Estructura de Computadores (240306): Prácticas_ARM

HISTORIAL DE REVISIONES

NÚMERO	FECHA	MODIFICACIONES	NOMBRE
0.6.0	2019-11-19		C.

Índice general

Capítulo 1

Introducción

- La arquitectura de una computadora se puede clasificar en dos grandes grupos: la arquitectura **CISC** y la arquitectura **RISC**. Un ejemplo de la arquitectura CISC son los microprocesadores con arquitectura x86 y x86-64 como son los procesadores de Intel en computadoras de sobremesa y servidores. Un ejemplo de la arquitectura RISC son los microprocesadores ARM en computadores portátiles como la telefonía móvil, tablet, etc … y los sistemas empotrados como la automoción, instrumentación médica, etc donde el coste y consumo energético juegan un papel prioritario.

Capítulo 2

Kit de Evaluación: Terasic DE1-SoC

- La tarjeta de evaluación Terasic DE1-SoC está basada en un chip de Silicio System On Chip (SoC) el Cúal integra dos componentes básicos:
- un módulo de lógica programable FPGA (Field Logic Gate Array) de la familia Cyclone V modelo 5CSEMA5F31C6N
- un procesador (Hard Processor System HPS) con dos núcleos ARM Cortex A9 La tarjeta de evaluación dispone además del chip SoC de una gran variedad de periféricos para interactuar con el mundo exterior como son:
  - LEDS
  - conmutadores
  - botones
  - displays de 7 segmentos
  - Comunicaciones: ethernet, usb, ...
  - multimedia: audio, video vga
  - lector para tarjetas microSD
  - otros

Capítulo 3

Acceso a las computadoras ARM de Terasic

- Mediante un navegador instalado en la computadora del alumno se puede acceder desde el campus de la Upna a través de la red intranet a la computadora Terasic DE1-SoC instalada en el laboratorio con las credenciales:
- la URL : *http://weblab-cpl.unavarra.es:8080/guacamole*
- usuario: *student*
- password: **upna**
- password de la pasarela: **arm_fpga**
- nombre de la cuenta del@ alumn@ en la computadora ARM: uno de los apellidos de la lista
- password de la cuenta del@ alumn@ en la computadora ARM: **student**
- Abriéndose sesión en la consola con el prompt del shell de linux: *student@de1-soc1:~$*
- Los programas fuente se encuentran en el archivo *fuentes_arm.zip*
- Si algún@ alumn@ desea transferir documentos fuera de la computadora es neceario archivar dichos documentos en un fichero tipo ZIP identificando el archivo con el nombre de usuario y la extensión zip, p.ej, *candido.zip*. Posteriormente los profesores enviaran dicho documento al alumno por correo electrónico.
- Para salir de la consola ejecutar dos veces: `exit`

Capítulo 4

Programación básica

4.1. Módulos fuente

- Colección de Módulos fuente:
- *datos.s*
- *puntero.s*
- *puntero_automatico.s*
- *hello_world.s*
- *entero_negativo.s*
- *desplaz_log-arit.s*
- *inmediato.s*
- *indirecto_indexado.s*
- *sum1toN.s*
- *maximum.s*
- *power.s*

4.2. ISA: arquitectura ARMv7 de 32 bits

- El procesador ARM Cortex-A9 esta diseñado para ejecutar el repertorio de instrucciones con arquitectura ARMv7 de 32 bits cuyas características más importantes son:
- tamaño del formato de instrucciones uniforme: 32 bits
- alineamiento en memoria de las instrucciones en direcciones múltiplo de 4 bits.
- para acceder a posiciones de la memoria principal es necesario emplear instrucciones load/store (mnemónicos LDR/SDR) mediante direccionamiento a memoria: directo, indirecto, indirecto con desplazamiento.
- Excepto las instrucciones load/store el resto de instrucciones hacen referencia a los operandos mediante los tipos de direccionamiento: inmediato y registro.

4.3. Assembler

- Sintaxis de una instrucción en lenguaje ensamblador armV7

```
ETIQUETA:       MNEMONICO       OPERANDO_DESTINO, OPERANDO_FUENTE       @COMENTARIO
```

- Comentarios: se puede utilizar las marcas */* comentario */* ó @

atención
El orden de los operandos en la arquitectura armv7 es el inverso de la arquitectura i386

4.4. Toolchain

4.4.1. herramientas

- Dado que se va a trabajar en una computadora con un procesador con la arquitectura ARM es necesario tener instaladas las herramientas de compilación, ensamblaje, linkaje, depurador, etc … para dicha arquitectura
- Herramientas en cadena para la compilación y análisis de programación de bajo nivel
  - colección de compiladores GNU: **gcc**
  - ensamblador: **as**
  - linker: **ld**
  - depurador: **gdb**
  - volcado de la memoria: **objdump**
  - analizador del módulo binario ejecutable: **readelf**

4.4.2. compilación

- `gcc -g -nostdlib -o modulo modulo.s`
  - modulo: nombre del fichero a compilar
  - -g : opción de depuración
  - -nostdlib : no se utilizan al linkar las librerías standard de inicialización en modo estático. Equivale a -nostartfiles que linka en modo dinámico.
- `gcc -g -nostartfiles -o modulo modulo.s`
- `gcc -g -nostartfiles -mcpu=cortex-a9 -march=armv7-a -o modulo modulo.s`
- `as -gstabs -o modulo.o modulo.s`
- `as -mcpu=cortex-a9 -march=armv7-a -gstabs -o modulo.o modulo.s`
  - se indica explicitamente la arquitectura del módulo objeto ejecutable de salida:
    - procesador físico cortex-a9 y arquitectura isa del repertorio de instrucciones : armv7-a
- `ld -o modulo modulo.o`
  - `objdump -i` : lista las arquitecturas de salida posibles

Capítulo 5

Vim editor

5.1. básico

- https://docstore.mik.ua/orelly/unix3/vi/index.htm
- https://vim.fandom.com/wiki/Cut/copy_and_paste_using_visual_selection
- https://www.thegeekdiary.com/basic-vi-commands-cheat-sheet/
  - vi hello_world.s
  - al entrar en vi el editor está en modo **vi**
  - con una de las órdenes **i,I,a,A,o,O** se pasa al modo INSERT
  - con la orden **ESC** se sale del modo INSERT al modo VI
  - con la orden **:** se entra al modo **comando**
    - con la orden **w** salva lo editado
    - con la orden **wq** se salva y se sale del editor
    - con la orden **q** se sale del editor
    - con la orden **q!** se sale del editor sin salvar los últimos cambios

5.2. modo insert

- Al escribir se inserta el texto
- borrar: supr, backspace
- con la orden ESC se sale del modo INSERT al modo VI

5.3. modo comando

- https://www.linux.com/tutorials/vim-tips-basics-search-and-replace/
- buscar → **/**
- buscar y reemplazar → **:s/search/replace/g**
- → en todo el fichero **: %s/item_busqueda/item_reemplazo/g**

5.4. modo visual

- desde el modo vi
  - v → visual caracter
  - V → visual línea
  - Ctrl-v → visual bloque
- navegar: flechas
- desde el modo **visual**
  - c → cortar
  - y → copiar
  - p → pegar

5.5. Editor bloqueado: Documento swap

- En caso de quedarse bloqueado el editor es necesario cerrar el "terminal" para poder desbloquear el editor. Se genera un documento backup-swap con lo último realizado.
- si el documento al abrir informa de que no se cerró correctamente y que existe un backup de dicho documento

```
Swap file ".datos.s.swp" already exists!
[O]pen Read-Only, (E)dit anyway, (R)ecover, (D)elete it, (Q)uit, (A)bort:
```

  - delete: elimina el fichero swap *.datos.s.swp*
  - recover: recupera el archivo con los cambios pero no elimina el fichero swap ***..swp*** → *rm* ..swp

Capítulo 6

Refs

- Hoja de referencia rápida
- Robert Plantz: capítulos 9 en adelante.
- Assembler:guía de usuario oficial de ARM
- Arquitectura ARM
- Repertorio de Instrucciones: Manual oficial
- Repertorio de Instrucciones: Manual KEIL
- Repertorio Instrucciones:Plantilla de referencia rápida
- ARM Assembly Language Programming Peter Knaggs April 7, 2016
- gnu assembler as ARM dependent options

Capítulo 7

datos.s

- módulo fuente

```
/*
 Universidad Pública de Navarra Octubre 2019
 Estructura de Computadores 240306
 Programa en lenguaje ensamblador AT&T para la arquitectura ARMv7 del procesador ARM Cortex A9

 Programa fuente: datos.s
 Descripción: crear e inicializar una variable puntero.
 Herramientas binarias de GNU: binutils: gcc,objdump,readelf
 Sistemas Operativo : GNU/Linux
 Toolchain de GNU : gcc, as, ld
 Compiler:      gcc -g -o datos  datos.s
 Assembler:     as -gstabs -o datos.o datos.s
 Linker:        ld -o datos datos.o
 Debugger:      gdb

*/


        .equ pies, 30           @parámetro constante
        .equ pulgadas, pies*12

        .data

a1:     .byte -1
        .align                  @alinea el siguiente dato en direcciones múltiplos de 4
var2:   .byte 'A'
var3:   .hword 25000            @equivale a la directiva .short -> 2 bytes
var4:   .word 0x12345678        @equivale a la directiva .int -> 4 bytes
b1:     .ascii "hola"
        .align 2                @alinea el siguiente dato en direcciones múltiplo de 2 al cuadrado
b2:     .asciz "ciao"
dat1:   .zero 300               @ 300 bytes inicializados con cero
dat2:   .space 200,4            @ 200 bytes inicializados con 4
        .text
        .global main
main:
        bx lr                   @ el registro lr guarda la dirección de retorno al sistema operativo
```

7.1. Cuestiones

- Editar el valor de la variable *a1* y compilar insertando en el módulo objeto la tabla de símbolos para el depurador
- Cuál es la diferencia entre *main* y *_start*
- Cúal es la dirección de memoria de la variable a1
- Cúal es el contenido en hexadecimal de la variable a1
- Cúal es la dirección de memoria de la variable var2
- Comprobar el efecto de la directiva *.align*
- Cúal es el contenido de la variable var2
- Cúal es el contenido del byte a la que apunta la dirección de memoria de la variable var4
- Cúal es la dirección de memoria de la variable b2
- Cúal es el contenido de los bytes de las direcciones de memoria &b2 y &b2+4
- Cúal es el contenido de los registros sp, lr y pc
- Cúal es el contenido de los registros r13, r14 y r15
- Interpretación de `bx lr`

Capítulo 8

puntero.s

- módulo fuente *puntero.s*

```
/*
 Universidad Pública de Navarra Octubre 2019
 Estructura de Computadores 240306
 Programa en lenguaje ensamblador AT&T para la arquitectura ARMv7 del procesador ARM Cortex A9

 Programa fuente: puntero.s
 Descripción: crear e inicializar una variable puntero.
 Herramientas binarias de GNU: binutils: gcc,objdump,readelf
 Sistemas Operativo : GNU/Linux
 Toolchain de GNU : gcc, as, ld
 Compiler:      gcc -g -o puntero puntero.s
 Assembler:     as -gstabs -o puntero.o puntero.s
 Linker:        ld -o puntero puntero.o
 Debugger:      gdb

*/

        .data
var1:           .word 0x0A
        .text
        .global main
main:
        ldr r1, var1_pt         @ direccionamiento directo
        ldr r2, [r1]            @ direccionamiento indirecto
        bx lr


var1_pt:        .word var1

        .end
```

8.1. Cuestiones

- Cual es la dirección de la variable var1 y su contenido
- Cual es la dirección de la variable var1_pt y su contenido
- `objdump -j .data -s puntero` : -j selecciona un segmento del módulo binario y -s muestra el contenido del segmento seleccionado.

  - cual es la dirección de var1 y comprobarlo con gdb

- `objdump -j .text -h puntero` : -h muestra la cabecera (descriptor con propiedades) de la sección seleccionada
  - cual es el tamaño de la sección .text
- `objdump -j .text -s puntero`
  - calcular las direcciones del primer y último dato de la sección .text.
- `objdump -d puntero | grep main` : la opción -d desensambla las sección con código. El comando grep captura la línea que contiene main
- `objdump -d puntero | grep _start`
  - ¿cual es la dirección de entrada al código ejecutable del programa?
- `objdump -d puntero | grep -B 1 -A 7 main` : la opción -B n muestra "n líneas" Before y la opción -A m muestra "m líneas" After
  - ¿Se pueden insertar datos en la sección de instrucciones?
  - ¿El direccionamiento indirecto indexado [pc, #4] con que instrucción del código fuente esta relacionada?
  - ¿A dónde debe de apuntar el contador de programa al ejecutar la instrucción "ldr r1, [pc, #4]" para que realice la misma operación que la instrucción fuente equivalente?
    - La razón por la que el contador de programa PC no apunta a la instrucción siguiente sino dos instrucciones más, es la ejecución en paralelo de las instrucciones en una arquitectura pipeline.
  - ¿Cuál es la instrucción máquina traducida de "ldr r1, [pc, #4]" y cuál es su tamaño?
  - Observar en que se ha TRADUCIDO el modo de direccionamiento DIRECTO de la instrucción "ldr r1, var1_pt" ¿Si el tamaño de las instrucciones máquina ARM únicamente pueden tener 4 ó 2 bytes, podría utilizarse en lenguaje máquina el direccionamiento directo con direcciones de 4 bytes?¿Por qué?
  - Por qué crees que no coincide literalmente para la ISA ARMv7 el código ensamblador y el código objeto máquina
- Gdb: Al ejecutarse en modo paso a paso, la CPU aunque tiene una arquitectura pipeline, no ejecuta las instrucciones en paralelo, sino de forma secuencial. Comprobar que en el modo paso a paso el contador de programa PC apunta a la instrucción siguiente y a dos instrucciones siguientes como en la ejecución en paralelo.
- `objdump -S puntero | grep -B 1 -A 7 main`: la opción -S muestra el código del programa fuente
- **literal pool**: se conoce con este nombre la reserva de memoria en la sección de instrucciones para almacenar datos literales.
  - Un ejemplo de dato literal es la dirección de la variable var1 que se guarda en var1_pt almacenado así un dato en la dirección var_pt de la sección de instrucciones y formando así un "literal pool".

Capítulo 9

puntero_automatico

- módulo fuente *puntero_automatico.s*

```
/*
 Universidad Pública de Navarra Octubre 2019
 Estructura de Computadores 240306
 Programa en lenguaje ensamblador AT&T para la arquitectura ARMv7 del procesador ARM ↩
     Cortex A9

 Programa fuente: puntero_automatico.s
 Descripción: crear e inicializar una variable puntero_automatico.
 Herramientas binarias de GNU: binutils: gcc,objdump,readelf
 Sistemas Operativo : GNU/Linux
 Toolchain de GNU : gcc, as, ld
 Compiler:      gcc -g -o puntero_automatico puntero_automatico.s
 Assembler:     as -gstabs -o puntero_automatico.o puntero_automatico.s
 Linker:        ld -o puntero_automatico puntero_automatico.o
 Debugger:      gdb

*/

        .data
var1:           .short 0x0A0B
var2:           .word 0b11110000
        .text
        .global main
main:
        ldr     r1, =var1       @ r1 <- &var1
        ldrsb   r1, [r1]        @ r1 <- *r1  @ operando Byte con extensión de Signo
        ldr     r2, =var2       @ r2 <- &var2
        ldrsb   r2, [r2]        @ r2 <- *r2  @ operando Byte con extensión de Signo
        bx lr                   @ pc <- lr

        .end
```

9.1. Cuestiones

- Comprobar el contenido de r1 antes y después de ejecutar la primera instrucción *ldrsb*
- Comprobar el contenido de r2 antes y después de ejecutar la segunda instrucción *ldrsb*
- Desensamblar el código para comprobar la diferencia entre el código fuente y el código objeto

- `objdump -S puntero_automatico | grep -B 1 -A 7 main`
  - No hay una repetición del código máquina aunque lo parezca. Fijarse que bajo la repetición de código fuente está en un caso las instrucciones máquina traducidas y en el otro caso la consitución de un "literal pool".
  - "literal pool" : el ensamblador durante el proceso de traducción almacena datos literales como las direcciones var1 y var2 en la sección de instrucciones.
- El direccionamiento [pc, #8] es un direccionamiento indexado: $pc+8$
- Por qué el compilador a traducido el código fuente en este tipo de direccionamiento y esos desplazamientos específicos

Capítulo 10

hello_world.s

- módulo fuente *hello_world.s*

```
/*
 Universidad Pública de Navarra Octubre 2019
 Estructura de Computadores 240306
 Programa en lenguaje ensamblador AT&T para la arquitectura ARMv7 del procesador ARM ↩
     Cortex A9

 Programa fuente: hello_world.s
 Descripción: Imprimir en la pantalla un saludo.
 Herramientas binarias de GNU: binutils: gcc,objdump,readelf
 Sistemas Operativo : GNU/Linux
 Toolchain de GNU : gcc, as, ld
 compiler:      gcc -g -nostdlib -o hello_world hello_world.s
                gcc -march=armv7-a+mp -mcpu=cortex-a9 -mfpu=neon -nostdlib -g -o ↩
                    hello_world hello_world.s
                gcc -march=armv7-a+mp -mcpu=cortex-a9 -mfpu=neon -mfloat-abi=hard -mlittle ↩
                    -endian -mabi=aapcs-linux -mtune=cortex-a9 -nostartfiles -o ↩
                    hello_world hello_world.s
 Assembler:       as -gstabs -o hello_world.o hello_world.s
 Linker:          ld -o hello_world hello_world.o
 Debugger:        gdb

*/
        .arch   armv7-a
        .arch_extension mp
        .cpu    cortex-a9
        .fpu    neon
        .syntax unified         @ modern syntax (thumb-2 and arm)
        .data

msg:
        .ascii "Hello, ARM World!\n"    @reserva de memoria inicializada con un string
        len = . - msg                   @reserva de memoria inicializada con la operación: ↩
             dirección actual - dirección msg


        .text

        .globl _start
_start:
        /* Llamada al sistema operativo Linux para ejecutar la función write(descriptor ↩
            fichero, string, tamaño)
           Manual en el shell de linux: man 2 write */
```

```
/* Llamada al sistema operativo Linux para ejecutar la función write(descriptor ↩
    fichero, string, tamaño)
   Manual en el shell de linux: man 2 write */
mov r0, $1      @ 1º argumento de la función write: descriptor de fichero : ↩
    pantalla
ldr r1, =msg    @ 2º argumento de la función write: dirección del string
ldr r2, =len    @ 3º argumento de la función write: longitud del string
mov r7, $4      @ Código de la función write: 4
swi $0          @ LLamada al sistema operativo Linux

/* Llamada al sistema operativo Linux para ejecutar la función exit(status code)
   Manual en el shell de linux: man 3 write */
mov r0, $0      @ 1º argumento de la función exit
mov r7, $1      @ Código de la función exit: 1
swi $0          @ LLamada al sistema operativo Linux
```

10.1. Cuestiones

- Explicar las opciones del compilador `gcc` que comienzan con *-m* com *-mcpu*
- Diferencia entre poner las opciones explícitamente en el compilador `gcc` o en el módulo fuente ensamblador.
- Opciones por defecto del compilador `gcc`
- Por qué ARM es una arquitectura load/store e i386 no.
- Cuál es la principal diferencia entre `mov` y `ldr` en cuanto a modos de direccionamiento
- Ejecutar el programa con el debugger `gdb` observando el contenido de los registros.
- Cuál es el puntero que apunta al string msg y cuál es la dirección del string.
- Significado de las siglas del mnemónico `swi`

Capítulo 11

entero_negativo.s

- módulo fuente

```
/*
 Universidad Pública de Navarra Octubre 2019
 Estructura de Computadores 240306
 Programa en lenguaje ensamblador AT&T para la arquitectura ARMv7 del procesador ARM ↩
     Cortex A9

 Programa fuente: entero_negativo.s
 Descripción: crear e inicializar una variable puntero.
 Herramientas binarias de GNU: binutils: gcc,objdump,readelf
 Sistemas Operativo : GNU/Linux
 Toolchain de GNU : gcc, as, ld
 Compiler:      gcc -g -o entero_negativo  entero_negativo.s
 Assembler:     as -gstabs -o entero_negativo.o entero_negativo.s
 Linker:        ld -o entero_negativo entero_negativo.o
 Debugger:      gdb

 CPSR: current program status register ->  registro de flags -> 8 bits 24:31
      Bits  28/29/30/31 -> Flags VCZN -> Overflow/Carry/Zero/Negative

*/


        .data
msg1:   .string "\n\t LOS ENTEROS NEGATIVOS SE REPRESENTAN EN COMPLEMENTO A 1\n"
msg2:   .string "\n\t LOS ENTEROS NEGATIVOS SE REPRESENTAN EN COMPLEMENTO A 2\n"

        .text
        .global main
main:
        mov r0, #0
        sub r1, r0, #4
        @ 4=0b00000100 y el complemento a 1 -> 11111011  =0xFB
        @ 4=0b00000100 y el complemento a 2 -> 11111011+1=0xFC

        cmp r1, #0xFFFFFFFC     @ -4 en C2 es 0xFFFFFFFC
        beq C2
        cmp r1, #0xFFFFFFFB     @ -4 en C1 es 0xFFFFFFFB
        beq C1
C1:     @ complemento a 1
        ldr r0,=msg1
        push {lr}
        bl puts
```

```
        pop {lr}
        bx lr
C2:     @ complemento a 2
        ldr r0,=msg2
        push {lr}
        bl puts
        pop {lr}
        bx lr
        .end
```

11.1. Cuestiones

- En que tipo de formato se representan los número negativos en la arquitectura armv7
- Cuál es la diferencia entre las instrucciones *b* , *bx* , y *bl*
  - b → salto incondicional ¿retorno? ¿qué ocurre si hacemos "b puts"?
  - bl → branch link → guarda la dirección de retorno en el registro "link register lr" (R14)
  - bx Reg → salta a donde apunta Reg.
- Concreta tres mnemónicos diferentes de instrucciones de salto condicionales.
- ¿Cómo se pasarían en asm 6 argumentos a una función de la librería standard libc?

Capítulo 12

desplaz_log-arit.s

- módulo fuente

```
/*
 Universidad Pública de Navarra Octubre 2019
 Estructura de Computadores 240306
 Programa en lenguaje ensamblador AT&T para la arquitectura ARMv7 del procesador ARM ↩
     Cortex A9

 Programa fuente: desplaz_log-arit.s
 Descripción: crear e inicializar una variable puntero.
 Herramientas binarias de GNU: binutils: gcc,objdump,readelf
 Sistemas Operativo : GNU/Linux
 Toolchain de GNU : gcc, as, ld
 Compiler:      gcc -g -o desplaz_log-arit  desplaz_log-arit.s
 Assembler:     as -gstabs -o desplaz_log-arit.o desplaz_log-arit.s
 Linker:        ld -o desplaz_log-arit desplaz_log-arit.o
 Debugger:      gdb

*/


        .data
arg1:   .string "\n\t EL RESULTADO ES %d \n"
        .text
        .global main
main:
        mov r0, #1
        add r2, r0, #2
        @ Logic Shift Left LSL
        mov r1, r2, LSL #1      @r1<-(r2<<1)
        mov r3, r1, LSL #2      @r3<-(r1<<2)  ; r3<-r1*4
        @ Cambiar el signo de r3
        mvn r4, r3      @ r4<-~r3
        add r4, r4, #1  @ r4<-r4+1
        @ Arithmetic Shift Right ASR
        mov r1, r4, ASR #3      @r1<-(r4>>3)  ; r1<-r4/8
        ldr r0, =arg1
        b printf
        bx lr
```

12.1. Cuestiones

- El operando fuente en la arquitectura ARMv7 puede llevar explícita una instrucción de **desplazamiento** o **rotación**. ¿Cuál es la diferencia entre las instrucciones LSL y ASL?
- ¿La instrucción MoVNot equivale a cambiar el signo de un número entero?

Capítulo 13

inmediato.s

- módulo fuente

```
/*
 Programa en lenguaje ensamblador AT&T para la arquitectura ARMv7 del procesador ARM  ↩
     Cortex A9

 Programa fuente: inmediato.s
 Descripción: crear e inicializar una variable puntero.
 Herramientas binarias de GNU: binutils: gcc,objdump,readelf
 Sistemas Operativo : GNU/Linux
 Toolchain de GNU : gcc, as, ld
 Compiler:      gcc -g -o inmediato  inmediato.s
 Assembler:     as -gstabs -o inmediato.o inmediato.s
 Linker:        ld -o inmediato inmediato.o
 Debugger:      gdb

 CPSR: current program status register ->  registro de flags -> 8 bits 24:31
      Bits  28/29/30/31 -> Flags VCZN -> Overflow/Carry/Zero/Negative

*/

        .equ CTE0,0x0354
        .equ CTE1,0x00354000
        .equ CTE2,0x03540000
        .equ CTE3,0xFF5BFFFF
        .equ CTE4,0x87FFFFFF
        .equ CTE5,0xFFFBFBFF
        .data
cte5:   .word 0xFFFBFBFF
        .text
        .global main
main:
        mov r0, #CTE0
        mov r1, #CTE1
        mov r2, #CTE2
        mov r3, #CTE3
        mov r4, #CTE4
        ldr r5, =cte5
        ldr r6, [r5]
        bx lr
        .end
```

13.1. Cuestiones

- La instrucción mov se utiliza para inicializar los registros. El rango de los números con valores válidos está limitado a 8 bits, para lo cual el ensamblador al traducir elimina la secuencia de *0* tanto a la izda como a la dcha de la secuencia significativa si el número es positivo y lo mismo ocurre con la secuencia de *1* a la izda y a la dcha para los valores negativos.
- Comprobar que ocurre si inicializamos el registro r6 con la instrucción mov.

Capítulo 14

indirecto_indexado.s

- módulo fuente

```
/*
 Universidad Pública de Navarra Octubre 2019
 Estructura de Computadores 240306
 Programa en lenguaje ensamblador AT&T para la arquitectura ARMv7 del procesador ARM ↩
     Cortex A9

 Programa fuente: indirecto_indexado.s
 Descripción: crear e inicializar una variable puntero.
 Herramientas binarias de GNU: binutils: gcc,objdump,readelf
 Sistemas Operativo : GNU/Linux
 Toolchain de GNU : gcc, as, ld
 Compiler:      gcc -g -o indirecto_indexado  indirecto_indexado.s
 Assembler:     as -gstabs -o indirecto_indexado.o indirecto_indexado.s
 Linker:        ld -o indirecto_indexado indirecto_indexado.o
 Debugger:      gdb

 CPSR: current program status register ->  registro de flags -> 8 bits 24:31
      Bits  28/29/30/31 -> Flags VCZN -> Overflow/Carry/Zero/Negative

*/
        .equ CTE0, 0x81
        .equ CTE1, 0x8
        .data
array:  .space 200,0
var0:   .word 0x81
var1:   .hword 0x8
        .text
        .global main
main:
        mov r0, #CTE0
        ldr r1, =array
        mov r2, #CTE1
        @ Direccionamiento Indirecto sin actualizar puntero
        str r0,[r1, #+12]       @M[r1+12]<-r0
        str r0,[r1, +r2]        @M[r1+r2]<-r0
        str r0,[r1, +r2, LSL #2]        @M[r1+r2*4]<-r0

        @ Direccionamiento Indirecto Post-indexado, actualizando puntero
        str r0, [r1], #+4       @M[r1+4]<-r0 ; r1<-r1+4
        str r0, [r1], #+4       @M[r1+4]<-r0 ; r1<-r1+4
        str r0, [r1], #+4       @M[r1+4]<-r0 ; r1<-r1+4
```

```
        str r0, [r1], -r2       @M[r1-r2]<-r0 ; r1<-r1+r2
        str r0, [r1], +r2, LSL #2       @M[r1+r2+4]<-r0 ; r1<-r1+r2+4
        @ Direccionamiento Indirecto Pre-indexado, actualizando puntero
        str r0, [r1, #4]!       @r1<-r1+4 ; M[r1]<-r0
        str r0, [r1, r2]!       @r1<-r1+r2 ;  M[r1]<-r0
        str r0, [r1, r2, LSL #2]!       @r1<-r1+r2*4 : M[r1]<-r0
        bx lr
        .end
```

14.1. Cuestiones

- Ejecutar en modo paso a paso el programa comprobando como varía el contenido de los registros r1, r2 y el buffer de memoria array.
- Cual es el contenido del registro r1 al finalizar la ejecución de:
  - El bloque de direccionamiento indirecto sin actualizar puntero
  - El bloque de direccionamiento indirecto post-indexado actualizando puntero.
  - El bloque de direccionamiento indirecto pre-indexado actualizando puntero.

Capítulo 15

sum1toN.s

- módulo fuente *sum1toN.s*

```
/*      Programa: sum1toN.s
        Descripción: realiza la suma de la serie 1,2,3,...N
        Es el programa en lenguaje ARM equivalente a sum1toN.ias de la máquina IAS de von ↩
            Neumann
        gcc  -g -nostartfiles -o sum1toN sum1toN.s
        Ensamblaje as  --gstabs sum1toN.s -o sum1toN.o
        linker -> ld     -o sum1toN sum1toN.o
*/
        @  Declaración de variables
        .section .data
n:      .int 5

        .global _start

        @ Comienzo del código
        .section .text
_start:
        mov r0,#0       @ R0 implementa la variable suma
        ldr r2,=n       @ R1 implementa la variable n indirectamente
        ldr r1,[r2]
/* Direccionamiento directo:
        mov r1,n da error porque mov no admite direccionamiento a memoria directo.
                 mov admite direccionamiento inmediato si el literal de  32 bits no tiene ↩
                     repetición de ceros a izda y dcha
                 para convertirlo en un literal de 8 bits seguido de desplazamientos
        ldr r1,n Error: reubicación_interna (tipo OFFSET_IMM) no compuesta
                 Da error al intentar codificar un literal (dirección n) de 32 bits.
*/

bucle:
        add r0,r1
        subs  r1,#1
        bne bucle

        @r0 es el argumento de salida al S.O. a través de EBX según convenio

        /* exit syscall */
        mov r7, #1
        swi #0

        .end
```

15.1. Cuestiones

- Analizar el código fuente e interpretarlo.

Capítulo 16

maximum.s

- módulo fuente *maximum.s*

```
/*
        Algoritmo: búsqueda del valor máximo de una lista de números enteros
        Resutado: echo $?
        Variables del algoritmo: Implementadas en los siguientes registros:
        r3 - Guarda la dirección del primer elemento de la lista
        r2 - Guarda el índice del siguiente número de la lista a analizar
        r1 - Número presente a analizar
        r0 - Máximo valor encontrado hasta el presente
        r7 - Código de la llamada al sistema
*/

        .section .data
data_items:
        .int 3,67,34,222,45,75,54,34,44,33,22,11,66,0 @ Lista de números enteros con signo
        .section .text
        .globl _start
_start:
        ldr   r3,=data_items    @ registro base
        ldr   r1, [r3]          @ cargar el 1º elemento de la lista
        mov   r0,r1             @ valor máximo inicial
        mov   r2,#1             @ índice al segundo elemento de la lista
start_loop:                     @ start loop
        cmp   r1,#0             @ chequear si es el valor marca de fin de lista
        beq    loop_exit                @ ir a la salida si elemento marca final
        ldr   r1, [r3, +r2, LSL #2]!    @ siguiente valor a analizar
                                @ desplazamiento con post-actualización de r3
                                @ base r3,índice r2, escala 4 bytes (tamaño dato de la ↩
                                    lista)
        cmp   r1,r0             @ comparación entre el máximo presente y elemento presente
        ble   start_loop        @ siguiente elemento de la lista

        mov   r0,r1             @ encontrada un nuevo máximo
        bal    start_loop       @ siguiente elemento de la lista
loop_exit:

        /* exit syscall */
        mov r7, #1
        swi #0
```

16.1. Cuestiones

- Analizar el código fuente e interpretarlo.

Capítulo 17

power.s

- módulo fuente *power.s*

```
/*
        PURPOSE: Program to illustrate how functions work
        This program will compute the value of  x elevado a y
        OUTPUT : echo $?
        Registros:
                r13: puntero de pila

        DEBUGGER COMMANDS
        (gdb) x /a (char **)($esp+8) -> doble puntero ya que esp+8 es un puntero que
            apunta a un string. string es un puntero a          caracter
        0xffffd078:     0xffffd2a7   ---------> 0xfffd078 es la dirección de la pila. Es
            un puntero. Apunta a  los strings de la                línea de comandos
        0xffffd2a7 es el puntero al string del 2° argumento.
        (gdb) x /c *(char **)($esp+8) -> Dereferencio el primer puntero obtengo la
            dirección del string y x examina esa dirección
        con formato caracter,
        0xffffd2a7:     50 '2'
        (gdb) x /s *(char **)($esp+8)  -> Dereferencio el primer puntero obtengo la
            dirección del string y x examina esa dirección
        con formato string
        0xffffd2a7:      "2"
*/

      .section .data
      .section .text
      .globl _start
_start:

        @ string argument pointers
        ldr r2,[r13, #+8]      @ r0 tiene el contendio de la pila= dirección del string.
            argv[1]
        ldr r3,[r13, #+12]     @ r1 tiene el contendio de la pila= dirección del string.
            argv[2]
        @ fetch string indirect
        ldr r0,[r2]        @ indirección para acceder al string referenciado por argv[1]
        ldr r1,[r3]        @/g indirección para acceder al string referenciado por argv[1]
        @ pongo a cero los dos bytes mas significativos del ASCII de 4 bytes
        mov r0,r0,LSL #16
        mov r0,r0,LSR #16
        mov r1,r1,LSL #16
        mov r1,r1,LSR #16
        @ convert ascii numbers to values
```

```
        sub r0,#0x30
        sub r1,#0x30
        bl  power           @ call    the function

        /* exit syscall */

        mov r7, #1
        swi #0

/*
 #PURPOSE: This function is used to compute the value of a number raised to a power.
 #
 #INPUT:        First argument - the base number
 #              Second argument - the power to raise it to
 #
 #OUTPUT:       Will give the result as a return value
 #
 #NOTES:        The power must be 1 or greater
 #
 #VARIABLES:
 #              r0 - holds the base number and the return value
 #              r1 - holds the power
 #              r2 - multiplicación parcial
 #
*/

power:
        mov     r2,r0           @ inicializo el resultado parcial con la base
power_loop_start:
        cmp     r1,#1           @ si la potencia ha llegado a 1 , fin de bucle.
        beq     end_power
        mul     r2,r2,r0        @ multiplico la base r0 por el temporal r2
        sub     r1,r1,#1        @ decrease the power
        bal     power_loop_start @ siguiente factor
end_power:
        mov     r0,r2           @ valor de retorno en r0
        bx lr
.end
```

17.1. Cuestiones

- Analizar el código fuente e interpretarlo.