The AArch64 processor (aka arm64), part 25: The ARM64EC ABI

devblogs.microsoft.com/oldnewthing/20220830-00

August 30, 2022

Raymond Chen

I mentioned that Windows has a second ABI for AArch64 named ARM64EC. The "EC" stands for "Emulation Compatible", and its purpose is to make it easier for ARM64 and x86-64 code to coexist within a single process.

The idea here is that you have a program written for x86-64, and you're porting it to 64-bit ARM, but you can't or don't want to do a complete port. You might not be able to do a complete port because some of the libraries you're using are available only for x86-64 and x86-32. And you may not want to do a complete port because the performance of the x86-64 emulator on 64-bit ARM systems is good enough for most of your usage scenarios, but there are a few performance-critical functions that you want to recompile as 64-bit ARM to avoid the emulation overhead. Or maybe your program has a plug-in model, and you want to be able to load plug-ins that were written for x86-64. Those plug-ins will run under emulation, but the rest of your program runs natively as AArch64.

What you do is you port some of your program to 64-bit ARM and leave the rest in x86-64. The x86-64 parts run in the emulator, and the AArch64 parts run natively.

The design of ARM64EC aligns the AArch64 conventions to match the x86-64 conventions, in order to minimize the mismatch at the architecture boundaries.

One way to reduce the mismatch is to assign each AArch64 register a buddy x86-64 register. The AArch64 register uses its buddy's slot in the `CONTEXT` structure, so that an x86-64 `CONTEXT` can be used to hold either an x86-64 context or an AArch64 context.

AArch64	x86-64	Notes
x0	*rcx*	Function parameter 1
x1	*rdx*	Function parameter 2
x2	*r8*	Function parameter 3

x3	*r9*	Function parameter 4
x4	*r10*	
x5	*r11*	
x6	*fp(1)*	Bottom 64 bits of fp(1)
x7	*fp(2)*	Bottom 64 bits of fp(2)
x8	*rax*	Return value
x9	*fp(3)*	Bottom 64 bits of fp(3)
x10	*fp(4)*	Bottom 64 bits of fp(4)
x11	*fp(5)*	Bottom 64 bits of fp(5)
x12	*fp(6)*	Bottom 64 bits of fp(6)
x13		Off-limits
x14		Off-limits
x15	*fp(7)*	Bottom 64 bits of fp(7)
x16	*fp(0..3)*	High 16 bits of fp(0) to f(3)
x17	*fp(4..7)*	High 16 bits of fp(4) to f(7)
x18		TEB
x19	*r12*	
x20	*r13*	
x21	*r14*	
x22	*r15*	
x23		Off-limits
x24		Off-limits
x25	*rsi*	
x26	*rdi*	
x27	*rbx*	
x28		Off-limits

fp	*rbp*	
lr	*fp(0)*	Bottom 64 bits of fp(0)
sp	*rsp*	
pc	*rip*	
Flags	Flags	
v0..v15	*xmm0..xmm15*	
v16..v31		Off-limits

There are some sneaky tricks happening here.

The classic 8087 floating point registers are 80-bit values, so they end up split into chunks. The lower-order 64 bits map to the buddy AArch64 register, and the upper 16 bits are gathered in groups of four to form a 64-bit value that gets stored in a helper register.

The AArch64 integer register mappings are chosen so that they have the same register preservation policies as their x86-64 buddies. For example, *x19* is a preserved register in the classic ARM calling convention, and its buddy *r12* is a preserved register in the x86-64 calling convention.

There are a few extra AArch64 registers that do not have an x86-64 buddy. These registers are off-limits to ARM64EC code. Do not use them, because their values are not preserved across context switches or asynchronous exceptions. (There’s nowhere to save them!)

Notice that the classic AArch64 calling convention uses *r0* to hold both the first integer parameter as well as the return value, whereas x86-64 uses different registers for those two purposes. This means that the match is imperfect, and we’ll have to do some extra work later to get the return values to line up.

Okay, so that aligns the registers. The next rule is that ARM64EC follows x86-64 data alignment rules. This makes structures binary-compatible between the two.

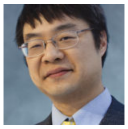
A third rule is that when an ARM64EC function calls an x86-64 function or vice versa, the call goes through a “thunk” that manages the last bit of mismatch between the two architectures. For example, the exit thunk for returning from x86-64 code to AArch64 code will move *r8* (buddy to *rax*) to *r0*, so that the return value is in a place that AArch64 code expects.

That’s the whirlwind tour of ARM64EC. There’s a lot more, but those are the parts you will notice when you’re debugging compiler-generated code. For even more details about ARM64EC, you can read Understanding Arm64EC ABI and assembly code on docs.microsoft.com.

Bonus chatter: When you compile your code as ARM64EC, the architecture preprocessor symbols will say that you are compiling for x86-64. There’s a reason for this. See the linked article for the answer.

Bonus reading: Arc64EC is now officially supported by the Microsoft Visual C++ compiler.

Raymond Chen



Follow