

Práctica 4. Programación en ensamblador

(Duración: 1 sesión)

Introducción

En esta práctica nos familiarizaremos con la programación en ensamblador del MIPS usando el entorno de desarrollo MPLAB X IDE y el compilador XC32.

Objetivos

Al concluir la práctica, el alumno deberá ser capaz de:

- Generar un listado en ensamblador de un programa en C para ver el código generado por el compilador.
- Insertar instrucciones en ensamblador en línea dentro de un programa en C.
- Escribir funciones en ensamblador para ser llamadas desde C.

Trabajo previo

Antes de ir al laboratorio ha de:

- Leer detenidamente la práctica, anotando las dudas que le surjan para preguntárselas al profesor.
- Escribir los programas y funciones solicitados en las secciones 3 y 4.

1. Crear un proyecto

En primer lugar ha de crear un proyecto para esta práctica, siguiendo los pasos indicados en el apartado 2 de la primera práctica. No olvide trabajar en la carpeta `D:\Micros\Grupo_XX`, creada en la práctica 1.


Al igual que en la práctica 3, es necesario configurar la tarjeta para que funcione con el oscilador de cuarzo externo. Como recordará basta con incluir en el proyecto el archivo `Pic32Ini.c` disponible en Moodle.

Por otro lado asegúrese de que los *jumper*s JP1 y JP3 están conectados, ya que en caso contrario no funcionarán ni el pulsador ni los LEDs.

2. Inspección del código máquina generado por el compilador

En este apartado vamos a ver cómo inspeccionar el código máquina generado por el compilador. Para ello vamos a partir del programa que enciende el LED RC0 cuando se pulsa el pulsador RB5 realizado en la práctica 2, que se incluye a continuación para mayor comodidad:

```
1  #include <xc.h>
2  #define PIN_PULSADOR 5
3
4  int main(void)
5  {
6      int pulsador;
7
8      TRISC = XX; // A completar por el alumno (trabajo previo Práctica 2)
9      LATC  = XX; // A completar por el alumno (trabajo previo Práctica 2)
10     TRISB = XX; // A completar por el alumno (trabajo previo Práctica 2)
11
12     while(1){
13         // Se lee el estado del pulsador
14         pulsador = (PORTB>>PIN_PULSADOR) & 1;
15         if(pulsador == 0){
16             LATC &= ~1;
17         }else{
18             LATC |= 1;
19         }
20     }
21 }
```

Incluya el programa anterior en el proyecto y seleccione **Debug** > **Debug Main Project** o pulse el botón  para compilar el programa en modo *debug* y cargarlo en el microcontrolador. Una vez hecho esto, está disponible el listado con el código máquina generado por el compilador sin más que seleccionar **Window** > **Debugging** > **Output** > **Dissassembly Listing File**.

El listado contiene las instrucciones en C del archivo fuente junto con el código máquina generado a partir de cada línea de código en C, tal como se muestra a continuación:

```
1:                #include <xc.h>
2:
3:                #define PIN_PULSADOR 5
4:
5:                int main(void)
6:                {
9D0000DC  27BDFFF0  ADDIU SP, SP, -16
9D0000E0  AFBF000C  SW S8, 12(SP)
9D0000E4  03A0F021  ADDU S8, SP, ZERO
```

El archivo consta de tres columnas: la primera contiene o bien el número de línea del código fuente en C o bien la dirección de la instrucción de código máquina. La segunda columna contiene el código máquina y la tercera o bien la instrucción en C o bien la instrucción en ensamblador.

2.1. Ejercicio

A partir del listado generado en el apartado anterior, elija la primera instrucción codificada como tipo-R y analice cada uno de los campos, para comprobar que el ensamblador ha hecho bien su trabajo. Repita el proceso para la primera instrucción de tipo-I y la primera de tipo-J. En Moodle encontrará el documento “*The MIPS32 Instruction Set Manual*” donde se detalla cómo se codifica cada una de las instrucciones del MIPS.

3. Ensamblador en línea

Si observa el listado generado en el apartado 2 verá que el código no está muy optimizado que se diga.¹ En ocasiones, cuando se quiere optimizar un programa, se pueden buscar los bucles que se ejecutan más frecuentemente y, a partir del código generado por el compilador, intentar arañar unos cuantos ciclos en cada iteración recodificando estas instrucciones a mano. Por ejemplo, en el código del apartado 2, la instrucción `LATC &= ~1;` se traduce a ensamblador de la siguiente forma:

```

19:                                LATC &= ~1;
9D00012C  3C02BF88    LUI V0, -16504
9D000130  8C426230    LW V0, 25136(V0)
9D000134  2404FFFE    ADDIU A0, ZERO, -2
9D000138  00441824    AND V1, V0, A0
9D00013C  3C02BF88    LUI V0, -16504
9D000140  AC436230    SW V1, 25136(V0)
20:                                }else{

```

Si se fija, en la primera instrucción carga la dirección base de los registros de periféricos en `v0` mediante la instrucción `lui`. En la siguiente instrucción sobrescribe este valor para cargar el valor de `LATC`, por lo que en la penúltima instrucción ha de volver a cargar mediante `lui` el mismo valor en `v0`. Así, si en lugar de usar `v0` en la segunda instrucción usamos `v1`, nos podemos ahorrar la penúltima instrucción. Para ello, podemos cambiar el código del apartado 2 por el siguiente:

```

1  #include <xc.h>
2  #define PIN_PULSADOR 5
3
4  int main(void)
5  {
6      int pulsador;
7
8      TRISC = XX; // A completar por el alumno (trabajo previo)
9      LATC  = XX; // A completar por el alumno (trabajo previo)
10     TRISB = XX; // A completar por el alumno (trabajo previo)
11
12     while(1){
13         // Se lee el estado del pulsador
14         pulsador = (PORTB>>PIN_PULSADOR) & 1;
15         if(pulsador == 0){
16             //LATC &= 1;
17             asm(" lui $v0, 0xBF88"); // 0xBF88 = -16504
18             asm(" lw  $v1, 25136($v0)");
19             // A completar por el alumno (trabajo previo)
20         }else{
21             LATC |= 1;
22         }
23     }
24 }

```

En donde deberá completar las instrucciones que faltan mediante directivas `asm` y comprobar que el programa sigue funcionando igual que en el apartado anterior.

¹La versión gratuita del compilador XC32 es así. Si quiere un compilador que optimice, es necesario adquirir la versión PRO, que cuesta 24,60 € al mes.

Tenga en cuenta que el compilador es un poco caprichoso y dentro de estas directivas los registros han de estar precedidos por el carácter \$. Por el contrario, al escribir una función en ensamblador en su propio archivo, que es lo que haremos en el siguiente apartado, los registros han de escribirse sin el \$. Por otro lado, aunque en el listado generado aparece en la primera línea un -16504, éste se ha traducido a su valor hexadecimal (0xBF88), ya que el compilador no acepta números negativos en la instrucción lui.

4. Función para generar retardos en ensamblador

Escriba una función en ensamblador para generar un retardo usando el temporizador 2. Dicha función tendrá el siguiente prototipo:

```
void Retardo(uint16_t retardo_ms);
```

en donde retardo_ms es el retardo deseado en milisegundos.

Para generar el retardo se configurará el temporizador 2 para generar un retardo fijo de 1 ms y mediante un bucle se repetirá la espera del fin del temporizador las veces que indique el parámetro retardo_ms.

Una vez escrita la función, se escribirá un programa principal para hacer parpadear el LED RC0 con una frecuencia de 1 Hz usando dicha función.

Para escribir la función en ensamblador tenga en cuenta las siguientes consideraciones:

- Tenga en cuenta la convención de uso de registros del MIPS, en particular que el primer parámetro de la función se recibe en a0, el segundo en a1, el tercero en a2 y el cuarto en a3. Si hay más parámetros, éstos se pasan por la pila. Recuerde también que puede usar sin ningún problema los registros temporales (t0 a t9) pero si desea usar los registros s0 a s7 ha de guardarlos en la pila antes de modificarlos y restaurarlos antes de retornar de la función.
- La función en ensamblador se llama desde C igual que una función escrita directamente en C. Eso sí, es necesario crear un prototipo de la función en ensamblador, preferiblemente en un archivo denominado Retardo.h.
- El archivo que contiene la función en ensamblador ha de llamarse Retardo.S. La S mayúscula es importante, pues así el compilador reconoce los includes de C (xc.h) donde se definen los registros de los periféricos. A pesar de ello el entorno de desarrollo MPLAB X se niega a ponerla por defecto, así que tendrá que cambiar el nombre a mano desde el sistema operativo.
- A continuación se muestra una plantilla para la escritura de la función en ensamblador:

```
#include <xc.h> // Define los registros de los periféricos

.text # sección de código

    # Se define el nombre de la función global para poder llamarla desde C
.global Retardo
.ent Retardo # Introduce el símbolo Retardo en el código para depuración
Retardo:
    # Escriba aquí el código de la función

.end Retardo # Indica el final de la función (para depuración)
```