Universidad Zaragoza

Practica 1 Proyecto Hardware

Jorge Sanz Alcaine 680182

Introducción

En esta práctica se ha implementado de tres formas distintas un programa que ayuda al usuario en la resolución de sudokus. El programa comunica al usuario los números que podrían ser introducidos en cada una de las casillas, así como el número de casillas vacías. Para ello se han utilizado los lenguajes, C y ensamblador arm y thumb. El objetivo era comparar las distintas ejecuciones para encontrar el código más rápido, compacto y sencillo.

Resumen

Se nos pide diseñar dos métodos, sudoku_recalcular y sudoku_candidatos, pero escritos en C y ARM el primero y C, ARM y THUMB el segundo y comprobar su ejecución combinada.

Independientemente del lenguaje el primer método recorre cada una de las casillas, llama al segundo método y si la casilla no está vacía incrementa en uno un contador.

El segundo método debe comprobar los números que pueden ser introducidos para una casilla determinada. Para dispone de una máscara y tres bucles, en el primer bucle comprueba las casillas que coinciden en columna y elimina sus valores de la máscara. En el segundo bucle se realiza lo mismo pero con la fila. En el tercero se recorre la región en la que se encuentra la casilla para eliminar los valores en la máscara. Una vez se ha recorrido los tres bucles, la máscara ya contiene los números que pueden ser introducidos por el usuario en esa casilla, así que se actualiza la casilla marcándolos y se comprueba su valor para saber si está vacía o no.

Tras una análisis de los distintos códigos se ha determinado que el código en arm es el más compacto y rápido, mientras que el código en C es el más sencillo. Esto es debido a que ARM dispone de instrucciones más potentes que las de Thumb y además no necesita la pila para apilar valores de los registros puesto que dispone de suficientes. Al tratarse de una máquina, el compilador no es capaz de realizar una buena optimización sin comprometer la funcionalidad del código.

Código fuente apartado B:

Existen 6 funciones sudoku_recalcular, una por cada combinación de lenguaje. A continuación se muestra el código empleado en las diferentes versiones de la función.

sudoku_recalcular_arm_arm

Esta función recibe como parámetro en r0, una tabla de enteros de 16 bits que representa el tablero de un sudoku. Recorre cada una de sus casillas y llama a la función sudoku_candidatos escrita en arm para calcular los números disponibles en cada una de ellas y si la casilla está vacía o no. Por cada casilla no vacía la función incrementa un contador, que al final devuelve en r0.

La función usa 7 registros. R0 contiene el puntero a la cuadricula y es ahí donde se almacena el número de casillas no vacías una vez se ha recorrido el sudoku. R1 y R2 se utilizan para pasarle los parámetros a la función sudoku_candidatos_arm. R4 y R5 son el índice de las filas y las columnas respectivamente. R6 es el contador de celdas no vacías mientras R0 está siendo usado. R7 se usa como almacén temporal del puntero al sudoku, debido a que tras la llamada a sudoku_candidatos_arm R0 cambia.

```
sudoku_recalcular_arm_arm:
     STMFD   sp!, {r14,r4-r7} //Se apilan los registros que se van a usar
     mov r7,r0
     mov r6,#0                //vacías = 0
     mov r4,#0                //i = 0
SRAA_BucleI:
     mov r5,#0                //j = 0
SRAA_BucleJ:
     mov r0,r7                //Se preparan los paramentos de la función
     mov r1,r4
     mov r2,r5
     bl sudoku_candidatos_arm //se llama a la función
     cmp r0,#1                 //se comprueba el resultado
     addne r6,r6,#1           //se aumenta el contador si estaba vacía
     add r5,r5,#1              //se aumenta el índice del bucle y se vuelve
                              //a evaluar
     cmp r5,#9
     bne SRAA_BucleJ
     add r4,r4,#1             //se aumenta el índice del bucle y se vuelve
                              //a evaluar
     cmp r4,#9
     bne SRAA_BucleI

     mov r0,r6                //devuelve el número de casillas escritas
     LDMFD   sp!, {r14,r4-r7} //se desapilan los registros usados
```

sudoku_recalcular_arm_c

Esta función recibe como parámetro en r0, una tabla de enteros de 16 bits que representa el tablero de un sudoku. Recorre cada una de sus casillas y llama a la función sudoku_candidatos escrita en c para calcular los números disponibles en cada una de ellas y si la casilla está vacía o no. Por cada casilla no vacía la función incrementa un contador, que al final devuelve en r0.

La función usa 6 registros. R0 contiene el puntero a la cuadricula y es ahí donde se almacena el número de casillas no vacías una vez se ha recorrido el sudoku. R1 y R2 se utilizan para pasarle los parámetros a la función sudoku_candidatos_c. R4 y R5 son el índice de las filas y las columnas respectivamente. R6 es el contador de celdas no vacías mientras R0 está siendo usado. R7 se usa como almacén temporal del puntero al sudoku, debido a que tras la llamada a sudoku_candidatos_c R0 cambia.

```
      sudoku_recalcular_arm_c:
      STMFD   sp!, {r14,r4-r7} //Se apilan los registros que se van a usar
      mov r7,r0
      mov r6,#0               //vacías = 0
      mov r4,#0               //i = 0
SRAC_BucleI:
      mov r5,#0               //j = 0
SRAC_BucleJ:
      mov r0,r7               //Se preparan los paramentos de la función
      mov r1,r4
      mov r2,r5
      bl sudoku_candidatos_c   //se llama a la función
      cmp r0,#1                //se comprueba el resultado
      addne r6,r6,#1           //se aumenta el contador si estaba vacía
      add r5,r5,#1             //se aumenta el índice del bucle y se vuelve
                               //a evaluar
      cmp r5,#9
      bne SRAC_BucleJ
      add r4,r4,#1             //se aumenta el índice del bucle y se vuelve
                               //a evaluar
      cmp r4,#9
      bne SRAC_BucleI
      LDMFD   sp!, {r14,r4-r7}   //se desapilan los registros usados
      BX      r14
```

sudoku_recalcular_arm_thumb

Esta función recibe como parámetro en r0, una tabla de enteros de 16 bits que representa el tablero de un sudoku. Recorre cada una de sus casillas y llama a la función sudoku_candidatos escrita en thumb para calcular los números disponibles en cada una de ellas y si la casilla está vacía o no. Por cada casilla no vacía la función incrementa un contador, que al final devuelve en r0.

La función usa 7 registros. R0 contiene el puntero a la cuadricula y es ahí donde se almacena el número de casillas no vacías una vez se ha recorrido el sudoku. R1 y R2 se utilizan para pasarle los parámetros a la función sudoku_candidatos_thumb. R4 y R5 son el índice de las filas y las columnas respectivamente. R6 es el contador de celdas no vacías mientras R0 está siendo usado. R7 se usa como almacén temporal del puntero al sudoku, debido a que tras la llamada a sudoku_candidatos_thumb R0 cambia. R8 guarda la dirección de la función sudoku_candidatos_thumb +1, porque cuando se salta a una dirección impar con bx, el procesador cambia a modo thumb.

```
sudoku_recalcular_arm_thumb:
     STMFD   sp!, {r4-r8}  //Se apilan los registros que se van a usar
     mov r7,r0
     mov r6,#0             //vacías = 0
     mov r4,#0             //i = 0
SRAT_BucleI:
     mov r5,#0             //j = 0
SRAT_BucleJ:
     mov r0,r7             //Se preparan los paramentos de la función
     mov r1,r4
     mov r2,r5
     ldr r8,=sudoku_candidatos_thumb+1
     PUSH {r14}
     ldr r14,=regreso
     bx r8
regreso:
     POP  {r14}
     cmp r0,#1             //se comprueba el resultado
     addne r6,r6,#1        //se aumenta el contador si estaba vacía
     add r5,r5,#1          //se aumenta el índice del bucle y se vuelve
                           //a evaluar
     cmp r5,#9
     bne SRAT_BucleJ
     add r4,r4,#1          //se aumenta el índice del bucle y se vuelve
                           //a evaluar
     cmp r4,#9
     bne SRAT_BucleI

     mov r0,r6                //devuelve el número de casillas escritas
     LDMFD   sp!, {r4-r8}  //se desapilan los registros usados
     BX      r14
```

sudoku_recalcular_c_arm

Esta función recibe como parámetro en r0, una tabla de enteros de 16 bits que representa el tablero de un sudoku. Recorre cada una de sus casillas y llama a la función sudoku_candidatos escrita en arm para calcular los números disponibles en cada una de ellas y si la casilla está vacía o no. Por cada casilla no vacía la función incrementa un contador, que al final devuelve en r0.

El compilador es quien se encarga de utilizar los registros que crea oportunos.

```
////////////////////////////////////////////////////////////////////////
////////
// Recalcula todo el tablero (9x9)
// Retorna el número de celdas vacías
int sudoku_recalcular_c_arm(CELDA cuadricula[NUM_FILAS][NUM_COLUMNAS]) {
      int vacias=0;
      //para cada fila
      int i=0;
      while(i<NUM_FILAS){
            //para cada columna
            int j=0;
            while(j<NUM_FILAS){
                  //determinar candidatos
                  if(!sudoku_candidatos_arm(cuadricula,i,j)){
                        //actualizar contador de celdas vacías
                        vacias++;
                  }
                  j++;
            }
            i++;
      }
    //retornar el número de celdas vacías
    return vacias;
}
```

sudoku_recalcular_c_c

Esta función recibe como parámetro en r0, una tabla de enteros de 16 bits que representa el tablero de un sudoku. Recorre cada una de sus casillas y llama a la función sudoku_candidatos escrita en c para calcular los números disponibles en cada una de ellas y si la casilla está vacía o no. Por cada casilla no vacía la función incrementa un contador, que al final devuelve en r0.

El compilador es quien se encarga de utilizar los registros que crea oportunos.

```
////////////////////////////////////////////////////////////////////////
///////////
// Recalcula todo el tablero (9x9)
// Retorna el número de celdas vacías
int sudoku_recalcular_c_c(CELDA cuadricula[NUM_FILAS][NUM_COLUMNAS])
{
     int vacias=0;
     //para cada fila
     int i=0;
     while(i<NUM_FILAS){
          //para cada columna
          int j=0;
          while(j<NUM_FILAS){
               //determinar candidatos
               if(!sudoku_candidatos_c(cuadricula,i,j)){
                    //actualizar contador de celdas vacías
                    vacias++;
               }
               j++;
          }
          i++;
     }
    //retornar el número de celdas vacías
    return vacias;
}
```

sudoku_recalcular_c_thumb

Esta función recibe como parámetro en r0, una tabla de enteros de 16 bits que representa el tablero de un sudoku. Recorre cada una de sus casillas y llama a la función sudoku_candidatos escrita en thumb para calcular los números disponibles en cada una de ellas y si la casilla está vacía o no. Por cada casilla no vacía la función incrementa un contador, que al final devuelve en r0.

El compilador es quien se encarga de utilizar los registros que crea oportunos.

Para ejecutar la función en thumb, es necesario saltar a la dirección impar más próxima y superior a ella, es decir, a sudoku_candidatos_thumb+1.

```
////////////////////////////////////////////////////////////////
/////////////
// Recalcula todo el tablero (9x9)
// Retorna el número de celdas vacías
int sudoku_recalcular_c_thumb(CELDA cuadricula[NUM_FILAS][NUM_CO-
LUMNAS]) {
      int vacias=0;
      //para cada fila
      int i=0;
      while(i<NUM_FILAS){
            //para cada columna
            int j=0;
            while(j<NUM_FILAS){
                  //determinar candidatos
                  if(!(sudoku_candidatos_thumb+1)(cuadricula,i,j)){
                        //actualizar contador de celdas vacías
                        vacias++;
                  }
                  j++;
            }
            i++;
      }
     //retornar el número de celdas vacías
     return vacias;
}
```

sudoku_candidatos_arm

Esta función recibe como parámetro en r0, una tabla de enteros de 16 bits que representa el tablero de un sudoku, en r1 la fila en la que está la casilla y en r2 la columna en la que está la casilla. Utiliza uno de los registros como máscara de los números disponibles. Al principio, esa máscara indica que todos están disponibles. La función recorre la fila, la columna y la región de la casilla y va eliminando los números que se encuentra de la máscara. Una vez hecho esto, la máscara contiene los números disponibles en la casilla, de forma que carga la casilla en un registro, la actualiza y comprueba si está vacía para devolver 1 o 0.

La función usa 11 registros. R0 contiene el puntero a la cuadricula y es ahí donde se almacena el número de casillas no vacías una vez se ha recorrido el sudoku. R1 y R2 son las filas y columnas respectivamente, en las que se encuentra la casilla. R3 es la máscara de números candidatos. R4 es el valor de los números que va cargando. R5 es el índice de filas y de columnas en los dos primeros bucles, en el tercero es el índice de la fila en la región. R6 y R7 marcan la región en la que se encuentra. R8 es el índice de columnas en la región. R9 contiene a veces un 1, para poder hacer desplazamiento sobre él, mientras que otras veces, contiene la cuadricula en la que se encuentran los índices, para poder comprobar si siguen perteneciendo a la región. R10 contiene la dirección de una tabla usada para la optimización

```
sudoku_candidatos_arm:
     STMFD   sp!, {r4-r10} //Se apilan los registros que se van a usar
     mov r3,#0x0FF
     orr r3, r3, #0x100    //candidatos = #0x01FF
     mov r5,#0  // i = 0
     mov r9,#1
SCA_BucleI1:
     mov r4,#0             //se inicializa r4 a 0
     add r4,r0,r1,LSL #5   //32 Byte por fila
     add r4,r4,r5,LSL #1   //Dirección del elemento. 2 Byte por columna
                           //y 32 por columna
     ldrh r4,[r4]          //elemento de la celda
     mov r4,r4,LSR #12     //valor del elemento de la celda
     cmp r4,#0
     subne r4,r4,#1
     mvnne r4,r9,LSL r4    //r4 = ~(1<<(valor-1))
     andne r3,r4,r3        //candidatos = (~(1<<(valor-1))) $ candidatos
     add r5,r5,#1
     cmp r5,#9
     bne SCA_BucleI1
     mov r5,#0             // i = 0
SCA_BucleI2:
     mov r4,#0             //se inicializa r4 a 0
     add r4,r0,r5,LSL #5   //32 Byte por fila
     add r4,r4,r2,LSL #1   //Dirección del elemento. 2 Byte por columna
                           //y 32 por columna
     ldrh r4,[r4]          //elemento de la celda
     mov r4,r4,LSR #12     //valor del elemento de la celda
     cmp r4,#0
     subne r4,r4,#1
     mvnne r4,r9,LSL r4    //r4 = ~(1<<(valor-1))
     andne r3,r4,r3        //candidatos=(~(1<<(valor-1))) $ candidatos
     add r5,r5,#1
     cmp r5,#9
     bne SCA_BucleI2       //se aumenta el índice del bucle y se vuelve
                           //a evaluar
```

```
      mov r5,#0             // i = 0

      ldr r10,=index_reticula
      ldrb r6,[r10,r1]
      mov r5,r6

SCA_BucleI3:

      ldrb r7,[r10,r2]
      mov r8,r7

SCA_BucleJ:
      mov r9,#1
      mov r4,#0             //se inicializa r4 a 0
      add r4,r0,r5,LSL #5   //32 Byte por fila
      add r4,r4,r8,LSL #1   //Dirección del elemento. 2 Byte por columna
                            //y 32 por columna
      ldrh r4,[r4]          //elemento de la celda
      mov r4,r4,LSR #12     //valor del elemento de la celda
      cmp r4,#0
      subne r4,r4,#1
      mvnne r4,r9,LSL r4    //r4 = ~(1<<(valor-1))
      andne r3,r4,r3        //candidatos = (~(1<<(valor-1))) $ candidatos
      add r8,r8,#1
      cmp r8,#9
      beq salida
      ldrb r9,[r10,r8]
      cmp r7,r9
      bne salida
      b SCA_BucleJ          //se aumenta el índice del bucle y se vuelve
                            //a evaluar

salida:
      add r5,r5,#1
      ldrb r9,[r10,r5]
      cmp r6,r9
      beq SCA_BucleI3



      mov r4,#0             //se inicializa r4 a 0
      add r4,r0,r1,LSL #5   //32 Byte por fila
      add r4,r4,r2,LSL #1   //Dirección del elemento. 2 Byte por columna
y 32 por columna
      ldrh r5,[r4]          //elemento de la celda
      mov r6,r5,LSR #12
      orr r6,r5,r3
      strh r6,[r4]

      and r5,r5,#0xF000
      cmp r5,#0             //Devuelve 0 si r5=0, 1 en caso contrario
      movne r0,#1
      moveq r0,#0

      LDMFD   sp!, {r4-r10} //se desapilan los registros usados
      BX      r14
```

sudoku_ candidatos_c

Esta función recibe como parámetro en r0, una tabla de enteros de 16 bits que representa el tablero de un sudoku, en r1 la fila en la que está la casilla y en r2 la columna en la que está la casilla. Utiliza uno de los registros como máscara de los números disponibles. Al principio, esa máscara indica que todos están disponibles. La función recorre la fila, la columna y la región de la casilla y va eliminando los números que se encuentra de la máscara. Una vez hecho esto, la máscara contiene los números disponibles en la casilla, de forma que carga la casilla en un registro, la actualiza y comprueba si está vacía para devolver 1 o 0.

El compilador es quien se encarga de utilizar los registros que crea oportunos.

```
int sudoku_candidatos_c(CELDA cuadricula[NUM_FILAS][NUM_COLUMNAS],
                  uint8_t fila, uint8_t columna) {
      uint16_t candidatos=0x01FF;     // iniciar candidatos
      uint8_t valor;
      int i=0;
      while(i<NUM_FILAS){ // recorrer fila recalculando candidatos
            valor=celda_leer_valor(cuadricula[fila][i]);
            if(valor!=0){
                  candidatos=(~(1<<(valor-1))) & candidatos;
            }
            i++;
      }
      i=0;
      while(i<NUM_FILAS){ // recorrer columna recalculando candidatos
            valor=celda_leer_valor(cuadricula[i][columna]);
            if(valor!=0){
                  candidatos=(~(1<<(valor-1))) & candidatos;
            }
            i++;
      }
      //recorrer región recalculando candidatos
      int bloqueF=index_reticula[fila];
      i=bloqueF;
      while(index_reticula[i]==bloqueF){
            int bloqueC=index_reticula[columna];
            int j=bloqueC;
            while((index_reticula[j]==bloqueC) & (j<NUM_FILAS)){
                  valor=celda_leer_valor(cuadricula[i][j]);
                  if(valor!=0){
                        candidatos=(~(1<<(valor-1))) & candidatos;
                  }
                  j++;
            }
            i++;
      }
      cuadricula[fila][columna]=(cuadricula[fila][columna] & 0xFE00) |
                                                      candidatos;
     // Retornar indicando si la celda tiene un valor o esta vacía
     if ( (cuadricula[fila][columna] & 0xF000) != 0) {
      return TRUE;
     }else{
      return FALSE;
     }
}
```

sudoku_ candidatos_thumb

Esta función recibe como parámetro en r0, una tabla de enteros de 16 bits que representa el tablero de un sudoku, en r1 la fila en la que está la casilla y en r2 la columna en la que está la casilla. Utiliza uno de los registros como máscara de los números disponibles. Al principio, esa máscara indica que todos están disponibles. La función recorre la fila, la columna y la región de la casilla y va eliminando los números que se encuentra de la máscara. Una vez hecho esto, la máscara contiene los números disponibles en la casilla, de forma que carga la casilla en un registro, la actualiza y comprueba si está vacía para devolver 1 o 0.

La función usa 8 registros. Estos registros no se usan siempre para lo mismo por motivos de eficiencia, pero la mayor parte del tiempo se usan para lo siguiente. R0 contiene el puntero a la cuadricula y es ahí donde se almacena el número de casillas no vacías una vez se ha recorrido el sudoku. R1 y R2 son las filas y columnas respectivamente, en las que se encuentra la casilla, aunque en la búsqueda en región desempeñan muchas otras funciones. R3 es la máscara de números candidatos. R4 es el valor de los números que va cargando. R5 es el índice de filas y de columnas en los dos primeros bucles, en el tercero es el índice de la fila en la región. R6 y R7 marcan la región en la que se encuentra. R7 se usa también como el índice de columnas en la región.

```
.thumb
sudoku_candidatos_thumb:
      PUSH  {r4-r7}    //Se apilan los registros que se van a usar
      mov r3,#0x0FF
      mov r4,#1
      LSL r4,r4,#8
      orr r3,r3,r4     //candidatos = #0x01FF
      mov r5,#0        // i = 0
SCT_BucleI1:
      mov r4,#0        //se inicializa r4 a 0
      LSL r7,r1,#5
      add r4,r0,r7     //32 Byte por fila
      LSL r7,r5,#1
      add r4,r4,r7     //Dirección del elemento. 2 Byte por columna y 32
por columna
      ldrh r4,[r4]     //elemento de la celda
      LSR r4,r4,#12    //valor del elemento de la celda
      cmp r4,#0
      beq SCT_If1
      sub r4,r4,#1
      mov r6,#1
      LSL r6,r6,r4
      mvn r4,r6        //r4 = ~(1<<(valor-1))
      and r3,r4,r3     //candidatos = (~(1<<(valor-1))) $ candidatos
SCT_If1:
      add r5,r5,#1
      cmp r5,#9
      bne SCT_BucleI1 //se aumenta el índice del bucle y se vuelve
                      //a evaluar
```

```
      mov r5,#0        // i = 0

SCT_BucleI2:
      mov r4,#0        //se inicializa r4 a 0

      LSL r7,r5,#5
      add r4,r0,r7     //32 Byte por fila
      LSL r7,r2,#1
      add r4,r4,r7     //Dirección del elemento. 2 Byte por columna y
32 por columna
      ldrh r4,[r4]     //elemento de la celda
      LSR r4,r4,#12    //valor del elemento de la celda
      cmp r4,#0
      beq SCT_If2
      sub r4,r4,#1
      mov r6,#1
      LSL r6,r6,r4
      mvn r4,r6        //r4 = ~(1<<(valor-1))
      and r3,r4,r3     //candidatos = (~(1<<(valor-1))) $ candidatos
SCT_If2:
      add r5,r5,#1
      cmp r5,#9
      bne SCT_BucleI2 //se aumenta el índice del bucle y se vuelve
                      //a evaluar


      mov r5,#0        // i = 0
      ldr r7,=index_reticula
      ldrb r6,[r7,r1]
      mov r5,r6
```

```
SCT_BucleI3:
      ldr r7,=index_reticula
      ldrb r7,[r7,r2]
      PUSH {r7}
SCT_BucleJ:
      mov r4,#0        //se inicializa r4 a 0
      PUSH {r1,r2}
      LSL r1,r5,#5
      add r4,r0,r1     //32 Byte por fila
      LSL r1,r7,#1
      add r4,r4,r1     //Dirección del elemento. 2 Byte por columna y 32
por columna
      ldrh r4,[r4]     //elemento de la celda
      LSR r4,r4,#12    //valor del elemento de la celda
      cmp r4,#0
      beq SCT_If3
      sub r4,r4,#1
      mov r1,#1
      LSL r1,r1,r4
      mvn r4,r1        //r4 = ~(1<<(valor-1))
      and r3,r4,r3     //candidatos = (~(1<<(valor-1))) $ candidatos
SCT_If3:
      add r7,r7,#1
      PUSH {r3}
      cmp r7,#9
      beq SCT_Salida
      ldr r1,=index_reticula
      ldrb r3,[r1,r7]
      ldr r2,[sp,#12]
      cmp r3,r2
      bne SCT_Salida
      POP  {r3}
      POP  {r1,r2}
      b SCT_BucleJ     //se aumenta el índice del bucle y se vuelve a
evaluar
SCT_Salida:

      add r5,r5,#1
      ldr r1,=index_reticula
      ldrb r2,[r1,r5]
      cmp r6,r2
      bne SCT_Salida2
      POP        {r3}
      POP        {r1,r2}
      POP        {r7}
      b SCT_BucleI3
SCT_Salida2:
      POP        {r3}
      POP        {r1,r2}
      POP        {r7}
```

```
      mov r4,#0       //se inicializa r4 a 0
      LSL r7,r1,#5
      add r4,r0,r7    //32 Byte por fila
      LSL r7,r2,#1
      add r4,r4,r7    //Dirección del elemento. 2 Byte por columna y 32
                      //por columna
      ldrh r5,[r4]    //elemento de la celda
      LSR r6,r5,#12
      orr r5,r3
      strh r5,[r4]

      cmp r6,#0
      beq Vacio

      mov r0,#0
      POP  {r4-r7}    //se desapilan los registros usados
      BX      r14

Vacio:
      mov r0,#1
      POP  {r4-r7}    //se desapilan los registros usados
      BX      r14
```

Optimizaciones

Para calcular la región en la que se encuentra una casilla, lo que se hacía era dividir por tres las filas y las columnas. Sin embargo, la división es una operación muy costosa en arm y por eso se sustituyó por una tabla llamada index_reticula. Esa tabla contiene 9 enteros que corresponden a la region a la que pertenece ese número, por ejemplo la fila 7 pertenece a la region 3 y la columna 2 a la 1.

Para actualizar la máscara de candidatos lo que se hace es desplazar el 1 (valor-1) veces a la izquierda para situarlo en su posición. Una vez desplazado se niega y se actualiza la máscara realizando un and entre ellos dos.

Comprobación de resultados

Para comprobar que las funciones hacen lo que se les pide, se ha realizado una función en c que compara el resultado de una de las funciones con un sudoku con los candidatos ya resueltos. Esta función devuelve True si coinciden y False en caso contrario.

```
int comprobar(CELDA cuadricula[NUM_FILAS][NUM_COLUMNAS],
                    CELDA objetivo[NUM_FILAS][NUM_COLUMNAS]){
      int i=0;
      while(i<NUM_FILAS){
            int j=0;
            while(j<NUM_FILAS){
                  if(cuadricula[i][j]!=objetivo[i][j]){
                        return FALSE;
                  }
                  j++;
            }
            i++;
      }
      return TRUE;
}
```

Comparación de los resultados

Para medir las líneas de código tan solo se ha mirado el número de línea final y se le ha restado el número de línea inicial. Para medir el número de instrucciones necesarias para obtener los candidatos de una casilla se ha inicializado una variable a 0 y se le indica a eclipse que aumente en 1 uno esa variable cada vez que se ejecute una instrucción, se ejecuta el código necesario para obtener los candidatos y se consulta su valor. Para calcular el tiempo de ejecución en cada combinación se llama a la función 2000 veces con el siguiente código:

```
        mov r7,#0
buc:

        //Llamada a la función


        add r7,r7,#1
        cmp r7,#2000
        bne buc
```

El tiempo empleado en realizar el bucle se mide con un cronometro y se divide por 2000.

Los resultados obtenidos son:

Método	Tamaño código	Instrucciones casilla	Tiempo (ms)
sudoku_recalcular_arm_arm	25	-	2.25
sudoku_recalcular_arm_c	25	-	4.465
sudoku_recalcular_arm_thumb	28	-	3.085
sudoku_recalcular_c_arm	20	-	2.29
sudoku_recalcular_c_c	20	-	4.461
sudoku_recalcular_c_thumb	20	-	3.078
sudoku_candidatos_arm	91	404	-
sudoku_candidatos_c	48	1480	-
sudoku_candidatos_thumb	130	529	-

Problemas

El principal problema a la hora de realizar el trabajo fue que hacía mucho que no escribía en ensamblador y no recordaba bien algunas de las instrucciones.
Para solucionarlos repasé los apuntes de AOC1 y busque por internet aquellas instrucciones que no conocía.

Otro de los problemas fue la llamada a una función en thumb. Buscando por internet, encontré que había que sumarle 1 a la dirección de salto y saltar con bx.

También tuve problemas para reducir el número de registros en thumb a 8. Para las búsquedas en fila y columnas lo conseguí, sin embargo en la búsqueda en región tuve que utilizar una pila.

Conclusiones

El código en arm es el más rápido y compacto debido a que utiliza instrucciones de 32 bits y que ha habido una persona para optimizarlo. El código en thumb necesita más instrucciones porque no dispone de algunas de las instrucciones de arm y además se emplean instrucciones para pasar valores a la pila. El código en c puede parecer el más sencillo pero como es el compilador el que lo traduce, este no realiza las mejores optimizaciones y acaba siendo el más largo y costoso