The ARM processor (Thumb-2), part 1: Introduction

devblogs.microsoft.com/oldnewthing/20210531-00

May 31, 2021

Raymond Chen

I've run out of historical processors that Windows supported, so I'm moving on to processors that are still in support. First up in this series is 32-bit ARM.

As with all of these series, I'm focusing on how Windows 10[1] uses the processor in user mode, with particular focus on the instructions you are most likely to encounter in compiler-generated code.

The classic ARM processor generally follows the principles of Reduced Instruction Set Computing (RISC): It has fixed-length instructions, a large uniform register set, and the only operations on memory are loading and storing. However, Windows doesn't use the ARM processor in classic mode, so some of the above statements aren't true any more.

Windows uses the ARM in a mode known as Thumb-2 mode.[2] In Thumb-2 mode, some classic features are not available, such as most forms of predication. The Thumb-2 mode instruction encoding is variable-length, with a mix of 16-bit instructions and 32-bit instructions. Every instruction is required to begin on an even address, but 32-bit instructions are permitted to straddle a 4-byte boundary.

In addition to classic ARM mode, Thumb mode, and Thumb-2 mode, there are also Jazelle mode (which executes Java bytecode) and ThumbEE mode. I'm not going to cover them at all in this series, since Windows doesn't use them. **From now on, I'm talking only about Thumb-2 mode**.

The ARM architecture permits little-endian or big-endian operation. Windows runs the processor in little-endian mode and disables the `SETEND` instruction, so you can't switch to big-endian even if you tried.

The architectural terms for data sizes are

Term	Size
byte	8 bits

halfword	16 bits
word	32 bits
doubleword	64 bits

The ARM instruction set has 16 general-purpose integer registers, each 32 bits wide, and formally named *r0* through *r15*. They are conventionally used as follows:

Register	**Mnemonic**	**Meaning**	**Preserved?**
r0	(*a1*)	argument 1 and return value	No
r1	(*a2*)	argument 2 and second return value	No
r2	(*a3*)	argument 3	No
r3	(*a4*)	argument 4	No
r4	(*v1*)		Yes
r5	(*v2*)		Yes
r6	(*v3*)		Yes
r7	(*v4*)		Yes
r8	(*v5*)		Yes
r9	(*v6*)		Yes
r10	(*v7*)		Yes
r11	*fp* (*v8*)	frame pointer	Yes
r12	(*ip*)	intraprocedure call scratch	Volatile
r13	*sp*	stack pointer	Yes
r14	*lr*	link register	No
r15	*pc*	program counter	N/A

The names in parentheses are used by some assemblers, but Microsoft's toolchain doesn't use those names. Some operating systems use *r9* for special purposes (usually as a table of contents/gp or a thread-local pointer), but Windows does not assign it any special meaning. On Windows, it is available for general use, as long as the value is preserved across calls.

The meanings of the last three registers (*sp*, *lr*, *pc*) are architectural.[3] The rest are convention. We'll learn more about register conventions later.

The processor enforces 4-byte alignment for the *sp* register. Operations which misalign the stack result in unpredictable behavior.[4] Windows requires further that the stack be 8-byte aligned at function call boundaries.

The ARM is notable for putting the program counter in the general-purpose register category, a feature which has been called "overly uniform" by noted processor architect Mitch Alsup. The program counter register reads as the address of the current instruction plus four: The +4 is due to the pipelining of the original ARM implementation: By the time the pipeline gets to fetching the value of the register, the CPU has already advanced the instruction pointer four bytes. Even though later implementations of ARM have deeper pipelining, they continue to emulate the original pipelining for the purpose of reading from the program counter.[5] Writing to the program counter acts like a jump instruction: The next instruction to be executed is the one at the address you wrote.

This magic treatment of the program counter register is a bit mind-blowing when you first encounter it.

Floating point and SIMD support (Neon) is optional in the ARM architecture, but Windows requires both. This means that you also have 32 double-precision (64-bit) floating point registers, which can also be split into 64 single-precision (32-bit) floating point registers.

Registers		Preserved?
s0 + s1	*d0*	No
s2 + s3	*d1*	
⋮	⋮	
s14 + s15	*d7*	
s16 + s17	*d8*	Yes
⋮	⋮	
s30 + s31	*d15*	
s32 + s33	*d16*	No
⋮	⋮	
s62 + s63	*d31*	

The ARM does not have branch delay slots. You can breathe a sigh of relief.

The flags register is formally known as the Application Program Status Register (APSR). These flags are available to user mode:

Mnemonic	Meaning	Notes
N	Negative	Set if the result is negative
Z	Zero	Set if the result is zero
C	Carry	Multiple purposes
V	Overflow	Signed overflow
Q	Saturation	Accumulated overflow
GE[n]	Greater than or equal to	4 flags (SIMD)

The overflow flag records whether the most recent operation resulted in signed overflow. The saturation flag is used by multimedia instructions to accumulate whether any overflow occurred since it was last cleared. The GE flags record the result of SIMD operations. Flags are not preserved across calls.

Under the Windows ABI, there is an 8-byte red zone beneath the stack pointer. However, you'll never see the compiler using it because the red zone is reserved. It's there for intrusive profilers.

Intrusive profilers inject code into your binary to update hit counts. The ARM does not have an absolute addressing mode; access to memory is always indirect through registers. Therefore, the profiler needs to be able to "borrow" a register in order to access memory, and it does so by saving the current contents of two temporary registers to the red zone. This frees up just enough registers to be able to update profiling information.

```
    str     r12, [sp, #-4]  ; save r12 into the red zone
    str     r0,  [sp, #-8]  ; save r0  into the red zone

    ; We can now use r12 and r0 to update profiling statistics.
    ... do profiling stuff with r12 and r0 ...

    ; All done. Restore the registers we borrowed.
    ldr     r0,  [sp, #-8]  ; recover r0  from the red zone
    ldr     r12, [sp, #-4]  ; recover r12 from the red zone
```

[1] Windows CE also supported ARM, it supported both Thumb-2 mode and classic ARM, so its ABI was different. This series covers the Windows 10 ABI.

[2] Thumb-2 is an expansion of an earlier instruction set known unsurprisingly as Thumb. (Exercise: Why didn't they call it Thumb-1?) The idea of using a 16-bit instruction set came from the SuperH, and ARM licensed it from Hitachi for use in Thumb mode.

[3] The use of *r13* as the stack pointer is not architectural in classic ARM, but it is architectural in Thumb-2. Doing so frees up space in the tight 16-bit instruction encoding space.

[4] In processor-speak, *unpredictable* means that the processor can perform any operations it likes, provided they are permissible at the current privilege level. For example, an *unpredictable* operation in user mode can set all registers to 42. But it cannot perform privileged operations, and the result cannot be dependent upon state that is not visible to user mode.

[5] As with branch delay slots, the +4 effect of reading from the program counter is another example of how a clever hack in a processor's original architecture turns into a compatibility constraint for future implementations.

Raymond Chen

Follow