

Lab 1: Introducing the working environment

1.1 Goals

The main component of the equipment that we will use in the laboratory is an ARM microcontroller. In this first lab session we will try to get familiar with the architecture of this processor and its assembly language. Specifically, our goals in this session are:

- Gaining experience in the use of the ARM instruction set, including jumps and memory accesses.
- Getting familiar with the Eclipse development environment and GNU tools for ARM: assembler, linker and debugger, essentially.
- Understanding the structure of an assembly program, as well as the process of generating a binary code from it.

1.2 Introduction to the operation of a computer

As an electronic machine, a computer only understands electrical signals, which in digital electronics correspond to *off* and *on*. Therefore, the alphabet that can be understood by a computer corresponds to the two binary digits: 0 and 1 (binary alphabet).

Any command that we might want to send to the computer, has to be encoded as set of 0s and 1s, with a meaning that is known in advance. The computer must then decode the command to execute the specified function. Each of these individual orders is called an **instruction**.

Programming a computer using just 0s and 1s (**machine language**) is a very laborious and unrewarding task. To avoid this, it has been defined a symbolic language (called **assembly language**) made of simple commands that can be directly translated into the machine language, which the computer can understand. Thus, in assembly language the instructions are represented by means of mnemonic codes instead of bits. The program that performs the translation into machine language is called **assembler**.

If we can write programs that translate assembly language into machine language, would it be possible to write programs that are able to translate a high-level notation into assembly language? Yes, of course. In fact, nowadays most programmers write their programs in a much more user-friendly language that is called **high-level language** (e.g. C, Pascal, FORTRAN ...). The high level language is easier to learn and independent of the hardware architecture on which the program will finally run. Therefore, the implementation of algorithms in high-level languages is much faster than programming in assembly language, so increasing the productivity of programmers. The programs that translate high-level language into assembly language are called **compilers**. This sequence of translation steps is summarized in Figure 1.1.

| | | |
|--------------------|------------|--------------------------------------|
| C = A + B; | | High-level language |
| COMPILER | | |
| ldr | R1, A | Assembly language |
| ldr | R2, B | |
| add | R3, R1, R2 | |
| str | R3, C | |
| ASSEMBLER + LINKER | | |
| e51f1014 | | Machine language (in hexadecimal) |
| e51f2014 | | |
| e0813002 | | |
| e50f3018 | | |

Figure 1.1 Steps to generate computer executable commands (machine instructions).

1.3 Introducing the ARM architecture

The processors of the ARM family have a 32-bit RISC architecture, which is ideal for embedded systems with high performance and low power consumption such as mobile phones, PDAs, portable game consoles, etc. They have the main features of any RISC:

- A register file.
- A load / store architecture, i.e., arithmetic instructions operate only on registers, not directly on memory locations.
- Simple addressing modes. Memory locations accessed by load / store instructions, are determined only by the contents of a register and the value of a field of the instruction which is called “immediate value” (as we will see later, a second register can be alternatively used instead of the immediate value).
- Instructions with uniform format. All instructions have a width of 32-bits, with equally sized fields in similar instructions.

The ARM architecture follows a Von Neumann model, with a shared address space for instructions and data. This memory, which is byte addressable, is organized in 4-byte words.

In user mode, one can see 15 general purpose registers (R0-R14), the program counter (PC), which is also referred to as R15 for this architecture, and the status register (CPSR). All instructions can read and write each of the sixteen registers R0-R15 at any time. The status register, however, must be handled by means of special instructions.

The status register, CPSR, stores additional information necessary to determine the status of the program, such as the sign of the result of a previous operation, or the execution mode of the processor. It is the only register that has access restrictions. It is structured into fields, each with a well-defined meaning: flags, extension (reserved) and control, as illustrated in Figure 1.2. The flags field contains status indicators and the control field contains several bits that are used to control the execution mode. Table 1.1 describes the meaning of each bit of these fields. As we can see, some bits are reserved for future use and, therefore, cannot be changed (always read as zero). Bits N, Z, C and V (condition indicators) are modifiable in user mode, while the other bits can only be modified in privileged modes.

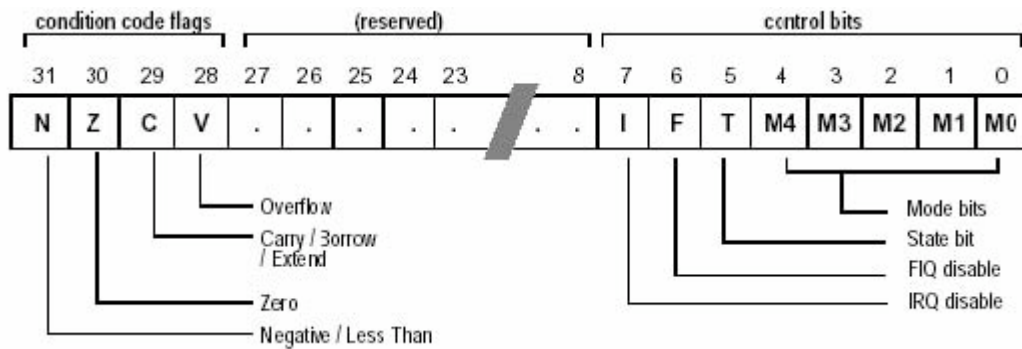


Figure 1.2. The Status Register (CPSR) of ARM

Table 1.1. Meaning of the condition code flags

| Bit | Significado |
|-----|---|
| N | Indicates whether the last executed operation has resulted in a negative (N = 1) or positive value (N=0). |
| Z | Indicates whether the result of the last executed operation has been zero. (Z = 1) |
| C | Its value depends on the operation type: <ul style="list-style-type: none"> - For additions and comparisons, C = 1 if a <i>carry</i> was detected - For shifts, it takes the value of the discarded bit |
| V | After executing an addition or subtraction, V = 1 indicates an <i>overflow</i> condition |

1.4 Instruction set

The instruction set can be divided into four groups:

- Arithmetic-logical
- Multiplication
- Memory access
- Branch

1.4.1 Arithmetic-logical instructions

In this section we study the instructions that use the arithmetic-logical unit (ALU) of the processor. We do not include here multiplication instructions, because they are executed in a different hardware module. In general, besides writing the result in the destination register, any instruction can also modify CPSR flags, if the suffix “S” is added to the mnemonic code.

The syntax of arithmetic-logical instructions is as follows:

Instruction{S} Rd, Rn, N @ Rd <- Rn Operation N

Where:

- Instruction: any of the mnemonics in Table 1.2
- S: If this field is included, the instruction will update the CPSR condition flags.
- Rd: destination register (where the result is stored).
- Rn: source register (first operand). For all instructions, except MOV.



- N: second operand, usually called *shifter_operand*. Although this operand offers a lot of alternatives, we will assume, for the moment, that it may only be a register (Ri) or a constant value (#N).

Some examples of arithmetic-logical instructions can be seen in Table 1.3.

Table 1.2: Some common arithmetic-logical instructions. ShiftOp represents the *shifter_operand*

| Mnemo | Operation | Action |
|-------|---------------------------------|--|
| AND | Logical AND | $Rd \leftarrow Rn \text{ AND } ShiftOp$ |
| ORR | Logical OR | $Rd \leftarrow Rn \text{ OR } ShiftOp$ |
| EOR | Logical XOR | $Rd \leftarrow Rn \text{ EOR } ShiftOp$ |
| ADD | Addition | $Rd \leftarrow Rn + ShiftOp$ |
| SUB | Subtraction | $Rd \leftarrow Rn - ShiftOp$ |
| RSB | Reversed subtraction | $Rd \leftarrow ShiftOp - Rn$ |
| ADC | Addition with carry | $Rd \leftarrow Rn + ShiftOp + \text{Carry Flag}$ |
| SBC | Subtraction with carry | $Rd \leftarrow Rn - ShiftOp - \text{NOT}(\text{Carry Flag})$ |
| RSC | Reversed subtraction with carry | $Rd \leftarrow ShiftOp - Rn - \text{NOT}(\text{Carry Flag})$ |
| CMP | Comparison | Compute $Rn - ShiftOp$ and update the CPSR flags according to the result |
| CMN | Complemented comparison | Compute $Rn + ShiftOp$ and update the CPSR flags according to the result |
| MOV | Register move | $Rd \leftarrow ShiftOp$ |
| MVN | Complemented move | $Rd \leftarrow \text{NOT}(ShiftOp)$ |
| BIC | Bit clear | $Rd \leftarrow Rn \text{ AND } \text{NOT}(ShiftOp)$ |

Table 1.3. Some examples of arithmetic-logical instructions

| | | |
|------|------------|--|
| ADD | R0, R1, #1 | $R0 \leftarrow R1 + 1$ |
| ADD | R0, R1, R2 | $R0 \leftarrow R1 + R2$ |
| MOV | R10, #5 | $R10 \leftarrow 5$ |
| BIC | R4, #0x05 | Clear bits 0 and 2 of R4 |
| SUB | R0, R2, R3 | $R0 \leftarrow R2 - R3$ |
| SUBS | R0, R2, R3 | $R0 \leftarrow R2 - R3$ and update N, Z, C and V accordingly |
| CMP | R4, R5 | Execute $R4 - R5$ and update N, Z, C and V accordingly |

1.4.2 Multiplication instructions

There are several versions of the multiplication instruction, because the multiplication of two n-bit operands will produce, in the general case, a 2n-bit result. Furthermore, there are signed and unsigned multiplications. The main variations are described in Table 1.4. All instructions can optionally modify Z and N bits of the status register (to detect whether the result has been, respectively, zero or negative) if we add the suffix “S”. The operands are always in registers and the result is also stored in one or two registers, depending on the selected size (32 or 64 bits).



Table 1.4. Multiplication instructions

| Mnemonic | Operation |
|---------------------------------|--|
| MUL Rd, Rm, Rs | Multiplication, 32-bit result: $Rd \leftarrow (Rm * Rs) [31..0]$ |
| MLA Rd, Rm, Rs, Rn | Multiplication plus addition, 32 bits: $Rd \leftarrow Rm * Rs + Rn [31..0]$ |
| SMULL RdLo, RdHi, Rm, Rs | Signed multiplication (2's complement), 64-bit result: $RdHi \leftarrow Rm * Rs [63..32]$; $RdLo \leftarrow Rm * Rs [31..0]$ |
| UMULL RdLo, RdHi, Rm, Rs | Unsigned multiplication, 64 bits: $RdHi \leftarrow Rm * Rs [63..32]$; $RdLo \leftarrow Rm * Rs [31..0]$ |

Table 1.5. Some examples of multiplication instructions

| | |
|-----------------------------|--|
| MUL R0, R1, R2 | $R0 \leftarrow R1 \times R2$ |
| MLA R0, R1, R2, R3 | $R0 \leftarrow (R1 \times R2) + R3$ |
| MULS R5, R8, R9 | $R5 \leftarrow R8 \times R9$ and update N, Z, C and V accordingly |
| UMULL R0, R1, R2, R3 | $R0 \leftarrow R2 \times R3 [31..0]$ and $R1 \leftarrow R2 \times R3 [63..32]$ |

1.4.3 Memory access instructions (load and store)

Load instructions (LDR) are used to load a data from memory to a register (Rd: destination register). Store instructions (STR) perform the opposite operation, the contents of a register (Rs: source register) is copied to memory

To compose the memory address we want to access, we can use a register (the so called base register) and an offset. The latter may be an immediate operand, or the contents of another register (which can be optionally shifted). There are several ways to combine the base register and the offset, but in this first lab session we will only explain the most common one:

Indirect register with immediate offset: The memory address to which we want to access (which is called the **effective address**) is obtained by adding an immediate offset to the value contained in the base register (Rb). If the offset is 0, then it can be omitted.

LDR Rd, [Rb, #offset] @ $Rd \leftarrow \text{Memory}(Rb + \text{offset})$
STR Rs, [Rb, #offset] @ $\text{Memory}(Rb + \text{offset}) \leftarrow Rs$

Table 1.6. Examples of load / store instructions

| | |
|-----------------------------|--|
| LDR R0, [R1] | $R0 \leftarrow \text{Mem}(R1)$ The memory address specified by R1 is read, and its contents is copied to R0 |
| LDR R0, [R1, #7] | $R0 \leftarrow \text{Mem}(R1 + 7)$ |
| STR R5, [R8, #-4] | $\text{Mem}(R8 - 4) \leftarrow R5$ Stores the contents of R5 to the memory address given by the result of (R8 - 4) |
| STR R5, [R8, #0x010] | $\text{Mem}(R8 + 16) \leftarrow R5$ (Any number expressed as 0xN..N is considered to be represented in hexadecimal) |

1.4.4 Branch instructions:

Branch instructions perform a backward or forward jump, which is relative to the current value of the PC. The jump distance is limited by the 24-bit offset field that is included within the instruction (immediate operand). The syntax is:

`B{Condition} Offset @ PC ← PC + Offset`

where *Condition* indicates one of the mnemonics in Table 1.7 (EQ, NE, GE, GT, LE, LT, ...) and *offset* is a signed immediate operand that represents an offset from the PC. In machine language this offset represents an amount of instructions, and thus the number of bytes to be added to the PC is obtained by multiplying the offset by 4. However, if we want to express an offset in assembly language, we must directly indicate the amount of bytes. Fortunately, we rarely need to express offsets in assembly code.

If the condition is true, the instruction that will be executed next is (Offset+2)¹ instructions ahead or behind the current instruction; otherwise, the execution will continue sequentially. For instance, assuming that the condition is true, if the offset is -6 then the next executed instruction is 4 instructions behind; while an offset of 6 means that the next executed instruction is 8 instructions ahead. To avoid the burden of calculating offsets manually, it is convenient to use **labels** to identify the destination of branches, so that the exact value of the offset will be computed by the assembler. The use of labels is exemplified in Tables 1.8 and 1.9.

Whether a condition is true or false is determined by evaluating the corresponding expression in the third column of Table 1.7. Therefore it is necessary that a previous instruction has activated the flags of the status register. This previous instruction is normally a CMP (see arithmetic-logical instructions), but it can also be any instruction with the S suffix.

Table 1.7. Correspondence between conditions and branch instructions

| Mnemonic | Description (Branch if...) | Implementation (Branch if ...) |
|----------|----------------------------|--------------------------------|
| EQ | Equal | Z=1 |
| NE | Not equal | Z=0 |
| HI | Higher (unsigned) | C=1 and Z=0 |
| LS | Lower or same (unsigned) | C=0 or Z=1 |
| HS | Higher or same (unsigned) | C=1 |
| LO | Lower (unsigned) | C=0 |
| GE | Greater or equal (signed) | N=V |
| LT | Less than (signed) | N≠V |
| GT | Greater than (signed) | Z=0 and N=V |
| LE | Less or equal (signed) | Z=1 or N≠V |
| (empty) | Always (unconditional) | Always (unconditional) |

¹ The reason for adding the constant 2 to the offset is that in this architecture, when an instruction is being executed, the value in the PC has already been incremented in 8 bytes (i.e. 2 instructions).



Table 1.8. Examples of branch instructions

| | | |
|-----|--------|---|
| B | loop | Branch always to the instruction with label “loop” |
| BEQ | loop | Branch to the instruction with label “loop” if flag Z=1 |
| BLS | loop2 | Branch to the instruction with label “loop2” if C=0 or Z=1 |
| BHI | part_B | Branch to the instruction with label “part_B” if C =1 and Z = 0 |

It should be observed how, in Table 1.8, we have always used labels instead of offsets. On the other hand, Table 1.9 provides additional examples on branches and labels, showing at the same time how some common high-level language constructs can be translated into assembly language.

Table 1.9. Translating high-level constructs into assembly language

| | |
|--|--|
| <pre>if (R1 > R2) R3 = R4 + R5; else R3 = R4 - R5 normal execution continues</pre> | <pre> CMP R1, R2 @CMP prepares the flags BLE else @We use the reverse condition ADD R3, R4, R5 B end_if else: SUB R3, R4, R5 end_if: normal execution continues</pre> |
| <pre>for (i=0;i<8;i++) { R3 = R1 + R2; } normal execution continues</pre> | <pre> MOV R0, #0 @R0 acts as the index i for: CMP R0, #8 BGE end_for @Reverse condition ADD R3, R1, R2 ADD R0, R0, #1 B for end_for: normal execution continues</pre> |
| <pre>i=0; do { R3 = R1 + R2; i=i+1; } while (i != 8); normal execution continues</pre> | <pre> do: MOV R0, #0 @R0 acts as the index i ADD R3, R1, R2 ADD R0, R0, #1 CMP R0, #8 BNE do @We use the direct condition at the @end of do-while loops normal execution continues</pre> |
| <pre>while (R1 < R2) { R2= R2-R3; } normal execution continues</pre> | <pre> while: CMP R1, R2 BGE end_w @Reverse condition SUB R2, R2, R3 B while end_w: normal execution continues</pre> |

1.5 Structure of an assembly language program

In order to explain the structure of an assembly program and the different assembly directives, we will use as a guide the simple program described in Table 1.10.

Table 1.10. A simple program in assembly language

```
.global start
.equ ONE, 0x01

.data
MYVAR: .word 0x02

.bss
RES:   .space 4

.text
start:
MOV    R0, #ONE
LDR    R1, =MYVAR
LDR    R2, [R1]
ADD    R3, R0, R2
LDR    R4, =RES
STR    R3, [R4]
END:   B      .
.end
```

The first thing we can see in the listing of Table 1.10 is that a program in assembly language is just a text, which is structured in lines with the following format:

```
label:  <instruction or directive>    @ comment
```

This text contains a section (i.e. a set of lines in sequence) devoted to define where we will store the data, either input or output. And another section devoted to describing the algorithm using assembly instructions. As the ARM is a Von Neumann machine, data and instructions use the same memory space. As in any piece of software (either written in assembly language or not), the input data are assigned to certain memory locations, and the output results are similarly written to memory locations reserved to that purpose. In our case, if we want to change the input values of the program, we will have to manually change the corresponding values in the file. Similarly, to verify that the program works correctly we will need to check the values stored in those memory locations reserved for the output, after having executed the program.

We use the following terms to describe each line:

- **Label** (optional): it is a string of text that the assembler will associate with the memory address corresponding to that point of the program. If at any other point of the program, this string appears in a place where an address is expected, the assembler will replace the label by the associated memory address, using the adequate addressing mode. For example, in the program of Table 1.10 the instruction `LDR R1, =MYVAR` will load register R1 with the memory address reserved for the variable MYVAR, so that from that moment the register R1 can act as though it were a pointer. Observe that the symbol “=” tells the assembler to generate machine code to load R1 with the address, and not the value, of MYVAR. The assembler is smart

enough to properly encode this instruction, automatically calculating the location of MYVAR. If we want to access the value of MYVAR, we can use the instruction LDR R2, [R1], provided that R1 contains the address of MYVAR.

- **Instruction:** the mnemonic code of an instruction of the target architecture (one of those described in Section 1.4, see Figure 1.4). Sometimes the operands may be modified by the use of labels in order to simplify program writing. This has been the case we have just described, where the memory address referenced by a load instruction is indicated as a label, and it is under the responsibility of the assembler to encode this address as a base register plus an offset.
- **Directive:** it is a command that the programmer gives to the assembler. Directives may instruct the assembler to initialize memory locations with a certain value, define symbols that make the program more readable, mark the starting and ending points of a program, etc. (See Figure 1.3). Directives are needed because, in addition to writing the algorithm using assembly instructions, the assembly language programmer has to reserve memory space for variables, and store their initial value (if defined) in the corresponding addresses. In Figures 1.3 and 1.4 we have used the following directives:

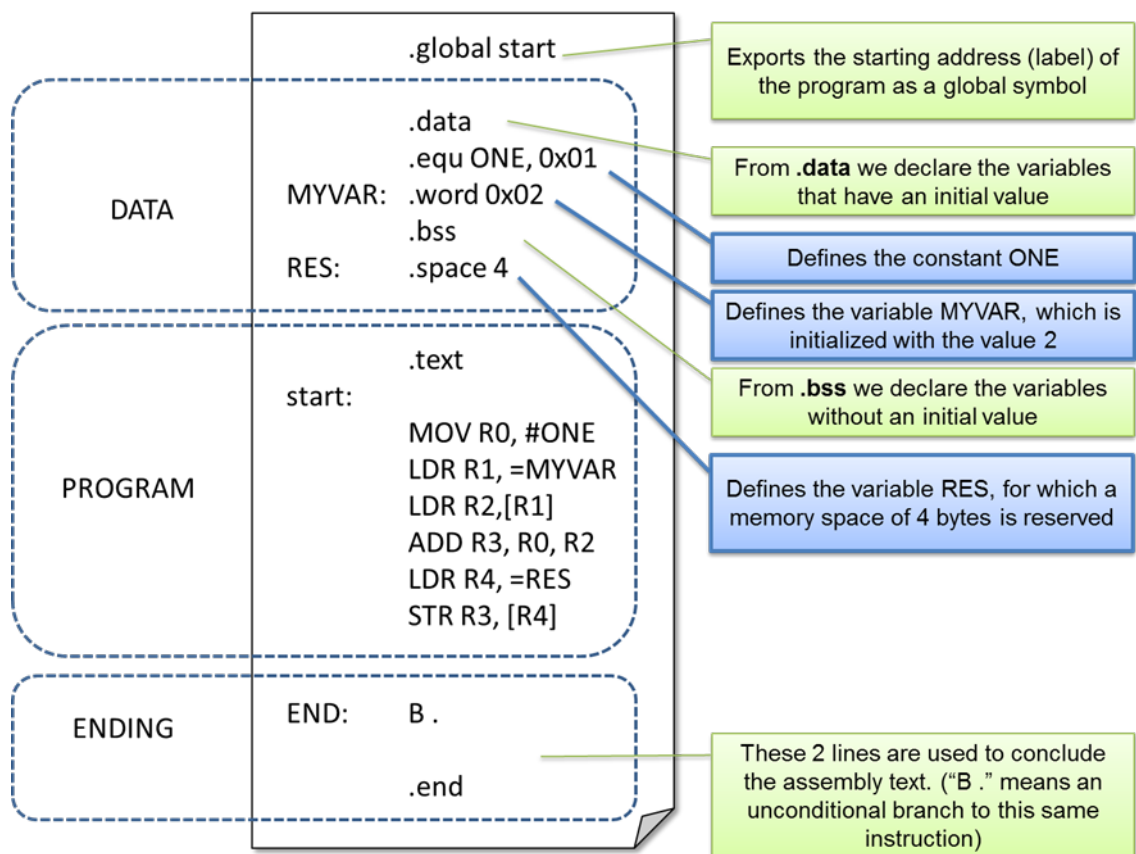


Figure 1.3 Structure of an assembly program: data

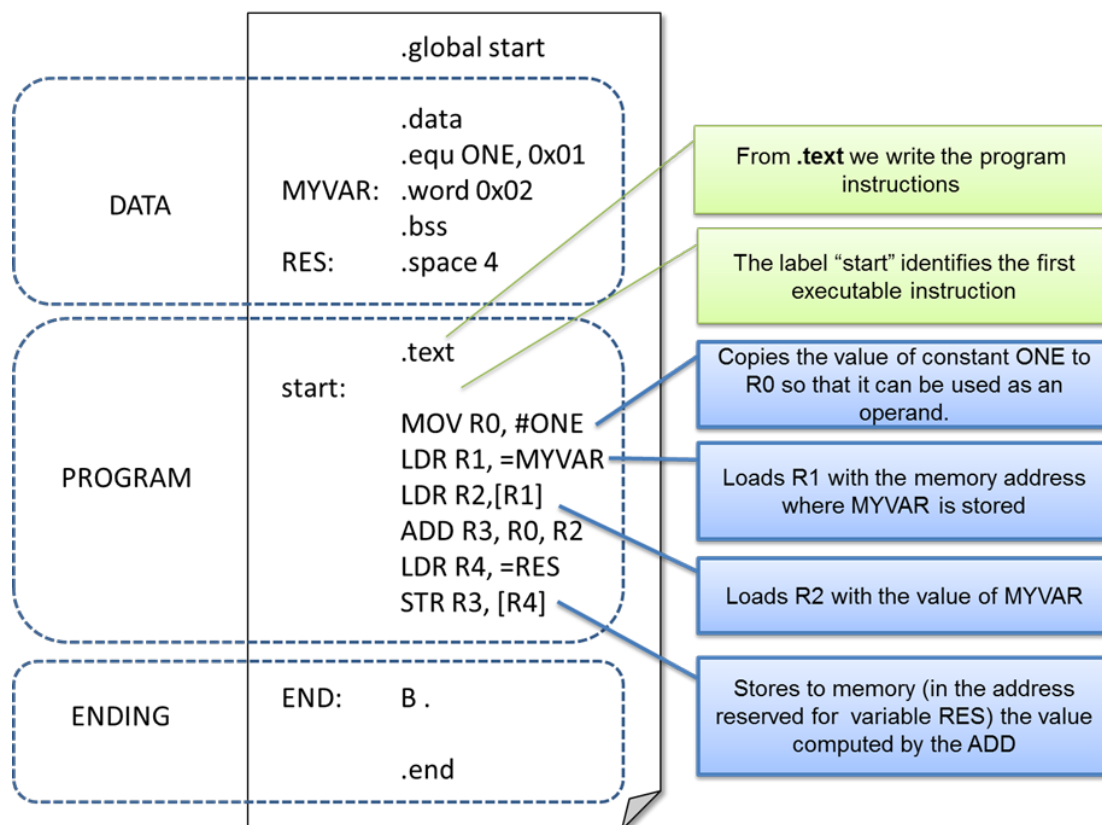


Figure 1.4 Structure of an assembly program: instructions

- **.global**: exports a symbol to be used by other files during the linking process. The beginning of the program is indicated by the directive **.global start**. The label **start** appears again just before the first instruction of the program, to indicate the location of the first executable instruction.
- **.equ**: provides an alternative way to represent constants. An **.equ** directive associates a symbolic name with a constant value, so that when such name appears as an operand in the program, the assembler will replace it by the corresponding value. As an example, the directive **.equ ONE, 0x01** defines the constant **ONE** with the value 1. A few lines below, the symbol **ONE** is used as an operand in the instruction **MOV R0, #ONE**. The assembler will replace this symbol by the value 1, so that the resulting executable code would have been exactly the same if we had used the instruction **MOV R0, #1** instead. Although in this simple example the possibility of using the symbol **ONE** instead of the number 1 may seem rather useless, there may be situations where the use of a symbol actually contributes to improve the readability of the program. Furthermore, it is worth noting that symbols defined with **.equ** directives are not assigned any memory space; they are only used by the assembler, which replaces them by constants.
- **.word**: commonly used to initialize the input variables of the program. It initializes the current memory location with the word-sized indicated value. We might have also used **.byte** instead of **.word**. The difference is that **.word** allocates 4 memory bytes (i.e. one word) to the variable,

whereas `.byte` allocates just one byte. For example: `MYVAR .word 0x02`, initializes the memory location represented by the label `MYVAR` with the value 2, where `0x` indicates hexadecimal representation.

- `.space`: reserves memory space to store variables without an initial value. It is commonly used to declare the output variables, when they are not also serving as input variables. The space we want to reserve must always be expressed as a number of bytes. For example, the directive `RES: .space 4` reserves four bytes (i.e. one word) that are not initialized. The `RES` label can be used in the rest of the program to represent the address of that word.
- `.end`: Finally, the assembler will conclude its work when it reaches the directive `.end`. Any text located after this directive will be ignored.
- **Sections**: usually the program is divided into sections (typically `.text`, `.data` and `.bss`). To define these sections we simply 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.
 - `.bss`: is the section that reserves space to store the results.
 - `.data`: is the section that is used to declare variables with an initial value.
 - `.text`: contains the program code.
- **comment**: (optional) a text string to comment the code. All the text included between the character `@` and the end of the current line is considered as a comment. We can also write multiline comments as we do in C (between `/*` and `*/`).

The steps followed by the compilation environment to translate the code presented in Table 1.10 are summarized in the following table.

Table 1.11. Translation from assembly code into machine code (binary)

| Assembly Code | Machine Code (Hex) | | Machine Code (Binary) | |
|-----------------------------|--------------------|---------------------------------|-----------------------|---|
| | Address | Contents | Address | Contents |
| <code>.global start</code> | | | | |
| <code>.data</code> | | | | |
| <code>.equ ONE, 0x01</code> | | | | |
| <code>MYVAR:</code> | | | | |
| <code>.word 0x02</code> | C000000 | 02 00 00 00 | C000000 | 0000 0010 0000 0000 0000 0000 0000 0000 |
| <code>.bss</code> | | | | |
| <code>RES:</code> | | | | |
| <code>.space 4</code> | C000004 | (no initial value) | C000004 | (no initial value) |
| <code>.text</code> | | | | |
| <code>start:</code> | | | | |
| <code>MOV R0, #ONE</code> | C000008 | <code>mov r0, #1</code> | C000008 | 0000 0001 0000 0000 1010 0000 1110 0011 |
| <code>LDR R1, =MYVAR</code> | C00000C | <code>ldr r1, [pc, #16]</code> | C00000C | 0001 0000 0001 0000 1001 1111 1110 0101 |
| <code>LDR R2, [R1]</code> | C000010 | <code>ldr r2, [r1]</code> | C000010 | 0000 0000 0010 0000 1001 0001 1110 0101 |
| <code>ADD R3, R0, R2</code> | C000014 | <code>add r3, r0, r2</code> | C000014 | 0000 0010 0011 0000 1000 0000 1110 0000 |
| <code>LDR R4, =RES</code> | C000018 | <code>ldr r4, [pc, #8]</code> | C000018 | 0000 1000 0100 0000 1001 1111 1110 0101 |
| <code>STR R3, [R4]</code> | C00001C | <code>str r3, [r4]</code> | C00001C | 0000 0000 0011 0000 1000 0100 1110 0101 |
| <code>END:</code> | C000020 | <code>b .</code> | C000020 | 1111 1110 1111 1111 1111 1111 1110 1010 |
| <code>B .</code> | | | | |
| <code>.end</code> | C000024 | (Reserved for address of MYVAR) | C000024 | 0000 0000 0000 0000 0000 0000 0000 1100 |
| | C000028 | (Reserved for address of RES) | C000028 | 0000 0100 0000 0000 0000 0000 0000 1100 |



The code in the first column of Table 1.11 corresponds to the code in assembly language with labels, as shown in Fig. 1.3. This code allows a quick writing of the algorithm without having to perform offset computations manually in order to specify the location of variables in memory.

If we made the assumption that the beginning of section `.data` is located at address `0x0C000000`, and `.bss` and `.text` are located just in sequence, the central column would show the assembly code without labels, as well as the address (in hexadecimal) where each data and instruction is located. There are some details of this code, which is called “disassembly code”, that deserve further explanation.

For instance, the variable `MYVAR` has been assigned to address `0C000000`. We should recall, however, that this variable represents a 32-bit value, and thus it is actually stored in 4 consecutive bytes: from address `0C000000` to `0C000003`. Given that the memory organization is “little endian” the least significant byte of `MYVAR` (02) is stored in `0C000000`, as we can see in the table.

On the other hand, label operands have been replaced by an addressing mode that we had called before “indirect register with immediate offset”, where the register, in this particular case, is the PC. So we are identifying operand addresses in a PC-relative mode (i.e. using the distance from the current value of the PC). For example, the instruction “`ldr r1, [pc, #16]`” will load `r1` with the value that is stored in the memory address given by $(\text{Instruction address} + 8) + 16 = 0C00000C + 8 + 16 = 0C000024$. Recall that when the ARM is executing the instruction “`ldr r1, [pc, #16]`”, which is stored in address `0C00000C`, the PC is pointing 8 bytes ahead, and we add an offset of 16 bytes to the contents of the PC. We can observe in the table that `0C000024` is the word address that follows the last instruction of the program. This is so because during the linking process, it will be the responsibility of the linker to record the address assigned to `MYVAR` and store it in the location `0C000024`. Similarly, the linker will store the address of `RES` in the location `0C000028`.

Finally, the linked code can be directly translated into machine code (i.e. into zeroes and ones) as shown in the third column of Table 1.11, where the last bits of each line correspond to the operation code of each instruction. For instance, one can see that the last bits of the three “`ldr`” instructions are identical. Needless to say that trying to manually write programs in machine language is a tedious and error prone activity; and therefore the assembly language appears as an efficient way to deal with much of the trouble involved in low-level programming.

1.6 First steps with the development environment

In this lab we will use Eclipse as an Integrated Development Environment (IDE). Eclipse is an application with a graphical user interface that will allow us to develop programs in both assembly and C languages, as well as to debug our software on a simulator of the ARM architecture. It would also be possible to debug on a real circuit board with an ARM processor.

Eclipse mainly offers a graphical interface and a project management system. When it has to perform other tasks it uses external GNU tools: the assembler (`as`), the compiler (`gcc`), the



linker (ld) and the debugger (gdb). Moreover, we should keep in mind that we are developing on a PC for an environment based on the ARM processor, and therefore we need to do cross-compilation.

In order to distinguish cross from native PC tools, it is quite common to add a prefix that describes the target architecture of the compilation process. In our case this prefix is: arm-none-eabi. These tools can also be used directly from a command line interpreter (terminal in Linux or Mac OS X, or cmd in Windows). In any case, we must always use the syntax and programming rules imposed by these tools.

1.6.1. Creating an Eclipse project

For each of the activities proposed in the lab script, we must create an Eclipse project for cross compilation and debugging using the GNU ARM plugin. To do this, we must proceed as follows:

1. Open Eclipse by double clicking on the desktop icon.
2. At the beginning, the application will display the window for workspace selection, as shown in Figure 1.5. We must select our own workspace, which will be the directory used by Eclipse to keep the information of all our projects. It is recommended to put our workspace in the folder C:\hlocal. It is also important to always keep a copy of our workspace in a safe place, because this folder is local to each lab seat, and thus shared by all the users of the seat.

REMARK: if you are working with the “predefined workspace”, you can proceed directly to step 9.

3. Once selected, the main Eclipse window will open. If you have just created the workspace, then the window will look like the one shown in Figure 1.6.
4. We must close the Welcome tab by clicking on the cross symbol. Then, the window will display as described in Figure 1.7: this is the C/C++ Eclipse perspective, which is organized as follows:
 - Left panel: the Project Explorer. We will see all the projects we have in the workspace (if we have any, of course).
 - Central panel: the editor. This allows us to edit the files that contain the source code of our programs.
 - Right panel. It allows us to explore the symbols of the current project (functions, variables, etc.).
 - Bottom panel. It has several tabs, among which the compile errors tab and the console tab are of special interest. From the first one, we can see the mistakes, and we can even jump to the source line that caused the error just by clicking on the error. In the second, we can see the commands executed by Eclipse during compilation.
5. To create the project, we must select File→ New→ C Project, which will open a window like that of Figure 1.8. As shown in the figure, we have to select the options ARM Cross Target Application, Empty Project and ARM GCC (Sourcery G++ Lite). Now we can chose the project name and click Finish. Our empty project will be then visible in the Project Explorer.



6. Now let's add a file with the source code of our first assembly program. To do this, select File→ New→ Source File, to open a window like the one in Figure 1.9. In order to tell Eclipse that the file contains assembly code, the file name must include the extension “.asm” or “.S”. Once created, we must double click on the file in the left panel. This will open a new editor tab, into which we must copy the code from Table 1.12.
7. Before configuring the compilation of our project, we will add to it a file that will be used to tell the linker how to build the memory map of the final executable file. In this case we are going to see another way to add a file to the project. In the Project Explorer, select the project, click the right mouse button and select New→ File. A window like the one in Figure 1.10 will then open. Select the project and put `ld_script.ld` as the filename. Once created, open it in the editor and copy the contents of Table 1.13.
8. Finally we must configure the project to perform a successful compilation. To do this, we select the project in the left panel, click the right mouse button, and select the Properties entry at the bottom of the pull-down menu. This will open a window like the one shown in Figure 1.11. In this window, we must select C/C++ Build→ Settings and:
 - Verify that in Target Processor ARM7TDMI is selected as the processor, none of the Thumb* boxes is marked, and the option Little Endian is selected.
 - We have to select ARM Sourcery GCC C Linker→General, and in the box Script file (-T) we must write the path to the file `ld_script.ld` that we added to the project. We can do it graphically by clicking the Browse button. When you are done, click Apply and OK.
9. We are now ready to compile the project. This is done by selecting Project→ Build Project. When this step is finished, we will have obtained the final executable file with the extension “.Elf” (*Executable Linked Format*), which will be located in the Debug subdirectory, within the project directory of our workspace.

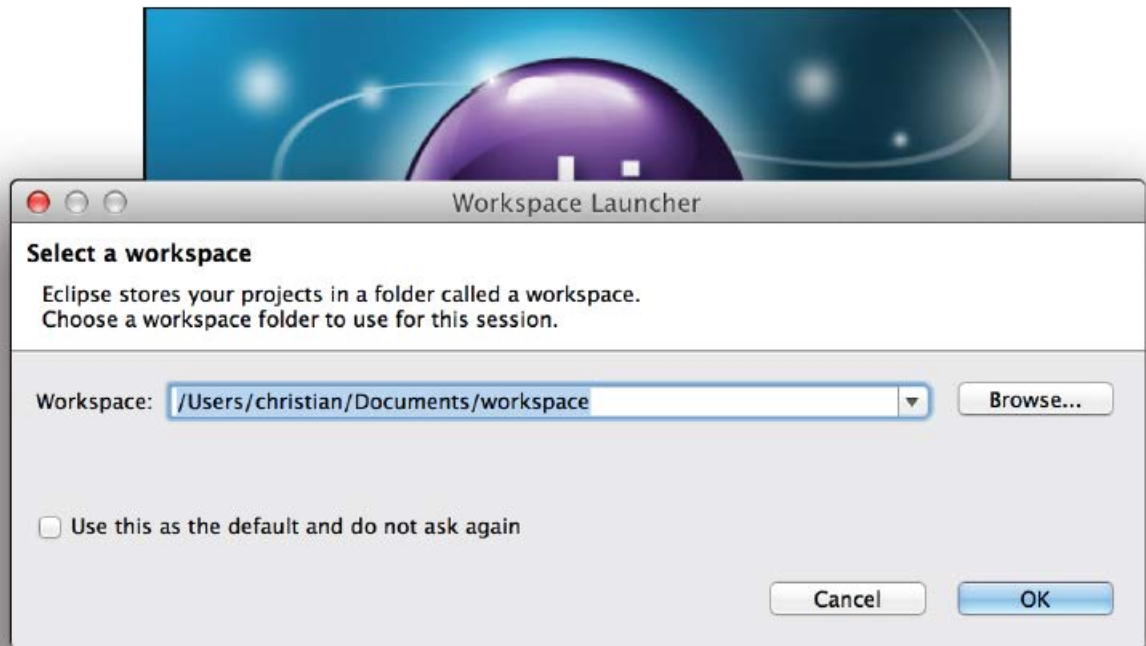


Figure 1.5 Workspace selection window.



Figure 1.6 The Eclipse window when an empty workspace is open

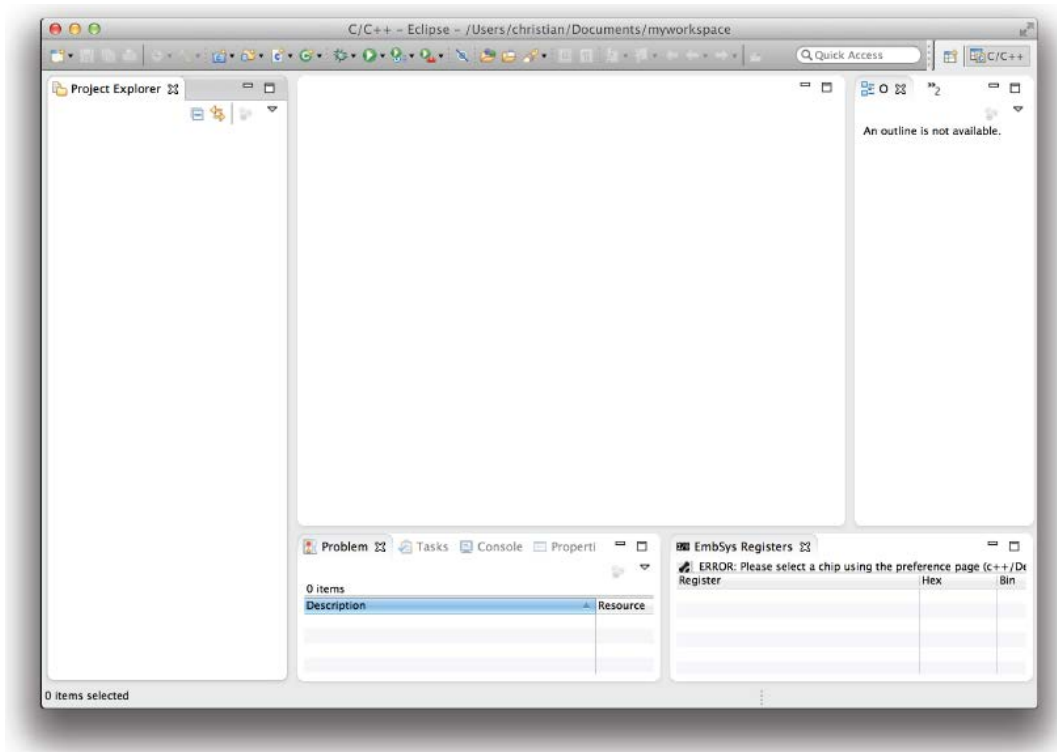


Figure 1.7 A blank Eclipse window with the C/C++ perspective

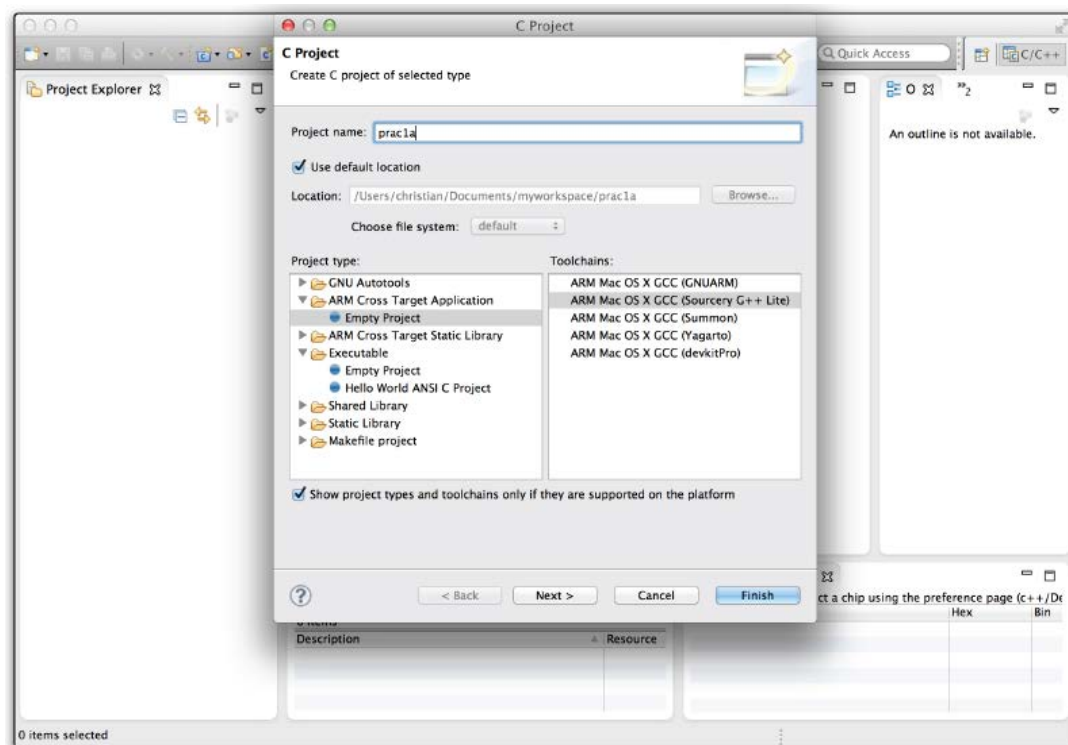


Figure 1.8 Creating a C project of GNU ARM type



Table 1.12 A program in assembly language that compares two numbers and selects the largest one

| | | |
|----------------------------|-------------------------------|---------------------------|
| <code>.global start</code> | | |
| <code>.data</code> | | |
| <code>X:</code> | <code>.word 0x03</code> | |
| <code>Y:</code> | <code>.word 0x0A</code> | |
| <code>.bss</code> | | |
| <code>Largest:</code> | <code>.space 4</code> | |
| <code>.text</code> | | |
| <code>start:</code> | <code>LDR R4, =X</code> | @R4 ← Address of X |
| | <code>LDR R3, =Y</code> | @R3 ← Address of Y |
| | <code>LDR R5, =Largest</code> | @R5 ← Address of Largest |
| | <code>LDR R1, [R4]</code> | @R1 ← Value of X |
| | <code>LDR R2, [R3]</code> | @R2 ← Value of Y |
| | <code>CMP R1, R2</code> | @Compute R1-R2 |
| | <code>BLE else</code> | @IF R1 <= R2, go to else |
| | <code>STR R1, [R5]</code> | @Largest ← Value of R1 |
| | <code>B Done</code> | @Go to Done |
| <code>else:</code> | <code>STR R2, [R5]</code> | @Largest ← Value of R2 |
| <code>Done:</code> | <code>B .</code> | @Work done, infinite loop |
| <code>.end</code> | | |

Table 1.13 Linker script

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

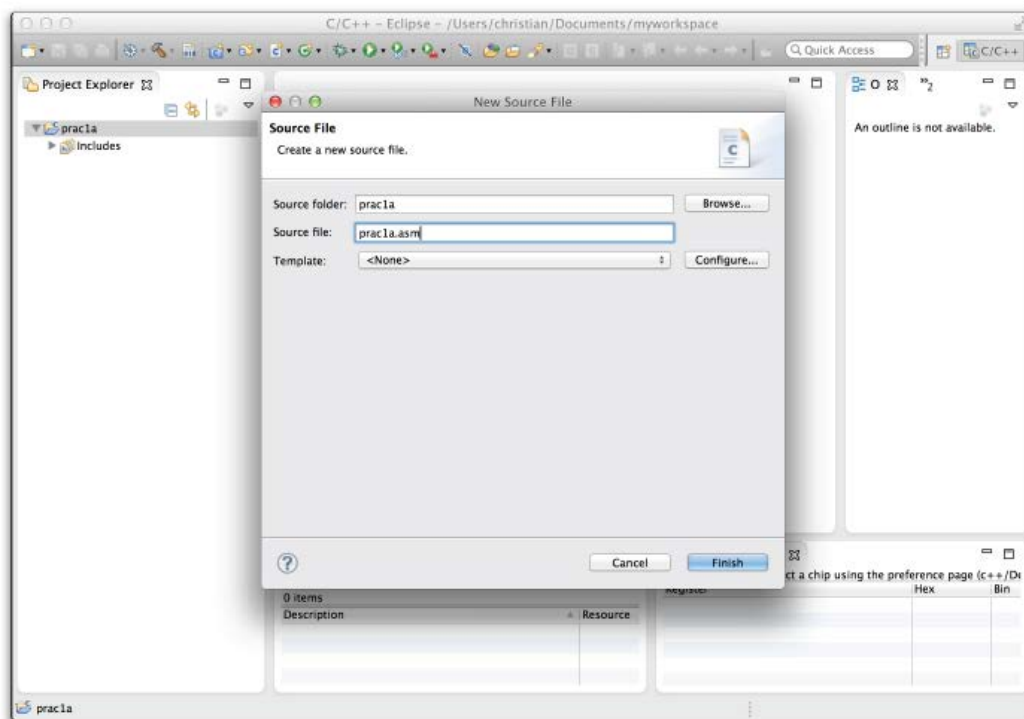


Figure 1.9 Creating a new source file

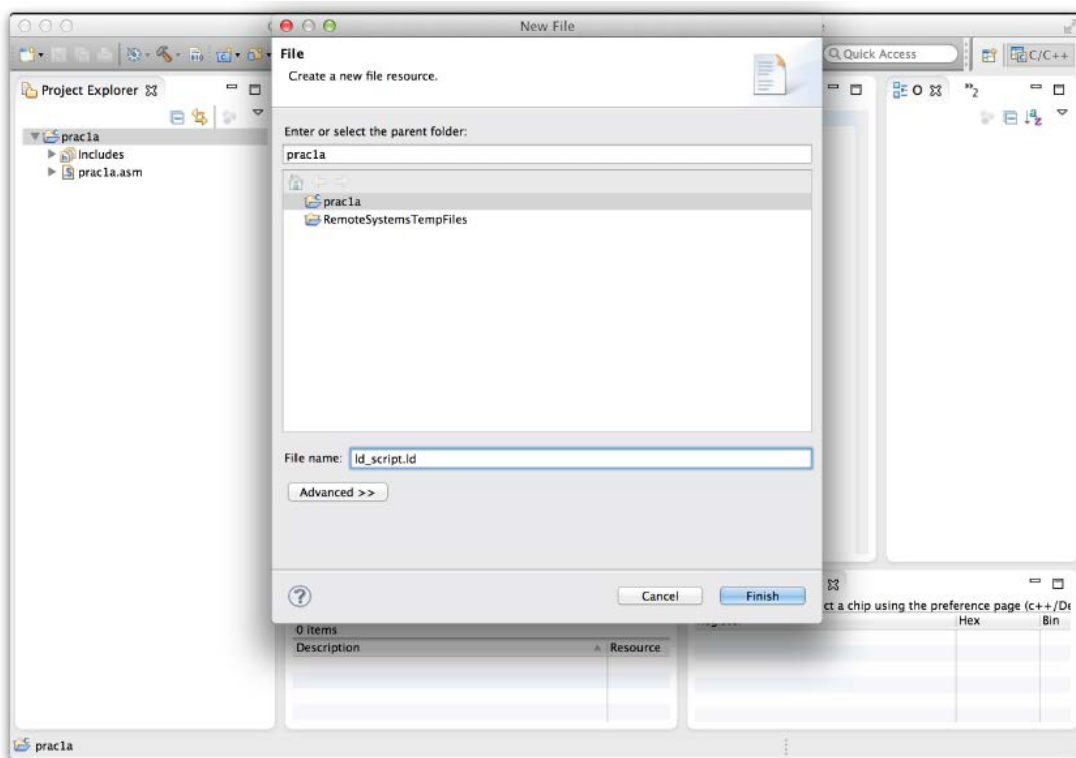


Figure 1.10 Adding a new file to the project

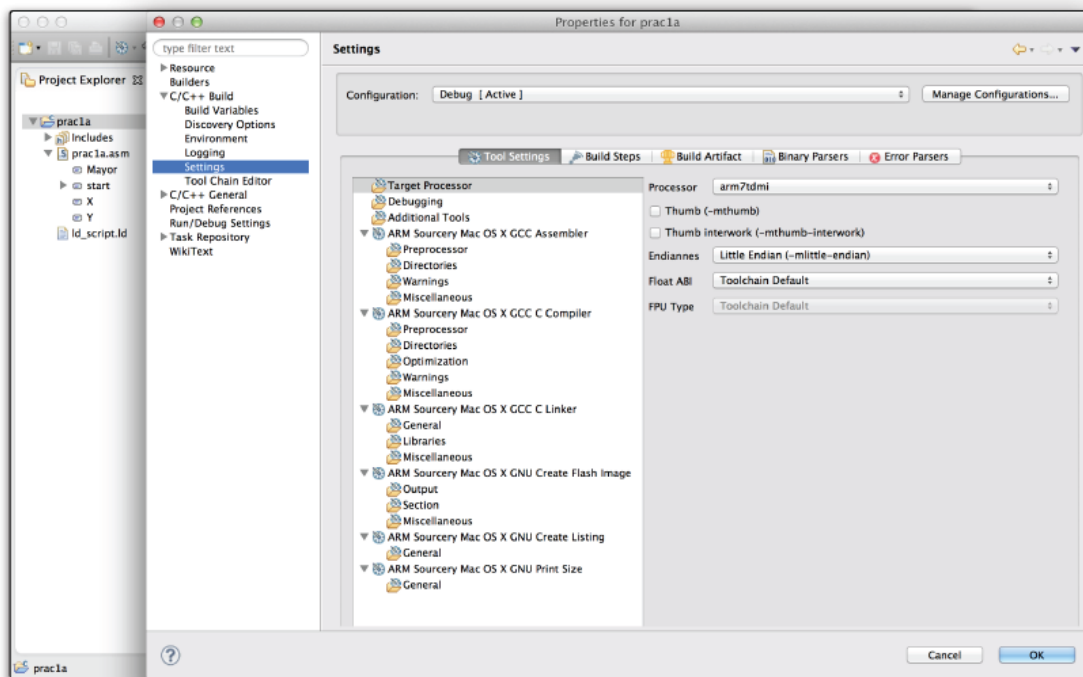


Figure 1.11 The properties window of the project

1.6.2. Debugging on the simulator.

After generating the executable file of our project, we have to verify that it works correctly. We will use the arm-none-eabi-gdb debugger, using the interface provided by Eclipse. The debugger will allow us to run the program step by step, set breakpoints, examine registers, memory, etc. Given that in the first lab sessions we will not use a real ARM processor, we must configure the debugger to use the simulator.

There are two Eclipse plugins that allow us to debug with GDB on ARM: GDB Hardware Debugging and Zylin Embedded debug. The first one is easier when debugging on the real circuit, while the second one is easier when debugging on the simulator. So, we will use Zylin for now.


To debug our project we must complete the following steps:

1. Open the Debug perspective. To do this, select Window→ Open Perspective→ Debug. The Eclipse window will then appear as shown in Figure 1.12. As we can see, the window is divided into several regions, each one with tabs:
 - Top Left: information about the debug process we have launched.
 - Top right: tabs where we can see registers, variables, breakpoints...
 - Central Left: the source code we are debugging. Initially, the only files that will appear are those we have open in the C/C++ perspective. New tabs will

automatically open if the running program jumps to a function that was defined in another file compiled with debug symbols.

- Central Right: Initially, it only shows a list of the symbols defined by the program. It will be helpful adding to this panel a new tab showing the disassembly code by selecting Window→Show View→Disassembly.
- Bottom: multiple tabs with several different purposes. For example, here we can put the memory viewer by selecting Window→Show View→Memory.

REMARK: if you are working with the “predefined workspace”, you can skip step 2. Just click **Run→Debug configurations**, select **Pract1a** (in the left part of the Debug configurations window) and proceed to step 3.

2. Create a debug configuration for the project using the Zylin plugin. This is done by first selecting Run→Debug configurations ... which will open the window shown in Figure 1.13. In the left panel, select Zylin Embedded debug (Native) and click the button in the upper left corner of the panel () to create the configuration. After that, we should have a window like that shown in Figure 1.14. Now we must correctly fill the following configuration tabs:

- Debugger: on this tab we choose the debugger to be used. In the laboratory, the path to the cross toolchain has already been added to the user's path, so just put the name of the debugger in the entry GDB debugger: arm-none-eabi-gdb. Additionally, at the top we can insert a temporary breakpoint if we want. We will place this breakpoint in the address of the start symbol, by writing *start in the box Stop on startup at:. The result is shown in Figure 1.15.
- Commands: on this tab we give the commands that gdb must execute in order to start the debugging. In the box Initialize commands, we write target sim. This indicates that the debugging target is the simulator. In the box Run commands, we write:

```
load
```


```
run
```






This tells the debugger to load the file selected on the main tab, and read the debug symbols from it. The resulting window is shown in Figure 1.16.






3. We are now ready to debug. To start with, click the Debug button. This will save the debug configuration with the name that was selected in the first tab, and will start a debugging session with this configuration. If you later want to debug again the same project, it will not be necessary to create a debug configuration; you can simply select the one you have just created. Figure 1.17 shows the initial state that the debug window should have, if we have followed all the steps:

- On the left side of the central panel we find the code in Table 1.12, with the first line of code after the start label marked in green and with an arrow in the left frame. This means that the program has successfully started its simulated execution and it is now stopped at the breakpoint we have set at the address of start.

- On the right side of the central panel we find the disassembly code. This is nothing else than the memory contents around the address of the current instruction. However, instead of showing the memory contents as binary or hexadecimal digits, the tool has reinterpreted the contents as instructions, which is, of course, highly convenient for the programmer.

It is worth noting the differences between both halves of the figure. In the left panel we have the source code, as we programmed it, making use of the facilities provided by the assembler. On the right side we have the instructions generated by the assembler. Consider, for example, the instruction `LDR R4, =X`, which is a pseudo-operation of the assembly language for writing in register R4 the address corresponding to label X. As we can see in the right panel, this instruction has been replaced by `LDR R4, [PC, #36]`, which uses a valid addressing mode for the load instruction of ARM. We can also observe that the address equivalent to PC+36 appears as a comment after the semicolon. We can use either of the two panels to insert breakpoints. If we activate the button  in the disassembly panel, the source code lines will be interleaved with the corresponding disassembly lines. This may be helpful when we have to track the disassembly code, especially when this code has been compiled from a high-level language, like C.

4. Before simulating our code we will open a memory viewer to check the value of our variables. To do this, we must first open the Memory tab in the lower panel, if it is not already open (Window→ Show View → Memory). We must select this tab and add a new viewer by clicking on the icon . A window similar to Figure 1.18 will open, where we will write the address from which we want to monitor the memory. If we write the download address, 0x0C000000 (where we have placed the section .data, followed by the sections .bss and .text), then the aspect of the resulting window should be similar Figure 1.19.
5. Now, in order to simulate the whole program, we must click on the Resume icon () or F8, and then click on the Suspend icon ().
 - How can we know whether the code has run successfully? If so, the largest of both input data must have been written to the memory location reserved for this purpose, which we had labeled as Largest. You can check its value in the memory viewer, which will appear marked in red as shown in Figure 1.20.
6. Nevertheless, to better understand how each of the instructions works, it will be advisable to execute the program again, but this time step by step. Furthermore, if the result of a program is not correct we will have to debug the code to find out the wrong instruction, for which a step by step execution will be very useful. To do this we have to:
 - Stop the simulation by clicking the Terminate button (). This will enable the button Remove all terminated launches () , which will allow us to clean the Debug panel. Note that sometimes this button might not be enabled. Should this happen, click the right mouse button on the debug session and select Terminate and Remove.
 - If we need to modify the code, we must open the C/C++ perspective, edit the files, save them and recompile the project. Next we can return to the debug perspective.

- We can launch a debugging session using the last debug configuration by pressing the () button.
- We can do a step by step execution/simulation using the following options:
 - Step Over (): the next instruction is executed. If this is a subroutine call, then the program will stop after the complete execution of the subroutine.
 - Step Into (): the next instruction is executed. However, in this case, if the instruction is a subroutine call, the program will stop at the first instruction of the subroutine.
 - Set a breakpoint and resume execution by pressing the Resume button (). To set a breakpoint we must double click on the left margin of any instruction (either in the source panel or the disassembly panel). The mark () will display. We can set several breakpoints. The execution will stop at the breakpoint that is reached first.

Additionally, we might be interested in monitoring the changes produced by our program while it is being executed. So:

- If we want to see how the values of the registers are changing as the execution of the program progresses, we must use the register viewer, located in the upper right panel.
- Similarly, on the memory viewer we can observe the evolution of the contents of the memory locations that are being addressed by our program.

Figure 1.21 shows an example of a debugging session, where we have been running the program step by step until the instruction following `BLE` else. We can see that the condition is false and thus the execution proceeds to the else block. We can see the state of the registers at this time, and how the PC value is `0x0C000030`, which corresponds to the address of the next instruction to be executed. We can also see how the disassembly panel highlights the latest executed instructions in green, while leaving unmarked the instructions of the then branch.

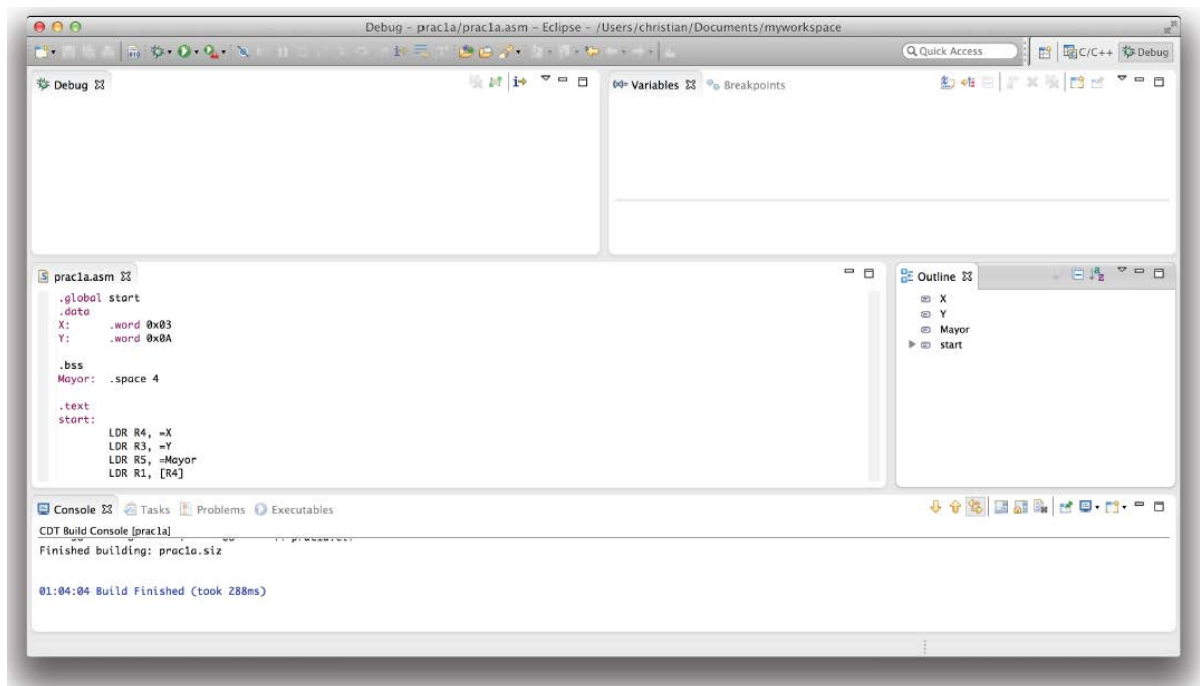


Figure 1.12 Debugging perspective

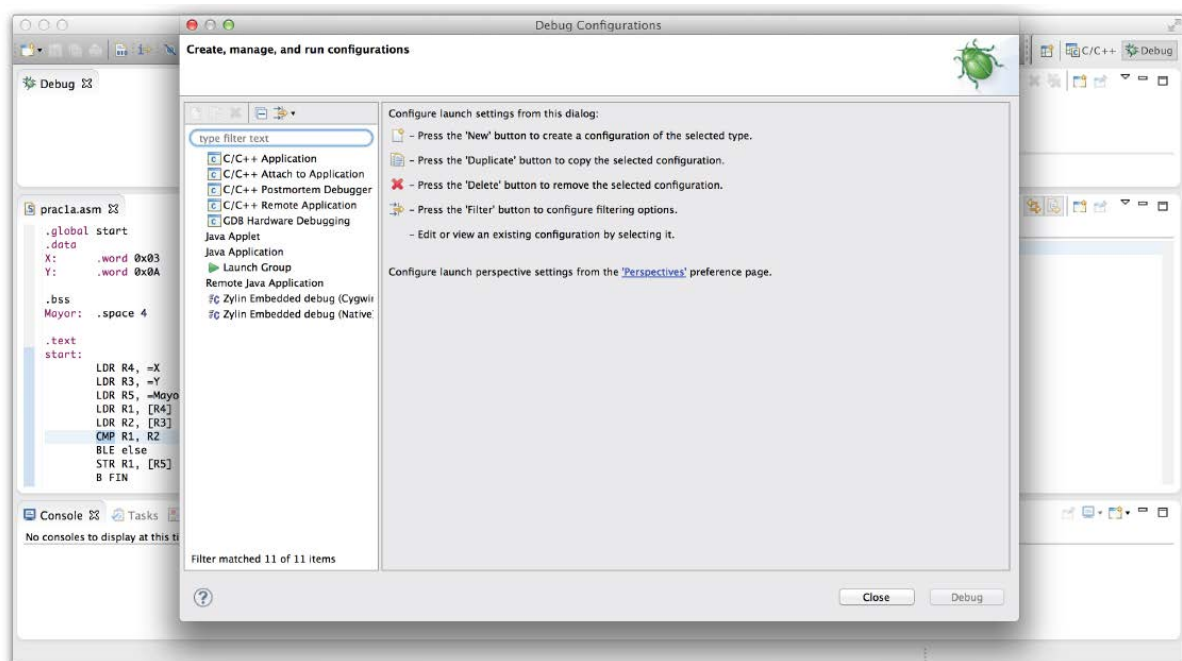


Figure 1.13 Debugging configurations window.

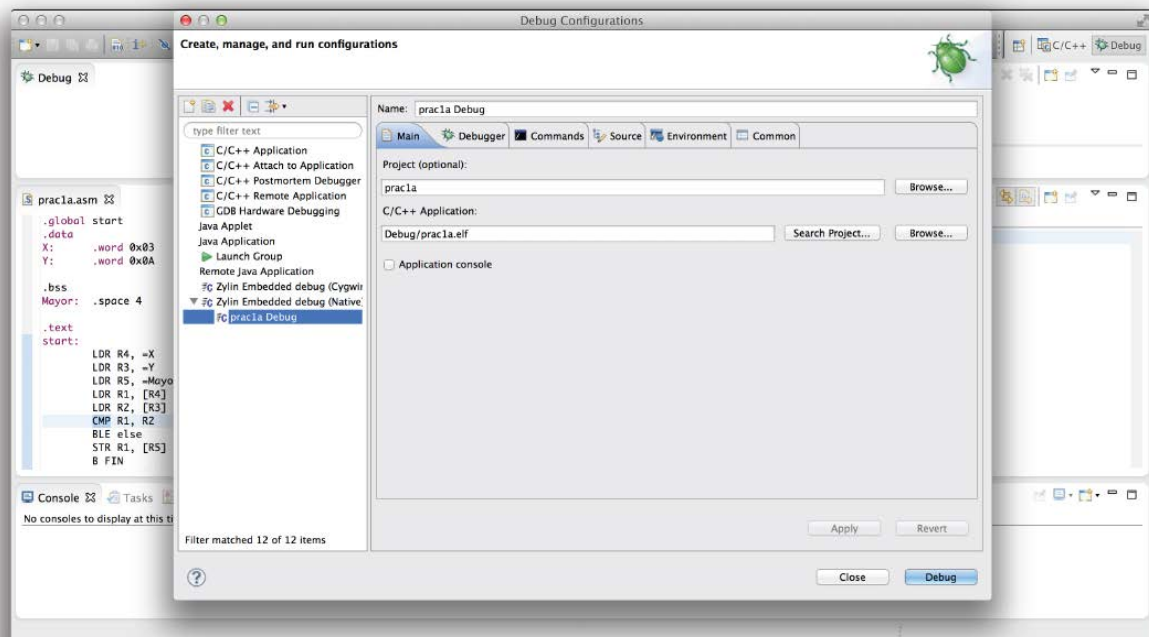


Figure 1.14 Creating a new Zylín native configuration for the project

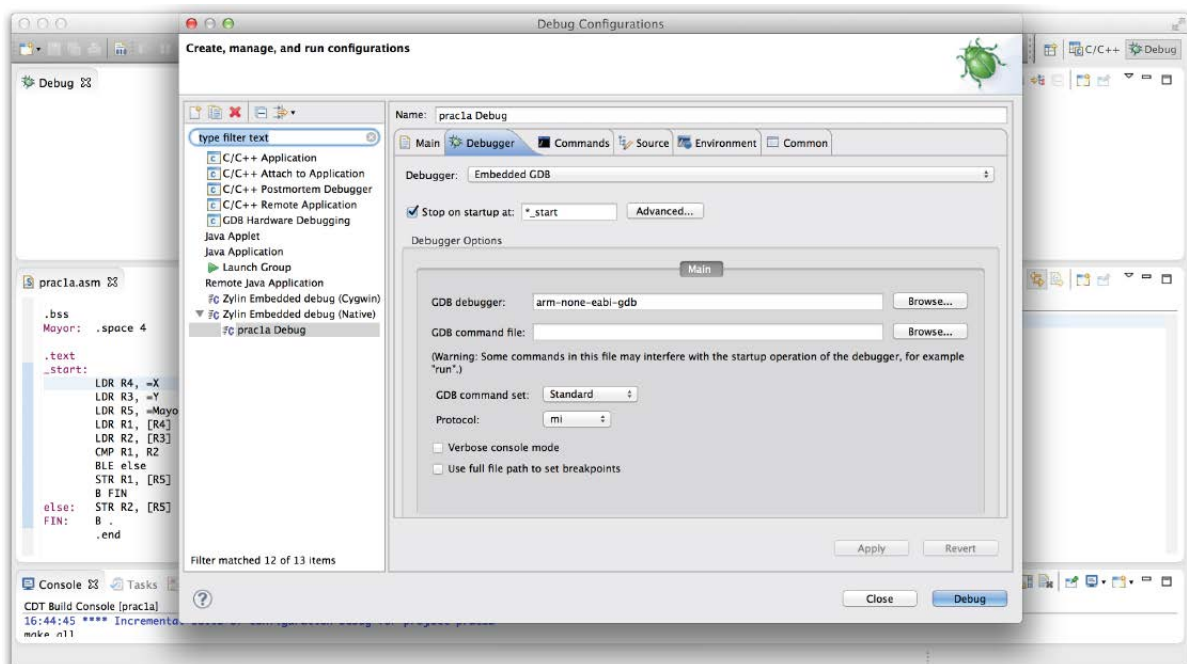


Figure 1.15 The debugger tab of the Zylín debugging configuration

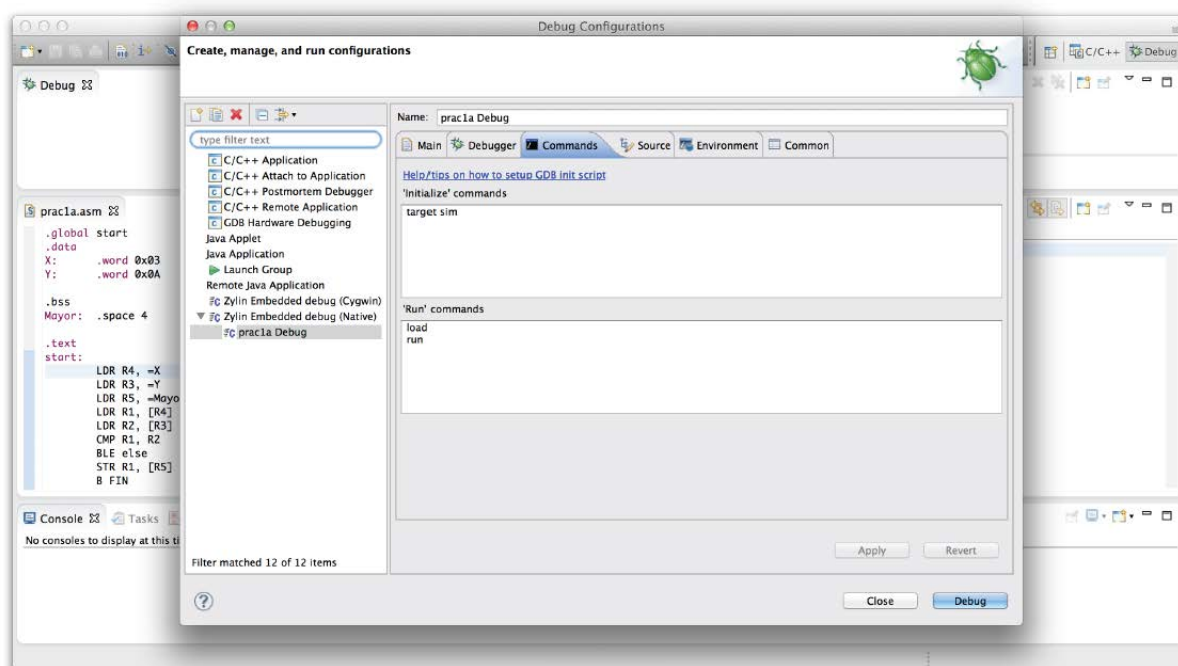


Figure 1.16 The commands tab of the Zylind debugging configuration

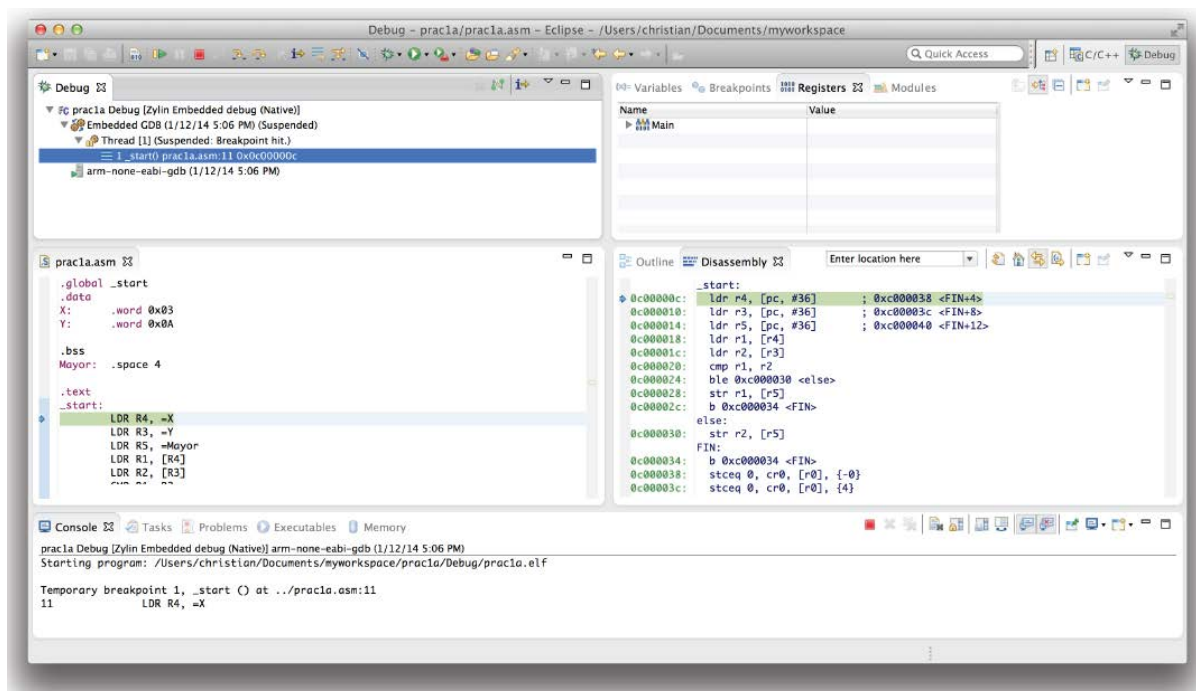


Figure 1.17 Initial look of the debugging window.

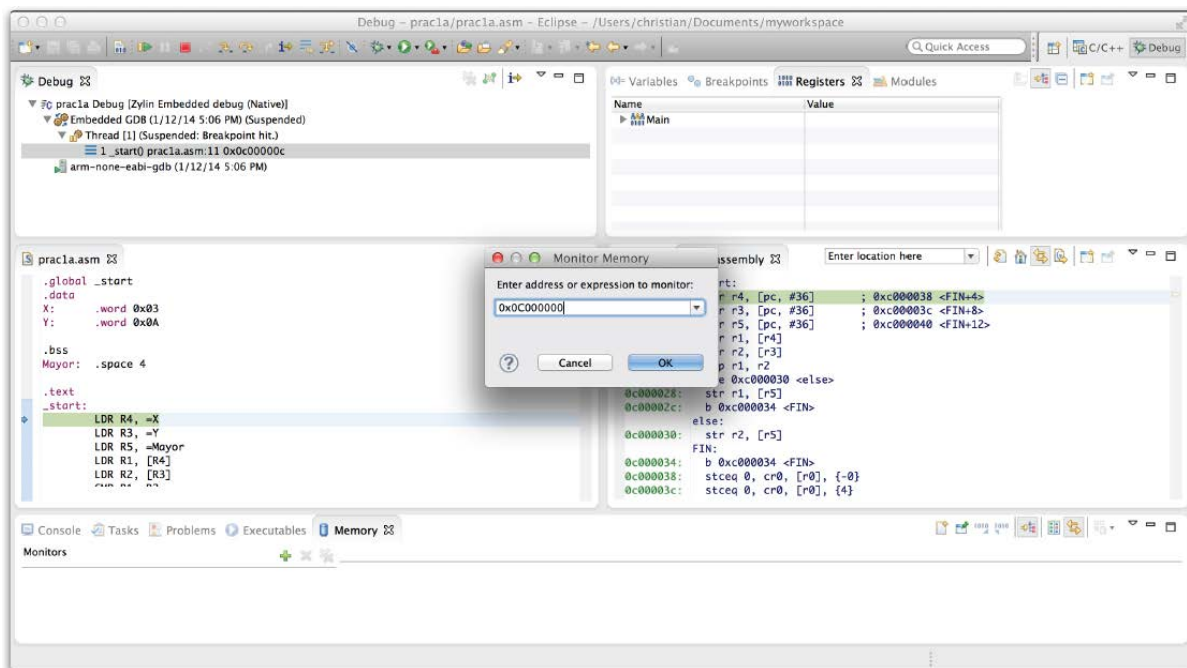


Figure 1.18 Specifying an address for the memory monitor.

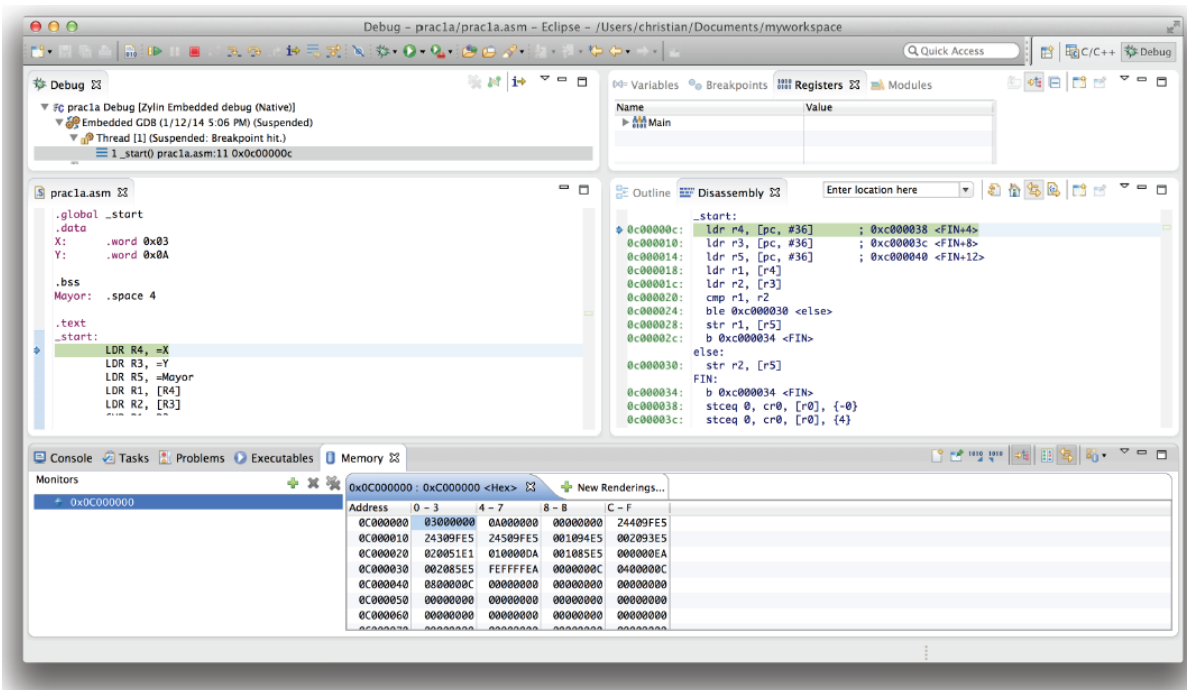


Figure 1.19 Memory contents displayed by the memory monitor (from address 0x0C000000).

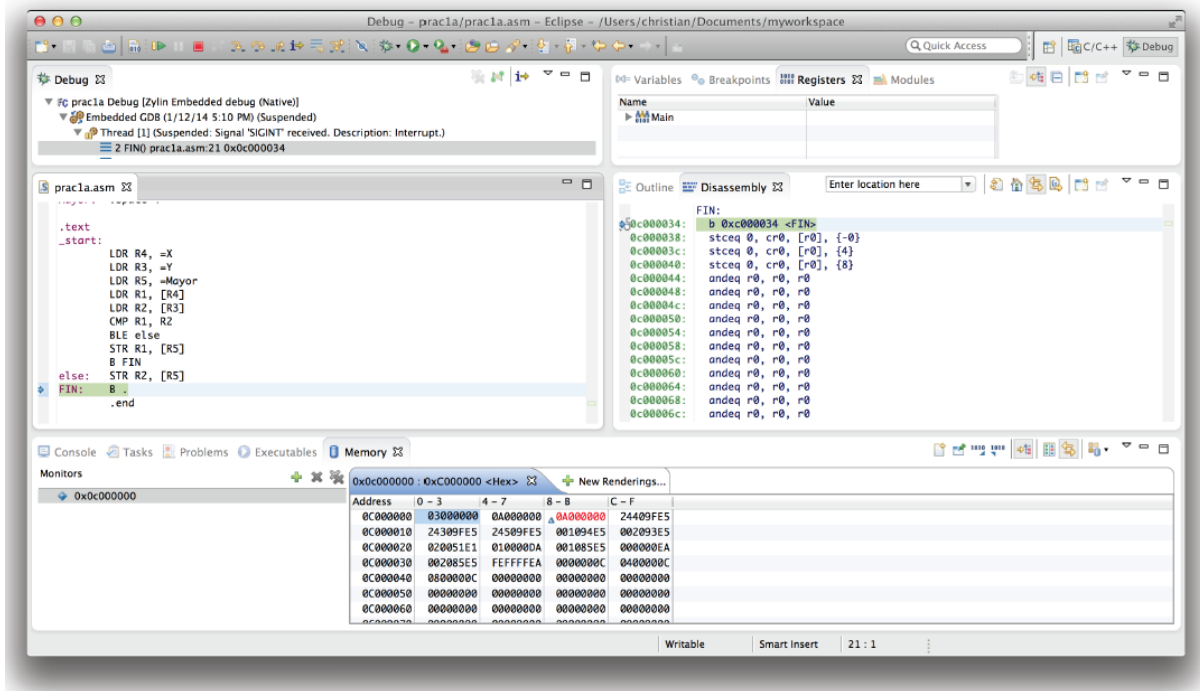


Figure 1.20 Memory contents after the complete execution of the program.

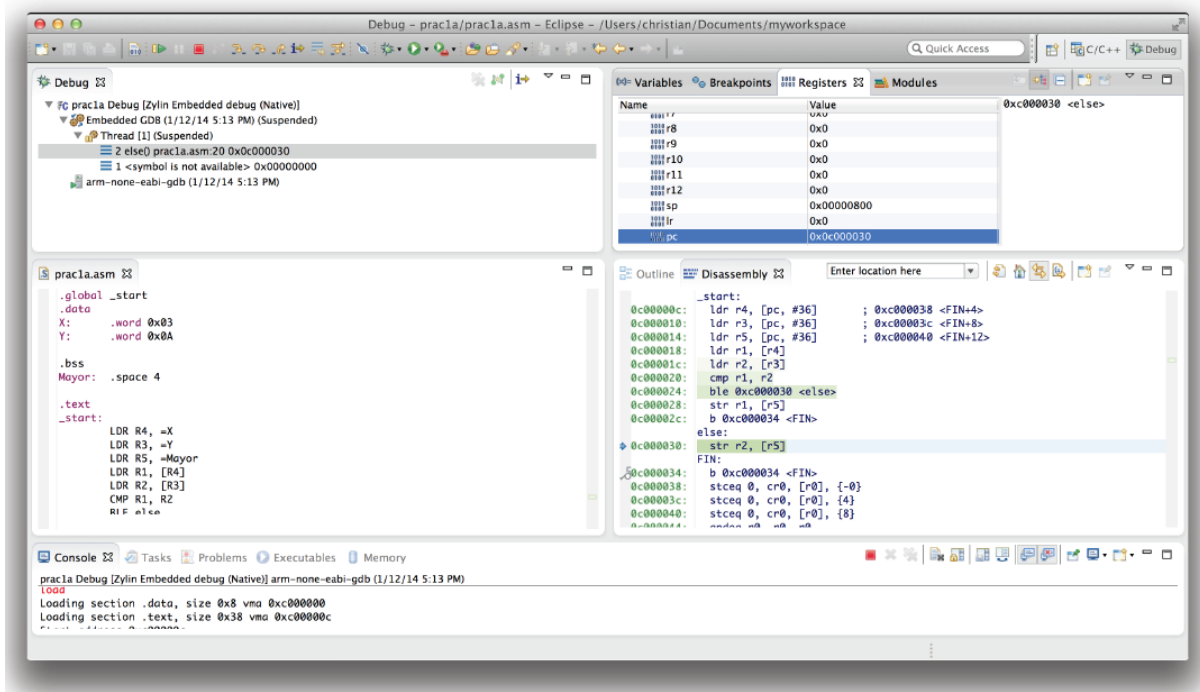


Figure 1.21 A debugging session showing the register viewer



1.7 Development of the lab session

The student must complete the following tasks and show the results to the teacher.

1. Detailed implementation of the example presented in section 1.6.
2. Develop and debug a program in ARM assembly code that implements the integer division of two word-sized positive numbers, A and B, by means of iterative subtractions, using the algorithm shown in Table 1.14. The quotient, C, must be stored in memory.
3. Develop a program, of similar characteristics to the previous one, which will be specified during the lab session.

Table 1.14 Pseudocode to implement the division A/B by iterative subtractions

```
C = 0
while (A >= B) {
    A = A - B
    C = C + 1
}
```