Section 3 / Bit Fields / Review of Newly Described Instructions

Overview

Our discussion of implementing ourselves what the C / C++ compiler gives us led us to use six new instructions. This chapter reviews those instructions. In addition to describing what an instruction does, we will try to indicate what the instruction is "good for."

`and`

The `and` instruction is pretty much what you would expect. It implements the `&` operator from C and C++. That is, the bitwise and operator.

The `and` instruction comes in a number of flavors including for use with immediate values.

`and` Immediate

This is the form of the instruction we used.

```
and    rd, rs, imm
```

This performs a bitwise and of the `imm` value with the source register `rs` placing the result in the destination register `rd`.

There are limits to the bit width of `imm` because it has to fit within the `and` instruction. If you exceed the allowable width of `imm`, the assembler will be glad to insult you.

It is possible that a `mov` instruction will allow your immediate value. You'd follow up with an `and` using a register than an immediate value.

If your immediate value is too large for a `mov` then put the value in RAM and `ldr` it into a register and proceed.

We would love to tell you what the rules are for an immediate value in the `and` instruction, but they are not obvious and in fact are very complex. Our advice, try the immediate value you have in mind and if it works, great. Otherwise, see above.

A slight variation on this `and` instruction uses a register where the preceding one uses an immediate value.

```
and    rd, rs, rm
```

This ands `rm` to `rs` and places the result in `rd`.

This instruction has a variation:

```
and    rd, rs, rm, *shift* num_bits
```

`*shift*` can be one of `lsl`, `lsr`, `asr` or `ror`.

These mean:

`*shift*`	Meaning	Meaning of the Meaning
`lsl`	logical shift left	shifts left introducing zeros on the right
`lsr`	logical shift right	shifts right introducing zeros on the left
`asr`	arithmetic shift right	shifts right introducing duplicates of the previous most significant bit
`ror`	rotate right	shifts right introducing the bits shifted out back in from the left

There are other two similar `and` instructions. These are the immediate and the register versions of `ands`. `ands` is the same as `and` with the addition that the CPU's condition bits are updated by the instruction permitting a conditional branch to follow the instruction.

`and` is the basis of bit bashing and is used to clear bits. Put `and` together with the or instructions and a couple of other basic logic instructions, out pops a computer.

A shorthand way of clearing specific bits is the `bic` instruction. Use of `bic` can avoid having to negate the bits in a mask prior to and'ing.

`bfi`

This instruction mnemonic stands for Bit Field Insert. The word *Insert* should really be copy. The official ARM documentation explains this instruction very well so we'll repeat it here:

Bit Field Insert copies any number of low-order bits from a source register into the same number of adjacent bits at any position in the destination register, leaving other bits unchanged.

The instruction has the following format:

```
bfi    rd, rs, lsb, width
```

Starting at bit 0 of `rs`, `width` bits are copied to `rd` starting at bit `lsb`.

The `bfi` instruction replaces as many as three instructions: likely a shift, an `and` and an `orr`.

The preceding has described what `bfi` does but what is it for? Suppose you have a multiple bit field in a device register. Using `bfi` makes it easy to copy the right number of bits from a source directly into the bits in the middle of the destination without need of other bit-bashing.

The *opposite* of `bfi` is `bfx`.

`ubfiz` first zeros the destination register then copies in the specified bits from the source. The `u` means don't consider the sign bit as special. The `z` is what causes the zero fill first and sets it apart from `bfi`.

See bottom for an example.

`mvn`

This instruction takes only takes two operands (but permits an optional shift to be explained below).

The basic syntax is:

```
mvn    rd, rs
```

This flips all the bits in the source and copies them to the destination.

In addition to the basic instruction, there is also:

```
mvn    rd, rs, *shift* num_bits
```

The *shift* and `num_bits` function the same way as described above including the option to use `*shift*` as one of `lsl`, `lsr`, `asr` or `ror`.

Note the `mvn` is different from `neg` in that `mvn` is a bitwise operation while `neg` is aware of what makes a negative integer value. That is, if your goal is to turn a positive integer into a negative integer, use `neg`. If your goal is to negate bits for bit bashing purposes, use `mvn`.

See bottom for an example.

`lsl`

Logical Shift Left moves bits to the left shifting 0 into any vacated positions.

```
lsl     rd, rs, *number of bits*
```

`lsl` is most often used for bit bashing as well as for a fast multiplication by a power of 2.

The instruction `asl` is a synonym for `lsl`. The `a` is `asl` means "arithmetic". There is no difference between a logical and an arithmetic shift in the **leftward** direction. There is, however, a difference in the **rightward** direction in that `asr` replicates the sign bit where `lsr` still moves in zeros in any bit positions vacated.

`orr`

This is the plain vanilla *or* instruction. It is used extensively for bit bashing as it is how bits can be set to 1.

```
    orr     rd, rs, rm  # rm is another register
    orr     rd, rs, imm
```

Example

Here is the code to an example:

```
        .global         main                                                    // 1
        .text                                                                   // 2
        .align          2                                                       // 3
/*    This demo should be run from gdb. `layout regs` would be helpful.         // 4
    This demo shows:                                                            // 5
    bfi - bit field insert                                                      // 6
*/                                                                              // 7
main:   mov         w1, wzr         //                                         // 8
        mvn         w1, w1          // set w1 to 0xFFFFFFFF                    // 9
        mov         w3, w1          // save for reuse                          // 10
        mov         w2, 3           // set bits 0 and 1                        // 11
        bfi         w1, w2, 4, 4                                               // 12
        // Look at w1. You will note that the bottom                           // 13
        // 4 bits of w2 (0011) have been copied into the second                // 14
        // 4 bits of w1 including the zeros.                                   // 15
        //                                                                     // 16
        // NEXT DEMO                                                           // 17
        mov         w1, w3          // reset w1                                // 18
        mov         w2, 3           // set bits 0 and 1                        // 19
        ubfiz       w1, w2, 4, 2                                               // 20
        // Look at w1. You will note that all the bits were                    // 21
        // zeroed and then 2 bits from w2 are copied into w1                   // 22
        // starting at bit 4. Compare this to bfi.                             // 23
        //                                                                     // 24
        // NEXT DEMO                                                           // 25
        bfxil       w3, w1, 4, 4                                               // 26
        // Look at w3. You will note that 4 bits (0011) from                   // 27
        // w1 were put into w3 starting at bit 0.                              // 28
        mov         w0, wzr                                                    // 29
        ret                                                                    // 30
```