Section 2 / Conversion of Floating Point and Integers

This chapter has been surprisingly difficult to research and write. Huh? All we're talking about is taking a floating point value and turning it into an integer - what could be hard?

It's hard because the AARCH64 has so many instructions that seemingly do the aforementioned job and each of them come in many variations. Even the language used is confusing.

For this chapter, I will use:

- Rounding means picking some fractional value and if the float's fraction is higher, you go one way and if lower, you go the other.
- Truncation means you don't look too closely at the fractional value. Instead, you just eliminate the fractional part and slam the whole number ... one way or the other.

Truncation Towards Zero

In C and C++, truncation is what we get from:

```
integer_variable = int(floating_variable);  // C++
integer_variable = (int) floating_variable; // C
```

Diving a little deeper, there is a choice to be made as to whether or not `integer_variable` is signed or unsigned. And, whether or not `integer_variable` is a 32 bit or 64 bit value.

The instruction is `fcvtz` - convert towards zero. Then, the choice as to whether to produce a signed or unsigned result is defined by the final letterL `u` or `s`.

Mnemonic	Meaning
fcvtzu	Truncate (always towards 0) producing an unsigned int
fcvtzs	Truncate (always towards 0) producing a signed int

As an example of how the ARM documentation is confusing - this instruction which completely discards the fractional value is said by the ARM documentation as doing rounding.

The the choice of source register defined whether you are converting a double or single precision floating point value.

Source Register	Converts a
dX	`double` to an integer

Source Register	Converts a
sX	`float` to an integer

Destination Register	Converts a
xX	64 bit integer
wX	32 bit or less integer

Examples where `d` is a `double` and `f` is a `float`:

C++	Instruction
`int32_t(d)`	`fcvtzs  w0, d0`
`uint32_t(d)`	`fcvtzu  w0, d0`
`int64_t(d)`	`fcvtzs  x0, d0`
`uint64_t(d)`	`fcvtzu  x0, d0`

Here is a program which demonstrates various ways of converting doubles to integers.

Let's look at:

```
//-fcvtzs----------------------                                  // 45
        fcvtzs    x1, dless                                     // 46
        fcvtzs    x2, dmore                                     // 47
        ldr       x0, =fmt4                                     // 48
        bl        Emit                                          // 49
                                                                // 50
        fcvtzs    x1, ndless                                    // 51
        fcvtzs    x2, ndmore                                    // 52
        ldr       x0, =fmt4                                     // 53
        bl        Emit                                          // 54
```

Reminder:

- `dless` is 5.49
- `dmore` is 5.51
- `ndless` is -5.49
- `ndmore` is -5.51

Here is the relevant output:

```
fcvtzs less: 5 more: 5
fcvtzs less: -5 more: -5
```

Notice all the values were truncated to the whole number that is *closer to zero*.

Truncation Away From Zero

Truncation away from zero is not as easy. In fact, it cannot be performed with a single instruction.

In C and C++:

```
iv = (int(fv) == fv) ? int(fv) : int(fv) + ((fv < 0) ? -1 : 1);
```

If the `fv` is already equal to a whole number, the integer value will be that whole number. Other wise the `iv` is the whole number further *away from zero*.

In C++, a more sophisticated version would require `<cmath>` and could look like:

```
template <typename T>
int MyTruncate(T x) {
    return int((x < 0) ? floor(x) : ceil(x));
}
```

- `floor()` always truncates downward (towards more negative).
- `ceil()` always truncates upwards (towards more positive).

Here is a program which demonstrates this:

In assembly language, a function is used which implements what is in essence, one instantiation of the templated function given above.

```
RoundAwayFromZero:
        fcmp    d0, 0
        ble     1f
        // Value is positive, truncate towards positive infinity (ceil)
        frintp  d0, d0
        b       2f
1:      // Value is negative, truncate towards negative infinity (floor)
        frintm  d0, d0
2:      fcvtzs  x0, d0
        ret
```

`frintp` and `frintm` will honor the source register already being a whole number (no fractional part). Thus a value of 5 will not be converted to 6 and -5 will not be converted to -6. But, a value of 5.000000001 **will** go to 6, etc.

Here is a program that demonstrates this:

```
        .text                                                   // 1
        .global main                                            // 2
        .align    2                                             // 3
                                                                // 4
main:   str       x30, [sp, -16]!                               // 5
                                                                // 6
        ldr       x0, =d                                        // 7
```

```
        ldr       d0, [x0]                                  // 8
        frintp    d0, d0                                    // 9
        ldr       x0, =fmt1                                 // 10
        bl        printf                                    // 11
                                                            // 12
        ldr       x0, =h                                    // 13
        ldr       d0, [x0]                                  // 14
        frintp    d0, d0                                    // 15
        ldr       x0, =fmt2                                 // 16
        bl        printf                                    // 17
                                                            // 18
        ldr       x30, [sp], 16                             // 19
        mov       w0, wzr                                   // 20
        ret                                                 // 21
                                                            // 22
        .data                                               // 23
fmt1:    .asciz   "with fraction:    %f\n"                  // 24
fmt2:    .asciz   "without fraction: %f\n"                  // 25
d:       .double  5.00000001                                // 26
h:       .double  5.0                                       // 27
        .end                                                // 28
```

The output is:

```
with fraction:    6.000000
without fraction: 5.000000
```

Rounding Conversion

An instruction which does what we normally think of as rounding is `frinta`. This is the conversion "to nearest with ties going away." So, 5.5 goes to 6 as one would expect from "rounding."

Converting an Integer to a Floating Point Value

In C / C++:

```
double_var = double(integer_var); // C++
double_var = (double)integer_var; // C
```

Is handled by two instructions:

- `scvtf` converts a signed integer to a floating point value
- `ucvtf` converts an unsigned integer to a floating point value

The name of the destination register controls which kind of floating point value is made. For example, specifying `dX` makes a double etc.