

An Introduction to Assembly Programming with the ARM 32-bit Processor Family

G. Agosta
Politecnico di Milano

December 3, 2011

Contents

1	Introduction	1
1.1	Prerequisites	2
1.2	The ARM architecture	2
2	A First Assembly Program	3
2.1	Assembler Program Template	3
2.2	Using the GNU Assembler	3
2.3	The GNU Debugger	4
3	The Assembly Language	5
3.1	Arithmetic-Logic Instructions	5
3.2	Branch Instructions	6
3.2.1	Translating Conditional Statements	7
3.2.2	Translating While and Do-While Statements	8
3.3	Load/Store Instructions	10
3.4	The Global Data Section	11
4	Function Invocation and Return	12
4.1	Runtime Architecture	13
4.2	The Program Stack	14

1 Introduction

Assembly languages are used to program computers at a level that is close enough to the hardware to enjoy significant insights into the actual hardware and its performances.

As computer engineers, we are interested in understanding assembler programming chiefly because it gives us a better grip on the way higher level languages such as the C language are implemented – what is implied in a function call, for loop or array access? How can such complex structures be expressed when only registers, memory, and simple three-operand instructions are available?

Just like high level language programs, assembly programs are “compiled” to a lower-level representation, the machine code. This process is called “assembling”, and the software tool that performs it is know as the *assembler*.

1.1 Prerequisites

To make the most from the present material, some understanding of the C programming language (or another high-level language) is needed. To check you fulfill this requirement, consider the following C program:

```
1 #include <string.h>
2
3 int main(int argc, char **argv){
4     int count=0;
5     int i;
6     for(i=1; i<argc; i++)
7         if (strlen(argv[i]>argc) count++;
8     return count;
9 }
```

What does the program compute as its exit value, as a function of the program arguments? ¹

1.2 The ARM architecture

The ARM architecture is a relatively simple implementation of a *load/store* architecture, i.e. an architecture where main memory (RAM) access is performed through dedicated *load* and *store* instructions, while all computation (sums, products, logical operations, etc.) is performed on values held in *registers*.

For example, `ADD R4, R1, R2` will store in register `R4` the sum of the values held in `R1` and `R2`.

The ARM architecture provides 16 registers, numbered from `R0` to `R15`, of which the last three have special hardware significance.

`R13`, also known as `SP`, is the stack pointer, that is it holds the address of the top element of the program stack. It is used automatically in the `PUSH` and `POP` instructions to manage storage and recovery of registers in the stack.

`R14`, also known as `LR` is the link register, and holds the return address, that is the address of the first instruction to be executed after returning from the current function.

`R15`, also known as `PC` is the program counter, and holds the address of the next instruction. ²

We will have a better look at special use registers in Section 4.1. For now, we will just avoid using them to store variable values needed in our programs.

¹No, I’m not going to give you the solution. If you can’t find it on your own, you’d better ask, because you need to fix your basic C language skills as soon as possible!

²Due to architectural specificities, in ARM this is the address of the *second* next instruction! Thus, it is better not to modify the `PC` by hand!

2 A First Assembly Program

In this section, we will write a very simple assembly program, which can be used as a template for others, compile and link it, and execute it. To do so, it is assumed that you have installed a Qemu ARM virtual machine, using the instructions provided on my website.³

Assembler languages only define instructions and hardware resources. To drive the assembly and linking processes, we also need to use *directives*, which are interpreted by the assembler. We will use the *GNU Assembler (as)*, with its specific directives.

2.1 Assembler Program Template

Here is the simplest program we can write in the ARM assembler language, one that immediately ends, returning control to the caller.

```
1 .arm    /* Specify this is an ARM 32-bit program */
2 .text  /* Start of the program code section */
3 .global main /* declares the main identifier */
4 .type  main, %function
5 main:  /* Address of the main function */
6        /* Program code would go here */
7        BX LR /* Return to the caller */
8 .end   /* End of the program */
```

The `.arm` directive specifies that this program is an ARM 32-bit assembly code. The ARM architecture includes other types of programs (which we will not see here), so this directive is needed for the Assembler to decide how to assemble the program.

The `.text` directive indicates the start of a section of the assembly program that contains code. Other sections could contain, e.g., the definition of global data (i.e., global variables).

The `.global main` line specifies that `main` is a globally visible identifier, which can be referred to from outside this file⁴ This is necessary, because `main` is the program entry point, and the corresponding function will be invoked from outside the program!

`.type main, %function` specifies that the `main` identifier refers to a function. More precisely, it is a *label* indicating the address of the first instruction of the main function.

`main: BX LR` is the first (and only) instruction of this program. It is prefixed with a label, which allows to refer to the instruction address from other points in the same compilation unit, or other units if, as in this case, the symbol `main` is defined as a global identifier.

The `.end` directive specifies the end of the program.

2.2 Using the GNU Assembler

The GNU Assembler can be invoked as a standalone program, with a line such as:

³<http://home.dei.polimi.it/agosta>

⁴More precisely, outside this *compilation unit*.

```
1 as -o filename.o filename.s
```

This will result in a *object file*, that is a machine code file, which is still not executable, because some external symbols (e.g., library functions) may not be resolved yet. To complete the compilation, we need to link the file, using the GNU linker (as we would do with an object file derived from the compilation of a high level language program).

However, the GNU C Compiler (`gcc`) wraps the invocation of the assembler and linker, so it is simpler for our purpose to just call the compiler itself:

```
1 gcc -o filename.x filename.s
```

This will result in a fully assembled and linked program, ready for execution.

Executing this program will not result in any fancy effect: indeed, the program does nothing, except for terminating. However, we do have two ways to check that it is actually working. The first exploits the fact that, in the ARM *Runtime Architecture*⁵ the first register (R0) is used to pass the first parameter to a function, and the return value from a function as well. So, our first program is equivalent to the following C language program:

```
1 int main(int argc){
2     return argc;
3 }
```

where the value of the `argc` variable is held in register *R0* for the entire life of the program.

So, we can check that the return value of the program is actually the same as the number of parameters passed to it, plus one. We can do so with the following command, given to the command line of our virtual machine:

```
1 echo $_
```

This command prints out the return value of the last executed program, thus allowing us to see the value of `argc` printed out to the console.

2.3 The GNU Debugger

The second way of checking on the behaviour of our program is to follow its execution through the *debugger*.

To prepare the executable file for execution with the debugger, we recompile it adding the `-g` flag, which specifies to the compiler that the debugging support should be activated:

```
1 gcc -g -o filename.x filename.s
```

The debugger is invoked with the following command:

```
1 gdb ./filename.x
```

where “filename.x” is the name of our executable file.

Within the debugger shell, we use the following command to set up the execution, start it and block it at the beginning of the main function:

⁵The Runtime Architecture is the set of conventions that allow programs and functions written by different programmers to cooperate, by specifying how parameters are passed, and which information is preserved across a function call.

```

1 set args myFirstArgument mySecondArgument etCetera
2 break main
3 run

```

Then, we can use the `print` command to show the value of any variable (when debugging high-level language programs), the `step` command to progress to the next instruction and the `info register` to see the contents of all registers.

The debugger has many more commands to control and inspect the execution. They are documented by means of an internal help, accessed with the `help` command.

3 The Assembly Language

As shown in the first example, assembly statements are structured in the following way:

```

1 label: instruction

```

Comments can be inserted in the C style (delimited by `/* */` strings). The label is optional – it is possible to define instructions without a label. The label is used to refer to the instruction address in the code – in particular, this is necessary to perform *branch* (also know as jump) instructions. Labels (like other symbols) can be any sequence of alphanumeric characters. The underscore (`'_'`) and dollar (`'$'`) characters can also be used, and numeric characters (digits) cannot appear as the first character of a symbol.

Instructions can be divided in three different sets:

Arithmetic-Logic instructions perform mathematical operations on data: these can be arithmetic (sum, subtraction, multiplication), logic (boolean operations), or relational (comparison of two values).

Branch instructions change the control flow, by modifying the value of the Program Counter (*R15*). They are needed to implement conditional statements, loops, and function calls.

Load/Store instructions move data to and from the main memory. Since all other operations only work with immediate constant values (encoded in the instruction itself) or with values from the registers, the load/store instructions are necessary to deal with all but the smallest data sets.

In the rest of this section, we will present a subset of the ARM instruction set that can be used to produce complete programs, but is much smaller than the full set.

3.1 Arithmetic-Logic Instructions

Arithmetic instructions include the following (where *Rd*, *Rn* and *Rm* are any three registers, and `#imm` are 8-bit immediates):

Syntax	Semantics
ADD Rd Rn Rm/#imm	Rd = Rn + Rm/#imm
SUB Rd Rn Rm/#imm	Rd = Rn - Rm/#imm
MUL Rd Rn Rm	Rd = (Rn * Rm/#imm)%2 ³² (i.e., truncated at 32 bits)
AND Rd Rn Rm/#imm	Rd = Rn & Rm/#imm (bitwise and)
EOR Rd Rn Rm/#imm	Rd = Rn ^ Rm/#imm (bitwise exclusive or)
ORR Rd Rn Rm/#imm	Rd = Rn Rm/#imm (bitwise or)
MVN Rd Rm/#imm	Rd = ~ Rm/#imm (bitwise negation)
CMP Rn Rm/#imm	Status ← comparison (<, >, ==) of Rn and Rm/#imm
MOV Rd Rm/#imm	Rd = Rm/#imm

Most arithmetic instructions are rather straightforward: the contents of `Rd` are replaced with the results of the operation performed on the values of `Rn` and `Rm` (or `Rn` and a constant `#imm`).

For example, the instruction `SUB R2 R2 #12` reduces the value of `R2` by 12. Also, it is easy to translate C language statements to ARM assembly instructions, as long as all variable values are stored in the registers. E.g., consider the following C fragment:

```
1 int i=0, j=2, k=4;
2 i=k | j; /* Bitwise or */
```

Assuming the values of the three variables are stored respectively in the registers `R4`, `R5`, and `R6`, then we can translate the fragment as follows:

```
1 MOV R4 #0 /* i=0; */
2 MOV R5 #2 /* j=2; */
3 MOV R6 #4 /* k=4; */
4 ORR R4 R5 R6 /* i=k | j; */
```

It is important to note that multiplication is more difficult to implement in hardware than other instructions. Thus, the `MUL` instruction presented truncates the result to the least significant 32 bits. Contrary to most other instructions, in `MUL Rd Rn Rm` `Rd` and `Rm` cannot be the same register.⁶

The comparison instruction, `CMP`, does not store its result in the general purpose registers. It uses instead a special (not directly accessible to the programmer) register, called the Processor Status Word. The comparison result tells the processor whether the value in `Rn` is greater, lesser or equal to `Rm` (or the constant operand). This information can then be used to decide whether to perform a successive conditional branch instruction or not.

3.2 Branch Instructions

The basic functionality of the branch instruction is to change the Program Counter, setting it to a new value (usually a constant, but not always). The set of branch instructions includes the following:

⁶This restriction has been removed in new version of the ARM instruction set, but the virtual machine provided for the course shows a slightly older and more common version, which features this restriction.

Syntax	Semantics
<code>B label</code>	jump to label (unconditional)
<code>BEQ label</code>	jump to label if previously compared values were equal
<code>BNE label</code>	jump to label if previously compared values were different
<code>BGT label</code>	jump to label if previously compared <code>Rn > Rm/#imm</code>
<code>BGE label</code>	jump to label if previously compared <code>Rn >= Rm/#imm</code>
<code>BLT label</code>	jump to label if previously compared <code>Rn < Rm/#imm</code>
<code>BLE label</code>	jump to label if previously compared <code>Rn <= Rm/#imm</code>
<code>BL label</code>	function call (label is the function name/entry point)
<code>BX Rd</code>	return from function (always as <code>BX lr</code>)

`B label` is the basic branch. Control is changed to run the instruction labeled with `label` next.

`BL label` performs a function call, which means it jumps as in `B label`, but also saves the value of the Program Counter in the Link Register (as by `MOV LR PC`).

Return from function is performed by `BX LR`. It is *almost* the same as `MOV PC LR`, but the PC should not be manipulated explicitly by programmer, unless there is a very good reason to do so.

All other instructions are conditional variants of the basic branch. They perform as `B label`, if the specified condition was met. E.g., consider the following fragment:

```

1 CMP R0 #0
2 BGT label
3 ADD R0 R0 #1

```

Here, we first compare `R0` to 0, then branch to `label` if (and only if) the value contained in `R0` is greater than zero. Otherwise, we step to instruction 3.

We can use any comparison condition – the table above lists the most common conditions.

3.2.1 Translating Conditional Statements

A key issue in understanding assembly is to understand how high level control statements can be implemented in the lower level assembly language. The first step is to learn how to represent in assembly the “if-then-else” construct of C (and other high-level languages).

Let us consider a C code fragment containing a conditional, such as this:

```

1 int x,y;
2 ...
3 if(x>0) {
4   y=1;
5 } else {
6   y=2;
7 }
8 return y;

```

Assuming `x` and `y` are stored in `R4` and `R5` respectively, we can translate the C fragment as follows:

```

1      CMP R4 #0 /* we need to check x>0 */
2      BLE else /* if (!(x>0)) go to else */
3      MOV R5 #1 /* y=1; */
4      B end /* skip the else part */
5 else : MOV R5 #2 /* y=2; */
6 end  : MOV R0 R5 /* R0 is the return value */
7      BX LR /* return from function */

```

We can see that, at line 1 and 2 in the assembly code, we are setting up and executing a conditional branch, which allows us to skip the “then” part of the code ($y=1$; in our example) if the condition ($x>0$) is not respected. Otherwise, the standard behaviour (falling through to the next instruction) is used, and the “then” part is executed. In this latter case, we need to ensure that the “else” part is not executed – in the original C construct, the two parts are mutually exclusive: only one is executed after the condition is resolved. Thus, at line 4, we add an unconditional branch `B end` to skip the “else” part and move directly to the end of the “if-then-else” statement.

3.2.2 Translating While and Do-While Statements

Having understood how to implement conditional statements, we need now to do the same for loop statements.

Let us start from a `while` statement, such as the one shown in the following C code fragment:

```

1 int x,y;
2 ...
3 y=1;
4 while(x>0) {
5     y*=x;
6     x--;
7 }
8 return y;

```

Note that the code fragment computes the following mathematical function, for the given value of x :

$$y \leftarrow x! \% 2^{32}$$

Assuming x and y are stored in $R4$ and $R5$ respectively, we can translate the C fragment as follows:

```

1      MOV R5 #1 /* y=1; */
2 loop : CMP R4 #0 /* we need to check x>0 */
3      BLE end /* if (!(x>0)) end the loop */
4      MUL R6 R5 R4 /* tmp=y*x; */
5      MOV R5 R6 /* y=tmp; */
6      SUB R4 R4 #1 /* x=x-1; */
7      B loop /* loop back to condition */
8 end  : MOV R0 R5 /* R0 is the return value */
9      BX LR /* return from function */

```

We can see that, at line 1 and 2 in the assembly code, we are setting up and executing a conditional branch, which allows us to exit the loop when the

condition in the original code ($x > 0$) is not met anymore. Otherwise, the standard behaviour (falling through to the next instruction) is used, and the loop body is executed. After the end of the loop body (line 6), we need to return to the check of the condition, to start a new iteration of the loop. Thus, at line 7, we add an unconditional branch `B loop` to go back to the beginning of the loop, marked by the `loop` label.

Given a `do-while` loop, we can perform a similar translation. Here is a C fragment that computes the same factorial function, except that for $x == 0$, the return value is 0 rather than 1.

```

1 int x,y;
2 ...
3 y=1;
4 do {
5     y*=x;
6     x--;
7 } while(x>0);
8 return y;
```

Using the same assumptions on data storage as before, we get the following translation:

```

1      MOV R5 #1      /* y=1;          */
2 loop : MUL R6 R5 R4 /* tmp=y*x;       */
3      MOV R5 R6      /* y=tmp;         */
4      SUB R4 R4 #1   /* x=x-1;        */
5      CMP R4 #0      /* we need to check x>0 */
6      BGT loop      /* loop back to if x>0 */
7 end   : MOV R0 R5   /* R0 is the return value */
8      BX LR        /* return from function */
```

The only notable difference with respect to the `while` version is that it is now the branch back to the loop head that is conditional, and we only need one branch, since there is no control at the beginning. Of course, this is because we do not check that $x > 0$ at the beginning of the loop.

It might seem that translating the `for` loop involves a greater difficulty, because the loop statement is more complex. However, it is not so, because the C `for` loop can be immediately transformed into a C `while` loop! For example, consider the following C code fragment:

```

1 int x,y;
2 ...
3 for(y=1; x>0; x--) y*=x;
4 return y;
```

It is easy to see that the loop at line 3 has exactly the same behaviour as the `while` loop. More in general, a C `for` loop has the following structure: `for(initialization; condition; loop_step)loop_body;`, where `initialization`, `condition`, and `loop_step` are any three C expressions, and `body` is a C statement (possibly compound). It can be transformed in a `while` loop in this way:

```

1 initialization;
```

```

2 while(condition){
3     loop_body;
4     loop_step;
5 }

```

Translation to assembly, in general, passes through this C-to-C transformation (called *lowering*).

3.3 Load/Store Instructions

Last but not least, load and store instructions perform data transfer from memory to registers and vice versa. The following instructions allow us to work with *word* and *byte* sized data:

Syntax	Semantics
LDR Rd [Rn, Rm/#imm]	Rd = mem[Rn+Rm/#imm] (32 bit copy)
STR Rd [Rn, Rm/#imm]	mem[Rn+Rm/#imm] = Rd (32 bit copy)
LDRB Rd [Rn, Rm/#imm]	Rd = mem[Rn+Rm/#imm] (8 bit copy)
STRB Rd [Rn, Rm/#imm]	mem[Rn+Rm/#imm] = Rd (8 bit copy)
PUSH { register list }	Push registers onto stack
POP { register list }	Pop registers from stack

In load instructions, values are read from a certain memory address and moved to register Rd, while in store instructions Rd is the source from which data is read, and the values are stored in memory. The memory address employed is, in both cases, computed as the sum of a base Rn and an offset (Rm or a constant). Usually, the programmer uses this capability to access arrays of data. Consider the following C fragment:

```

1 char AA[10], BB[10];
2 ...
3 AB[i]=BB[i];

```

where AA and BB are arrays. If the base addresses of AA and BB are kept in R4 and R5, and i is stored in R1, then the statement at line 3 can be translated as

```

1 LDRB R6 R5 R1
2 STRB R6 R4 R1

```

Note that, since the memory is byte-addressed (each address corresponds to a specific byte), when an array of integers is to be accessed, we need to modify the array index to be incremented in steps of 4 rather than steps of 1. For example, consider the following C code fragment:

```

1 int AA[10], BB[10];
2 ...
3 i=2;
4 AA[i]=AB[i];

```

The corresponding ARM assembly translation, using the same storage for addresses and variables, would be:

```

1 MOV R1 #8 /* 2*4: addressing in bytes! */
2 LDR R6 R5 R1
3 STR R6 R4 R1

```

The `PUSH` and `POP` instructions are used to handle the program stack, that is the collection of *activation records* for all functions that are active at the same time, as a result of nested function calls. We will study the behaviour of these instructions together with that of the branch instructions used to perform the function invocation and return in greater details in the next section.

3.4 The Global Data Section

Any program sees a memory space reserved to it. This memory space can be used to hold data: for example, we may have global scalar variables, strings, arrays, or matrices.

To reserve space in the global data memory, we just need to give the assembler enough information on the size of the memory space needed, as well as the initialization values. Directives `.int`, `.byte`, `.float`, `.string` respectively reserve memory for a 32-bit integer, 8-bit integer, 32-bit floating point number, and a zero-terminated string (the size of the string is one byte per character, plus one for the terminating zero).

We can use the load (and store) instructions to access such data. The following example loads and returns a value from a global variable `x`.

```

1  .arm    /* An ARM 32-bit program    */
2  .data
3  x:  .int 42 /* A global variable named x */
4  s:  .string "hello,␣world!"
5  .text /* Start of the program code section */
6  .global main /* declares the main identifier */
7  .type main, %function
8  main: /* Address of the main function */
9  MOV R1 x /* Load address x          */
10 LDR RO R1 #0 /* Load data from x        */
11 BX LR /* Return to the caller      */
12 .end    /* End of the program      */

```

A second example loads and returns the second character from a global string `s`.

```

1  .arm    /* An ARM 32-bit program    */
2  .data
3  s:  .string "hello,␣world!"
4  .text /* Start of the program code section */
5  .global main /* declares the main identifier */
6  .type main, %function
7  main: /* Address of the main function */
8  MOV R1 s /* Load address s          */
9  LDRB RO R1 #1 /* Load second character of s */
10 BX LR /* Return to the caller      */
11 .end    /* End of the program      */

```

Note that global variables are accessed through the memory address of the first byte of storage they use.

4 Function Invocation and Return

Understanding the working of function invocation and return from function are critical to understanding how programming languages are implemented. To this end, we need to study the implementation details at the assembly level, since most, if not all, high level programming languages hide such details from the programmer.

Thus, let us go back to the original assembler program template, and add a function call to it.

```
1 .arm      /* Specify this is an ARM 32-bit program */
2 .text    /* Start of the program code section      */
3 .global  main /* declares the main identifier      */
4 .type   main, %function
5 main:    /* Address of the main function          */
6     BL factorial /* A function call!              */
7     BX LR /* Return to the caller                 */
8 .end     /* End of the program                    */
```

The code above relies on a new function, `factorial`, which we will need to implement. We have, on the other hand, already seen that parameters are passed through `R0`, so the corresponding C program would be the following:

```
1 int factorial(int x){
2     int y;
3     for(y=1; x>0; x--) y*=x;
4     return y;
5 }
6
7 int main(int argc){
8     return argc;
9 }
```

In this example, `main` is the *caller* function, while `factorial` is the callee.

Since we already have a code fragment that computes the factorial (see Section 3.2.2), translated into assembly, and we have already seen that `R0` is used to pass a single integer parameter, as well as to pass back the result value, we only need to make sure that parameter `x` is correctly looked for in `R0`.

The following program results:

```
1 .arm      /* Specify this is an ARM 32-bit program */
2 .text    /* Start of the program code section      */
3 .global  factorial
4 .type   factorial, %function
5 factorial: /* Address of the main function          */
6     MOV R5 #1 /* y=1;                               */
7 loop :   CMP R0 #0 /* we need to check x>0          */
8     BLE end /* if (!(x>0)) end the loop          */
9     MUL R6 R5 R0 /* tmp=y*x;                          */
10    MOV R5 R6 /* y=tmp;                               */
11    SUB R0 R0 #1 /* x=x-1;                               */
12    B loop /* loop back to condition          */
13 end :   MOV R0 R5 /* R0 is the return value          */
```

```

14         BX LR          /* return from function */
15
16 .global main /* declares the main identifier */
17 .type main, %function
18 main:      /* Address of the main function */
19         BL factorial /* A function call! */
20         BX LR /* Return to the caller */
21 .end       /* End of the program */

```

4.1 Runtime Architecture

For more complex functions, just having R0 is not sufficient, and we need a better understanding of parameter passing in ARM.

The convention for parameter passing in the Runtime Architecture for the ARM processor family ⁷ is that the first four word-sized parameters are stored in registers R0-R3 by the caller function. The remaining parameters will be stored in memory. Since the R0-R3 registers contain values that are prepared by the caller function for the callee, the caller function should save the values of those registers, which may be changed by the callee. On the contrary, all other register values must be preserved by the callee, so if the callee function needs more than four registers, it should save the register values before using them!

Overall, the following table shows the role of each register in a function call:

Register	Usage	Saved by
R0	First argument & Return value	Caller
R1	Second argument	Caller
R2	Third argument	Caller
R3	Fourth argument	Caller
R4	Local variable 1	Callee
R5	Local variable 2	Callee
R6	Local variable 3	Callee
R7	Local variable 4	Callee
R8	Local variable 5	Callee
R9	Local variable 6	Callee
R10	Local variable 7	Callee
R11	Local variable 8	Callee
R12	Linker register (IP)	Caller
R13	Stack Pointer (SP)	Callee
R14	Link Register (LR)	Callee
R15	Program Counter (PC)	Special

Note that the Program Counter, Stack Pointer and Link Register all have a special role in function calls.

- The Program Counter is changed by both the caller (to the address of the first instruction of the callee) and then by the callee (to the value passed by the caller in the Link Register).
- The Link Register is set by the caller, and must be preserved by the callee, which will use it at its return, as mentioned above.

⁷In the ARM context, this is known as the ARM Application Binary Interface, or ABI

- The Stack Pointer is used to handle free memory, and avoid that the callee overwrites data in memory that is needed by the caller. It can be changed by the callee, but needs to be restored before returning to the caller.

Finally, R12 may be used by the linker *between* the caller and the callee. We will not delve into this issue, except to note that the value of R12 must be saved by the caller.

With this new information, we know understand that our `factorial` function is not well-defined: it uses registers that are supposed to be callee-save without actually saving them. We therefore need to find some place in memory where we can save the contents of those registers, and restore them before returning to `main`.

We could try to use the `.data` section, that is the global memory, but we need to reserve space there when we assemble the program, and there we may not know how many function calls will be active at the same time, and therefore how much space we will need.

4.2 The Program Stack

To address the need to hold callee-saved values, as well as local variables and arguments in excess of the registers, the Runtime Architecture includes a Program Stack, that is an area of memory that, for each function invocation, is guaranteed to be used up to a certain address (stored in the Stack Pointer register, R13), and free from that address up. The Program Stack is managed in a descending way, that is the free memory is found at lower memory address than the Stack Pointer address.

To work with the Program Stack, we use the `PUSH` and `POP` instructions defined in Section 3.3. The `PUSH { registers }` instruction will store in the Program Stack all the operand registers (it can store even the entire register set) and update the Stack Pointer to reflect the memory used to store those values.

The `POP { registers }` instruction does the opposite, restoring the register values from memory, and freeing the memory by moving back the Stack Pointer to the original value.

We can modify our `factorial` program to take into account that registers R5 and R6 are callee-save.

```

1  .arm      /* Specify this is an ARM 32-bit program */
2  .text    /* Start of the program code section    */
3  .global factorial
4  .type factorial, %function
5  factorial: /* Address of the main function */
6             PUSH { R5, R6 } /* Save R5 and R6, not R0*/
7             MOV R5 #1      /* y=1;          */
8  loop :    CMP R0 #0      /* we need to check x>0 */
9             BLE end       /* if (!(x>0)) end the loop */
10            MUL R6 R5 R0  /* tmp=y*x;        */
11            MOV R5 R6     /* y=tmp;          */
12            SUB R0 R0 #1  /* x=x-1;          */
13            B loop       /* loop back to condition */
14  end :    MOV R0 R5      /* R0 is the return value */

```

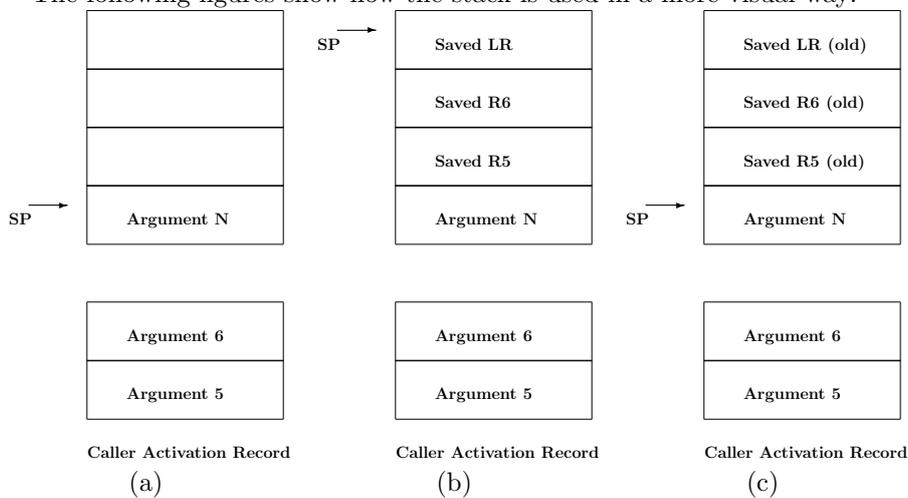
```

15     POP { R5, R6 } /* Restore R5 and R6 */
16     BX LR          /* return from function */
17
18     .global main /* declares the main identifier */
19     .type main, %function
20 main: /* Address of the main function */
21     BL factorial /* A function call! */
22     BX LR /* Return to the caller */
23     .end /* End of the program */

```

We do not need to modify `main`, because we know that the caller saved registers will not be used by `main` after returning from the call.

The following figures show how the stack is used in a more visual way:



Immediately before the function call (a), the caller function stores any parameter value (except the first four parameters, which are stored in R0–R3) into the stack. Assuming the parameters values are held in registers R4–Rn, the following operation suffices:

```

1 PUSH { R4, R5, ... Rn }

```

This also updates `SP`, moving it to the position shown in (a).

Then, after the `BL` (function call) instruction is executed, the callee function will store the values of the callee-save registers it needs to use, by performing a second `PUSH` operation (`PUSH { R5, R6, LR }` in the figure), leading to the stack state shown in (b).

Finally, before completing its execution with the `BX LR` instruction, the callee will restore the stack with the reverse operation, `POP { R5, R6, LR }`, leading back to (c). Note that the old values are not overwritten, but since `SP` has been moved back to its original position, they caller will ignore them.