ARM

ARM surge por la necesidad de tener procesadores de baja potencia y poder programar lenguajes de ensamblado para los mismos (internet de las cosas, heladeras, celulares, etc).

Caracteristicas principales:

-Posee una arquitectura Load/Store, lo que impacta en como se procesan las instrucciones. La forma de comunicarse con la memoria se basan solamente en estas instrucciones.

-Todas las instrucciones son de 32 bits. En Intel son de longitud variable

-Todas las instrucciones poseen 3 direcciones: 2 registros de operandos y 1 registro de resultado. A diferencia de intel que el resultado suele quedar en el primer operando.

-ARM tiene una ejecución condicional de TODAS las instrucciones. Todas las instrucciones se ejecutan si se cumplen ciertas condiciones. Lo que hace ARM no se basa en bifurcaciones, si no que se tienen ciertas funciones que se ejecutan en base a un resultado previo.

-Existen instrucciones de Load Store de registros Múltiples. Puedo cargar información de la memoria a varios registros sin necesidad de ejecutar varias instrucciones.

Organización de registros

- R0 a R12 son registros de propósito general (32 bits)
  - Usados por el programador para (casi) cualquier propósito sin restricción
- R13 es el *Stack Pointer* (SP)
- R14 es el *Link Register* (LR)
- R15 es el *Program Counter* (PC)
- El *Current Program Status Register* (CPSR) contiene indicadores condicionales y otros bits de estado

Todos los registros son de 32 bits.

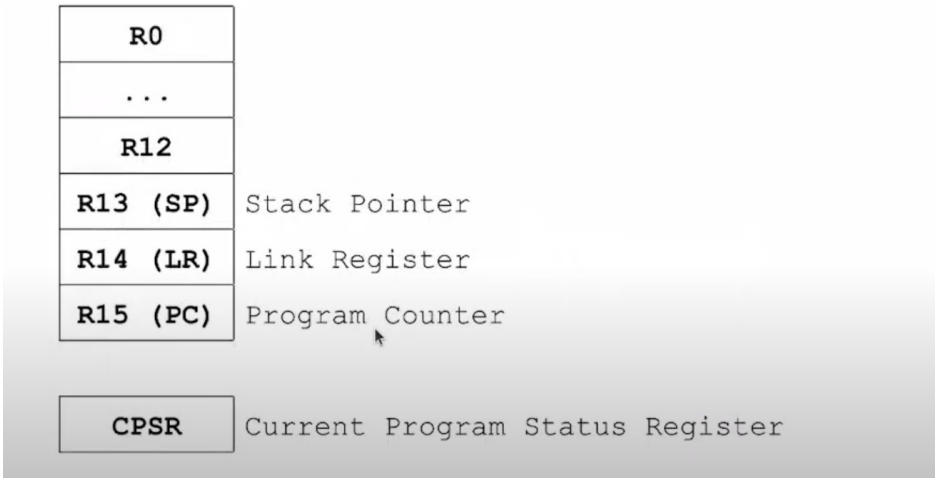
Si bien los generales los puedo usar para todo, pero hay que tener en cuenta que algunos se utilizan para interrupciones de software para resolver cosas con el sistema operativo.

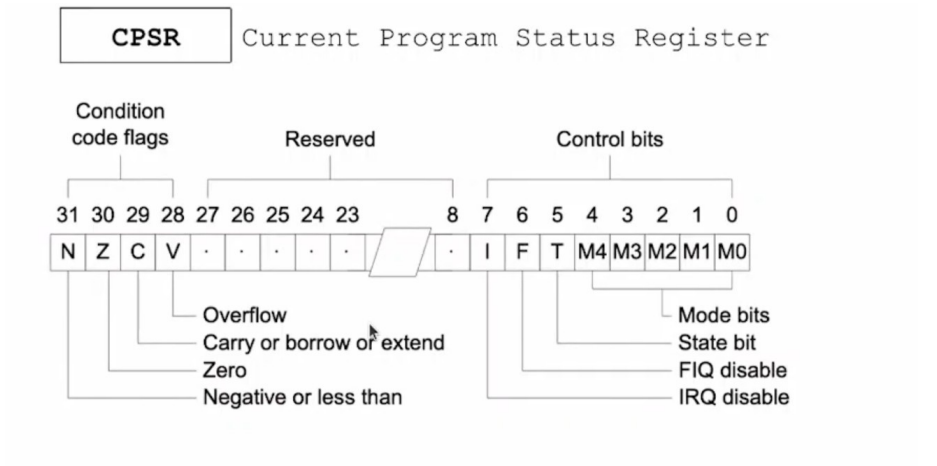
R13 y SP es lo mismo, apunta a la primera posición de la pila.

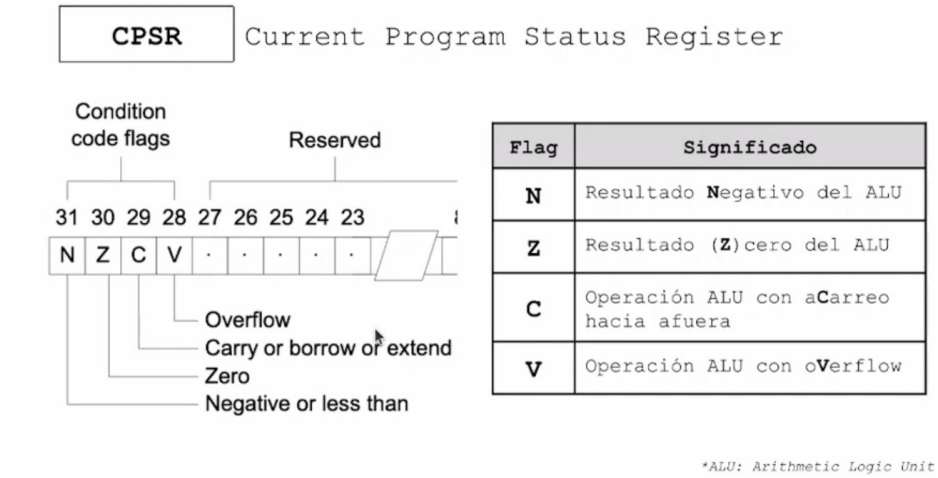
Con el R14 o LR puedo volver a un flujo principal, con esto puedo crear rutinas internas.

R15 es como el RPI en abacus, tiene la dirección de la próxima dirección a ser ejecutada.

El CPSR tiene los flags de estado.





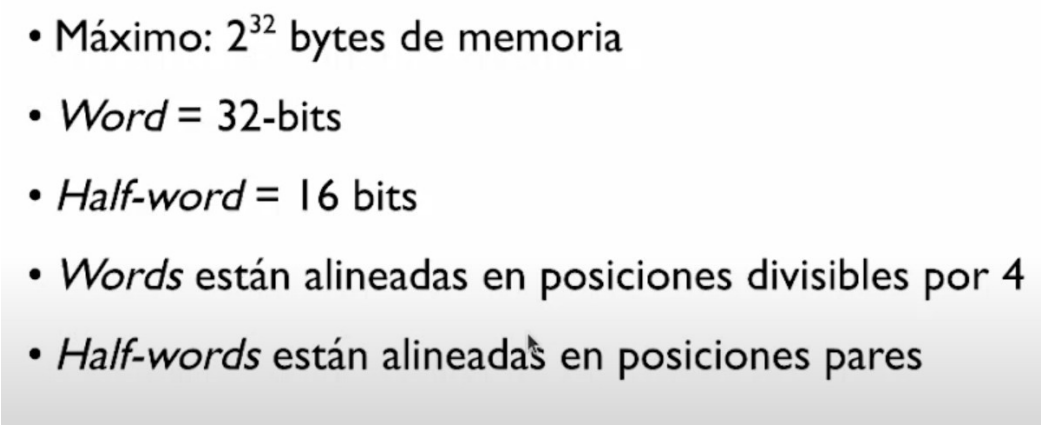


Flag	Significado
N	Resultado **N**egativo del ALU
Z	Resultado (**Z**)cero del ALU
C	Operación ALU con a**C**arreo hacia afuera
V	Operación ALU con o**V**erflow

**ALU: Arithmetic Logic Unit*

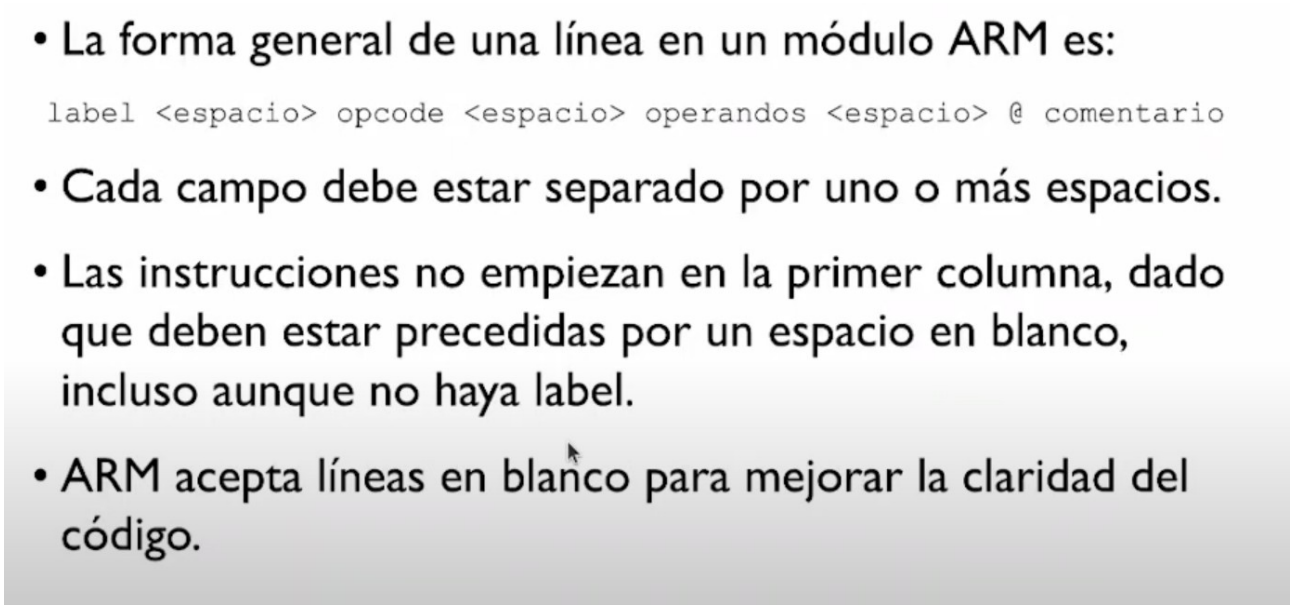
Esto se accede indirectamente cuando quiero ver algún resultado lógico o aritmetico.

Organización de la memoria



Cuando se guarda algo del tamaño de una Word, las posiciones dentro de esa WORD son divisiones de 4 bits. En una Half-words se alinean las posiciones de a pares (8 bits). Si yo defino una WORD en una posición no divisible por cuatro me quedan 2 bytes libres (???).

Estructura de un Programa



el opcode no puede ser lo primero de la instrucción, como mínimo tiene que haber un espacio antes.

```
        .text                           @ Indica que los siguientes
                                        @ ítems en memoria son
                                        @ instrucciones
start:
        mov     r0, #15                 @ Seteo de parámetros
        mov     r1, #20
        bl      func                    @ Llamado a subrutina
        swi     0x11                    @ Fin de programa
func:                                   @ Subrutina
        add     r0, r0, r1              @ r0 = r0 + r1
        mov     pc, lr                  @ Retornar desde subrutina
        .end                            @ Marcar fin de archivo
```

Con # agarro operandos inmediatos númericos. No necesariamente es inmediato, depende de la instrucción. Puedo querer guardarlo en memoria y referenciarlo también.

Para llamar a una subrutina uso **bl func (branch with link)**. La diferencia entre branch y branch with link es que puedo volver a la parte desde donde se llamo.

Los operandos se ordenan asi:

```
func:                                   @ Subrutina
        add     r0, r0, r1              @ r0 = r0 + r1
```

Para volver a la subrutina, al program counter tengo que decirle que copie el valor del link register, que cuando hice bl, me guarde la instrucción que le sigue a la bifurcación para volver.

```
mov     pc, lr                  @ Retornar desde subrutina
```

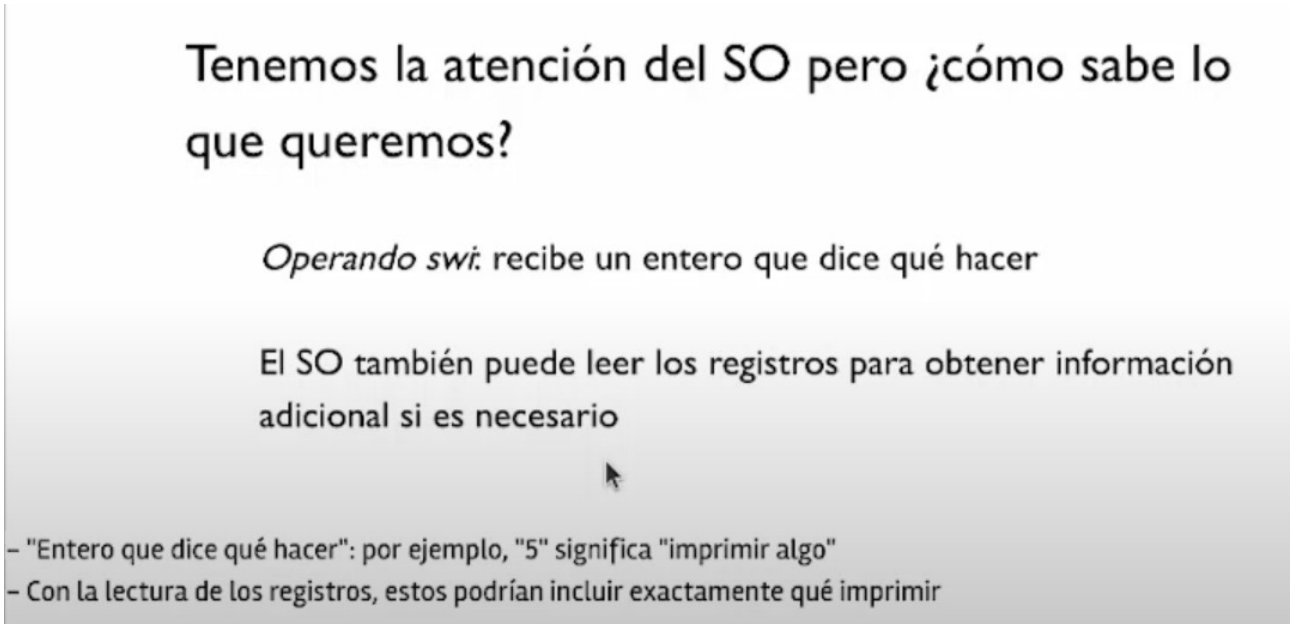
Interrupciones de Software

Mediante los mismos le solicitamos al sistema operativo que vaya a buscar información y que nos devuelva un resultado. Es una manera de decirle al sistema operativo que tome el control.

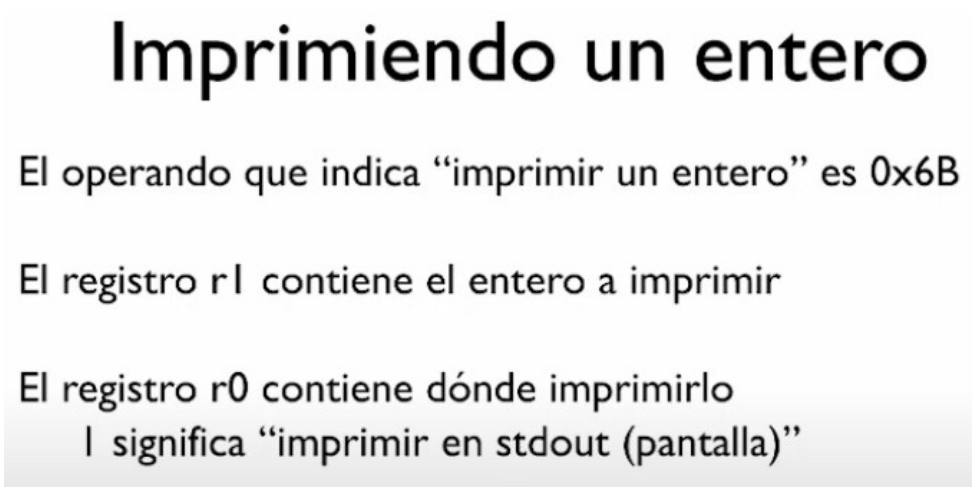
Por ejemplo para imprimir algo:

ARM cuenta con una instrucción *swi* que provoca una interrupción de software

Para que el SO sepa que es lo que quiero que haga se le pasa un entero que significa que es lo que quiero que haga.



Ejemplo:




```
mov   r0, #5
mov   r1, #7
add   r2, r0, r1
mov   r1, r2        ;  r1: entero a imprimir
mov   r0, #1        ;  r0: dónde imprimir
swi   0x6B          ;  0x6B: imprimir entero
```

Finalizar el programa

Necesitamos indicar el fin del programa

¿Cómo puede hacerse?

swi con un operando particular (0x11)

```
mov   r0, #5
mov   r1, #7
add   r2, r0, r1
mov   r1, r2        ;  r1: entero a imprimir
mov   r0, #1        ;  r0: dónde imprimir
swi   0x6B          ;  0x6B: imprimir entero
swi   0x11          ;  0x11: salir del programa
```

Secciones del programa

Entendiendo al todo como un gran conjunto de bits, las secciones le dicen al ensamblador que bits deben colocarse en qué parte de la memoria. Por ejemplo la directiva **.text** indica que lo de abajo que tenemos es código. La directiva **.data** específica la sección de variables:

```
   .data
string1:
   .asciz "hola"
string2:
   .asciz "chau"
```

Podes escribir .ascii o .asciz, la diferencia que hay entre ambos es que .asciz te define un cero al final para marcar el fin de string.

ARMSim#

Estructura básica de un programa

```
    .data
cadena:
    .asciz "linea"
entero:
    .word 78
    .text
    .global _start
_start:
    @ Comentario
    swi 0x11
    .end
```

La razón de la identación rara es porque las directivas (lo que señalan las secciones del programa) no pueden arrancar con identación 0, si no que tienen que tener un espacio antes. En cambio, los label si que tienen que arrancar con espacio 0 para que sean tomados como tal. Por eso parece que .text y .global _start están definidos como parte de entero, pero en realidad dan arranque a una sección distinta.

Imprimir cadena de caractéres

ARMSim#

Salida standard de cadena de caracteres

```
    .data
cadena:
    .asciz "Soy una cadena"
    .text
    .global _start
_start:
    ldr r0, =cadena
    swi 0x02
```

Contexto en el que surge ARM

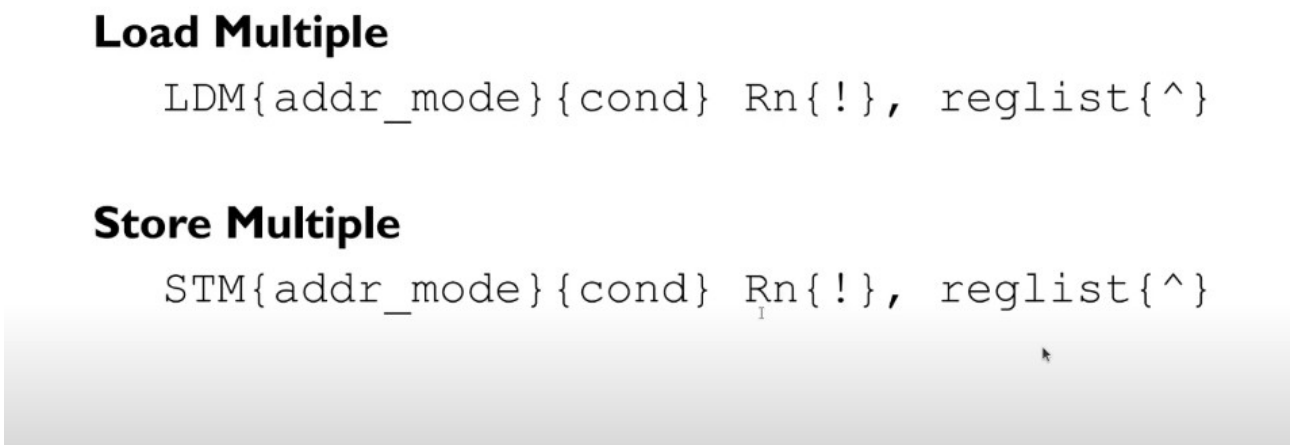
ARM surge ante la necesidad de un procesador para los asistentes digitales apple. Los procesadores hasta ese entonces en el mercado se caracterizaban por tener como prioridad más que nada el desempeño y este se caracterizó por tener en cuenta también el consumo como un factor primordial, dado que se requería para dispositivos portatiles de baterias con tecnología no tan avanzada como al dia de hoy.

El modelo de negocio de ARM era a partir de licencias, en vez de lo que comunmente se hacía que cada empresa creaba su propio procesador. A partir de esto las empresas compran su licencia y cada una a partir de conocer el código fuente fue integrando distintos perísfericos a partir del mismo procesador.

Otra caracteristica distintiva de ARM para la época fue desligar el hardware del software. Antes el software de cada procesador era exclusivo para el mismo. ARM en cambio crea el ISA (Instructions Set Arquitecture). Esto refiere a la generalización del set de instrucciones para el desarrollo del software, para que sea independiente el echo de que el procesador sea orientado al desempeño o al consumo.

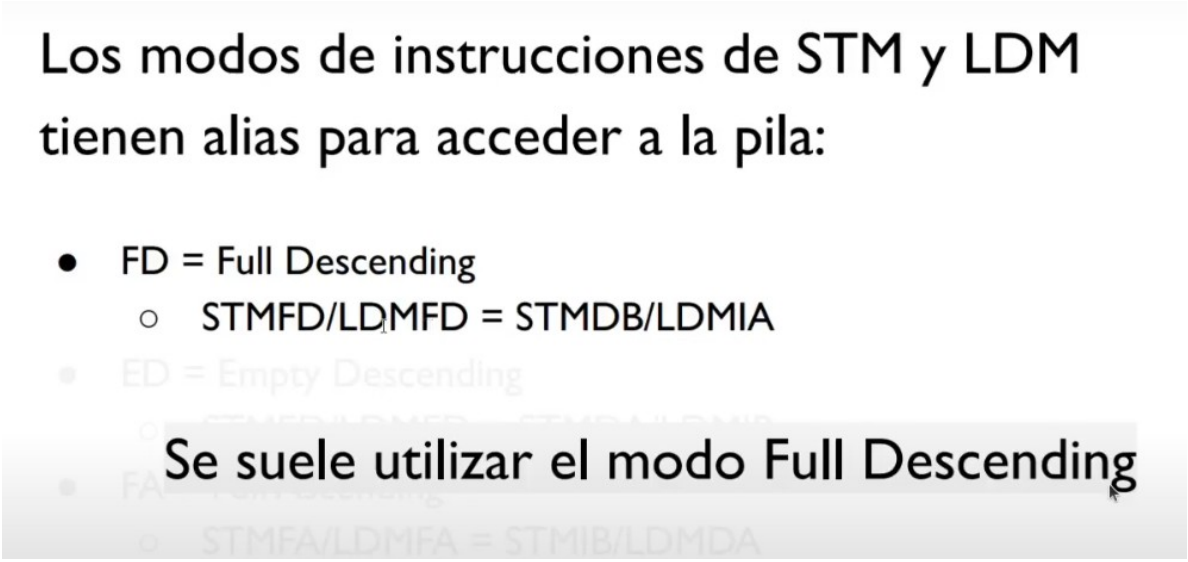
Stack + Subrutinas

Al igual que en Intel, para llamar a subrutinas tengo que reservar espacio en el Stack tal que se alteran las instrucciones ya dadas de por sí por el sistema operativo. Para interactuar con el Stack (tal como se hace en Intel con push y pop) se puede usar **LDM (Load Multiple)** y **STM (Store Multiple)**. El proposito de estas instrucciones es la de cargar múltiples direcciones en memoria en registros y viceversa



Es posible determinar el sentido en el cual se hace con dos letras más reservadas para ello:

Por lo general se usa el Full descending:



Acá para llamar a una subrutina usas exclusivamente bl (branch linker). La razón por la cual solamente hay una única instrucción y no con condicionales como en Intel es porque ya de por si las instrucciones de ARM son de ejecución condicional.

Por ejemplo, en esta secuencia de instrucciones, añado al stack pointer el r0 y el lr a partir de store multiple. Luego cargo r0 con el valor 1 (imprimir por stdout), llamo a la subrutina de impresión (que se encargará de mostrar el R1 por consola). Se imprime un carácter de fin de linea como una cadena de caractéres (cuando se muestra el EOL por cónsola).

Llamado a subrutina

```
        bl print_r1_int
       ...
```

Subrutina

```
print_r1_int:
        stmfd sp!, {r0,lr} @ Stack R0
        ldr r0, #0x1
        swi SWI_Print_Int  @ Mostrar entero en R1 por consola
        ldr r1, =EOL
        swi SWI_Print_Str  @ Mostrar EOL por consola
        ldmfd sp!, {r0,pc} @ Unstack r0
```

Instrucciones aritmeticas

Add

```
ADD{cond}{S} Rd, Rn, <Oprnd2>
```

Subtract

```
SUB{cond}{S} Rd, Rn, <Oprnd2>
```

Reverse subtract

```
RSB{cond}{S} Rd, Rn, <Oprnd2>
```

Multiply

```
MUL{cond}{S} Rd, Rm, Rs
```

La condición puede estar como no, si no se pone nada se ejecuta siempre

Instrucciones lógicas

And

```
AND{cond}{S} Rd, Rn, <Oprnd2>
```

Exclusive Or

```
EOR{cond}{S} Rd, Rn, <Oprnd2>
```

Or

```
ORR{cond}{S} Rd, Rn, <Oprnd2>
```

Desplazamientos

El desplazamiento lógico hace una rotación entre los operandos y el desplazamiento aritmético se hace cuando se completa con 1 o 0 para conservar el signo

Se puede rotar una cantidad específicada en otro registro con **Barrel Shifter:**

Cuando se especifica que el segundo operando es un registro *shifteado*, la operación del Barrel Shifter es controlada por el campo Shift en la instrucción.

Este campo indica el tipo de *shift* a realizar.

La cantidad de bits a *shiftear* puede estar contenida en un campo inmediato o en el byte inferior de otro registro (que no sea el R15)

Cuando hago un AND entre dos números, comparo bit a bit:

```
@ r0 = 5 = 0101
@ r1 = 2 = 0010
@ r5 = 0000
```

Archivos

Definiciones

```
    .data
filename:
    .asciz "archivo.txt"
    .align
InFileHandle:
    .word  0
```

Apertura

```
ldr r0,=filename        @ (R0) -> nombre de archivo
mov r1,#0               @ (R1) -> modo: entrada
swi 0x66
bcs InFileError
ldr r1,=InFileHandle    @ (R1) -> InFileHandle
str r0,[r1]             @ almacena handler
```

en r1 pones el modo. 0 = lectura y 1 = escritura

Leer entero desde archivo

```
ldr r0,=InFileHandle    @ (R0) -> InFileHandle
ldr r0,[r0]             @ (R0) = InFileHandle
swi 0x6C
                        @ (R0) = Entero leído
```

Cerrar archivo

```
ldr r0,=InFileHandle    @ (R0) -> InFileHandle
ldr r0,[r0]             @ (R0) = InFileHandle
swi 0x68
```

Salida standard de entero

```
mov r0,#1 @ (R0) = Stdout (salida por pantalla)
mov r1,r2 @ (R1) = entero a mostrar
swi 0x6B
```

NOTA:

MOV es para mover datos entre registros.

LDR es para traer algo de la memoria a un registro (Load register)

STR es para guardar algo de un registro en la memoria (Store register)

Compare

La instrucción compare funciona a partir de la resta de dos operandos, descartando el resultado. En este proceso se setean los flags, cuyos bits de estado me reflejan el resultado de la comparación:

Los bits de estado dicen algo sobre el resultado de las comparaciones aritméticas

Los operandos son iguales	`mov r0, #5` `cmp r0, #5`	Setea bit/flag cero (resultado es cero)
1° operando < 2° operando	`mov r0, #5` `cmp r0, #20`	Setea bit/flag negativo (resultado es negativo)

Ejecución condicional

A cada instrucción podes añadirle dos letras para que se ejecuten condicionalmente por un bit de estado:

`mov`**`mi`** `r0, #42` — mover si el bit negativo está encendido

mi = minus. Esto sucederá por ejemplo si cmp activo el bit de negativos. Por ende tiene que haber una ejecución previa que justamente lo haga.

`mov`**`pl`** `r0, #42` — mover si el bit negativo **no** está encendido

Pl = plus

`mov`**`eq`** `r0, #42` — mover si el bit cero está encendido

eq = equal

ne = not equal

Instrucción Branch

Sirve para saltar a una etiqueta en específico. Si quiero guardar la siguiente instrucción en el link register para volver hacía donde estaba, uso bl. En este caso mov r0, #5 no se ejecuta.

```
  mov r0, #1
  b otra
  mov r0, #5
otra:
  mov r1, r0
```

Branch condicional

Tras un cmp puedo usar los postfijos eq, min, pl, ne , etc para hacer un salto condicional:

```
  mov r0, #0
  mov r1, #5
  cmp r1, #5
  beq otrolado
  mov r0, #25
otrolado:
  mov r2, r0
```

En este caso tras hacer el cmp el flag de 0 esta en 1 y salta para el otro lado, haciendo que mov r0, #25 no se ejecute.

Loops

Se utiliza la instrucción Branch con saltos al inicio o al final del loop.

Operaciones en memoria

La arquitectura ARM sabemos que se basa en una lógica Load/Store, en el sentido de que no se interactua directamente con la memoria cuando se quiere operar con algún dato guardado allí, si no que se translada dicha información a un registro y es a partir de ahí que se interactua con el mismo.

Esta caracteristica de la arquitectura permite un mejor desempeño del hardware, permitiendo que una parte del procesador se encargue de dicha instrucción de Load mientras otra esta ejecutando la instrucción actual.

De por sí sabemos que con un **ldr** es posible cargar una dirección en memoria a un registro:

```
    .data
mensaje:
    .asciz "hola_mundo"
entero1:
    .word 42
entero2:
    .word 38
```

Teniendo en cuenta que estamos trabajando en un procesador de 32 bits, pudo definir un entero de dicho tamaño bajo un rotulo:

```
    .data
mensaje:
    .asciz "hola_mundo"

    .text
    ldr r0, =mensaje
```

Leer enteros desde memoria

En principio lo que hago es cargar la dirección en memoria de dicho entero a un registro con ldr:

```
   .data
entero1:
   .word 42
entero2:
   .word 38
   .text
   ldr r0, =entero1
```

Una vez que hago esto, en otro registro me cargo el contenido a partir de corchetes:

```
   ldr r0, =entero1
   ldr r1, [r0]
```

Almacenar enteros en memoria

En este caso tengo que tener en un registro cargado la dirección a donde voy a querer almacenar mi entero. A partir de tener dicho entero en otro registro, hago **str (store)** al “contenido” del registro original donde tengo donde lo quiero guardar:

```
   .data
entero1:
   .word 42
entero2:
   .word 38
   .text
   ldr r0, =entero1
   mov r1, #57
   str r1, [r0]
```

Arrays

De la misma manera en la que yo me guardaba enteros bajo un rotulo, puedo guardarme una secuencia de numeros bajo dicho rotulo:

```
   .data
entero1:
   .word 42
array:
   .word 32, 65, 76, 87
```

Visto desde un enfoque básico, cada celda en memoria tendría 4 bytes, por ende cuando me quiero desplazar entre los distintos campos tengo que ir sumando de a 4:

```
   .data
arr:
   .word 32, 65, 76
   .text
   ldr r0, =arr
   ldr r1, [r0]
   add r0, r0, #4
   ldr r2, [r0]
   add r0, r0, #4
   ldr r3, [r0]
```

ARM 64 BITS

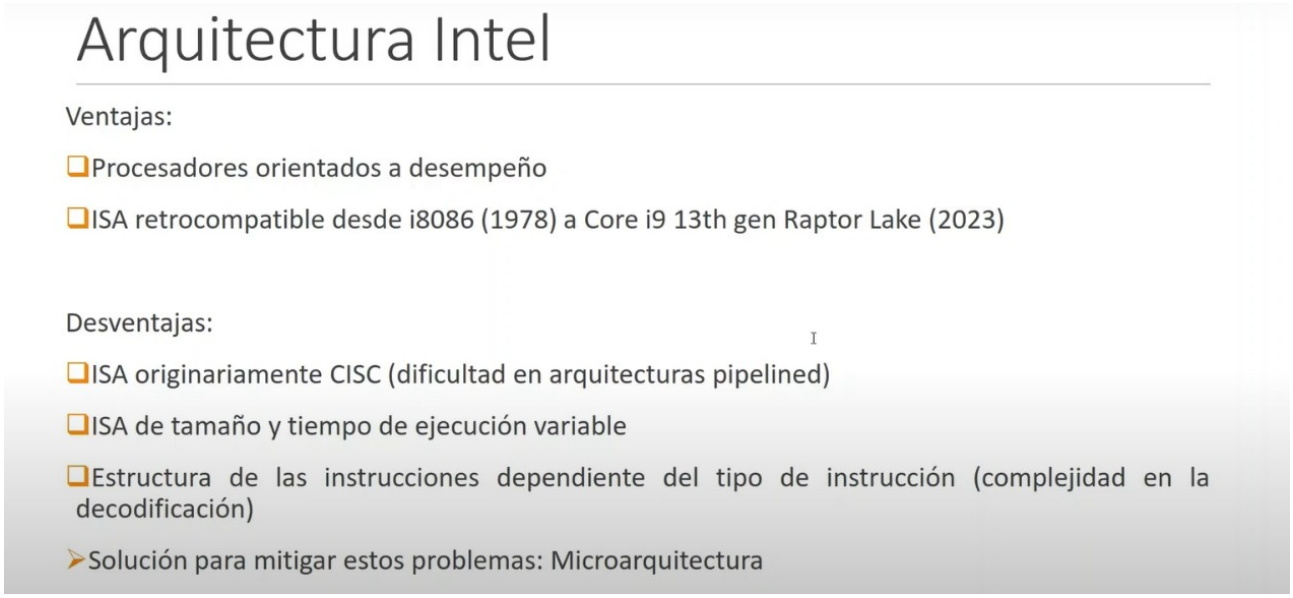
Los procesadores ARM se caracterizarón desde un inicio por tener software compatible independientemente del hardware, de modo que la manera en que se gestionaba el consumo de las aplicaciones era algo que se encargaba el sistema operativo. De esta forma las aplicaciones se podian desligar de en que hardware se estaba corriendo.

Si vamos a lo que es arquitectura de procesadores, sabemos que la arquitectura intel siempre estuvo orientada a desempeño y que tiene software retrocompatible. Todos los procesadores de intel son capaces de leer software mas antiguo.

La desventaja de intel es que en un principio las instrucciones que interactuaban con el procesador se escribian a mano, lo que llevaba a generar instrucciones complejas en las que se realizaban más de una operación, comparación, etc. El tener estas instrucciones requería tener un hardware complejo tal que pueda realizar todo eso que se solicitaba. Esto si bien es comodo para el programador, trae dificultades para arquitecturas pipelined.

El tamaño y tiempo de ejecución de un software en intel depende directamente de la cantidad de comparaciones y operaciones que se hagan en dicha instrucción. Esto no es bueno tal que para medir performance del software yo no quiero tener variaciones de lo que puede llegar a pesar una instrucción.

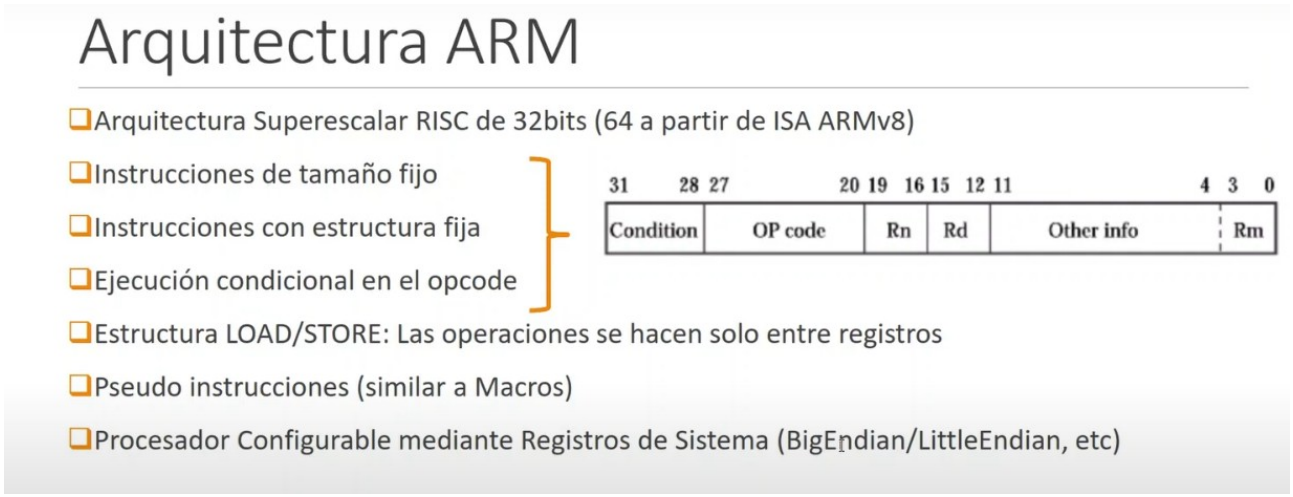
El formato en el que se da la instrucción también hace variar la complejidad de su decodificación. En intel tengo que leer la instrucción, entender que es lo que hace y en función de lo que hace interpretar cada campo de bits de dicha instrucción. Esto en ARM no pasa tal que tengo ciertos códigos representativos para cada instrucción. La forma en la que Intel solucionó este problema es a partir de **microarquitecturas**. Esta remplaza una secuencia de instrucciones en assembler a una instrucción más simple con un tiempo definido de ejecución. Esto ayuda más a hacer predicciones.



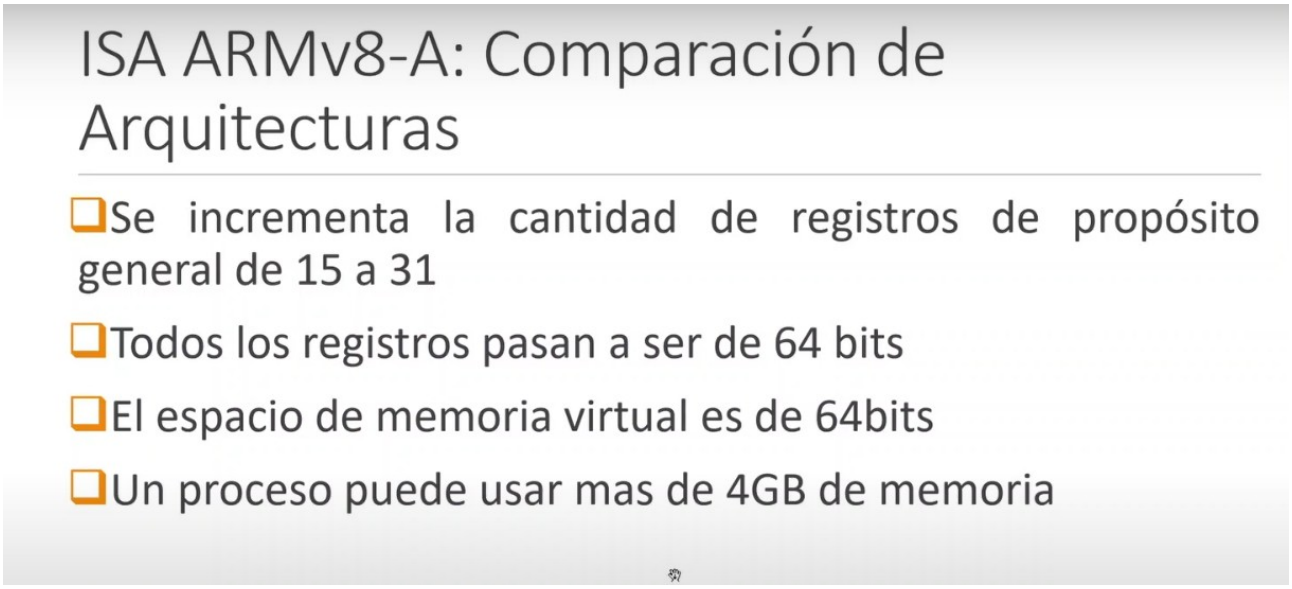
ARM intenta ofrecer un sistema más simple para que esto no pase. Todas las instrucciones son simples (constantes), de modo que todas las instrucciones ocupan 32 bits (o 64 a partir de ISA ARMv8). El tamaño y estructura es fijo y la ejecución condicional se encuentra en el mismo opcode.

La estructura de LOAD/STORE me permiten poner todos los argumentos dentro de una misma instrucción sin necesidad de hacer algo como una búsqueda en memoria.

Tengo pseudo instrucciones similares a las macros y los procesadores se pueden configurar según como registre el sistema (little o big endian).



Otras ventajas de ARM 64bits:



Hay más...