TECNOLÓGICO
NACIONAL DE MÉXICO
INSTITUTO TECNOLÓGICO DE TIJUANA

INSTITUTO TECNOLÓGICO DE TIJUANA

DEPARTAMENTO DE SISTEMAS Y COMPUTACIÓN

INGENIERÍA EN SISTEMAS COMPUTACIONALES

SEMESTRE: ENERO - JUNIO 2023

LENGUAJE DE INTERFAZ SC6A

MAESTRO: RENE SOLIS REYES

UNIDAD A EVALUAR: 2

TEMA:

“Presentación y prácticas de ARM Assembly”

ALUMNOS:

GONZALEZ FERNANDEZ HIRAM(20211784)

VALDES UBIETA IVAN ALEJANDRO(20210647)

GONZALEZ LEMUS LUZ YEDID(20211786)

ORDOÑEZ MARTINEZ KAREN LESLIE(20212677)

TIJUANA B.C. 15 DE MARZO DEL 2023

Instrucciones ARM Assembly.

1. MOV: esta instrucción se utiliza para mover datos de un registro a otro registro. La sintaxis es la siguiente:
   MOV destino, origen
   Por ejemplo: MOV r0, #1 mueve el valor 1 al registro r0.
2. ADD: esta instrucción se utiliza para sumar dos valores y almacenar el resultado en un registro. La sintaxis es la siguiente:
   ADD destino, origen1, origen2
   Por ejemplo: ADD r0, r1, #2 suma el valor 2 al contenido del registro r1 y almacena el resultado en r0.
3. SUB: esta instrucción se utiliza para restar un valor de otro valor y almacenar el resultado en un registro. La sintaxis es la siguiente:
   SUB destino, origen1, origen2
   Por ejemplo: SUB r0, r1, #2 resta el valor 2 al contenido del registro r1 y almacena el resultado en r0.
4. CMP: esta instrucción se utiliza para comparar dos valores. La sintaxis es la siguiente:
   CMP valor1, valor2
   Por ejemplo: CMP r0, #0 compara el contenido del registro r0 con el valor 0.
5. B: esta instrucción se utiliza para saltar a una dirección de memoria específica. La sintaxis es la siguiente:
   B etiqueta
   Por ejemplo: B inicio salta a la etiqueta "inicio" en el código.

Ejemplo:

ROR{S}{cond} Rd, Rm, #<shift>

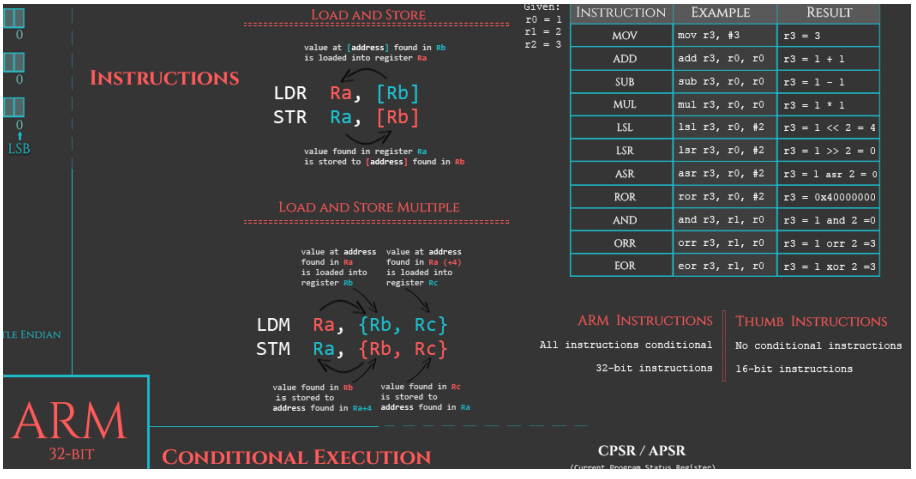
Donde:

- `{S}` es opcional y especifica si se debe actualizar el registro de estado (CPSR) con los resultados de la operación.
- `{cond}` es opcional y especifica una condición de ejecución de la instrucción.
- `Rd` es el registro de destino donde se almacenan los resultados de la operación.
- `Rm` es el registro fuente que contiene los datos que se van a rotar.
- `#<shift>` es un valor inmediato que especifica la cantidad de bits que se van a rotar.

Por ejemplo, la instrucción ROR con una rotación de 8 bits se puede escribir como sigue:

ROR R1, R2, #8

Esta instrucción rotará los bits en el registro R2 hacia la derecha en 8 posiciones, y el resultado se almacenará en el registro R1.



Características principales del
Conjunto de Instrucciones ARM
Todas las instrucciones son de 32 bits.
La mayoría de las instrucciones se ejecutan en un ciclo.
Cada instrucción se puede ejecutar condicionalmente.

Una arquitectura load/store
Instrucciones de procesamiento de datos actúan solo en registros
Formato de tres operandos
ALU y shifter combinados para una manipulación rápida de bits
Instrucciones de acceso a memoria específicas con potentes modos
indexados de direccionamiento
32 bit y 8 bit tipos de datos
Y también 16 bit tipos de datos en ARM 4.
Instrucciones de registro multiple flexible load and store
Conjunto de extensión de instrucciones via coprocesadores

TECNOLÓGICO
NACIONAL DE MÉXICO
INSTITUTO TECNOLÓGICO DE TIJUANA

Modos del Procesador
ARM tiene seis modos de operación:
User (unprivileged mode under which most tasks run)
FIQ (entered when a high priority (fast) interrupt is raised)
IRQ (entered when a low priority (normal) interrupt is raised)
Supervisor (entered on reset and when a Software Interrupt instruction is executed)
Abort (used to handle memory access violations)
Undef (used to handle undefined instructions)
Arquitectura ARM Versión 4 agrega un séptimo modo:
System (privileged mode using the same registers as user mode)

El Contador de Programa (R15)
Cuando el procesador está ejecutando en el estado ARM:
Todas las instrucciones son de 32 bits
Todas las instrucciones deben estar alineadas por palabra
Entonces el valor del PC está almacenado en bits [31:2] con bits
[1:0] iguales a cero(la instrucción no puede ser halfword o
alineada por byte).
R14 se usa como el registro link de subrutina (LR) y almacena
la dirección de retorno cuando operaciones Branch con Link se ejecutan calculadas desde el PC.

Así para retornar desde un linked Branch
MOV r15,r14
o
MOV pc,lr

Hay una ubicación de memoria dentro del proceso llamada Stack. El puntero de pila (SP) es un registro que, en circunstancias normales, siempre apuntará a una dirección dentro de la región de memoria de la pila. Las aplicaciones a menudo usan Stack para el almacenamiento temporal de datos. Y como se mencionó anteriormente, ARM usa un modelo Load/Store para el acceso a la memoria, lo que significa que las instrucciones LDR / STR o sus derivados (LDM.. /STM..) se usan para operaciones de memoria. En x86, usamos PUSH y POP para cargar y almacenar desde y hacia la pila. En ARM, también podemos usar estas dos instrucciones:

Cuando EMPUJAMOS algo en la pila descendente completa (más información sobre las diferencias de pila: Pila y funciones), sucede lo siguiente:

Primero, la dirección en SP se DISMINUYE en 4.
En segundo lugar, la información se almacena en la nueva dirección señalada por SP.

Cuando sacamos algo de la pila, sucede lo siguiente:

El valor en la dirección SP actual se carga en un registro determinado,
La dirección en SP se AUMENTA en 4.

En el siguiente ejemplo, usamos PUSH/POP y LDMIA/STMDB:

```
.text
.global _start

_start:
   mov r0, #3
   mov r1, #4
   push {r0, r1}
   pop {r2, r3}
   stmdb sp!, {r0, r1}
   ldmia sp!, {r4, r5}
   bkpt
```

Veamos el desmontaje de este código.

```
azeria@labs:~$ as pushpop.s -o pushpop.o
azeria@labs:~$ ld pushpop.o -o pushpop
azeria@labs:~$ objdump -D pushpop
pushpop: file format elf32-littlearm

Disassembly of section .text:

00008054 <_start>:
 8054: e3a00003 mov r0, #3
 8058: e3a01004 mov r1, #4
 805c: e92d0003 push {r0, r1}
 8060: e8bd000c pop {r2, r3}
 8064: e92d0003 push {r0, r1}
 8068: e8bd0030 pop {r4, r5}
 806c: e1200070 bkpt 0x0000
```

Como puede ver, nuestras instrucciones LDMIA y STMDB se tradujeron a PUSH y POP. Eso es porque PUSH es sinónimo de STMDB sp!, reglist y POP es sinónimo de LDMIA sp! registro

Ejecutemos este código en GDB.

```
gef> break _start
gef> run
gef> nexti 2
[...]
gef> x/w $sp
0xbefff7e0: 0x00000001
```

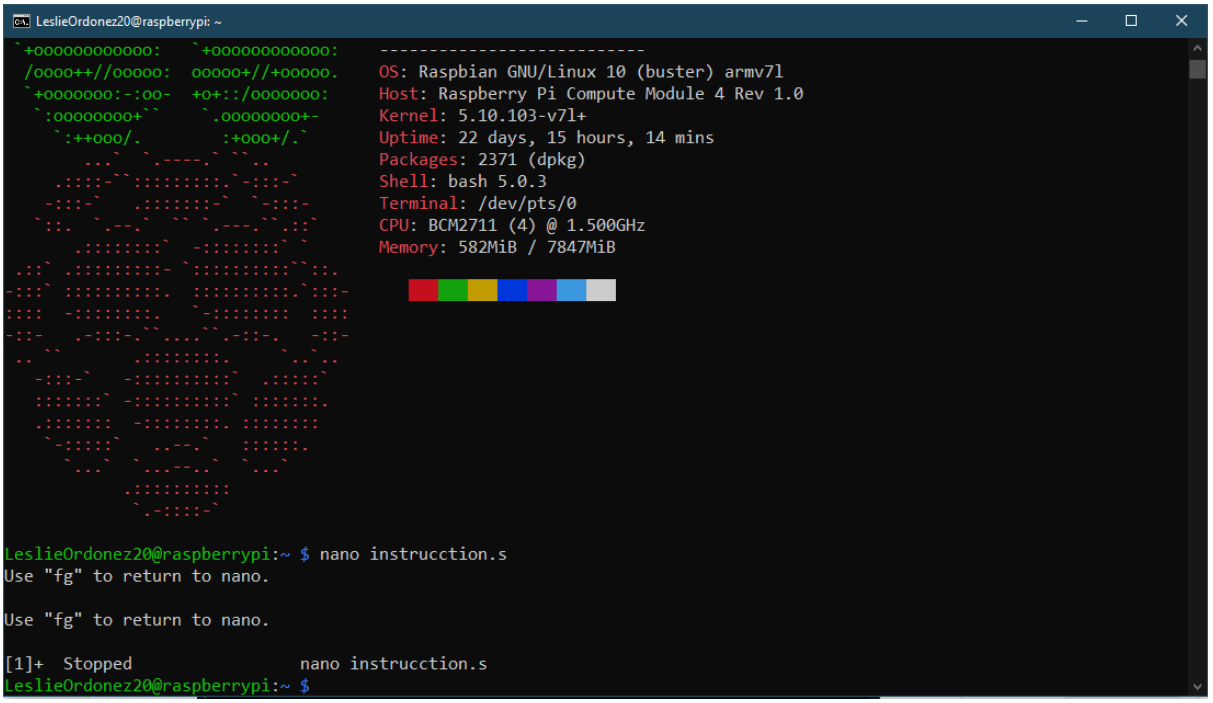
Después de ejecutar las dos primeras instrucciones, comprobamos rápidamente a qué dirección de memoria y valor apunta SP. La siguiente instrucción PUSH debería disminuir SP en 8 y almacenar el valor de R1 y R0 (en ese orden) en la pila.

```
gef> nexti
[...] ----- Stack -----
0xbefff7d8|+0x00: 0x3 <- $sp
0xbefff7dc|+0x04: 0x4
0xbefff7e0|+0x08: 0x1
[...]
gef> x/w $sp
0xbefff7d8: 0x00000003
```

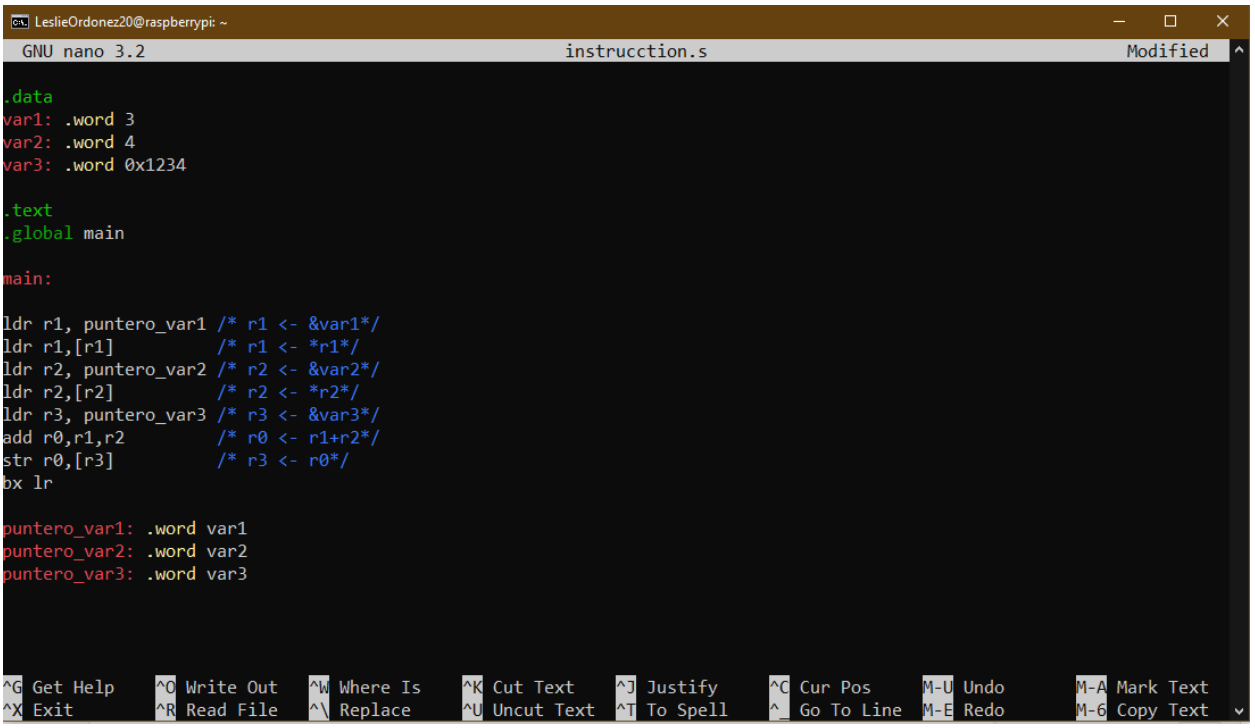
Luego, estos dos valores (0x3 y 0x4) se extraen de la pila a los registros, de modo que R2 = 0x3 y R3 = 0x4. SP se incrementa en 8:

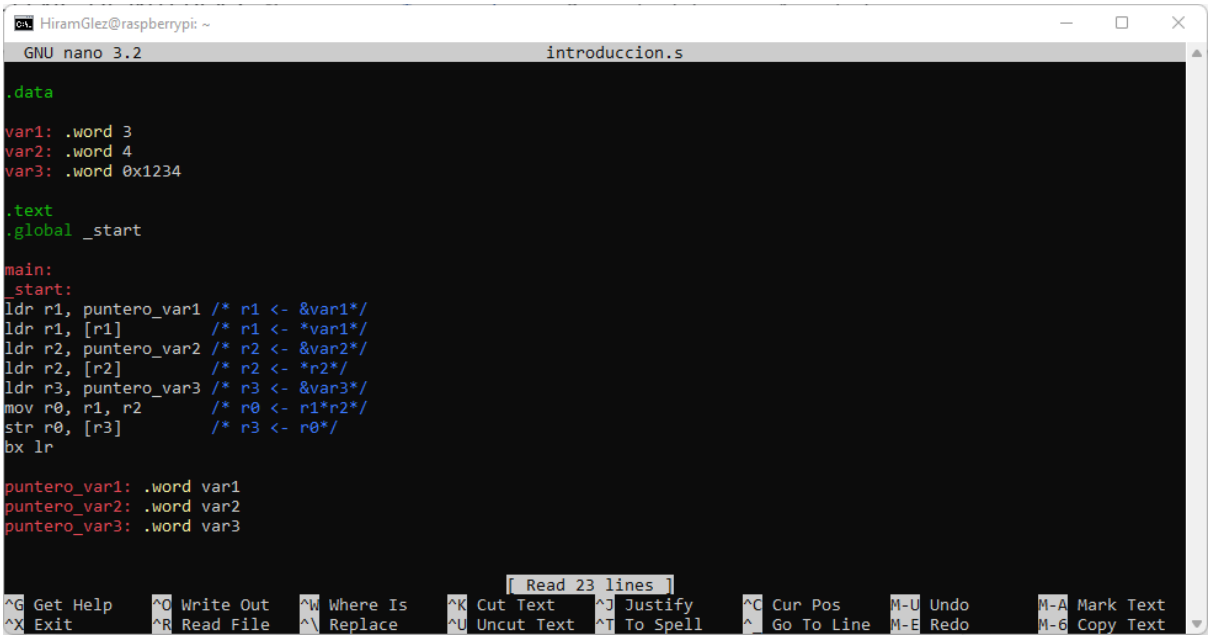
```
gef> nexti
gef> info register r2 r3
r2     0x3    3
r3     0x4    4
gef> x/w $sp
0xbefff7e0: 0x00000001
```

Ejemplos en GNU nano:



TECNOLÓGICO NACIONAL DE MÉXICO
INSTITUTO TECNOLÓGICO DE TIJUANA





Fuente bibliográfica:
https://www.programacion11.com/uploads/2/6/0/5/26056302/armsinstrucciones.pdf

TECNOLÓGICO
NACIONAL DE MÉXICO
INSTITUTO TECNOLÓGICO DE TIJUANA

https://azeria-labs.com/load-and-store-multiple-part-5/