ABI (Application Binary Interface)

ABI es una convención que define como dos módulos de programa pueden interactuar entre sí, pudiendo ser uno de los módulos una biblioteca de software y el otro el código que lo llama, o bien códigos escritos en diferentes lenguajes de programación, tales como C y Assembler.

En ARM la ABI estándar se llama AAPCS, que es la ARM Architecture Procedure Call Standard. Existen algunos estándares más antiguos, tales como ATPCS, APCS y TPCS que se consideran obsoletos.

Se puede seleccionar el ABI que se necesita utilizar mediante la opción **`-mabi=`** del compilador gcc. Para el ensamblador no hay opción equivalente porque el programador deberá implementar la ABI como se indica a continuación.

En algunos procesadores ARM a partir de la versión 8 se pueden ejecutar programas escritos en 64 bits que usan la arquitectura AArch64. En este caso se usa la ABI denominada AAPCS64. Para diferenciarla de ésta, la ABI que vamos a describir se llama AAPCS32.

Este documento es una introducción a AAPCS32. Se puede obtener información detallada en https://github.com/ARM-software/abi-aa/releases aca esta la info completa de la ABI

Tipos de datos fundamentales

Estos se dividen en:

- Números enteros de 1 (byte / `char`), 2 (half word / `short`), 4 (word / `int`) u 8 (double word / `long long int`) bytes que pueden ser signados o no.

- Números de punto flotante de 4 (precisión simple / `float`) u 8 (precisión doble / `double`) bytes.

- Vectores de 8 y 16 bytes (usados en SIMD).
  los vectores son paquetes de datos(no es lo mismo que usamos siempre)

- Punteros de 4 bytes (a código o datos, sin importar el tipo).

La variable debe comenzar en una dirección múltiplo del tamaño del dato, excepto en el caso de vectores de 16 bytes que pueden comenzar en una dirección múltiplo de 8. Esto puede hacer que algunos bytes no se puedan utilizar para almacenar datos.
un int empieza con una direccion multiplo de 4, un short con una direc multiplo de 2, ect.

Tipos de datos compuestos

Los siguientes datos se forman a partir de los datos fundamentales o de otros tipos compuestos:

structs

- Estructuras, donde los miembros se encuentran secuencialmente en memoria, respetando su alineación.

- Uniones, donde cada uno de los miembros comienza en la misma dirección de memoria.

- Arreglos (arrays), que son secuencias repetidas de algún tipo fundamental o compuesto.

Esto quiere decir que es posible por ejemplo tener un array de estructuras que tengan varios elementos de diferentes tipos.

La alineación de cada elemento y de la variable de tipo compuesto se obtiene de manera recursiva. Por ejemplo:

Código	**Alineación**
`struct stMiEstructura1 {`	2 bytes
`   char elem1;`	1 byte
`   unsigned short elem2[10];`	2 bytes
`};`	
`struct stMiEstructura2 {`	4 bytes
`   struct stMiEstructura1 elem3;`	2 bytes
`   char elem4;`	1 byte
`   int elem5;`	4 bytes
`};`	

primero agarro la struct1, me fijo cual es el elemento con la alineacion mas grande y ese va a ser la alineacion de la estructura

Uso de registros del procesador

Los registros R0, R1, R2 y R3 se utilizan para pasar argumentos a una subrutina y para retornar el valor del resultado de la función. Dentro de una función se pueden utilizar para almacenar datos temporarios.

Los registros R4, R5, R6, R7, R8, R10 y R11 se utilizan para variables locales de la función. R11 también se usa como frame pointer. Estos registros deben ser preservados por la subrutina.

El registro R12 se utiliza para poder hacer llamadas a subrutinas cuyo destino esté muy alejado (en ese caso la instrucción BL no se puede utilizar) o bien para cambiar de modo, de ARM a Thumb y viceversa. Para una llamada a función, el compilador genera una instrucción BL. Si el linker determina que el destino está cerca de la llamada, simplemente sobrescribe la instrucción para que apunte al principio de la función. En caso contrario, el linker agrega un *veneer* (enchapado) al módulo objeto que incluye el salto a la subrutina. El código original llama al *veneer,* que debe estar en el mismo modo de operación (Thumb o ARM). Su contenido en modo Thumb es:

si quiero mas registros {r0,r1,r2,...} se debe hacer un solo push y pop

cada inst ocupa 2 bytes

```
push {r0}
ldr r0, [pc, #8]   // Thumb no permite cargar R12 desde memoria
mov ip, r0
pop {r0}
bx ip
nop                // Alinear dirección de subrutina a 4 bytes.
dcd R_ARM_ABS32(x)  // Par (modo ARM) o impar (Thumb).
```

dice pc + 8 porque pc apunta a la siguiente instruccion.

ip es r12

direccion de inicio de la subrutina

La instrucción NOP (No operation) sirve para que la dirección de inicio de la subrutina a llamar se encuentre alineada a 4 bytes. El ensamblador genera el código de operación de MOV R0, R0.

Los registros R12, R13, R14 y R15 tienen nombres especiales reconocidos por el ensamblador que son: IP (Intra-Procedure call register), SP (Stack Pointer), LR (Link Register) y PC (Program Counter) respectivamente.

Con respecto al puntero de pila, cuando se ingresa un elemento, se decrementa SP y cuando se retira un elemento, se incrementa SP (generalmente 4 u 8 bytes). La pila también se usa para variables locales de la función, por lo que cuando arranca dicha función, se deberá restar a SP la cantidad de bytes necesarios para almacenar dichas variables locales. Al final de la función se deberá sumar a SP este mismo valor.

Pasaje de argumentos

Si el tipo de datos tiene 4 bytes o menos, se usan los registros R0 a R3 para los cuatro primeros argumentos (de izquierda a derecha) y luego se usa la pila para ubicar más argumentos. Si el argumento tiene menos bytes, se extiende con ceros o con signo según el tipo del dato para completar los cuatro bytes antes de ubicarlo en el registro.

Si un argumento es de 8 bytes, entonces debe ir en R0:R1 o en R2:R3 o en caso contrario en la pila. Es posible que algún registro no se use.

El resto de los argumentos se pasan por la pila ubicando primero el que corresponde al parámetro de la derecha. Siempre se debe tener en cuenta la alineación de los datos.

si tengo 6 parametros "A,B,C,D,E,F", de A a D van en r0 a r3, y luego se guardan en la pila F y E en ese orden, porque la funcion necesita los parámetros en el orden A,B,C,D,E,F

Si el tipo de argumento es una estructura, entonces se almacenan sus campos de la estructura como si fueran argumentos independientes. Si es un array, se almacena el puntero al array que ocupa cuatro bytes como si fuera un int.

Retorno de resultados

- Datos fundamentales de menos de 4 bytes: el dato se completa con ceros o con el bit de signo y se almacena en el registro R0.

- Datos fundamentales de 4 bytes: se almacenan en R0.

- Datos fundamentales de 8 bytes: se almacenan en R0:R1.

- Datos fundamentales de 16 bytes: se almacenan en R0:R1:R2:R3.

Ejemplo

Dado el siguiente módulo en C:

```
int suma_c(int a, int b)
{
   int c;
   c = a + b;
   return c;
}
```

Lo compilamos sin optimización, para poder observar en detalle el manejo de la ABI, mediante:

```
arm-linux-gnueabihf-gcc -g -O0 -std=gnu99 -
Wall -mcpu=cortex-a8 -c suma_c.c -o suma_c.o
```

La opción `-O0` compila sin optimización y `-c` compila el código fuente suma_c.c, pero no linkea (solo generamos el código objeto suma_c.o).

Luego obtenemos el desensamblado a partir del código objeto mediante el comando:

```
arm-linux-gnueabihf-objdump -d suma_c.o
```

El resultado es:

suma_c.o: file format elf32-littlearm

Disassembly of section .text:

```
00000000 <suma_c>:
   0:   b480        push {r7}            // Salvar registro a preservar.
   2:   b085        sub  sp, #20         // Espacio para variables locales.
   4:   af00        add  r7, sp, #0
// Guardar ambos argumentos como variables locales.
   6:   6078        str  r0, [r7, #4]    // Guardar 1er argumento (a).
   8:   6039        str  r1, [r7, #0]    // Guardar 2do argumento (b).
   a:   687a        ldr  r2, [r7, #4]    // Obtener 1er argumento (a).
   c:   683b        ldr  r3, [r7, #0]    // Obtener 2do argumento (b).
   e:   4413        add  r3, r2          // Realizar la suma.
  10:   60fb        str  r3, [r7, #12]   // Almacenar resultado (c).
  12:   68fb        ldr  r3, [r7, #12]   // Obtener resultado (c).
  14:   4618        mov  r0, r3           // Guardar en reg. de retorno.
  16:   3714        adds r7, #20          // Retirar vars locales de la pila.
  18:   46bd        mov  sp, r7
  1a:   f85d 7b04   ldr.w r7, [sp], #4    // Recuperar viejo valor de R7.
  1e:   4770        bx   lr               // Volver a rutina llamadora.
```

Compilado con optimización (reemplazando en el primer comando la opción `-O0` por `-O2`) resulta:

```
00000000 <suma_c>:
   0:   4408        add r0, r1           // Hacer la suma.
   2:   4770        bx lr                // Volver a rutina llamadora.
```

recordar que la ROM inicia en 0x70000000