Section 1 / Interlude - Load and Store

In this section we will review the `ldr` and `str` families of instructions.

Several example programs will be presented.

As has been explained previously, modern CPUs are so much faster than RAM that fewer and fewer instructions are designed to operate on RAM directly. Instead, values from RAM are loaded from RAM into registers where they are used and possibly modified. If modified and desirable, the changed value might be stored from a register back to RAM.

Loading Data From RAM into Registers

The instructions typically used to retrieve information from memory are `ldr` and `ldp`. The characters `ld` in these mnemonics bring to mind `load`. `ldr` is "load a register (from RAM)" while `ldp` is "load a pair of registers (from RAM)".

Both of these instructions possess many variations, only a few of which will be described here. Common to all variations of the `ldr` and `ldp` instructions are the notions of *where to fetch from* and *where to store what's been fetched*.

Like many AARCH64 instructions, the most basic form of the load instructions are read right to left as in:

```
    ldr     x0, [x1]
```

which means "go to the location in RAM specified by `x1` and load what's there into `x0`."

Similarly,

```
    ldp     x0, x1, [sp]
```

loads a *pair* of registers from RAM at the address specified by the stack pointer. Any `x` register could also have been used, the `sp` is shown here to demonstrate that it too can be used. In fact, loads relative to the stack pointer is how compilers implement local variables.

What goes inside the `[]` is always a pointer so must be a 64 bit wide entity such as any `x` register or the stack *pointer*.

Offsets

To facilitate dereferencing `structs` and for accessing `arrays`, an offset may be specified.

There are significant restrictions placed on offsets because (among other reasons) the entire instruction (including the encoding of the offset) must fit within the constant 4 byte width of all AARCH64 instructions.

Here is text from an ARM manual:

```
1) LDR Xt, [Xn|SP{, #pimm}] ; 64-bit general registers
2) LDR Xt, [Xn|SP], #simm ; 64-bit general registers, Post-index
3) LDR Xt, [Xn|SP, #simm]! ; 64-bit general registers, Pre-index
```

These say you can load an `x` register (for simplicity we have ignored `w` registers) by dereferencing another `x` register or the stack pointer (i.e. `[Xn|SP]`).

Line 1 says you can *optionally* specify an offset.

Lines 2 and 3 says you can specify a *change* to the dereferenced register either before the actual fetch or after.

Assume `ptr` is a pointer to a `long`:

- Line 2 can correspond to: `*(ptr++)`.
- Line 3 can correspond to: `*(++ptr)`.

Note this is for illustration only in that the `++` syntax in C and C++ increment by 1. In lines 2 and 3, `#simm` can have values other than 1 including negative values for decrements.

Also note that when used with the stack pointer `sp`, `#simm` must be a multiple of 16. That is, all modifications to the stack pointer must be in multiples of 16.

Concerning the restrictions placed on the offsets:

- `simm` can be in the range of -256 to 255 (10 byte signed value).
- `pimm` can be in the range of 0 to 32760 in multiples of 8.

`w` registers are used for `int`, `short` and `char`. When working with `int`, `simm` must be a multiple of 4. When working with `short`, `simm` must be even. See the next example.

Note that there is another set of registers for floating point values. These are pretty cool in that they support half precision floats, single precision floats, double precision floats and also have double double precision floats (16 bytes in length)! These super big registers are often used when executing SIMD instructions.

SIMD is *Single Instruction - Multiple Data*. For example, 4 single precision floats might be multiplied by a scalar in a single instruction. The SIMD instruction set that is / will be covered in this book is called NEON.

The AARCH64 ISA includes an even more exotic means of performing mass calculation called SVE. We will probably never cover AVE as no generally available processor implements it. This includes Apple Silicon.

I wonder how sad this makes certain engineers at ARM. After all, imagine if YOU threw an ISA and nobody came.

Examples

Loading (Storing) Various Sizes of Integers

Instruction	Meaning
`ldr    x0, [x1]`	Fetches a 64 bit value from the address specified by `x1` and places it in `x0`
`ldr    w0, [x1]`	Fetches a 32 bit value from the address specified by `x1` and places it in `w0`
`ldrh    w0, [x1]`	Fetches a 16 bit value from the address specified by `x1` and places it in `x0`
`ldrb    w0, [x1]`	Fetches an 8 bit value from the address specified by `x1` and places it in `x0`

Notice the following:

- Pointers and longs use `x` registers.
- All other integer sizes use `w` registers where the instruction itself specifies the size.

Array Indexing 1 - Wasteful

Consider this code to sum up the values in an array:

```
long Sum(long * values, long length)                     /* 1 */
{                                                        /* 2 */
    long sum = 0;                                        /* 3 */
    for (long i = 0; i < length; i++)                    /* 4 */
    {                                                    /* 5 */
        sum += values[i];                                /* 6 */
    }                                                    /* 7 */
    return sum;                                          /* 8 */
}                                                        /* 9 */
```

We're not going to translate this to assembly language. Instead, we will call out how inefficient this code is. Notice we're using the index variable `i` for nothing more than traipsing through the array. This is fantastically inefficient (in this case).

Array Indexing 2 - More Efficiently

Consider the following code that performs the same function:

```
long Sum(long * values, long length)                          /* 1 */
{                                                             /* 2 */
    long sum = 0;                                             /* 3 */
    long * end = values + length;                             /* 4 */
    while (values < end)                                      /* 5 */
    {                                                         /* 6 */
        sum += *(values++);                                   /* 7 */
    }                                                         /* 8 */
    return sum;                                               /* 9 */
}                                                             /* 10 */
```

Notice we don't use an index variable any longer. Instead, we use the pointer itself for both the dereferencing *and* to tell us when to stop the loop.

`values` begins as the address of the first `long` in the array. On `line 4` we leverage *address arithmetic* to determine where to stop. `end` gets the address of the `long` just beyond the end of the array. When we get there, we stop.

This approach, which avoids the overhead of a loop variable, works well in both `C` and `C++`. It is *similar in spirit* to this in `C++`:

```
    vector<Foo> foov;
    for (auto it = foov.begin(); it < foov.end(); it++)
```

Here is a hand translation of the above `C` code for function `Sum()`:

```
    .global Sum                                                 // 1
    .text                                                       // 2
    .align  4                                                   // 3

//  x0 is the pointer to data                                   // 5
//  x1 is the length and is reused as `end`                     // 6
//  x2 is the sum                                               // 7
//  x3 is the current dereferenced value                        // 8

Sum:                                                            // 10
    mov     x2, xzr                                             // 11
    add     x1, x0, x1, lsl 3                                   // 12
    b       2f                                                  // 13

1:  ldr     x3, [x0], 8                                         // 15
    add     x2, x2, x3                                          // 16
2:  cmp     x0, x1                                              // 17
    blt     1b                                                  // 18

    mov     x0, x2                                              // 20
    ret                                                         // 21
`
    .end                                                        // 23
```

Recall that `Sum(long * values, long length)` means that `x0` has the address of the first long in the array.

- We know it's an `x` register because it is an address.
- We know it is the `0` register because it is the first argument.

`x1` contains `length`.

- We know it is an `x` register because it is a `long`.
- We know it is the `1` register because it is the second argument.

`Line 11` shows the first use of a "zero register," in this case `xzr`. Reading from a zero register always returns zero. Writing to a zero register is ignored. There also exists `wzr` for other integer sizes.

`Line 12` is the first really interesting line. It implements `line 4` of the higher level language. That is:

```
long * end = values + length;
```

is implemented as:

```
add     x1, x0, x1, lsl 3
```

We are performing address arithmetic on `longs`. Each `long` is 8 bytes wide. `x1, lsl 3` means "before adding the value of `x1` to `x0`, multiply `x1` by 8." Eight is 2 raised to the power of 3. `lsl 3` means shift left by 3 bits ... shifting is a fast way of integer multiplication (and division) by powers of 2.

`Line 13` is the branch to the *bottom* of the loop where the decision code is written. We saw how this can save an instruction here.

`Line 15` is the `ldr` instruction which performs not only the load (dereference) but also the *post increment* of the pointer.

```
sum += *(values++);                                  /* 7 */
```

is implemented by both `lines 15` and `16` in the assembly language.

```
1:  ldr     x3, [x0], 8                              // 15
    add     x2, x2, x3                               // 16
```

`Line 17` compares the pointer to where we are now in the array to the address of just past the end of the array.

`Line 18` says that as long as `x0` (or "where we are now") is less than the end of the array (in `x1`), we keep looping.

`Line 20` copies the accumulated sum into `x0` where values returned from functions are expected to be found.

`Lines 5 through 8` are an example of what we call a "dictionary" that serves as a memory aid to remember which register is being used for what purpose.

Nothing introduces bugs faster then forgetting this information and using a register for a purpose other than that for which it was intended.

`Line 1` makes `Sum` available to the linker so that this function can be called from other source code files.

`Line 2` tells the assembler (and linker) that what follows is code. Code sections are marked as read / execute only so that self-modifying code is disallowed. Self-modifying code was really fun to write but really dangerous. We miss the ability to write such dangerous code. :(

`Line 4` isn't strictly necessary. All instructions in the ARM 64 bit ISA are 32 bits (4 bytes) in length. The ARM processor likes it when something is located at a multiple of its size. For example, the preferred alignment for a `long int` in a `struct` is at an address that is a multiple of 8. This will become apparent when we discuss `structs`.

Faster Memory Copy

This is a *heavily* contrived example. In reality it is a fun challenge to write an optimal general purpose `memcpy` function. Or, you can just use `memcpy`.

For the purposes of this discussion, ignore issues relating to alignment.

Suppose you needed to copy 16 bytes of memory from one place to another. You might do it like this:

```
void SillyCopy16(uint8_t * dest, uint8_t * src)
{
    for (int i = 0; i < 16; i++)
        *(dest++) = *(src++);
}
```

This is especially silly as why would you go through 16 loops when you could have simply:

```
void SillyCopy16(uint64_t * dest, uint64_t * src)
{
    *(dest++) = *(src++); // 3
    *dest = *src;         // 4
}
```

`Line 3` dereferences `src`, holds the value that's there and increments `src` by the size of a `long`. The assignment puts the value dereferenced from `src` into the location specified by `dest` and increments the pointer afterwards.

`Line 4` is simplified because this silly move is only two `longs` long. Since this is the second copy of 8 out of 16 bytes, we have no need to increment the pointers.

In assembly language, this could be written:

```
SillyCopy16:                // 1
    ldr    x2, [x0], 8      // 2
    str    x2, [x1], 8      // 3
    ldr    x2, [x0]         // 4
    str    x2, [x1]         // 5
    ret
```

`Lines` 2 and 3 increment `x0` and `x1` to the next `long` **after** dereferencing them.

Then again, what about the *pair* load and store instructions? Can these help? Yes!

```
SillyCopy16:
    ldp    x2, x3, [x0]
    stp    x2, x3, [x1]
    ret
```

As an interesting aside, remember the `q` registers? They are 16 bytes wide by themselves.

```
SillyCopy16:
    ldr    q2, [x0]
    str    q2, [x1]
    ret
```

Indexing Through An Array of `struct`

You should read the chapter on `struct` found here.

Here is a more elaborate case study. Given this:

```
#include <stdio.h>                                              /* 1 */

struct Person                                                   /* 3 */
{                                                               /* 4 */
    char * fname;                                               /* 5 */
    char * lname;                                               /* 6 */
    int age;                                                    /* 7 */
};                                                              /* 8 */

extern int rand();                                             /* 10 */
extern struct Person * FindOldestPerson(struct Person *, int);  /* 11 */

struct Person * OriginalFindOldestPerson(struct Person * people, int length) /* 13 */
{                                                              /* 14 */
    int oldest_age = 0;                                        /* 15 */
    struct Person * oldest_ptr = NULL;                         /* 16 */

    if (people)                                                /* 18 */
    {                                                           /* 19 */
        struct Person * end_ptr = people + length;              /* 20 */
        while (people < end_ptr)                                /* 21 */
        {                                                       /* 22 */
            if (people->age > oldest_age)                       /* 23 */
            {                                                   /* 24 */
                oldest_age = people->age;                       /* 25 */
                oldest_ptr = people;                            /* 26 */
            }                                                   /* 27 */
            people++;                                           /* 28 */
        }                                                       /* 29 */
    }                                                           /* 30 */
    return oldest_ptr;                                          /* 31 */
}                                                               /* 32 */

#define LENGTH  20                                              /* 34 */

int main()                                                      /* 36 */
{                                                               /* 37 */
    struct Person array[LENGTH];                                /* 38 */
    for (int i = 0; i < LENGTH; i++)                            /* 39 */
    {                                                           /* 40 */
        array[i].age = rand() % 5000;                           /* 41 */
    }                                                           /* 42 */
    struct Person * oldest = FindOldestPerson(array, LENGTH);   /* 43 */
    for (int i = 0; i < LENGTH; i++)                            /* 44 */
    {                                                           /* 45 */
        printf("%d", array[i].age);                             /* 46 */
        if (oldest == &array[i])                                /* 47 */
            printf("*");                                        /* 48 */
        printf("\n");                                           /* 49 */
    }                                                           /* 50 */
}                                                               /* 51 */
```

This program defines a `struct` called `Person`. See `line 3`.

It will create an array of these `structs` with length 20. See `line 38`.

It will initialize the `age` data member of each instance with a random value between 0 and 5000 (Biblical people maybe?). See `lines 39 to 42`. Note that the pointers to first name and last name are indeed left uninitialized because these values are unimportant to this demo. They serve only to move the location of the `age` member away from offset 0.

`Line 11` tells us that somewhere else, there is a function called `FindOldestPerson`. That function must have a `.global` specifying the same name so that the linker can reconcile the reference to `FindOldestPerson`.

`OriginalFindOldestPerson` takes a pointer to an instance of `struct Person`. Being a pointer, this can be used as a way of finding just one instance or, as in our case, an array of these `structs`.

The function finds the largest value in the `age` member using the expected algorithm. It initializes an `oldest_age` found so far with 0 and a pointer to that instance. It marches through the array at most `length` times. If it finds an instance with an age older than the oldest found so far, it updates both values.

Upon reaching the end of the array, it will return a pointer to the instance containing the oldest age. If there is a tie, it will return the first oldest instance.

`Line 18` is **defensive programming**. It ensures that no search is performed if the function is handed a null pointer.

`gcc` with `-O2` or `-O3` optimization rendered `OriginalFindOldestPerson()` into 18 lines of assembly language.

Here is an assembly language implementation.

This example is more “real world” in that it offers us the chance to work with `w` registers (`int`). It also demonstrates `csel` which is like the `C` and `C++ ternary operator`.

```
        .global FindOldestPerson                                   // 1
        .text                                                      // 2
        .align  2                                                  // 3
                                                                   // 4
//  x0  has struct Person * people                                 // 5
//      will be used for oldest_ptr as this is the return value    // 6
//  w1  has int length                                             // 7
//  w2  used for oldest_age                                        // 8
//  x3  used for Person *                                          // 9
//  x4  used for end_ptr                                           // 10
//  w5  used for scratch                                           // 11
                                                                   // 12
FindOldestPerson:                                                  // 13
        cbz     x0, 99f             // short circuit               // 14
        mov     w2, wzr             // initial oldest age is 0     // 15
        mov     x3, x0              // initialize loop pointer     // 16
        mov     x0, xzr             // initialize return value     // 17
        mov     w5, 24              // struct is 24 bytes wide     // 18
        smaddl  x4, w1, w5, x3      // initialize end_ptr          // 19
        b       10f                 // enter loop                  // 20
                                                                   // 21
1:      ldr     w5, [x3, p.age]     // fetch loop ptr -> age       // 22
        cmp     w2, w5              // compare to oldest_age       // 23
        csel    w2, w2, w5, gt      // update based on cmp         // 24
        csel    x0, x0, x3, gt      // update based on cmp         // 25
        add     x3, x3, 24          // increment loop ptr               // 26
10:     cmp     x3, x4              // has loop ptr reached end_ptr?    // 27
        blt     1b                  // no, not yet                      // 28
                                                                        // 29
99:     ret                                                             // 30
                                                                        // 31
        .data                                                           // 32
        .struct 0                                                       // 33
p.fn:   .skip   8                                                       // 34
p.ln:   .skip   8                                                       // 35
p.age:  .skip   4                                                       // 36
p.pad:  .skip   4                                                       // 37
                                                                        // 38
        .end                                                            // 39
```

Before we get to the explanation, permit us a small pat on the back. The above version, written by us humans, rendered `FindOldestPerson()` in 15 lines of actual code.

`Lines 5` through `11` are vitally important comments. You should always write comments like these as they will serve as your "dictionary" to help you keep track of what particular registers will be used for. Notice this is the second time we have suggested this. It isn't that we forgot that we suggested it above. Rather we suggest it a second time, and belabor the point, because it is *that* important.

`x0` begins as the pointer to `struct Person` being passed to us. `x0` is also used for returning values from a function so we'll copy `x0` to `x3` on `line 16`. This will save us an instruction later as we won't have to copy the intended return value back to `x0` prior to the `ret` on `line 30`.

`w1` is passed to us as the length of the array. It is in a `w` register because we defined it as an `int`. This is the first time you're seeing a `w` register in actual use.

`w2` will hold the oldest age found so far. It is a `w` register because we defined age as an `int`.

`x3` is described above under `x0`.

`x4` will be set to the address after the end of the array and will be used to stop our loop.

`w5` is used for scratch.

Recall that registers 0 through 7 are scratch registers and do not have to be backed up or restored.

`Line 14` is a combination compare AND branch instruction.

```
        cbz     x0, 99f
```

says “Check `x0`. If it is zero, branch forward to temporary label 99.” The `cbz` mnemonic means “compare and branch if zero.” There is also a `cbnz` instruction branching if not zero.

The `cbz` and `cbnz` instructions exist because testing against zero is so common.

Our choice of naming a temporary label `99` is a matter of personal *style*. We use `99` to indicate from where a function is going to exit. This is an aid to remembering and understanding the code.

The `cbz` instruction is the same as:

```
        cmp     x0, xzr
        beq     99f
```

`Line 14` implements `line 18` of the `C` code. It ensures we will handle being passed a null pointer as input. This is an example of **defensive programming**. Without this check, we would crash if handed a null pointer. Crashing is what experts call **Bad**.

The closing brace found on `line 30` of the `C` code is implemented on `line 30` of the assembly language code. A coincidence, surely.

`Line 15` establishes the oldest age found so far as being 0. It makes use of the `wzr` zero register. We use `w` because the destination is `w` register.

`Line 16` copies the base address of the array to `x3` from `x0`. The value arrives in `x0` because it is the first parameter to the function. It must be an `x` register because it is a pointer. We need a pointer to march through the array. `x0` serves double duty as holding the first parameter but also is the place where function return values are found.

We copy `x0` out to `x3` so that we can use `x0` to store a pointer to the array element representing the oldest person found so far. If we iterated over the array using `x0`, we would still a) need another `x` register to hold the pointer to the oldest person so far and b) have to copy this register to `x0` before we return anyway. Doing the marching through the array is a register *other* than `x0` saves us one instruction.

`Line 17` initializes `x0` after we’ve preserved its original value in `x3`.

`Line 18` puts the value of 24 into `w5`. This register is used for scratch or intermediate calculation purposes. We’re setting up the calculation which ends with the pointer to just beyond the end of the array. The size of the `struct Person` is 24 bytes (not 20). We considered allowing the assembler to compute this for us but chose instead to hard code the value.

There’s the beginning of a lesson here. Notice that the apparent length of `Person` is 20 bytes, being two pointers plus an int. But the actual length is 24 bytes. This is because the natural alignment of data is at addresses which are multiples of their width. The first data member of the struct is a pointer (8 bytes). Therefore, the alignment of the struct must be a multiple of 8. Four wasted bytes are added to the `struct` to make its length come out as a multiple of 8.

`Line 19` is a mouthful. The mnemonic `smaddl` means *signed multiply add long*. Here is the instruction:

```
smaddl  x4, w1, w5, x3      // initialize end_ptr          // 19
```

`w1` (the length) will be multiplied by `w5` (the size of each array member), added to `x3` (the base address of the array) and the result will be placed into `x4`. This assembly language instruction implements this in C:

```
struct Person * end_ptr = people + length;                  /* 20 */
```

The compiler itself knows the true length of a `Person` (it is 24). When doing "address arithmetic" the compiler automatically scales the computation by the actual size of the thing being calculated. In this case, `line 19` says:

"Take the value in length. Multiply it by the size of one `Person` (24). Then add this value to the address contained in `people`."

`Line 20` branches to the `while` loop's decision test. Putting the decision test of a loop at the loop's bottom rather than the top has previously been shown to save one instruction.

`Line 22` begins the main loop of this function. `w5` is loaded with the `int` found 16 bytes away from the address pointed to by `x3`. In this case, we allowed the assembler to compute the 16 for us - you can see this on `lines 33` through `37`. A `w` register is used because `age` is an `int`.

`Line 23` compares the current age to the largest age found so far. This is a key line in that the `cmp` sets *status bits* which are used by the next two, very cool, instructions.

`Line 24` and `25` both make use of the `csel` instruction. The mnemonic means "conditional select". The comparison **has already been made** (on `line 23`) setting the CPU's status bits recording if the comparison resulted in a less than zero, zero, or more than zero result.

`Lines 24` and `25` read:

```
csel    w2, w2, w5, gt      // update based on cmp         // 24
csel    x0, x0, x3, gt      // update based on cmp         // 25
```

These are identical to this:

```
w2 = (w5 > w2) ? w5 : w2;
x0 = (x5 > x2) ? x3 : x0;
```

Remember that the condition or status bits have already been set based upon whether or not the current age is greater than (or equal to) the oldest age found so far. Both of the `csel` instructions leverage the outcome of the comparison, done just once.

`csel`, like the `C` and `C++` ternary operator, is quite cool in that we get the results of an `if` statement without the overhead of branching instructions!

`Line 26` increments the loop pointer to the next array member or to the end of the array.

`Line 27` compares the new value of the loop pointer to the address coming after the array.

`Line 28` will branch to the next iteration of the loop if `x3` has not yet advanced as far as `x4` sitting past the end of the array.

`Line 30` is simply a `ret` without no other bookkeeping because the value we want to return has been sitting in `x0` all along! A reminder that we did not need to preserve the value of `x30`, for example, because this function makes no function calls. `x30`, our return address, remains safely unchanged.

What Did We Learn?

In the preceding example we saw:

- Use of `w` registers.
- Use of `cbz`, a special case of a compare and branch in one.
- `smaddl` for doing address arithmetic.
- `csel` for efficiently choosing one of two values like the C and C++ ternary operator.
- Use of a `struct`.
- Brief discussion of alignment within `structs`.

WOW!

Questions

TO COME LATER