Exploring AArch64 assembler – Chapter 2

Oct 8, 2016 • Roger Ferrer Ibáñez • AArch64

In the first installment of this series we did a very first simple program. In this chapter we will continue learning a bit more about AArch64.

Registers

Computers only work with binary data, so programs are encoded in what it is called machine code. But writing machine code is very unwieldy, so instead assembly language is used. In assembly we can specify instructions (and their operands) and the data of the program. Instructions tell the computer what to do (so they have a meaning).

The CPU is the part of the computer that executes programs. The instructions of a CPU that implements the AArch64 architecture, can only work in data that is found inside the CPU. The place where this data is located is called the *registers*. Any data that is not in registers and must be manipulated must be loaded first in the registers. Usually the loaded data will come from the memory but it can come from peripherals. As a way to communicate with the outer world, data can also be taken out of the registers to memory or peripherals.

In AArch64 there are 31 general-purpose registers. They are called general-purpose because they can hold any kind of data. In general they hold only integer or addresses (we will talk about addresses in a later chapter) but anything that can be encoded in 64 bits can be stored in a register. These 31 registers are called `x0`, `x1`, ..., `x30`. You may be wondering why 31 and not 32, which fits as a more natural power of 2 value. The reason is that what would be the *x31* is actually called `xzr` and means the *Zero Register*. It is a very special register with limited usage. Later we will see some examples on how to use this register. In general all registers can be used for any purpose, but in a later chapter we will see that there are some conventions on how to use them.

The AArch64 architecture defines more registers but they have more specific purposes and we will unveil them in later chapters.

While working with 64-bit wide registers could be enough, this would imply that all the operations happen in a 64-bit domain. Many times we do not need so many bits, in fact most programs have enough with 32-bit data (or even less). In order to provide 32-bit processing, it is possible to access the

lower 32-bit of a x*n* register using the name w*n*. So the lower 32-bit of register `x6` is `w6`. It is not possible to name the upper 32-bits. Register `xzr` has an equivalent 32-bit name called `wzr`.

This is the reason why our program in the first chapter was just.

```
mov w0, #2           // w0 ← 2
```

In C the return value of `main` is an `int` value. Technically C does not specify what is the specific width in bits of a `int` value (it just states some minimal ranges of values it has to be able to represent), but for economical reasons (given that `int` is the most used type in C) almost all 64-bit environments (including AArch64 and x86-64) make `int` a 32-bit integer type.

Working with data in registers

Almost all instructions in AArch64 have three operands. A destination register and two source registers. For instance, we can store in register `w5` the result of adding registers `w3`, and `w4` doing:

```
add w5, w3, w4       // w5 ← w3 + w4
```

similarly

```
add x5, x3, x4       // x5 ← x3 + x4
```

but note that in general we cannot name w*n* and x*n* registers in the same operation.

```
add w5, w3, x4
```

will fail with a message suggesting valid alternatives

```
add.s:6: Error: operand mismatch -- `add w5,w3,x4'
add.s:6: Info:    did you mean this?
add.s:6: Info:          add w5,w3,w4
add.s:6: Info:    other valid variant(s):
add.s:6: Info:          add x5,x3,x4
```

The zero register

The zero register `zr` (or `wzr`) is only useful as a source register. It does not represent a real register, it simply is a way to say «assume a zero here as the value of the operand».

Move

There are several exceptions to the one destination register and two source registers schema mentioned above. A notable one is the `mov` instruction. It takes a single source register.

```
mov w0, w1    // w0 ← w1
```

Note that this is a convenience instruction and it can be implemented using other instructions. A way could be adding the source register to zero. An instruction that would achieve the same as the mov above could be:

```
add w0, w1, wzr   // w0 ← w1 + 0
```

Actually in AArch64 `mov` is implemented using `orr` that is an instruction that performs a bitwise or operation using as the first source operand `wzr`.

Immediates

If source operands of instructions were restricted to registers, it would be impossible to load initial values to registers. This is the reason why some instructions allow what is called *immediates*. An immediate is an integer that is encoded in the instruction itself. This means that not any value will be possible to encode using an immediate, but fortunately many will. The ranges of allowed values of immediates depends on the instruction but many of them allow numbers in the range [-4096, 4095] (i.e. 12-bit). Due to the encoding used, any number in that range multiplied by 2^{12} (4096) is also allowed as an immediate. For instance 12288 and 16384 can be used as immediates as well (but not any other number inbetween). Immediates are represented in the assembler syntax preceding them with a `#`.

```
mov w0, #2      // w0 ← 2
mov w1, #-2     // w1 ← -2
```

Because immediates are encoded in the instruction itself and the space constraints mentioned above, only one immediate is usually allowed. It may vary depend instructions but in general the second source operand is allowed to be an immediate.

```
add w0, w1, #2   // w0 ← w1 + 2
add w0, w1, #-2   // w0 ← w1 + (-2)
```

These are not allowed:

```
add w0, #1, w1   // ERROR: second operand should be an integer register
add w0, #1, #2   // ERROR: second operand should be an integer register.
                 // This case is actually better expressed as
                 //    mov w0, #3
```

32-bit registers as destination

When the destination register of an instruction is a 32-bit register, the upper 32-bits are set to zero. They are not preserved.

A a silly example

At this point we cannot do much things yet but we can play a bit with our program. The number of arguments to our program is found in w0 when the program starts. Let's just return this same number plus 1.

```
// test.s
.text
.globl main

main:
  add w0, w0, #1   // w0 ← w0 + 1
  ret              // return from main
```

```
$ aarch64-linux-gnu-gcc -c test.s
$ aarch64-linux-gnu-gcc -o test test.o
$ ./test ; echo $?
2
$ ./test foo ; echo $?
3
$ ./test foo bar ; echo $?
4
```

Yay! If you wonder why the first case returns 2 instead of 1, it is because in UNIX the `main` function of a C program always receives a first parameter with the name of the program executed.

That's all for today.

« How (not) to write a C++ front end – Part 2 Exploring AArch64 assembler – Chapter 3 »

Powered by Jekyll. Theme based on whiteglass
Subscribe via RSS