

Lab 2: Introduction to the addressing modes

2.1 Objectives

After having gained some familiarity with the ARM instruction set and the Eclipse development environment, this lab session is intended to strengthen the following aspects of assembly programming:

- Handling data vectors: learn how to load and store memory data knowing the starting address of a vector and its length.
- Translating high-level language statements into the ARM assembly language.
- Understanding the purpose of the compilation, assembly and linking steps.

2.2 Addressing modes

The addressing modes provide several alternative ways to calculate the effective memory address of an operand by using the information contained in the different fields of a machine instruction.

The ARM architecture has a rich set of addressing modes. Let's briefly review the three addressing modes of load/store instructions that are useful for the development of this lab session:

- **Indirect register.** The memory address to which want to access (i.e. the *effective memory address*) is contained in a register of the register file (called the *base register*). Figure 2.1 illustrates the result of executing the instruction `ldr r2, [r1]`

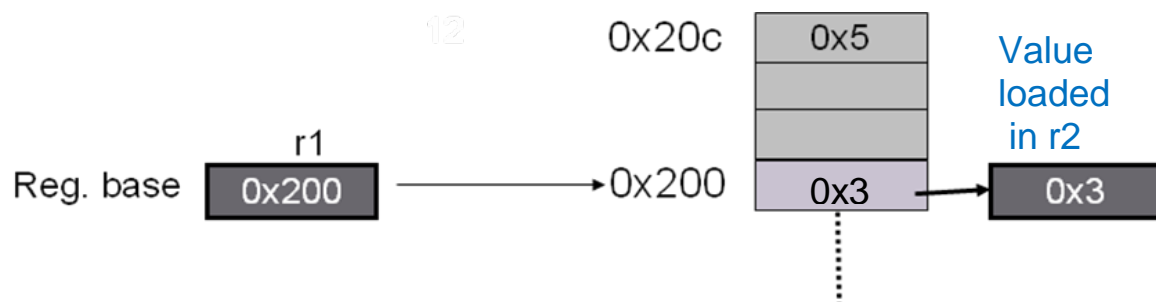


Figure 2.1 Example: executing the instruction `ldr r2, [r1]`. Here r2 is the destination register, and r1 the base register

- **Indirect register with offset.** The effective memory address is calculated by adding a constant (*offset*) to the address contained in the base register. The offset is encoded within the instruction as a 12-bit number, which is represented in 2's complement. Figure 2.2 illustrates the result of executing the instruction `ldr r2,[r1,#12]`.

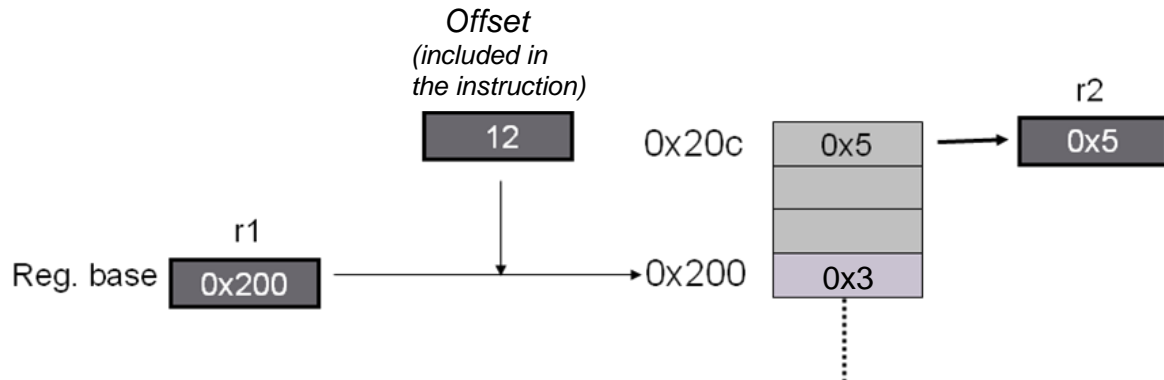


Figure 2.2 Example: executing the instruction `ldr r2,[r1,#12]`

- **Indirect register with register offset.** The effective memory address is calculated by adding the address stored in the base register to the integer value stored in another register (which plays the role of an offset). The second register may be optionally multiplied / divided by a power of 2. Actually, there are five possible operations that can be performed on the offset register (see Module 8), but we will only see here two of them¹:
 - Arithmetic shift right (ASR). The general form of this modifier is “ASR #n”. If included, the value provided by the offset register is previously shifted n bits to the right, which is equivalent to dividing by 2ⁿ. It should be observed that the contents of the offset register does not change.
 - Logical shift left (LSL). The general form of this modifier is “LSL #n”. If included, the value provided by the offset register is previously shifted n bits to the left, which is equivalent to multiplying by 2ⁿ. Figure 2.3 illustrates the result of the operation `str r3,[r5, r1,LSL #2]`

¹ Recall the “Shift” and “Shift Amount” fields that were described in module 8, when comparing the machine language and the assembly language of ARM.

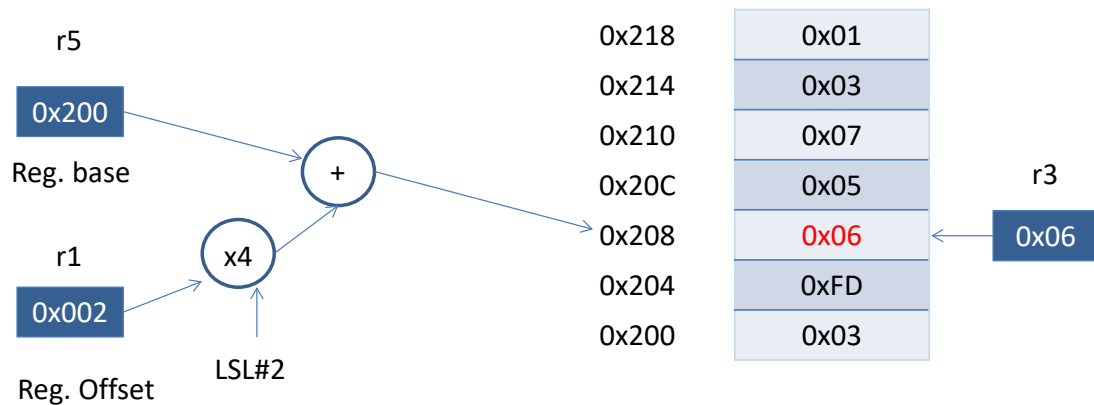


Figure 2.3 Example: executing the instruction `STR r3,[r5,r1,LSL #2]`. The value from `r1` is multiplied by 2^2 , and then added to the contents of `r5`, to compose the effective address. The contents of `r1` remains unchanged.

Some examples of load and store instructions with different addressing modes are shown below:

<code>LDR R1,[R0]</code>	Loads to R1 the contents of Mem[R0]
<code>LDR R8,[R3, #4]</code>	Loads to R8 the contents of Mem[R3+4]
<code>LDR R12,[R13, #-4]</code>	Loads to R12 the contents of Mem[R13-4]
<code>STR R2,[R1, #0x100]</code>	Stores to Mem[R1+256] the contents of R2
<code>STR R4,[R5,R1,LSL#2]</code>	Stores to Mem[R5+ (R1* 2^2)] the contents of R4
<code>LDR R11,[R3,R5,LSL#3]</code>	Loads to R11 the content of Mem[R3+ (R5* 2^3)]

Using an offset register is the recommended addressing mode to **traverse arrays**: we use a register (base register) to maintain the starting address of the array, while in the other one (offset register) we maintain the index of the component we are accessing. In this way, we just have to multiply the contents of the offset register (using LSL) by the appropriate data size (4 bytes in the case of integers), and add the result to the contents of the base register, to get the memory address of the desired item. The following example illustrates the utility of this addressing mode to traverse an *array*:

C code	Assembly code
<pre>int A[100]; for(i=0;i<100;i++){ ... = A[i] ...; }</pre>	<pre>/* load to r1 the starting address of A */ LDR r1,=A MOV r2,#0 @ initialize the index i for: CMP r2,#100 BGE end ... /* read the i-th element of A and load it to r3 */ LDR r3,[r1,r2,ls1#2] ... ADD r2,r2,#1 B for end:</pre>

As we have just seen, we must previously load the starting address of the array in a register (r1, in the above example). It is therefore necessary to use the pseudo-instruction:

LDR R<n>, =LABEL

which loads to register R<n> the address associated with *Label*.

Figure 2.4 illustrates the behavior of this pseudo-instruction, where it is assumed that the variable *A* has been previously initialized using the directive “*A: .word 6*”, and that this variable has been allocated to the memory address 0x208.

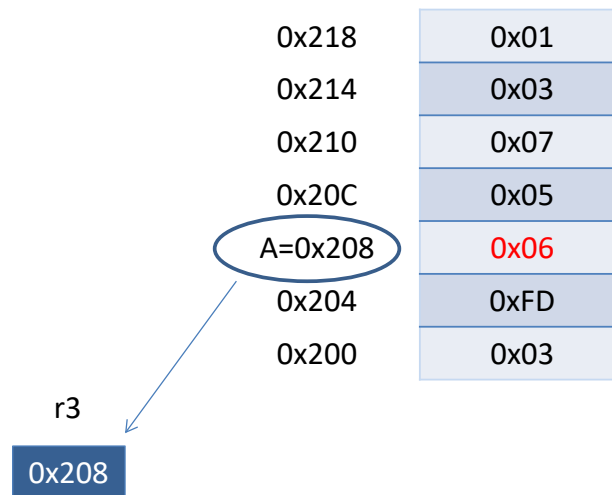


Figure 2.4 Example: executing the instruction **ldr r3,=A**

2.3 Sections of an executable file

Figure 2.5 illustrates the process of generating an executable file (also called executable image) from one or more source files, and some libraries. Each C code file is passed through the preprocessor, which modifies the original source code following preprocessing directives (# define, # include, etc.). The resulting code is then compiled, thus transforming it into assembly code. Finally, the executable file for the target architecture (ARM in our case) is generated by combining the different assembly files.

It is possible that an assembly program, either generated by a compiler or manually written by a programmer, reference some symbols that are still to be defined (e.g. the starting address of a subroutine that is defined in another file). This will make the assembler create an entry in the symbol table of the resulting object code to indicate that the symbol must be imported from another object file. This symbol table will also contain an entry for each exported symbol, these normally being either global functions or variables that we want to export so that they can be used in other files. External symbols will be fully studied in lab 4.

Object code files are binary files with a certain structure. In particular, we must know that these files are organized into sections, each one having a name. Usually there are three sections: **.text** for the code, **.data** for data (global variables) with an initial value, and **.bss** for uninitialized data.

The so called **linker** is the responsible for generating the final executable code. To do that, the linker will merge several object code files, from external libraries and / or from our compiled C or assembly source code, each with its own code and data sections. These sections will be mapped to different memory addresses to form the final executable sections, thus determining the memory map of the program. By the end of this process all the symbols will have been resolved.

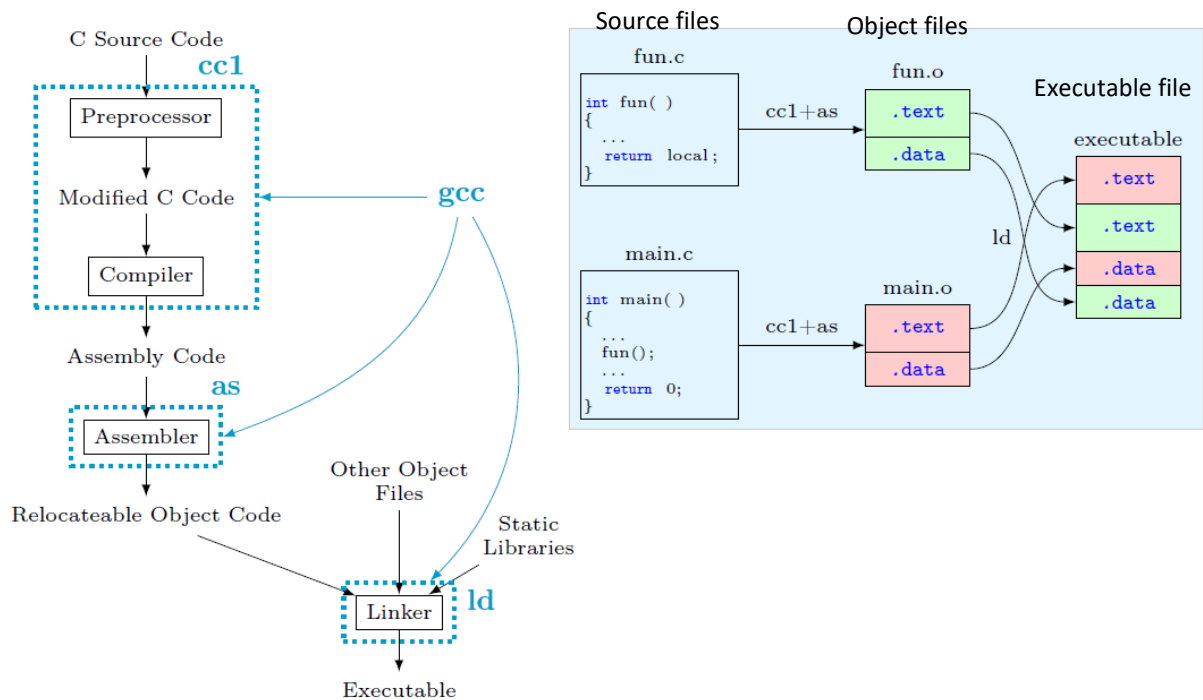


Figure 2.5 Generation of an executable file

As explained in Lab 1, to define the different program sections we just have to insert a line with the name of the section. From that moment the assembler will consider that the subsequent contents should be placed in the section with that name. For example, the program in Figure 2.6 has a **.bss** section in which four bytes are reserved for storing the result, a **.data** section to declare variables with an initial value and a **.text** section containing the program code. The directive **.equ** is used to declare constants (recall that these constants are not stored in memory).

Another relevant aspect of the code in Figure 2.6 is the use of the pseudo-instruction “`LDR R2, =A`”. In this case it is **mandatory** to use this pseudo-instruction, because the variable `A` is defined in a different code section, and thus its memory address will only be decided when the linker has merged the different code sections into a single executable file.

```
.global start

.equ ONE, 0x01

.bss
RES: .space 4

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

.text
start:
    MOV R0, #ONE
    LDR R2, =A           @ load the address of A to R2
    LDR R1, [R2]         @ load the value of A to R1
    ADD R2, R0, R1
    LDR R3, =B           @ load the address of B to R3
    LDR R4, [R3]         @ load the value of B to R4
    ADD R4, R2, R4
    LDR R2, =RES         @ load the address of RES to R2
    STR R4, [R2]         @ store the result (A+B+1) to RES
End: B .
.end
```

Figure 2.6. An example of code with sections.

To clarify the difference between the pseudo-instructions “`LDR R2, =A`” and “`LDR R2, A`” we can consider the program of Figure 2.7, which executes the same computation as the previous one. While “`LDR R2, =A`” would be valid in both programs, “`LDR R2, A`” would be valid only in the second one. The reason is that in Figure 2.7 we have a single section (`.text`), and thus the reference to variable `A` can be resolved by the assembler. So, in Figure 2.7, the pseudo-instruction “`LDR R2, A`” will be transformed into a `LDR` instruction that copies to `R2` the contents of the memory location that is three words above. On the contrary, if we try to use “`LDR R2, A`” in Figure 2.6, as a way to copy the value of `A` to register `R2`, the assembler will report an error, because it doesn’t know where the variable `A` will be finally mapped to memory.²

² Recall that the assignment of memory space to each program section is under the responsibility of the linker. Thus, when processing a section, it is impossible for the assembler to know the address of any instruction or data that lie in a different section.



```
.global start

.equ ONE, 0x01

.text
RES: .space 4

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

start:
    MOV R0, #ONE
    LDR R1, A           @ load the value of A to R1
    ADD R2, R0, R1
    LDR R4, B           @ load the value of B to R4
    ADD R4, R2, R4
    STR R4, RES         @ store the result (A+B+1) to RES
End: B .
.end
```

Figure 2.7 A program with a single section that gives the same result as the program of the previous figure.

2.4 Preparing the development environment

REMARK: if you are working with the “predefined workspace”, you can proceed directly to section 2.5.

In lab 1 we studied how to create an Eclipse project with the GNU ARM plugin. In this session we might repeat the same procedure to add a new project to our workspace for each program that we have to develop. However, in many situations it will be more convenient to create new projects starting from copies of previously existing ones. To illustrate this possibility, we will create a new project, `prac2a`, from the previous project `prac1a`, following this procedure:

1. Open Eclipse and select your workspace in the Workspace Launcher window.
2. Select the C/C++ perspective.
3. Open the project to be copied, `prac1a`, and click on it with the right mouse button. This will display a menu similar to Figure 2.8, where you must select the Copy option.
4. Click on the Project Explorer window with the right mouse button, which will display the same menu again, and select the Paste option. This will open a window that will allow us to select a name for the new project, as illustrated in Figure 2.9.
5. It is recommended to delete the Debug folder of the new project. The reason is that this folder contains the object files and the executable of the previous project. The first compilation of our project will recreate the Debug folder to

store there the new object files and the new executable.

6. Finally, click in the project with the right mouse button, and select Properties → C/C++ Build → Refresh Policy. The box will display the name of the original project. Select it and click the Delete button. Then click the Add Resource button ... and select the new project in the window that will open, as shown in Figure 2.11. Click Ok. This will automatically refresh our new project within the Project Explorer when we launch the compilation.

On completion of the previous steps, our new project will be created. However, as illustrated in Figure 2.10, the new project still has a copy of the source files of the original one. Usually, we will be interested in renaming the source files, e.g. rename `prac1a.asm` to `prac2a.asm`. This can be done by selecting the file with the right mouse button, and clicking Rename on the displayed menu. Another alternative consists of deleting the original source files, using the Delete option of the same menu, and adding later the new source files, exactly as we did in lab session 1.

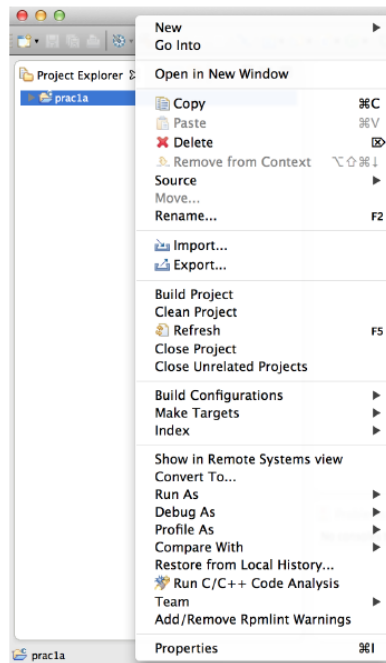


Figure 2.8 The menu displayed after clicking the right mouse button on a project of the workspace.

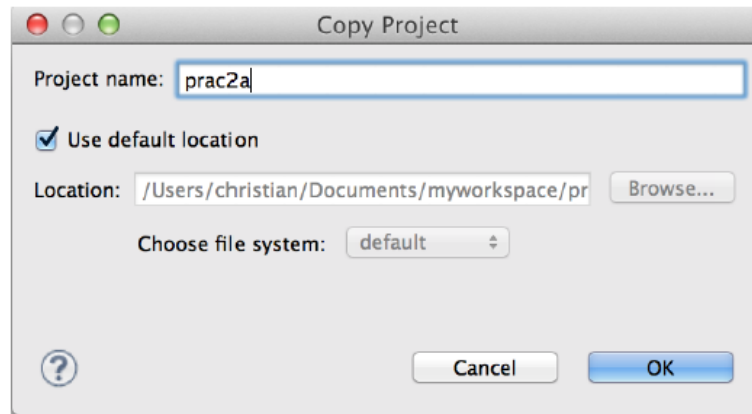


Figure 2.9 Selecting a name for the new project

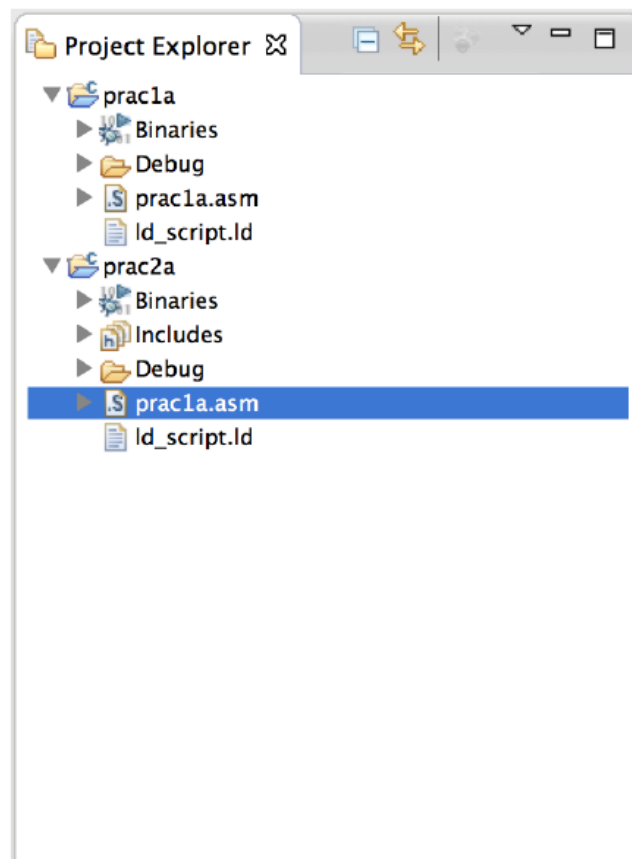


Figure 2.10 Aspect of the Project Explorer after having copied the project prac1a as prac2a.

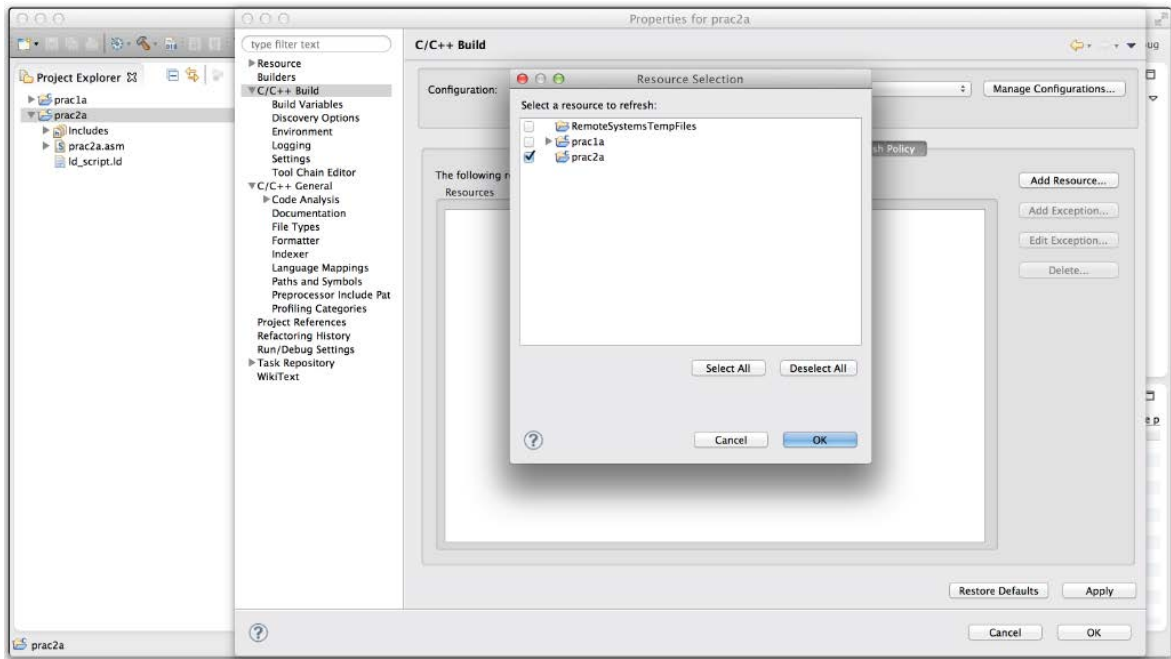



Figure 2.11 Selecting the refresh options of our new project.

2.4.1 Copying the debug configurations

In order to simulate and debug our project, we must create the debug configurations. Of course, we can do it as explained in the previous lab session. However, this time we will learn how to copy the debug configurations from a previously existing project of our workspace. To do this, we must open the project from which we are going to obtain the debug configurations; otherwise, such configurations will not be visible. Next, we must follow this procedure:

- Switch to the Debug perspective.
- Select Run → Debug Configurations ... This will open the window shown in Figure 2.12. The configuration that we want to copy (prac1a Debug, in this case) can be seen in the left side of this window.
- Select the desired configuration and click the button (). This will create an exact copy of this configuration, in which we have to change the project name (prac1a to prac2a), and the name of the executable file (Debug\prac1a.elf to Debug\prac2a.elf). Now click Apply, and the window will look as shown in Figure 2.13.
- Finally, we must click the Close button. It is important to take into account that our executable file has not been created yet (recall that the executable file will be created when we successfully complete the Build Project step). Therefore if, by mistake, we click the Debug button, an error will be reported. We can ignore this error message.

From this moment our original project is no longer needed, so we can close it by activating the C/C++ perspective, clicking on the project with the right mouse button, and selecting Close Project on the displayed menu.

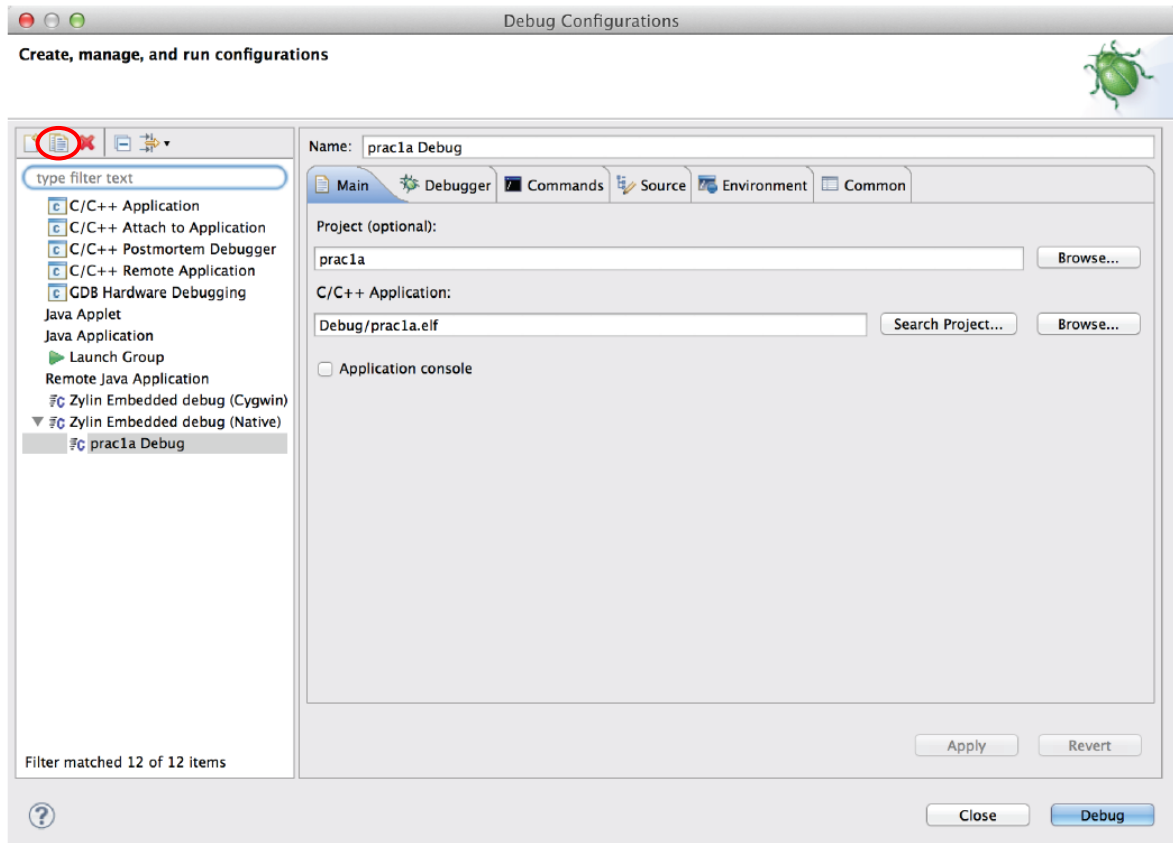


Figure 2.12 The Debug Configurations window.

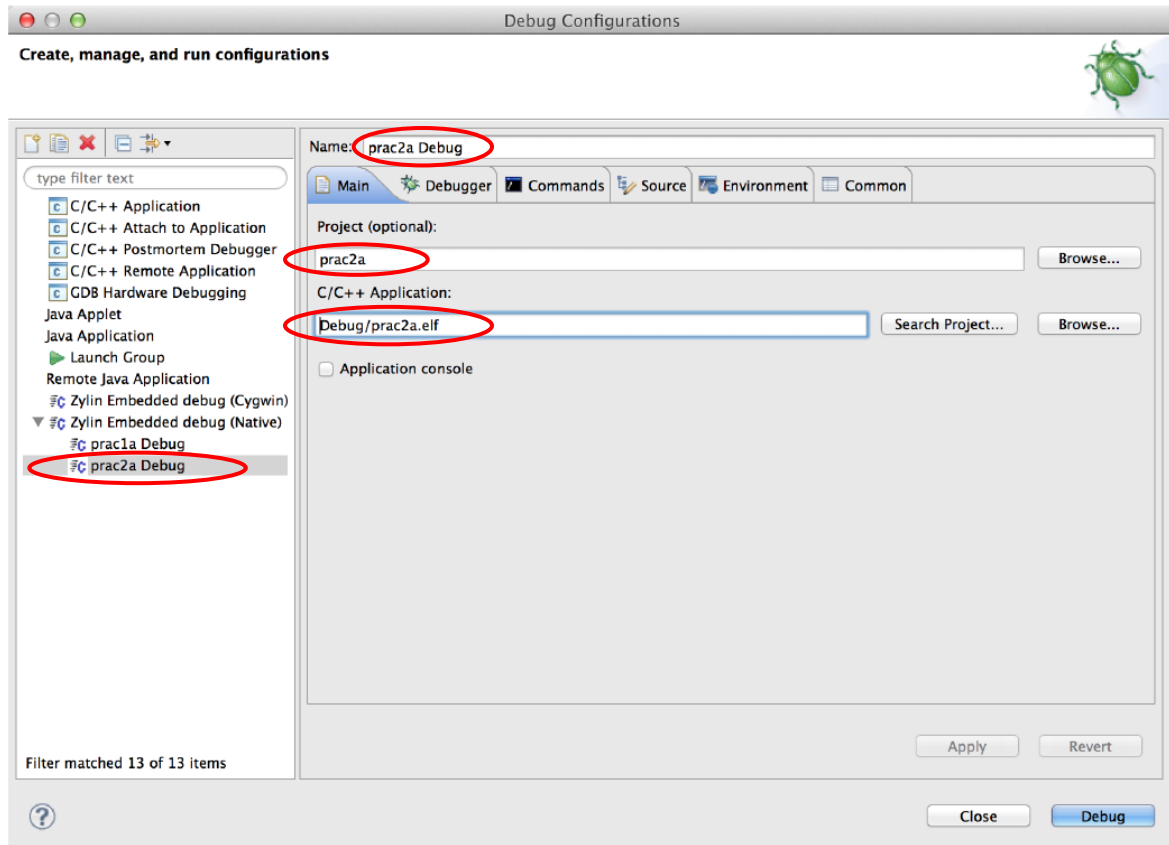


Figure 2.13 The Debug Configurations window after the creation of the copied configuration prc2a Debug.

2.5 Development of the lab session

Before the lab session, the student must have developed and tested the codes corresponding to the programs a) and b) mentioned below. During the lab session some modification of one of these programs will be proposed to the student, who must code, test and debug the corresponding program in two hours.

In **EACH** program the student must define three sections:

- **.data** for variables with an initial value(declared with **.word**)
- **.bss** for variables without an initial value (declared with **.space**)
- **.text** for the code

- a. Write an ARM assembly program that implements the following C code, which is intended to find the maximum value in a vector of N positive integers. This value must be stored in the variable **max**. You **MUST** write the value of **max** to memory each time it changes (i.e., it is not enough to update a register; you have to execute a **str** statement).



```
#define N 8
int A[N]={7,3,25,4,75,2,1,1};
int max;

max=0;
for(i=0; i<N; i++){
    if(A[i]>max)
        max=A[i];
}
```

```
.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 @Read address of max
    str r0,[r1] @store 0 to max

@ Complete the rest of the code
```

- b. Write a sorting algorithm in the ARM assembly language, using the code you have developed in the previous section. Suppose we have a vector, A, of N integers greater than 0. Your program must fill a vector, B, with the values from A in descending order. To do this, follow the high-level code that is shown below. You **MUST** store to memory the value of **ind** and **max** each time they change (i.e, it is not enough to update a register, you have to execute a **str** statement).

```
#define N 8
int A[N]={7,3,25,4,75,2,1,1};
int B[N];
int max, ind;

for(j=0; j<N; j++){
    max=0;
    for(i=0; i<N; i++){
        if(A[i]>max){
            max=A[i];
            ind=i;
        }
    }
    B[j]=A[ind];
    A[ind]=0;
}
```

```
.global start
.EQU N, 8

.data
A: .word 7,3,25,4,75,2,1,1

.bss
B: .space N*4
max: .space 4
ind: .space 4

.text

start:

@ Complete the rest of the code
```

- c. Modify the program of one of the two previous sections, according to the instructions that will be given at the lab.