Section 1 / The `if` Statement

We will begin with the `if` statement followed by a discussion of the `if / else`.

`if / else if` is not discussed as it is a repeat of the discussion provided here.

`if` in `C` and `C++`

Here is a basic `if` statement in `C++`:

```
if (a > b)                                                    // 1
{                                                             // 2
    // CODE BLOCK                                             // 3
}                                                             // 4
```

For simplicity, let us assume that both `a` and `b` are defined as `long int`. Being 64 bits in width, this means `x` registers will be used in the assembly language. If `a` or `b` are not pointers and are not longs, `w` registers would sneak in somewhere. See Interlude - Registers for more information.

`if` in `AARCH64`

Here is the above `if` statement rendered into ARM V8 assembly language:

```
    // Assume value of a is in x0                             // 1
    // Assume value of b is in x1                             // 2
    cmp     x0, x1                                            // 3
    ble     1f                                                // 4
    // CODE BLOCK                                             // 5
1:                                                            // 6
```

`Lines 1` and `2` indicate that the values of variables `a` and `b` are found in registers `x0` and `x1` respectively. Recall that values in memory cannot be operated upon directly by the CPU (with very few exceptions).

The contents of memory can be loaded into registers and memory can be overwritten from registers. All the interesting action takes place in registers. The choice of `x` registers is made based on the assumption that `a` and `b` are long integers.

Line 3

The `cmp` instruction is actually a shorthand for a subtraction instruction that discards the result of the subtraction but keeps a record of whether or not the result was less than, equal to or greater than zero.

The second operand is subtracted from the first.

This means that the condition bits (status of a previous `cmp`) are formed using `x0 - x1`.

If `a > b` then `x0 - x1` will be *greater than zero.*

If `a == b` then `x0 - x1` will be *equal to zero.*

If `a < b` then `x0 - x1` will be *less than zero.*

Handling of `>=` and `<=` follow from the above.

Line 4

Using the state of the condition bits (which are set by the faux subtraction of `x1` from `x0` performed by `cmp`), branch (a jump or goto) if the previous computation shows `less than or equal to` zero. Notice the use of the *opposite* condition as found in the `C` code. This use of the opposite condition is not a hard and fast rule. In this case, it allows the body of the `if` statement to be written directly below the branch so as to emulate the skipping of the code block contained between the `if` statement's braces.

This is a matter of style.

In the higher level language, you want to *enter* the following code block if the condition is true. In assembly language, you want to *avoid* the following code block if the condition is false.

Use of temporary labels

The target of the branch instruction is given as `1f`. This is an example of a *temporary label.*

There are a lot of braces used in C and C++. Since labels frequently function as equivalents to `{` and `}`, there can be a lot of labels used in assembly language.

A temporary label is a label made using just a number. Such labels can appear over and over again (i.e. they can be reused). They are made unique by virtue of their placement relative to where they are being used. `1f` looks `f`orward in the code for the next label `1`. `1b` looks in the `b`ackward direction for the most recent label `1`.

Line 6

This line acts in place of the `if` statement's closing `}`. Notice it is the target of the `ble` found on `Line 4`.

`if / else`

Here is a basic `if / else`:

```
if (a > b)                                                    // 1
{                                                             // 2
    // CODE BLOCK IF TRUE                                     // 3
```

```
    }                                                                   // 4
    else                                                                // 5
    {                                                                   // 6
        // CODE BLOCK IF FALSE                                          // 7
    }                                                                   // 8
```

There are two branches built into this code!

First, the *true* block has to be skipped over if the condition is *false*.

Second, the *true* block (if taken) must skip over the *false* block.

Here is the corresponding assembly language.

```
    // Assume value of a is in x0                                       // 1
    // Assume value of b is in x1                                       // 2
    cmp     x0, x1                                                      // 3
    ble     1f                                                          // 4
    // CODE BLOCK IF TRUE                                               // 5
    b       2f                                                          // 6
1:                                                                      // 7
    // CODE BLOCK IF FALSE                                              // 8
2:                                                                      // 9
```

Lines 1 Through 6

These lines are unchanged from the previous example.

Line 7

`Line 7` acts like the `{` in the `else`.

Line 9

`Line 9` acts like the `}` of the `else`.

A complete program

Without much explanation, here is a complete program you can play around with:

```
    .global main                                                        // 1
    .text                                                               // 2
                                                                        // 3
main:                                                                   // 4
    stp     x29, x30, [sp, -16]!                                        // 5
    mov     x1, 10                                                      // 6
    mov     x0, 5                                                       // 7
                                                                        // 8
    cmp     x0, x1                                                      // 9
```

```
    ble     1f                                          // 10
    ldr     x0, =T                                      // 11
    bl      puts                                        // 12
    b       2f                                          // 13
                                                        // 14
1:  ldr     x0, =F                                      // 15
    bl      puts                                        // 16
                                                        // 17
2:  ldp     x29, x30, [sp], 16                          // 18
    mov     x0, xzr                                     // 19
    ret                                                 // 20
                                                        // 21
    .data                                               // 22
F:  .asciz  "FALSE"                                     // 23
T:  .asciz  "TRUE"                                      // 24
                                                        // 25
    .end                                                // 26
```

Here is the original code.

`Line 11` is one way of loading the address represented by a label. In this case, the label `T` corresponds to the address to the first letter of the C string "TRUE". `Line 15` loads the address of the C string containing "FALSE".

The occurrences of `.asciz` on `line 23` and `line 24` are invocations of an *assembler directive* the creates a C string. Recall that C strings are NULL terminated. The NULL termination is indicated by the `z` which ends `.asciz`.

There is a similar directive `.ascii` that *does not NULL terminate* the string.

Summary

`if` statements are implemented by some code that causes the condition bits to be set (less than zero, less than or equal to zero, equal to zero, greater than or equal to zero and greater than zero). Then, a branch is taken if a specific condition is present.

Labels are used to mark where code blocks end and in the case of an `else`, where code blocks begin.

A label marking the end of a code block is used as the target of a branch meant to skip the code block. A label marking the beginning of a code block allow a branch to that code block, such as the beginning of an `else`.

Questions

1

(T | F) If statements in assembly language always test for the opposite condition as the equivalent `if` statement in a high level language.

Answer: False - it is a matter of style but you may be able to save an instruction or two by doing so.

2

(T | F) `cmp` isn't a "real" instruction but rather is an alias for a subtraction.

Answer: True - `cmp` is an alias for `subs` which is a subtract that discards the resulting value but does set the condition bits.

3

We claim `if05.s` (the complete program given above) is too long! By two instructions, that is. Copy `if05.s` to make `if06.s`. Then, modify `if06.s` to be two instructions shorter.

To do so, notice that there are two occurrences of `bl puts` in `if05.s`. Refactor the code to have only one.

Answer: The shorter version is found here. It is well documented and should be studied.