Section 3 / Bit Fields / Without Bit Fields

Overview

Many C and C++ programmers have never seen bit fields.

Bit fields are a feature of the C and C++ language which completely hide what is often called “bit bashing”.

Bit bashing is the manipulation of individual bits. Bit bashing goes to the very core of the C language. Remember that C is a high level assembly language, as we argue in Section 1 of this book. And C is the (later) language in which Unix was implemented and indeed, C was developed specifically to implement Unix.

Since an operating system directly interfaces with hardware - the C language grew to have features to aid Unix implementers.

With that said, consider this WARNING: the ordering of bits in a bit field is not guaranteed to be the same on different platforms and even between different compilers on the same platform.

Bit fields are implemented within a `struct` by appending a colon plus a number after the declaration of integer types.

For example:

```
struct BF {
    unsigned char a : 1;
    unsigned char b : 2;
    unsigned char c : 5;
};
```

The above declares a `struct` whose size is 1 byte. Members of the `struct` are `a`, `b` and `c` which are 1, 2 and 5 bits in size, respectively.

Bit Fields Aren’t Just For Hardware

Consider a data structure for which there will be potentially millions of instances in RAM. Or, perhaps billions of instances on disc. Suppose you need 8 boolean members in every instance. The C++ standard does not define the size of a `bool` instead leaving it to be implementation dependent. Some implementations equate `bool` to `int`, four bytes in length. Some implement `bool` with a `char`, or 1 byte in length.

Let’s assume the smallest case and equate a `bool` with `char`. Our `struct`, for which there may be millions or billions of instances requires 8 `bool` so therefore 8 bytes. Times millions or billions.

Ouch.

Bit fields can come to your aid here by using a single bit per boolean value. In the best case, 8 bytes collapse to 1 byte. In a worse case, 8 x 4 = 32 bytes collapsed into 1.

Without Bit Fields

Before we examine using bit fields, let's look at what life would be like without them.

Let's assume we're working with a byte that is comprised of three fields laid out as in `struct BF` above. That is, a one, two and five bit field inside one byte.

Without bit fields, we would have to write this code to clear `a` to zero:

```
void ClearA(unsigned char * byte) {
    *byte &= ~1;
}
```

This function takes the address of the byte containing the `a`, `b` and `c` portions.

Good programming practice would check `byte` against NULL or `nullptr`.

The `~` operator is a bitwise negation. All the bits in the value are flipped from 0 to 1 or 1 to 0. `~1` in an unsigned char will produce `0xFE`, or all ones except for bit 0. `and`ing this value to `*byte` ensures that its bit 0 is 0 and all other bits are left alone.

In assembly language, written *naively*, this would look like this:

```
ClearA: ldrb    w1, [x0]                                           // 1
        mov     w2, 1                                              // 2
        mvn     w2, w2                                             // 3
        and     w1, w1, w2                                         // 4
        strb    w1, [x0]                                           // 5
        ret                                                        // 6
```

`x30` does not have to be backed up or restored as this function is a "leaf."

`Line 3` uses the instruction `mvn` to flip all the bits in `w2`.

This code completely tracks the C / C++ code.

We have no obligation to follow the C / C++ code exactly. Instead we could write:

```
ClearA: ldrb    w1, [x0]                                           // 1
        and     w1, w1, 0xFE                                       // 2
        strb    w1, [x0]                                           // 3
        ret                                                        // 4
```

Here, the `0xFE` literal takes the place of `lines 2 and 3` in the previous version. We do this by pre-computing what the `mov` and `mvn` would have produced.

For setting the `a` bit, we would do this:

```
void SetA(unsigned char * byte) {
    *byte |= 1;
}
```

This is an anomaly for bit bashing. In almost all cases when setting bit values, the bits must be cleared first because an *or* instruction is responsible for setting any 1 bits to 1.

It is important you get that when needing to set a number of bits to a specific value, those bit must be cleared first so that an `orr` can do the right thing.

In this case, it is a single bit we're setting so we can just or it in.

In assembly language:

```
SetA:   ldrb    w1, [x0]                                            // 1
        orr     w1, w1, 1                                           // 2
        strb    w1, [x0]                                            // 3
        ret                                                         // 4
```

`orr` is one of several or instructions in AARCH64. It is the one that maps most closely to `|` in C and C++.

Moving onto the `b` field, things begin to get a little more interesting. To clear the `b` field we might do this in C | C++.

```
void ClearB(unsigned char * byte) {
    *byte &= ~6;
}
```

This could *naively* be written as:

```
ClearB: ldrb    w1, [x0]                                            // 1
        mov     w2, 6                                               // 2
        mvn     w2, w2                                              // 3
        and     w1, w1, w2                                          // 4
        strb    w1, [x0]                                            // 5
        ret                                                         // 6
```

This code is essentially the same as the *naive* version of `ClearA` given above. Once again, we can pre-compute the results of `lines 2 and 3` to make:

```
ClearB: ldrb    w1, [x0]                                            // 1
        and     w1, w1, 0xF9                                        // 2
        strb    w1, [x0]                                            // 3
        ret                                                         // 4
```

Turning to setting `b`, the code gets a little more complicated as for the first time, we have to accept a parameter for the value to place into `b`. And, `b` is more than one bit.

```c
void SetB(unsigned char * byte, unsigned char value) {              // 1
    value &= 3;             // ensures only bits 0 and 1 can be set   // 2
    *byte &= ~6;            // clears bits 1 and 2 in byte            // 3
    *byte |= (value << 1);  // stores bits 0 and 1 into bits 2 and 3  // 4
}                                                                   // 5
```

`Line 2` is necessary to prevent stray 1's from being or'ed into `*byte`.

`Line 3` is necessary to squash the existing target bits to zero prior to being `orr`'ed.

Notice `value` is being shifted left by 1 bit as the `b` field begins at bit index 1.

In *naive* assembly language we could write this:

```
SetB:   ldrb    w3, [x0]                                    // 1
        and     w1, w1, 3          // value &= 3            // 2
        lsl     w1, w1, 1                                   // 3
        mov     w2, 6                                       // 4
        mvn     w2, w2                                      // 5
        and     w3, w3, w2         // B is cleared          // 6
        orr     w3, w3, w1                                  // 7
        strb    w3, [x0]                                    // 8
        ret                                                 // 9
```

The only interesting thing in this code is that we chose to perform the left shift (`lsl`) by one bit earlier in the code rather than later. There is ill no side effect to changing this order.

`lsl` means "left shift logical" which fills the right side recently vacated bits with zero.

Now, we present a more sophisticated version of `SetB`:

```
SetB:   ldrb    w3, [x0]                                            // 1
        bfi     w3, w1, 1, 2 // copy bit 0..1 in w1 to bit 1..2 in w3  // 2
        strb    w3, [x0]                                            // 3
        ret                                                         // 4
```

Whoa. Nine instructions down to four! What the heck is `bfi`?

`bfi dst, src, start, width` copies `width` bits starting at 0 in `src` to bits starting at `start` in `dst`.

It obviates the need for `line 2` in the naive code because it plucks only bits 0 and 1 and no others from the original value of `w1`.

The `bfi` then internally does the shift appropriate to move bit 0 of `w1` to bit `start` along with `width - 1` subsequent bits. Finally, the shifted bits overwrite the same bits in `w3`.

Some might argue that instructions like `bfi` (and `ubfiz` described below) is an example of `ISA creep` where ISA's get more and more cumbersome with the latest instructions du jure. This is definitely true in the x86 ISA. Perhaps this is true in the AARCH64 ISA as well, but certainly not to the extent of the x86.

Remember that the ARM family of processors are examples of RISC machines - *reduced instruction set* architectures.

Finally, we come to handling field `c`. Recall `c` is 5 bits long starting at bit 3.

Clearing the bits in `c` is easily accomplished:

```
void ClearC(unsigned char * byte) {
    *byte &= 7;             // squashes bits 3 to 7 to 0
}
```

This is optimally implemented using:

```
ClearC: ldrb    w1, [x0]                                                // 1
        and     w1, w1, 7                                               // 2
        strb    w1, [x0]                                                // 3
        ret                                                             // 4
```

As for setting the value of `c`, we have this in C / C++:

```
void SetC(unsigned char * byte, unsigned char value) {
    value &= 0x1F;          // ensures only bits 0 to 4 can be set
    *byte &= ~(0x1F << 3);  // squashes correct bits in byte
    *byte |= (value << 3);  // or's in the bits at the right place
}
```

In naive assembly language, this function would look like this:

```
SetC:   ldrb    w3, [x0]                                                // 1
        mov     w2, 0x1F                                                // 2
        and     w1, w1, w2                                              // 3
        lsl     w1, w1, 3                                               // 4
        lsl     w2, w2, 3                                               // 5
        mvn     w2, w2                                                  // 6
        and     w3, w3, w2                                              // 7
        orr     w3, w3, w1                                              // 8
        strb    w3, [x0]                                                // 9
        ret                                                             // 10
```

`Lines 1 and 2` in the assembly language performs `line 1` of the C code.

`Line 4` shifts `value` up to where `c` starts. `Line 5` similarly shifts the mask up to where `c` starts. Its bits are negated on `line 6`. `Line 7` squashes the upper five bits to zero followed by the `orr`ing on `line 8`.

A more sophisticated version of the assembly language, leveraging some fancy bit insertion / copying instructions, is far shorter.

```
SetC:   ldrb    w2, [x0]        // put *byte into w2                    // 1
        ubfiz   w1, w1, 3, 5    // zero new w1, copy bits 0..4 to 3..7  // 2
```

```
and     w2, w2, 7        // preserve only 1st 3 bits in *byte    // 3
orr     w2, w2, w1       // or in value into *byte               // 4
strb    w2, [x0]                                                 // 5
ret                                                              // 6
```

`Line 2` uses the instruction `ubfiz` which means Unsigned Bit Field Insert Zeroed. This instruction:

- Zeros out a new copy of `value` (`w1`), the destination and
- Copies 5 bits starting at bit 0 of the old `value` to bits 3 through 7 in the new version of `value`.

This one instruction does the work of `lines 2, 3 and 4` in the naive version of the assembly language.

`Line 3` of the new assembly language replaces `lines 4, 5 and 6` in the naive. This works because the enlightened human saw an easier way to zero out *byte* except* for the first 3 bits (where `a` and `b` live).

The remainder is as expected.

Summary

In this chapter we saw was life was like without bit fields. We saw that we had to implement our own bit bashing functions to do things like:

- Ensure parameters are in the right range
- Shift values around to line up with their destination
- Zero out destination fields
- Or in new values, having been shifted to the right position

and more.

We brushed upon the idea that bit bashing and bit fields are critical to directly interfacing with hardware but are also useful in decreasing the size of data structures in memory and on disc.

Space Versus Time

In Computer Science there is an eternal tension between space and time. The following is a **law**:

If you want something to go faster, it will cost more memory.

If you want to save memory, what you're doing will take more time.

This law shows up here... recall the example of where we wanted to save memory by collapsing 8 `bool` into 1 byte? To save that memory we will slow down because accessing the right bits takes a couple of instructions where overwriting a `bool` implemented as an `int` takes just one instruction.