Section 1 / More About `ldr`

Overview

In this chapter we examine the difference between loading the address of (a pointer to) a data label versus loading the data at the label. Both use the `ldr` instruction however, the assembler (on Linux) actually does some trickery behind the scenes to accomplish the loads.

Note - this chapter describes `ldr` from the perspective of Linux. For a brief discussion of how Apple Mac OS avoids using `ldr` to load the address of labels, please see here and below.

Length of Instructions

All AARCH64 instructions are 4 bytes in width.

Length of Pointers

All AARCH64 pointers are 8 bytes in width†.

†While this is technically true, typically only the lower 39, 42 or 48 bits of addresses in Linux systems are used - i.e. the virtual address space of an ARM Linux process is smaller than 64 bits. The upper bits are set to zero when considering the address as an 8-byte value.

How to Specify an Address Too Big to Fit in an Instruction?

The title of this section sets the table for the need for trickery. All labels refer to addresses. Addresses are 8 bytes† in width but all instructions are 4 bytes in width. Clearly, we cannot fit the full address of a label in an instruction.

Some ISAs (not ARM) have variable length instructions. The instruction may be four bytes wide but it tells the CPU that the next eight bytes are an operand of the instruction. Thus the true instruction width is 12 bytes. This is not true of the ARM ISA.

All instructions are 4 bytes wide. All of them.

"`ldr` x_register, =label" is a Pseudo Instruction

When you assemble an instruction looking like:

```
ldr     x1, =label
```

the assembler puts the address of the label into a special region of memory called a "literal pool." What matters is this region of memory is placed immediately after (therefore nearby) your code.

Then, the assembler computes the difference between the address of the current instruction (the `ldr` itself) and the address of the data in the literal pool made from the labeled data.

The assembler generates a different `ldr` instruction which uses the difference (or offset) of the data relative to the program counter (`pc`). The `pc` is non-other the address of the current instruction.

Because the literal pool for your code is located nearby your code, the offset from the current instruction to the data in the pool is a relatively **small** number. Small enough, to fit inside a four byte `ldr` instruction.

```
ldr    x1, [pc, offset to data in literal pool]
```

Example Program for Demonstrating Use of Literal Pool

Here is a sample program demonstrating the difference between:

```
ldr    x1, =q
```

and

```
ldr     x1, q
```

Note the difference is that the first has an `=` sign before the label and the second does not.

Also note, that when `line 15` is executed, the program will **crash**.

```
        .global     main                                                  // 1
        .text                                                             // 2
        .align      2                                                     // 3
                                                                          // 4
main:   str         x30, [sp, -16]!                                       // 5
                                                                          // 6
        ldr         x0, =fmt                // Loads the address of fmt   // 7
        ldr         x1, =q                  // Loads the address of q     // 8
        ldr         x2, [x1]                // Loads the value at q       // 9
        bl          printf                  // Calls printf()             // 10
                                                                          // 11
                                                                          // 12
        ldr         x0, =fmt                // Loads the address of fmt   // 13
        ldr         x1, q                   // Loads the VALUE at q       // 14
        ldr         x2, [x1]                // CRASH!                     // 15
        bl          printf                                                // 16
                                                                          // 17
        ldr         x30, [sp], 16                                         // 18
        mov         w0, wzr                                               // 19
        ret                                                               // 20
                                                                          // 21
```

```
        .data
q:      .quad       0x1122334455667788
fmt:    .asciz      "address: %p value: %lx\n"

        .end

```

Disassembling the binary machine code of the executable generated with the above source code will include:

```
00000000000007a0 <main>:
 7a0:   f81f0ffe   str  x30, [sp, #-16]!
 7a4:   58000160   ldr  x0, 7d0 <main+0x30>
 7a8:   58000181   ldr  x1, 7d8 <main+0x38>
 7ac:   f9400022   ldr  x2, [x1]
 7b0:   97ffffb4   bl   680 <printf@plt>
 7b4:   580000e0   ldr  x0, 7d0 <main+0x30>
 7b8:   580842c1   ldr  x1, 11010 <q>
 7bc:   f9400022   ldr  x2, [x1]
 7c0:   97ffffb0   bl   680 <printf@plt>
 7c4:   f84107fe   ldr  x30, [sp], #16
 7c8:   2a1f03e0   mov  w0, wzr
 7cc:   d65f03c0   ret
```

and

```
000000000011010 <q>:
   11010:   55667788
   11014:   11223344
```

Let's examine the second snippet first.

It says 000000000011010 <q>:. This means that what comes next is the data corresponding to what is labeled `q` in our source code. Notice the relocatable address of `11010`. We will explain "relocatable address" below.

Now, look at the disassembled code on the line beginning with `7b8`. It reads `ldr x1, 11010`. So the disassembled executable is saying "go to address 11010 and fetch its contents" which are our `1122334455667788`.

This is not the whole story.

Relocation of Addresses When Executing

None of the addresses we have seen so far are the final addresses that will be used once the program is actually running. All addresses will be *relocated*.

One reason for this is a guard against malware. A technique called Address Space Layout Randomization (ASLR) prevents malware writers from being able to know ahead where to modify your executable in order to accomplish their nefarious purposes.

This image shows `gdb` in `layout regs` at the time our program is loaded.

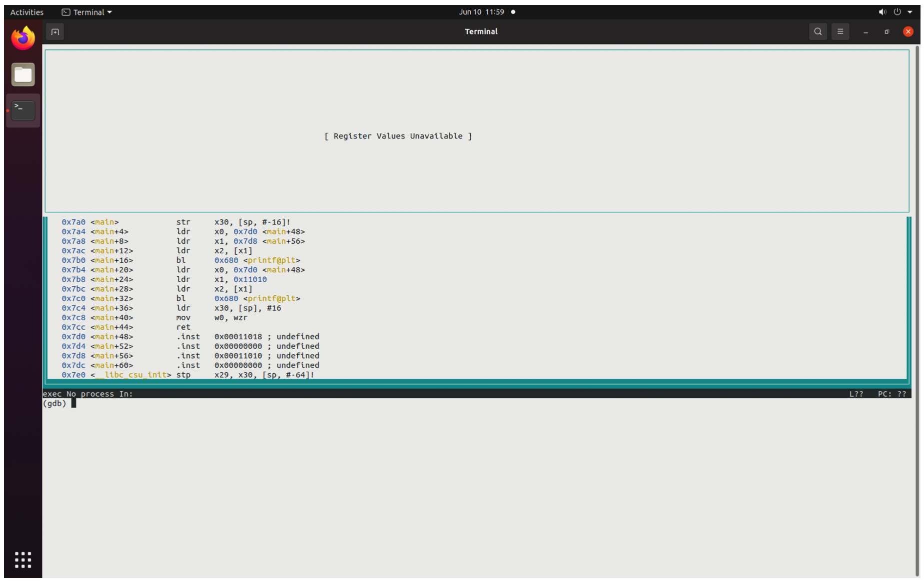

Figure 1: Prior to Launch

Notice that all of the addresses match the disassemblies given above. For example `main()` starts at `7a0`.

Now watch what happens the the program is actually launched (see Figure 2):

Suddenly all the address change to much larger values.

In fact, the addresses all seem to be six bytes long!

Why are these addresses only six bytes long when all pointers are 8 bytes long?

Sixty four bit ARM Linux kernels allocate 39, 42 or 48 bits for the size of a process's virtual address space. Notice 42 and 48 bit values require 6 bytes to hold them. A virtual address space is all of the addresses a process can generate / use. Further, all addresses used by processes are virtual addresses.

Kernels supporting other VA spaces, including 52 bit address spaces are possible but less common.

The salient point is that even six bytes is far too large to fit in a four byte instruction. GDB is masking the pseudo instruction and showing what the effective addresses are.**

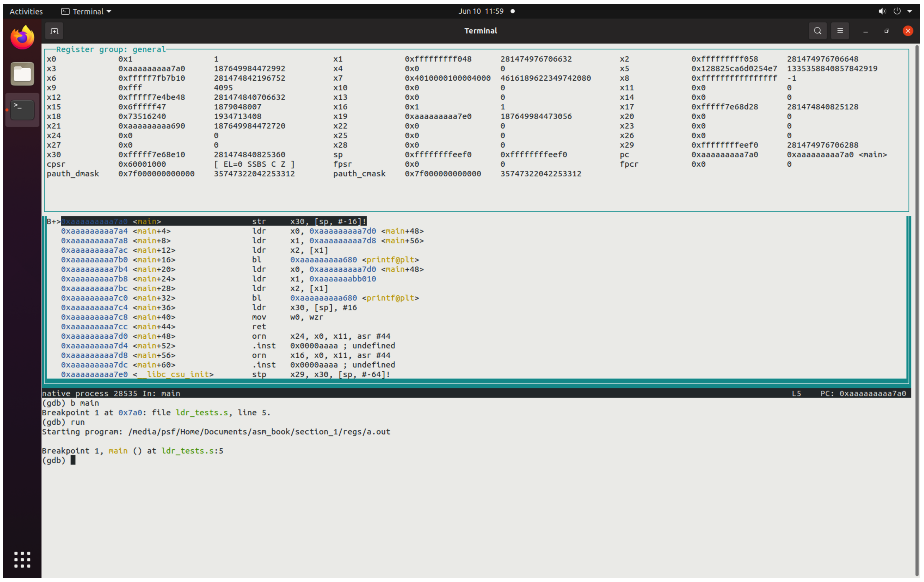

Figure 2: After breakpoint and launch

Now lets step forward to see the results of the first `ldr` of the `printf()` template / format string into `x0`. See Figure 3.

There is a pointer in `x0` ending in `b018`. Notice this is **NOT** the value encoded in the instruction ending in `a7d0`. This is our only indirect evidence that the instruction we wrote has been modified to use some calculated offset from the `pc`.

To finish, here is how we confirm `x0` is indeed correct. See Figure 4.

Notice down below the `x/s $x0` prints the value in memory corresponding to the address contained in `x0`.

Finally see Figure 5.

At the outset of this discussion we said that this program will crash on source code `line 15`. See if you can work out why. Take a moment before reading further.

Now that you have a hypothesis in mind, take a look at this screenshot showing the state of `x1` after this instruction: `ldr x1, q` is executed. See Figure 6.

Notice that what is in `x1` this time looks very different from the previous attempt at printing. Notice still more that the value now in `x1` is the value of `q`, not its address.

Naturally, the next instruction which tries to dereference the value of `q` rather than its address, causes a crash. See Figure 7.

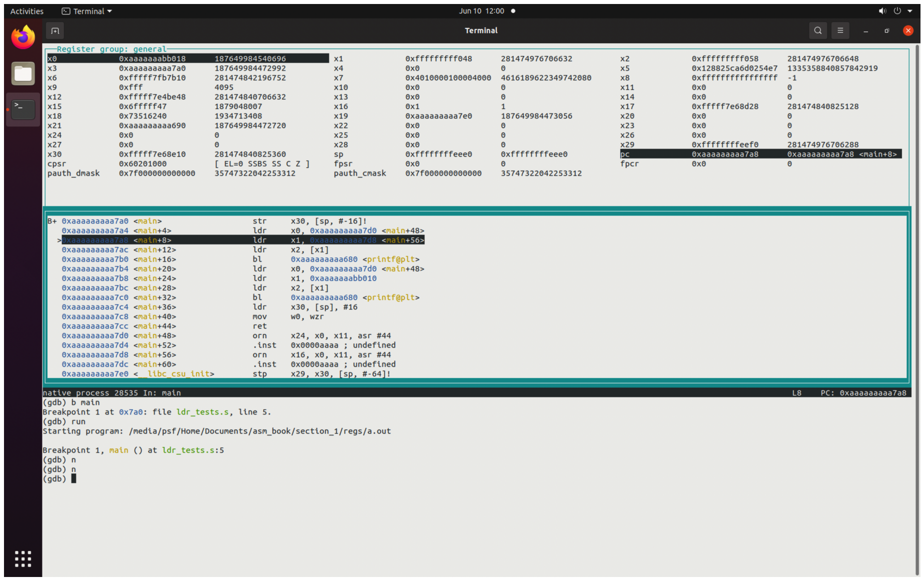

Figure 3: Results of first ldr

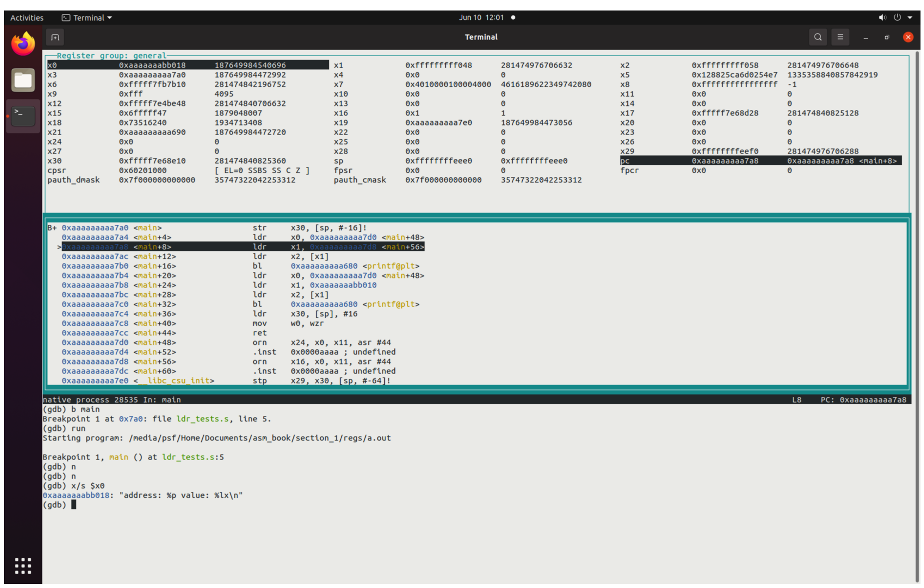

Figure 4: Confirming x0

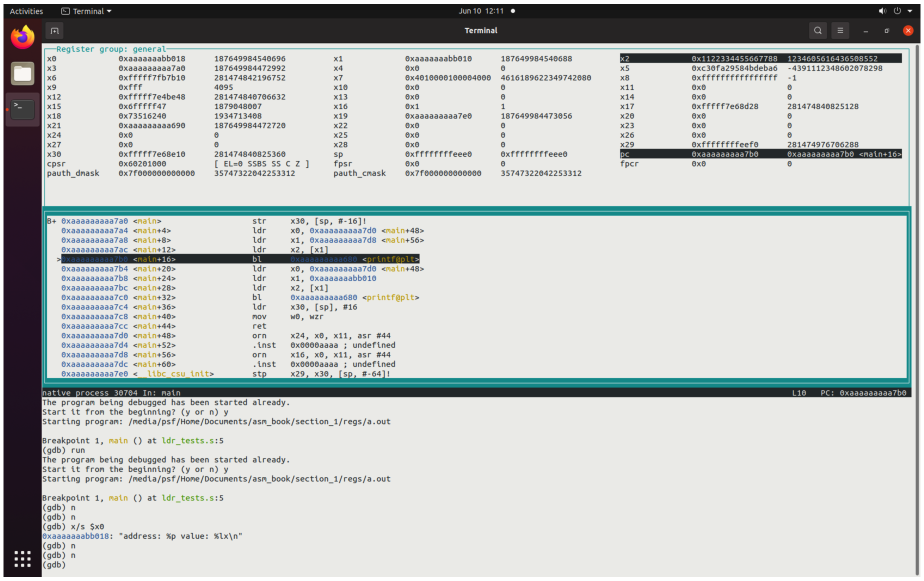

Figure 5: Confirming x2

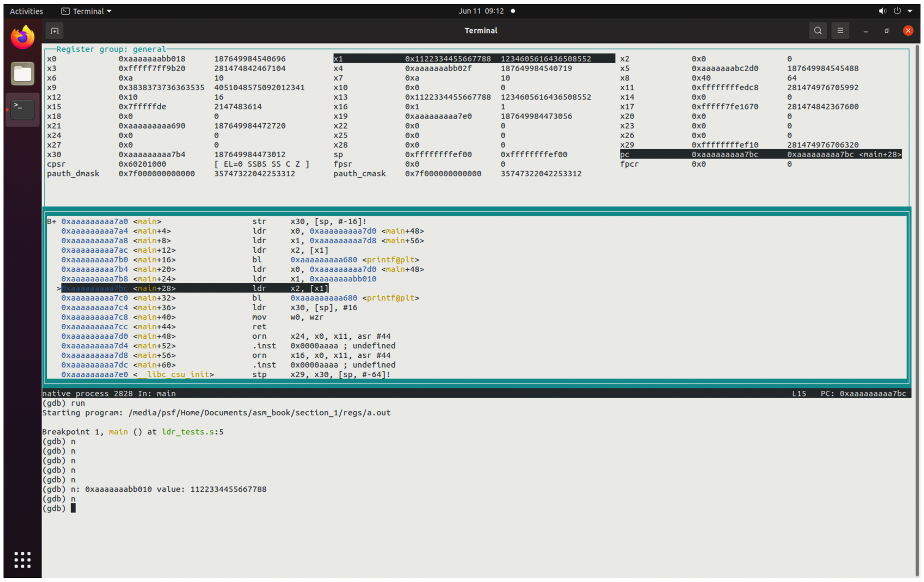

Figure 6: After bad load

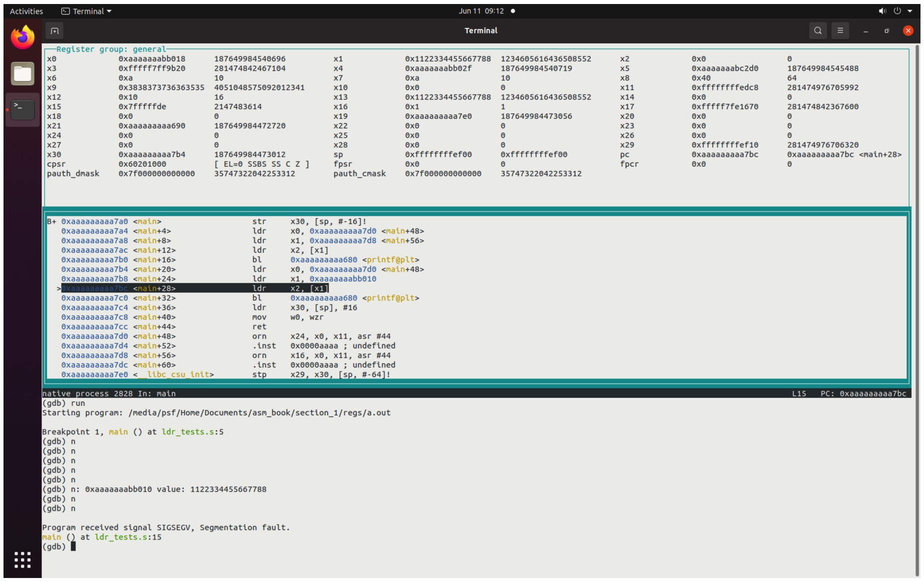

Figure 7: After crash

Summary

We have learned how the addresses corresponding to labels can be found. We also have learned how the contents of memory at those labels can be retrieved.

Instruction	Meaning
ldr r, =label	Load the address of the label into r
ldr r, label	Load the value found at the label into r

In both cases, the assembler will likely do some magical translation of your simple `ldr` instruction into something involving offsets so that the resulting offset can fit into an instruction where the full address cannot.

To store a value back to memory at the address given by a label, the address corresponding to the label will have first been loaded as is described above. Then, once the address is in a register, an `str` instruction can be used to properly locate the values to be written.

Questions

To be written.

Apple Silicon

You've seen that under Linux, there is a pseudo-instruction which hides some trickery:

```
ldr     x0, =label
```

This works if `label` is +/- four mebibytes (as megabytes are now called) away from the `ldr` instruction.

A downside of this approach is that the literal pool, from which the address is loaded, resides in RAM. This means each of these `ldr` pseudo instructions incurs a memory reference.

Apple "thinks different." The above instruction will not pass the assembler on a Mac OS machine. Instead, Apple uses two techniques which can access labels no matter where they are *without incurring a reference to memory.*

Apple accomplishes this by splitting the loading of the address of a label into two instructions. The first causes the base address of the *page* on which the label resides to be loaded. The page number is calculated for you based upon the label. The second adds to the base address, the offset in the page at which the label can be found.

Both of these values are computed at build time and therefore do not need to reference memory. This is a good thing.

Here is how one would load the address of a label which is outside the source code module:

```
.macro  GLD_ADDR    xreg, label     // Get a global address
#if defined(__APPLE__)
        adrp        \xreg, _\label@GOTPAGE
        add         \xreg, \xreg, _\label@GOTPAGEOFF
#else
        ldr         \xreg, =\label
#endif
.endm
```

This is a macro from our Apple Linux Convergence Suite.

It shows how, on Apple systems, the higher bits of the address is loaded from the starting address of the page on which the symbol sits and then the remainder (the offset) is added in.

The `G` in `GLD_ADDR` stands for global.

If the label is defined in the same source code module:

```
.macro  LLD_ADDR xreg, label
#if defined(__APPLE__)
        adrp        \xreg, \label@PAGE
```

```
        add         \xreg, \xreg, \label@PAGEOFF
#else
        ldr         \xreg, =\label
#endif
.endm
```

The difference being `@PAGE` versus `@GOTPAGE`, etc.

The first `L` in `LLD_ADDR` stands for local.

Avoiding the Memory Reference on Linux

The technique Apple uses for loading the address of labels can be used on Linux as well so as to avoid the reference to memory (literal pool).

Suppose `s` is a locally defined symbol. Then:

```
adrp    x0, s
add     x0, x0, :lo12:s
```

duplicates the approach Apple uses.