ANTONIO JOSÉ VILLENA GODOY
RAFAEL ASENJO PLAZA
FRANCISCO J. CORBERA PEÑA

PRÁCTICAS DE ENSAMBLADOR BASADAS EN RASPBERRY PI

Departamento de Arquitectura de Computadores

UNIVERSIDAD DE MÁLAGA / MANUALES

Acrónimos

AAPCS	ARM Architecture Procedure Call Standard
ARM	Advanced RISC Machines
CPSR	Current Program Status Register
CPU	Central Processing Unit
CHI	system timer Counter HIgher
CLO	system timer Counter LOwer
CS	system timer Control/Status
E/S	Entrada/Salida
ETSII	Escuela Técnica Superior de Ingeniería Informática
FIQ	Fast Interrupt reQuest
GNU	GNU is Not Unix
GCC	GNU C Compiler
GDB	GNU DeBugger
GPAFEN	GPIO Pin Async. Falling Edge Detect
GPAREN	GPIO Pin Async. Rising Edge Detect
GPEDS	GPIO Pin Event Detect Status
GPFEN	GPIO Pin Falling Edge Detect Enable
GPHEN	GPIO Pin High Detect Enable
GPIO	General-Purpose Input/Output

GPL	General Public License
GPLEN	GPIO Pin Low Detect Enable
GPLEV	GPIO Pin LEVel
GPPUD	GPIO Pin High Detect Enable
GPPUDCLK	GPIO Pin High Detect Enable CLocK
GPREN	GPIO Pin Rising Edge Detect Enable
GPU	Graphics Processing Unit
IRQ	Interrupt ReQuest
LED	Light Emitting Diode
LR	Link Register
PFC	Proyecto Fin de Carrera
PC	Personal Computer
RAM	Random-Access Memory
RISC	Reduced Instruction Set Computer
ROM	Read-Only Memory
RTI	Rutina de Tratamiento de Interrupción
SoC	System on a Chip
SP	Stack Pointer
SPSR	Saved Program Status Register
UMA	Universidad de MÁlaga
VFP	Vector Floating-Point
abt	ABorT mode
mon	secure MONitor mode
svc	Supervisor mode (antiguamente SuperVisor Calls)
und	UNDefined mode

Prólogo

El minicomputador Raspberry Pi es una placa del tamaño de una tarjeta de crédito y un precio de sólo 30€. El objetivo principal de sus creadores, la Fundación Raspberry Pi, era promover la enseñanza de conceptos básicos de informática en los colegios e institutos. Sin embargo, ha terminado convirtiéndose también en un pequeño computador de bajo coste que se destina a muy diversos usos: servidor multimedia conectado al televisor, estación base para domótica en el hogar, estaciones meteorológicas, servidor de discos en red para copias de seguridad, o como un simple ordenador que puede ejecutar aplicaciones de internet, juegos, ofimática, etc. Esto ha llegado a ser así gracias a un vertiginoso crecimiento de la comunidad de desarrolladores para Raspberry Pi, y que estos han explorado casi todas las posibilidades para sacar el máximo partido de este ordenador de 30€. Esa gran funcionalidad y el bajo coste constituyen el principal atractivo de esta plataforma para los estudiantes. Sin embargo, para los docentes del Dept. de Arquitectura de Computadores, la Raspberry Pi ofrece una excusa perfecta para hacer más amenos y atractivos conceptos a veces complejos, y a veces también áridos, de asignaturas del área.

Este trabajo se enmarca dentro del Proyecto de Innovación Educativa PIE13-082, "Motivando al alumno de ingeniería mediante la plataforma Raspberry Pi" cuyo principal objetivo es aumentar el grado de motivación del alumno que cursa asignaturas impartidas por el Departamento de Arquitectura de Computadores. La estrategia propuesta se apoya en el hecho de que muchos alumnos de Ingeniería perciben que las asignaturas de la carrera están alejadas de su realidad cotidiana, y que por ello, pierden cierto atractivo. Sin embargo, bastantes de estos alumnos han comprado o piensan comprar un minicomputador Raspberry Pi que se caracteriza por proporcionar una gran funcionalidad, gracias a estar basado en un procesador y Sistema Operativo de referencia en los dispositivos móviles. En este proyecto proponemos aprovechar el interés que los alumnos ya demuestran por la plataforma Raspberry Pi, para ponerlo a trabajar en pro del siguiente objetivo docente: facilitar el estudio de conceptos y técnicas impartidas en varias asignaturas del Departamento. Cuatro de estas asignaturas son:

- Tecnología de Computadores: Asignatura obligatoria del módulo de Formación

Común de las titulaciones de Grado en Ingeniería Informática, Grado en Ingeniería de Computadores y Grado en Ingeniería del Software. Es una asignatura que se imparte en el primer curso.

- Estructura de Computadores: Asignatura obligatoria del módulo de Formación Común de las titulaciones de Grado en Ingeniería Informática, Grado en Ingeniería de Computadores y Grado en Ingeniería del Software. Es una asignatura que se imparte en el segundo curso.

- Sistemas Operativos: Asignatura obligatoria del módulo de Formación Común de las titulaciones de Grado en Ingeniería Informática, Grado en Ingeniería de Computadores y Grado en Ingeniería del Software. Se imparte en segundo curso.

- Diseño de Sistemas Operativos: Asignatura obligatoria del módulo de Tecnologías Específicas del Grado de Ingeniería de Computadores. Se imparte en tercer curso.

En esas cuatro asignaturas, uno de los conceptos más básicos es el de gestión de interrupciones a bajo nivel. En particular, en Estructura de Computadores, esos conceptos se ilustraban en el pasado mediante prácticas en PCs con MSDOS y programación en ensamblador, pero el uso de ese sistema operativo ya no tiene ningún atractivo y además crea problemas de seguridad en los laboratorios del departamento. Sin embargo, la plataforma Raspberry Pi se convierte en una herramienta adecuada para trabajar a nivel de sistema, es económica y ya disponemos de unidades suficientes para usarlas en los laboratorios (30 equipos para ser exactos).

El principal objetivo de este trabajo es la creación de un conjunto de prácticas enfocadas al aprendizaje de la programación en ensamblador, en concreto del ARMv6 que es el procesador de la plataforma que se va a utilizar para el desarrollo de las prácticas, así como al manejo a bajo nivel de las interrupciones y la entrada/salida en dicho procesador. El aprendizaje del lenguaje ensamblador del procesador ARM usado en la Raspberry Pi se puede completar leyendo la documentación disponible en [1] y haciendo los tutoriales de [2]. Para la parte más centrada en el hardware también se puede consultar la amplia documentación disponible en internet, como por ejemplo los tutoriales disponibles en [3] y la descripción los modos de operación de los periféricos conectados al procesador ARM [4].

La presente memoria está dividida cinco capítulos y cuatro apéndices. De los 5 capítulos, el primero es introductorio. Los dos siguientes se centran en la programación de ejecutables en Linux, tratando las estructuras de control en el capítulo 2 y las subrutinas (funciones) en el capítulo 3. Los dos últimos capítulos muestran la programación en Bare Metal, explicando el subsistema de entrada/salida (puertos de entrada/salida y temporizadores) de la plataforma Raspberry Pi y su manejo a

bajo nivel en el capítulo 4 y las interrupciones en el capítulo 5. En los apéndices hemos añadido aspectos laterales pero de suficiente relevancia como para ser considerados en la memoria, como el apendice A que explica el funcionamiento de la macro ADDEXC, el apéndice B que muestra todos los detalles de la placa auxiliar, el apéndice C que nos enseña a agilizar la carga de programas Bare Metal y por último tenemos el apéndice D, que profundiza en aspectos del GPIO como las resistencias programables.

Capítulo 1

Introducción al ensamblador

Contenido

Objetivo: En esta sesión vamos a conocer el entorno de trabajo. Veremos qué aspecto tiene un programa en ensamblador, veremos cómo funcionan los tres programas que vamos a utilizar: el ensamblador, el *enlazador (linker)* y el *depurador (debugger)*. Del *debugger* sólo mostraremos unos pocos comandos, que ampliaremos en las próximas sesiones. También veremos la representación de los números naturales y de los enteros, y el funcionamiento de algunas de las instrucciones del ARM. Se repasarán también los conceptos de registros, flags e instrucciones para la manipulación de bits.

1.1. Lectura previa

1.1.1. Características generales de la arquitectura ARM

ARM es una arquitectura RISC (Reduced Instruction Set Computer=Ordenador con Conjunto Reducido de Instrucciones) de 32 bits, salvo la versión del core ARMv8-A que es mixta 32/64 bits (bus de 32 bits con registros de 64 bits). Se trata de una arquitectura licenciable, quiere decir que la empresa desarrolladora ARM Holdings diseña la arquitectura, pero son otras compañías las que fabrican y venden los chips, llevándose ARM Holdings un pequeño porcentaje por la licencia.

El chip en concreto que lleva la Raspberry Pi es el BCM2835, se trata de un SoC (System on a Chip=Sistema en un sólo chip) que contiene además de la CPU otros elementos como un núcleo GPU (hardware acelerado OpenGL ES/OpenVG/Open EGL/OpenMAX y decodificación H.264 por hardware) y un núcleo DSP (Digital signal processing=Procesamiento digital de señales) que es un procesador más pequeño y simple que el principal, pero especializado en el procesado y representación de señales analógicas. La CPU en cuestión es la ARM1176JZF-S, un chip de la familia ARM11 que usa la arquitectura ARMv6k.

Familia	**Arquitectura**	**Bits**	**Ejemplos de dispositivos**
ARM1	ARMv1	32/26	Segundo procesador BBC Micro
ARM2, ARM3, Amber	ARMv2	32/26	Acorn Archimedes
ARM6, ARM7	ARMv3	32	Apple Newton Serie 100
ARM8, StrongARM	ARMv4	32	Apple Newton serie 2x00
ARM7TDMI, ARM9TDMI	ARMv4T	32	Game Boy Advance
ARM7EJ, ARM9E, ARM10E, XScale	ARMv5	32	Samsung Omnia, Blackberry 8700
ARM11	ARMv6	32	iPhone 3G, Raspberry Pi
Cortex-M0/M0+/M1	ARMv6-M	32	
Cortex-M3/M4	ARMv7-M ARMv7E-M	32	Texas Instruments Stellaris
Cortex-R4/R5/R7	ARMv7-R	32	Texas Instruments TMS570
Cortex-A5/A7/A8/A9 A12/15/17, Apple A6	ARMv7-A	32	Apple iPad
Cortex-A53/A57, X-Gene, Apple A7	ARMv8-A	64/32	Apple iPhone 5S

Tabla 1.1: Lista de familias y arquitecturas ARM

Las extensiones de la arquitectura ARMv6k frente a la básica ARMv6 son míni-

mas por lo que a efectos prácticos trabajaremos con la arquitectura ARMv6.

Registros

La arquitectura ARMv6 presenta un conjunto de 17 registros (16 principales más uno de estado) de 32 bits cada uno.

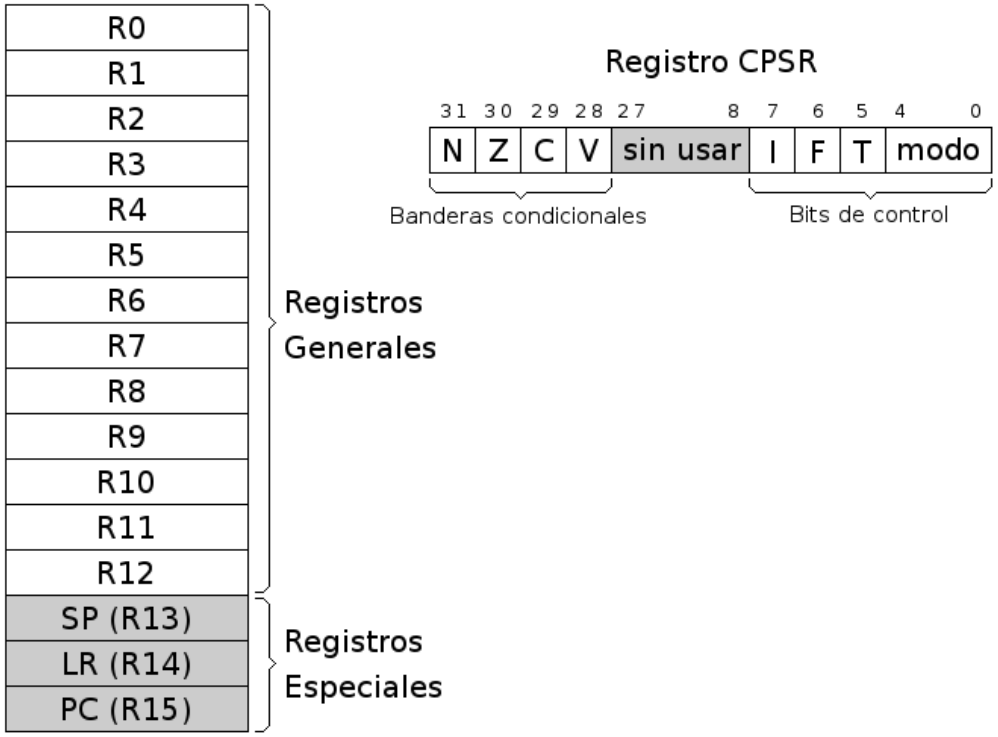

Figura 1.1: Registros de la arquitectura ARM

Registros Generales. Su función es el almacenamiento temporal de datos. Son los 13 registros que van R0 hasta R12.

Registros Especiales. Son los últimos 3 registros principales: R13, R14 y R15. Como son de propósito especial, tienen nombres alternativos.

- **SP**/R13. Stack Pointer ó Puntero de Pila. Sirve como puntero para almacenar variables locales y registros en llamadas a funciones.
- **LR**/R14. Link Register ó Registro de Enlace. Almacena la dirección de retorno cuando una instrucción BL ó BLX ejecuta una llamada a una rutina.

- **PC**/R15. Program Counter ó Contador de Programa. Es un registro que indica la posición donde está el procesador en su secuencia de instrucciones. Se incrementa de 4 en 4 cada vez que se ejecuta una instrucción, salvo que ésta provoque un salto.

Registro CPSR. Almacena las banderas condicionales y los bits de control. Los bits de control definen la habilitación de interrupciones normales (I), interrupciones rápidas (F), modo Thumb [1] (T) y el modo de operación de la CPU. Existen hasta 8 modos de operación, pero por ahora desde nuestra aplicación sólo vamos a trabajar en uno de ellos, el *Modo Usuario*. Los demás son modos privilegiados usados exclusivamente por el sistema operativo.

Desde el *Modo Usuario* sólo podemos acceder a las banderas condicionales, que contienen información sobre el estado de la última operación realizada por la ALU. A diferencia de otras arquitecturas en ARMv6 podemos elegir si queremos que una instrucción actualice o no las banderas condicionales, poniendo una "s" detrás del nemotécnico [2]. Existen 4 banderas y son las siguientes:

- **N**. Se activa cuando el resultado es negativo.
- **Z**. Se activa cuando el resultado es cero o una comparación es cierta.
- **C**. Indica acarreo en las operaciones aritméticas.
- **V**. Desbordamiento aritmético.

Esquema de almacenamiento

El procesador es *Bi-Endian*, quiere decir que es configurable entre *Big Endian* y *Little Endian*. Aunque nuestro sistema operativo nos lo limita a *Little Endian*.

Por tanto la regla que sigue es "el byte menos significativo ocupa la posición más baja". Cuando escribimos un dato en una posición de memoria, dependiendo de si es byte, half word o word,... se ubica en memoria según el esquema de la figura 1.2. La dirección de un dato es la de su byte menos significativo. La memoria siempre se referencia a nivel de byte, es decir si decimos la posición N nos estamos refiriendo al byte N-ésimo, aunque se escriba media palabra, una palabra,...

[1] Es un modo simplificado donde las instrucciones son de 16 bits en lugar de 32 y se acceden a menos registros (hasta r7), con la ventaja de que el código ocupa menos espacio.

[2] Es la forma de nombrar las instrucciones desde ensamblador, normalmente derivadas de una abreviatura del verbo en inglés. Por ejemplo la instrucción *MOV* viene de "move" (mover)

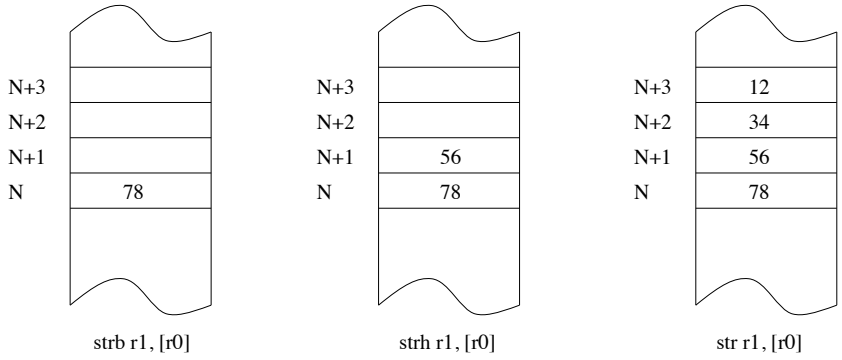

Figura 1.2: Ubicación de datos en memoria

1.1.2. El lenguaje ensamblador

El ensamblador es un lenguaje de bajo nivel que permite un control directo de la CPU y todos los elementos asociados. Cada línea de un programa ensamblador consta de una instrucción del procesador y la posición que ocupan los datos de esa instrucción.

Desarrollar programas en lenguaje ensamblador es un proceso laborioso. El procedimiento es similar al de cualquier lenguaje compilado. Un conjunto de instrucciones y/o datos forman un módulo fuente. Este módulo es la entrada del compilador, que chequea la sintaxis y lo traduce a código máquina formando un módulo objeto. Finalmente, un enlazador (montador ó *linker*) traduce todas las referencias relativas a direcciones absolutas y termina generando el ejecutable.

El ensamblador presenta una serie de ventajas e inconvenientes con respecto a otros lenguajes de más alto nivel. Al ser un lenguaje de bajo nivel, presenta como principal característica la flexibilidad y la posibilidad de acceso directo a nivel de registro. En contrapartida, programar en ensamblador es laborioso puesto que los programas contienen un número elevado de líneas y la corrección y depuración de éstos se hace difícil.

Generalmente, y dado que crear programas un poco extensos es laborioso, el ensamblador se utiliza como apoyo a otros lenguajes de alto nivel para 3 tipos de situaciones:

- Operaciones que se repitan un número elevado de veces.
- Cuando se requiera una gran velocidad de proceso.
- Para utilización y aprovechamiento de dispositivos y recursos del sistema.

1.1.3. El entorno

Los pasos habituales para hacer un programa (en cualquier lenguaje) son los siguientes: lo primero es escribir el programa en el lenguaje fuente mediante un editor de programas. El resultado es un fichero en un lenguaje que puede entender el usuario, pero no la máquina. Para traducirlo a lenguaje máquina hay que utilizar un programa traductor. Éste genera un fichero con la traducción de dicho programa, pero todavía no es un programa ejecutable. Un fichero ejecutable contiene el programa traducido más una serie de códigos que debe tener todo programa que vaya a ser ejecutado en una máquina determinada. Entre estos códigos comunes se encuentran las librerías del lenguaje. El encargado de unir el código del programa con el código de estas librerías es un programa llamado montador (*linker*) que genera el programa ejecutable (ver la figura 1.3)

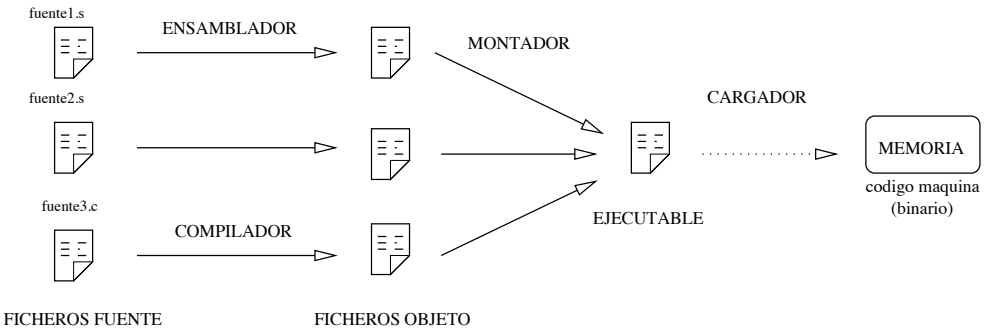

Figura 1.3: Entorno típico de programación

Durante el proceso de creación de un programa se suelen producir errores. Hay dos tipos de errores: los sintácticos o detectables en tiempo de traducción y los errores semánticos o detectables en tiempo de ejecución. Los errores sintácticos son, por ejemplo, escribir mal una instrucción o hacer una operación entre dos tipos de datos incompatibles. Estos errores son detectados por el traductor y se deben solucionar para poder generar un ejecutable.

Una vez que se tiene un programa sintácticamente correcto lo podemos ejecutar, pero ésto no implica que el programa sea correcto. Todas las instrucciones pueden ser correctas, pero se puede haber olvidado poner la condición de salida de un bucle (y que no termine nunca) o que sencillamente el programa no haga lo que queremos.

Estos errores sólo se pueden detectar en tiempo de ejecución. Para poder eliminarlos se utiliza un depurador de programas (*debugger*). El depurador nos permite ejecutar el programa instrucción a instrucción y ver todos los valores que se van a calcular, de manera que podemos encontrar los errores.

En el laboratorio utilizaremos el editor **nano** para crear y editar los módulos fuente de nuestros programas. El traductor (que en el caso de traducir de un len-

guaje ensamblador a lenguaje máquina recibe el nombre de ensamblador), el *linker* y el *debugger* son respectivamente GNU Assembler (**as**), GNU Compiler Collection (**gcc**) y GNU Debbuger (**gdb**). Todas estas herramientas forman parte de la GNU toolchain que viene instalada por defecto en la mayoría de las distribuciones basadas en Linux, en concreto Raspbian. Para obtener más información sobre estos comandos se puede recurrir a la ayuda del sistema con `man as`, `man gcc` y `man gdb`.

1.1.4. Configuración del entorno para realizar las prácticas en casa

Las instrucciones vienen detalladas en esta dirección:
http://elinux.org/RPi_Easy_SD_Card_Setup

Vamos a hacer un resumen de cómo se haría en Windows. Para otros sistemas operativos (Linux, Mac OS) seguir las instrucciones antes mencionadas.

1. Descargamos la última versión de RASPBIAN en la siguiente url:
   http://www.raspberrypi.org/downloads/
2. Extraemos del .zip el archivo de imagen, en nuestro caso se llama 2014-01-07-wheezy-raspbian.img, aunque seguramente tu versión será más moderna.
3. Insertamos una tarjeta SD en tu PC (slot SD o adaptador USB) y nos aseguramos de que funcione correctamente. Si no, la formateamos en FAT32.
4. Nos bajamos e instalamos la utilidad Win32DiskImager.
   http://sourceforge.net/projects/win32diskimager
5. Ejecutamos como Administrador la utilidad anterior.
6. Dentro de la utilidad, seleccionamos el archivo de imagen anterior, 2014-01-07-wheezy-raspbian.img
7. Seleccionamos en Device la letra de unidad que nos apareció en el paso 3. Debemos asegurarnos de que la letra sea la correcta, de lo contrario podríamos destruir los datos de nuestro disco duro.
8. Pulsamos el botón Write y esperamos a que se complete la escritura.
9. Salimos de la utilidad y extraemos la tarjeta SD.
10. Ya estamos listos para introducir la tarjeta SD en nuestra Raspberry Pi.

9. Escribimos `sudo halt` para salir limpiamente del sistema emulado.

10. Cerramos la ventana de QEMU y creamos el siguiente archivo `lanzador.bat`.

```
qemu-system-armw -kernel kernel-qemu -cpu arm1176
-m 256 -M versatilepb -no-reboot -serial stdio -append
"root=/dev/sda2 panic=1 rootfstype=ext4 rw"
-hda 2014-01-07-wheezy-raspbian.img
```

11. Ejecutamos el archivo `lanzador.bat` que acabamos de crear. Ya hemos terminado. Todos los archivos que vayamos creando se almacenan en la imagen como si se tratase de una SD real corriendo sobre una Raspberry Pi real.

1.1.5. Aspecto de un programa en ensamblador

En el listado 1.1 se muestra el código de la primera práctica que probaremos. En el código hay una serie de elementos que aparecerán en todos los programas y que estudiaremos a continuación.

Listado 1.1: Código del programa intro1.s

```
.data

var1:    .word   3
var2:    .word   4
var3:    .word   0x1234

.text
.global main

main:    ldr     r1, puntero_var1    /* r1 <- &var1    */
         ldr     r1, [r1]            /* r1 <- *r1      */
         ldr     r2, puntero_var2    /* r2 <- &var2    */
         ldr     r2, [r2]            /* r2 <- *r2      */
         ldr     r3, puntero_var3    /* r3 <- &var3    */
         add     r0, r1, r2          /* r0 <- r1 + r2  */
         str     r0, [r3]            /* *r3 <- r0      */
         bx      lr

puntero_var1:    .word   var1
puntero_var2:    .word   var2
puntero_var3:    .word   var3
```

La principal característica de un módulo fuente en ensamblador es que existe una clara separación entre las instrucciones y los datos. La *estructura más general de un módulo fuente* es:

* **Sección de datos.** Viene identificada por la directiva *.data*. En esta zona se definen todas las variables que utiliza el programa con el objeto de reservar memoria para contener los valores asignados. Hay que tener especial cuidado para que los datos estén alineados en palabras de 4 bytes, sobre todo después de las cadenas. Alinear significa rellenar con ceros el final de un dato para que el siguiente dato comience en una dirección múltiplo de 4 (con los dos bits menos significativos a cero). Los datos son modificables.

* **Sección de código.** Se indica con la directiva *.text*, y sólo puede contener código o datos no modificables. Como todas las instrucciones son de 32 bits no hay que tener especial cuidado en que estén alineadas. Si tratamos de escribir en esta zona el ensamblador nos mostrará un mensaje de error.

De estas dos secciones la única que obligatoriamente debe existir es la sección .text (o sección de código). En el ejemplo 1.1 comprobamos que están las dos.

Un módulo fuente, como el del ejemplo, está formado por instrucciones, datos, símbolos y directivas. Las instrucciones son representaciones nemotécnicas del juego de instrucciones del procesador. Un dato es una entidad que aporta un valor numérico, que puede expresarse en distintas bases o incluso a través de una cadena. Los símbolos son representaciones abstractas que el ensamblador maneja en tiempo de ensamblado pero que en el código binario resultante tendrá un valor numérico concreto. Hay tres tipos de símbolos: las etiquetas, las macros y las constantes simbólicas. Por último tenemos las directivas, que sirven para indicarle ciertas cosas al ensamblador, como delimitar secciones, insertar datos, crear macros, constantes simbólicas, etc... Las instrucciones se aplican en tiempo de ejecución mientras que las directivas se aplican en tiempo de ensamblado.

Datos

Los datos se pueden representar de distintas maneras. Para representar números tenemos 4 bases. La más habitual es en su forma decimal, la cual no lleva ningún delimitador especial. Luego tenemos otra muy útil que es la representación hexadecimal, que indicaremos con el prefijo *0x*. Otra interesante es la binaria, que emplea el prefijo *0b* antes del número en binario. La cuarta y última base es la octal, que usaremos en raras ocasiones y se especifica con el prefijo *0*. Sí, un cero a la izquierda de cualquier valor convierte en octal dicho número. Por ejemplo 015 equivale a 13 en decimal. Todas estas bases pueden ir con un signo menos delante, codificando el valor negativo en complemento a dos. Para representar carácteres y cadenas emplearemos las comillas simples y las comillas dobles respectivamente.

Símbolos

Como las etiquetas se pueden ubicar tanto en la sección de datos como en la de código, la versatilidad que nos dan las mismas es enorme. En la zona de datos, las etiquetas pueden representar variables, constantes y cadenas. En la zona de código podemos usar etiquetas de salto, funciones y punteros a zona de datos.

Las macros y las constantes simbólicas son símbolos cuyo ámbito pertenece al preprocesador, a diferencia de las etiquetas que pertenecen al del ensamblador. Se especifican con las directivas .macro y .equ respectivamente y permiten que el código sea más legible y menos repetitivo.

Instrucciones

Las instrucciones del **as** (a partir de ahora usamos **as** para referirnos al ensamblador) responden al formato general:

```
Etiqueta:  Nemotécnico  Operando/s  /* Comentario */
```

De estos campos, sólo el nemónico (nombre de la instrucción) es obligatorio. En la sintaxis del **as** cada instrucción ocupa una línea terminando preferiblemente con el ASCII 10 (LF), aunque son aceptadas las 4 combinaciones: CR, LF, CR LF y LF CR. Los campos se separan entre sí por al menos un carácter espacio (ASCII 32) o un tabulador y no existe distinción entre mayúsculas y minúsculas.

```
main:   ldr     r1, puntero_var1    /* r1 <- &var1    */
```

El Campo *etiqueta*, si aparece, debe estar formado por una cadena alfanumérica. La cadena no debe comenzar con un dígito y no se puede utilizar como cadena alguna palabra reservada del **as** ni nombre de registro del microprocesador. En el ejemplo, la etiqueta es `main:`.

El campo *Nemotécnico* (`ldr` en el ejemplo) es una forma abreviada de nombrar la instrucción del procesador. Está formado por caracteres alfabéticos (entre 1 y 11 caracteres).

El campo *Operando/s* indica dónde se encuentran los datos. Puede haber 0, 1 ó más operandos en una instrucción. Si hay más de uno normalmente al primero se le denomina destino (salvo excepciones como `str`) y a los demás fuentes, y deben ir separados por una coma. Los operandos pueden ser registros, etiquetas, valores inmediatos o incluso elementos más complejos como desplazadores/rotadores o indicadores de pre/post-incrementos. En cualquiera de los casos el tamaño debe ser una palabra (32 bits), salvo contadas excepciones como `ldr` y `str` donde puede ser media palabra (16 bits) o un byte (8 bits). En el ejemplo `r1` es el operando destino, de tipo registro, y `puntero_var1` es el operando fuente, una etiqueta. Tanto `r1` como `puntero_var1` hacen referencia a un valor de tamaño palabra (32 bits).

El campo *Comentario* es opcional (`r1 <- &var1`, en el ejemplo) y debe comenzar con la secuencia `/*` y acabar con `*/` al igual que los comentarios multilínea en C. No es obligatorio que estén a la derecha de las instrucciones, aunque es lo habitual. También es común verlos al comienzo de una función (ocupando varias líneas) para explicar los parámetros y funcionalidad de la misma.

Cada instrucción del **as** se refiere a una operación que puede realizar el microprocesador. También hay pseudoinstrucciones que son tratadas por el preprocesador como si fueran macros y codifican otras instrucciones, como `lsl rn, #x` que codifica `mov rn, rn, lsl #x`, o bien `push/pop` que se traducen instrucciones `stm/ldm` más complejas y difíciles de recordar para el programador. Podemos agrupar el conjunto de instrucciones del **as**, según el tipo de función que realice el microprocesador, en las siguientes categorías:

- *Instrucciones de transferencia de datos* Mueven información entre registros y posiciones de memoria. En la arquitectura ARMv6 no existen puertos ya que la E/S está mapeada en memoria. Pertenecen a este grupo las siguientes instrucciones: **mov, ldr, str, ldm, stm, push, pop**.
- *Instrucciones aritméticas.* Realizan operaciones aritméticas sobre números binarios o BCD. Son instrucciones de este grupo **add, cmp, adc, sbc, mul**.
- *Instrucciones de manejo de bits.* Realizan operaciones de desplazamiento, rotación y lógicas sobre registros o posiciones de memoria. Están en este grupo las instrucciones: **and, tst, eor, orr, LSL, LSR, ASR, ROR, RRX**.
- *Instrucciones de transferencia de control.* Se utilizan para controlar el flujo de ejecución de las instrucciones del programa. Tales como **b, bl, bx, blx** y sus variantes condicionales.

En esta sesión práctica se explorarán algunas de estas instrucciones. Para buscar información sobre cualquiera de ellas durante las prácticas, recuerda que puedes utilizar el manual técnico del ARM1176JZF-S [6].

Directivas

Las directivas son expresiones que aparecen en el módulo fuente e indican al compilador que realice determinadas tareas en el proceso de compilación. Son fácilmente distinguibles de las instrucciones porque siempre comienzan con un punto. El uso de directivas es aplicable sólo al entorno del compilador, por tanto varían de un compilador a otro y para diferentes versiones de un mismo compilador. Las directivas más frecuentes en el **as** son:

- *Directivas de asignación:* Se utilizan para dar valores a las constantes o reservar posiciones de memoria para las variables (con un posible valor inicial). **.byte, .hword, .word, .ascii, .asciz, .zero y .space** son directivas que indican al compilador que reserve memoria para las variables del tipo indicado. Por ejemplo:

```
a1:   .byte  1        /* tipo byte, inicializada a 1 */
var2: .byte  'A'      /* tipo byte, al caracter 'A'  */
var3: .hword 25000    /* tipo hword (16 bits) a 25000*/
var4: .word 0x12345678 /* tipo word de 32 bits       */
b1:   .ascii "hola"   /* define cadena normal        */
b2:   .asciz "ciao"   /* define cadena acabada en NUL*/
dat1: .zero  300      /* 300 bytes de valor cero     */
dat2: .space 200, 4   /* 200 bytes de valor 4        */
```

  La directiva **.equ** (ó **.set**) es utilizada para asignar un valor a una constante simbólica:

```
.equ N, -3      /* en adelante N se sustituye por -3 */
```

- *Directivas de control:* **.text y .data** sirven para delimitar las distintas secciones de nuestro módulo. **.align alineamiento** es para alinear el siguiente dato, rellenando con ceros, de tal forma que comience en una dirección múltiplos del número que especifiquemos en *alineamiento*, normalmente potencia de 2. Si no especificamos *alineamiento* por defecto toma el valor de 4 (alineamiento a palabra):

```
a1: .byte 25    /* definimos un byte con el valor 25 */
    .align      /* directiva que rellena con 3 bytes */
a2: .word 4     /* variable alineada a tamaño palabra*/
```

  .include para incluir un archivo fuente dentro del actual. **.global** hace visible al enlazador el símbolo que hemos definido con la etiqueta del mismo nombre.

- *Directivas de operando:* Se aplican a los datos en tiempo de compilación. En general, incluyen las operaciones lógicas `&`, `|`, $\sim$, aritméticas `+`, `-`, `*`, `/`, `%` y de desplazamiento $<$, $>$, $<<$, $>>$:

```
.equ pies, 9      /* definimos a 9 la constante pies */
.equ yardas, pies/3     /* calculamos las yardas= 3 */
.equ pulgadas, pies*12  /* calculamos pulgadas= 108 */
```

- *Directivas de Macros:* Una *.macro* es un conjunto de sentencias en ensamblador (directivas e instrucciones) que pueden aparecer varias veces repetidas en un

programa con algunas modificaciones (opcionales). Por ejemplo, supongamos que a lo largo de un programa realizamos varias veces la operación n^2+1 donde n y el resultado son registros. Para acortar el código a escribir podríamos usar una macro como la siguiente:

```
.macro CuadM1 input, aux, output
    mul     aux, input, input
    add     output, aux, #1
.endm
```

Esta macro se llama *CuadM1* y tiene tres parámetros (input, aux y output). Si posteriormente usamos la macro de la siguiente forma:

```
        CuadM1  r1, r8, r0
```

el ensamblador se encargará de expandir la macro, es decir, en lugar de la macro coloca:

```
        mul     r8, r1, r1
        add     r0, r8, #1
```

No hay que confundir las macros con los procedimientos. Por un lado, el código de un procedimiento es único, todas las llamadas usan el mismo, mientras que el de una macro aparece (se expande) cada vez que se referencia, por lo que ocuparán más memoria. Las macros serán más rápidas en su ejecución, pues es secuencial, frente a los procedimientos, ya que implican un salto cuando aparece la llamada y un retorno cuando se termina. La decisión de usar una macro o un procedimiento dependerá de cada situación en concreto, aunque las macros son muy flexibles (ofrecen muchísimas más posibilidades de las comentadas aquí). Esta posibilidad será explotada en sesiones más avanzadas.

1.1.6. Ensamblar y *linkar* un programa

La traducción o ensamblado de un módulo fuente (**nombreprograma.s**) se realiza con el programa Gnu Assembler, con el siguiente comando:

```
as -o nombreprograma.o nombreprograma.s
```

NOTA: tanto el comando **as** como el nombre del programa son sensibles a las mayúsculas. Por tanto el comando debe ir en minúsculas y el nombre como queramos, pero recomendamos minúsculas también. Las opción **-o nombreprograma.o** puede ir después de **nombreprograma.s**.

El **as** genera un fichero nombreprograma.o.

Para montar (*linkar*) hay que hacer:

```
gcc -o nombreprograma nombreprograma.o
```

NOTA: Nuevamente, tanto **gcc** como el nombre del programa deben estar en minúsculas. Este comando es muy parecido al anterior, podemos poner si queremos **-o nombreprograma** detrás de **nombreprograma.o**. La única diferencia es que el archivo no tiene extensión, que por otro lado es una práctica muy recomendable para ejecutables en Linux.

Una vez hecho ésto, ya tenemos un fichero ejecutable (nombreprograma) que podemos ejecutar o depurar con el **gdb**.

1.2. Enunciados de la práctica

1.2.1. Cómo empezar

Recuerda que en laboratorio las raspberries no tienen monitor ni teclado, la única conexión con el mundo real es el puerto Ethernet. En el apéndice C se explica otro mecanismo para conectar.

Así que antes de nada averigua cuál es la dirección IP de la Raspberry Pi dentro de la red local. Por defecto el usuario es `pi` y la contraseña `raspberry`. Suponiendo que la dirección IP asignada es `192.168.1.42`, utilizaremos ssh:

```
ssh pi@192.168.1.42
```

Y luego introduce la contraseña. Ya conectado a la Raspberry Pi desde un PC a través de ssh. Todos los alumnos se conectarán con el mismo nombre de usuario, por lo que hay que dejar limpio el directorio de trabajo antes de terminar la sesión.

También es buena idea conectarte por ssh a la Raspberry que tengas en casa como acabamos de explicar, así te ahorras el tener que disponer de teclado y monitor extra. Desde Windows puedes bajarte putty.exe en
http://www.chiark.greenend.org.uk/~sgtatham/putty/download.html (no requiere instalación) y crear un acceso directo en el escritorio con el siguiente destino, cambiando **ruta** y **192.168.1.42** por lo que corresponda.

```
C:\ruta\putty.exe 192.168.1.42 -l pi -pw raspberry
```

Comenzaremos con el programa que hemos visto en el listado 1.1, y que se encuentra en el fichero `intro1.s`. Edítalo con el programa **nano** para verlo (y practicar un poco):

```
nano intro1.s
```