6

Lab 4: The A32 Calling Convention

In this assignment, we will explore the A32 calling convention. A32 dictates how a C compiler running on a 32-bit ARM Linux computer should organize its memory—particularly its call stack—to ensure interoperability.

Every C implementation on every platform is expected to adhere to a calling convention. Although this lab is specific to Raspberry Pi and similar computers, what you learn is largely transferrable other platforms like Intel x86 and AMD64 with minor changes. Furthermore, many programming languages—like Java, Python, and the .NET languages—go to great lengths to interoperate with C code, which is the lingua franca[1] of library code. So even if you never touch C code again in your life, learning about calling conventions will give you deep insight into other languages and serve you well in the future.

[1] The term *lingua franca* literally refers to the "language of the Franks," a language that was widely spoken in the Mediterranean throughout the middle ages. At the time, the people of Western Europe were referred to as "the Franks." The language itself was a *pidgin* comprised of Italian, Greek, Slavic, Arabic, and Turkish. The term has since come to refer to any language regularly used for communication between people who do not share a native language, like English, Hindi, and Spanish.

6.1 *Learning Goals*

In this lab, you will learn:

- how to preprocess C macros to create expanded C code;
- how to generate an assembly listing from expanded C source code;
- how to generate object code from assembly; and
- how to link objects to produce an executable.

You will also learn:

- how C generates machine code to maintain the A32 call stack;
- how arguments are passed during function calls; and finally,
- how values are returned from a function call.

6.2 *Requirements*

Collaboration. This is an ungraded assignment. You are encouraged to work with a partner.

Platform. This assignment must be completed on your Raspberry Pi, as it is specific to the ARMv6 architecture, the Linux operating system, and the C programming language.

6.3 *Starter Code*

Type the following programs into a text editor. We start with a `Makefile`.

```
.PHONY: all clean
all: lab4

lab4_expanded.c: lab4.c
	cpp lab4.c -o lab4_expanded.c

lab4_expanded.s: lab4_expanded.c
	/usr/lib/gcc/arm-linux-gnueabihf/8/cc1 lab4_expanded.c

lab4.o: lab4_expanded.s
	as lab4_expanded.s -o lab4.o

lab4: lab4.o
	ld \
	  -dynamic-linker=/lib/ld-linux-armhf.so.3 \
	  /usr/lib/gcc/arm-linux-gnueabihf/8/../../../arm-linux-gnueabihf/crti.o \
	  /usr/lib/gcc/arm-linux-gnueabihf/8/crtbegin.o \
	  /usr/lib/gcc/arm-linux-gnueabihf/8/crtend.o \
	  /usr/lib/gcc/arm-linux-gnueabihf/8/../../../arm-linux-gnueabihf/crtn.o \
	  /usr/lib/gcc/arm-linux-gnueabihf/8/../../../arm-linux-gnueabihf/crt1.o \
	  lab4.o \
	  -o lab4 \
	  -lc

clean:
	@rm -f lab4_expanded.c
	@rm -f *.s
	@rm -f *.o
	@rm -f lab4
```

The above `Makefile` performs all the same steps that occur when simply running `gcc -o lab4 lab4.c`. Separating them out makes it easy to inspect the outputs of all the steps.

You will also need the following C program, called `lab4.c`.

```
#include <stdio.h>

#define FORMAT_STRING "%s"
#define MESSAGE       "Hello world!\n"

int main() {
  printf(FORMAT_STRING, MESSAGE);
  return 0;
}
```

6.4 *Compilation*

When you ask a C compiler like `gcc` or `clang` to compile your program, it works in at least four distinct phases. Those phases are *preprocessing*, *translation*, *assembly*, and *linking*.

Preprocessor. The C preprocessor converts the C prepreprocessor *macros* into C code, inserting them into a programmer's C program. Macros are widely used to save typing, to name symbolic constants, and to make code more portable across platforms.

Translator. C code must be converted into assembly opcodes for the target platform. Assembly is human readable output. Although a C compiler could just as easily produce machine code from C source code—because there is a 1:1 correspondence between assembly code and machine code—virtually all compilers keep this step separate. This allows compilers for different languages to reuse assembler programs, which are often created by hardware vendors themselves.

Assembler. Assembler code is then converted into machine code by the assembler. At this stage, because of separate compilation, each assembled machine code object is not yet executable, which is why it is called *object code*.

Linker. The linker produces a single binary executable file from multiple object code files, linking a user's code and libraries together, along with other necessary libraries like the C runtime. C runtime files are usually named `crt*.o`. These small snippets of the C language are written in assembly language and are usually assembled ahead of time and placed in a common location by an operating system's designers. The C runtime tells the operating system how to implement the calling convention, among other low-level details.

6.5 *Part 1: What does each step do?*

With your partner, answer the following questions.

1. What happens when you run `make lab4_expanded.c`? Use a text editor or `less` to examine the output. In particular, what happens to the symbols `#include <stdio.h>`, `FORMAT_STRING`, and `MESSAGE`? What purpose do you think `cpp` serves?

2. What happens when you run `make lab4_expanded.s`? Use a text editor or `less` to examine the output. What purpose does does `cc1` serve?

3. What happens when you run `make lab4.o`?

   (a) Use `file` to learn what kind of file `lab4.o` is:

   ```
   $ file lab4.o
   ```

   (b) The previous step should suggest why we can't just view `lab4.o` in a text editor. Use `hexdump` to view `lab4.o` in human-readable form.

   ```
   $ hexdump -C lab4.o
   ```

   (c) When examining *object code*, we can use the `objdump` utility to recover opcodes from machine code. The output of this tool should remind you of the output of another tool. Which one was it?

   ```
   $ objdump -D lab4.o
   ```

4. What happens when you run `make lab4`? Use `file` again and compare `lab4` with `lab4.o`. What's the difference? Try running `objdump` on `lab4`. You should see a lot of extra output. In general, what function do you think that extra output performs?

6.6 *Part 2: Simulate a program on paper*

Modify your `Makefile` to build the following program, `doesnothing.c`:

```
void foo() {}

int main() {
  foo();
  return 0;
}
```

Generate an assembly listing for this program, and then simulate this program on paper. You will need to refer to the *Appendix A: ARM Reference* handout. You should draw all 12 steps made by this program.

Note that certain opcodes can be created by multiple instruction mnemonics. For example, `push`ing one register to the call stack is the same as copying that same register to memory using a "side-effecting store." For this lab, refer to the following simplifications.

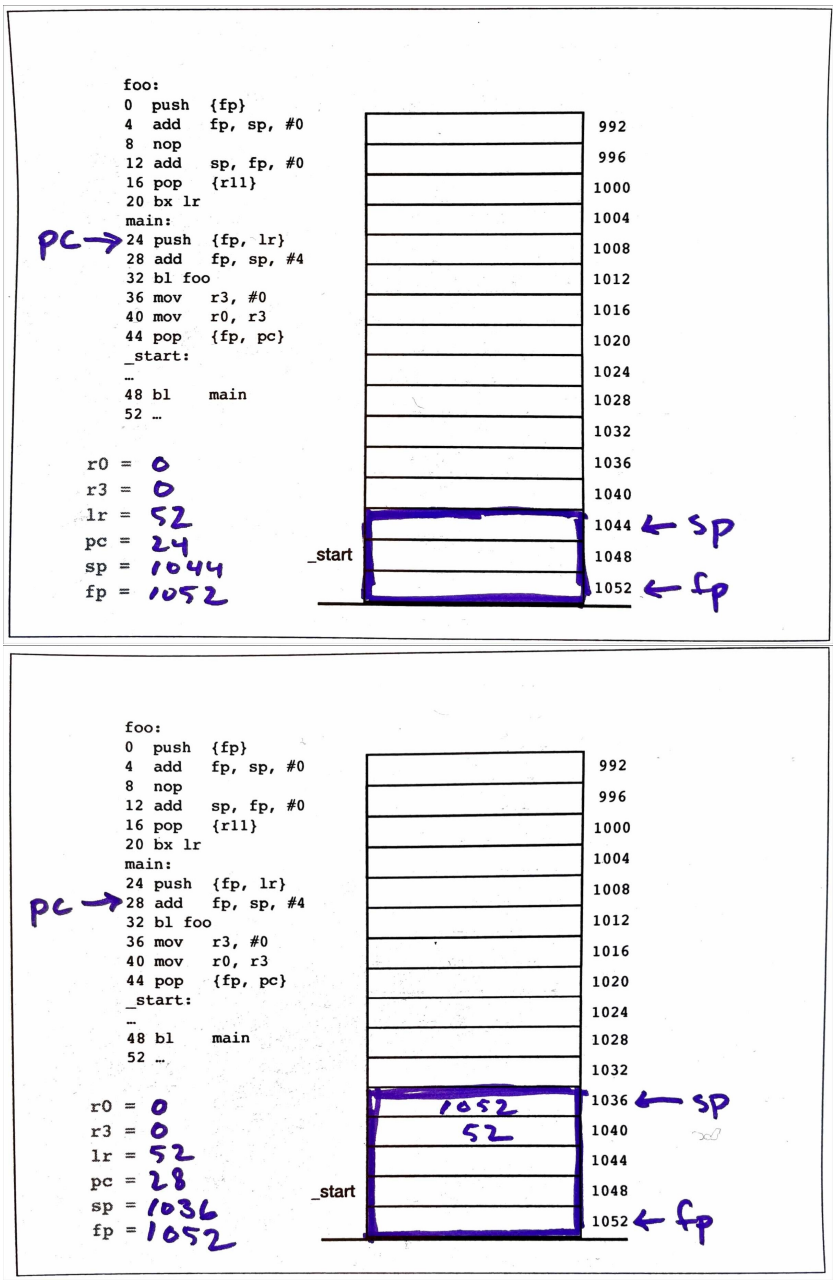
- `str fp, [sp, #-4]!` is equivalent to the simpler `push {fp, lr}`.
- `ldr fp, [sp], #4` is equivalent to the simpler `pop {fp}`.

Note that `objdump` usually prints the simpler mnemonic when there are multiple possibilities.

When executing each instruction, be sure to follow these rules:

1. Put a *frame* (a box) around the values between `sp` and `fp`, inclusive.
2. Once it is clear that a frame belongs to a different function, write that different function's name to the left of the frame. E.g., the first one is labeled `_start`.
3. Every instruction adds 4 to the `pc` except the branch instructions, `b`, `bl`, `bx`, etc.
4. When updating `pc`, `sp`, and `fp`, be sure to draw in an arrow representing the location that it points into the instruction buffer or the call stack, respectively.

To start you off, I've drawn the first two steps here.



6.7 *Part 3: Did you get it right?*

In this part, we are going to verify that we got the steps correct using `gdb`. You may have used `gdb` before to debug C code. Here, we are going to

use gdb to debug assembly code.

Note that, for simplicity, I changed the addresses of instructions and the call stack in the paper example above. Since you are now going to run this program for real, those addresses will be different. In other words, although things will be at different locations, the basic stack structure should remain the same.

1. Start gdb.

```
$ gdbtui doesnothing
```

2. We are going to set a breakpoint at the start of `main`. One funny thing about gdb is that it sets breakpoints after the end of the *function preamble*, which is the set of instructions that set up the stack frame for the callee. We will first set our breakpoint at `main` so that we can find `main`'s address, then back up and set the breakpoint at the start of `instead`.

```
(gdb) b main
```

3. Start the program running.

```
(gdb) r
```

   It should break at `main+8`.

4. After starting gdb, switch it into assembly mode.

```
(gdb) layout asm
```

5. Find the start of `main`. On my computer, it is `0x103e8`.

6. Now restart the program. The easiest way to do this without confusing gdb is to quit and start over.

```
(gdb) quit
A debugging session is active.

        Inferior 1 [process 18345] will be killed.

Quit anyway? (y or n) y
$ gdbtui doesnothing
(gdb) b *0x103e8
Breakpoint 1 at 0x103e8
(gdb) r
Starting program: doesnothing

Breakpoint 1, 0x000103e8 in main ()
(gdb) layout asm
```

   Note that you have to use the `*` above to let gdb know that you mean the *location* `0x103e8` and not the *function* named `0x103e8`.

7. You can inspect the processor's registers with the following command:

```
(gdb) info registers
```

8. You can print a specific register, like `sp`, with the following:

   ```
   (gdb) p/x $sp
   ```

9. You can also print the stack "as an array" using `gdb`'s *artificial array* syntax. For example, to print the three words between `sp` and `fp`

   ```
   (gdb) p/x *0xbefff538@3
   $2 = {0xb6fb7000, 0xbefff684, 0x1}
   ```

   Observe that in the above, I used the raw address stored in `$sp`. If you want, you can use `$sp` instead, but you need to tell `gdb` how it should interpret the pointer. In other words, what kind of pointer is it? For example, this following does not work.

   ```
   (gdb) p/x *$sp@3
   Attempt to dereference a generic pointer.
   ```

   But this does.

   ```
   (gdb) p/x *(int*)$sp@3
   $3 = {0xb6fb7000, 0xbefff684, 0x1}
   ```

10. Finally, you can *step* an instruction using the `si` command. To step to the next instruction *after a function call*, use `ni` instead. `ni` is useful because we sometimes don't want to step inside certain functions, like `printf`.

6.8 *Part 4: Where are the following sections?*

With your partner, identify which sections of the code correspond to the following purposes.

- *Function preamble*: sets up the call stack for the callee.
- *Function epilogue*: restores the call stack in order to return control to the caller.
- *Transfer of control*: causes the program to jump to a different sequence of instructions.
- *Preparing return value*: puts the return value in a standard location, usually `r0`.
- *Function body*: the section of code that performs the function's purpose.

6.9 *Part 5: Modify the program*

In this last part, change the program so that `main` passes an argument to `foo` and `foo` returns something. Start simply. Now, use the skills you just learned in this lab to observe how your program passes arguments. Feel free to experiment with this. For example, recall that at some point, C will *spill* extra arguments to the call stack instead of passing them through registers. Can you observe that happening?