Práctica 3

Introducción al enlazador y el resto de las binutils

Introducción

Una vez que sabemos escribir pequeños programas para nuestra plataforma, en esta práctica aprenderemos a crear variables globales y a gestionar el mapa de memoria de nuestro sistema y a mapear nuestras aplicaciones adecuadamente a la plataforma mediante el *script* de enlazado y el enlazador. También introduciremos el uso de otras *binutils* que nos ayudarán a comprobar si nuestra imagen ejecutable se ha construido correctamente.

Objetivos

- Saber cómo se define y se usa una variable global
- Comprender la diferencia entre el uso de variables y constantes, así como las ventajas e inconvenientes de cada una de las opciones a la hora de gestionar datos en un programa
- Familiarizarse con el enlazador de GNU.
- Conocer el resto de las *binutils* de GNU.

Primeros pasos con el enlazador y el resto de binutils

Dado que en la lección 7 se detalla cómo particularizar un *script* de enlazado para cualquier plataforma, en esta parte de la práctica vamos a aplicar los contenidos de dicha lección para modificar el *script* de enlazado del *"Hola Mundo"* para poder gestionar más regiones de memoria en nuestra plataforma y secciones en nuestros ejecutables. Además entraremos en contacto con una serie de *binutils* que nos permitirán comprobar si las modificaciones hechas en el *script* de enlazado se han llevado a cabo correctamente.

La tabla de símbolos

Informa sobre todos los símbolos definidos en el ejecutable, indicando su dirección, tipo, tamaño, etc. Se puede consultar mediante las *binutils nm, readelf* u *objdump*. Es bastante útil para comprobar que las variables están definidas en la dirección correcta, que tengan un alineamiento adecuado, etc.

La lista de secciones del ejecutable

Las utilidades *objdump* y *readelf* nos pueden listar la tabla de secciones, en la que podemos comprobar las secciones que se han generado en nuestro ejecutable, sus direcciones física y virtual, su tamaño y su alineación. Es de utilidad para comprobar que cada sección se ha asignado al módulo de memoria adecuado.

Desensamblado

La utilidad *objdump* permite desensamblar un fichero ELF, los que nos permitirá observar cómo ha traducido el compilador nuestro programa y qué direcciones que se han asignado a cada instrucción máquina. Si mediante el *script* de enlazado se han asignado distintas secciones a diferentes funciones, podremos comprobar si la asignación se ha realizado correctamente.

Generación de la imagen

La utilidad *objcopy* permite extraer del fichero ELF la imagen ejecutable de nuestro *firmware*, descartando toda la información de depuración, tabla de símbolos, cabeceras, etc.

Ejercicios

El objetivo final de esta serie de ejercicios es modificar el *script* de enlazado de la practica anterior para soportar el uso de variables globales y de más secciones en nuestra aplicación. Para ello, también tendremos que modificar el fichero `hello.s` para que haga uso de las nuevas secciones y símbolos definidos en el *script* de enlazado.

Dado que los cambios que hagamos en el *script* de enlazado afectarán a la asignación de direcciones de memoria de las secciones y los símbolos de nuestra imagen ejecutable, también comentaremos cómo hacer uso de algunas *binutils* que nos permitirán comprobar que los cambios se han realizado correctamente.

El mapa de memoria del sistema

Si miramos en el capítulo 4 del manual de usuario del *Freescale* MC13224V podemos encontrar el mapa de memoria del sistema. Observando el *script* de enlazado, podemos comprobar que sólo se ha tenido en cuenta la región de memoria RAM0, ya que es más que suficiente para alojar nuestro programa *"Hola Mundo"*. Modificar el *script* de enlazado para añadir las regiones de memoria RAM1, RAM2 y RAM3. De momento no necesitaremos usar más regiones de memoria, ya que estamos cargando el programa directamente en la RAM mediante *OpenOCD*.

Para comprobar que hemos incluido correctamente la definición de las regiones de memoria, podemos definir las secciones `.ram1`, `.ram2` y `.ram3`, asignando cada una de ellas las regiones de memoria RAM1, RAM2 y RAM3 respectivamente. Dentro de cada una de estas secciones definiremos los símbolos `s1`, `s2` y `s3`, apuntando cada uno de ellos al comienzo de su respectiva sección. Una vez hechos estos cambios, compilar el programa y comprobar en la tabla de símbolos del ejecutable que los símbolos `s1`, `s2` y `s3` tienen la dirección de inicio de cada una de las regiones de memoria.

Una vez hecha la comprobación en la tabla de símbolos de que las direcciones de `s1`, `s2` y `s3` son `0x00402000`, `0x00408000` y `0x00410000`, podemos estar seguros de que las regiones de memoria están bien definidas, por lo que podemos borrar las secciones `.ram1`, `.ram2` y `.ram3` del *script* de enlazado.

Uso de variables globales

Hasta ahora nuestro programa sólo ha hecho uso de los registros de la CPU. Esto es claramente insuficiente para implementar una aplicación real. Además, cualquier lenguaje de alto nivel permite definir variables globales, por lo que es conveniente que dotemos a nuestro *script* de enlazado de la

posibilidad de usarlas en nuestra plataforma. Para ello, definiremos la sección `.data` en el *script* de enlazado para que recoja todas las secciones `.data` de cualquier fichero objeto que se enlace para formar el ejecutable final. Asignaremos la sección `.data` a la región de memoria RAM1.

Una vez definida dicha sección en el *script* de enlazado, podemos editar nuestro programa `hello.s` para que las máscaras necesarias para acceder a los botones y a los leds, así como el retardo del parpadeo, estén almacenadas en variables globales en vez de estar codificadas mediante constantes. Para ello será necesario definir una sección `.data` en `hello.s`, definir una etiqueta para cada una de las variables globales, y almacenar en cada variable su máscara correspondiente (mediante la directiva `.word`). Dado que estas variables sólo se van a usar desde el fichero `hello.s`, no es necesario declarar sus etiquetas como globales.

Una vez hecha esta modificación es importante comprender la diferencia entre el uso de constantes y variables. En ambos casos se usan símbolos, ya que, a fin de cuentas, tanto en el *linker script* como en un fichero de código, un símbolo es simplemente una etiqueta que asignamos a un valor, para que la programación sea más sencilla. La diferencia está en que en el caso de usar constantes, el símbolo será reemplazado a la hora de enlazar por el valor de la constante. Sin embargo, en el caso de las variables, el símbolo no representa al valor de la variable, sino a la dirección de memoria en la que dicha variable está almacenada (concepto de puntero en C), por lo que para poder acceder al contenido de la variable será necesario el uso de una instrucción de transferencia de memoria para traer (o llevar) el contenido de la variable hacia (o desde) el procesador. Por tanto, el acceso al contenido de una variable siempre va a suponer más tiempo (se necesita hacer una transferencia de memoria) que el uso directo de una constante, aunque, por otro lado, el uso de variables permite que el programador pueda cambiar su contenido en cualquier momento, cosa que no ocurre con los símbolos, ya que al estar definidos en los ficheros fuente, no se pueden cambiar una vez que se ha construido el ejecutable.

Gestión de mapa de E/S desde el *script* de enlazado

Dado que en nuestro sistema la entrada/salida está mapeada a memoria, y que el enlazador es la herramienta que asigna direcciones de memoria a los símbolos, vamos modificar el *script* de enlazado para definir símbolos que faciliten la gestión del GPIO. La idea es definir un símbolo por cada registro del GPIO que vayamos a usar y asignarle a dicho símbolo la dirección de memoria en la que está mapeado su registro asociado. Estos símbolos serán globales y accesibles desde cualquier parte de nuestra aplicación (tanto desde ensamblador como desde *C*), de forma que podremos usar dicho símbolo para acceder a su registro de E/S asociado.

Para ello definiremos los siguientes símbolos en el *script* de enlazado:

- `gpio_base = 0x80000000;`
- `gpio_pad_dir0 = gpio_base + 0x00000000;`
- `gpio_pad_dir1 = gpio_base + 0x00000004;`
- `gpio_data0 = gpio_base + 0x00000008;`
- `gpio_data_set0 = gpio_base + 0x00000048;`
- `gpio_data_set1 = gpio_base + 0x0000004c;`
- `gpio_data_reset1 = gpio_base + 0x00000054;`

La definición de estos símbolos se puede hacer de forma absoluta (fuera de cualquier sección de memoria), o bien podemos declarar una región de memoria que se llame `gpio` a partir de la

dirección `0x80000000`, asignar una sección de memoria a dicha región, y declarar los símbolos dentro de la sección. Se recomienda hacerlo de las dos formas para comprobar que el resultado es el mismo.

Ahora sólo nos falta modificar nuestro programa para que haga uso de estos símbolos en vez de los definidos previamente mediante la directiva `.set`. De hecho, si hemos resuelto todos los ejercicios satisfactoriamente, podemos borrar todas las directivas `.set` de `hello.s` y el programa debería seguir funcionando.

Obtención de la imagen ejecutable

Generar la imagen en formato binario de nuestra aplicación. Comprobar que su tamaño es bastante mayor que el del fichero ELF. Al colocar el código y los datos separados, en las regiones de memoria RAM0 y RAM1 respectivamente, el enlazador, una vez que ha colocado el código a partir de la dirección `0x00400000`, rellena con ceros la imagen hasta llegar a la dirección `0x00402000`, dirección de inicio de la región RAM1, en la que coloca los datos. Podemos comprobarlo abriendo la imagen binaria con un editor binario (por ejemplo `ghex`).

Una vez comprobado que la separación de las secciones `.text` y `.data` en las regiones RAM0 y RAM1 ha generado una imagen demasiado grande, prácticamente rellena de ceros, se puede probar a cambiar el *script* de enlazado para colocar los datos en la región RAM0, justo a continuación del código, obteniéndose una imagen binaria significativamente más pequeña (unas 30 veces más pequeña), y por lo tanto más adecuada para un sistema empotrado.