Exploring AArch64 assembler – Chapter 5

Nov 13, 2016 • Roger Ferrer Ibáñez • AArch64

In this chapter we will see how we can access the memory in AArch64.

Memory

Random access memory, or simply, *memory* is an essential component of any architecture. Recall that the memory can be seen as an array of bytes each paired with a consecutive number called the address. In AArch64 the address is a 64-bit number (which does not mean all the bits are meaningful for addresses).

Algebra of addresses

Given that addresses are numbers we can operate them as such. Not all arithmetic operations make sense on addresses, though. A higher address can be subtracted to a lower address. The result is not an address but an offset. Offsets can be added to addresses to form other addresses. Many times we need to access elements of size `b` that are consecutive in memory, thus their addresses are consecutive as well. This means that it is a common operation to compute an address of the form `A + b*i` to compute the address of the `i`-th element.

The occurrence of these common address operations influences the instructions of the architecture as we will see below.

Load and store

Due to its RISC heritage, AArch64 instructions do not operate with memory. Only two specific instructions can: load and store. These two instructions have in their most basic incarnation two operands: a register and an address. The address is computed based on an addressing mode, that we will see below. A load retrieves a number of bytes from the computed address and puts them in the register. A store takes some bytes from a register and puts them in a number of memory locations identified by the address.

AArch64 supports a few load/store instructions, but for the purpose of this chapter we will consider only `ldr` (load) and `str` (store). Their syntax looks like this

```
ldr Xn, address-mode  // Xn ← 8 bytes of memory at address computed by address-mode
ldr Wn, address-mode  // Wn ← 4 bytes of memory at address computed by address-mode

str Xn, address-mode  // 8 bytes of memory at address computed by address-mode ← Xn
str Wn, address-mode  // 4 bytes of memory at address computed by address-mode ← Wn
```

AArch64 can be configured as big-endian or little-endian. This makes little difference in general but it determines where the 8/4 bytes we load/store go to/come from our register. We will assume a little-endian setting (which is the most usual). This means when doing a load/store of 8 bytes, the first byte corresponds to the lest significant byte of the register, the next byte goes to the next least significant byte, and so on. A big endian machine would work in the opposite way, the first byte would correspond to the most significant byte.

Addressing modes

An addressing mode is the process by which a load/store instruction will compute the address it will access. Instructions in AArch64 are encoded in 32-bit but we already mentioned that addresses are 64-bit. This means that the most basic addressing mode using an immediate will not be available. Some architectures are able to encode full addresses in their instructions. Programs in these architecture rarely do this since it may take a lot of space.

Base

The next simplest addressing mode we can consider is when, by some mechanism not yet discussed, we have an address in a `Xn` register. In this case the address calculation just uses the value in the register. This mode is called *base register* and the syntax for it is `[Xn]`. Only a 64-bit register can be used as base.

```
ldr W2, [X1]  // W2 ← *X1 (32-bit load)
ldr X2, [X1]  // X2 ← *X1 (64-bit load)
```

Base plus immediate offset

As mentioned above when discussing the algebra of addresses, we can add an offset to an address to form another address. The syntax is `[Xn, #offset]`. The offset may range from -256 to 255 for 32-bit and 64-bit accesses. Larger offsets are a bit more limited: for 32-bit it must be multiple of 4 in the range 0 to 16380, for 64-bit it must be a multiple of 8 in the range of 0 to 32760.

```
ldr W2, [X1, #4]        // W2 ← *(X1 + 4)   [32-bit load]
ldr W2, [X1, #-4]       // W2 ← *(X1 - 4)   [32-bit load]
ldr X2, [X1, #240]      // X2 ← *(X1 + 240) [64-bit load]
```

```
ldr X2, [X1, #400]      // X2 ← *(X1 + 400) [64-bit load]
// ldr X2, [X1, #404]   // Invalid offset, not multiple of 8!
// ldr X2, [X1, #-400]  // Invalid offset, must be positive!
// ldr X2, [X1, #32768] // Invalid offset, out of the range!
```

Base plus register offset

Being able to use an immediate offset is handy, but sometimes the offset cannot be encoded as an immediate or may be unknown before the program is run. For these cases a register can be used instead.

```
ldr W1, [X2, X3]  // W1 ← *(X2 + X3) [32-bit load]
ldr X1, [X2, X3]  // X1 ← *(X2 + X3) [64-bit load]
```

It is possible to scale the value of the offset register, using a shifting operator `lsl`. This way `lsl #n` multiplies the offset by 2^n.

```
ldr W1, [X2, X3, lsl #3] // W1 ← *(X2 + (X3 << 3)) [32-bit load]
                         // this is the same as
                         // W1 ← *(X2 + X3*8)      [32-bit load]
ldr X1, [X2, X3, lsl #3] // X1 ← *(X2 + (X3 << 3)) [64-bit load]
                         // this is the same as
                         // X1 ← *(X2 + X3*8)      [64-bit load]
```

In contrast to the base register, the offset register can also be a 32-bit register but then we are forced to specify how we extend the 32-bit value to a 64-bit value. We have to use the extending operators we saw in chapter 3. Given that the source is a 32-bit value only `sxtw` and `uxtw` are allowed.

```
ldr W1, [X2, W3, sxtw] // W1 ← *(X2 + ExtendSigned32To64(W3))    [32-bit load]
ldr W1, [X2, W3, uxtw] // W1 ← *(X2 + ExtendUnsigned32To64(W3))  [64-bit load]
```

As we already know, it is possible to combine an extending operator with a shift.

```
ldr W1, [X2, W3, sxtw #3] // W1 ← *(X2 + ExtendSigned32To64(W3 << 3)) [32-bit-load]
```

Indexing modes

Sometimes it may happen that we are reading consecutive positions of memory in a way that we only care about the current element being read. In the worst of the cases, we can always compute the address using arithmetic operations in a register and then using a base indexed mode. Or, a bit better, keep the first address in a register that will act as the base and then compute the offset. If we take the latter approach most of the time our offset will be updated with a relatively simple operation like an addition or maybe an addition plus a shift (due to a multiplication with a power of two). We can exploit the fact that most of the time the address computation will have that shape. In these cases we may want to use an indexing mode.

There are two kinds of indexing modes in AArch64: pre-indexing and post-indexing. In a pre-indexing mode, the base register is added to an offset to compute the address, and then this address is written back to the base register. In a post-indexing mode, the base register is used to compute the address, as usual, but at the end of the memory access the base register is updated with the address value added to a given offset.

These two modes look pretty similar in that they update the base register with an offset. They differ at the point where the offset is accounted: a pre-indexing mode accounts it before the access, a post-indexing mode accounts it after the access. The offset we can use must be in the range -256 to 255.

Pre-index

The syntax for pre-index access mode is `[Xn, #offset]!`, mind the `!` symbol, otherwise you would be descriving a base plus offset without any indexing. In practice this acts pretty much like base plus offset but the `!` reminds us of the side effect of updating the base register.

```
ldr X1, [X2, #4]! // X1 ← *(X2 + 4)
                  // X2 ← X2 + 4
```

Post-index

The syntax is `[Xn], #offset`. It would be nice if there were a `!` after `#offset` in the syntax to get a visual cue similar to pre-indexing.

```
ldr X1, [X2], #4  // X1 ← *X2
                  // X2 ← X2 + 4
```

Load a literal address

Global objects, like global variables or functions, have addresses that are constant. This means that it should be possible to load them as literals. But as we know in AArch64 is not possible to load directly from a literal. So we need to use a two-step approach (which is by the way very common in RISC architectures). First we need to tell the assembler to put the address of the global variable in some position close to the current instruction. Then we load the address into a register using a special form of the load instruction (called load immediate).

In most of our examples it will look like this

```
ldr Xn, addr_of_var // Xn ← &var
...
addr_of_var : .dword variable // This tells the assembler that
                              // we want here the address of var
                              // (This is not to be executed!)
```

Once we have the address of the variable loaded in a register, we can do a second load to load the amount of bytes we want.

```
ldr Xm, [Xn]  // Xm ← *Xn    [64-bit load]
ldr Wm, [Xn]  // Wm ← *Xn    [32-bit load]
```

Using 32-bit addresses

Using 64-bit addresses is correct but a bit wasteful. The reason is that many times our programs will not need more than 32-bits to encode all the addresses for code and data. The addresses of our global variables will always have the upper 32-bit as zero. So we may want to use 32-bit addresses instead.

```
ldr Wn, addr_of_var // Wn ← &var
...
addr_of_var : .word variable // This tells the assembler that
                             // we want here the address of var
                             // (This is not to be executed!)
                             // 32-bit address here
```

Recall that when writing a 32-bit value to a register, its upper 32-bits are cleared. So after the `ldr` above we can use `[Xn]` in a load or store without problems.

Global variables

As an example of today topic, we will load and store a couple of global variables. This program does not do anything useful.

Global variables are defined in the `.data` section. To do this, we just simply define their initial contents. If we want to define a 32-bit variable we will use the directive `.word`. If we want to define a 64-bit variable we use the directive `.dword`.

```
 1 // globalvar.s
 2 .data
 3
 4 .balign 8 // Align to 8 bytes
 5 .byte 1
 6 global_var64 : .dword 0x1234  // a 64-bit value of 0x1234
 7 // alternatively: .word 0x1234, 0x0
 8
 9 .balign 4 // Align to 4 bytes
10 .byte 1
11 global_var32 : .word 0x5678   // a 32-bit value of 0
```

AArch64 in Linux does not require most memory accesses to be aligned. But if they are they can be executed faster by the hardware. So we use the directive `.balign` to align each variable to the size of the data (in bytes).

Now we can load the variables. For this example we will add 1 to each one.

```
13 .text
14
15 .globl main
16 main :
17   ldr X0, address_of_global_var64 // X0 ← &global_var64
18   ldr X1, [X0]                   // X1 ← *X0
19   add X1, X1, #1                 // X1 ← X1 + 1
20   str X1, [X0]                   // *X0 ← X1
21
22   ldr X0, address_of_global_var32 // X0 ← &global_var32
23   ldr W1, [X0]                   // W1 ← *X0
24   add W1, W1, #1                 // W1 ← W1 + 1
25   str W1, [X0]                   // *X0 ← W1
26
27   mov W0, #0                     // W0 ← 0
28   ret                            // exit program
29 address_of_global_var64 : .dword global_var64
30 address_of_global_var32 : .dword global_var32
```

Using 32-bit addresses

As mentioned above, storing addresses of 64-bits to our variables is usually a bit wasteful. Here are the changes required to use 32-bit addresses instead.

```
13 .text
14
15 .globl main
16 main :
17   ldr W0, address_of_global_var64 // W0 ← &global_var64
18   ldr X1, [X0]                   // X1 ← *X0
19   add X1, X1, #1                 // X1 ← X1 + 1
20   str X1, [X0]                   // *X0 ← X1
21
22   ldr W0, address_of_global_var32 // W0 ← &global_var32
23   ldr W1, [X0]                   // W1 ← *X0
24   add W1, W1, #1                 // W1 ← W1 + 1
25   str W1, [X0]                   // *X0 ← W1
26
27   mov W0, #0                     // W0 ← 0
28   ret                            // exit program
29 address_of_global_var64 : .word global_var64 // note the usage of .word here
30 address_of_global_var32 : .word global_var32 // note the usage of .word here
```

Note that when using this it is necessary to use the flag `-static` when doing the final link. This will create a static file which is loaded straightforwardly to memory. By default the linker creates dynamic files which are loaded by the dynamic linker when they are run. The dynamic linker will

load the program at an address beyond 2^{32} rendering these addresses invalid. When using `.dword` the static linker ensures an annotation for the dynamic linker is emitted, so the latter can fix the 64-bit address at runtime.

There are other, slightly better ways, to get the address of global variables but this will be enough for now. Maybe in a later chapter we will revisit this.

This is all for today.

« ARM assembler in Raspberry Pi – Chapter 26 Exploring AArch64 assembler – Chapter 6 »

Powered by Jekyll. Theme based on whiteglass
Subscribe via RSS