Section 2 / Floating Point Literals

Recall that all AARCH64 instructions are 4 bytes long. Recall also that this means that there are constraints on what can be specified as a literal since the literal must be encoded into the 4 byte instruction. If the literal is too large, an assembler error will result.

Given that floating point values are always at least 4 bytes long themselves, using floating point literals is extremely constrained. For example:

```
1          fmov    d0, 1
2          fmov    d0, 1.1
```

`Line 1` will pass muster but `Line 2` will cause an error.

To load a `float`, you could translate the value to binary and do as the following:

```
1                   .text
2                   .global main
3                   .align    2
4
5          main:    str        x30, [sp, -16]!
6                   ldr        s0, =0x3fc00000
7                   fcvt       d0, s0
8                   ldr        x0, =fmt
9                   bl         printf
10                  ldr        x30, [sp], 16
11                  mov        w0, wzr
12                  ret
13
14                  .data
15         fmt:     .asciz    "%f\n"
16                  .end
```

The above code is found here.

`Line 6` puts the translated value of 1.5 into `s0` (since the value is a `float` it goes in an `s` register). The assembler performs some magic getting a 32 bit value seemingly fit into a 32 bit instruction. See below.

`Line 7` converts the single precision number into a double precision number for printing.

`printf()` only knows how to print double precision values. When you specify a `float`, it will convert it to a `double` before emitting it.

Translating `floats` and `doubles` by hand isn't a common practice for humans, though compilers are happy to do so.

Instead for us humans, the assembler directives `.float` and `.double` are used more frequently to specify `float` and `double` values putting them into RAM.

The following example prints an array of floats and doubles:

```
        .global main
        .text
        .align  2

counter .req    x20
dptr    .req    x21
fptr    .req    x22
        .equ    max, 4

main:   stp     counter, x30, [sp, -16]!
        stp     dptr, fptr, [sp, -16]!
        ldr     dptr, =d
        ldr     fptr, =f
        mov     counter, xzr

1:      cmp     counter, max
        beq     2f

        ldr     d0, [dptr, counter, lsl 3]
        ldr     s1, [fptr, counter, lsl 2]
        fcvt    d1, s1
        ldr     x0, =fmt
        add     counter, counter, 1
        mov     x1, counter
        bl      printf
        b       1b

2:      ldp     dptr, fptr, [sp], 16
        ldp     counter, x30, [sp], 16
        mov     w0, wzr
        ret

        .data
fmt:    .asciz  "%d %f %f\n"
d:      .double 1.111111, 2.222222, 3.333333, 4.444444
f:      .float  1.111111, 2.222222, 3.333333, 4.444444

        .end
```

The above code is found here.

A number of interesting things in this source code:

- We use `.req` to give symbolic names to various registers. This can help you in remembering which register is being used for what purpose.

- We use `.equ` to encode a small integer literal value to give it a symbolic name, eliminating the use of a "magic number."
- `Lines 19` and `20` use address arithmetic to march through an array of doubles (8 bytes each) and an array of floats (4 bytes each).

`Line 19` is equivalent to:

```
//   ldr     d0, [dptr, counter, lsl 3]
d0 = dptr[counter];
```

`counter` is multiplied by 8 then added to `dptr`.

`Line 20` is equivalent to:

`counter` is multiplied by 4 then added to `fptr`.

```
//   ldr     s1, [fptr, counter, lsl 2]
s1 = fptr[counter];
```

Cool huh?

Fitting 32 bits into a 32 bit bag

AARCH64 instructions are 32 bits in width. Yet, `line 6` from this program reads:

```
        ldr         s0, =0x3fc00000                                  // 6
```

This appears to show a 32 bit constant being held in an instruction that itself is 32 bits wide. Well, the Assembler does some magic. Let's see what that magic is.

Build the program with the `-g` option to enable debugging using GDB.

```
$ gcc -g t.s
```

Then launch GDB on the executable:

```
$ gdb a.out
```

Set a breakpoint on line 6.

```
(gdb) b 6
Breakpoint 1 at 0x784: file t.s, line 6.
(gdb)
```

Enter a cool GDB layout (one of several cool layouts):

```
layout asm
```

You should see something like this:

We expected `line 6` to read:

```
        ldr         s0, =0x3fc00000
```

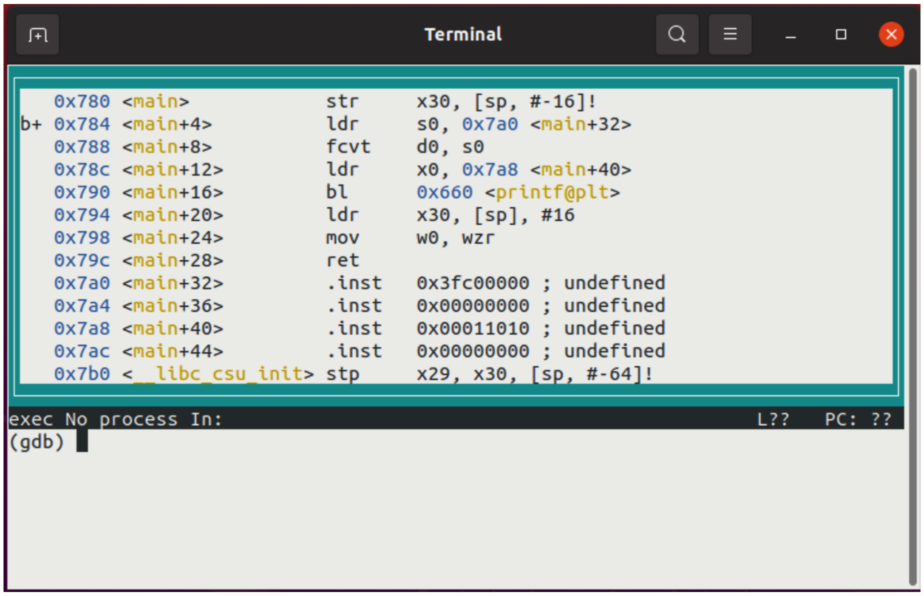

Figure 1: gdb01

Instead we find:

```
b+ 0x784 <main+4>          ldr     s0, 0x7a0 <main+32>
```

Scan downward to find `0x7a0`:

```
   0x7a0 <main+32>          .inst   0x3fc00000 ; undefined
```

Hey look! Here's our literal float. The `.inst` is an ARM specific GNU assembler directive what allows the programmer to encode their own instruction. Note, the encoded instruction does not have to make any sense - instead the compiler has emitted a make believe instruction that happens to have the value of our literal.

What we're seeing the actual `line 6` doing is reaching ahead a short distance to load the value of another "instruction" when really it is our constant.

Let us take this explanation further. Notice we see:

```
   0x78c <main+12>          ldr     x0, 0x7a8 <main+40>
```

where we expected:

```
        ldr         x0, =fmt
```

Scan down to `0x7a8`:

```
  0x7a8 <main+40>          .inst   0x00011010 ; undefined
```

`x0` is serving as a pointer to the format string of a call to `printf()`. Let's follow the pointer...

```
(gdb) x/s 0x00011010
0x11010:        "%f\n"
(gdb)
```

Magic.