

Introducción al entorno de desarrollo

El contenido de este documento ha sido publicado originalmente bajo la licencia:
Creative Commons License (<http://creativecommons.org/licenses/by-nc-sa/3.0>)
Prácticas de Estructura de Computadores empleando un MCU ARM by Luis Piñuel y
Christian Tenllado is licensed under a Creative Commons Attribution-NonCommercial-
ShareAlike 3.0 Unported License.



1.1. Uso de Eclipse para cross-compiling

En este laboratorio vamos a utilizar Eclipse como Entorno de Desarrollo Integrado (IDE). Es una aplicación con interfaz gráfica de usuario que nos permitirá desarrollar tanto en lenguaje ensamblador como en C, y depurar sobre un simulador de la arquitectura ARM o depurar en circuito sobre una placa con un procesador de ARM.

Eclipse nos ofrece principalmente la interfaz gráfica y la gestión de los proyectos. Para realizar el resto de tareas hace uso de otras herramientas de GNU: el ensamblador (`as`), el compilador (`gcc`), el enlazador (`ld`) y el depurador (`gdb`). Además, debemos tener en cuenta que desarrollamos sobre un PC para un entorno con procesador ARM, y por tanto necesitamos hacer *compilación cruzada*. Para distinguir las herramientas cruzadas de las nativas del PC suele añadírselas un prefijo que describe la arquitectura objetivo para la que se compila, en nuestro caso este prefijo es: `arm-none-eabi-`. Estas herramientas pueden utilizarse también directamente desde un interprete de línea de comandos (terminal en Linux o Mac OS X o `cmd` en Windows). En cualquier caso, debemos siempre emplear la sintaxis y reglas de programación propias de estas herramientas.

1.1.1. Creación de un proyecto en Eclipse

Para todas nuestras prácticas deberemos crear un proyecto Eclipse para compilación y depuración cruzadas utilizando el plugin *GNU ARM*. Para ello seguiremos los siguientes pasos:

1. Abrimos Eclipse haciendo doble click sobre el icono del escritorio.
2. Al iniciarse la aplicación nos mostrará una ventana de selección de `workspace`, como la que la Figura 1.1. Debemos seleccionar un `workspace` propio, que no es más que un directorio en el que Eclipse guardará información sobre los proyectos de todas nuestras prácticas. Para que el ordenador del laboratorio vaya lo más ágil posible, es conveniente que pongamos nuestro `workspace` en la carpeta `C:\hlocal`. Es importante también que siempre guardemos una copia de nuestro `workspace` en un lugar seguro, porque el directorio `hlocal` es local al puesto y común para todos los usuarios.
3. Una vez seleccionado se abrirá la ventana principal de Eclipse. Si acabamos de crear el `workspace` entonces el aspecto será como el que se muestra en la Figura 1.2.
4. Debemos cerrar la pestaña `Welcome` haciendo click en la cruz de la pestaña. Entonces la ventana quedará como la que muestra la Figura 1.3, es la perspectiva C/C++ de Eclipse, que está organizada de la siguiente manera:
 - Panel Izquierdo: el explorador de proyectos. Aparecerán todos los proyectos que tengamos en el `workspace`, cuando los tengamos.
 - Panel central: el editor . Nos permitirá editar los ficheros que contendrán el código fuente de nuestros programas.
 - Panel derecho. Nos permite explorar los símbolos del proyecto activo (funciones, variables, etc).

- Panel inferior. Tiene varias pestañas, entre las que destacan la pestaña de errores de compilación y la consola. Desde la primera podemos ver los errores y saltar a la línea de código fuente que los causó haciendo click sobre el error. En la segunda podemos ver los comandos que ejecuta Eclipse para hacer la compilación.
5. Para crear el proyecto seleccionamos **File→New→C Project**, con lo que se abrirá una ventana como la de la Figura 1.4. Como indica la figura, seleccionamos las opciones **ARM Cross Target Application**, **Empty Project** y **ARM GCC (Sourcery G++ Lite)**. Elegimos el nombre del proyecto y pulsamos **Finish**. Tendremos el proyecto vacío visible en el explorador de proyectos.
 6. Ahora vamos a añadirle un fichero con el código fuente de nuestro primer programa en ensamblador. Para ello seleccionamos **File→New→Source File**, con lo que se abrirá una ventana como la de la Figura 1.5. Para que Eclipse tome el fichero como un fichero fuente con código ensamblador le ponemos extensión **.asm** o **.S**¹. Una vez creado, haciendo doble click sobre el fichero en el panel izquierdo se abrirá una pestaña en editor, en la que copiamos el código del cuadro 1.
 7. Antes de configurar la compilación de nuestro proyecto vamos a añadir a él un fichero más, que servirá para indicar al enlazador cómo debe construir el mapa de memoria del ejecutable final. Aprovechamos para ver otra forma de añadir un fichero al proyecto. En el explorador del proyecto seleccionamos el proyecto y pulsamos el botón derecho del ratón, y seleccionamos **New→File**. Se abrirá una ventana como la de la Figura 1.6, seleccionamos el proyecto y ponemos **ld_script.ld** como nombre del fichero. Una vez creado, lo abrimos en el editor y copiamos el contenido del cuadro 2.
 8. Finalmente debemos configurar el proyecto para que la compilación se realice correctamente. Para ello seleccionamos el proyecto en el panel izquierdo, pulsamos el botón derecho del ratón y seleccionamos la entrada **Properties** en la parte inferior del desplegable. Con ello se abrirá una ventana como la que muestra la Figura 1.7. En esta ventana seleccionamos **C/C++ Build→Settings** y
 - Debemos comprobar que en **Target Processor** está seleccionado **arm7tdmi** como procesador, no están marcadas ninguna de las casillas **Thumb*** y está seleccionada la opción **Little Endian**.
 - Debemos seleccionar **ARM Sourcery GCC C Linker→General**, y en la casilla **Script file (-T)** debemos escribir la ruta al fichero **ld_script.ld** que hemos añadido al fichero. Podemos seleccionarlo gráficamente pulsando el botón **Browse**.
 9. Compilamos el proyecto. Para ello seleccionamos **Project→Build Project**. Acabado este paso habremos obtenido el ejecutable final, con extensión **.elf** (*Executable Linked Format*), que se encontrará en el subdirectorio **Debug**, dentro del directorio de proyecto de nuestro **workspace**.

¹Recomendamos el uso de **.S** porque Eclipse lo reconoce automáticamente

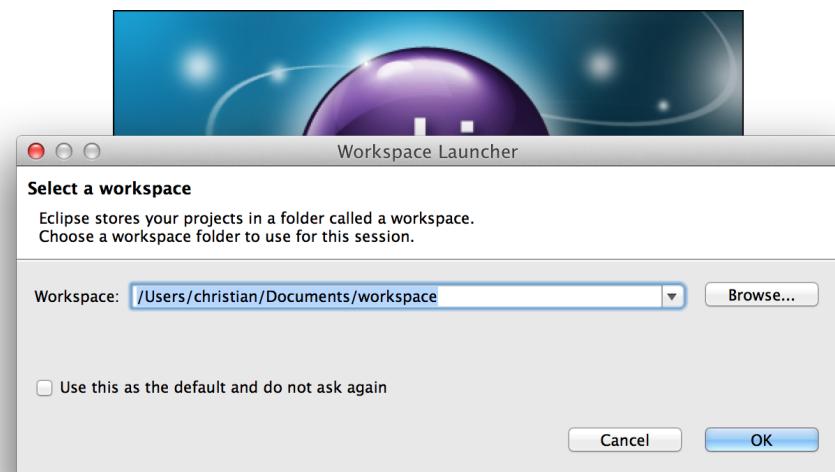


Figura 1.1: Ventana de selección de workspace.



Figura 1.2: Ventana de eclipse al abrir un workspace vacío.

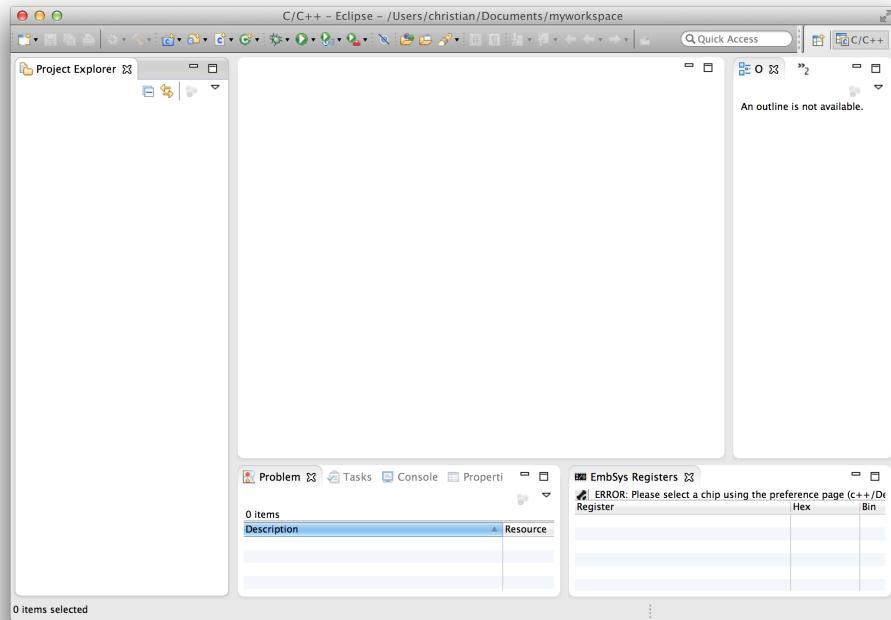


Figura 1.3: Ventana de eclipse con la perspectiva C/C++ sin proyectos.

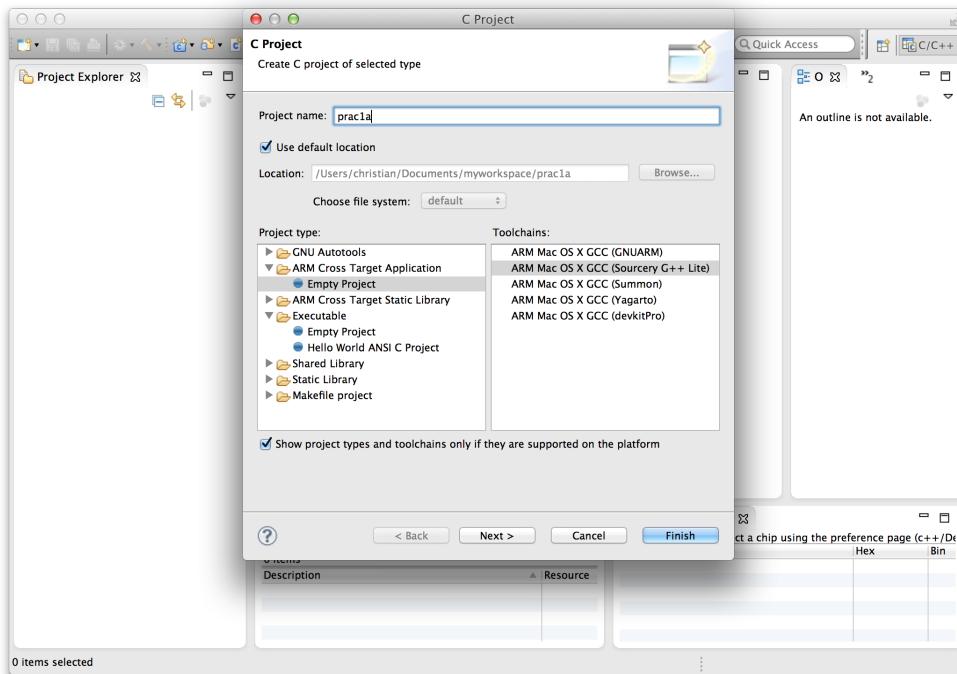


Figura 1.4: Ventana de creación de C project de tipo GNU ARM.

Cuadro 1 Programa en lenguaje ensamblador que compara dos números y se queda con el mayor.

```
.global start
.data
X:      .word 0x03
Y:      .word 0x0A

.bss
Mayor: .space 4

.text
start:
    LDR R4, =X
    LDR R3, =Y
    LDR R5, =Mayor
    LDR R1, [R4]
    LDR R2, [R3]
    CMP R1, R2
    BLE else
    STR R1, [R5]
    B FIN
else:  STR R2, [R5]
FIN:   B .
.end
```

Cuadro 2 Script de enlazado.

```
SECTIONS
{
    . = 0x0C000000;
    .data : {
        *(.data)
        *(.rodata)
    }
    .bss : {
        *(.bss)
        *(COMMON)
    }
    .text : {
        *(.text)
    }
    PROVIDE(end = .);
    PROVIDE(_stack = 0x0C7FF000 );
}
```

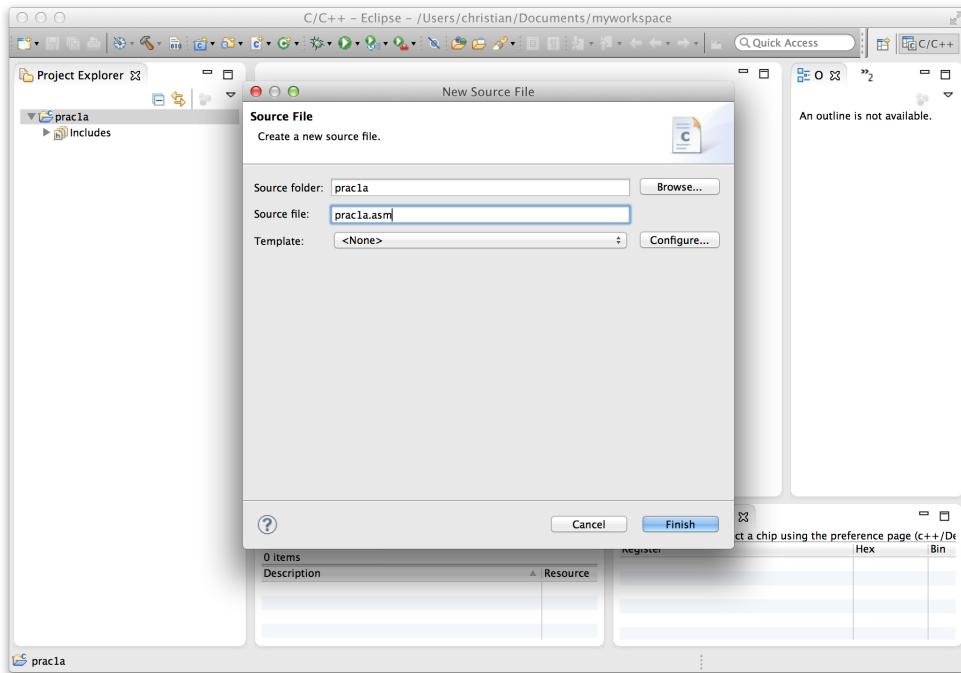


Figura 1.5: Ventana de creación de nuevo fichero fuente.

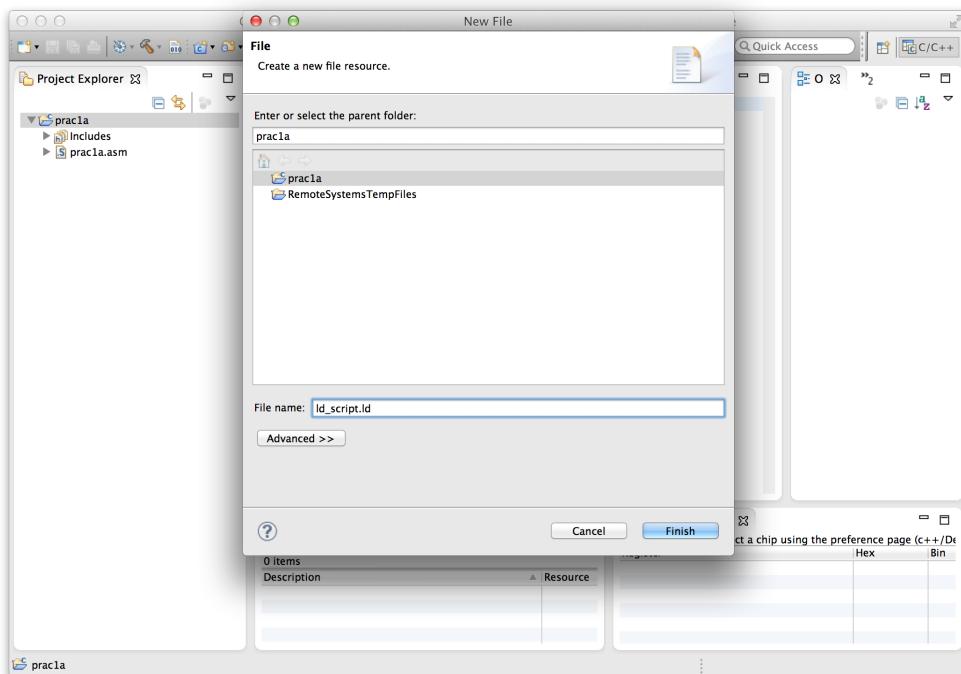


Figura 1.6: Ventana para añadir un nuevo fichero al proyecto.

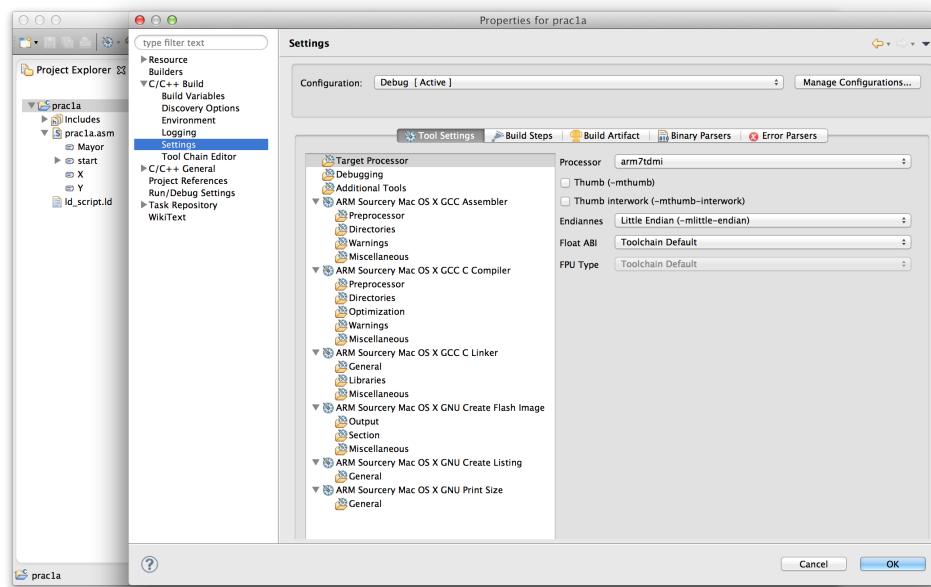


Figura 1.7: Ventana de propiedades (properties) del proyecto.

1.1.2. Depuración sobre la placa ARM

Tras generar el ejecutable de nuestro proyecto, nos toca comprobar que funciona correctamente. Para ello usaremos el depurador `arm-none-eabi-gdb` utilizando Eclipse como interfaz. El depurador nos permitirá ejecutar el programa instrucción a instrucción, poner puntos de parada (**breakpoints**), examinar registros, memoria, etc.

Si bien en la primera práctica podríamos usar el depurador en modo simulador, en general necesitaremos utilizar el procesador ARM real. Hay dos plugins de Eclipse que nos permiten hacer la depuración con GDB sobre el ARM: `GDB Hardware Debugging` y `Zylin Embedded debug`. Para la depuración en placa es aconsejable usar el primero.

Para depurar nuestro proyecto seguimos los siguientes pasos:

1. Conectamos el ARM al PC mediante el conector USB presente en la placa. Nos aseguramos de poner el *switch* de alimentación en `USBPower` de modo que la placa reciba alimentación a través del puerto USB. La placa debería encenderse y se debería ejecutar la aplicación de test (visible en la pantalla LCD).
2. Abrimos la perspectiva `Debug`. Para ello seleccionamos `Window→Open Perspective→Debug`. La apariencia de la ventana de Eclipse cambiará a la mostrada en la Figura 1.8. Como podemos ver, está dividida en varias regiones, cada una de ellas con pestañas:
 - Superior Izquierda: información sobre el proceso de depuración lanzado.
 - Superior Derecha: pestañas donde podemos visualizar los registros, las variables, los breakpoints, ...
 - Central Izquierda: código fuente en depuración. Al principio sólo aparecen los ficheros que tenemos abiertos en la perspectiva C/C++. Se irán abriendo automáticamente nuevas pestañas si el programa en ejecución salta a alguna función definida en otro fichero compilado con símbolos de depuración.
 - Central Derecha: en principio sólo nos muestra una lista de los símbolos definidos por el programa. Es interesante añadir a este panel una pestaña con el código desensamblado. Para ello seleccionamos `Window→Show View→Disassembly`.
 - Inferior: múltiples pestañas con distinto propósito. Por ejemplo aquí podemos poner el visor de memoria, seleccionando `Window→Show View→Memory`.
3. Para la conexión a la placa y el posterior volcado de código, usaremos una herramienta externa llamada `OpenOCD`. Para ello seleccionamos `Run→External Tools→External Tools Configurations...` (también puede llegarse al mismo sitio pinchando en la flecha del botón). Se abrirá una ventana en la que podemos configurar el uso de una nueva herramienta externa. Para ello debemos pinchar en `Program`, con lo que se habilitarán unos botones y pinchamos en el primero de ellos (). La Figura 1.10 muestra cómo debemos llenar el resto de la ventana. Para llenar la primera entrada podemos pinchar en el botón `Browse File System...` y buscamos el ejecutable de `OpenOCD` en el sistema (téngase en cuenta que la ruta mostrada en la figura puede ser distinta a la ruta que tengamos que poner en el laboratorio). Del mismo modo, para seleccionar el directorio de trabajo podemos pinchar en el botón `Browse Workspace....`. Finalmente debemos llenar los argumentos al programa

(`-f test/arm-fdi-ucm.cfg`) tal y como se indica en la Figura 1.10. Terminamos pinchando en **Apply** y **Close**. Esta herramienta externa la tendremos que ejecutar antes de comenzar la depuración, con la placa conectada a un puerto USB del equipo de laboratorio, tal y como indicamos en la siguiente sección.

4. Creamos una configuración de depuración para el proyecto usando el plugin GDB **Hardware Debugging**. Para ello seleccionamos **Run→Debug configurations...**, y se abrirá una ventana como la mostrada en la Figura 1.9. Ahora debemos seleccionar **GDB Hardware Debugging**, como muestra la Figura 1.11. Damos un nombre a la configuración y seleccionamos el proyecto y el ejecutable (probablemente, ya seleccionados correctamente por defecto). Para configurar el depurador, será necesario:
 - En este caso debemos además pinchar en un enlace azul que aparece en la base de la ventana que pone **Select Other...**. Se abrirá entonces una ventana como la mostrada en la Figura 1.12, en donde deberemos marcar la casilla **Use configuration specific settings** y seleccionar **Standard GDB Hardware Debugging Launcher**.
 - A continuación debemos seleccionar la pestaña **Debugger**, y rellenarla como indica la Figura 1.13. Primero deberemos seleccionar como depurador el **arm-none-eabi-gdb**. Después, en la parte baja de la ventana, deberemos seleccionar a qué *gdb server* debe conectarse. Este servidor lo pone OpenOCD, y está configurado para escuchar en el puerto 3333. Por ello seleccionaremos como JTAG Device **Generic TCP/IP**, como dirección IP **localhost** y como puerto 3333.
 - Finalmente deberemos llenar la pestaña **Startup**, tal y como se ilustra en la Figura 1.14. En la parte superior desmarcamos las casillas de **Reset** y **Halt**, y escribimos en el cuadro **monitor reset init**. En la parte inferior de la ventana marcamos las casillas **Set Breakpoint at** y **resume**, y ponemos ***start** en la casilla para la dirección del breakpoint. Pinchamos en **Apply** y luego en **Close**. La configuración de depuración queda entonces lista para ser utilizada cuando queramos depurar.

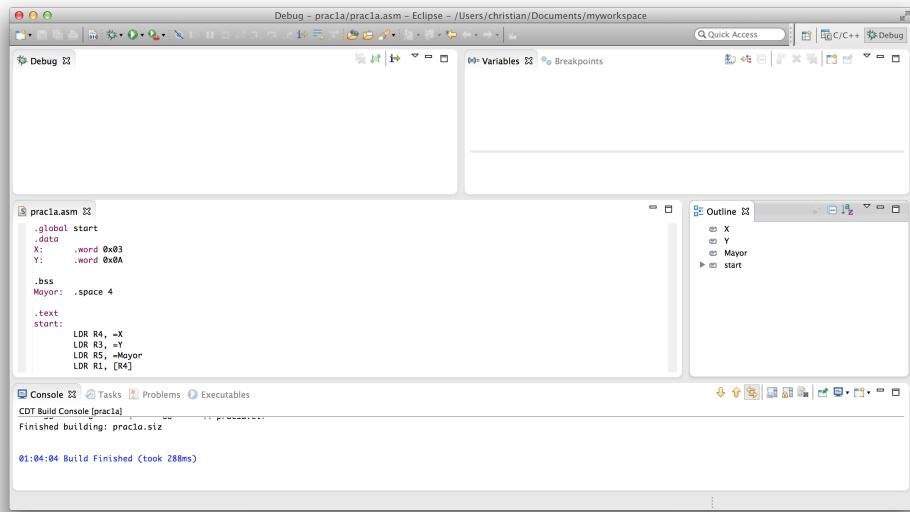


Figura 1.8: Perspectiva de depuración.

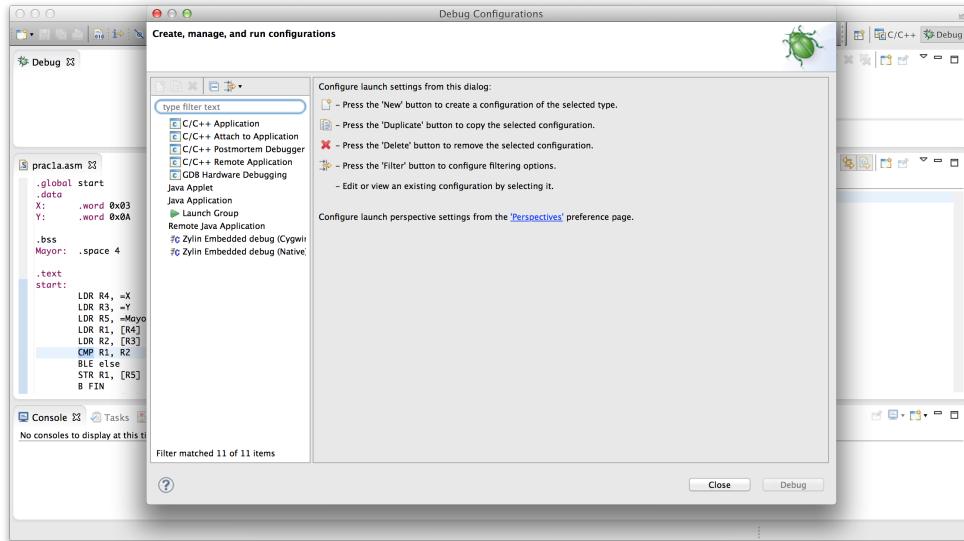


Figura 1.9: Ventana de configuraciones de depuración.

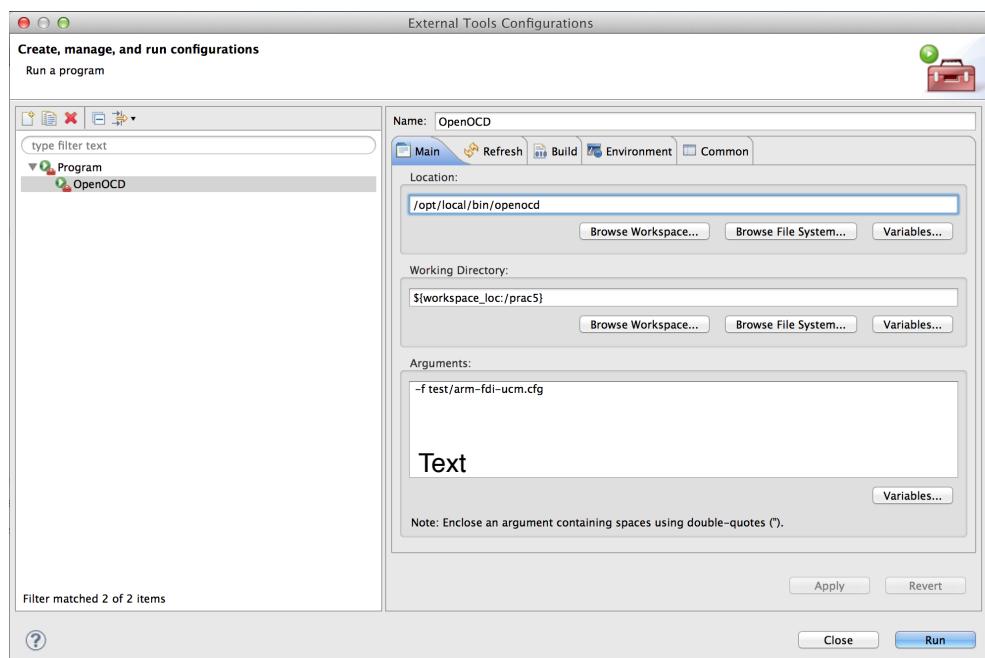


Figura 1.10: Ventana de configuración de OpenOCD como herramienta externa.

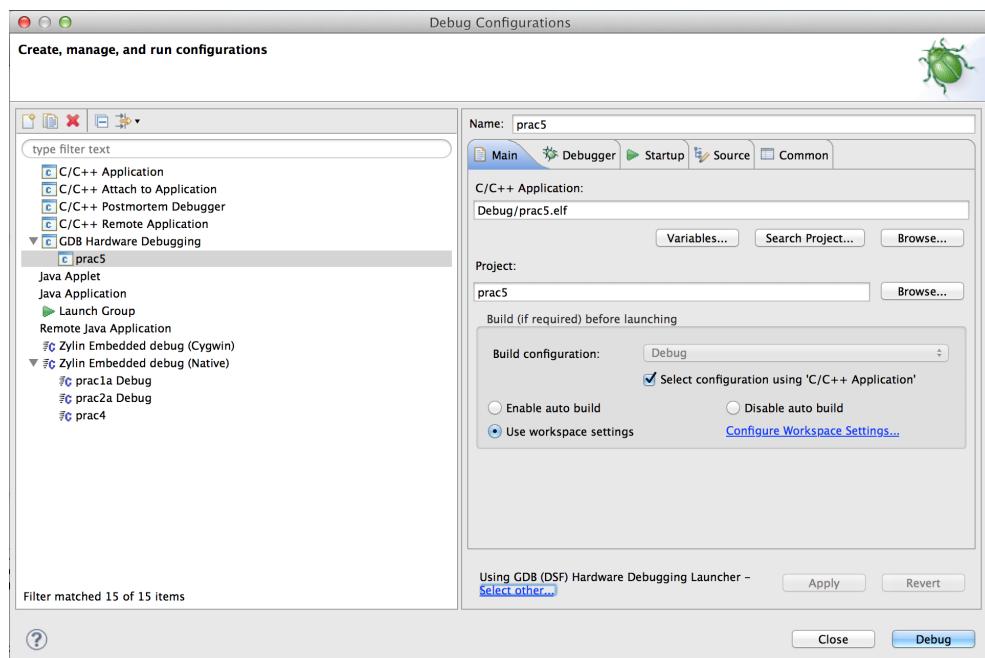


Figura 1.11: Configuración de depuración para conexión a placa: pestaña Main.

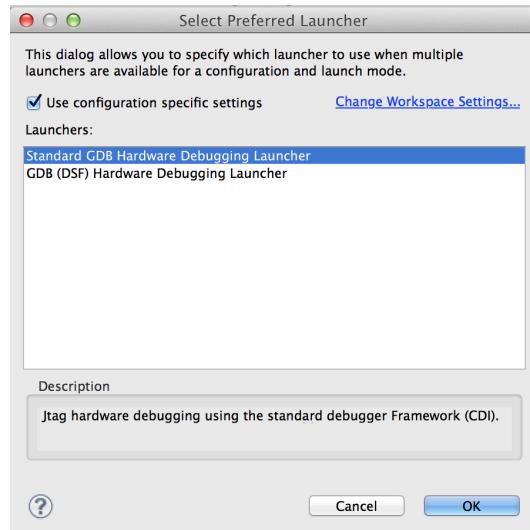


Figura 1.12: Configuración de depuración para conexión a placa: selección de *Standard GDB Hardware Debugging Launcher*.

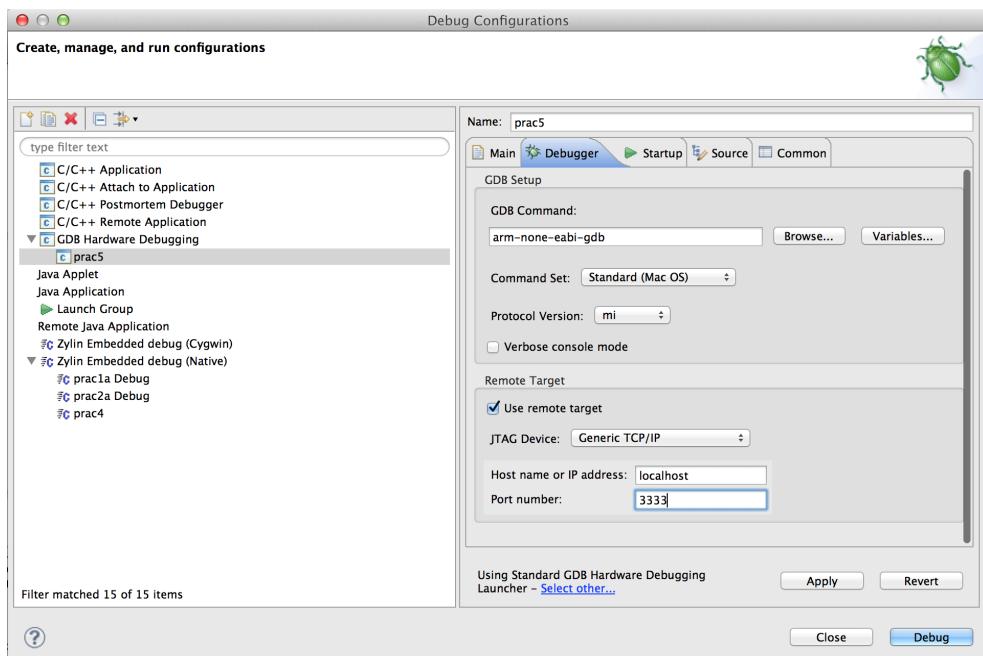


Figura 1.13: Configuración de depuración para conexión a placa: pestaña Debugger.

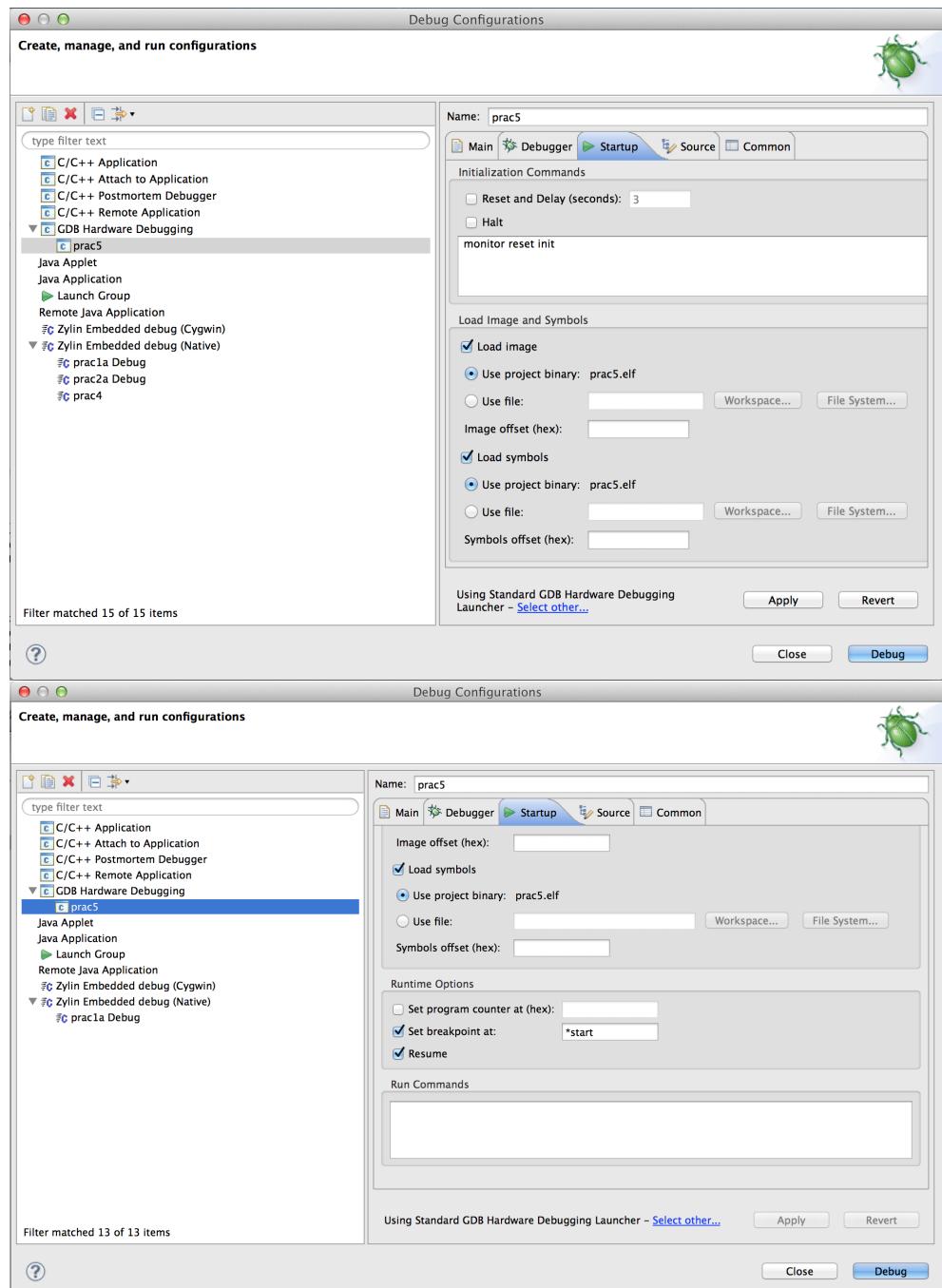


Figura 1.14: Configuración de depuración para conexión a placa: pestaña Startup.

1.1.3. Depuración sobre simulador

Para completar la presentación del entorno, y para facilitar el trabajo desde casa cuando no sea necesario el uso del procesador ARM real, veremos a continuación cómo depurar nuestro proyecto en modo simulador (sin necesidad de tener el procesador ARM conectado a nuestro PC). Esta vez usaremos el plugin **Zylin Embedded debug**.

Para depurar nuestro proyecto seguimos los siguientes pasos:

1. Abrimos la perspectiva **Debug**. Para ello seleccionamos **Window→Open Perspective→Debug**. La apariencia de la ventana de Eclipse cambiará a la mostrada en la Figura 1.15. Como podemos ver, está dividida en varias regiones, cada una de ellas con pestañas:
 - Superior Izquierda: información sobre el proceso de depuración lanzado.
 - Superior Derecha: pestañas donde podemos visualizar los registros, las variables, los breakpoints, ...
 - Central Izquierda: código fuente en depuración. Al principio sólo aparecen los ficheros que tenemos abiertos en la perspectiva C/C++. Se irán abriendo automáticamente nuevas pestañas si el programa en ejecución salta a alguna función definida en otro fichero compilado con símbolos de depuración.
 - Central Derecha: en principio sólo nos muestra una lista de los símbolos definidos por el programa. Es interesante añadir a este panel una pestaña con el código desensamblado. Para ello seleccionamos **Window→Show View→Disassembly**.
 - Inferior: múltiples pestañas con distinto propósito. Por ejemplo aquí podemos poner el visor de memoria, seleccionando **Window→Show View→Memory**.
2. Creamos una configuración de depuración para el proyecto usando el plugin **Zylin**. Para ello seleccionamos **Run→Debug configurations...**, y se abrirá una ventana como la mostrada en la Figura 1.16. En el panel izquierdo seleccionamos **Zylin Embedded debug (Native)** y pulsamos el botón que está en la parte superior izquierda del panel () para crear la configuración. Deberíamos tener una ventana como la mostrada en la Figura 1.17. Ahora debemos llenar correctamente las siguientes pestañas de la configuración:
 - Debugger: en esta pestaña indicamos el depurador a utilizar. En el laboratorio la ruta al toolchain cruzado se ha añadido al path del usuario, por tanto basta con poner el nombre del depurador en la entrada **GDB debugger**: **arm-none-eabi-gdb**. Además de esto, en la parte superior podemos colocar un breakpoint temporal en donde nosotros queramos. Vamos a colocarlo en la dirección del símbolo **start**, para ello escribimos ***start** en el cuadro **Stop on startup at..**. El resultado se muestra en la Figura 1.18.
 - Commands: en esta pestaña damos los comandos que gdb debe ejecutar en la inicialización al iniciar la depuración. En el cuadro **Initialize commands** escribimos **target sim**. Esto indica que el objetivo de depuración es el simulador. En el cuadro **Run commands** escribimos:

```
load
run
```

Esto indica al depurador que cargue el fichero seleccionado en la pestaña principal, leyendo de éste los símbolos de depuración. La ventana resultante se muestra en la Figura 1.19.

3. Ya estamos listos para depurar, para ello hacemos click en el botón **Debug**. Esto guardará la configuración de depuración con el nombre seleccionado en la primera pestaña e iniciará una sesión de depuración con esta configuración. Si más adelante queremos volver a depurar este proyecto no será necesario crear una configuración de depuración, bastará con seleccionar la que acabamos de crear. La Figura 1.20 nos muestra el estado inicial que debería tener la ventana en depuración si hemos seguido todos los pasos:

- En la parte izquierda del panel central debemos encontrar el código del cuadro 1, con la primera línea de código tras la etiqueta **start** marcada en verde, con una flecha en el marco izquierdo. Esto quiere decir que el programa a comenzado correctamente su ejecución simulada y que se ha detenido en el breakpoint que hemos puesto en la dirección de **start**.
- En la parte derecha del panel central debemos encontrar el código desensamblado. Es el contenido de la memoria en el entorno de la dirección de la instrucción actual, reinterpretada como instrucciones.

Notemos la diferencia con el anterior. En el panel izquierdo tenemos el código fuente, tal y como lo programamos, haciendo uso de las facilidades proporcionadas por el ensamblador. En el lado derecho tenemos las instrucciones tal y como las ha generado el ensamblador. Fijémonos por ejemplo en la instrucción **LDR R4,=X**, que aprovecha una facilidad del ensamblador para escribir en el registro R4 la dirección correspondiente a la etiqueta X. Como podemos ver en el panel derecho, se ha traducido por **LDR R4, [PC,#36]**, que utiliza un modo de direccionamiento correcto para las instrucciones de load en ARM². También podemos observar que la dirección equivalente a PC+36 aparece como un comentario tras el signo ;. En cualquiera de los dos paneles podemos poner breakpoints. Si marcamos el botón  en el panel de desensamblado, las instrucciones fuente se mezclarán con las instrucciones máquina correspondientes. Esto facilita el seguimiento del código máquina desensamblado, sobre todo cuando el código fuente es de un lenguaje de alto nivel como C, como veremos en las últimas prácticas.

4. Antes de simular nuestro código vamos a abrir un visor de memoria para poder evaluar el valor de nuestras variables. Para ello, lo primero que tenemos que hacer es abrir la pestaña **Memory** en el panel inferior, si no la tenemos abierta ya (**Window→Show View→Memory**). Seleccionamos esta pestaña y añadimos un nuevo monitor pinchando en el ícono . Se abrirá una ventana como la de la Figura 1.21 en la que debemos escribir la dirección a partir de la que queremos monitorizar la memoria. Si ponemos la dirección de descarga, 0x0C000000, en la que hemos colocado la sección .data, seguida en orden por las secciones .bss y .text, el aspecto de la ventana resultante debe ser similar al mostrado en la Figura 1.22.

²El proceso de esta traducción implica tanto al ensamblador como al enlazador. El ensamblador pone esta instrucción, que lee un valor de una dirección de memoria donde el enlazador habrá de escribir la dirección correspondiente a la etiqueta X. Estudiaremos este proceso en más detalle en la práctica 4.

5. Ahora, para simular todo el código se puede pulsar sobre el icono de **Resume**  o F8 y después pulsar sobre el icono de **Suspend** .

- **¿Cómo sabemos si el código se ha ejecutado correctamente?** El dato mayor se ha escrito en la posición de memoria reservada y etiquetada como **Mayor**. Se puede comprobar su valor en el visor de memoria, que habrá quedado marcado en rojo como ilustra la Figura 1.23.

6. Sin embargo, para entender mejor el funcionamiento de cada una de las instrucciones conviene realizar una ejecución paso a paso. Además, si el resultado no es correcto tendremos que depurar el código para encontrar cuál es la instrucción incorrecta, para lo que una ejecución paso a paso nos será muy útil. Para ello:

- Paramos la simulación pulsando el botón **Terminate**, . Se habilitará el botón **Remove all terminated launches**  que nos permitirá limpiar el panel **Debug** (a veces no se habilita este botón, entonces deberemos pinchar con el botón derecho del ratón en la sesión de depuración y seleccionar **Terminate and Remove**).
- Si necesitamos modificar el código, pasamos a la perspectiva **C/C++**, editamos los ficheros, los guardamos y recompilamos el proyecto. Luego volvemos a la perspectiva **debug**.
- Podemos iniciar una sesión de depuración con la última configuración de depuración utilizada pulsando el botón .
- Para ejecutar/simular paso a paso tenemos las siguientes opciones:
 - Step Over () : ejecuta hasta la siguiente instrucción. Si es una llamada a subrutina para después de la ejecución completa de la subrutina.
 - Step Into () : ejecuta hasta la siguiente instrucción. Si es una llamada a subrutina se detiene en la primera instrucción de la subrutina.
 - Poner un **Breakpoint** (punto de parada) y reanudar la ejecución pulsando el botón **Resume** . Para poner un **Breakpoint** nos ponemos en el margen izquierdo de alguna instrucción (tanto en el panel de código fuente como en el panel de desensamblado) y hacemos un doble click. Aparecerá la marca . Podemos poner varios **Breakpoints**, la ejecución se detendrá en aquel que se alcance primero.

Además, nos interesaría observar los cambios que va realizando nuestro programa, para ello:

- Para ver cómo va cambiando el valor de los registros a medida que ejecutamos las instrucciones usaremos el visor de registros, que se encuentra en el panel superior derecho. Conviene seleccionar para este visor, con el botón , el layout horizontal.
- En el visor de memoria veremos los cambios que nuestro programa realice sobre la memoria.

La Figura 1.24 muestra un ejemplo de una sesión de depuración, donde hemos ido ejecutando paso a paso hasta la instrucción siguiente a **BLE else**. Podemos ver que la condición no se cumple y que se pasa a ejecutar el bloque **else**. Podemos ver el estado de los registros en este momento, y como el PC contiene

la dirección 0x0C000030 correspondiente a la siguiente instrucción a ejecutar. También podemos ver cómo el panel de desensamblado nos marca en verde las últimas instrucciones ejecutadas, quedando sin marcar las instrucciones en la rama del then.

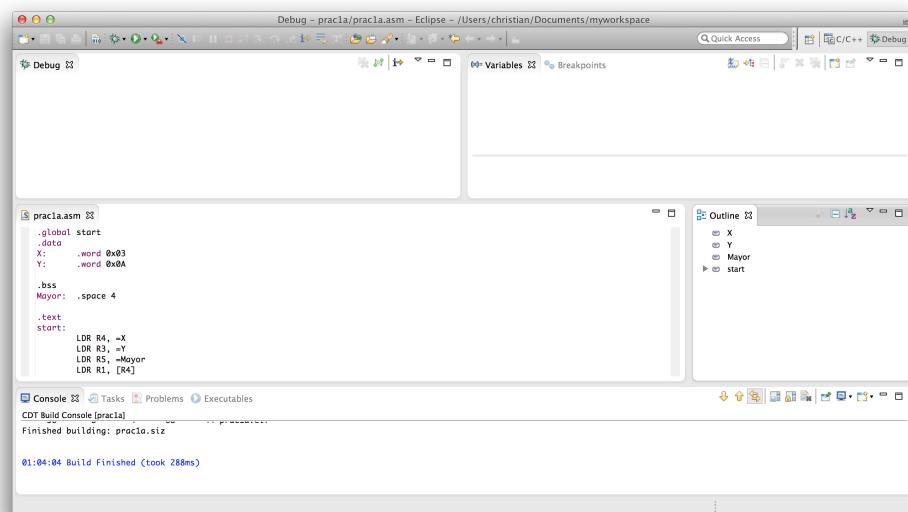


Figura 1.15: Perspectiva de depuración.

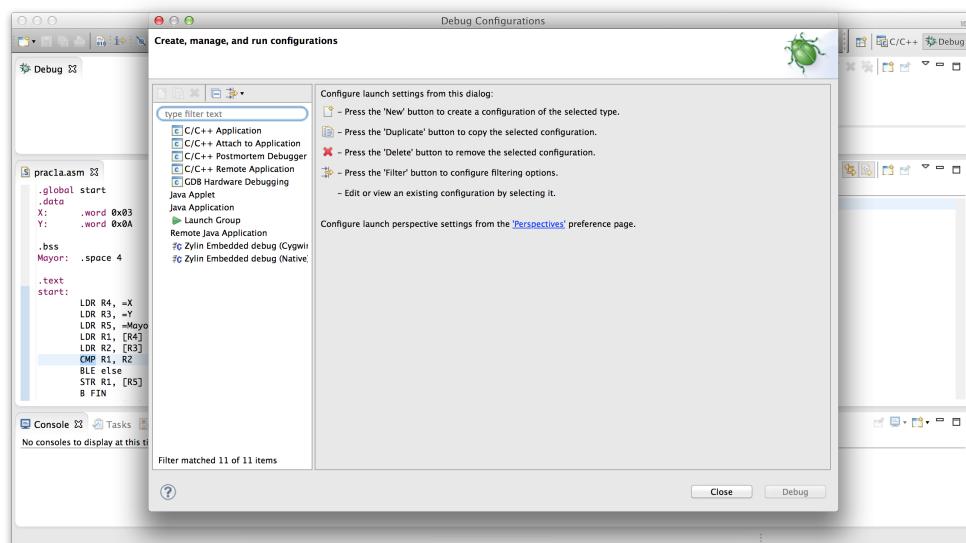


Figura 1.16: Ventana de configuraciones de depuración.

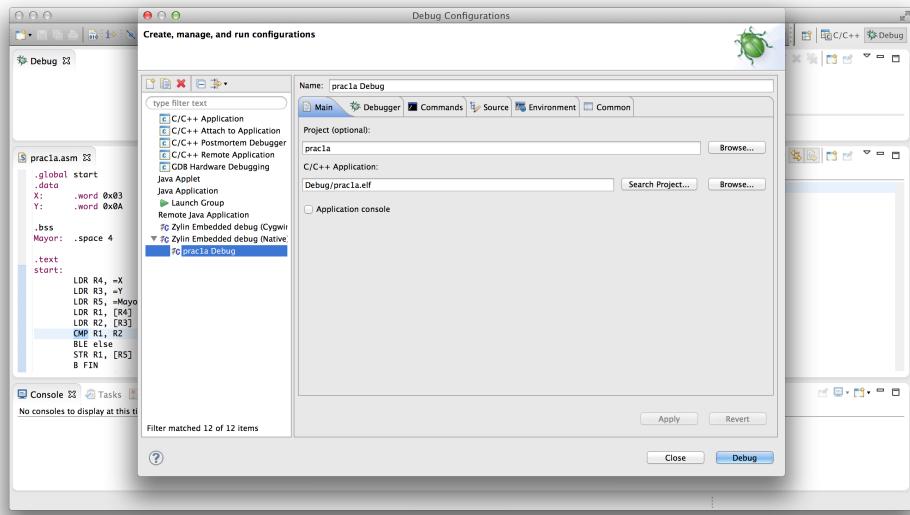


Figura 1.17: Ventana de creación de una nueva configuración Zylin nativa para el proyecto.

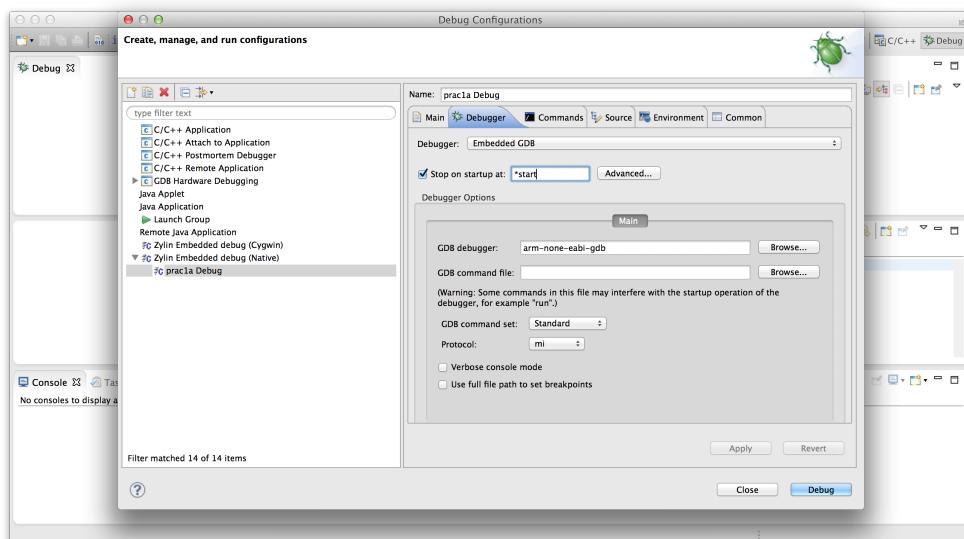


Figura 1.18: Pestaña **debugger** de la configuración de depuración Zylin.

1.1. USO DE ECLIPSE PARA CROSS-COMPILING

21

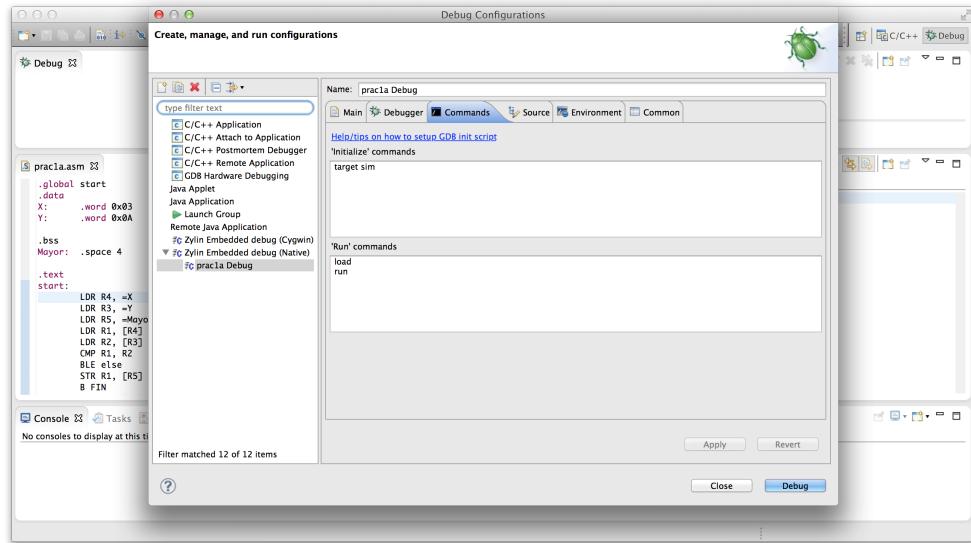


Figura 1.19: Pestaña commands de la configuración de depuración Zylin.

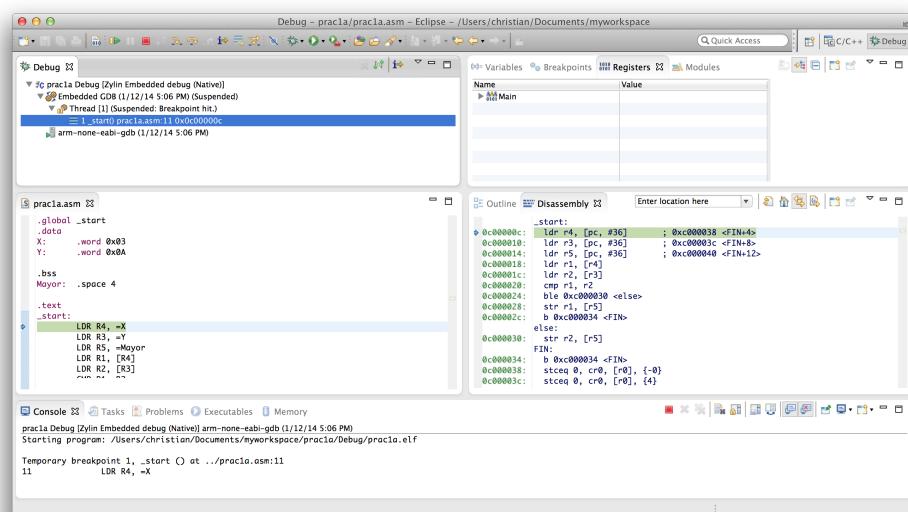


Figura 1.20: Aspecto inicial de la ventana de depuración.

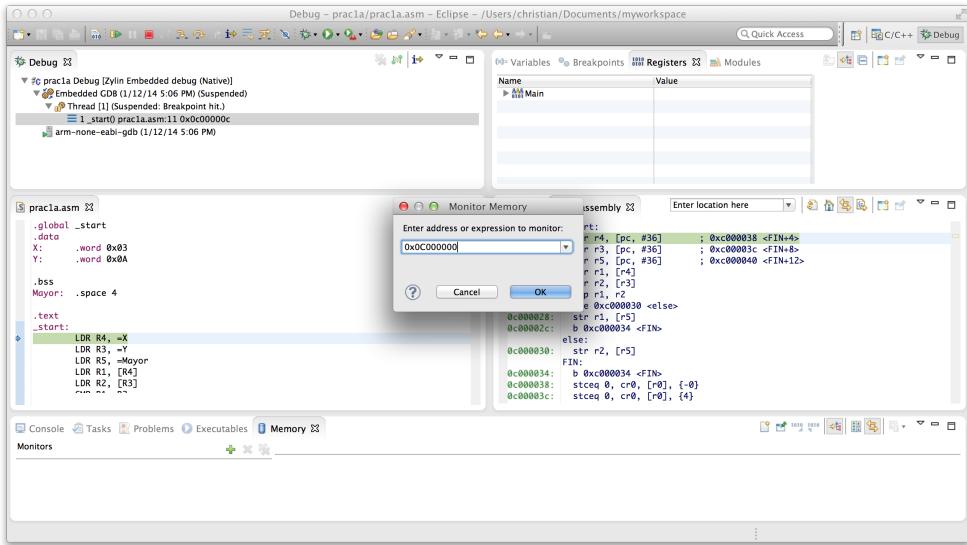


Figura 1.21: Ventana para indicar la dirección del monitor de memoria.

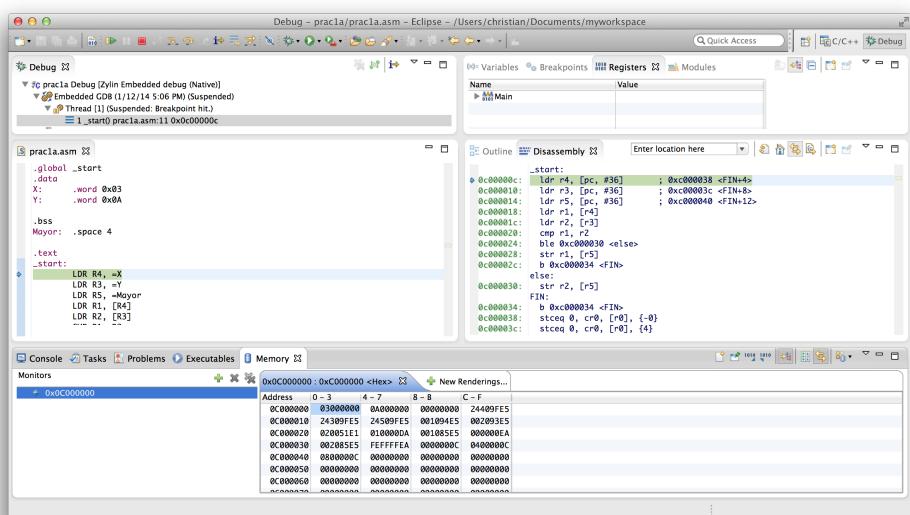


Figura 1.22: Ventana con el aspecto del monitor de memoria en la dirección 0x0C000000.

1.1. USO DE ECLIPSE PARA CROSS-COMPILING

23

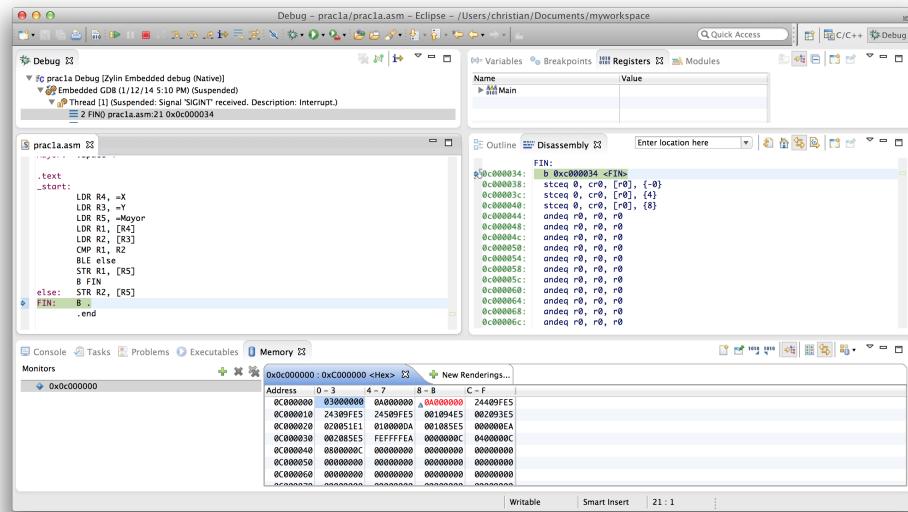


Figura 1.23: Ventana con el aspecto del monitor de memoria tras la simulación completa del programa.

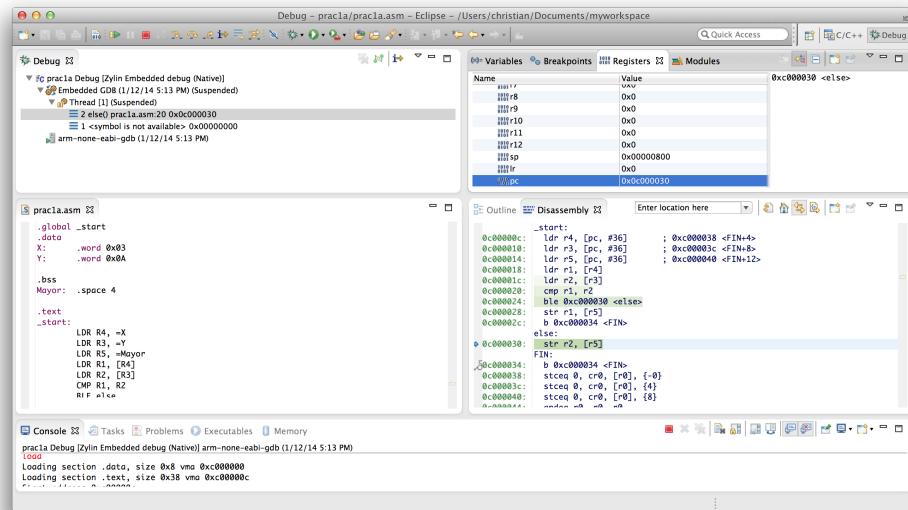


Figura 1.24: Ejemplo de una sesión de depuración con el visor de registros.