**2.4.1 Compiling, making, debugging, and running**

**Copy the code from Building a program on page 2-3 into CodeWarrior. There are separate functions in CodeWarrior to compile, make, debug and run a program. Experiment with all four and describe what each does.**

arm-elf-gcc compila o arquivo, que lê e escreve, arm-elf-gdb debuga e para rodar é só escrever ./<nome do arquivo> que não possui extensão.

**2.4.2 Stepping and stepping in**

**Debug the code from Building a program on page 2-3. Instead of running the code, step all the way through the code using both the step method and the step in method. What is the difference between the two methods of stepping through the assembly code?**

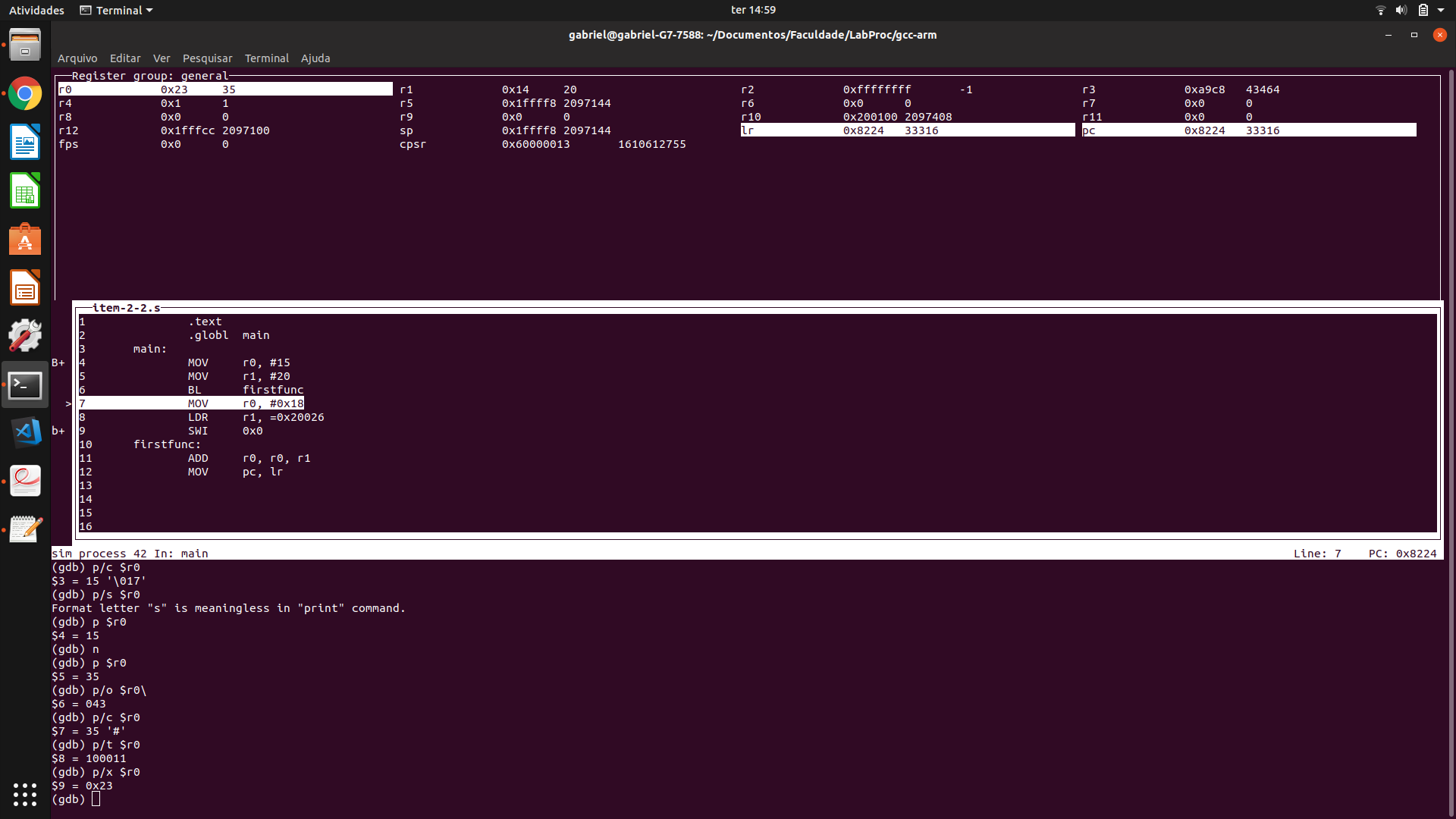
Step segue linha por linha entrando em todas as subrotinas.

Next segue linha por linha apenas da função principal, pulando todas as chamadas de subrotinas.

**2.4.3 Data formats**

**Sometimes it is very useful to view registers in different formats to check results more efficiently. Run the code from Building a program on page 2-3. Upon completion, view the different formats of r0 and record your results. Specifically, view the data in hexadecimal, decimal, octal, binary, and ASCII.**

A imagem abaixo mostra os diferentes formatos do registrador r0.



**3.10.1 Signed and unsigned addition**

**For the following values of A and B, predict the values of the N, Z, V and C flags produced by performing the operation A + B. Load these values into two ARM registers and modify the program created in Building a program on page 2-3 to perform an addition of the two registers. Using the debugger, record the flags after each addition and compare those results with your predictions. When the data values are signed numbers, what do the flags mean? Does their meaning change when the data values are unsigned numbers?**

**0xFFFF0000 0xFFFFFFFF 0x67654321 (A)**

**+ 0x87654321 + 0x12345678 + 0x23110000 (B)**

Como a soma é realizada em complemento de 2, então não é possível fazer a conta sem sinal. E a primeira soma retorna N = 1, Z = 0, C = 1, V = 0, a segunda soma retorna N = 1, Z = 0, C = 1, V = 0 e a terceira soma retorna N = 1, Z = 0, C = 0, V = 1.

**3.10.2 Multiplication**

**Change the ADD instruction in the example code from Building a program on page 2-3 to a MULS. Also change one of the operand registers so that the source registers are different from the destination register, as the convention for multiplication instructions requires. Put 0xFFFFFFFF and 0x80000000 into the source registers. Now rerun your program and check the result.**

**1.Does your result make sense? Why or why not?**

Para termos um resultado que faz sentido, precisamos utilizar números que caibam em 16 bits nos parâmetros do MULS. Porém, esse não é o caso com esses valores

**2.Assuming that these two numbers are signed integers, is it possible to overflow in this case?**

Sim, que o caso do resultado do item anterior

**3. Why is there a need for two separate long multiply instructions, UMULL and SMULL? Give an example to support your answer**

Porque para operações com endereços, por exemplo, não existem negativos: queremos que o processador interprete o resultado da multiplicação corretamente em cada caso.

**3.10.3 Multiplication shortcuts**

**Assume that you have a microprocessor that takes up to eight cycles to perform a multiplication. To save cycles in your program, construct an ARM instruction that performs a multiplication by 32 in a single cycle.**

**3.10.4 Register-swap algorithm**

**The EOR instruction is a fast way to swap the contents of two registers without using an intermediate storage location such as a memory location or another register. Suppose two values A and B are to be exchanged.**

**The following algorithm could be used:**

**A = A ⊕ B**

**B = A ⊕ B**

**A = A ⊕ B**

**Write the ARM code to implement the above algorithm, and test it with the values of A=0xF631024C and B =0x17539ABD. Show your instructor the contents before and after the program has run.**