The ARM processor (Thumb-2), part 3: Addressing modes

devblogs.microsoft.com/oldnewthing/20210602-00

June 2, 2021

Raymond Chen

The ARM processor employs a load-store architecture, but that doesn't mean that it has to skimp on the addressing modes.

Every addressing mode starts with a base register. A base register of *pc*, may be used only in load instructions, and the value is rounded down to the nearest multiple of 4 before being used in calculations. The *pc*-relative addressing mode is typically used to load constants that are embedded in the code segment.

For demonstration purposes, I'll use the `LDR` instruction (load 32-bit register) to illustrate the addressing modes.

Register indirect

```
ldr r0, [r1]            ; r0 = *r1
```

Register indirect is the simplest addressing mode. The address is provided entirely by the base register.

Register with immediate offset

```
ldr r0, [r1, #imm]      ; r0 = *(r1 + imm)
ldr r0, [r1, #-imm]     ; r0 = *(r1 - imm)
```

The offset is added to or subtracted from base register, and the result is the address to be accessed. The offset can be in the range −255 … +4095, with small positive offsets offering the possibility of a 16-bit encoding.

Register with register offset

```
ldr r0, [r1, r2]        ; r0 = *(r1 + r2)
ldr r0, [r1, -r2]       ; r0 = *(r1 - r2)
```

The value of the offset register is added to or subtracted from the base register to form the effective address.

Register with scaled register offset

```
ldr r0, [r1, r2, LSL #2]    ; r0 = *(r1 + (r2 << 2))
ldr r0, [r1, -r2, ASR #1]   ; r0 = *(r1 - (r2 >> 1))  signed shift
```

The scale is an operation performed by the barrel shifter on the offset before it is combined with the base register. The ARM processor is very proud of its barrel shifter.

The barrel shifter can perform the following operations:

Mnemonic	Meaning	Range	Notes
	Do nothing		No scaling applied
`LSL #imm`	Logical shift left	1 ≤ imm ≤ 31	Shift left with zero-fill
`LSR #imm`	Logical shift right	1 ≤ imm ≤ 32	Shift right with zero-fill
`ASR #imm`	Arithmetic shift right	1 ≤ imm ≤ 32	Shift right with sign-fill
`ROR #imm`	Rotate right	1 ≤ imm ≤ 31	32-bit rotation
`RRX`	Rotate right extended	1	33-bit rotation (carry is the extra bit)

Some shift operations seem to be missing, but they aren't. Arithmetic shift left (`ASL`) is the same as logical shift left, and rotate left (`ROL`) is the same as right rotation by 32 – #imm.

On the other hand `RLX` truly is missing. But then again, who cares? I have no idea who would ever use `RRX` anyway.

The assembly syntax separates offset from the the scale with a comma, which looks a bit odd. A more natural-looking syntax would be

```
ldr r0, [r1, r2 lsl #2]     ; r0 = *(r1 + (r2 << 2))
```

to emphasize that the `lsl #2` is applied to *r2*. But the syntax is what it is, and you just have to deal with it.

The scale you're going to see the most often is `LSL` because it is what lets you convert an index into an element offset. You use `LSL #1` for a halfword index and `LSL #2` for a word index.

The `RRX` scale operation is very strange because it alters the carry flag as a side effect: The bit that rotates out of the bottom bit of the offset register becomes the new carry flag.[1]

The full menu of scaling is available only for word access. For byte and halfword access, the only available scaling operation is `LSL`, and the maximum shift amount is `#3`. For doubleword access, no scaling is permitted.

The next level of complexity is pre-indexing and post-indexing.

Pre-indexed

If you put an exclamation point after the close-bracket, then the base register is updated to contain the resulting effective address. This is called *pre-indexed* because the update occurs before the dereference. It corresponds roughly to the C preincrement operator.

```
    ldr r0, [r1, #4]!           ; r1 = r1 + 4
                                ; r0 = *r1

    ldr r0, [r1, r2, lsl #2]!   ; r1 = r1 + (r2 << 2)
                                ; r0 = *r1
```

Post-indexed

If you put the offset and scale outside the close-bracket, then the base register is used as the effective address, but the base register is then updated by the amount specified by the offset and scale. This is called *post-indexed* because the update occurs after the dereference. It corresponds roughly to the C postincrement operator.

```
    ldr r0, [r1], #4            ; r0 = *r1
                                ; r1 = r1 + 4

    ldr r0, [r1], r2, lsl #2    ; r0 = *r1
                                ; r1 = r1 + (r2 << 2)
```

If you use pre-indexing or post-indexing, then the base register cannot be the register being loaded or stored.

Some special rules kick in if the base register is *pc*. First of all, you cannot use pre-indexing or post-indexing with *pc*. Next, as we noted in the introduction, reading the *pc* register reads as the address of the instruction plus 4. On top of that, the resulting value is then rounded down to the nearest multiple of 4, in order to make it possible to load *pc*-relative words from memory.

```
    ldr     Rd, [pc, #offset]   ; load value from code segment
```

This special pattern has a special assembly pseudo-instruction:

```
    ldr     Rd, =imm32          ; load pseudo-immediate (constant stored in code
segment)
```

The assembler will first try to generate the constant in one instruction (which we will learn about next time), but if that's not possible, it will place the constant into a *literal pool* and generate a *pc*-relative `LDR` instruction to load the constant from the code segment.

The disassembler understands this convention and regenerates the literal constant in the disassembly, saving you the trouble of having to count bytes to look it up.

The assembler's convention for saying "address of label" is to put the label inside vertical bars, so `|label|` means "address of label". This shows up a lot when using the `=` pseudo-form of the `LDR` instruction. (I don't think other assemblers use this notation. It appears to be a Windows-specific convention.)

The assembler automatically emits a literal pool between subroutines, but the immediate offset cannot reach more than about 4KB. If you have a large function, you may need to help the assembler out by issuing the `LTORG` pseudo-op to tell the assembler to emit a literal pool immediately. (Of course, you want to do this at a point where the literal pool is not at risk of being executed as code!)

Okay, next time, we'll look at those one-instruction constants.

[1] Unless the operation itself modifies flags, in which case the carry comes from the result of the operation.

Raymond Chen