Exploring AArch64 assembler – Chapter 3

Oct 23, 2016 • Roger Ferrer Ibáñez • AArch64

In the last chapter we saw that instructions may have register operands and immediate operands. We also mentioned that mixing 32-bit and 64-bit register was not allowed. Today we will talk a bit more about register operands.

Operators for register operands

Many instructions that take a register as the second source operand of an instruction can also apply some extra operation to the value of that source register. This can be used as a way to increase density of computation by requiring less instructions and also to allow some common operations, e.g. conversions, in one of the operands.

We can distinguish two kinds of operators here: shifting operators and extending operators.

Shifting operators

There are three shifting operators in AArch64: LSL, LSR, ASR and ROR. Their syntax is as follows:

```
reg, LSL, #amount
reg, LSR, #amount
reg, ASR, #amount
reg, ROR, #amount
```

where `reg` can be a 64-bit register `Xn` or a 32-bit register `Wn` and `amount` is a number whose range depends on the register used and ranges from 0 to 31 for 32-bit registers and from 0 to 63 for 64-bit registers.

Operator `LSL` performs a **l**ogical **s**hift **l**eft to the value in `reg` (it does not change the contents of `reg` though). Shifting *n* bits to the left means introducing `n` zeros as the least significant bits and discarding `n` most significant bits from the original value. Shifting left n-bits is equivalent to multiply to $2^{n</n>.}$ </p>

```
add r1, r2, r3, LSL #4 /* r1 ← r2 + (r3 << 4)
                          this is the same as
```

```
                            r1 ← r2 + r3 * 16 */
add r0, r0, r0, LSL #2  /* r0  ← r0 + r0 << 2
                           this is the same as
                           r0 ← r0 + r0 * 4
                           which happens to be the same as
                           r0 ← r0 * 5
                           assuming no overflow happens
                         */
```

Operator LSR performs a **l**ogical **s**hift **r**ight. This operation is the dual of LSL, but zeros are introduced in the *n* most significant bits and the *n* least significant bits are discarded. For unsigned arithmetic numbers, this operation is equivalent to division by 2^n.

Operator ASR performs an **a**rithmetic **s**hift **r**ight. This is like LSR but instead of introducing zeros in the *n* most significant bits the most significant bit is replicated *n* times in the *n* most significant bits. As in LSR, the *n* least significant bits are discarded. If the most significant bit of the register is zero, ASR is equivalent to LSR. This shift operator is useful for two's complement numbers as it propagates the sign bit (which would be the most significant bit in the register if interpreted as a binary number) and it can be used for dividing by 2^n negative numbers as well. A LSR on a two's complement negative number does not make sense for the purpose of a division.

Operator ROR performs a **ro**tate **r**ight of the register. This is commonly used for cryptography and its usage is less usual than the other shifting operands. A rotation is similar to LSR but rather than dropping bits and introducing zeros, the least signficant bits that would be dropped are introduced as the most significant bits. There is no rotate left because a rotate right can be used for this: just rotate all bits minus the number of steps we want to rotate to the left.

In AArch64 only a few instructions (mainly logical ones) can use the ROR shifting operator.

```
mov w2, #0x1234            // w2 ← 0x1234
mov w1, wzr                // w1 ← 0
orr w0, w1, w2, ROR #4     // w0 ← BitwiseOr(w1, RotateRight(w2, 4))
                           // this sets w0 to 0x40000123
orr w0, w1, w2, ROR #28    // w0 ← BitwiseOr(w1, RotateRight(w2, 32-4))
                           // this is in practice like RotateLeft(w2, 4)
                           // so this sets w0 to 0x12340
```

Extending operators

Extending operators main purpose is to widen a narrower value found in a register to match the number of bits for the operation. An extending operator is of the form *k*`xt`*w*, where *k* is the kind of integer we want to widen and *w* is the width of the narrow value. For the former, the kind of integer can be `U` (unsigned) or `S` (signed, i.e. two's complement). For the latter the width can be `B`, `H` or `W` which means respectively **b**yte (least 8 significant bits of the register), **h**alf-word (least 16 significant bits of the register) or **w**ord (least significant 32 bits of the register).

This means that the extending operators are `uxtb`, `sxtb`, `uxth`, `sxth`, `uxtw`, `sxtw`.

These operators exist because sometimes we have to *lift* the range of the source value from a smaller bit width to a bigger one. In later chapters we will see many cases where this happens. For instance, it may happen that we need to add a 32-bit register to a 64-bit register. If both registers represent two's complement integers then

```
add x0, x1, w2, sxtw  // x0 ← x1 + ExtendSigned32To64(w2)
```

There is some kind of context that has to be taken into account when using these extension operators. For instance, the two instructions below have slight different meanings:

```
add x0, x1, w2, sxtb // x0 ← x1 + ExtendSigned8To64(w2)
add w0, w1, w2, sxtb // w0 ← w1 + ExtendSigned8To32(w2)
```

In both cases the least significant 8 bits of `w2` are extended but in the first case they are extended to 64 bit and in the second case to 32-bit.

Extension and shift

It is possible to extend a value and then shift it left 1, 2, 3 or 4 bits by specifying an amount after the extension operator. For instance

```
mov x0, #0                // x0 ← 0
mov x1, #0x1234           // x0 ← 0x1234
add x2, x0, x1, sxtw #1   // x2 ← x0 + (ExtendSigned16To64(x1) << 1)
                          // this sets x2 to 0x2468
add x2, x0, x1, sxtw #2   // x2 ← x0 + (ExtendSigned16To64(x1) << 2)
                          // this sets x2 to 0x48d0
add x2, x0, x1, sxtw #3   // x2 ← x0 + (ExtendSigned16To64(x1) << 3)
                          // this sets x2 to 0x91a0
add x2, x0, x1, sxtw #4   // x2 ← x0 + (ExtendSigned16To64(x1) << 4)
                          // this sets x2 to 0x12340
```

This may seem a bit odd and arbitrary at this point but in later chapters we will see that this is actually useful in many cases.

This is all for today.

« Exploring AArch64 assembler – Chapter 2 Exploring AArch64 assembler – Chapter 4 »

Powered by Jekyll. Theme based on whiteglass
Subscribe via RSS