volverá a hacer las acciones que realizaría ficha_valida en patron_volteo explicadas anteriormente y saltará al inicio del bucle.

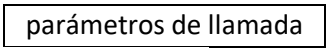
```
    add r2, r2, r4
    add r3, r3, r5

    ldr r7, [r1, #0]
    add r7, r7, #1
    str r7, [r1, #0]

    cmp r2, #8
    bge else1

    cmp r2, #0
    blt else1

    cmp r3, #8
    bge else1

    cmp r3, #0
    blt else1

    add r7, r3, r2, lsl #3
    ldrb r7, [r0, r7]

    cmp r7, #0
    beq else1

    ldr r9, =espacio
    mov r8, #1
    str r8, [r9, #0]

    b end_if1
else1:

    ldr r9, =espacio
    mov r8, #0
    str r8, [r9, #0]

    mov r7, #0

end_if1:
    b bucle
```

Figura 3

Una vez se sale del bucle se realiza las siguientes comprobaciones [Figura 4]:

- Si la posición es válida (posicion_valida == 1).
- Si el color de la posición del tablero es el mismo que la variable color (r7 == r6).
- Si la longitud es mayor que 0.

Las acciones que se realizan en el caso de que se incumplan o se cumplan las condiciones anteriores es muy similar. En ambos casos se recupera el contexto de la pila, se actualiza el marco de pila [Marco_pila 10], y en la última instrucción se salta a la instrucción siguiente de la que ha llamado a patron_volteo_arm_arm. Pero la diferencia entre las dos posibilidades es que antes de saltar si se cumplen las condiciones se pone a 1 el r0 indicando que ha habido patrón, mientras que si se incumplen se pone a 0 indicando lo contrario.



Marco de pila 10

```
end_if1:
    b bucle

salir_bucle:
    ldr r9, =espacio
    ldr r9, [r9]

    cmp r9, #1
    bne no_patron

    cmp r7, r6
    bne no_patron

    ldr r7, [r1, #0]
    cmp r7, #0
    bls no_patron

    ldmia sp!, {r0,r1,r4-r9, lr}
    mov r0, #1
    bx lr

no_patron:

    ldmia sp!, {r0,r1,r4-r9, lr}
    mov r0, #0
    bx lr

.end
```

Figura 4

La principal optimización que se consigue con esta versión de patron_volteo respecto al resto es que no se produce el salto a ficha_valida, sino que está incluida en el código, esto es una gran mejora ya que aparte de ahorrarnos las instrucciones de salto, también nos ahorra las instrucciones que estaban hechas para que recibiera los parámetros en los registros adecuados. Otra optimización que se ha realizado es el uso de load y store múltiples disminuyendo el número de instrucciones con acceso a memoria necesarias. Por último, destacar que hemos usado una operación de barrel shifter para realizar la multiplicación necesaria para calcular la posición del tablero de la que queríamos obtener el valor.

4.4 Verificador de jugadas

La función patron_volteo retorna un número entero que puede adoptar el valor de 0 si no se cumple el patrón y el valor de 1 si se cumple el patrón, este patrón depende de una serie de factores tales como el tablero, la dirección en la que se desea comprobar el patrón, el color y la dirección (o rumbo) en el que va a buscar el patrón.

Para verificar todas las posibles situaciones en las que nuestro patrón se ejecutará hemos diseñado un escenario más complejo al inicial [Figura 6] que te permite encontrar patrones en todas las direcciones.

Para calcular casos afirmativos en todas las direcciones se han testeado las siguientes jugadas: (1, 4), (2, 5), (3, 5), (6, 6), (6, 4), (5, 2), (3, 1), (2, 2) representadas en la [Figura 8].

```
const uint8_t fila_jugada[NUM_JUGADAS] =     {1, 2, 3, 6, 6, 5, 3, 2 };
uint8_t columnas_jugada[NUM_JUGADAS] =  {4, 5, 5, 6, 4, 2, 1, 2 };
```

Figura 5

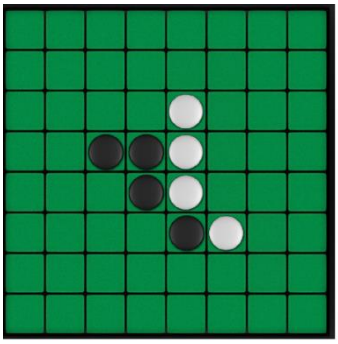


Figura 6

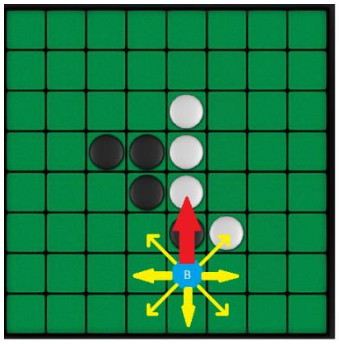


Figura 7

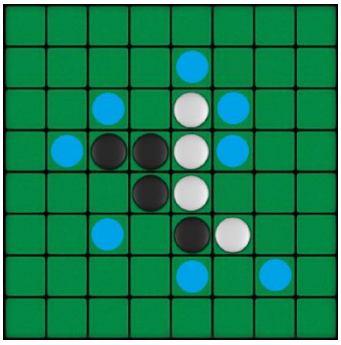


Figura 8

Del mismo modo cada una de estas jugadas tiene un color de ficha asociado que se guarda en el vector colores que contendrá el color de la ficha en el mismo índice del vector que los vectores de jugada.

Asumiendo que el conjunto de jugadas es lo suficientemente representativo aceptaremos la corrección de las funciones si las tres (ARM-ARM y ARM-C y C-C) retornan el mismo resultado para todas jugadas, si alguno es negativo el test retornará 0.

4.5 Timer 2

Para la realización del timer2 estudiamos el timer0 que se nos proporcionó y la documentación de la placa. El timer tiene tres funcionalidades: (empezar, leer y parar) además de su propia inicialización.

El timer consiste en un contador que se va decrementando hasta llegar a un valor inferior que provocará el reseteo del contador y generará una interrupción, aumentando así una variable que cuente el número de interrupciones [Figura 9].

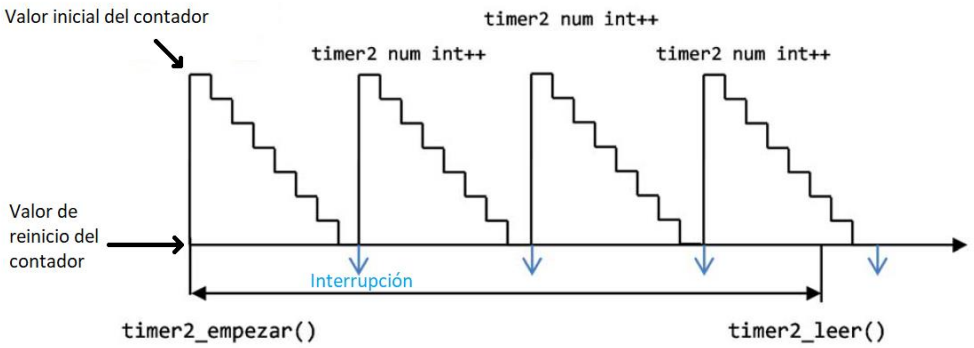


Figura 9 [Apuntes de la materia Proyecto hardware Universidad de Zaragoza].

Previamente a la explicación merece la pena explicar los registros que vamos a utilizar.

Nombre	Descripción	bits de interés
rINTMOD	Configura el modo de las interrupciones (IRQ = 0 FIQ = 1)	Timer2 -> Bit 11
rINTMSK	Habilita o deshabilita las líneas de interrupción (0 = habilitada, 1 = deshabilitada)	Timer2 ->Bit 11
rTCFG0	Registro de configuración, configura (entre otras cosas) el prescalado de los timers. (0 a 255)	Timer 2 a 4 -> Bits 15 a 18
rTCFG1	Registro de configuración, configura (entre otras cosas) el divisor de los timers.	Timer 2 -> Bits 9 a 11
rTCNT2	Valor inicial del contador	
rTCON	Registro de control del timer (manual update, auto-increment, start y output inverter)	Timer2 -> Bit 12 a 15 bit 12 -> start/stop bit 13 -> manual-update bit 14 -> output inverter bit 15 -> auto-increment

Para inicializar el timer primero configuramos las interrupciones y luego el contador:

- Configuración de las interrupciones:
  En primer lugar, configuramos el modo de las interrupciones que serán del tipo IRQ, para ello pondremos a 0 el undécimo bit del registro rINTMOD. En segundo lugar, tendremos que enmascarar todas las líneas menos la del timer dos por lo que enmascaramos el registro rINTMSK.

- Configuración del reloj:
  Ajustamos el prescalado y el divisor con los valores mínimos (prescalado = 0 y divisor = 1 / 2) para ajustar al máximo la precisión.

```
rINTMOD &= 0xFFFFF7FF;
rINTMSK &= ~(BIT_TIMER2);
```

Figura 10

Asignamos un valor inicial del contador en el registro rTCNTB2 (Valor desde el que empezará a contar de forma descendente el timer) al máximo valor posible para minimizar el número de interrupciones, y puesto que son 16 bits dedicados a este propósito será 0xFFFF (65535).

El valor 0 al registro rTCMPB2 determinará en qué punto de la cuenta descendente se deberá lanzar la interrupción.

```
rTCNTB2 = 65535;
rTCMPB2 = 0;
```

Figura 11

Además, modificamos el registro de configuración para habilitar el manual-update en primer lugar y el auto-increment en segundo lugar, hacemos el primer manual update para que cargue por primera vez los valores del contador y el auto-increment para que lo haga cada vez que lance una excepción.

```
rTCON = 0x2000;
rTCON = 0x8000;
```

Figura 12

La función leer nos da el número de microsegundos transcurrido y la hemos calculado mediante los siguientes pasos:

Puesto que la frecuencia de entrada es de 64MHz hemos calculado que primero debe pasar por el módulo de prescalado que la reducirá N +1 veces (En nuestro caso es 1, luego no se reduce). A continuación, pasará por el divisor de frecuencia que lo dividirá D veces (2 veces en nuestro caso) por lo que nuestra frecuencia de actualización de cada “tick” será de 64MHz/2.

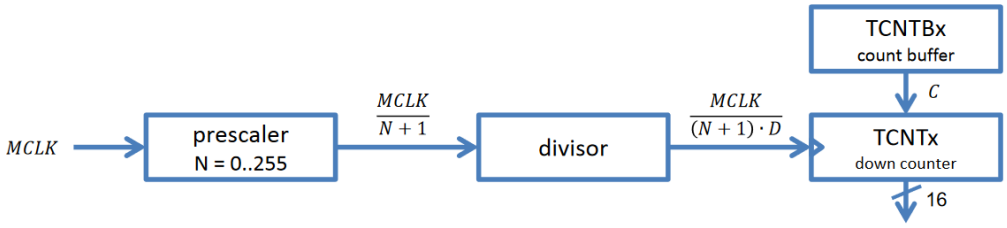


Figura 13 [Apuntes de la asignatura Programación de sistemas y dispositivos de la Universidad Complutense de Madrid]

Una vez que conocemos la frecuencia del contador es conveniente saber cuántos ticks han transcurrido, es decir, el número de ciclos completos (valor máximo del contador * número interrupciones) más los ticks que quedan desde la última interrupción:

N = (Tics_ciclo * num_interrupciones) + (Tics_ciclo – valor_contador)

Con los valores del número de ticks y la frecuencia de los tics podemos asumir que el tiempo transcurrido será N/32 µs.

En cuanto a la codificación caben a destacar dos cosas, en primer lugar el uso de desplazamientos en las operaciones tics_ciclo * num_interrupciones (num_interrupciones << 65535[1]) del mismo modo que al dividirlo por la frecuencia es lo mismo que desplazar 5 bits a la derecha (5 >> N).

En segundo lugar, se puede dar el caso que hagamos una lectura justo al saltar la interrupción, pero antes de que se incremente la variable (zona gris en el grafico X) por lo que en ese momento el número de tics sería 0 y el número de interrupciones sería también 0 (no ha transcurrido tiempo) luego sería un error.

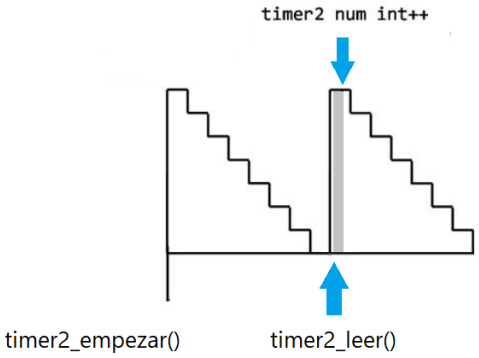


Figura 14

Para solucionarlo realizamos dos mediciones del número de interrupciones, lo que implica que si ambas mediciones son iguales (permanecemos en el mismo ciclo) añadiremos los tics desde el momento de la interrupción, en caso contrario únicamente haremos el cálculo con la segunda medida de num_interrupciones que será la correcta.

[1] -> 65535 se corresponde con el número de ticks por interrupción.

```
unsigned int timer2_leer(void){
    unsigned int int_antes = timer2_num_int;
    unsigned int ticks = rTCNTB2 - rTCNTO2;
    unsigned int int_despues = timer2_num_int;

    unsigned int total_ticks = (int_despues << 16);

    if (int_antes == int_despues) {
        total_ticks += ticks;
    }

    return total_ticks >> 5;
}
```

Figura 15

Por último la función parar() para el timer modificando el registro rTCON y llama a la función de lectura para retornar el tiempo que ha durado la medición.

4.6 Recolección de información.

El número de instrucciones se han realizado utilizando la herramienta disassembly del depurador de eclipse, registrando el valor de PC en la primera instrucción función de las dos subrutinas (patron_volteo y ficha_valida) y el último valor de cada una de las subrutinas.

Para la medición hemos realizado un método analisis_tiempos que calcula el tiempo transcurrido de la ejecución de patrón_volteo en todas las direcciones que parten de la posición (2, 3).

```
timer2_inicializar();
timer2_empezar();
for (i = 0; i < DIM; i++){
        int SF = vSF[i];
        int SC = vSC[i];
        int patron = patron_volteo_arm_c(tablero, &longitud, 2, 3, SF, SC, FICHA_NEGRA);
}
lectura_arm_c = timer2_parar();
Delay(lectura_arm_c);
```

Figura 16

4.7 Resultados

Se han realizado tres mediciones para cada una de las implementaciones: Número de instrucciones, tamaño de la implementación y el tiempo de ejecución de las diferentes versiones y en los diferentes grados de optimización.
El conjunto de los resultados los adjuntamos en el Anexo 1.

Dadas estas mediciones podemos hacer las siguientes apreciaciones:

- O0 No optimiza nada, por eso nuestros programas en ensamblador son mejor que los del compilador.
- O1 no optimiza demasiado, lo que implica que nuestros programas siguen siendo mejor que los del compilador.
- O2 optimiza en profundidad y ahí ya se aprecia como el tiempo de ejecución ya supera en rendimiento a nuestra implementación.
- Os optimiza en espacio, por eso como se puede comprobar el espacio reflejado en nuestras mediciones es menor al resto de optimizaciones y no tanto en cuanto al tiempo.
- O3 realiza inline expansión, lo que significa que copia el código de una función en el lugar de su llamada, esto en nuestro código, puesto que las llamadas se realizan dentro de un bucle, disminuye el número de llamadas a función y por tanto cambios de contexto, mejorando significativamente el rendimiento.

4.8 Problemas

- ### Chars

  Char es el tipo de dato en el que algunas variables estaban configuradas en el proyecto reversi, sin embargo, tal vez no es la mejor elección ya que, aunque char es un tipo de dato de un byte de tamaño no sabemos con seguridad cuántos bits se van a usar.

  Para solucionar esto hemos decidido cambiarlo por uint8_t e int8_t para enteros de 8 bits con y sin signo respectivamente.

- ### Uso volatile

  Todas las variables que se modifican en las rutinas de servicio de las interrupciones debemos marcarlas como volatile para que cada vez que acceda o escriba en ellas las guarde en memoria. Esto es importante porque si se guarda en registro podría modificar la variable la rutina de servicio de la interrupción y que el cambio no se llevara a cabo al terminar la rutina de servicio.

5. Conclusiones

De entre todas las implementaciones que hemos diseñado y entre todos los grados de optimización hemos llegado a la conclusión que la implementación más óptima es la implementación C - C en el grado de optimización O2. Es por esto que en futuras prácticas utilizaremos esta implementación.

6. Bibliografía

- Apuntes de la asignatura Programación de sistemas y dispositivos de la Universidad Complutense de Madrid. [http://www.fdi.ucm.es/profesor/mendias/PSyD/PSyD.html].
- Manual de usuario de la placa S3C44B0X proporcionada en Moodle.
- Apuntes de la asignatura Proyecto Hardware de la Universidad de Zaragoza.

ANEXO 1. Resultados de las mediciones.

-O0					
C - C					
Función	dirección inicio	dirección final	nº instrucciones	bytes	Tiempo
patron_voteo	0c00111c	0c00122c	68	272	
ficha_valida	0c001068	0c001118	44	176	
total			112	448	257
ARM - C					
Función	dirección inicio	dirección final	nº instrucciones	bytes	Tiempo
patron_voteo	0c000d50	0c000e0c	47	188	
ficha_valida	0c001068	0c001118	44	176	
total			91	364	230
ARM - ARM					
Función	dirección inicio	dirección final	nº instrucciones	bytes	Tiempo
patron_voteo	0c000c2c	0c000d48	71	284	
total			71	284	142

-O1					
C - C					
Función	dirección inicio	dirección final	nº instrucciones	bytes	Tiempo
patron_voteo	0c000cec	0c000dd8	59	236	
ficha_valida	0c000cb4	0c000ce8	13	52	
total			72	288	122
ARM - C					
Función	dirección inicio	dirección final	nº instrucciones	bytes	Tiempo
patron_voteo	0c000b44	0c000c00	47	188	
ficha_valida	0c000cb4	0c000ce8	13	52	
total			60	240	117
ARM - ARM					
Función	dirección inicio	dirección final	nº instrucciones	bits	Tiempo
patron_voteo	0c000a1c	0c000b38	71	284	
total			71	284	98

-O2					
C - C					
Función	dirección inicio	dirección final	nº instrucciones	bytes	Tiempo
patron_voteo	0c000d6c	0c000e30	49	196	
ficha_valida	0c000d34	c000d68	13	52	
total			62	248	80
ARM - C					
Función	dirección inicio	dirección final	nº instrucciones	bytes	Tiempo
patron_voteo	0c000aac	0c000b68	47	188	
ficha_valida	0c000d34	c000d68	13	52	
total			60	240	100
ARM - ARM					
Función	dirección inicio	dirección final	nº instrucciones	bits	Tiempo
patron_voteo	0c000984	0c000aa0	71	284	
total			71	284	87

-O3					
C - C					
Función	dirección inicio	dirección final	nº instrucciones	bytes	Tiempo
patron_voteo	0c00120c	0c0012d0	49	196	
ficha_valida	0c0011d4	0c001208	13	52	
total			62	248	47
ARM - C					
Función	dirección inicio	dirección final	nº instrucciones	bytes	Tiempo
patron_voteo	0c000bac	0c000c68	47	188	
ficha_valida	0c0011d4	0c001208	13	52	
total			60	240	106
ARM - ARM					
Función	dirección inicio	dirección final	nº instrucciones	bytes	Tiempo
patron_voteo	0c000984	0c000aa0	71	284	
total			71	284	84

-Os					
C - C					
Función	dirección inicio	dirección final	nº instrucciones	bytes	Tiempo
patron_voteo	0c000c54	0c000d0c	46	184	
ficha_valida	0c000c20	0c000c50	12	48	
total			58	232	125
ARM - C					
Función	dirección inicio	dirección final	nº instrucciones	bytes	Tiempo
patron_voteo	0c000aac	0c000b68	47	188	
ficha_valida	0c000c20	0c000c50	12	48	
total			59	236	126
ARM - ARM					
Función	dirección inicio	dirección final	nº instrucciones	bits	Tiempo
patron_voteo	0c000984	0c000aa0	71	284	
total			71	284	103