IS-603 Arquitectura de Computadoras

I Parcial - Introducción a QtARMSim y ARM Thumb

IS-UNAH

I PAC 2024

Contenido

Introducción a QtARMSim

Lenguaje ensamblador ARM

La arquitectura ARM proporciona dos **juegos de instrucciones** diferenciados.

Un juego de instrucciones estándar (32 bits), y un juego de instrucciones reducido, llamado **Thumb**, en el que la mayoría de las instrucciones ocupan **16 bits**.

Por otro lado, se puede optar por dos convenios, el de ARM y **el de GNU**, que describen sintaxis similar de instrucciones, pero contexto del programa en ensamblador totalmente diferente.

ARM Thumb

El código máquina es el lenguaje que entiende el procesador (**sistema numérico binario**). Un programa en código máquina es un conjunto de instrucciones máquina, y datos, con los que el procesador realiza alguna tarea.

Notación RTL (Register Transfer Languaje) describe operaciones de manera genérica, es decir, independiente de la arquitectura.

Es similar a un pseudocódigo.

RTL vs Ensamblador

r3 ← r1 + r2

Notación RTL

add r3, r2, r1

ARM Thumb

0001100010001011

Código máquina

add $r3, $r1, $r2

Arquitectura MIPS

Particularidades de Ensamblador GNU para ARM

1. **Comentarios:** de línea #, y @, /* */ de bloque. Se prefiere @
2. **Seudoinstrucciones:** No pueden codificarse en lenguaje máquina, pero sí se traducen a una instrucción o secuencia de éstas, que realice la operación correspondiente.
3. **Etiquetas:** es una referencia a la dirección de memoria de contenido de la línea que señalan. Cuando el ensamblador encuentra la etiqueta, la reemplaza con un valor numérico que puede ser la dirección de memoria guardada, o un desplazamiento relativo a esa dirección.

Particularidades de Ensamblador GNU para ARM

4 **Directivas:** son indicadores para informar al ensamblador sobre cómo interpretar el código fuente.

En resumen, una instrucción en lenguaje ensamblador GNU para ARM sigue la forma general:
Etiqueta: operación oper1, oper2, oper3 @ Comentario

Instrucciones en lenguaje ensamblador ARM Thumb

```
1        .text
2  main:   mov r0, #0        @ Total a 0
3          mov r1, #10       @ Inicializa n a 10
4  loop:   mov r2, r1        @ Copia n a r2
5          mul r2, r1        @ Almacena n al cuadrado en r2
6          mul r2, r1        @ Almacena n al cubo en r2
7          add r0, r0, r2    @ Suma r0 y el cubo de n
8          sub r1, r1, #1    @ Decrementa n en 1
9          bne loop          @ Salta a loop si n != 0
10 stop:   wfi
11         .end
```

Ensamblar el código en QtARMSim, ver la estructura del código y los datos en registro.

En la línea 7 del código anterior, ¿qué modo(s) de direccionamiento se empleará(n)?

En la línea 7 del código anterior, ¿qué modo(s) de direccionamiento se empleará(n)?

Direccionamiento directo a registro.

En la línea 8 del código anterior, ¿qué modo(s) de direccionamiento se empleará(n)?

En la línea 8 del código anterior, ¿qué modo(s) de direccionamiento se empleará(n)?

Direccionamiento directo a registro y Direccionamiento inmediato.

Estructura de QtARMSim

Modo de simulación

```
@@ ----------------------------------------
@@ DISASSEMBLED CODE STARTING AT 0x00180000
@@ ----------------------------------------
[0x00180000] 0x2002 movs r0, #2          ;   2 main:       mov r0, #2
[0x00180002] 0x2103 movs r1, #3          ;   3             mov r1, #3
[0x00180004] 0x1842 adds r2, r0, r1      ;   4             add r2, r0, r1
[0x00180006] 0xBF30 wfi                  ;   5 stop:       wfi
[0x00180008] 0x0000 movs r0, r0
```

Identificar las partes del código desensamblado, las instrucciones codificadas y las direcciones de memoria ROM.

Representación de valores

Los registros admiten ingresar valores desde la interfaz. Los prefijos son importantes para indicar el sistema de numeración utilizado:

1. 3 en Decimal (sin prefijo)
2. 03 en Octal
3. 0x3 en Hexadecimal
4. 0b11 en Binario

¿Qué valor vemos en r2 después de escribir 0b11 y presionar enter?

Literales y constantes de ensamblador de ARM

Literales

Un literal es un número (dec, oct, hex bin), un carácter o una cadena de caracteres que se indica como tal, en el programa ensamblador.

Los prefijos de número son idénticos a los mencionados para cambiar valor en los registros.

Para definir un literal, se antepone el símbolo «#», ejemplo #10

Ejemplo de literales

```
1         .text
2  main:    mov r1, #0x1E
3           mov r2, #036
4           mov r3, #0b00011110
5  stop:    wfi
```

¿A qué valor en decimal equivale lo que se almacenará en r3?
¿En qué sistema de numeración se escribió el literal de la línea 3?

Constantes y cambio de nombre a registros

Directivas para constantes (.equ, .equiv, .eqv)

```
.equ constant, value
```

Directivas para registros

```
day .req r7
.unreq r7
```

Ejemplo de directivas para constantes y registros

```
1           .equ Monday, 1
2           .equ Tuesday, 2
3           @ ...
4
5           .text
6           day .req r7
7   main:   mov day, #Monday
8           mov day, #Tuesday
9           .unreq day
10          @ ...
11  stop:   wfi
```

Al finalizar el programa, ¿qué valor queda almacenado en r7?

Inicialización de datos y reserva de espacio

Generalidades

En un programa de lenguaje de alto nivel es el compilador (o intérprete) el que se encarga de hacer la reserva de espacio en memoria (principal) para los diferentes tipos de datos que se le indican.

En caso de un programa en ensamblador, sí debe indicarse el espacio que tendrá cada uno de los datos que se almacenarán, e inicializarlos si fuese el caso.

Tamaño de palabra en ARM

1. La unidad básica para medir la memoria es el **byte**
2. El tamaño de palabra define bloques de almacenamiento en memoria, **en el caso de ARM la palabra se define con el tamaño de 4 bytes**, y los datos se alinean en posiciones de memoria múltiplos de 4.
3. En ARM, se puede trabajar también con **medias palabras** y **palabras dobles**, con direcciones de memoria **múltiplo de 2 y de 4**, respectivamente.

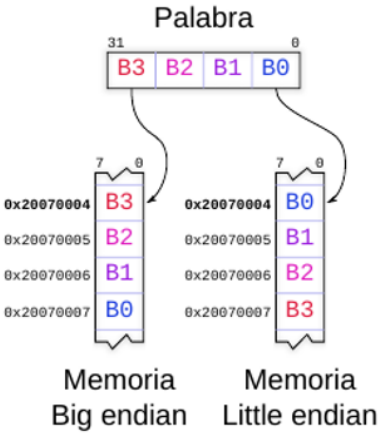
Para definir reservas de espacio de memoria se utiliza la directiva .data

Ejemplo de directivas para constantes y registros

```
1           .data       @ Comienzo de la zona de datos
2  word1:   .word 15
3  word2:   .word 0x15
4  word3:   .word 015
5  word4:   .word 0b11
6                      @ fin de la zona de datos
7           .text
8  stop:    wfi
```

Observar que los bytes "no utilizados"quedan en 0.

Organización de datos en memoria



Observar en qué dirección de memoria se almacena el Byte menos significativo de la palabra.

Otras directivas de inicialización

- `.byte <value8>`
- `.hword <Value16>`
- `.quad <Value64>`
- `.ascii "cadena" → UTF-8 codes`
- `.asciz "cadena"→ agrega 0 al final de la cadena`
- `.space N → reserva N bytes e inicializa en 0`

Alineación de datos en memoria

.balign N

Indica que el siguiente dato debe almacenarse en una dirección de memoria múltiplo de N

.align N

Indica que el siguiente dato debe almacenarse en una dirección de memoria múltiplo de 2^N

Ejemplo

```
1           .data
2  byte1:   .byte 0x11
3  space:   .space 4
4  byte2:   .byte 0x22
5  char1:   .ascii "aAbBcC"
6  word:    .word 0xAABBCCDD
7  hword:   .hword 0b00001101
8  char2:   .asciz "Hola Arqui"
9
10          .text
11 @main:   mov r0, #0b1111
12 stop:    wfi
13          .end
```

¿A qué dirección apunta cada etiqueta? A la dirección en la que fue almacenado el dato.

Ejercicios

Desarollar los siguientes ejercicios:

1. 2.12
2. 2.18
3. 2.19
4. 2.20
5. 2.24 (obviar la parte de mostrar datos en la LCD)

Habrá un enlace en la plataforma para que puedan subir un .zip que contenga los archivos .s (1 .s por cada ejercicio).