The ARM processor (Thumb-2), part 10: Memory access and alignment

devblogs.microsoft.com/oldnewthing/20210611-00

June 11, 2021

Raymond Chen

Accessing memory is done primarily through load and store instructions.

```
    LDR     Rd, [...]       ; load word
    STR     Rd, [...]       ; store word

    LDRD    Rd, Rd2, [...]  ; load doubleword into Rd and Rd2
    LDRH    Rd, [...]       ; load halfword, zero-extended
    LDRSH   Rd, [...]       ; load halfword, sign-extended
    LDRB    Rd, [...]       ; load byte, zero-extended
    LDRSB   Rd, [...]       ; load byte, sign-extended

    STRD    Rd, Rd2, [...]  ; store doubleword from Rd and Rd2
    STRH    Rd, [...]       ; store halfword
    STRB    Rd, [...]       ; store byte
```

You cannot have multiple updates to a register, and you cannot modify the register being stored.

```
    LDR     r0, [r0, #8]!   ; illegal: modifies r0 twice
    STR     r0, [r0, #8]!   ; illegal: modifies r0 while it is being stored
    LDRD    r0, r0, [r1]    ; illegal: modifies r0 twice
```

The doubleword load and store instructions require doubleword alignment of the effective address and do not support register-plus-register addressing modes.[1]

Misaligned memory accesses normally generate an alignment exception that traps into the kernel, which typically emulates the memory operation before returning back to user mode. Of course, trapping into kernel mode is going to be a lot slower than dealing with the misalignment in the code, so the trap is really just a backstop and shouldn't be your primary means of dealing with misaligned data.

The ARM architecture permits the operating system to put alignment enforcement into a relaxed mode, which Windows does. When alignment enforcement is relaxed, then misaligned reads and writes of a single word or halfword are fixed up automatically in the processor without generating an exception. Note, however, that the fixed-up memory operation is not atomic: You can get torn reads or writes if the memory is being accessed by another device at the same time.

I noted above that relaxed enforcement of alignment kicks in only for single words or halfwords. One source of multi-word memory access is the `LDRD` and `STRD` instructions which store a pair of registers. There are also instructions specifically designed for reading and writing multiple registers:

```
    ; load multiple registers starting at Rn
    ldm     Rn,  { registers }

    ; load multiple registers starting at Rn
    ; and update Rn
    ldm     Rn!, { registers }

    ; store multiple registers ending at Rn
    stm     Rn,  { registers }

    ; store multiple registers ending at Rn
    ; and update Rn
    stm     Rn!, { registers }
```

The load/store multiple instructions let you load or store multiple registers to a block of memory starting at *Rn*. The registers are stored with the lowest-numbered register at the lowest address, and subsequent registers in adjacent memory locations.

When loading, you can load any register except *sp*. When storing, you cannot store *pc* or *sp*.

The updating versions allow you to treat *Rn* as a stack pointer: When storing, the registers are stored *below* the address in the base register, and then the base register is decremented past the written-to bytes. When loading, the registers are loaded *from* the address in the base register, and then the base register is incremented past the read-from bytes.[2] In the updating versions, the base register may not be among the registers being loaded or stored.

The list of registers must have at least two entries. There’s no point to loading or storing zero registers, and if you wanted only one register, you could have used a regular load or store instruction. (For the updating versions, you can use a pre-indexed store and a post-indexed load.) As a courtesy, the assembler accepts `LDM` and `STM` with a single register and automatically converts it into the corresponding `LDR` or `STR` instruction for you.

There are dedicated `PUSH` and `POP` instructions for the common case where *sp* is the base register.

```
    ; push multiple registers
    push    { registers }       ; stm sp!, { registers }

    ; pop multiple registers
    pop     { registers }       ; ldm sp!, { registers }
```

For some reason, there is a separate 32-bit encoding for the case of pushing or popping a single register, even though it could have been done with a pre-indexed store (push single register) or post-indexed load (pop single register). I'm guessing that this is a case of offloading work to the compiler: Having a dedicated instruction for a common special case makes it easier to recognize in the CPU.

The CPU itself provides a compact 16-bit encoding for the case where all of the registers being pushed or popped are low.

You cannot pop the *sp* register, because that would create two writes to the *sp* register in a single instruction, one from the pop and one from incrementing the *sp* register. (Technically, the processor lets you do it, but the resulting value in *sp* is architecturally unpredictable.)

I noted some time ago that anybody who writes `#pragma pack(1)` may as well just wear a sign on their forehead that says "I hate RISC". You can see this happening on ARM when it wants to perform block copies.

```
struct S
{
    int a, b, c, d;
};

// aligned copy
S* p = ...; // assume in register r0
S* q = ...; // assume in register r1
*p = *q;

    ; copy four words using multi-register load/store
    ldm     r1, {r2-r5}
    stm     r0, {r2-r5}
```

Since the structure is 4-byte aligned, the memory can be copied in two instructions by using the multi-word load and store instructions.

```
// unaligned copy
__unaligned S* p = ...; // assume in register r0
__unaligned S* q = ...; // assume in register r1
*p = *q;

    ; copy four words one at a time
    ldr     r2, [r1]
    str     r2, [r0]
    ldr     r2, [r1, #4]
    str     r2, [r0, #4]
    ldr     r2, [r1, #8]
    str     r2, [r0, #8]
    ldr     r2, [r1, #12]
    str     r2, [r0, #12]
```

If the structure is unaligned, then the compiler cannot use the multi-word load and store instructions, because they will trap if the pointers are misaligned. Instead, the values have to be copied one word at a time.

So those are the regular load/store instructions. Next time, we'll look at the instructions for atomic memory access.

[1] In classic ARM, *Rd* must be even, and *Rd2* must be one greater than *Rd*. Thumb-2 removes this restriction and lets you target any pair of registers. (Well, *almost* any pair. Some registers are disallowed, like *sp* and *pc*.)

[2] In classic ARM, there are eight versions of the updating multi-word instructions:

`LD`	`M`	`I`	`B`
`ST`		`D`	`A`

You can choose to load (`LD`) or store (`ST`), you can choose whether the base register is incremented (`I`) or decremented (`D`), and you can choose whether the effective address adjustment occurs before (`B`) or after (`A`) the memory is accessed.

Classic ARM also provides alternate mnemonics for these operations based not on what the instruction literally does, but rather describing the usage pattern.

Opcode	Meaning	Equivalent to
`STMFD` `LDMFD`	Full Descending	`STMDB` `LDMIA`
`STMED` `LDMED`	Empty Descending	`STMDA` `LDMIB`

`STMFA` `LDMFA`	Full Ascending	`STMIB` `LDMDA`
`STMEA` `LDMEA`	Empty Ascending	`STMIA` `LDMDB`

The "Descending" version is for managing a stack that grows downward, and the "Ascending" version grows upward.

The "Full" version has the register pointing to the item most recently stored on the stack, and the "Empty" version has the register pointing to the place where the next item would go.

In practice, nearly everyone uses Full Descending (`STMDB` and `LDMIA`), and Thumb-2 abandoned the other variations. Note that the underlying semantics are still available for single-register loads and stores in Thumb-2. You just lose the ability to do multi-register operations.

Raymond Chen

Follow