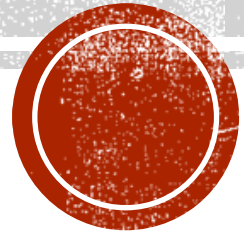
ENSAMBLADOR

Con Énfasis en el Motorola 88110



La charla se subirá a https://www.youtube.com/channel/UCdu-SvdBJ2G5K3mkOyj-sNQ

CONSIDERACIONES PREVIAS

No soy un experto

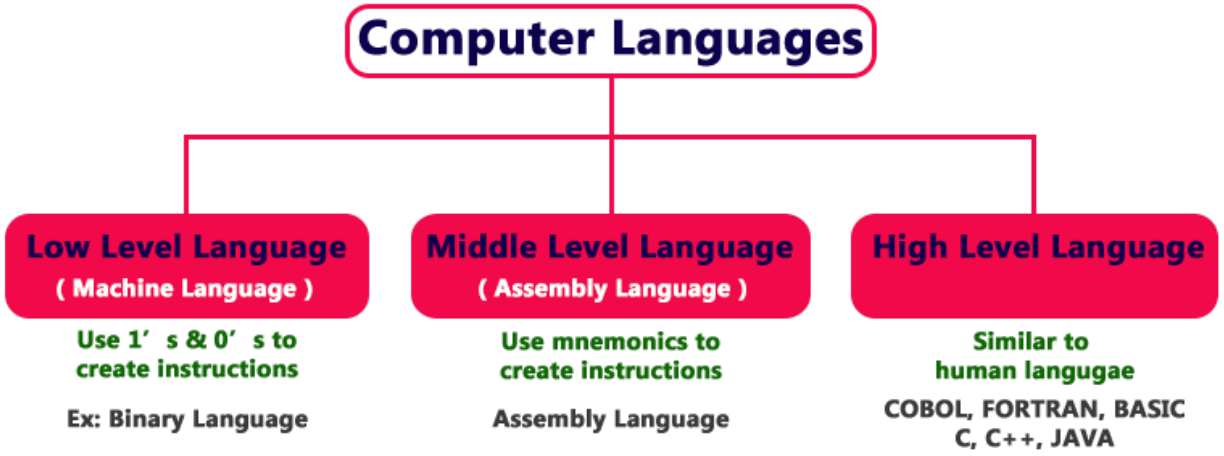
Si hay errores, podéis decirlo en los comentarios y corregiré el vídeo o avisaré de los mismos lo antes posible.

Solo pretendo ofreceros mi forma de entender el lenguaje ensamblador.

Cada apartado teórico irá acompañado de apartado práctico.

Las instrucciones que aparezcan no tienen porqué ser todas las instrucciones disponibles.

Lo que aquí expongo carece de relación con el material expuesto en cualquier institución del conocimiento salvo en las herramientas usadas y cuya relación queda explícitamente definida con su introducción.



- https://www.cs.mtsu.edu/~xyang/2170/computerLanguages.html
- https://en.wikipedia.org/wiki/Assembly_language

¿QUÉ ES ENSAMBLADOR?

MATERIAL A UTILIZAR

- Emulador del Motorola 881100 del Departamento de Arquitectura y Tecnología de Sistemas Informáticos de la UPM disponible en https://www.datsi.fi.upm.es/docencia/Estructura_09/Proyecto_Ensamblador/
- Opcionalmente:
  - Scripts de Compilación y Ejecución "fácil": https://github.com/M-T3K/UPM/tree/master/Estructura
  - Visual Studio Code: https://code.visualstudio.com
  - Extensión M88K para VSCode: https://marketplace.visualstudio.com/items?itemName=Kiwii.m88k
  - Repositorio de la charla:

    https://github.com/M-T3K/UPM/tree/master/Clases%20de%20Apoyo/Ensamblador-18Diciembre2021/

    Podéis descargarlo con **git (https://git-scm.com/downloads)** con el mandato
    git clone https://M-T3K/UPM.git
- Además, podéis descargar el manual oficial de los procesadores MC88110: http://praxibetel.org/reference/motorola/MC88110_Users_Manual.pdf

ENSAMBLADOR: TIPOS DE INSTRUCCIONES

CONTROL DE FLUJO

- Son las que permiten decidir qué partes del programa ejecutamos y cuando

LÓGICAS

- Son las que permiten realizar comparaciones lógicas

ARITMÉTICAS

- Nos permiten realizar operaciones sobre registros

MEMORIA

- Nos permiten acceder y realizar modificaciones sobre la memoria principal.

MIXTAS

- Muchas operaciones, aunque no lo parezca, realmente son mixtas y realizan varias funciones.
- En M88110, **bb0** combina el control de flujo de una instrucción tipo **branch** con una lógica tipo **compare**.

PSEUDOINSTRUCCIONES

- No suelen tener uso para el hardware
- Sirven para dirigir el software ensamblador o compilador.

ENSAMBLADOR: MODOS DE DIRECCIONAMIENTO

- Viene a definir los operandos disponibles y su funcionamiento como instrucción.
- Tienen que ver con la dirección de memoria y cómo se calcula.
- No entro en detalle, solo lo vemos por encima.
- Generalmente afectan a la cantidad de operandos en una instrucción, ya que permite operar con más o menos registros simultáneamente. Por ejemplo:

INSTRUCCION1 OP1, OP2, OP3	INSTRUCCION2 OP1, OP2
MOTOROLA 88110	MOTOROLA 68000
MAYORMENTE, ALGUNAS INSTRUCCIONES TENDRÁN MENOS (GENERALMENTE LAS DE CONTROL DE FLUJO)	

INSTRUCCIONES ARITMÉTICAS ENTERAS

- Existen instrucciones que permiten operar con números enteros con signo:
  - add
  - sub
  - muls
  - divs
- En algunos casos, necesitamos operar con números sin signo. Existen versiones **unsigned** de estas instrucciones:
  - addu
  - subu
  - mulu
  - divu

- Problema #1
  Queremos guardar en r5 el resultado de multiplicar el 7 y el 2 sin signo y luego dividirlo por -2.
- Solución:

```
addu r4, r0, 2 ;  r4 = 2
mulu r4, r4, 7 ;  r4 = r4*2 = 7 * 2 = 14
divs r5, r4, -2 ; r5 = r4 / -2 = 14 / -2 = -7
```

- Resultado:
  R05 = FFFFFFF9 h (0x FFFFFFF9 = -7)

PROBLEMA #1: EJECUCIÓN DEL PROGRAMA

- Para ejecutar y compilar usamos los siguientes comandos (en Windows):
  - Compilar: **88k_Windows_v10\88110e -ml -o files\test.bin files\problema1.ens**
  - Ejecutar: **start 88k_Windows_v10\88110e mc88110.bat files\test.bin**
  - Como todavía no se están usando etiquetas, no es recomendable usar el script de compilación.
- Una vez ejecutado compilado y ejecutado el emulador, obtenemos la siguiente ventana:

```
PC=0          addu     r04,r00,2        Tot. Inst: 0   ¡¡ Ciclo : 1
FL=1 FE=1 FC=0 FV=0 FR=0
R01 = 00000000 h R02 = 00000000 h R03 = 00000000 h R04 = 00000000 h
R05 = 00000000 h R06 = 00000000 h R07 = 00000000 h R08 = 00000000 h
R09 = 00000000 h R10 = 00000000 h R11 = 00000000 h R12 = 00000000 h
R13 = 00000000 h R14 = 00000000 h R15 = 00000000 h R16 = 00000000 h
R17 = 00000000 h R18 = 00000000 h R19 = 00000000 h R20 = 00000000 h
R21 = 00000000 h R22 = 00000000 h R23 = 00000000 h R24 = 00000000 h
R25 = 00000000 h R26 = 00000000 h R27 = 00000000 h R28 = 00000000 h
R29 = 00000000 h R30 = 00000000 h R31 = 00000000 h
```

PROBLEMA #1: EJECUCIÓN DEL PROGRAMA

- Arriba puede verse la instrucción en la que estamos y más información:

```
PC=0          addu     r04,r00,2        Tot. Inst: 0   ¡¡ Ciclo : 1
FL=1 FE=1 FC=0 FV=0 FR=0
```

- También podemos pedir información de uso:

```
88110> h
 Q .- Termina la simulación.
 H .- Ayuda de comandos.
 R [número valor] .- Presenta/Modifica los registros de máquina
 D <esp_dirección> [<n_i>].- Desemsambla n_i instrucciones a partir de dirección
 P [(+|-) esp_dirección] .- Puntos de ruptura, + añade, - suprime.
 E .- Ejecuta el programa.
 V <esp_dirección> [<n_pal>].- Presenta el contenido de n_pal palabras de memoria.
 T [número].- Ejecuta una o varias instrucciones
 I [esp_dirección] valor .- Modifica el contenido de una palabra de memoria
 L.- Activa/Desactiva la salida al fichero de traza
 C.- Presenta los parámetros de configuración de la máquina
     El parámetro esp_dirección puede ser una etiqueta o una dirección
     (decimal o hexadecimal)
```

PROBLEMA #1: EJECUCIÓN DEL PROGRAMA

- Para ejecutar el programa podemos escribir ***e*** pero como se trata de un programa corto y queremos ver la ejecución entera, es mejor ir paso a paso.
- Por ello, usamos ***t*** *seguido de un número: 1 .*
- *Esto quiere decir que ejecutaremos exactamente 1 instrucción: la instrucción addu r4, r0, 2*
- Como se puede apreciar, pasamos a la siguiente instrucción y R4 = 2

```
88110> t 1
 PC=4          mulu     r04,r04,7        Tot. Inst: 1   ¡¡ Ciclo : 14
 FL=1 FE=1 FC=0 FV=0 FR=0
 R01 = 00000000 h R02 = 00000000 h R03 = 00000000 h R04 = 00000002 h
 R05 = 00000000 h R06 = 00000000 h R07 = 00000000 h R08 = 00000000 h
 R09 = 00000000 h R10 = 00000000 h R11 = 00000000 h R12 = 00000000 h
 R13 = 00000000 h R14 = 00000000 h R15 = 00000000 h R16 = 00000000 h
 R17 = 00000000 h R18 = 00000000 h R19 = 00000000 h R20 = 00000000 h
 R21 = 00000000 h R22 = 00000000 h R23 = 00000000 h R24 = 00000000 h
 R25 = 00000000 h R26 = 00000000 h R27 = 00000000 h R28 = 00000000 h
 R29 = 00000000 h R30 = 00000000 h R31 = 00000000 h
```

PROBLEMA #1: EJECUCIÓN DEL PROGRAMA

- Lo mismo pasa en las siguientes.
- Al final, aparece instrucción incorrecta, pero en este caso realmente significa que no hay más instrucciones.
- Como podemos ver, R05 = 0xFFFFFFF9, es decir, -7. Nuestro código es correcto.

```
88110> t 1
 PC=4          mulu     r04,r04,7        Tot. Inst: 1   ¡¡ Ciclo : 14
 FL=1 FE=1 FC=0 FV=0 FR=0
 R01 = 00000000 h R02 = 00000000 h R03 = 00000000 h R04 = 00000002 h
 R05 = 00000000 h R06 = 00000000 h R07 = 00000000 h R08 = 00000000 h
 R09 = 00000000 h R10 = 00000000 h R11 = 00000000 h R12 = 00000000 h
 R13 = 00000000 h R14 = 00000000 h R15 = 00000000 h R16 = 00000000 h
 R17 = 00000000 h R18 = 00000000 h R19 = 00000000 h R20 = 00000000 h
 R21 = 00000000 h R22 = 00000000 h R23 = 00000000 h R24 = 00000000 h
 R25 = 00000000 h R26 = 00000000 h R27 = 00000000 h R28 = 00000000 h
 R29 = 00000000 h R30 = 00000000 h R31 = 00000000 h
88110> t 1
 PC=8          divs     r05,r04,65534    Tot. Inst: 2   ¡¡ Ciclo : 29
 FL=1 FE=1 FC=0 FV=0 FR=0
 R01 = 00000000 h R02 = 00000000 h R03 = 00000000 h R04 = 0000000E h
 R05 = 00000000 h R06 = 00000000 h R07 = 00000000 h R08 = 00000000 h
 R09 = 00000000 h R10 = 00000000 h R11 = 00000000 h R12 = 00000000 h
 R13 = 00000000 h R14 = 00000000 h R15 = 00000000 h R16 = 00000000 h
 R17 = 00000000 h R18 = 00000000 h R19 = 00000000 h R20 = 00000000 h
 R21 = 00000000 h R22 = 00000000 h R23 = 00000000 h R24 = 00000000 h
 R25 = 00000000 h R26 = 00000000 h R27 = 00000000 h R28 = 00000000 h
 R29 = 00000000 h R30 = 00000000 h R31 = 00000000 h
88110> t 1
 PC=12         instrucción incorrecta    Tot. Inst: 3   ¡¡ Ciclo : 54
 FL=1 FE=1 FC=0 FV=0 FR=0
 R01 = 00000000 h R02 = 00000000 h R03 = 00000000 h R04 = 0000000E h
 R05 = FFFFFFF9 h R06 = 00000000 h R07 = 00000000 h R08 = 00000000 h
 R09 = 00000000 h R10 = 00000000 h R11 = 00000000 h R12 = 00000000 h
 R13 = 00000000 h R14 = 00000000 h R15 = 00000000 h R16 = 00000000 h
 R17 = 00000000 h R18 = 00000000 h R19 = 00000000 h R20 = 00000000 h
 R21 = 00000000 h R22 = 00000000 h R23 = 00000000 h R24 = 00000000 h
 R25 = 00000000 h R26 = 00000000 h R27 = 00000000 h R28 = 00000000 h
 R29 = 00000000 h R30 = 00000000 h R31 = 00000000 h
88110>
```

INSTRUCCIONES LÓGICAS

- Existen instrucciones que nos permiten operar a nivel lógico, es decir, simulan una puerta lógica:
  - and
  - or
  - xor
- Existen más instrucciones, pero estas son las más usadas.
- **Problema #2**

  Queremos poner en r1, r2, y r3 los números 999, 1000, y 1001 respectivamente.
  Luego, queremos dejar r1 y r3 a 0 usando las instrucciones AND y XOR, respectivamente.

- Solución:

; Primera Parte: Queremos poner en r1, r2, y r3 los números 999, 1000, y 1001 respectivamente.

or r1, r0, 999

or r2, r0, 0x3e8 ; Se puede hacer en hexadecimal

or r3, r0, 1001

; Segunda Parte: queremos dejar r1 y r3 a 0 usando las instrucciones AND y XOR, respectivamente.

and r1, r1, r0

xor r3, r3, r3

- Resultado:
  R02 = 000003E8 h (0x 000003E8 = 1000)

INSTRUCCIONES DE CONTROL DE FLUJO Y DE BIT

- Existen instrucciones que nos permiten controlar el flujo del programa, de la misma forma que haríamos con un **if**.
- Es decir, nos permiten controlar qué porción del código se ejecuta.
- Las más communes en 88110 son:
  - br (Branch) y cmp (Compare)
    - bb0
    - bb1
- Las instrucciones bb0 y bb1 realizan una comparación a nivel de bit.
- Asimismo, existen instrucciones que nos permiten cambiar bits específicos:
  - set, clr, ext, mak, rot, …
  - Forma General:
    instrucción regFinal, reg1, numero_bit
- Para usar una etiqueta, basta con poner **nombre_etiqueta:** incluyendo los ":"

COMPILACIÓN Y PROBLEMA #3

- Para compilar con una etiqueta como punto de entrada, se pueden usar los scripts proporcionados o usar el siguiente comando:

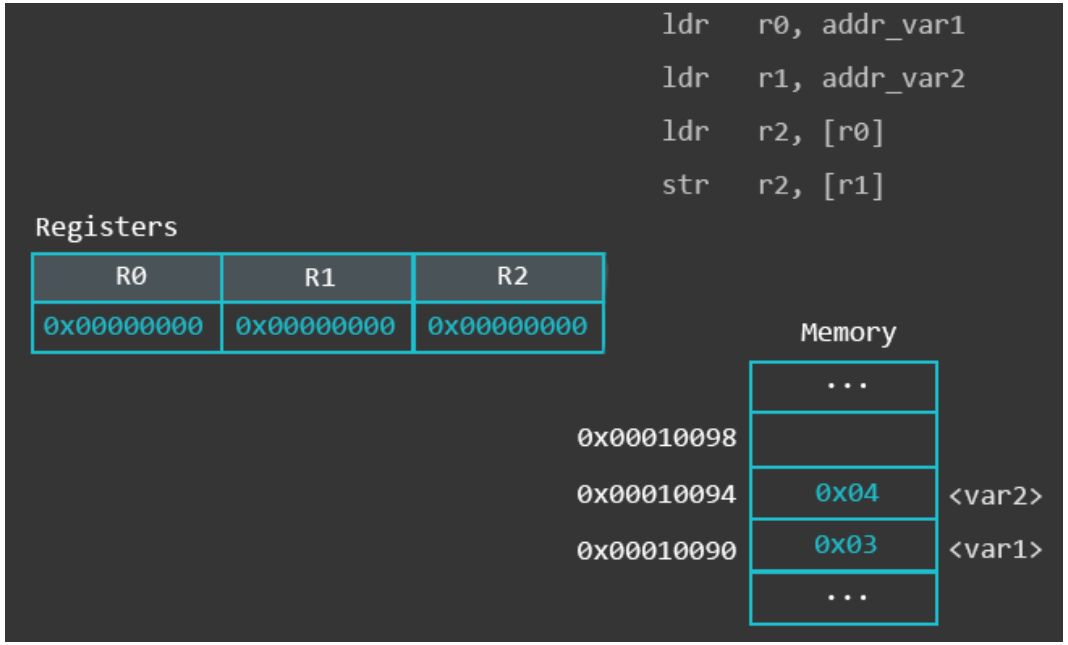
  88110e -e **nombre_de_etiqueta** -ml –o **fichero_destino.bin fichero_ensamblador.ens**

- Esto permitirá ejecutar el código empezando en la etiqueta especificada. Esto es muy útil para probar cosas específicas.
- Problema #3:
  - Queremos sumar dos números y negar el último bit del resultado. Almacenamos en r5, sumándole 7.
- La solución al problema está en Github

https://github.com/M-T3K/UPM/tree/master/Clases%20de%20Apoyo/Ensamblador-18Diciembre2021/

INSTRUCCIONES DE MEMORIA Y PSEUDOINSTRUCCIONES

- Existen instrucciones que nos permiten operar con la memoria:
  - ld (load / cargar)
  - st (store / almacenar)



- También existen las siguientes pseudoinstrucciones:
  - org (organizar)
    Sirve para organizar el Código y las regiones correspondientes de memoria, porque **el propio código ocupa espacio en memoria** y por lo tanto **podemos cambiarlo si modificamos la memoria.**
  - data (datos)
    Sirve para reservar en memoria el espacio correspondiente a una serie de datos especificados.
  - res (reservar)
    Debe ser múltiplo de la longitud de palabra (4 en este caso) y se inicializa a un valor indefinido. Se usa para un espacio en el que quieres guardar cosas.

https://azeria-labs.com/memory-instructions-load-and-store-part-4/

MACROS

- Para hacernos la vida más fácil, de la misma forma que hay etiquetas existe la posibilidad de designar **macros**.
- Las macros nos permiten reútilizar Código de forma sencilla y se asemejan a funciones o métodos de un lenguaje de programación.
- Algunas útiles son las siguientes:
  - LEA (LOAD EFFECTIVE ADDRESS) permite cargar la dirección efectiva de una etiqueta y cosas similares que se pasan por dirección.
  - LOAD hace un LEA y posteriormente carga en un registro lo que haya.
  - DBNZ (Decrement branch if not zero) decrementa un Contador y hace un branch si el Contador no ha llegado a 0.

Implementación		

LEA: MACRO(ra, eti)
or ra, r0, low(eti)
or.u ra, ra, high(eti)
ENDMACRO
	LOAD: MACRO(ra, eti)
	LEA(ra, eti)
	ld ra, ra, 0
	ENDMACRO
		DBNZ: MACRO(ra, eti)
		sub ra, ra, 1
		cmp r2, ra, 0
		bb0 eq, r2, eti
		ENDMACRO

PROBLEMA #4

- Se proporciona un texto de una longitude máxima de 50 caracteres en la dirección 0.
- Se pretende invertir este texto, es decir, darle la Vuelta, e insertarlo en una sección reservada inmediatamente después del texto original.
- Además, en la dirección 1000 existirá una variable global **numchars.** Por cada letra del texto original, se deberá sumar uno a esta variable global.
- Finalmente, se devolvera en r29 el primer carácter leído.
  - Está prohibido almacenarlo simplemente en r29 al principio de la ejecución.
- Para ello es importante tener en cuenta los siguientes detalles:
  - Todas las letras del texto serán del tipo **char**, por lo que ocuparán **un solo byte sin signo**. Es decir, se deberá usar la extensión **.bu** o **.b** siempre que se esté trabajando con este tipo de datos y resulte posible orientar a byte.
  - No se podrán realizar subrutinas.
  - Se debe recordar que la pseudoinstrucción de reserva **res** debe estar orientada a palabra. Por lo tanto, no se puede usar para reservar 50 bytes.
  - Se recomienda usar la macro LEA para cargar las direcciones de etiquetas.

PROBLEMA #4

Situación Final:

1ª Imagen variable numchars y 2ª resultado de la inversión del texto.

```
88110> v 1000 10

      992                                       33000000    00000000
     1008       00000000    00000000    00000000    00000000
     1024       00000000    00000000    00000000    00000000
```

Situación Inicial:

```
  0    74657874    6F746578    746F7465    78746F74
 16    6578746F    74657874    6F746578    746F7465
 32    78746F74    6578746F    74657874    6F746578
 48    746F0000    00000000    00000000    00000000
 64    00000000    00000000    00000000    00000000
 80    00000000    00000000    00000000    00000000
 96    00000000    00000000    00000000    00000000
112    00000000    00000000    00000000    00000000
128    00000000    00000000    00000000    00000000
144    00000000    00000000    00000000    00000000
```

```
  0    74657874    6F746578    746F7465    78746F74
 16    6578746F    74657874    6F746578    746F7465
 32    78746F74    6578746F    74657874    6F746578
 48    746F0000    006F7478    65746F74    7865746F
 64    74786574    6F747865    746F7478    65746F74
 80    7865746F    74786574    6F747865    746F7478
 96    65746F74    78657400    00000000    00000000
112    00000000    00000000    00000000    00000000
128    00000000    00000000    00000000    00000000
```

Comprobado con el mandato **v 0 100** para ver los contenidos de las direcciones 0 a 100

Estrictamente hablando, el Código hace lo que se ha pedido, pero…

¿podéis mejorarlo?

Ideas:

Está *hardcoded*, es decir, se pone un número como longitud del texto. Se podría medir. Mirad como lo hace C. Esto implica que añade 0s al principio si no se mide la distancia correctamente en bytes.

MARCO DE PILA

- Para usar subrutinas, que es lo más parecido a funciones de lo que dispone este ensamblador, tenemos que utilizar la pila para el paso de parámetros.
- Esto implica, que tenemos que saber construirla y destruirla **continuamente**.
- Debemos estar cómodos con este proceso, pero para facilitarlo se proporcionan dos macros:

Implementación	

PUSH: MACRO(ra)
subu r30, r30, 4
st ra, r30, 0
ENDMACRO
	POP: MACRO(ra)
	ld ra, r30, 0
	addu r30, r30, 4
	ENDMACRO

- En su forma más básica, la pila no es más que una sección de memoria sobre la que operamos para almacenar datos.
- Partimos de una dirección base lejana, como la 65000 (la última dirección posible).
- Cada vez que añadimos un dato, restamos una palabra (4 bytes) a la dirección y luego hacemos un store.
- Cada vez que extraemos un dato, lo cargamos mediante load y incrementamos en 4 bytes la dirección de la pila.

MARCO DE PILA

- Partiendo de que establecemos la pila en la dirección 0x65000. Pongamos el siguiente ejemplo*:
  - PUSH(12); Restamos 4 a la dirección de pila y almacenamos.
    PUSH(6); Restamos 4 a la dirección de pila y almacenamos
    PUSH(3); Restamos 4 a la dirección de pila y almacenamos
    PUSH(6); Restamos 4 a la dirección de pila y almacenamos
    PUSH(12); Restamos 4 a la dirección de pila y almacenamos
- La pila del programa quedaría como en la derecha.
- *En realidad no podemos pasar los datos directamente, tendríamos que **meterlos en un registro** antes del PUSH tal que así:

```
or r5,r0,12
PUSH(r5)
```

	Memoria
64FE8	???????
64FEC	12
64FF0	6
64FF4	3
64FF8	6
64FFC	12
65000	FIN DE MEMORIA

Definición de MACROs

```
PUSH:   MACRO(ra)
   subu r30, r30, 4
   st ra, r30, 0
ENDMACRO
```

```
POP:    MACRO(ra)
   ld ra, r30, 0
   addu r30, r30, 4
ENDMACRO
```

MARCO DE PILA

- Partiendo de que establecemos la pila en la dirección 0x65000. Pongamos el siguiente ejemplo:
  - PUSH(12); Restamos 4 a la dirección de pila y almacenamos.
    PUSH(6); Restamos 4 a la dirección de pila y almacenamos
    PUSH(3); Restamos 4 a la dirección de pila y almacenamos
    PUSH(6); Restamos 4 a la dirección de pila y almacenamos
    POP(r5); almacenamos
- ¿Cómo quedaría el programa? Y ¿los registros?
  - Se comprueba en orden secuencial
    - Primero los cuatro PUSH
    - Finalmente, el POP
  - Vayamos paso a paso…

	Memoria
64FE8	???????
64FEC	???????
64FF0	???????
64FF4	???????
64FF8	???????
64FFC	???????
65000	FIN DE MEMORIA

MARCO DE PILA

- Partiendo de que establecemos la pila en la dirección 0x65000. Pongamos el siguiente ejemplo:
  - **PUSH(12)**; Restamos 4 a la dirección de pila y almacenamos.
    PUSH(6); Restamos 4 a la dirección de pila y almacenamos
    PUSH(3); Restamos 4 a la dirección de pila y almacenamos
    PUSH(6); Restamos 4 a la dirección de pila y almacenamos
    POP(r5); almacenamos
- El puntero de pila está en la dirección 0x65000. Cuando se realiza el primer PUSH, se disminuye en 4 bytes esta dirección (hasta ser 0x64FFC) y se almacena con **st** el número 12.

	Memoria
64FE8	???????
64FEC	???????
64FF0	???????
64FF4	???????
64FF8	???????
64FFC	12
65000	FIN DE MEMORIA

MARCO DE PILA

- Partiendo de que establecemos la pila en la dirección 0x65000. Pongamos el siguiente ejemplo:
  - **PUSH(12)**; Restamos 4 a la dirección de pila y almacenamos.
    PUSH(6); Restamos 4 a la dirección de pila y almacenamos
    PUSH(3); Restamos 4 a la dirección de pila y almacenamos
    PUSH(6); Restamos 4 a la dirección de pila y almacenamos
    POP(r5); almacenamos
- El puntero de pila está en la dirección 0x64FFC. Cuando se realiza el segundo PUSH, se disminuye en 4 bytes esta dirección (hasta ser 0x64FF8) y se almacena con **st** el número 6.

	Memoria
64FE8	???????
64FEC	???????
64FF0	???????
64FF4	???????
64FF8	6
64FFC	12
65000	FIN DE MEMORIA

MARCO DE PILA

- Partiendo de que establecemos la pila en la dirección 0x65000. Pongamos el siguiente ejemplo:
  - **PUSH(12)**; Restamos 4 a la dirección de pila y almacenamos.
    PUSH(6); Restamos 4 a la dirección de pila y almacenamos
    PUSH(3); Restamos 4 a la dirección de pila y almacenamos
    PUSH(6); Restamos 4 a la dirección de pila y almacenamos
    POP(r5); almacenamos
- Se continúa el proceso hasta llegar al POP. Una vez termina el último PUSH, el puntero de la dirección de pila es la 64FF0.
- **¿Qué pasará con el POP?**

	Memoria
64FE8	???????
64FEC	???????
64FF0	6
64FF4	3
64FF8	6
64FFC	12
65000	FIN DE MEMORIA

MARCO DE PILA

- Partiendo de que establecemos la pila en la dirección 0x65000. Pongamos el siguiente ejemplo:
  - **PUSH(12)**; Restamos 4 a la dirección de pila y almacenamos.
    PUSH(6); Restamos 4 a la dirección de pila y almacenamos
    PUSH(3); Restamos 4 a la dirección de pila y almacenamos
    PUSH(6); Restamos 4 a la dirección de pila y almacenamos
    POP(r5); almacenamos
- La dirección actual de pila es la 0x64FF0.
- De acuerdo a su definición, el POP primero cargará el valor de la dirección de pila en el registro especificado. R5 = 6
- Luego, incrementará el puntero de pila en 4 bytes, por lo que la pila apuntará a 0x64FF4.
- Sin embargo, el dato que había en la pila sigue estando ahí hasta que otra parte de nuestro código lo sobrescriba.

	Memoria
64FE8	???????
64FEC	???????
64FF0	6
64FF4	3
64FF8	6
64FFC	12
65000	FIN DE MEMORIA

SUBRUTINAS

- Son parecidas a una etiqueta, pero para llamarlas, se utiliza la instrucción **bsr**. Esta permite retornar y conservar la dirección de retorno.
- Sin esto, **no sabríamos donde estábamos ejecutando antes de llamar a la subrutina.**
- Debemos construir el marco de pila individualmente para cada una, con el objetivo de **salvaguardar los registros importantes.**
- Esto consigue crear una "pila recursiva" para el que, dentro de la pila, cada subrutina tiene su propio espacio para operar sin el riesgo de tocar datos que no le pertenecen.
- A la derecha se muestran los mecanismos de construcción y destrucción de pila, que deberían ser lo primero y último en hacerse respectivamente en una nueva subrutina.

MACROs

```
cPILA:    MACRO()
          PUSH(r1)
          PUSH(r31)
          or r31, r30, r30
ENDMACRO
```

```
dPILA:    MACRO()
          or r30, r31, r31
          POP(r31)
          POP(r1)
ENDMACRO
```

PROBLEMA #5: AHORA CON SUBRUTINAS

- Partimos de la base establecida en el anterior problema, con algunas diferencias.
- Estamos obligados a llamar a una subrutina para incrementar el valor de la variable **numchars**.
- Todo partirá de un programa principal que llamará a la subrutina "invertir" pasándole como parámetros la dirección de los dos buffers y de la variable **numchars**, así como un valor inicial que le sumaremos al primer caracter leído antes de retornarlo.
- Resumiendo, debemos:
- Usar un Programa Principal (ppal) para preparar argumentos y ejecutar la subrutina **invertir**, que devolverá en r29 el primer caracter leído, con los siguientes argumentos:
  - Dirección del Buffer original de texto a invertir
  - Dirección del Buffer final de texto invertido.
  - Dirección de la variable **numchars**.
  - Valor que debemos sumar al primer caracter leido antes de retornarlo.
- El incremento de la variable **numchars** deberá usar una subrutina.

Solución: https://github.com/M-T3K/UPM/tree/master/Clases%20de%20Apoyo/Ensamblador-18Diciembre2021/

PROBLEMA #5: PISTAS

- Lo primero a la hora de usar la pila, es inicializarla a un valor razonable, como 65000. En los ejemplos anteriores, se ponía 65000 como hexadecimal, pero esto no es necesario: un número 65000 decimal es más que suficiente. Para ello, debemos hacer esto:

  LEA(r30, 65000); en Hex esto sería 0xFDE8
- El estado de la pila, tras la primera llamada a la subrutina **nChars** con la función de incrementar el valor de la variable **numchars** es el que se ve a la derecha, donde:
  - & significa "Dirección de"
  - * significa "Valor de"
  - Los registros implican el valor del registro **en el momento del PUSH** y pueden **no ser iguales** a pesar de representar al mismo registro.

PILA tras la primera llamada a **nChars**	
0xFDC4	R31
0xFDC8	R1
0xFDCC	&numchars
0xFDD0	R31
0xFDD4	R1
0xFDD8	*valsuma
0xFDDC	&numchars
0xFDE0	&inverso
0xFDE4	&texto
0xFDE8	FIN

PROBLEMA #5: PISTAS

- Además, se ponen en naranja las partes pertinentes a la subrutina **nChars**.
- El almacenamiento de R1 y r31 realizado por la construcción de pila al inicio de cada subrutina.
- Al construir la pila y añadir los dos registros, realmente se está aumentando el tamaño de la pila en 8, por lo que tendréis que tenerlo en cuenta al cargar los elementos.
  - Es decir, tendréis que cargarlos a partir de la dirección r31 + 8.
- Como se puede apreciar, la pila funciona de forma recursiva. Esto es importante para poder llamar a subrutinas dentro de otras subrutinas.
- Tened en cuenta que cambiar un registro en una subrutina sin salvaguardarlo implica que ese registro y su valor tambien habrá cambiado en las demás subrutinas. **IMPORTANTE.**

PILA tras la primera llamada a **nChars**	
0xFDC4	R31
0xFDC8	R1
0xFDCC	&numchars
0xFDD0	R31
0xFDD4	R1
0xFDD8	*valsuma
0xFDDC	&numchars
0xFDE0	&inverso
0xFDE4	&texto
0xFDE8	FIN

ENSAMBLADOR – TESTING Y ENDIANNESS

- El testeo es vital, y más en ensamblador: un error pequeño puede ser un quebradero de cabeza.
- Se lleva acabo con un programa principal de pruebas y con datos previamente especificados con data, res y org.
- Recordad el tipo de Endianness del emulador.
- Ejemplo:
  Para un caso de prueba, os proporcionan la siguiente matriz:

$$\begin{bmatrix} 0x0 & 0x12345678 \\ 0x87654321 & 0x1 \end{bmatrix}$$

- Cuando queremos ponerla en el código, no podemos ponerla tal cual. Tenemos que tener en cuenta el modo de endian y cambiarlo adecuadamente.
- Por lo tanto, la matriz

$$\begin{bmatrix} 0x0 & 0x12345678 \\ 0x87654321 & 0x1 \end{bmatrix}$$

- Realmente debería ser introducida de la siguiente forma:

$$\begin{bmatrix} 0x00000000 & 0x78563412 \\ 0x21436587 & 0x01000000 \end{bmatrix}$$

ENSAMBLADOR – TESTING Y ENDIANNESS

- La Matriz

$$\begin{bmatrix} 0x00000000 & 0x78563412 \\ 0x21436587 & 0x01000000 \end{bmatrix}$$

Se puede introducir con las siguientes pseudoinstrucciones:

```
data    0x00000000 , 0x78563412
data    0x21436587, 0x01000000
```

- Si usáis la extensión M88K mencionada al principio de la charla, cambiar el modo de endian es muy sencillo: escribís un número hexadecimal y con la combinación "alt + e" aparecerá un desplegable. La primera opción lo hará por vosotros.

- Es decir, podéis escribir las cosas como las entendáis, y luego, cambiar el endian de forma sencilla
- Lo mismo pasa si os dan información salida de la pantalla del emulador: para poder introducirla en el código tendréis que cambiar el modo de endian.
- La forma sencilla de hacerlo en un número hexadecimal es la siguiente:

0x 11 00 10 00
0x 00 10 00 11

Es decir, los pares de en medio se invierten, y los de los extremos también.