Lab 4: Implementing projects with several source files. Combining C and Assembly languages.

4.1. Main goals

In this lab session we will broaden and deepen our knowledge of ARM assembly programming, building projects that enclose several source files, some of them written in C language and the others in assembly language. Specifically, our goals in this session are:

- Understand the differences between local and global variables.
- Understand the meaning of static symbols (variables and functions).
- Analyze the problems that appear when using several source files within a project, and understand how to resolve the symbols.
- Understand the relation between our own written C program and the machine code generated by a *gcc* compiler.
- Being able to use both variables and functions defined in C from an assembly code, and vice versa.
- Have an initial contact with the representation of structured data types in high level languages.

In this session, the student will create a program that combines C and assembly languages, in which C functions can invoke assembly routines and vice versa.

4.2. Local variables

Let us start with the following example, which introduces the concept of local variables:

```
int funA( int a1, int a2, int a3, int a4, int a5);
```

This declaration states that the function will have, at least, 5 local variables, named *a1-a5*. We shouldn´t confuse *local variables* with *function parameters*, which correspond to the initial values that we give to these 5 variables. For example, we could invoke the function as follows:

```
a = funA(1,3,6,4,b)
```

where *a1-a4* take initial values 1, 3, 6 and 4 respectively, whereas *a5* takes the value of variable *b*.

In every invocation of the function, a private version of the local variables is created. For example, suppose that *funA* is a recursive function (i.e., it invokes itself), so during the execution of the program we will have several *funA* active instances at the same time. Each *funA* instance should have its own local variables (*a1-a5*) allocated in a private space in memory. The most natural way for managing this is to use the subroutine *activation record* (also called *stack frame*).

Note that the variables that we have been using thus far were *global variables*. Those variables are allocated in memory sections *.data* or *.bss*, and they stay in memory from the beginning until the end of the execution. Conversely, local variables are created during the subroutine prolog and destroyed during its epilog.

4.2.1 Subroutine *activation record* survey

Figure 4.1 illustrates a detailed scheme of the *activation record* organization, which includes space for storing registers, local variables and function parameters. After creating the *activation record* and saving the context, the subroutine allocates the required space for the local variables. According to the AAPCS standard, the size of this space must be a multiple of 4 bytes. The specific placement of the local variables within this space is determined by the programmer. During the execution of the subroutine body, local variables are addressed with negative offsets relative to FP.

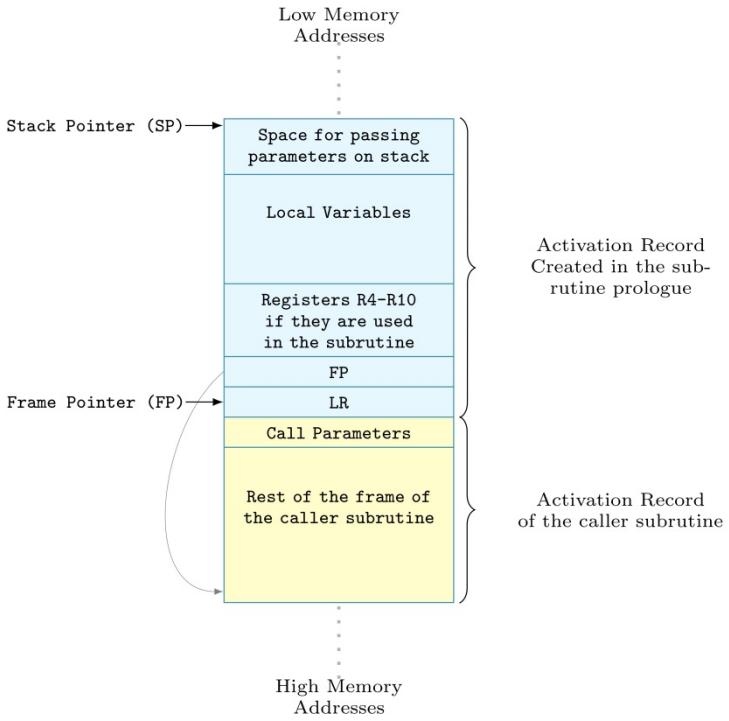

Figure 4.1. Detailed organization of the *activation record*

We should highlight an important exception that we can find frequently in the memory placement of local variables: Suppose that SubA calls SubB with 5 parameters, in which case the fifth parameter must be passed through the top of SubA stack frame. In this case, it would be a waste if SubB reserved space for the corresponding local variable in its own *activation record*, so, instead, it takes advantage of the space reserved by SubA for the fifth parameter. Note that, obviously, the variable must be addressed from SubB using a positive offset relative to FP.

The prolog and the epilog that we employed in the previous session are totally compatible with the *activation record* shown in Figure 4.1. We should just point out a minor aspect concerning the third instruction that we used in the prolog:

```
SUB SP, SP, #4*AmountOfExtraWords       @ Required extra space
```

Now, the variable *AmountOfExtraWords* must include not just the space for the parameters passed to invoked subroutines, but also the space for its own local variables.

After the prolog, the first task of the subroutine is to initialize those local variables that are defined with an initial value. Among these, the subroutine must include the variables declared in the function prototype, which are initialized with the corresponding function parameters.

4.2.2 Example

Let us use the following C function, which returns the highest of two integer numbers:

```
int Highest(int X, int Y){
      int m;
      if(X>Y)
      m = X;
      else
      m = Y;
      return m;
}
```

This function owns three local variables, so it needs to reserve 12 additional bytes in the stack. It can, for example, place *X* first ([fp, #-4]), *Y* next ([fp, #-8]), and finally *m* ([fp, #-12]). The resulting assembly code would then be:

```
1  Highest:
2    push {fp}
3    mov fp, sp
4    sub sp, sp, #12
5    str r0, [fp,#-4] @ initialize X with the first parameter
6    str r1, [fp,#-8] @ initialize Y with the second parameter
7
8    @ if( X > Y )
9    ldr r0, [fp,#-4] @ r0 ← X
10   ldr r1, [fp,#-8] @ r1 ← Y
11   cmp r0, r1
12   ble ELS
13   @ then
14   ldr r0, [fp,#-4] @ m = X;
15   str r0, [fp,#-12]
16   b RET
17   @ else
18 ELS:
19   ldr r0, [fp,#-8] @ m = Y;
20   str r0, [fp,#-12]
21   @ return m
22 RET:
23   ldr r0, [fp,#-12] @ return value
24   mov sp, fp
25   pop{fp}
26   mov pc, lr
```

This code translates line by line the original C code to assembly language. For that reason, it may seem quite redundant and inefficient. For example, line 9 loads *X* into *r0*, but line 5 has just initialized that local variable with *r0*. Given that r0 remains untouched in between, we could simply remove line 9. However, as we will see in next section, this way of translating the code from C to assembly language is the easiest to understand and debug, so we strongly recommend the student to employ this programming style while learning.

4.3. Translating C to assembly language

Compilers are usually structured in three parts: *front end* (also called *parser*), *middle end*, and *back end*. The first part is in charge of checking whether the code is syntactically correct and translating it into a representation independent of the employed high-level language. The *middle end* is responsible of analyzing the code from the previous step and performing some optimizations on it, which may pursue different goals, like, for example, reducing execution time, decreasing memory utilization, or lowering energy consumption. Finally, the *back end*, is in charge of generating the code suitable for the target architecture. More specifically, this step often produces assembly code which is then translated into the final machine code by the assembler.

If the *middle end* performs no optimizations, the assembly code that we obtain is a line-by-line translation of the original high-level program, that is, each C sentence is translated independently of the others. Figure 4.2 compares the translation of a simple C program to assembly code without optimizations (-O0) and with level 2 optimizations (-O2). In the non-optimized version, we can easily identify the set of assembly instructions that correspond to each C sentence. The process is the following: variables in the right part of the sentence are loaded into registers, the proper operation is then performed, and the result is finally stored in the memory variable specified in the left part.

Note that this line-by-line translation is necessary for debugging. For example, if we are debugging a code, we can be interested in stopping the execution at a specific C sentence, modify the values of the variables, and resume execution. In the non-optimized version this is easily achievable. However, in an optimized version, it may be the case that the instruction that loads the variable into a register has been removed because the register has not been touched and preserves its original value. We can see this situation in the optimized version of the code of Figure 4.2, where the variable *i* is not accessed within the loop because a register is being used as the loop counter instead. The value of *i* is only updated at the exit of the loop. So, if we were debugging this optimized version and we tried to modify the value of *i* inside the loop, the modification would have no effect at all.

Another typically tricky situation is to find unexpected branches in the debugger, due to code reorganization derived from optimization. For example, if the optimization has moved a given instruction to a previous position, the debugged execution will jump backwards, for no apparent reason. This does not mean that the program contains errors, but that the C code does not longer have a straightforward relation with the machine code due to the optimization.

As we said above, for pedagogical reasons, we recommend the student to get used to writing the assembly code in a similar way as the code from the left part of Figure 4.2. This way, every variable should get updated in memory as soon as it gets modified. Furthermore, a local variable should always be loaded from memory right before using it, even though it is already in a register due to a previous operation.

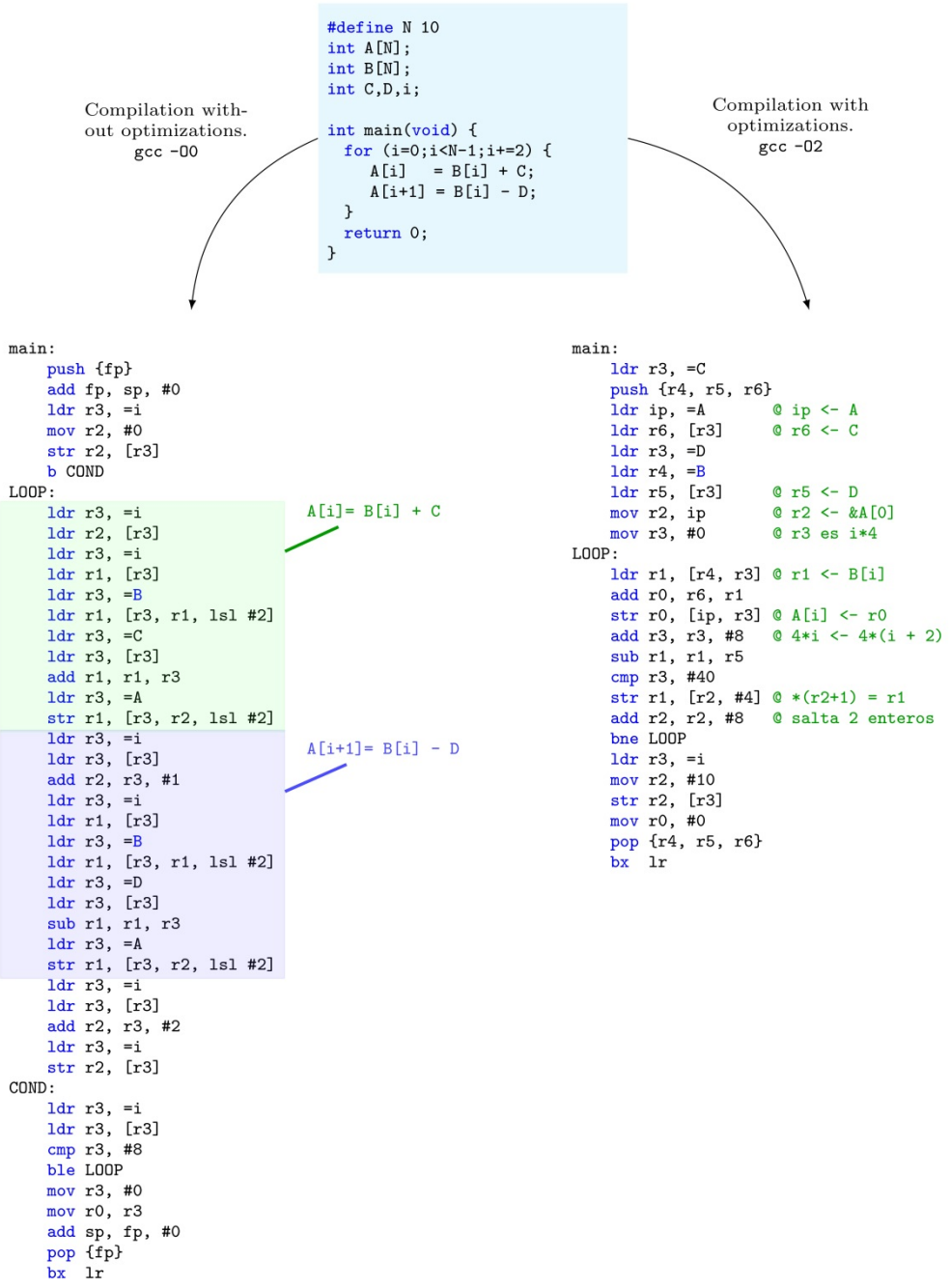

Figure 4.2 Translation of a C code into assembly language using gcc-4.7. Optimized code (-O2) in the right part of the figure. Non-optimized code (-O0) in the left part.

4.4. Memory accesses: size and alignment

So far, all memory accesses performed with load/store instructions were word-sized, that is, moving a 32-bit datum from register to memory or vice versa. However, most programming languages allow working with other data sizes. For example, C language provides the following access sizes: *char*, with a size of 1 byte; *short int*, with a size of 2 bytes; *long int*, with a size of 4 bytes (i.e. 1 word); and *long long int*, with a size of 8 bytes. Besides, C also provides an *unsigned* modifier for encoding an unsigned integer of 1, 2, 4 or 8 bytes.

For being able to efficiently work with these data types, machine language must provide new instructions that access memory with the proper size. In ARMv4, load and store instructions can use four different sizes: 1, 2 (half word), 4 (word) and 8 (double word) bytes. For specifying the access size, the assembly language allows adding a suffix to ldr/str instructions, as follows:

- **LDRB**: load an unsigned integer of size 1 byte. The datum is copied in the 8 least significant bits of the destination register, and the rest is filled with 0s.
- **LDRSB**: load a signed integer of size 1 byte. The datum is copied in the 8 least significant bits, and the rest is sign-extended.
- **STRB**: store in memory an integer of size 1 byte, obtained from the 8 least significant bits of the source register.
- **LDRH**: load an unsigned integer of size 2 bytes. The datum is copied in the 16 least significant bits and the rest is filled with 0s.
- **LDRSH**: load a signed integer of size 2 bytes. The datum is copied in the 16 least significant bits and the rest is sign-extended.
- **STRH**: store in memory an integer of size 2 bytes, obtained from the 16 least significant bits of the source register.
- **LDRD**: loads two consecutive registers with two consecutive words from memory. A restriction is established so that that the first of the two registers must be an even register (for example R8 and R9).
- **STRD**: Opposite instruction to the previous one.

In all these instructions, all addressing modes studied in previous lab sessions are supported (i.e. register indirect, register plus immediate offset and register plus register offset).

One important aspect about the ARMv4 that we should highlight here is that this architecture establishes some alignment constraints to memory accesses. A memory access is considered *aligned* if the address is a multiple of the accessed datum size in bytes. For example, a 1 byte datum access has no restrictions at all, as all addresses are multiple of 1. However, a half-word datum access will only be valid for addresses multiple of 2, whereas a word or double-word datum access will only be valid for addresses multiple of 4 and 8 respectively. If an alignment constraint is violated, a *data abort* exception will happen. Given that exception handling is out of the scope of this subject, the student would have to restart the simulation in such situation.

Next we illustrate several examples:

```
LDRB R5, [R9]           @ It loads into the 8 least significant  bits of R5, 8 bits
                        @ from Mem(R9), and resets the remaining bits of R5.
LDRB R3, [R8, #3]       @ It loads into the 8 least significant bits of R3, 8 bits from
                        @ Mem(R8+3), and resets the remaining bits of R3.
STRB R4, [R10, #0x200]  @ It stores into Mem(R10+0x200) the 8 least significant bits of R4.
STRB R10, [R7, R4]      @ It stores into Mem(R7+R4) the 8 least significant bits of R10.
LDRH R1, [R0]           @ It loads into the 16 least significant bits of R1, 16 bits from
                        @ Mem(R0), and resets the remaining bits of R1.
LDRH R8, [R3, #2]       @ It loads into the 16 least significant bits of R8, 16 bits
                        @ from Mem(R3+2), and resets the remaining bits of R8.
STRH R2, [R1, #0X80]    @ It stores into Mem(R1+0x80) the 16 least significant bits of R2.
LDRD R4, [R9]           @ It loads into R4 32 bits from Mem(R9), and into R5 32 bits
```

```
                        @ from Mem(R9+4).
STRD R8, [R2, #0x28]    @ It stores into Mem(R2+0x28) the 32 bits of R8, and into
                        @ Mem(R2+0x2C) the 32 bits of R9.
```

4.5. Combining several source files

The projects that we have created thus far were composed of a single source file. However, real projects are usually composed of several source files. Besides, it is common that most source files are programmed in C language, and only those parts of the program that require special treatment (for improving efficiency or for using special instructions) are programmed in assembly language. Note that those files programmed in C language need to be compiled, whereas those files programmed in assembly language only need to be assembled.

When a project is composed of several source files, some variables/subroutines will be used/called from files different to the one where they are defined. As we already saw in session 2, each file is independently compiled and assembled. It is only at the final stage of the executable file generation process, i.e. the *linking stage*, that all the source files are combined together. Thus, in the linking stage, all cross references among source files must be resolved. The main goal of this section is to study this mechanism and its influence on the generated code.

4.5.1 Symbol table

The object file is independent of the language employed for writing the source file. It consists of a structured binary file that includes a list of the program sections with their contents, as well as several additional structures. One of these structures is the *symbol table*, which, as its name suggests, contains information about the symbols that the source file uses. Every object file has its own *symbol table*. These tables are used during the linkage for resolving all pending references.

The *nm* utility allows the programmer to obtain the symbol table of a given *elf* object file. For example:

```
> arm-none-eabi-nm -SP -f sysv example.o
Symbols from example.o:

 Name          Value  Class   Type       Size       Line    Section

globalA    |00000000  | D   | OBJECT   |00000002  |          |.data
globalB    |          | U   | NOTYPE   |          |          |*UND*
main       |00000000  | T   | FUNC     |00000054  |          |.text
printf     |          | U   | NOTYPE   |          |          |*UND*
```

Without knowing the source file, this information reveals that:

- There exists a symbol called *globalA*, mapped at entry 0 of the data section (.data) and with a size of 2 bytes.

- There is an undefined symbol called *globalB*, therefore we must import it from another file.

- There exists a symbol called *main*, located at entry 0 of the code section (.text) and with a size of 54 bytes. It corresponds to the entry point of the C program.

- Finally, there is an undefined symbol called *printf*, which belongs to a standard C library and hence must be imported from it.

4.5.2 Global symbols in C language

Table 1 shows an example that combines two different C files. Each one uses global symbols which are defined in the other. The two source files will be linked in the final stage, making up a single executable file.

In C language, all functions and global variables are *exported global symbols* by default. In order to use a function which is defined in another file, a forward declaration of the function must be done. For example, for using function *FOO*, defined in another file, which receives or returns no parameters, we must use the following forward declaration before invoking it:

```
extern void FOO(void);
```

where the *extern* modifier is optional.

In the case of global variables, a forward declaration must also be done for using a variable defined in another file. For example, for using global variable *aux*, defined in another file (or later in the same file), we must use the following forward declaration before using it:

```
extern int aux;
```

If the *extern* modifier is not used, the variable is considered as a *COMMON* symbol. The linker resolves all *COMMON* symbols with identical name by the same address, allocating an amount of memory which suffices for the biggest one. For example, if there are two declarations of a global variable, *char Name[10]* and *char Name[20]*, the linker will use one unique *COMMON* symbol for both and allocate 20 bytes in memory.

If the programmer wants to restrict the visibility of a function or a global variable to the source file where it was declared, the *static* modifier must be employed. The compiler will not include a *static global variable* in the symbol table of the object file. This allows the programmer to declare two global variables with the same name, each in a different source file.

Table 1. Example: exporting symbols

// *fun.c* file

```
// global variable defined in another file
extern int var1;

// restricted global variable (can only be
// accessed from fun.c)
static int var2;

// forward declaration of a global function
void one(void)

// restricted function
Static void two(void)
{
   …
   var1++;
   …
}

void fun(void)
{
   …
   // access to the only var1 that exists
   var1+=5;
   // access to var2 of fun.c
   var2=var1+1;
   …
   one();
   two();
   …
}
```

// *main.c* file

```
// global variable defined later in this file
extern int var1;

// restricted global variable (can only be
// accessed from main.c)
static int var2;

// forward declaration
void one(void)

// forward declaration
void fun(void)

int main(void)
{
   …
   // access to the only var1 that exists
   var1=1;
   …
   one();
   fun();
   …
}

// var1 definition
int var1;

void one(void)
{
   …
   // access to var1
   var1++;
   // access to var2 of main.c
   var2=var1-1;
   …
}
```

4.5.3 Global symbols in assembly language

Unlike C, in assembly language symbols are local by default, i.e. invisible from another file. For turning them into global symbols we must export them using the *.global* directive. The *start* symbol, which specifies the entry point to a program, constitutes a case of particular relevance, since it must always be declared as *global*. One of the reasons is to avoid the existence of more than one *start* symbol.

Moreover, when the programmer wants to use a symbol defined in another file, the *.extern* directive must be used. For example:

```
    .extern FOO;      @ an extern symbol is made visible.
    .global start;    @ a local symbol is exported.

start:
    bl FOO;
    …
```

4.5.4 Combining C and assembly languages

If we want to use in a C program, an *exported global symbol* defined in an assembly file, a forward declaration must be used. Note that the symbol is associated with the address of the identifier.

If the identifier is a subroutine, then the address corresponds to the starting address of the subroutine. In the C file, we must declare the function using the same name as the extern symbol. Moreover, we must define the type of all input and output parameters of the function, since this information is needed by the compiler. For example, if we want to use a subroutine *FOO* with two input parameters and no output parameter, we must use the following forward declaration:

```
extern void FOO(int ,int);
```

If the identifier is a variable, the symbol is associated with its address. For importing the symbol from the C program, the variable must be declared as *extern*. Like in the previous case, the programmer must indicate the variable type. For example, if we want to use a short integer variable *var1*, we must use the following forward declaration:

```
extern short int var1;
```

Suppose now that there is a table defined in an assembly code, which we want to access from a C code. In the assembly program, we must assign a label to the table and export the label with the *.global* directive. In this case, the symbol is associated with the address of the first byte of the table. Furthermore, in the C program, we must declare an array with the same name as the label, and use the *extern* modifier.

4.5.5 Symbol resolution

Figure 4.3 illustrates an example of the symbol resolution procedure of a project with two source files: an assembly program, called *init.s*, and a C program, called *main.c*. The first code has a global symbol called *MYVAR*, which corresponds to a label in the *.data* section, with a size of 4 bytes, and an initial value of 0x2. In the second code, an integer variable, called *MYVAR*, is forward declared as *extern*.

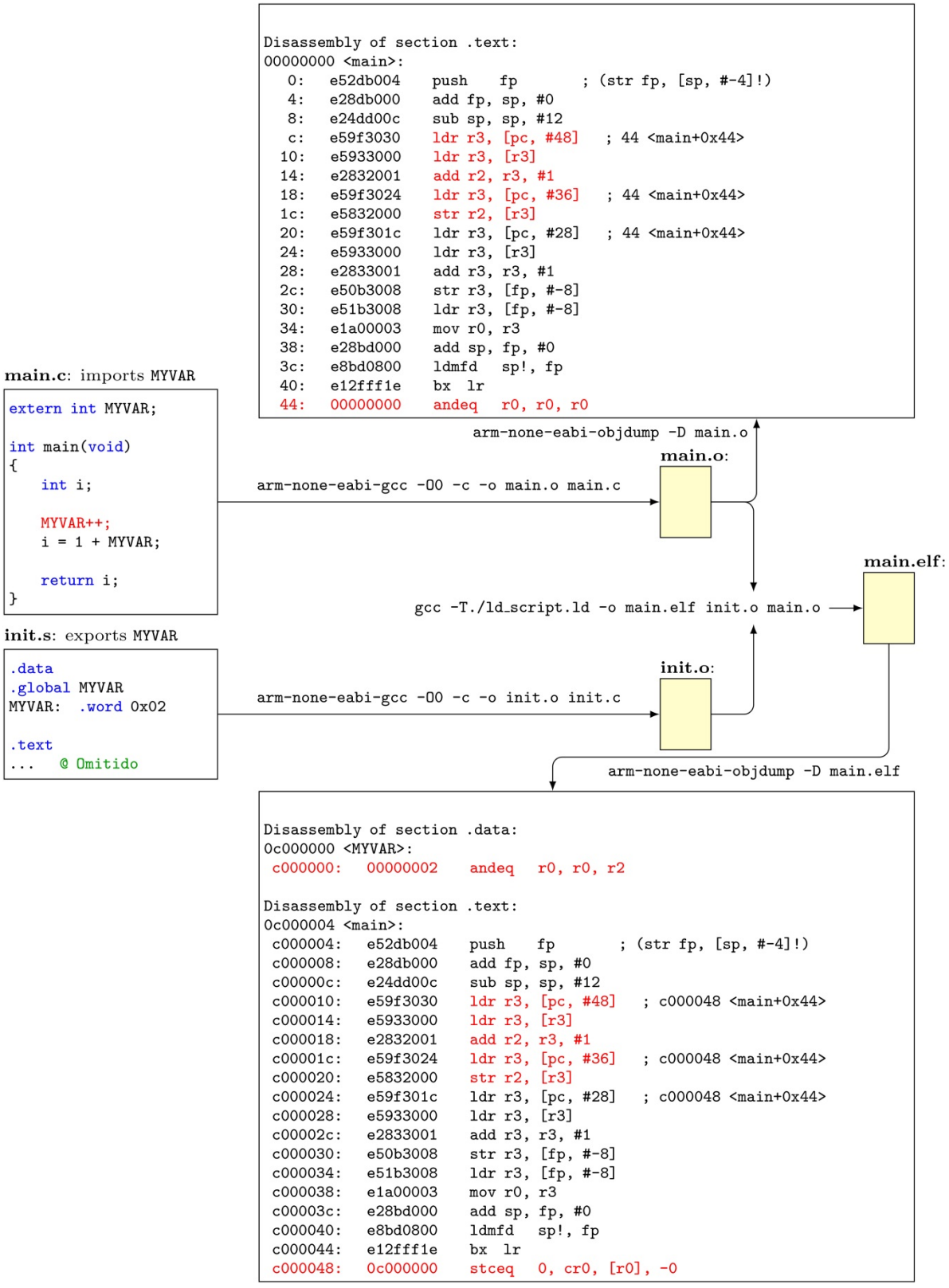

Figure 4.3 Example of symbol resolution.

These two files are independently compiled, producing two object files, *init.o* and *main.o*. The top of the figure shows the *main.o* file disassembled, which can be obtained as follows:

```
> arm-none-eabi-gcc –O0 –c –o main.o main.c
```

```
> arm-none-eabi-objdump –D main.o
```

Highlighted in red are shown the assembly and C instructions related with the access to *MYVAR* (note that the C operation `MYVAR++` is equivalent to `MYVAR=MYVAR+1`). In assembly language, this simple C operation is translated into several instructions. First, we have to load the contents of *MYVAR* from memory into a register. The problem is that the address of *MYVAR* is unknown by the compiler, since this symbol will not be resolved until linking time. How can the compiler load the datum at *MYVAR* without knowing its address? The solution is the following:

The compiler saves some space at the end of *.text* section for storing the so-called *literal pool*. At linking time, when the addresses of the symbols get resolved, the address where *MYVAR* is mapped in memory will be stored in one entry of this pool. In the example from Figure 4.3, position *0x44* of the *.text* section of file *main.o* holds the entry for *MYVAR*. The distance to this position from any other position of the same section is independent of the linkage, so the compiler can load the address from the adequate *literal pool* entry, using *PC* as the base register.

The first red instruction from the code on the top of Figure 4.3 (`ldr r3, [pc, #48]`) loads the contents of position *0x44* into r3, so the address where *MYVAR* is mapped is now in r3. The second red instruction (`ldr r3, [r3]`) loads into r3 the value of *MYVAR*, and the next one (`add r2, r3, #1`) adds 1 to this value and saves it into r2. Finally, we load again into r3 the address of *MYVAR*, and update its contents with the value of r2.

Note however that, in *main.o* (top part of Figure 4.3), position *0x44* contains a 0 value instead of the address where *MYVAR* is mapped! Why does this happen? The reason is that the address of *MYVAR* is unknown at compilation time. Thus, the compiler resets that position (*0x44*) and includes in the *relocation table* an entry for *MYVAR*, with the subsequent information:

```
>  arm-none-eabi-objdump -r main.o

    main.o:  file format elf32-littlearm

    RELOCATION RECORDS FOR [.text]:
    OFFSET   TYPE              VALUE
    00000040      R_ARM_V4BX       *ABS*
    00000044      R_ARM_ABS32      MYVAR
```

The last entry of the *relocation table* tells the linker to write in entry *0x44* of the *.text* section a 32-bit unsigned integer containing the address of *MYVAR*.

The bottom part of Figure 4.3 illustrates the disassembly version of the final executable program. The *.text* section of *main* is allocated in memory address *0x0C000004.* Thus, entry *0x44* is mapped in memory position *0x0C000048* and contains the value *0x0C000000.* This value corresponds to the memory address where *MYVAR* is located in memory, and that address contains a *0x2* value.

4.6. Launching a program in C language

A C program is made up by one or more functions. Among them, there always exists a *main* function that constitutes the program entry point. Like any other function, *main* will create its *activation record* at the stack, and then, when it finishes execution, it will return to the place pointed by the *LR* register.

Before the system can launch the *main* function, the stack must have been initialized. For that reason (and some others that are out of the scope of this subject), the program does not start in the *main* function, but in a special boot code, called *C Real Time 0* or just *crt0* in the *GNU toolchain*. This code is defined within the operating system. However, given that in this subject we are in a *bare metal environment* (i.e. a system without operating system), we must also create the boot code. Table 2 provides an example of this code, which we will henceforth use for our programs.

Table 2. Example of boot code

```
    .extern main
    .extern _stack
    .global start

start:
    ldr sp,=_stack
    mov fp,#0

    bl main

End:
    b End
.end
```

4.7. Composite data types

High level languages like C allow the programmer to use *composite data types*, like arrays, structures or unions. Next, we will study the low level implementation of this kind of data.

4.7.1 Arrays

An array is a homogeneous structure, made up by a collection of elements of the same type, which are allocated consecutively in memory. In C language, a given array element can be accessed by simply using its name followed by its position enclosed in square brackets. For example, if we declare a string of characters as a global variable:

```
char StrChar[] = ”Hello World\n”;
```

the compiler allocates 12 consecutive bytes in *.data* section (Figure 4.4). Note that in C all strings of characters end with a 0 value. Note also that the array name (*StrChar*) is associated with the address of the first component (in this example 0x0c0002B8).

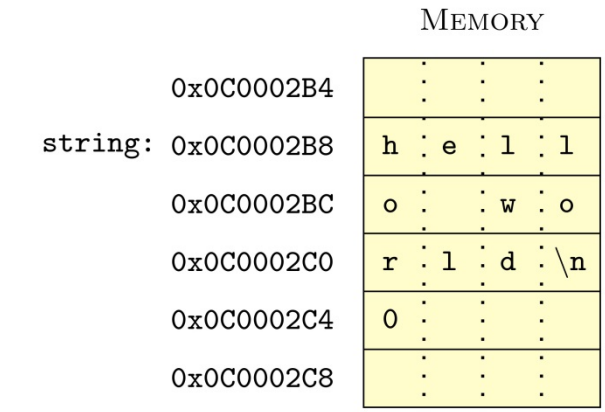

Figure 4.4. Allocation of an array of characters in memory.

It is important to highlight that the starting address of the array must satisfy the alignment constraints imposed by the architecture to the data type stored in the array.

4.7.2 Structures

A structure is a collection of variables, possibly of different types, with a given name. Next, we can see an example:

```
struct mystruct{
  char first;
  short int second;
  int third;
};

struct mystruct rec;
```

This code defines a structure named *mystruct* and declares a variable of this type named *rec*. The structure contains three fields, named *first*, *second* and *third*, each of a different type.

Like with the array, the compiler must allocate each structure field in a position that fulfils the alignment constraints imposed by the architecture. Frequently, the compiler will have to leave some holes in memory between the structure fields in order to accomplish these restrictions. For example, the structure defined above would result in the following mapping:

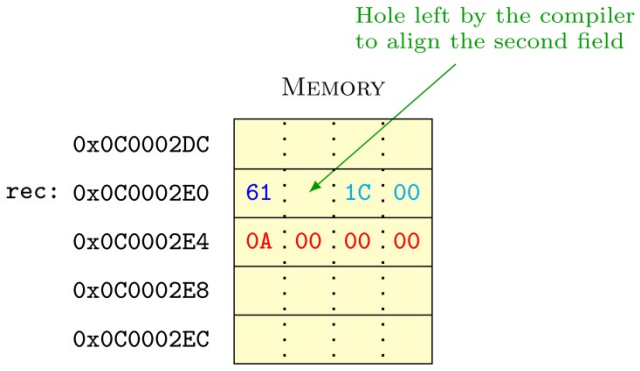

Figure 4.5. Mapping the *rec* structure to memory. The fields *first*, *second* and *third* have the values 0x61, 0x1C and 0x0A, respectively.

4.7.3 Unions

A union, like a structure, comprises several fields, possibly of different types, but unlike the structure, uses a shared memory space for allocating them all. Therefore, the necessary amount of memory for a union corresponds to the size of its longest field. Furthermore, the union must be placed in a position that fulfils the alignment constraints of all its fields. Next we can see an example:

```
union myunion{
  char first;
  short int second;
  int third;
};

union myunion un;
```

In this example, the programmer declares a union similar to the structure from the previous section. However, their *memory footprints* are very different (compare Figures 4.5 and 4.6).

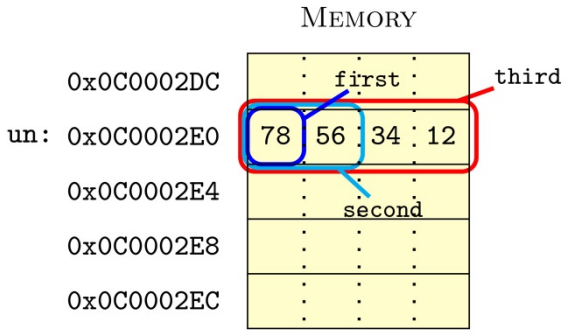

Figure 4.6. Storage of union declared above as *un*.

Depending on the field that we access, different sizes are employed. However, the same address (0x0C0002E0, in the example) is always accessed. For example, when we read the *first field*, a 1-byte access is performed, obtaining a value of *0x78*. Moreover, reading the *second field* returns the half-word *0x5678* (assuming *little endian*). Finally, when the *third field* is read, a word is accessed, obtaining the word *0x12345678*.

4.8. How to debug a C program in *Eclipse*

At this point, it is advisable that the student learn some of the different debugging options that *Eclipse* provides. For example, Figure 4.7 illustrates a debug session of a C program. The top right part of the figure displays the program variables (we can open that window by clicking "Window → ShowView → Variables"). Note that, automatically, the local variables of the *activation record* of the subroutine in execution are displayed. We can add a global variable by just clicking the mouse right button and selecting the option "Add Global Variables …".

When debugging, we can also be interested in watching the variables in memory, specially the arrays, since this way their components can be examined easily. For this purpose, we have

to open the *Memory Monitor* (bottom part of Figure 4.7. In the example, we are monitoring address *0x0C01C000*).

Finally, in some situations, it will be convenient to employ the utility *Expressions*, which lets us evaluating a C expression on the fly. For example, if we type *&num*, being *num* a variable that belongs to our program, the result will be the memory address where *num* is allocated.

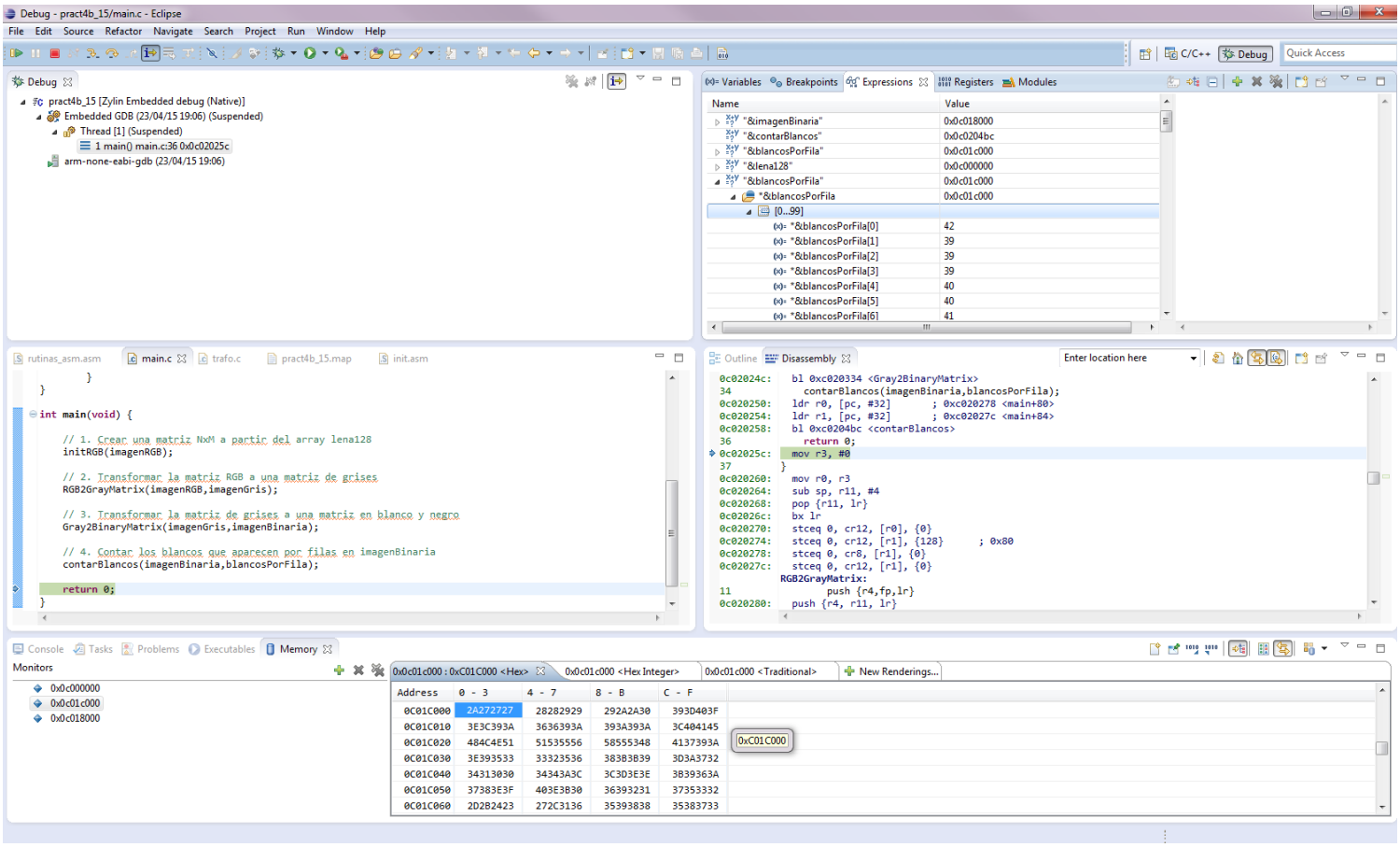

Figure 4.7 Example: debugging a C program.

4.9. Development of the lab session

In this lab, the point of departure will be a pre-defined project that allows the execution of a C program in the Eclipse environment. This program processes an image stored in the memory with three color (RGB) channels, and generates firstly a grayscale version of the image, and secondly a pure black and white version.

(a) RGB image (b) Grayscale image (c) B&W image

Figure 4.8. Transformation of an RGB image into gray scale and B&W

An image can be represented as a matrix of *pixels*, where each element of the matrix expresses the value of a pixel in some given scale. In RGB, each pixel is characterized by three values: the luminous intensity of the red (R), green (G) and blue (B) components, respectively. Therefore, each pixel of a color image will be a three-component vector. In the lab, we will use the following definition for the *RGB pixel* type:

```
typedef struct _pixel_RGB_t {
      unsigned char R;
      unsigned char G;
      unsigned char B;
} pixelRGB;
```

As shown in the type definition, each color channel is represented by 8 bits (`unsigned char`), so that we can distinguish among 256 different intensity levels in each channel, for a total of 24 bits per pixel (24bpp), which is quite common in current digital image formats.

On the other hand, to represent grayscale images we just need a single value (single channel) to indicate the brightness of each pixel. In this case, we use 8 bits to represent each pixel.

Consequently, to define two images of N rows and M columns, one in color and the other in grayscale, we could just use the following statements:

```
#define N 128
#define M 128
pixelRGB RGBImage[N][M];
unsigned char grayImage [N][M];
```

When accessing these matrices in assembly, we must know how to calculate the address of an element (*i*,*j*), given the starting address of the array. Knowing that, according to the ANSI C standard, two-dimensional arrays are stored in memory by rows, the address of the pixel in row *i* and column *j* is obtained by adding the offset (*i***M* + *j*)**B* to the address of the first element, where *B* is the number of bytes occupied by each pixel (3 in RGB and 1 in grayscale).

4.9.1 Transforming RGB into grayscale

The transformation between both color spaces is performed through a simple linear function. As shown in Figure 4.9, for each pixel we simply multiply the value of each color channel by a constant (the weight of the color), and the sum of the three products will give the grayscale value:

```
gray = 0.2126*pixel.R + 0.7152*pixel.G + 0.0722*pixel.B;
```

Given that the sum of the constants is the unity, the resulting gray value will be always within the range 0-255, i.e. it can be represented with one byte. However, to use these constants we need to operate with real numbers, and the ARM processor model with which we have been working does not have a floating-point representation, so we cannot operate with real numbers. To solve this, we will use an approach based on integer arithmetic, in which the constants are integer numbers and its sum is a power of 2 (in this case the sum will be 2^{14}). Of course, to reduce the final gray value to the range 0-255 we must divide de sum by 2^{14}, which is easily done with a 14-bit right shift. So, the final transformation is given by:

```
gray = (3483*orig.R + 11718*orig.G + 1183*orig.B) / 16384;
```

There is still another difficulty with this code because we have decided to represent the values of each RGB channel, as well as luminance of gray pixels with only 8 bits. While the final result of the above operation will certainly be in the range 0-255, we should observe that the results of intermediate operations will not. Therefore, before starting the operations we will extend each 8-bit channel to a whole word size (32 bits). In this way, we could safely do the multiplications, addition and division. The final result will always be less than or equal to 255, so we can store it as a single byte.

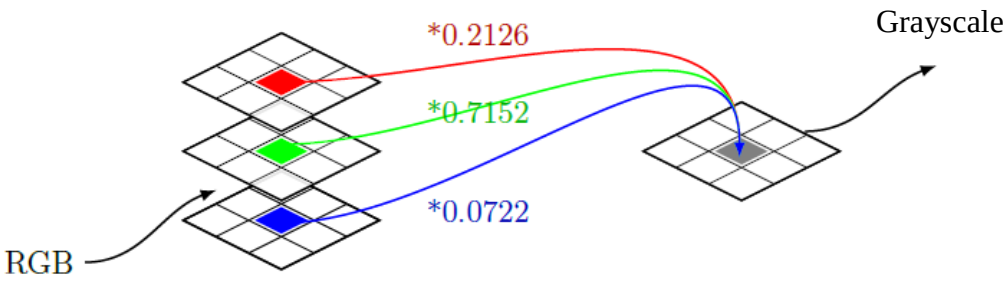

Figure 4.9: Transforming from RGB to grayscale

4.9.2 Transforming into a binary image

There are multiple algorithms to *binarize* a grayscale image, but they all agree on one point: they compare each pixel with a threshold, and if the pixel value is above the threshold, then that pixel is marked as white; otherwise, it will be black. The difference between the various methods lies in the mechanism for selecting the threshold, which may be even variable depending on the area of the image that we are processing. In this lab session, we use a fixed threshold value of 127.

The matrix that represents the binary image needs only one bit to store each pixel, but since most standard formats employ a byte for each pixel, in this lab we use an unsigned char (i.e. 1 byte) to represent each pixel, with the value 0 for black and 255 for white. Therefore, the following code can serve as a basis for the binary transformation with a given threshold:

```
for (i=0;i<N;i++)
     for (j=0;j<M;j++)
          if (grayImage[i][j] > threshold)
               binaryImage[i][j]= 255;
          else
               binaryImage[i][j]= 0;
```

4.9.3 Program structure

The code consists of several source files: `main.c`, `trafo.c`, `trafo.h`, `types.h`, `lena128.h`, `lena128.c`, `init.asm` and `routines_asm.asm`. The `.h` files contain forward declarations of functions or variables, as well as type declarations and constants. The `.c` files contain implementations of functions and variable declarations. The `init.asm` file contains the initialization code that invokes `main`, and the `routines_asm.asm` file contains an assembly implementation of the function `rgb2gray`.

The code of the main function will be as follows:

```
int main(void) {
     // 1. Create an NxM matrix from the array lena128
     initRGB(RGBImage);

     // 2. Transform the RGB matrix into a grayscale matrix
     RGB2GrayMatrix(RGBImage,grayImage);

     // 3. Transform the grayscale matrix into B&W
     gray2BinaryMatrix(grayImage,binaryImage);

     // 4. Count the white pixels in each row of binaryImage
     countWhites(binaryImage,whitesPerRow);

     return 0;
}
```

All these functions have been declared in the file trafo.h and implemented in the file trafo.c.

4.9.4 Results visualization

After having made the transformations, but before the end of the execution (i.e. setting a breakpoint at the `return` statement of the `main` function), we can dump the contents of some memory regions to disk files. To do this, we will use the *dump* command of the *GDB* debugger:

- In the Debug perspective, visualize the *Console* window.
- Write the following command in the console:

  `dump value` *outputFileName imageMatrix*

  For example, to create a file with the information contained in the matrix `grayImage` in the file `C:\hlocal\gray.dat`, we should write:

  ```
  dump value C:\hlocal\gray.dat grayImage
  ```

- This file does not use a widely accepted format, but it can be easily converted. To do this, you can use the `dump2ppm` program provided by the teacher. This program is a Windows executable file that reads the dumped file and creates a new one in PPM format. The program must be launched from the Command Prompt window. If you run the program without any parameters, it will provide help on how to use it. The program accepts the following parameters:
  - Name of the file containing the memory dump
  - Output filename
  - Number of image rows
  - Number of image columns
  - Number of channels (1 or 3).

  For example, we can obtain the PPM grayscale image from the corresponding dump file with the following command:

  ```
  dump2ppm C:\hlocal\gray.dat C:\hlocal\gray.ppm 128 128 1
  ```

- If we have some tool to visualize images in that format, the picture can be displayed by double clicking on it. The teacher will provide such tool.

4.9.5 Tasks to be completed

1. Compile the project and check for proper operation. The student must examine the resulting assembly code and verify that he/she roughly understands the functionality of that code. To check that the program works properly we can dump and display the images, as explained in section 4.9.4.
2. Obtain a C function equivalent to the subroutine *rgb2gray*.
3. Translate the function *RGB2GrayMatrix* into an assembly equivalent.