Práctica 2

Introducción al ensamblador

Introducción

Una vez que sabemos hacer uso de *OpenOCD* para gestionar nuestro sistema empotrado, ya podemos empezar a desarrollar código para nuestra plataforma. Para ello, el primer paso consiste en conocer adecuadamente el lenguaje ensamblador de la CPU de nuestra plataforma.

Objetivos

- Familiarizarse con el lenguaje ensamblador de ARM.

Primeros pasos en ensamblador

Dado que en las lecciones 5 y 6 de teoría se explica en detalle la arquitectura del ARM7TDMI y su repertorio de instrucciones, en esta parte de la práctica se proponen una serie de ejercicios para realizar una primera toma de contacto con la *Econotag* en ensamblador. Para ello partiremos del código del *"Hola Mundo"* usado en el seminario 2 para comprobar la correcta instalación de las herramientas.

Por otro lado, como de momento no disponemos de ningún entorno de ejecución del lenguaje *C*, nuestros programas tendrán que diseñarse para su ejecución sin el uso de una pila, por lo que no podremos usar llamadas a funciones anidadas ni variables locales. Más adelante, y una vez que estemos familiarizados con las *binutils,* pasaremos a diseñar un *Runtime* de *C* para poder hacer un uso completo de nuestra plataforma.

Directivas

Las directivas son todas aquellas órdenes que se escriben en un fichero fuente y que no son para que las ejecute el procesador en tiempo de ejecución, sino más bien para indicarle al ensamblador cómo hacer la traducción del código o al enlazador cómo generar la imagen ejecutable. Por tanto, se usan directivas para definir las diferentes secciones de un ejecutable (para datos, código, etc.), el tipo de código que se debe generar (ARM o *Thumb),* para definir símbolos, para definir el ámbito y el tipo de los símbolos (externo, global, función, entero, cadena, etc.), para forzar alineamiento de determinadas partes de nuestro código o datos, etc. Todas las directivas empiezan con un punto. Se pueden consultar en el manual del ensamblador de GNU[1].

Echando un vistazo al fichero `hello.s` del seminario 2, podemos hacernos una idea de las directivas básicas:

- `.set`: Definición de símbolos que podrán ser usados más tarde en nuestro programa.
- `.text`: Indica que a continuación hay instrucciones, por lo que se deben emitir a la sección de código del ejecutable final.

1 https://sourceware.org/binutils/docs-2.23.1/as/

- `.data`: Indica que lo que hay a continuación son datos en memoria, por lo que se deben emitir a la sección de datos del ejecutable final. Esta directiva no aparece en `hello.s`, ya que sólo usa los registros de la CPU (no usa variables en RAM).
- `.global`: Modifica el ámbito de un símbolo para que sea global. De esta forma, otros ficheros objeto de nuestro programa, o bien el enlazador, podrán acceder a dicho símbolo.

Además de las directivas estándar del ensamblador de GNU, cada arquitectura a la que se ha portado la *toolchain* de *GNU* añade características específicas para dicha arquitectura[2]. En nuestro caso, tenemos la directiva:

- `.code`: Indica qué tipo de código máquina hay a continuación, código ARM de 32 bits o código *Thumb* de 16 bits.

Etiquetas

Son símbolos acabados en el carácter ':'. Se usan para etiquetar partes del programa (variables, funciones, destinos de saltos, etc.), y básicamente consisten una forma de asignar un nombre a la dirección de memoria que tendrá dicha parte de nuestro programa. Dado que las direcciones finales de cada parte del ejecutable se calculan por el enlazador, es conveniente asignar etiquetas a aquellas partes del programa a las que queramos acceder desde ensamblador.

Por defecto el ámbito de las etiquetas que declaremos en un fichero es local, es decir, sólo serán visibles dentro de dicho fichero. Si deseamos que una etiqueta, y por ende la parte del programa a la que hace referencia, sea accesible desde otros ficheros de nuestro programa, deberemos declararla global.

Una etiqueta importante es `_start:`, que suele usarse para indicar el punto de entrada de nuestro ejecutable y que debe ser declarada global para que el enlazador pueda saber dónde está declarada.

Programación estructurada

Si ya es importante estructurar un programa en diferentes funciones cuando se programa en un lenguaje de alto nivel, lo es todavía más cuando se programa en ensamblador. Dado el bajo nivel de abstracción en el que estamos trabajando, es necesario dividir el código en pequeñas funciones que hagan una tarea determinada, como por ejemplo inicializar el GPIO, forzar una pausa mediante espera ocupada, detectar una pulsación de un botón, etc., y comprobar el correcto funcionamiento de cada una de estas funciones por separado mediante un pequeño programa que las llame y verifique su resultado antes de integrar dichas funciones en el programa principal. De esta forma el desarrollo de nuestro programa se basará en la invocación de funciones que previamente se han testado y funcionan, por lo que en caso de que el resultado no sea el correcto, será relativamente sencillo aislar el fallo y corregirlo. Para ello, se recomienda que el alumno haya seguido las instrucciones dadas en el seminario 2 acerca de la depuración remota mediante usando *OpenOCD*, *gdb* y el IDE *Eclipse*.

Ejercicios

El objetivo final de esta serie de ejercicios es modificar el fichero `hello.s` para poder cambiar el led que parpadea en la *Econotag* en función del botón que se pulse en la placa. El resultado final será un programa que siempre debe hacer parpadear un led de la placa. Por defecto, al iniciarse el

2 https://sourceware.org/binutils/docs-2.23.1/as/ARM_002dDependent.html

programa, comenzará a parpadear el led rojo. Mediante pulsaciones de los botones el usuario podrá cambiar el led que está parpadeando del rojo al verde y viceversa. La pulsación del botón S3 provocará que el deje de parpadear el led rojo y pase a parpadear el verde, mientras que el botón S2 volverá a dejar la placa en su estado por defecto, es decir, el led verde apagado y el rojo parpadeando.

Para conseguir nuestro objetivo vamos a ir acometiendo una serie de cambios incrementales que darán como resultado el resultado esperado.

Declaración de los símbolos

Dado que vamos a manejar los dos leds y los botones S2 y S3 de la placa, el primer paso consistirá en añadir al principio del programa los símbolos necesarios para acceder de forma cómoda a estos dispositivos mediante el GPIO. Para ello necesitaremos símbolos para almacenar las direcciones de algunos registros del GPIO y las máscaras para acceder a los pines a los que están conectados los leds y los botones.

En el esquemático de la placa[3] podemos comprobar que usaremos los pines GPIO22, GPIO23, GPIO26, GPIO27, GPIO44, GPIO45. Por otro lado, si miramos en el capítulo 11 del manual del procesador de la placa podremos comprobar que, dado que disponemos de 64 pines controlados por el GPIO y que el tamaño de los registros es de 32 bits, el controlador GPIO está organizado en parejas de registros (`GPIO_PAD_DIR0` y `GPIO_PAD_DIR1`, `GPIO_DATA_SET0` y `DATA_SET1`, etc.), ocupándose los registros `GPIO_XXX0` de los primeros 32 pines GPIO y los registros `GPIO_XXX1` del resto. Como los botones están conectados al primer grupo de pines y los leds al segundo, necesitaremos usar los siguientes registros:

- `GPIO_PAD_DIR0` para fijar la dirección de los pines que gestionan los botones.
- `GPIO_PAD_DIR1` para fijar la dirección de los pines que gestionan los leds.
- `GPIO_DATA0` para consultar si se ha pulsado algún botón.
- `GPIO_DATA_SET0` para inicializar los GPIO de los botones.
- `GPIO_DATA_SET1` para encender los leds.
- `GPIO_DATA_RESET1` para apagar los leds.

Inicialización del GPIO

Para poder manejar los botones y los leds desde nuestro programa es necesario que configuremos los pines a los que están conectados adecuadamente. Para ello diseñaremos una función llamada `gpio_init`, que será llamada justo al comienzo de nuestro programa (etiqueta `_start:`) y que realizará las siguientes operaciones:

- Fijar los pines a los que están conectados los leds como de salida, teniendo en cuenta que las líneas del programa original que fijaban la dirección del pin del led rojo pasarán ahora a ester incluidas en la nueva función `gpio_init`.
- Cada botón está conectado a dos pines, de forma que cuando se pulsa, causa un cortocircuito. Para poder detectar las pulsaciones lo más sencillo es fijar uno de los pines como de salida con un valor de salida estable de 1 y el otro pin como de entrada, de forma

3 https://github.com/malvira/econotag

que cuando se lea el pin de entrada se obtendrá un 0 si el botón no está pulsado o un 1 si el botón ha sido pulsado.

Detección de pulsaciones en los botones

Una vez configurado el GPIO es bastante sencillo detectar el estado de los botones. Simplemente hay que leer el registro de datos `GPIO_DATA0` y ver si alguno de los pines de entrada de los botones está activo. Por tanto, para terminar nuestro programa diseñaremos una función llamada `test_buttons`, que será llamada dentro del cuerpo del bucle principal, y que en función del estado de los botones fijará el valor del registro `r5` al de la máscara del led que haya que encender en función del botón pulsado. Dado que de momento no tenemos *runtime* de C (no tenemos variables, ni locales ni globales), reservaremos el registro `r5` exclusivamente para mantener la máscara del led que debe parpadearse en cada momento.

Como el estado de los botones sólo se consulta cuando se llama a esta función, es conveniente llamarla un par de veces en el cuerpo del bucle para tener más posibilidades de detectar la pulsación de los botones. Por ejemplo, justo antes de la instrucción que enciende el led y justo después de la instrucción que apaga el led. De esta forma, testamos el estado de los botones una vez entre cada llamada a la función `pause`.