ARM32

1)Codificar un programa en assembler ARM de 32 bits que imprima tres cadenas de caracteres(definidas en el propio programa) por la salida estándar, haciendo uso de una subrutina interna.

```
    .equ SWI_Exit, 0x11

    .equ SWI_Print_Str, 0x02

    .data



mensaje1:

    .asciz "hola! \n"

mensaje2:

    .asciz "segunda \n"

mensaje3:

    .asciz "tercera \n"

mensaje4:

    .asciz "cuarta\n"

mensaje5:

    .asciz "chau\n"

mensaje6:

    .asciz "cuaderno\n"



     .text

     .global _start
```

```
_start:
   ldr r0, =mensaje1
   bl   imprimir_cadena

   ldr r0, =mensaje2
   bl   imprimir_cadena

   ldr r0, =mensaje3
   bl   imprimir_cadena

   ldr r0, =mensaje4
   bl   imprimir_cadena
   ldr r0, =mensaje5
   bl   imprimir_cadena
   ldr r0, =mensaje6
   bl   imprimir_cadena
   swi SWI_Exit
imprimir_cadena:
   swi SWI_Print_Str
   mov pc,lr
```

A continuación, se definen seis cadenas de caracteres, que se almacenarán en la sección de datos (".data") de la memoria. Cada cadena se define con la directiva ".asciz", que define una cadena ASCII con el caracter nulo '\0' agregado automáticamente al final.

Luego, el programa comienza en la etiqueta "_start". Primero, se carga la dirección de la cadena "mensaje1" en el registro R0 y se llama a la función "imprimir_cadena" a través de la instrucción "bl" (branch and link).

Luego, se repite este proceso para cada cadena de caracteres, llamando a la función "imprimir_cadena" para imprimir cada una de ellas.

La función "imprimir_cadena" simplemente hace una llamada al sistema operativo a través de la instrucción "swi SWI_Print_Str" para imprimir la cadena de caracteres en la pantalla.

Finalmente, se llama a la función "swi SWI_Exit" para finalizar el programa y salir del programa.

En resumen, el código muestra cómo imprimir varias cadenas de caracteres en la pantalla y salir del programa utilizando llamadas al sistema operativo.

2)Ahora podes hacer un programa para imprimir 5 cadenas?

```
.data
    @ Definir las cadenas de caracteres
    @ Cada cadena debe terminar con un carácter nulo (0x00)
    @ Esto es importante para que la función de impresión sepa cuándo parar.
mensaje1:
    .asciz "Cadena 1\n"
mensaje2:
    .asciz "Cadena 2\n"
mensaje3:
    .asciz "Cadena 3\n"
mensaje4:
    .asciz "Cadena 4\n"
mensaje5:
    .asciz "Cadena 5\n"

    .text
    .global _start
```

```
_start:
    @ Llamar a la función de impresión para cada cadena
    @ La función de impresión espera la dirección de la cadena en r0
    @ y luego llama a la función de sistema "puts" para imprimir la cadena.
    @ El valor de retorno de "puts" se ignora.

    @ Imprimir la primera cadena
    ldr r0, =mensaje1
    bl imprimir_cadena

    @ Imprimir la segunda cadena
    ldr r0, =mensaje2
    bl imprimir_cadena

    @ Imprimir la tercera cadena
    ldr r0, =mensaje3
    bl imprimir_cadena
```

```
    @ Imprimir la cuarta cadena
    ldr r0, =mensaje4
    bl imprimir_cadena

    @ Imprimir la quinta cadena
    ldr r0, =mensaje5
    bl imprimir_cadena

    @ Salir del programa
    mov r7, #1    @ Cargar el número de llamada de sistema para "exit"
    mov r0, #0    @ Cargar el código de salida
    swi 0x0       @ Llamar a la función de sistema "exit"

imprimir_cadena:
    swi 0x02
    mov pc, lr
```

3)Codificar un programa en assembler ARM de 32 bits que recorra un vector de enteros y los imprima por la salida estándar agregando la

leyenda “PAR” a continuación de todos aquellos números que así lo sean.

```
    .equ SWI_Print_Int, 0x6B
    .equ SWI_Exit, 0x11
    .equ SWI_Print_Str, 0x69
    .equ Stdout, 1

    .data
array_origen:
    .word 1,2,5,6,7,8,9
array_length:
    .word 7
eol:
    .asciz "\n"
par:
    .asciz " PAR"
    .text
    .global _start
_start:
    ldr r0, =array_origen
    ldr r2, =array_length
    ldr r2, [r2]
```

```
loop:
    ldr r4, [r0]
bl imprimir
    add r0, r0, #4
    subs r2, r2, #1
    cmp r2, #0
    bne loop
    b exit
imprimir:
    stmfd sp!, {r0,r1,lr}
    ldr r0, =Stdout
    mov r1, r4
    swi SWI_Print_Int
    and r5,r4,#1 @Hago and en el último bit
    cmp r5, #0 @comparo para saber si es par o no
    bne impSig
    ldr r1, =par
    swi SWI_Print_Str
impSig:
    ldr r1, =eol
    swi SWI_Print_Str
    ldmfd sp!, {r0,r1,pc}
exit:
    swi SWI_Exit
    .end
```

En la sección .data, se define el vector de enteros array_origen, la longitud del vector array_length, la cadena de caracteres para agregar después de los números pares par, y la cadena de caracteres para imprimir una nueva línea eol.

En la sección .text, se define la etiqueta _start como el punto de entrada del programa. Se cargan los registros r0 y r2 con la dirección del vector de enteros y su longitud, respectivamente. Luego, se entra en un bucle que recorre el vector y llama a la subrutina imprimir para cada elemento.

La subrutina imprimir guarda los registros necesarios en la pila y carga r0 con el valor de la salida estándar. Luego, utiliza la función de sistema swi SWI_Print_Int para imprimir el número entero en r1 por la salida estándar. Después, se utiliza una operación AND para determinar si el número es par o impar, comparando el resultado con cero. Si es par, se carga r1 con la dirección de la cadena "PAR" y se imprime utilizando la función de sistema swi SWI_Print_Str. Finalmente, se carga r1 con la dirección de la cadena de nueva línea y se imprime.

En el bucle principal, se utiliza la instrucción subs para restar uno del registro r2 que contiene la longitud del vector, y se utiliza la instrucción bne para saltar al inicio del bucle si el registro r2 no es cero.

Al final del programa, se utiliza la función de sistema swi SWI_Exit para salir del programa.

4)Codificar un programa en assembler ARM de 32 bits que recorra un vector de enteros y genere un nuevo vector formado por elementos que resultan de sumar pares de elementos del vector original. Ej. vector original{1,2,5,6}, vector nuevo {3,11}

```
1    .equ SWI_Print_Int, 0x6B
2    .equ SWI_Exit, 0x11
3    .equ SWI_Print_Str, 0x69
4    .equ Stdout, 1
5    .data
6  array_origen:
7    .word 1,2,5,6
8  array_destino:
9    .word 0, 0, 0, 0
10 array_length:
11   .word 4
12 eol:
13   .asciz "\n"
14   .text
15   .global _start
16 _start:
17   ldr r0, =array_origen
18   ldr r1, =array_destino
19   ldr r2, =array_length
20   ldr r2, [r2]
```

```
loop_suma:
  ldr r4, [r0]
  add r0, r0, #4
  sub r2, r2, #1
  ldr r5, [r0]
  add r6, r4, r5 @Si es resta se cambia la operacion por subs
  str r6, [r1]
  add r0, r0, #4
  add r1, r1, #4
  sub r2, r2, #1
  cmp r2, #0
  bne loop_suma
  ldr r2, =array_destino
  ldr r3, =array_length
  ldr r3, [r3]
loop_mostrar:
  cmp r3, #0
  beq exit
  ldr r0, =Stdout
  ldr r1, [r2]
  swi SWI_Print_Int
  ldr r1, =eol
  swi SWI_Print_Str
  add r2, r2, #4
  sub r3, r3, #1
  b loop_mostrar
exit:
  swi SWI_Exit
.end
```

Este código es un programa en lenguaje ensamblador ARM de 32 bits que utiliza las interrupciones de software (SWI) para imprimir el resultado de la suma de pares de elementos de un vector en la salida estándar.

Primero se definen algunas constantes usando la directiva .equ, que asigna un valor a un símbolo. Estas constantes representan los números de las interrupciones de software que se van a utilizar, así como el identificador de la salida estándar.

A continuación, se definen algunas variables en la sección de datos (.data) del programa. El vector de entrada se llama "array_origen" y contiene los números que se van a sumar. El vector de salida se llama "array_destino" y se inicializa con ceros. También se define una variable llamada "array_length" que indica la longitud del vector de entrada. Por último, se define una cadena de caracteres llamada "eol" que contiene un salto de línea.

Después, el programa entra en la sección de código (.text) y se define el punto de entrada (_start) que es el primer lugar donde se ejecutará el programa. Dentro de esta sección, el programa carga las direcciones de memoria de los vectores de entrada y salida y la longitud del vector de entrada en los registros r0, r1 y r2, respectivamente.

El programa utiliza un bucle para sumar los elementos de dos en dos, empezando por los dos primeros elementos. Cada elemento se carga en un registro, se suman los dos registros y se guarda el resultado en el vector de salida. Luego, los registros y punteros se actualizan para pasar al siguiente par de elementos. Este proceso se repite hasta que se han sumado todos los pares de elementos.

Una vez que se han sumado todos los pares, el programa entra en otro bucle para mostrar los resultados en la salida estándar. En cada iteración del bucle, se carga la dirección de la salida estándar en el registro r0 y el siguiente número del vector de salida en el registro r1. A continuación, se llama a la interrupción de software SWI_Print_Int para imprimir el número en la salida estándar y se llama a la interrupción de software SWI_Print_Str para imprimir el salto de línea. Los registros y punteros se actualizan para pasar al siguiente elemento y se comprueba si se han mostrado todos los elementos del vector.

5)Codificar un programa en Assembler ARM de 32 bits que lea desde un archivo números enteros e imprima SU SUMATORIA POR LA SALIDA ESTANDAR.

```
1     .equ SWI_Open_File, 0x66
2     .equ SWI_Read_Int, 0x6C
3     .equ SWI_Print_Int, 0x6B
4     .equ SWI_Close_File, 0x68
5     .equ SWI_Exit, 0x11
6     .equ SWI_Print_Char, 0x00
7     .equ SWI_Print_Str, 0x69
8     .equ Stdout, 1
9
10    .data
11  filename:
12    .asciz "enteros.txt"
13  eol:
14    .asciz "\n"
15    .align
16  InFileHandle:
17    .word 0
18  sum:
19    .word 0
20
21    .text
22    .global _start
23
24  _start:
25      @ Abrir archivo
26      ldr r0, =filename @ nombre de archivo de entrada
27      mov r1, #0 @ modo: entrada
28      swi SWI_Open_File @ abre archivo
29      bcs InFileError @ chequear si hubo error
30      ldr r1, =InFileHandle @ cargar dirección donde almacenar el handler
31      str r0, [r1] @ almacenar handler
```

```
33  read_loop:
34      @ Leer entero de archivo
35      ldr r0, =InFileHandle
36      ldr r0, [r0]
37      swi SWI_Read_Int
38      bcs EofReached
39      @ Sumar valor al acumulador
40      ldr r1, =sum
41      ldr r2, [r1]
42      add r2, r2, r0
43      str r2, [r1]
44      b read_loop
45
46  InFileError:
47  EofReached:
48      @ Imprimir la sumatoria
49      ldr r0, =Stdout
50      ldr r1, =sum
51      ldr r1, [r1]
52      swi SWI_Print_Int
53      @ Cerrar archivo y salir
54      ldr r0, =InFileHandle
55      ldr r0, [r0]
56      swi SWI_Close_File
57      swi SWI_Exit
58
59  .end
```

En esta sección se definen los datos y variables que utiliza el programa. En primer lugar, se define una cadena de caracteres filename que contiene el nombre del archivo que se va a leer. Luego, se define una cadena de caracteres eol que se utiliza para imprimir un salto de línea en la salida estándar. La instrucción .align se utiliza para alinear las direcciones de memoria de las variables que siguen a un múltiplo de 4 bytes. Luego se define una variable InFileHandle para almacenar el handler del archivo que se abre, que inicialmente se pone a cero. Finalmente, se define una variable sum para almacenar la sumatoria de los números enteros leídos del archivo, que también se inicializa en cero.

En esta sección se abre el archivo de entrada filename para lectura y se almacena el handler del archivo en la variable InFileHandle. Se utiliza la instrucción ldr para cargar la dirección de memoria de la cadena filename en el registro r0. Luego se utiliza la instrucción mov para colocar el modo de apertura del archivo en el registro r1, que en este caso es 0 para indicar que se abre el archivo en modo de lectura. La instrucción swi se utiliza para llamar a la función del sistema operativo SWI_Open_File, que abre el archivo y devuelve su handler. Si hay un error al abrir el archivo, la bandera de carry (indicada por bcs) se establece y el programa salta a la etiqueta InFileError. Si el archivo se abre correctamente, se carga la dirección de la variable InFileHandle en el registro r1.

se tiene una sección .text que contiene el código principal del programa. El símbolo _start marca el inicio de la sección. Primero, se carga en r0 la dirección de memoria donde se encuentra el nombre del archivo que se quiere abrir. Luego, se mueve el valor 0 a r1 para indicar que se quiere abrir el archivo en modo de lectura. Se utiliza la llamada al sistema SWI_Open_File para abrir el archivo y se verifica si hubo algún error usando bcs InFileError.

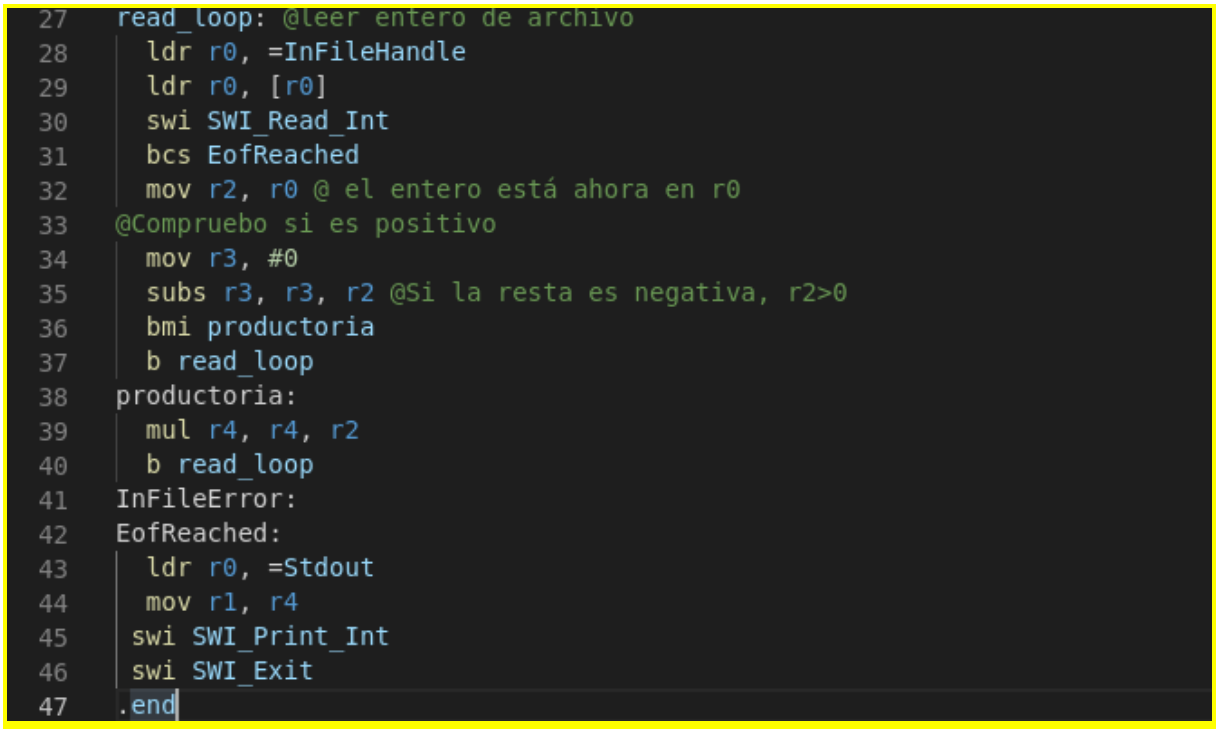
Si no hubo errores al abrir el archivo, se carga la dirección de memoria donde se almacenará el identificador del archivo en r1 y se almacena el valor de retorno de SWI_Open_File en la dirección de memoria apuntada por r1.

Luego, se inicializa el acumulador r4 con el valor 1. Se comienza un bucle que leerá un entero del archivo usando SWI_Read_Int y se almacenará en r0. Si se alcanzó el final del archivo, se sale del bucle. Si el entero leído es positivo, se agrega su valor al acumulador r4. Luego, se vuelve a empezar el bucle para leer el siguiente entero.

Una vez que se leyeron todos los enteros del archivo, se carga el valor 1 en r0 para indicar que se quiere escribir en la salida estándar. El acumulador r4 se mueve a r1 y se llama a la función SWI_Print_Int para imprimir el resultado por pantalla. Finalmente, se utiliza la llamada al sistema SWI_Exit para salir del programa.

6)Codificar un programa en Assembler ARM de 32 bits que lea desde un archivo números enteros e imprima por la salida estándar la productoria de aquellos números que sean positivos.

```
1    .equ SWI_Open_File, 0x66
2    .equ SWI_Read_Int, 0x6C
3    .equ SWI_Print_Int, 0x6B
4    .equ SWI_Close_File, 0x68
5    .equ SWI_Exit, 0x11
6    .equ SWI_Print_Char, 0x00
7    .equ SWI_Print_Str, 0x69
8    .equ Stdout, 1
9    .data
10   filename:
11     .asciz "enteros.txt"
12   eol:
13     .asciz "\n"
14     .align
15   InFileHandle:
16     .word 0
17     .text
18   .  global _start
19   _start: @abrir archivo
20     ldr r0, =filename @ nombre de archivo de entrada
21     mov r1, #0 @ modo: entrada
22     swi SWI_Open_File @ abre archivo
23     bcs InFileError @ chequear si hubo error
24     ldr r1,=InFileHandle @ cargar dirección donde almacenar el handler
25     str r0, [r1] @ almacenar handler
26     mov r4, #1 @acumulador de productoria
```



```
27  read_loop: @leer entero de archivo
28    ldr r0, =InFileHandle
29    ldr r0, [r0]
30    swi SWI_Read_Int
31    bcs EofReached
32    mov r2, r0 @ el entero está ahora en r0
33  @Compruebo si es positivo
34    mov r3, #0
35    subs r3, r3, r2 @Si la resta es negativa, r2>0
36    bmi productoria
37    b read_loop
38  productoria:
39    mul r4, r4, r2
40    b read_loop
41  InFileError:
42  EofReached:
43    ldr r0, =Stdout
44    mov r1, r4
45   swi SWI_Print_Int
46   swi SWI_Exit
47  .end
```

el programa comienza su ejecución en la etiqueta "_start". En primer lugar, se carga el nombre del archivo en el registro r0 y se establece el modo de apertura en 0 (modo de entrada) en el registro r1. A continuación, se llama a la interrupción SWI_Open_File para abrir el archivo. Si hay un error en la apertura del archivo, el programa salta a la etiqueta InFileError.

Si el archivo se abre correctamente, se almacena el identificador del archivo en la variable InFileHandle. Luego, se inicializa el acumulador de productoria en 1 en el registro r4.

El programa entra en un bucle de lectura llamado "read_loop". En cada iteración, se llama a la interrupción SWI_Read_Int para leer un entero desde el archivo abierto. Si se llega al final del archivo, el programa salta a la etiqueta EofReached. Si el entero leído es positivo, se multiplica por el acumulador de productoria en el registro r4. Si el entero es negativo o cero, el programa salta de nuevo al inicio del bucle de lectura.

Después de que se han leído todos los enteros del archivo, el programa imprime el resultado de la productoria en la salida estándar utilizando la interrupción SWI_Print_Int. Finalmente, el programa llama a la interrupción SWI_Exit para terminar su ejecución.

7) codificar un programa en ARM de 32 bits que recorra un vector de enteros y genere un archivo de salida con el resultado de aplicar la funcion AND en cada uno de los elementos del vector original contra una constante.

```
  .equ SWI_Print_Int, 0x6B
  .equ SWI_Exit, 0x11
  .equ SWI_Print_Str, 0x69
  .equ SWI_Open_File, 0x66
  .equ SWI_Write, 0x05
  .equ SWI_Close_File, 0x68

  .data
array_origen:
   .word 1,2,5,6
array_length:
   .word 4
filename:
   .asciz "resultado.txt"
eol:
   .asciz "\n"
   .align
outFileHandle:
   .skip 4

   .text
```

```
   .global _start

_start:

   ldr r0, =filename @nombre del archivo de salida

   mov r1, #1 @modo: salida

   mov r2, #384 @permisos de escritura

   swi SWI_Open_File @abre archivo

   bcs outFileError @chequea si hubo error

   ldr r1, =outFileHandle @carga dirección donde almacenar el handler

   str r0, [r1]



   ldr r1, =array_origen

   ldr r2, =array_length

   ldr r2, [r2]

loop:

   ldr r4, [r1]

   bl imprimir

   add r1, r1, #4

   subs r2, r2, #1

   cmp r2, #0

   bne loop

   b exit

imprimir:

   stmfd sp!, {r1,r2,lr}
```

```
   and r4, r4, #1 @Hago and contra una constante

   ldr r0, =outFileHandle

   ldr r0, [r0]

   mov r1, r4

   swi SWI_Print_Int @ Escribir el número en el archivo

   ldr r1, =eol @ Cargar el salto de línea

   swi SWI_Print_Str @ Escribir el salto de línea en el archivo

   ldmfd sp!, {r1,r2,pc}

outFileError: @sale del programa con error

exit:

   ldr r0, =filename

   swi SWI_Close_File @cierra el archivo de salida

   swi SWI_Exit

  .end
```

8)hacer un programa en assembler ARM 32 bits que recorra un vector de enteros y genere un nuevo vector adicionando un valor constante a cada uno de los elementos del vector original e imprimiendo por la salida estándar cada uno de los elementos del nuevo vector

```
  .equ SWI_Print_Int, 0x6B
```

```
  .equ SWI_Exit, 0x11

  .equ SWI_Print_Str, 0x69

  .equ Stdout, 1



  .data

array_origen:

   .word 1,2,5,6

array_length:

   .word 4

eol:

   .asciz "\n"



   .text

   .global _start

_start:

   ldr r0, =array_origen

   ldr r2, =array_length

   ldr r2, [r2]

loop:

   ldr r4, [r0] @cargo en r4 el numero del array para sumar con una
constante

   bl imprimir

   add r0, r0, #4
```

```
   subs r2, r2, #1

   cmp r2, #0

   bne loop

   b exit

imprimir:

   stmfd sp!, {r0,r1,lr}

   ldr r0, =Stdout

   add r1, r4, #10 @sumo la constante "10" al numero y lo guardo en r1

   swi SWI_Print_Int

   ldr r1, =eol

   swi SWI_Print_Str

   ldmfd sp!, {r0,r1,pc}

exit:

   swi SWI_Exit

  .end
```

Carga la dirección de memoria del primer elemento del array array_origen en el registro r0.
Carga la dirección de memoria de la variable array_length en el registro r2.
Carga el valor de la variable array_length en el registro r2.
Carga el valor del elemento actual del array en el registro r4.
Salta a la función imprimir.
Añade 4 a la dirección del elemento actual del array, para pasar al siguiente elemento.
Resta 1 al contador de elementos, que se encuentra en r2.
Compara el contador con 0, si no es cero, salta de nuevo al inicio del bucle en loop. Si es cero, sale del bucle y salta a la etiqueta exit.

Guarda los registros r0, r1 y lr en la pila para poder recuperarlos más tarde.
Carga la dirección de la salida estándar en el registro r0.
Añade la constante 10 al valor del elemento actual del array, que se encuentra en el registro r4, y almacena el resultado en r1.
Imprime el valor en r1 en la salida estándar usando la llamada al sistema SWI_Print_Int.
Carga la dirección de la cadena de caracteres eol (que contiene un salto de línea) en el registro r1.

Imprime la cadena de caracteres en r1 en la salida estándar usando la llamada al sistema SWI_Print_Str.

Recupera los registros r0, r1 y pc de la pila y salta de vuelta a la instrucción que sigue a la llamada a b

9) Negar enteros desde archivo

```
@ Escribir el código ARM que ejecutado bajo ARMSim# lea dos enteros
desde un archivo e
@ imprima:
@ El primer entero en su propia línea.
@ El resultado de aplicar NOT al primer entero en su propia línea.
@ El segundo entero en su propia línea.
@ El resultado de aplicar NOT al segundo entero en su propia línea.

   .equ SWI_Open_File, 0x66
   .equ SWI_Read_Int, 0x6C
   .equ SWI_Print_Int, 0x6B
   .equ SWI_Close_File, 0x68
   .equ SWI_Exit, 0x11
   .equ SWI_Print_Char, 0x00
       .equ SWI_Print_Str, 0x69

       .equ Stdout, 1

   .data
filename:
   .asciz "dos_enteros.txt"
eol:
       .asciz "\n"
       .align
InFileHandle:
       .word  0
```

```
   .text
   .global _start
_start:
       @abrir archivo
       ldr r0,=filename        @ nombre de archivo de entrada
       mov r1,#0               @ modo: entrada
       swi SWI_Open_File       @ abre archivo
       bcs InFileError         @ chequear si hubo error
       ldr r1,=InFileHandle    @ cargar dirección donde almacenar el
handler
       str r0,[r1]             @ almacenar handler

read_loop:
       @leer entero de archivo
       ldr r0,=InFileHandle
       ldr r0,[r0]
       swi SWI_Read_Int
       bcs EofReached  @bcs carry es 0, fin de prgm
       @ el entero está ahora en r0

       mov r2, r0

@imprimo el primer entero en su propia linea
       mov r1, r2
       bl print_r1_int @imprime el numero en r1 como un entero
       mov r3, #-1     @ (*) cargo -1 en un registro
       @r3 = 1111 1110
       @63 [10] = 0011 1111
       @r3  = 1111 1111
       @xor (resultado de aplicar not al primer entero)
       @ma da el opuesto xq si habia 1, contra 1 da 0. si habia 0,
contra 1 da 0.
       @r2  = 0011 1111
       @r1  = 1100 0001
       @(literal hace not (-63) y resta 1 -> -64)
       @5 -> -6
       EOR r1, r2, r3          @ (*) (r1)=NOT(r2)
       @EOR r1, r2, #-1    @ESTO NO SIRVE XQ -1 LO TOMA COMO UNA CTE
       @NO COMO UN NUMERO      @ (r1)=NOT(r2)

       bl print_r1_int
       b read_loop
```

```
print_r1_int:
       stmfd sp!, {r0,r1,lr}
       ldr r0, =Stdout
       swi SWI_Print_Int
       ldr r1, =eol
       swi SWI_Print_Str
       ldmfd sp!, {r0,r1,pc}

InFileError:
EofReached:
       @cierro archivo
       ldr r0,= InFileHandle @[ro] -> InFileHanlde
       ldr r0,[r0] @[ro] = InFileHanlde
       swi 0x68

   swi SWI_Exit
   .end
```

10)Escribir el código ARM que ejecutado bajo ARMSim# imprima el mensaje “Hola Mundo”

```
@ constantes
       .equ SWI_Print_String, 0x02
       .equ SWI_Exit, 0x11

       .data
       @todo lo q son variables

mensaje:
       @ .asciz : string terminado en byte nulo
       .asciz "Hola Mundo"
       @existe .ascii pero eso define la cadena de caracteres pero sin
el byte nulo

       @ sección de código - porción ejecutable del código
       .text

       @ hace _start disponible al linker
       @ esto es effectivamente parte de la definición del "main"
       .global _start
_start:
       @ carga el puntero al string en el registro r0
       @carga en R0 la direccion de mensaje
```

```
        ldr r0, =mensaje
        @imprime hasta que encuentra una valor nulo

        @ solicita a ARMSim que imprima el string
        swi SWI_Print_String

        @ solicita a ARMSim que salga del programa
        swi SWI_Exit
        .end
```

11)Escribir el código ARM que ejecutado bajo ARMSim# imprima dos cadenas de caracteres predefinidas en memoria incluyendo salto de línea “Hola” y ”Chau” utilizando una subrutina que imprima un string cuya dirección esté en el R3.

```
        .equ SWI_Print_String, 0x02
        .equ SWI_Exit, 0x11

        .data
first_string:
        .asciz "Hola\n"
second_string:
        .asciz "Chau\n"

        .text
        .global _start
_start:
        ldr r3, =first_string
        bl      print_r3 @bifurco a print_r3
        ldr r3, =second_string
        bl      print_r3 @ojo, b and link vuelve
        b       fin @b etiqueta
        @en cambio b solo bifruca, no vuelve

print_r3:
        @guardo en el stack el valor del lr que tendra la sig
instruccion
        @ y el de r0 (noi se xa q)
        stmfd sp!, {r0,lr}
        mov r0, r3 @pongo en r0 la direc del string a imprimir
        swi SWI_Print_String @lo imprimo
        ldmfd sp!, {r0,pc}
        @piso el valor del pc con el de lr q guarde en el stack
```

```
fin:
       swi SWI_Exit
       .end
```

12)Escribir el código ARM que ejecutado bajo ARMSim# lea los valores de un vector (vector) de longitud long_vector, sume un valor específico (valor) y guarde el resultado en otro vector(vector_suma).

```
.equ SWI_Print_Int, 0x6B
       .equ SWI_Exit, 0x11
       .equ SWI_Print_Str, 0x69
       .equ Stdout, 1

       .data
array_origen:
       .word 76, 14, 49, 27, -9, 108, 99
array_destino:
       .word 0, 0, 0, 0, 0, 0, 0
array_length:
       .word 7
constante:
       .word 9
eol:
   .asciz "\n"

       .text
       .global _start
_start:

   ldr r0, =array_origen
   ldr r1, =array_destino
   ldr r2, =array_length
   ldr r2, [r2]        @r2 = long del vector orig
   ldr r3, =constante
   ldr r3, [r3]        @r3 = valor de la cte

loop_suma:
   ldr r4, [r0]        @r4 pivote = contenido de la direc del primer
ele
   add r4, r4,r3       @r4=1er ele + cte
   str r4, [r1]        @dejo en la direc de memoria del nuevo vector
```

```
   add r0, r0, #4       @avanzo en vector orig
   add r1, r1, #4       @avanzo en vector destomp
   sub r2, r2, #1       @resto 1 a la long del vector orig
   cmp r2, #0
   bne loop_suma        @si no es igual, sig elemento

   ldr r2, =array_destino  @direc de destino
   ldr r3, =array_length
   ldr r3, [r3]     @r3 = tam array orig, guardado en var

loop_mostrar:
       cmp r3, #0
       beq exit

       ldr r0, =Stdout
       ldr r1, [r2]     @num en esa direc
       swi SWI_Print_Int
       ldr r1, =eol
   swi SWI_Print_Str

       add r2, r2, #4
       sub r3, r3, #1
       b loop_mostrar

exit:
       swi SWI_Exit
       .end
```

13)Escribir el código ARM que ejecutado bajo ARMSim# encuentre e imprima el menor elemento de un vector, donde el vector está especificado con el label vector y la longitud del vector con el label long_vector.

```
   .equ SWI_Print_Int, 0x6B
   .equ SWI_Exit, 0x11
   .equ SWI_Print_Str, 0x69
   .equ Stdout, 1

   .data
array:
   .word 76, 14, 49, 27, -9, 108, 99, -25
```

```
array_length:
   .word 8
eol:
   .asciz "\n"

   .text
   .global _start
_start:

   ldr r0, =array @r0 = direc del array
   ldr r1, [r0] @r1 = Primer ele
   ldr r2, =array_length
   ldr r2, [r2] @r2 - cant de eles

buscar_min:

   @con post incremento,
   @ldr r3, [r0], #4 @guardo el ele actual y me muevo al sig
   @pero ojo, dsps tendria q hacer ver(*)
   ldr r3, [r0]    @r3 = primer ele del array (ele actual en la iter)
   cmp r1, r3  @r1<r3? minAnt < r3(NumActual)
   ble es_menor @el ant es menor? si lo es, sig num
   @mov r1, r3 @ (*) xq el [r0] ya no apunta al prox (2 instr xa abajo)
   ldr r1, [r0]    @sino, pispo r1= nuevoMin
es_menor:
   add r0, r0, #4 @me muevo al sig ele
   sub r2, r2, #1 @resto 1 al cont del tam del array
   cmp r2, #0 @pregunto si el contador llego a 0
   bne buscar_min  @si no llego a 0, hay nums en el vector

print_r1_int:
   stmfd sp!, {r0,r1,lr}
   ldr r0, =Stdout
   swi SWI_Print_Int
   ldr r1, =eol
   swi SWI_Print_Str
   ldmfd sp!, {r0,r1,pc}

exit:
   swi SWI_Exit
   .end
```

14)Modificar el ejercicio para utilizar direccionamiento por registro indirecto con registro indexado escalado. ejercicio 15.

```
   .equ SWI_Print_Int, 0x6B
   .equ SWI_Exit, 0x11
   .equ SWI_Print_Str, 0x69

   .equ Stdout, 1
   .data
array:
   .word 4, 8, 12, 9
eol:
   .asciz "\n"

   .text
   .global _start
start:
   @ r2: base
   @ r4: offset
   ldr     r2, =array
   mov     r3, #4              @ (r3): longitud array
   mov     r4, #0

mostrar_loop:
   bl      mostrar_elemento
   subs    r3, r3, #1
   bne     mostrar_loop
   b       fin

   @ mostrar entero apuntado por r2
mostrar_elemento:
   stmfd   sp!, {r0,r1,lr}     @ Stack: r0 y r1
   ldr     r0, =Stdout
   ldr     r1, [r2, r4, LSL #2]
   add     r4, r4, #1
   swi     SWI_Print_Int
   ldr     r1, =eol
   swi     SWI_Print_Str
   ldmfd   sp!, {r0,r1,pc}     @ Unstack: r0 y r1

fin:
   swi SWI_Exit
   .end
```

15)Escribir el código ARM que ejecutado bajo ARMSim# imprima los valores de un vector de cuatro enteros definidos en memoria, recorriendo el vector mediante una subrutina que utilice direccionamiento por registro indirecto.

```
   .equ SWI_Print_Int, 0x6B
   .equ SWI_Exit, 0x11
   .equ SWI_Print_Str, 0x69

   .equ Stdout, 1
   .data
array:
   .word 4, 8, 12, 9
eol:
   .asciz "\n"

   .text
   .global _start
start:
   @ r2: base
   @ r4: offset
   ldr     r2, =array
   mov     r3, #4              @ (r3): longitud array

mostrar_loop:
   bl      mostrar_elemento
   add     r2, r2, #4  @r2 +=4
   subs    r3, r3, #1  @r3 -=1 (vueltas  )
   bne     mostrar_loop
   @si no es igual a 0, es decir me quedan vueltas para terminar de
recorrer el array,
   @vuelvo a mostrar loop
   b       fin

   @ mostrar entero apuntado por r2
mostrar_elemento:
   stmfd   sp!, {r0,r1,lr}     @ Stack: r0 y r1
   ldr     r0, =Stdout
   ldr     r1, [r2]
   swi     SWI_Print_Int
   ldr     r1, =eol
   swi     SWI_Print_Str
   ldmfd   sp!, {r0,r1,pc}     @ Unstack: r0 y r1
```

```
fin:
   swi SWI_Exit
   .end
```

16)Escribir el código ARM que ejecutado bajo ARMSim# imprima dos valores enteros definidos en memoria, los reemplace por otros dos valores e imprima los dos nuevos valores.

```
   .equ SWI_Print_Int, 0x6B
   .equ SWI_Exit, 0x11
   .equ SWI_Print_Char, 0x00

   .data
entero1:
   .word 5
entero2:
   .word 28

   .text
   .global _start
_start:
   @ imprimimos el contenido de ambas variables

   @ muestra entero1
   mov r0, #1          @ salida por stdout
   ldr r2, =entero1    @ (r2) = DIR(entero1)
   ldr r1, [r2]        @ (r1) = entero1
   swi SWI_Print_Int

   mov r0, #'\n
   swi SWI_Print_Char

   @ entero2
   mov r0, #1          @ salida por stdout
   ldr r2, =entero2    @ (r2) = DIR(entero2)
   ldr r1, [r2]        @ (r1) = entero2
   swi SWI_Print_Int

   mov r0, #'\n
   swi SWI_Print_Char
```

```
@ cargar 43 en entero1
ldr r0, =entero1    @ (r0) = DIR(entero1)
mov r1, #43
str r1, [r0]        @ almacena 43 in la memoria apuntada por r0

@ cargar 79 en entero2
ldr r0, =entero2    @ (r0) = DIR(entero2)
mov r1, #79
str r1, [r0]        @ almacena 79 in la memoria apuntada por r0

@ muestra entero1
mov r0, #1
ldr r1, =entero1
ldr r1, [r1]
swi SWI_Print_Int

mov r0, #'\n
swi SWI_Print_Char

@ muestra entero2
mov r0, #1
ldr r1, =entero2
ldr r1, [r1]
swi SWI_Print_Int

swi SWI_Exit
.end
```

17)Escribir el código ARM que ejecutado bajo ARMSim# lea un entero desde un archivo, calcule el valor de la posición que corresponde a ese entero en la sucesión de Fibonacci.

```
.equ SWI_Open_File, 0x66
.equ SWI_Read_Int, 0x6C
.equ SWI_Print_Int, 0x6B
.equ SWI_Close_File, 0x68
.equ SWI_Exit, 0x11
.equ SWI_Print_Char, 0x00

.equ SWI_Print_Str, 0x69
```

```
   .equ Stdout, 1

   .data
filename:
   .asciz "entero.txt"
eol:
   .asciz "\n"
   .align
InFileHandle:
   .word  0

   .text
   .global _start
_start:
   @open file
   ldr r0,=filename        @ set Name for input file
   mov r1,#0               @ mode is input
   swi SWI_Open_File       @ open file for input
   bcs InFileError         @ if error?
   ldr r1,=InFileHandle    @ load input file handle
   str r0,[r1]             @ save the file handle

   bl read_int             @loads 1 value on r0
   bl fibonacci
   bl print_r1_int
   b exit

   @ (r1) = fibonacci(r0)
fibonacci:
   stmfd sp!, {r0,r2,lr}
   cmp r0, #1
   ble fibonacci1
   sub r0, r0, #1
   bl fibonacci
   mov r2, r1
   sub r0, r0, #1
   bl fibonacci
   add r1, r1, r2
   b fibonacci_end
fibonacci1:
   mov r1, r0
fibonacci_end:
```

```
   ldmfd sp!, {r0,r2,pc}

   @read integer from file
read_int:
   stmfd sp!, {lr}
   ldr r0,=InFileHandle
   ldr r0,[r0]
   swi SWI_Read_Int
   ldmfd sp!, {pc}

print_r1_int:
   stmfd sp!, {r0,r1,lr}      @ Stack r1
   ldr r0, =Stdout
   swi SWI_Print_Int          @ Print integer in register r1 to stdout
   ldr r1, =eol
   swi SWI_Print_Str          @ Print EOL in register r1 to stdout
   ldmfd sp!, {r0,r1,pc}      @ Unstack r1

InFileError:
exit:
   swi SWI_Exit
   .end
```

18)Escribir el código ARM que ejecutado bajo ARMSim# lea un entero desde un archivo, calcule el factorial de ese entero haciendo llamadas recursivas a la misma subrutina y muestre los valores intermedios del proceso.

```

   .equ SWI_Open_File, 0x66
   .equ SWI_Read_Int, 0x6C
   .equ SWI_Print_Int, 0x6B
   .equ SWI_Close_File, 0x68
   .equ SWI_Exit, 0x11
   .equ SWI_Print_Char, 0x00

   .equ SWI_Print_Str, 0x69

   .equ Stdout, 1

   .data
filename:
   .asciz "entero.txt"
```

```
eol:
   .asciz "\n"
   .align
InFileHandle:
   .word  0

   .text
   .global _start
_start:
   @open file
   ldr r0,=filename        @ set Name for input file
   mov r1,#0               @ mode is input
   swi SWI_Open_File       @ open file for input
   bcs InFileError         @ if error?
   ldr r1,=InFileHandle    @ load input file handle
   str r0,[r1]             @ save the file handle

   bl read_int             @loads 1 value on r0

   bl factorial

   bl print_r1_int

   b exit

   @ (r1) = (r0)!
   @todas las direcciones de memoria que voy pusheando sirven
   @xa volver a factorial salvo la primera
factorial:
   @Voy guardando en la pila 7,6,5,4,3,2,1
   stmfd sp!, {r0,lr}
   cmp r0, #1
   @caso base, corto el loop y salto al factorial de 1
   beq factorial1
   @r0 -=1
   sub r0, r0, #1
   @abajo, calculo el factorial de ese numero (6)
   @5
   bl factorial
   @dejo el link en el r14...
   @r0 +=1
   @vuelve a ser 7
   add r0, r0, #1
```

```
   @ 7*6! = 7 * factorial de 6, q todavia no calcule
   @ r1 = r0 * r1
   @ r1 =
   mul r1, r0, r1
   b factorial_end

factorial1:
   mov r1, #1
factorial_end:
   bl print_r1_int
   ldmfd sp!, {r0,pc}

   @read integer from file
read_int:
   stmfd sp!, {lr}
   ldr r0,=InFileHandle
   ldr r0,[r0]
   swi SWI_Read_Int
   ldmfd sp!, {pc}

print_r1_int:
   stmfd sp!, {r0,r1,lr}      @ Stack r1
   ldr r0, =Stdout
   swi SWI_Print_Int       @ Print integer in register r1 to stdout
   ldr r1, =eol
   swi SWI_Print_Str       @ Print EOL in register r1 to stdout
   ldmfd sp!, {r0,r1,pc}      @ Unstack r1

InFileError:
exit:
   swi SWI_Exit
   .end
```

20)Escribir el código ARM que ejecutado bajo ARMSim# lea tres enteros desde un archivo e imprima la mediana, siendo la mediana el valor de la variable de posición central en un conjunto de datos ordenados. Por ejemplo, si los valores fueran 5, 8 y 9, la mediana sería 8.

```
.equ SWI_Open_File, 0x66
   .equ SWI_Read_Int, 0x6C
   .equ SWI_Print_Int, 0x6B
   .equ SWI_Close_File, 0x68
   .equ SWI_Exit, 0x11
```

```
   .equ SWI_Print_Str, 0x69

   .equ Stdout, 1

   .data
filename:
   .asciz "tres_enteros.txt"
eol:
   .asciz "\n"
   .align
InFileHandle:
   .word  0

   .text
   .global _start
_start:

   @open file
   ldr r0,=filename        @ set Name for input file
   mov r1,#0               @ mode is input
   swi SWI_Open_File       @ open file for input
   bcs InFileError         @ if error?
   ldr r1,=InFileHandle    @ load input file handle
   str r0,[r1]             @ save the file handle

   @loads 3 values on r2, r3 and r4
   bl read_int
   mov r2, r0
   bl read_int
   mov r3, r0
   bl read_int
   mov r4, r0

   mov r5, r2
   mov r6, r3
   cmp r5, r6
   blgt swapr5r6
   mov r2, r5
   mov r3, r6

   mov r5, r3
   mov r6, r4
```

```
   cmp r5, r6
   blgt swapr5r6
   mov r3, r5
   mov r4, r6

   mov r5, r2
   mov r6, r3
   cmp r5, r6
   blgt swapr5r6
   mov r2, r5
   mov r3, r6

   mov r1, r3
   bl print_r1_int

   b exit

@read integer from file
read_int:
   stmfd sp!, {lr}
   ldr r0,=InFileHandle
   ldr r0,[r0]
   swi SWI_Read_Int
   ldmfd sp!, {pc}

swapr5r6:
   stmfd sp!, {r1,lr}
   mov r1, r5
   mov r5, r6
   mov r6, r1
   ldmfd sp!, {r1,pc}

print_r1_int:
   stmfd sp!, {r0,r1,lr}      @ Stack r1
   ldr r0, =Stdout
   swi SWI_Print_Int          @ Print integer in register r1 to stdout
   ldr r1, =eol
   swi SWI_Print_Str          @ Print EOL in register r1 to stdout
   ldmfd sp!, {r0,r1,pc}      @ Unstack r1

InFileError:
EofReached:
exit:
```

```
    swi SWI_Exit
    .end
```

21)Escribir el código ARM que ejecutado bajo ARMSim# lea un entero desde un archivo e imprima el valor absoluto del entero. Utilizar instrucciones ejecutadas condicionalmente y no utilizar bifurcaciones condicionales.

```
    .equ SWI_Open_File, 0x66
    .equ SWI_Read_Int, 0x6C
    .equ SWI_Print_Int, 0x6B
    .equ SWI_Close_File, 0x68
    .equ SWI_Exit, 0x11

    .data
filename:
    .asciz "entero.txt"

    .text
    .global _start
_start:
    @ abrir archivo y cargar manejador en r5
    ldr r0, =filename
    mov r1, #0
    swi SWI_Open_File
    mov r5, r0

    @ leer entero y almacenarlo en r2
    swi SWI_Read_Int
    mov r2, r0

    @ cerrar archivo
    mov r0, r5
    swi SWI_Close_File

    @ chequear si el entero leído es menor a 0
    cmp r2, #0  @r2 - 0
    @r2 = int

    @ si es: sobrescribir el entero con su negación aritmética
    @alternativa para no restar contra cero
    @rsbmi r2,#0
```

```
   mov r3, #0  @r3 =0
   submi r2, r3, r2 @deja el valor absoluto en r2
   @r3 - r2 = 0 - r2 (solo se ejecuta si la cmp da negativo)

   @ mostrar el entero
   mov r0, #1
   mov r1, r2
   swi SWI_Print_Int

   @ salir del programa
   swi SWI_Exit
   .end
```

22)Escribir el código ARM que ejecutado bajo ARMSim# lea dos enteros desde un archivo e imprima:
El primer entero en su propia línea.
El resultado de aplicar NOT al primer entero en su propia línea.
El segundo entero en su propia línea.
El resultado de aplicar NOT al segundo entero en su propia línea.

```
   .equ SWI_Open_File, 0x66
   .equ SWI_Read_Int, 0x6C
   .equ SWI_Print_Int, 0x6B
   .equ SWI_Close_File, 0x68
   .equ SWI_Exit, 0x11
   .equ SWI_Print_Char, 0x00
   .equ SWI_Print_Str, 0x69

   .equ Stdout, 1

   .data
filename:
   .asciz "dos_enteros.txt"
eol:
       .asciz "\n"
       .align
InFileHandle:
       .word  0

   .text
   .global _start
```

```
_start:
	@abrir archivo
	ldr r0,=filename	@ nombre de archivo de entrada
	mov r1,#0		@ modo: entrada
	swi SWI_Open_File	@ abre archivo
	bcs InFileError		@ chequear si hubo error
	ldr r1,=InFileHandle	@ cargar dirección donde almacenar el
handler
	str r0,[r1]		@ almacenar handler

read_loop:
	@leer entero de archivo
	ldr r0,=InFileHandle
	ldr r0,[r0]
	swi SWI_Read_Int
	bcs EofReached  @bcs carry es 0, fin de prgm
	@ el entero está ahora en r0

	mov r2, r0

@imprimo el primer entero en su propia linea
	mov r1, r2
	bl print_r1_int @imprime el numero en r1 como un entero
	mov r3, #-1	@ (*) cargo -1 en un registro
	@r3 = 1111 1110
	@63 [10] = 0011 1111
	@r3  = 1111 1111
	@xor (resultado de aplicar not al primer entero)
	@ma da el opuesto xq si habia 1, contra 1 da 0. si habia 0,
contra 1 da 0.
	@r2  = 0011 1111
	@r1  = 1100 0001
	@(literal hace not (-63) y resta 1 -> -64)
	@5 -> -6
	EOR r1, r2, r3		@ (*) (r1)=NOT(r2)
	@EOR r1, r2, #-1    @ESTO NO SIRVE XQ -1 LO TOMA COMO UNA CTE
	@NO COMO UN NUMERO	@ (r1)=NOT(r2)

	bl print_r1_int
	b read_loop

print_r1_int:
	stmfd sp!, {r0,r1,lr}
```

```
        ldr r0, =Stdout
        swi SWI_Print_Int
        ldr r1, =eol
        swi SWI_Print_Str
        ldmfd sp!, {r0,r1,pc}

InFileError:
EofReached:
        @cierro archivo
        ldr r0,= InFileHandle @[ro] -> InFileHanlde
        ldr r0,[r0] @[ro] = InFileHanlde
        swi 0x68

    swi SWI_Exit
    .end
```

PREGUNTAS TEORICAS DE ARM32

1)Explique claramente que es y cómo funciona el barrel shifter en la arquitectura ARM de 32 bit.De ejemplo concretos con instrucciones assembler.

El barrel shifter es un componente importante de la arquitectura ARM de 32 bits que permite realizar operaciones de desplazamiento y rotación de bits en un registro. El barrel shifter se utiliza comúnmente para realizar operaciones aritméticas y lógicas de forma más eficiente.

En la arquitectura ARM, el barrel shifter puede realizar desplazamientos hacia la izquierda o hacia la derecha de un registro de 32 bits en un rango de 0 a 31 bits. Además, el barrel shifter puede realizar rotaciones de un registro de 32 bits en un rango de 1 a 31 bits.

El barrel shifter puede ser utilizado en combinación con otras instrucciones para realizar operaciones más complejas. Por ejemplo, la instrucción ADD (Sumar) se puede utilizar con el barrel shifter para sumar un valor inmediato a un registro. La instrucción LSL (Left Shift) se utiliza para realizar un desplazamiento hacia la izquierda en un registro, mientras que la instrucción ROR (Rotate Right) se utiliza para realizar una rotación hacia la derecha en un registro.

Ejemplos

mov r2, r0, lsl #2 @ R2 = R0x4
add r9, r5, r5, lsl #3 @ R9 = R5+R5x8 ó R9=R5x9
rsb r9, r5, r5, lsl #3 @ R9 = R5x8-R5 ó R9=R5x7
sub r10, r9, r8, lsr #4 @ R10 = R9-R8/16
mov r12, r4, ror r3 @ R12 = R4 rotado der. por el valor en R3

2)Explique claramente que son los flags de codigo de condición en la arquitectura ARM 32 bits. ¿cuales identifica? para que se usan? ejemplifique con una porción de código assembler.

Los flags de código de condición son un conjunto de banderas que se utilizan en la arquitectura ARM de 32 bits para indicar el resultado de una operación y para controlar la ejecución condicional de las instrucciones.

Los flags de código de condición se almacenan en el registro CPSR (Current Program Status Register), que es un registro de 32 bits que contiene información sobre el estado actual de la CPU. El registro CPSR se actualiza automáticamente después de cada instrucción ejecutada.

Los flags de código de condición en la arquitectura ARM de 32 bits identifican las siguientes condiciones:

- Z: Bandera de estado cero (Zero). Se establece en 1 si el resultado de una operación es cero.
- C: Bandera de acarreo (Carry). Se establece en 1 si una operación de suma o resta genera un acarreo o préstamo.
- N: Bandera de estado negativo (Negative). Se establece en 1 si el resultado de una operación es negativo.
- V: Bandera de desbordamiento (Overflow). Se establece en 1 si una operación de suma o resta genera un desbordamiento.

Estos flags de código de condición se utilizan para controlar la ejecución condicional de las instrucciones en la arquitectura ARM. Las instrucciones condicionales solo se ejecutan si se cumple una determinada condición, que se especifica mediante el uso de los flags de código de condición.

A continuación, se muestra un ejemplo de código en assembler que utiliza los flags de código de condición en la arquitectura ARM para controlar la ejecución condicional de una instrucción:

```
MOV r0, #5      ; Mueve el valor 5 al registro r0
MOV r1, #10     ; Mueve el valor 10 al registro r1
CMP r0, r1      ; Compara el valor en r0 con el valor en r1
BGE greater_than_or_equal ; Salta a la etiqueta "greater_than_or_equal"
si el valor en r0 es mayor o igual que el valor en r1
MOV r2, #0      ; Mueve el valor 0 al registro r2
```

```
B end            ; Salta a la etiqueta "end"

greater_than_or_equal:
MOV r2, #1       ; Mueve el valor 1 al registro r2

end:
; Continúa con el resto del código
```

En este ejemplo, se utiliza la instrucción CMP para comparar el valor en el registro r0 con el valor en el registro r1. Si el valor en r0 es mayor o igual que el valor en r1, se salta a la etiqueta "greater_than_or_equal" y se ejecuta la instrucción MOV r2, #1. De lo contrario, se ejecuta la instrucción MOV r2, #0 y se salta a la etiqueta "end". En este ejemplo, se utiliza la bandera de estado N (Negative) para controlar la ejecución condicional de la instrucción.

3)En la arquitectura ARM de 32 bits, ¿a qué se denomina ejecución condicional de una instrucción? De un ejemplo de su uso en assembler.

La ejecución condicional de una instrucción en la arquitectura ARM de 32 bits se refiere a la capacidad de ejecutar una instrucción solo si se cumple una determinada condición. Esto se logra mediante el uso de las instrucciones condicionales, que se identifican mediante un código de dos letras que se agrega al final de la instrucción.

Los códigos de condición se basan en los flags de código de condición que se establecen después de la ejecución de una instrucción.

Tambien elimina la necesidad de utilizar muchas bifurcaciones. El costo en tiempo de no ejecutar una instrucción condicional es frecuentemente menor que el uso de una bifurcación o llamado a una subrutina que, de otra manera, sería necesaria.

Códigos de Condición

Code [31:28]	–Mnemonic	– Interpretación –	Status flag state required
0000	EQ	Igual / Igual a cero	Z seteado
0001	NE	Distinto	Z vacío
0010	CS/HS	Carry seteado / ≥ sin signo	C seteado
0011	CC/LO	Carry vacío / < sin signo	C vacío
0100	MI	Menor / negativo	N seteado
0101	PL	Mayor / positivo o cero	N vacío

Por ejemplo, una instrucción de suma tiene la siguiente forma:

ADD r0, r1, r2

y para ejecutarla sólo si el flag cero está seteado:

ADDEQ r0, r1, r2

Esto mejora la densidad del código y la performance reduciendo el número de instrucciones de

bifurcación. Ej.:

CMP r3, #0

BEQ skip

ADD r0, r1, r2

4)Explique claramente y ejemplifique al menos cuatro modos de direccionamiento presentes en la arquitectura ARM 32 bits.

Registro a registro: El modo de direccionamiento de registro a registro en la arquitectura ARM se refiere a un tipo de instrucción que mueve o manipula datos entre dos registros. En este modo de direccionamiento, los datos no están almacenados en la memoria, sino que se encuentran en los registros.

Por ejemplo, la instrucción ADD r1, r2, r3 suma los valores de los registros R2 y R3 y almacena el resultado en el registro R1. En este caso, la instrucción está realizando una operación de registro a registro, ya que los operandos están ubicados en los registros R2 y R3, y el resultado se guarda en el registro R1.

Este modo de direccionamiento es útil para realizar operaciones aritméticas o lógicas, como sumas, restas, multiplicaciones, entre otras, entre los datos almacenados en los registros. También se utiliza para transferir datos de un registro a otro, ya sea para hacer copias de datos o para realizar operaciones entre ellos.

Es importante tener en cuenta que, en la arquitectura ARM, los registros son muy importantes ya que son de uso general, es decir, pueden ser utilizados para múltiples operaciones. Por lo tanto, el modo de direccionamiento de registro a registro es una de las formas más comunes de manipular datos en la arquitectura ARM, ya que es muy rápido y eficiente.

Pre-indexado,autoindexado:son dos modos de direccionamiento por registro que se utilizan en la arquitectura ARM para acceder a datos almacenados en la memoria.

En el modo de direccionamiento por registro pre-indexado, se utiliza un registro base y un valor de desplazamiento que se suman para obtener la dirección de memoria deseada antes de acceder a los datos. Es decir, el registro base se modifica antes de la operación de acceso a memoria. Por ejemplo, la instrucción LDR r1, [r2, #4]! carga el valor almacenado en la dirección de memoria r2+4 en el registro r1 y aumenta el valor de r2 en 4 antes de la operación de acceso a memoria. El signo de admiración (!) indica que se está utilizando el modo de direccionamiento pre-indexado.

En el modo de direccionamiento por registro autoindexado, el registro base se modifica después de la operación de acceso a memoria. En este caso, se utiliza el signo de admiración después del registro base para indicar que se está utilizando el modo autoindexado. Por ejemplo, la instrucción LDR r1, [r2], #4 carga el valor almacenado en la dirección de memoria r2 en el registro r1 y aumenta el valor de r2 en 4 después de la operación de acceso a memoria.

Ambos modos de direccionamiento por registro son útiles para acceder a datos en la memoria de manera eficiente y rápida, ya que permiten modificar los registros base para acceder a diferentes ubicaciones de memoria sin tener que realizar operaciones adicionales. Sin embargo, es importante tener en cuenta que el uso incorrecto de estos modos de direccionamiento puede llevar a errores de programación, como la sobrescritura de datos en la memoria o la pérdida de información almacenada en los registros.

Doble registro indirecto escalado:El modo de direccionamiento por doble registro indirecto y el escalado de registro son dos modos adicionales de direccionamiento en la arquitectura ARM.

En el modo de direccionamiento por doble registro indirecto, se utiliza un par de registros para acceder a los datos almacenados en la memoria. El primer registro se utiliza para almacenar la dirección base de la memoria, mientras que el segundo registro se utiliza para almacenar el desplazamiento desde la dirección base. La instrucción de acceso a memoria utiliza los valores de ambos registros para calcular la dirección de memoria deseada. Por ejemplo, la instrucción LDR r1, [r2, r3] carga el valor almacenado en la dirección de memoria r2 + r3 en el registro r1.

El escalado de registro es otro modo de direccionamiento que permite ajustar el desplazamiento utilizado para acceder a los datos almacenados en la memoria. En este modo, se utiliza un registro base y un registro de índice, y se multiplica el valor del registro de índice por una constante para ajustar el desplazamiento. Por ejemplo, la instrucción LDR r1, [r2, r3, lsl #2] carga el valor almacenado en la dirección de memoria r2 + (r3 * 4) en el registro r1, ya que se multiplica el valor del registro r3 por 4 (lsl #2) antes de sumarlo al registro base r2.

Estos modos de direccionamiento por doble registro indirecto y escalado de registro son útiles para acceder a datos almacenados en matrices o estructuras de datos en la memoria, ya que permiten ajustar la dirección de memoria de manera flexible y eficiente. Sin embargo, es importante tener en cuenta que estos modos de

direccionamiento pueden ser más complicados de usar y pueden requerir más cálculos para obtener la dirección de memoria correcta.

Modo post-indexado:La dirección efectiva del operando es el contenido de Rn. El desplazamiento se agrega a esta dirección y el resultado se escribe de nuevo en Rn.

`[Rn], #offset`	`DE=[Rn]` `Rn⬅[Rn]+offset`
`[Rn], ±Rm, shift`	`DE=[Rn]` `Rn⬅[Rn]±[Rm] shifteado`

5)Explique claramente que es y cómo funciona el modo de direccionamiento post-indexado autoindexado (registro indirecto con post-incremento) en la arquitectura ARM de 32 bits. De un ejemplo concreto con una instrucción.

El modo de direccionamiento post-indexado (también conocido como autoindexado o registro indirecto con post-incremento) es una técnica de direccionamiento utilizada en la arquitectura ARM de 32 bits para acceder a datos almacenados en la memoria. En este modo de direccionamiento, se utiliza un registro base como punto de partida para acceder a un dato específico en la memoria, y después de acceder al dato, el valor del registro base se incrementa automáticamente.

El proceso para acceder a un dato específico utilizando el modo de direccionamiento post-indexado en ARM de 32 bits es el siguiente:

1. Cargar en un registro un valor que sirva como dirección base, es decir, la dirección de inicio desde la que se empezará a buscar el dato específico.
2. Acceder al dato en la dirección base contenida en el registro.
3. Incrementar el valor del registro base automáticamente después de acceder al dato.
4. Utilizar el dato accedido para realizar una operación.

A continuación, se presenta un ejemplo concreto utilizando la instrucción "LDR" (Load Register), que se utiliza para cargar un dato desde la memoria en un registro en ARM de 32 bits:

Supongamos que se tiene la siguiente secuencia de instrucciones en un programa ARM de 32 bits:

```
LDR r0, [r1], #4
```

En esta instrucción, se utiliza el modo de direccionamiento post-indexado para cargar en el registro r0 el dato almacenado en la dirección base contenida en el registro r1, y después de acceder al dato, se incrementa automáticamente el valor de r1 en 4 bytes. En otras palabras, se está accediendo al dato ubicado en la dirección de memoria r1, y luego incrementando r1 en 4 para apuntar al siguiente dato en la memoria.

En resumen, el modo de direccionamiento post-indexado es una técnica útil para acceder a datos en la memoria utilizando un registro base y un incremento automático, lo que permite acceder a datos específicos de manera más eficiente y flexible en la arquitectura ARM de 32 bits.

6)Explique claramente que es y como funciona el modo de direccionamiento de registro indirecto con offset en arquitectura ARM de 32 bits.De un ejemplo concreto con una instrucción.

El modo de direccionamiento de registro indirecto con offset es una técnica de direccionamiento utilizada en la arquitectura ARM de 32 bits para acceder a datos almacenados en la memoria. En este modo de direccionamiento, se utiliza un registro base como punto de partida para acceder a un dato específico en la memoria, mediante la adición de un desplazamiento (offset) a la dirección base.

El proceso para acceder a un dato específico utilizando el modo de direccionamiento de registro indirecto con offset en ARM de 32 bits es el siguiente:

1. Cargar en un registro un valor que sirva como dirección base, es decir, la dirección de inicio desde la que se empezará a buscar el dato específico.
2. Seleccionar un registro que contenga el valor del desplazamiento (offset) a aplicar a la dirección base.
3. Realizar una operación de suma entre la dirección base y el desplazamiento para obtener la dirección final del dato en la memoria.
4. Acceder al dato en la dirección final obtenida.

A continuación, se presenta un ejemplo concreto utilizando la instrucción "LDR" (Load Register), que se utiliza para cargar un dato desde la memoria en un registro en ARM de 32 bits:

Supongamos que se tiene la siguiente secuencia de instrucciones en un programa ARM de 32 bits:

```
LDR r0, [r1, #8]
```

En esta instrucción, se utiliza el modo de direccionamiento de registro indirecto con offset para cargar en el registro r0 el dato almacenado en la dirección base contenida en el registro r1, sumándole un desplazamiento de 8 bytes. En otras palabras, se está accediendo al dato ubicado en la dirección de memoria r1 + 8.

En resumen, el modo de direccionamiento de registro indirecto con offset es una técnica útil para acceder a datos en la memoria utilizando un registro base y un desplazamiento. Esto permite acceder a datos específicos de manera más eficiente y flexible en la arquitectura ARM de 32 bits.