

## Práctica 2

Programación en ensamblador y etapas de compilación.

## 2.1. Objetivos

Una vez familiarizados con el repertorio de instrucciones del ARM y con el entorno de desarrollo, en esta práctica se persiguen los siguientes objetivos:

- Comprender la finalidad de las etapas de compilación, ensamblado y enlazado.
- Conocer algunos modos de direccionamiento adicionales en instrucciones LDR/STR
- Adquirir práctica en el manejo de vectores de datos (arrays) conociendo la posición de inicio del vector y su longitud.
- Aprender a codificar en ensamblador del ARM sentencias especificadas en lenguaje de alto nivel.

## 2.2. Generación de un ejecutable

La figura 2.1 ilustra el proceso de generación de un ejecutable a partir de uno o más ficheros fuente y de algunas bibliotecas. Cada fichero en código C pasa por el preprocesador que modifica el código fuente original siguiendo directivas de preprocesado (`#define`, `#include`, etc.). El código resultante es entonces compilado, transformándolo en código ensamblador. El ensamblador genera entonces el código objeto para la arquitectura destino (ARM en nuestro caso).

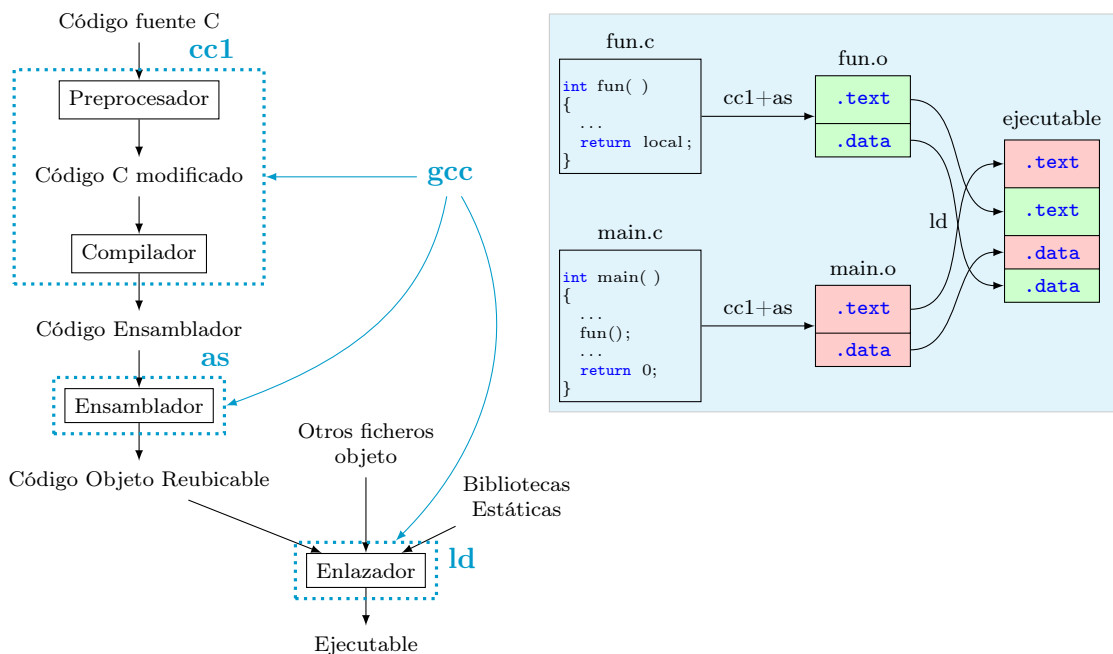


Figura 2.1: Proceso de generación de código ejecutable

Los ficheros de código objeto son ficheros binarios con cierta estructura. En particular debemos saber que estos ficheros están organizados en secciones con nombre. Normalmente

suelen definirse siempre tres secciones: `.text` para el código, `.data` para datos (variables globales) con valor inicial y `.bss` para datos no inicializados. Para definir estas secciones simplemente tenemos que poner una línea con el nombre de la sección. A partir de ese momento el ensamblador considerará que debe colocar el contenido subsiguiente en la sección con dicho nombre. Por ejemplo el programa del cuadro 1 tiene una sección `.bss` en la que se reserva espacio para almacenar el resultado, una sección `.data` para declarar variables con valor inicial y una sección `.text` que contiene el código del programa. La etiqueta `.equ` permite declarar constantes (que recordamos no se almacenarán en memoria).

---

**Cuadro 1** Ejemplo de código con secciones.

---

```

.global start

.equ UN0, 0x01

.bss
RES: .word 0x03

.data
A:   .word 0x03
B:   .word 0x02

.text
start:
    MOV R0, #UN0
    LDR R2, =A      @ carga en R2 la dirección de A
    LDR R1, [R2]    @ carga el valor de A
    ADD R2, R0, R1
    LDR R3, =B      @ carga en R3 la dirección de B
    LDR R4, [R3]    @ carga el valor de B
    ADD R4, R2, R4
    LDR R3, =RES    @ carga en R3 la dirección de RES
    STR R4, [R3]    @ guarda el contenido de R4 en RES
FIN:  B .
      .end

```

---

Los ficheros objeto no son todavía ejecutables. El ensamblador ha generado el contenido de cada una de las secciones, pero falta decidir en qué direcciones de memoria se van a colocar dichas secciones y resolver los símbolos. Por ejemplo el valor de cada etiqueta no se conoce hasta que no se decide en qué dirección de memoria se va a colocar la sección en la que aparece.

Podríamos preguntarnos por qué no decide el ensamblador la dirección de cada sección y genera directamente el ejecutable final. La respuesta es que nuestro programa se implementará generalmente con más de un fichero fuente, pero las etapas de compilación y ensamblado se hacen fichero a fichero para reducir la complejidad del problema. Es más, utilizaremos frecuentemente bibliotecas de las que ni siquiera tendremos el código fuente. Por lo tanto, es necesario añadir una etapa de enlazado para componer el ejecutable final. El programa que la realiza se conoce como enlazador.

Como ilustra la Figura 2.1, el enlazador tomará los ficheros de código objeto procedentes

de la compilación de nuestros ficheros fuente y de las bibliotecas externas con las que enlacemos, y reubicará sus secciones en distintas direcciones de memoria para formar las secciones del ejecutable final. En este proceso todos los símbolos habrán quedado resueltos, y como consecuencia todas nuestras etiquetas tendrán asignadas una dirección definitiva. En esta etapa el usuario puede indicar al enlazador cómo colocar cada una de las secciones utilizando un *script* de enlazado, como hicimos en la práctica anterior con el `ld_script.ld`.

## 2.3. Modos de direccionamiento de LDR/STR

Un modo de direccionamiento especifica la forma de calcular la dirección de memoria efectiva de un operando mediante el uso de la información contenida en registros y/o valores inmediatos contenidos en la propia instrucción de la máquina. En la práctica anterior estudiamos dos de los modos de direccionamiento válidos en ARMv4 para las instrucciones `ldr/str`. Sin embargo, para el desarrollo de esta práctica nos será muy útil considerar uno más:

- Indirecto de registro con desplazamiento por registro** La dirección de memoria a la que deseamos acceder se calcula sumando la dirección almacenada en un registro (base) al valor entero (offset) almacenado en otro registro, llamado registro índice. Este segundo registro puede estar opcionalmente multiplicado o dividido por una potencia de 2. En realidad, hay cinco posibles operaciones que se pueden realizar sobre el registro de offset, pero sólo veremos dos de ellas:
  - Desplazamiento aritmético a la derecha (ASR). Equivalente a dividir por una potencia de 2. En ensamblador la instrucción se escribiría como:  
`LDR Rd, [Rb, Ri, ASR #Potencia de dos]`
  - Desplazamiento lógico a la izquierda (LSL). Equivalente a multiplicar por una potencia de 2. En ensamblador la instrucción se escribiría como:  
`LDR Rd, [Rb, Ri, LSL #Potencia de dos]`

En la Figura 2.2 se ilustra el resultado de realizar la operación `ldr r2, [r1, r5, LSL #2]`

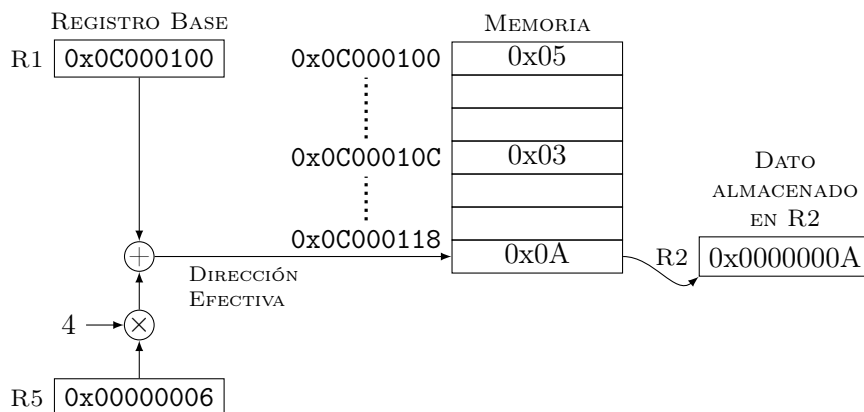


Figura 2.2: Ejemplo de ejecución de la instrucción `ldr r2, [r1, r5, LSL #2]`.

Como se ilustra en el cuadro 2, éste nuevo modo de direccionamiento es el más adecuado para recorrer arrays:

- En un registro mantendremos la dirección de comienzo del array y lo usamos como registro base.
- En otro registro mantenemos el índice del array que queremos acceder y será el offset. Bastará con multiplicar dicho registro (usando LSL) por el número de bytes que ocupa cada elemento del array (4 para números enteros) para conseguir la dirección del elemento deseado.

**Cuadro 2** Ejemplo de recorrido de arrays con direccionamiento indirecto de registro con desplazamiento por registro.

Código C	Ensamblador
<pre>int A[100]={0};  for (i=0; i &lt; 100; i++){     ... = A[i] ...; }</pre>	<pre>/* reservamos memoria para A, inicializada a 0 */ .data A: .space 100, 0  .text /* almacenamos en r1 la dirección de comienzo de A */ LDR r1,=A MOV r2,#0 @ inicializamos i for: CMP r2,#100 BGE fin  ... /* leemos el elemento i-ésimo del vector A y lo almacenamos en r3 */ LDR r3,[r1,r2,ls1#2]  ... ADD r2,r2,#1 B for fin:</pre>

## Ejemplos

```
LDR R1, [R0]           @ Carga en R1 lo que hay en Mem[R0]
LDR R8, [R3, #4]        @ Carga en R8 lo que hay en Mem[R3 + 4]
LDR R12,[R13, #-4]      @ Carga en R12 lo que hay en Mem[R13 - 4]
STR R2, [R1, #0x100]    @ Almacena en Mem[R1 + 256] lo que hay en R2
STR R4, [PC, R1, LSL #2] @ Almacena en Mem[PC + R1*4] lo que hay en R4
LDR R11,[R3, R5, LSL #3] @ Carga en R11 lo que hay en Mem[R3 + R5*8]
```

## 2.4. Pseudo-instrucción LDR

Como hemos visto en el ejemplo anterior, para recorrer los arrays necesitamos almacenar su dirección de comienzo en un registro (`r1`, en el Cuadro 2). Para ello resulta necesaria la pseudoinstrucción:

```
LDR R1,=A
```

La Figura 2.3 ilustra el funcionamiento de esta pseudo-instrucción, donde suponemos que tras el enlazado la dirección asociada a `A` es `0x208` y que en algún momento el programa ha escrito nuevos valores en nuestro array `A`. Utilizando `LDR R1,=A` conseguimos cargar la dirección `0x208` en `R1`.

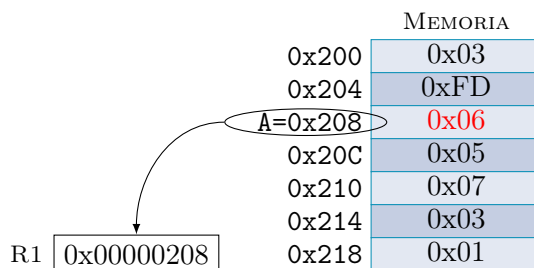


Figura 2.3: Ilustración del funcionamiento de la pseudoinstrucción `LDR R1,=A`.

Esta codificación de `LDR` no se corresponde exactamente con ninguno de los modos de direccionamiento válidos, es decir, no es una instrucción ARM válida, es una facilidad proporcionada por el ensamblador que nos permite guardar en el registro `R1` la dirección asociada a `A`. De hecho, parecería mucho más lógico utilizar una instrucción para escribir un valor inmediato en un registro, algo como:

```
MOV R1, #A
```

Sin embargo el número de bits reservados en el código de la instrucción para codificar el operando inmediato de las instrucciones aritméticas no permite codificar un valor de 32 bits, que es lo que ocupa la dirección representada por `A`.

Una solución inteligente sería escribir en memoria la dirección representada por `A`, y luego hacer un load de la dirección donde hemos escrito `A`. Esto es fácil hacerlo, basta con escribir al final de la sección `.text`, después de la última instrucción de nuestro programa:

```
dirA: .word A
```

y entonces cargaríamos el valor de la etiqueta `A` con:

```
ldr r1, dirA
```

que el ensamblador transformaría en

```
LDR R1,[PC,#Desplazamiento]
```

donde el `Desplazamiento` lo calcularía el ensamblador como la distancia relativa entre la instrucción `LDR` y la etiqueta `dirA` (como están en la misma sección este desplazamiento no depende del enlazado). Pues bien, esto es justo lo que se hace con la pseudo-instrucción `LDR R1,=A`, que es más cómoda de utilizar.

Volveremos de nuevo sobre la pseudo instrucción `LDR R<n>, =Etiqueta` en la práctica 4, donde analizaremos en detalle la resolución de símbolos.

## 2.5. Preparación del entorno de desarrollo

En la práctica anterior aprendimos a crear un proyecto en eclipse con el plugin de GNU ARM. En esta práctica podríamos repetir el procedimiento para añadir un nuevo proyecto a nuestro **workspace** para cada uno de los programas que tendremos que desarrollar. Sin embargo resultará más conveniente crear los nuevos proyectos como copias de los anteriores. Para crear un nuevo proyecto **prac2a** a partir de un proyecto anterior de nuestro **workspace**, **prac1a**, procederemos de la siguiente manera:

1. Abrimos eclipse, seleccionando nuestro **workspace** en la ventana de **Workspace Launcher**.
2. Seleccionamos la perspectiva **C/C++**.
3. Pinchamos con el botón derecho del ratón sobre el proyecto a copiar, **prac1a**, con lo que se desplegará un menú como el que muestra la Figura 2.4, y seleccionamos la opción **Copy**. Alternativamente podemos usar la combinación de teclas del sistema (**Ctrl c** en windows y Linux, **⌘ c** en Mac Os X) para copiar el proyecto.
4. Pinchamos con el botón derecho del ratón sobre el explorador de proyectos, con lo que se despliega de nuevo el mismo menú, y seleccionamos la opción **Paste**. Alternativamente podemos usar la combinación de teclas del sistema (**Ctrl v** en windows y Linux, **⌘ v** en Mac Os X) para pegar el proyecto copiado. Se abrirá una ventana que nos permitirá seleccionar un nombre para el nuevo proyecto, como ilustra la Figura 2.5.
5. Será conveniente borrar la carpeta **Debug** del nuevo proyecto. Esta carpeta contendrá los ficheros objeto y el ejecutable del proyecto anterior. La primera compilación de nuestro proyecto volverá a crear la carpeta para alojar los nuevos ficheros objeto y el nuevo ejecutable.
6. Finalmente, pinchamos en el proyecto con el botón derecho y seleccionamos **Properties**→**C/C++ Build**→**Refresh Policy**. En el cuadro aparecerá el nombre de la práctica original, lo seleccionamos y pulsamos el botón **Delete**. Después pulsamos el botón **Add Resource...** y seleccionamos el nuevo proyecto en la ventana que se abre, como muestra la Figura 2.7. Pulsamos **Ok**. Esto hará que se refresque automáticamente el proyecto en el explorador de proyectos cuando compilemos.

Con esto tenemos creado un nuevo proyecto. Como ilustra la Figura 2.6, el proyecto contiene una copia de los ficheros del proyecto original. Generalmente nos interesará renombrar el fichero fuente, por ejemplo cambiar **prac1a.asm** por **prac2a.asm**. Esto podemos hacerlo seleccionando el fichero con el botón derecho del ratón y seleccionando **Rename...** en el menú desplegado. Otra alternativa es directamente borrar los ficheros fuente que queramos cambiar, seleccionando la opción **Delete** en el mismo menú, y luego añadir nuevos ficheros como lo hicimos en la práctica anterior.

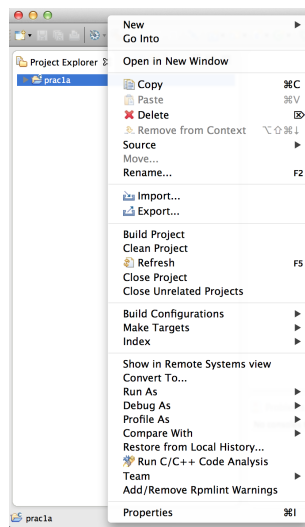


Figura 2.4: Menu desplegado al pinchar con el botón derecho del ratón sobre un proyecto de nuestro `workspace`.

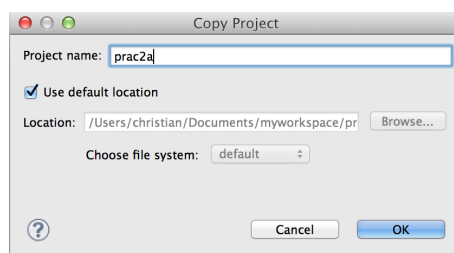


Figura 2.5: Ventana abierta al pegar el proyecto copiado. Nos permite seleccionar un nombre para el nuevo proyecto.

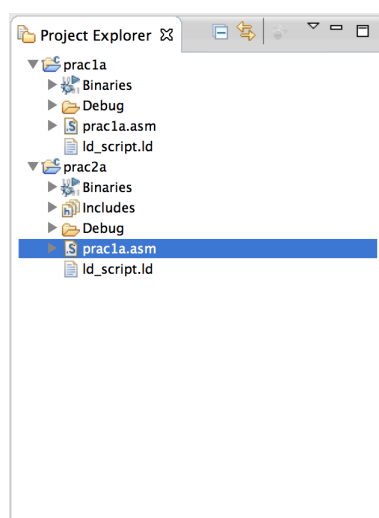



Figura 2.6: Aspecto que tendrá el explorador de proyectos cuando hayamos copiado el proyecto `prac1a` con el nombre `prac2a`.



### 2.5.1. Copia de configuraciones de depuración

Aunque aún no hemos preparado el proyecto, cuando lo hagamos deberemos simularlo y depurarlo. Ya explicamos en la práctica anterior cómo crear una configuración de depuración, sin embargo existe una alternativa: copiar una configuración de depuración de otro proyecto de nuestro **workspace**. Para esto el proyecto del que queremos copiar la configuración debe estar abierto, de lo contrario sus configuraciones de depuración no serán visibles. Para hacer la copia haremos lo siguiente:

- Cambiamos a la perspectiva de **Debug**.
- Seleccionamos **Run→Debug Configurations...** Se abrirá una ventana como la de la Figura 2.8, y podremos ver en su panel izquierdo la configuración de depuración que queremos copiar (**prac1a Debug** en nuestro ejemplo).
- Seleccionamos la configuración a copiar y pulsamos el botón . Entonces se creará una nueva configuración como copia exacta de la anterior, en la que debemos cambiar el nombre del proyecto (**prac1a** por **prac2a**) y el nombre del ejecutable a depurar (**Debug/prac1a.elf** por **Debug/prac2a.elf**) y pulsamos el botón **Apply**. La Figura 2.9 ilustra el aspecto final que tendría la ventana.
- Finalmente pulsamos el botón **Close**. Si nos equivocamos y pulsamos **Debug** nos dirá que se ha producido un error. Esto es lógico porque estaremos intentando depurar un ejecutable que aún no hemos creado. Podemos ignorar el error.

Una vez hecho esto, podemos cerrar el proyecto original. Para ello, en el explorador de proyectos de la perspectiva **C/C++** pinchamos sobre el proyecto a cerrar y seleccionamos **Close Project** en el menú desplegado.

## 2.6. Desarrollo de la práctica

Es obligatorio traer realizados los apartados a) y b) de casa. Durante la sesión de laboratorio se solicitará una modificación de uno de ellos, que habrá que realizar en esas 2 horas. En **TODOS** los apartados será obligatorio definir tres secciones:

- **.data** para variables con valor inicial (declaradas como **.word**)
- **.bss** para variables sin valor inicial (declaras como **.space**)
- **.text** para el código

- a) Codificar en ensamblador del ARM el siguiente código C, encargado de buscar el valor máximo de un vector de enteros positivos **A** de longitud **N** y almacenarlo en la variable **max**. Es **OBLIGATORIO** escribir en memoria el valor de **max** cada vez que éste cambie (es decir, no basta con actualizar un registro; hay que realizar una instrucción **str**).

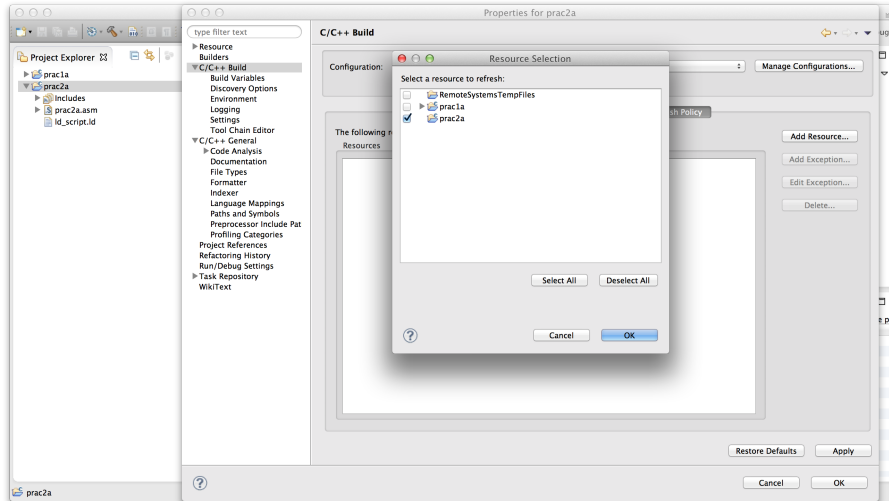


Figura 2.7: Ventana para seleccionar las acciones de refresco del proyecto.

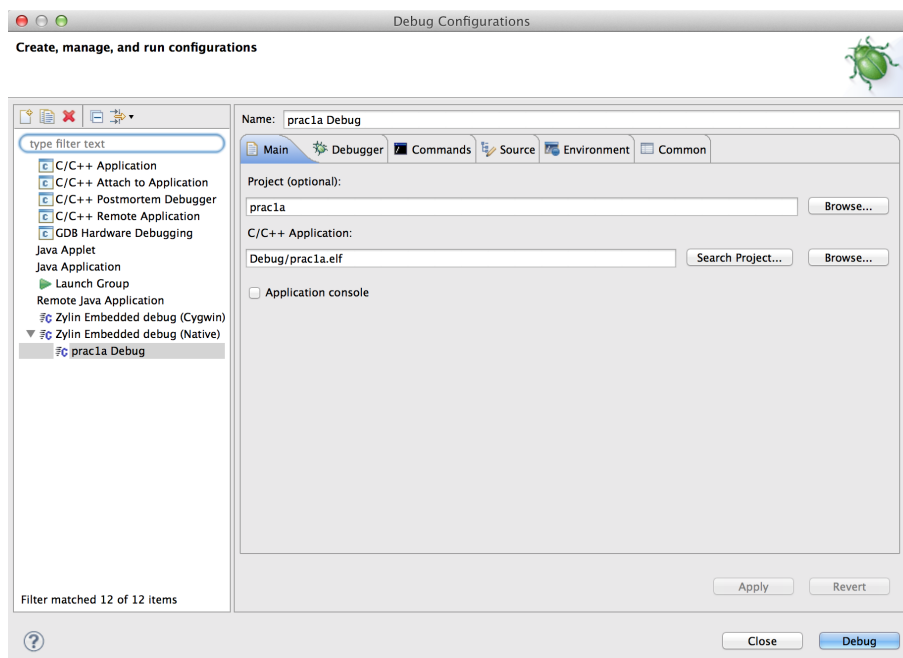


Figura 2.8: Aspecto que tendrá la ventana Debug Configurations cuando hayamos copiado el proyecto prac1a.

Código C	Ensamblador
<pre> #define N 8 int A[N]={7,3,25,4,75,2,1,1}; int max;  max=0; for(i=0; i&lt;N; i++){     if(A[i]&gt;max)         max=A[i]; } </pre>	<pre> .global start  .EQU N, 8  .data A:    .word 7,3,25,4,75,2,1,1  .bss max:  .space 4  .text start:     mov r0, #0     ldr r1,=max @ Leo la dir. de max     str r0,[r1] @ Escribo 0 en max                 @ Terminar de codificar </pre>

- b) Codificar en ensamblador del ARM un algoritmo de ordenación basado en el código del apartado anterior. Supongamos un vector A de N enteros mayores de 0, queremos rellenar un vector B con los valores de A ordenados de mayor a menor. Para ello nos podemos basar en el siguiente código de alto nivel. Es **OBLIGATORIO** escribir en memoria el valor de `ind` y `max` cada vez que cambien (es decir, no basta con actualizar un registro; hay que utilizar la instrucción `str` para actualizar su valor en memoria).

Código C	Ensamblador
<pre> #define N 8 int A[N]={7,3,25,4,75,2,1,1}; int B[N]; int max, ind;  for(j=0; j&lt;N; j++){     max=0;     for(i=0; i&lt;N; i++){         if(A[i]&gt;max){             max=A[i];             ind=i;         }     }     B[j]=A[ind];     A[ind]=0; } </pre>	<pre> .EQU N, 8  .global start  .data A:    .word 7,3,25,4,75,2,1,1  .bss B:    .space N*4 max:  .space 4 ind:  .space 4  .text start:     @ Terminar de codificar </pre>

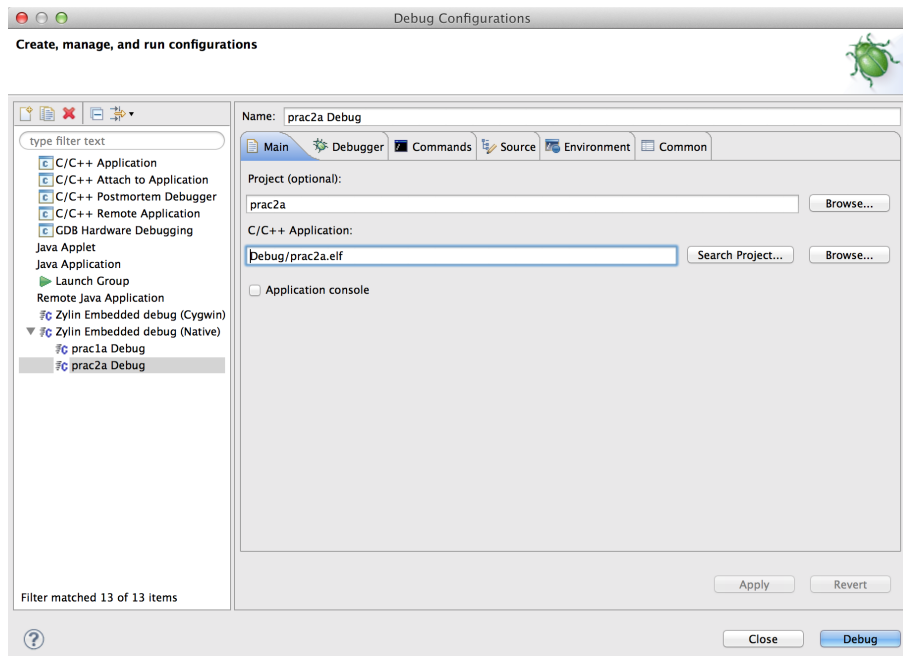


Figura 2.9: Aspecto que tendrá la ventana Debug Configurations tras la copia de la configuración prac1a Debug.