Jump or Branch Tables

A jump or branch table is a powerful instruction saving technique that can be used to switch between multiple single instructions or even choose one of a series of functions to call (or branches to take).

This concept can be found as the implementation of some `switch` statements and is found at the very very lowest end of an Operating System (interrupt vectors, for example).

The

Single Instructions a la Duff's Device

Duff's Device shoe horned a jump table into the middle of a `while` loop. At the same time, it also demonstrates a simple case of *loop unrolling*. It's very creative.

Let's expand on Duff's Device.

The full source code for this example can be found here. It demonstrates a branch table consisting of instructions which are meant to be executed in sequence after jumping into the middle of the sequence.

Here:

```
    mov         x6, 8
    MOD         x2, x6, x4, x5  // x4 gets l % 8
    cbz         x4, 10f         // Handle evenly divisible case.
    sub         x4, x6, x4      // Invert sense of x4 e.g. 3 becomes 5
```

we are performing this: *x4 is getting the result of modding the number of times we want the instructions executed by the number of times we unrolled the loop.*

Specifically, this example does `length % 8`. However, the AARCH64 ISA does not include a *mod* instruction. The `MOD` macro used above is defined as:

```
.macro  MOD         src_a, src_b, dest, scratch
        sdiv        \scratch, \src_a, \src_b
        msub        \dest, \scratch, \src_b, \src_a
.endm
```

`msub` is a cool instruction. It does this:

`d = c - (b * a)`

Example: 13 % 8 == 5. First the `sdiv`: 13 / 8 is 1. Then, the `msub`: 13 - (1 * 8) is 5.

Next:

```
        cbz         x4, 10f         // Handle evenly divisible case.
        sub         x4, x6, x4      // Invert sense of x4 e.g. 5 becomes 3
```

This code is key.

If the result of the `mod` is 0, then the entire table must be executed. This is implemented by the `cbz`.

If the result of the `mod` is not 0, then its value must be *flipped*. The idea here is that if the result of the mod is 5, for example, we have 5 stragglers. We want to execute 5 of the sequential instructions below. So, we want to jump 3 instructions into the table. Notice that 3 is 8 - 5.

Finally, we have the computation of the address to where we jump into the middle of the table.

```
        LLD_ADDR    x5, 10f
        add         x5, x5, x4, lsl 2
        br          x5
```

Each of the lines above bears description:

The `LLD_ADDR` is from the *convergence macros*. It loads the address of the beginning of the table.

Next, the `add` instruction multiplies the flipped result of the `mod` by 4 (the length of one instruction) THEN adds it to the base address of the table. We have calculated *instruction addresses* exactly the way we would with array dereferences. Thank you John von Neumann.

Finally, we `br` which means branch to an address contained in a register.

```
10:     str         w1, [x0], 1
        str         w1, [x0], 1
        str         w1, [x0], 1
        str         w1, [x0], 1
        str         w1, [x0], 1
        str         w1, [x0], 1
        str         w1, [x0], 1
        str         w1, [x0], 1
        // loop code not shown
```

Performing Multiple Instructions

If you need to execute more than one instruction you have two choices:

Multiple Instructions by Address Arithmetic

Suppose you needed two instructions in each step of the sequence. Simply multiply the index by 8 instead of 4 (i.e. the length of two instructions). The same technique works with a larger number. E.g. you need three instructions per step: multiply by 12.

Suppose some need 3 instruction and some need 2. You must handle this because using this technique requires that all steps in the sequence of steps must be the same length so that the address arithmetic works.

To deal with some cases being shorter than others, insert the occasional `nop` instruction in the indexes that are shorter than the others.

Multiple Instructions by Branch / Branch

Here's another example of code that implements a branch or jump table:

```
jt:     b       0f
        b       1f
        b       2f
        b       3f
        b       4f
        b       5f
        b       6f
        b       7f
```

You jump into the middle of the table as per above and then immediately jump some place else. This is like:

```
if (index == 0) {
    blah
} else if (index == 1) {
    blah
} else if (index == 2) {
    blah
} etc.
```

Multiple Instructions by Branch / Call

You can modify the above techniques to make something like:

```
jt:     bl       func_0
        bl       func_1
        bl       func_2
        bl       func_3
        bl       func_4
        bl       func_5
        bl       func_6
        bl       func_7
```

or to be more similar to a `break` statement coming after each case:

```
jt:     bl       func_0
        b        common_label
        bl       func_1
        b        common_label
        bl      func_2
        b       common_label
        bl      func_3
        b       common_label
        bl      func_4
        b       common_label
        bl      func_5
        b       common_label
        bl      func_6
        b       common_label
        bl      func_7
        b       common_label

        // perhaps  some  loop control... if none, the preceding
        // b can be removed since can fall through to the common
        // label.
common_label:
```

Small Gaps in Sequential Indexes

Suppose your range of indexes was 0 through 8 inclusive (notice there are 9 integers in the range) but index 7 is skipped. That is, your potential indexes are 0 through 6 inclusive and then 8 but never 7.

In a `switch` statement, this would look like:

```
/*
// Ensure index is a valid value before getting here. In this case the
// valid range is 0 through 8 inclusive (a range of 9 values). To fill
// out to the next power of 2 (which would be 16), one could put in
// empty cases plus a default.
*/
switch (index & 0xF) {
    case 0: blah blah;
            break;
    case 1: blah blah;
            break;
    case 2: blah blah;
            break;
    case 3: blah blah;
            break;
    case 4: blah blah;
            break;
    case 5: blah blah;
            break;
    case 6: blah blah;
            break;
    case 8: blah blah;
            break;
}
```

Gaps in the potential indexes presents a surmountable problem if the gaps are few.

In the case where there are a small number of gaps simple fill them with a branch to a common, otherwise "do nothing", label. For example, you might have:

```
b_table:    b       label0
            b       label1
            b       label2
            b       label3
            b       label4
            b       label5
            b       label6
            b       do_nothing
            b       label8
```

in the style of Duff's Device where you are executing sequential single instructions, it might loop like this:

```
x_fer:      str     w1, [x0], 1
            str     w1, [x0], 1
            str     w1, [x0], 1
            str     w1, [x0], 1
            str     w1, [x0], 1
            str     w1, [x0], 1
            str     w1, [x0], 1
            nop
            str     w1, [x0], 1
```

Here, the `nop` instruction means "no operation". It does nothing but is a valid instruction meant to take up space (and decades ago, take up time).

In a high level language this might look like this:

```
for (int i = 0; i <= 8; i++) {
    if (i == 7)
        continue;
    blah blah
}
```

More about the `switch` statement

`switch` statements are optimized using many techniques than suggested here. In fact, the implementation of optimized `switch` statements is fascinating. There might be:

- binary searches for large numbers of cases
- separation of ranges where each sub-range is optimized in a different way
- degeneration into streams of if / else ifs

and other techniques. The people who work on the compilers we take for granted really are due some respect and *free beer*.