Arm64EC: Microsoft's Emulation Frankenstein

Peter Cawley (@corsix), 4th February 2024, FOSDEM

Why am I *here*?

LuaJIT / LuaJIT

Type / to search

Code | Issues 42 | Pull requests 15 | Discussions | Actions | Security | Insights | Settings

Support for the ARM64EC ABI on Windows ARM64 #1096

Closed — **kobykahane** opened this issue on Sep 23, 2023 · 10 comments

kobykahane commented on Sep 23, 2023 • edited by corsix

LuaJIT recently introduced support for Windows ARM64 targets, based on the original Windows ARM64 ABI.

In Windows 11, an additional ABI was introduced, ARM64EC. This ABI allows mixing native ARM64 code and emulated x64 code in the same process. This is useful for projects that want to support, e.g., legacy x64-compiled plugin DLLs on ARM64 devices. It would be useful if LuaJIT could be built for ARM64EC, so it could be consumed by hosting programs with such requirements.

Presently only MSVC supports this ABI, although there's been some initial work in clang as well. To target this ABI, the `/arm64EC` option is passed in the cl.exe command line, or `/machine:arm64ec` to lib.exe or link.exe. Details of the ABI are described in Overview of ARM64EC ABI conventions
and Understanding Arm64EC ABI and assembly code. Some key issues that appear to impact a potential port of LuaJIT:

- A one-to-one mapping between the ARM64 processor context and the emulated x64 processor context is defined. To facilitate this, the ABI bans certain ARM64 registers from being used, including x13, x14, x23, x24, x28 and v16-v31. The interpreter and the JIT would need to avoid using the banned registers.
- When the JIT allocates executable memory, it needs to do so with `VirtualAlloc2` and specify `MEM_EXTENDED_PARAMETER_EC_CODE`, so the system knows the dynamically generated code is ARM64 code and not x64 code.
- When a call is made to an externally provided function pointer (i.e., a lua_CFunction or via FFI), in the general case the provided function might be native ARM64 code or it might be x64 code that needs to be run by the emulator, so the indirect call needs to be made via a call checker provided by the OS, e.g., `__os_arm64x_check_icall`. The call checkers are automatically used by compiler-generated code, but need to be invoked explicitly by assembly code.

Disclaimers

- I do not work for Microsoft.
- I do not work for Intel.
- I do not work for Arm.
- I am not Mike Pall.
- Views are my own.

Agenda

1. General landscape of emulating x64 on arm64. ←
2. What is Arm64EC?
3. Lessons learnt porting LuaJIT to Arm64EC.

Emulation main loop, 101

Turn:

```
sub eax, dword ptr [rsp + 259]
```

Into:

```
add  x16, sp, #259
ldr  w16, [x16]
subs w0 , w0, w16
```

“Just” repeat this for every instruction.

Emulation main loop, 201

Turn:

```
sub eax, dword ptr [rsp + 259]
```

Into:

```
// TODO: memory ordering?
add  x16, sp, #259
ldr  w16, [x16] // TODO: MMU / devices?
subs w0 , w0, w16
// TODO: fixup flags?
```

Flags

x64:	AF	CF	OF	PF	SF	ZF
	↕	↕ **inv?**	↕	↕	↕	↕
arm64:	**?**	C	V	**?**	N	Z

Existing ways to emulate x64 on arm64

qemu-system, qemu-user

jart/blink, FEX-Emu, Box64

Rosetta 2

Apple’s pitch to developers

1. Target x64, get fast emulation on custom hardware.
2. Port to arm64, get even faster native execution.

Microsoft's less appealing pitch

1. Target x64, get slow emulation.
2. Can't port to arm64 if closed-source libraries / plugins.

Performance example

LuaJIT benchmark suite, on M1 Max MacBook Pro:

- macOS native arm64: 33 seconds.
- macOS Rosetta 2 x64: 44 seconds (+33%).

Same hardware, Windows on Arm in hypervisor VM:

- Windows native arm64: 37 seconds.
- Windows emulated x64: 106 seconds **(+186%)**.

Option 1 is too slow, and option 2 is impossible, but maybe option 1½ is both fast and possible?

Agenda

1. General landscape of emulating x64 on arm64.
2. What is Arm64EC? ←
3. Lessons learnt porting LuaJIT to Arm64EC.

Let an application mix arm64 and x64, with cheap interop between native arm64 parts and emulated x64 parts.

And thus Arm64EC was born.

Cheap arm64/x64 interop means:

1. Shared virtual address space.
2. Shared data structure layouts.
3. Shared call stacks.
4. Mode switch only at function call or return.
5. Adjust calling conventions a little bit.

Shared virtual address space

1. Executable memory needs tagging as arm64 or x64 (OS maintains a bit per page, code can query for it).

2. Emulated x64 code issues lots of memory barriers.

3. Native arm64 code issues memory barriers where required (care required by the porting programmer).

Shared data structure layouts (1)

```
struct foo {
  long x;
  double y;
  void* p;
  void (*fn)(void);
};
```

Shared data structure layouts (2)

```
struct my_exception_state {
  jmp_buf on_error_jump_to;
};
```

Shared data structure layouts (3)

```
struct my_thread_state {
  CONTEXT ctx;
};
```

Making jmp_buf, CONTEXT, etc. compatible

	Emulated x64	arm64
64-bit GPRs	16 (rax, rcx, …, r15)	32 (x0 … x30, sp)
Special registers	rip, rflags, mxcsr, gs	pc, pstate, fpcr, fpsr
128-bit FPRs	16 (xmm0 … xmm15)	32 (v0 … v31)
80-bit FPRs	8 (the x87 stack)	0

Casualties: x13, x14, x23, x24, x28, v16 … v31.

Shared call stacks

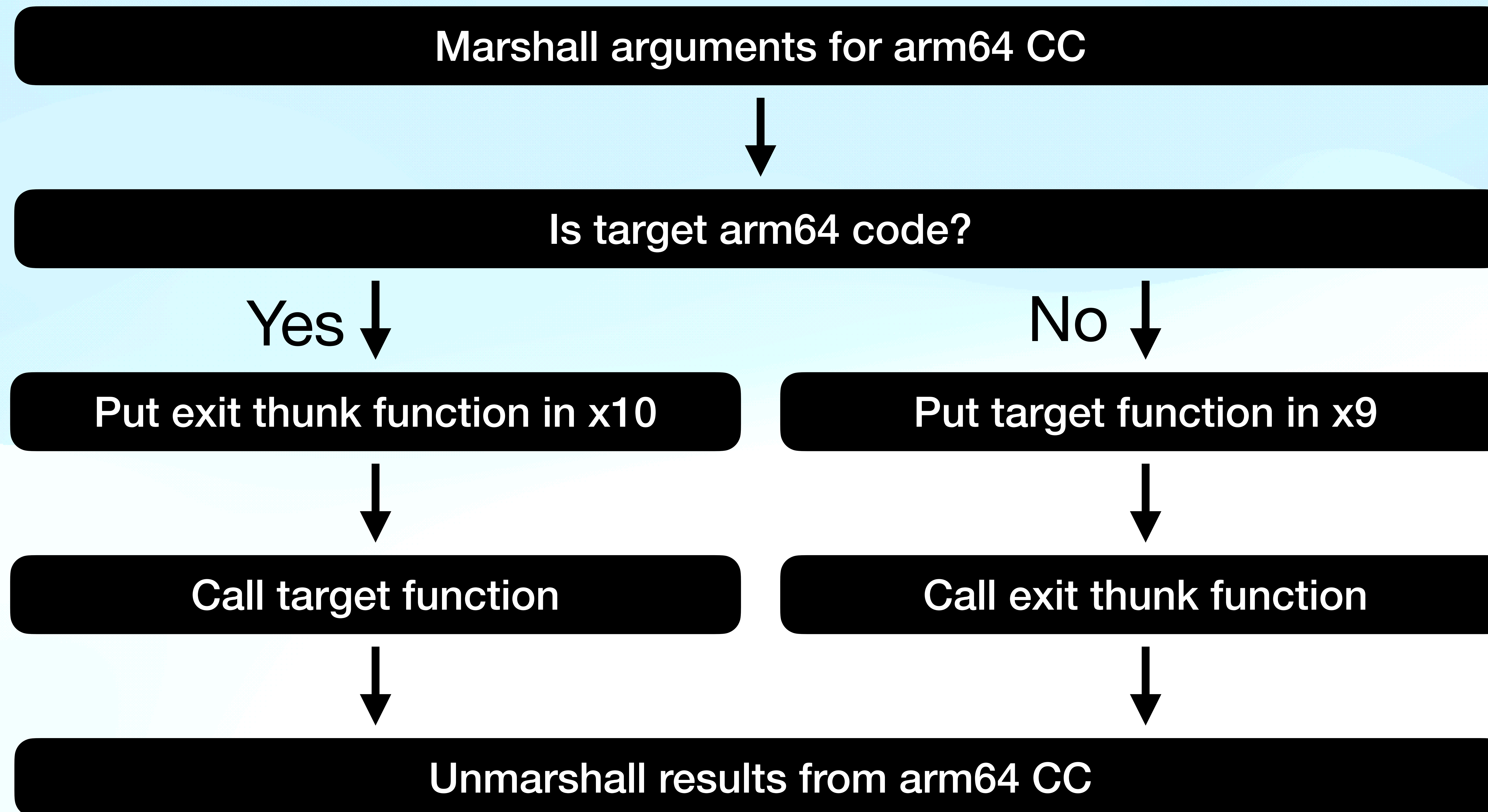
1. arm64 has LR, x64 expects return address on stack.
2. arm64 requires SP aligned to 16 bytes in load/store, x64 merely *strongly recommends* 16 byte alignment.

So some fixup work required on mode switches.

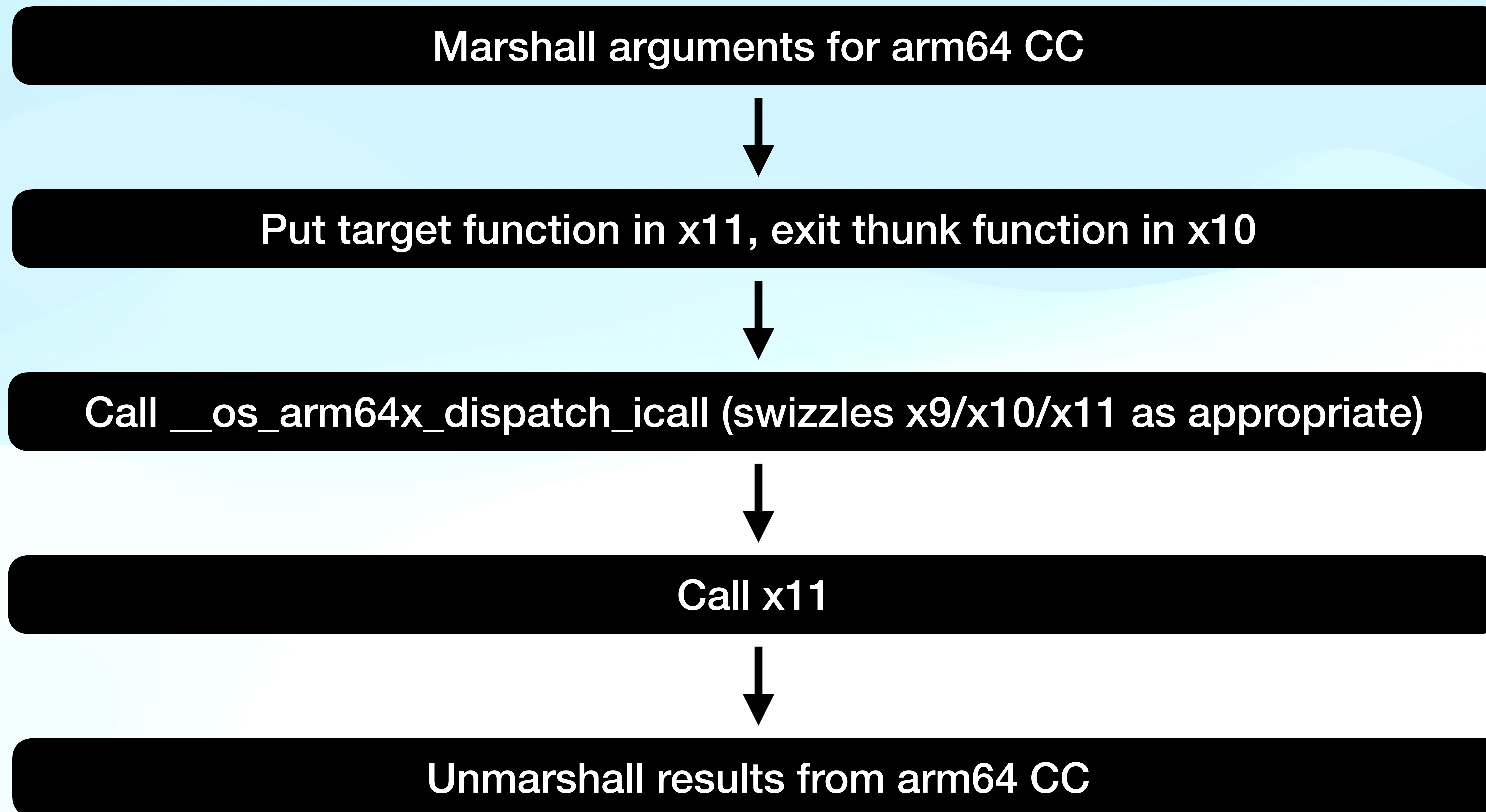
About those mode switches

1. Calling conventions mostly unchanged (ex. varargs).
2. But the arm64 and x64 conventions are different, so some conversion work required at mode switches.
3. Exact conversion logic depends on the types involved.
4. arm64 code responsible for doing most of the work.

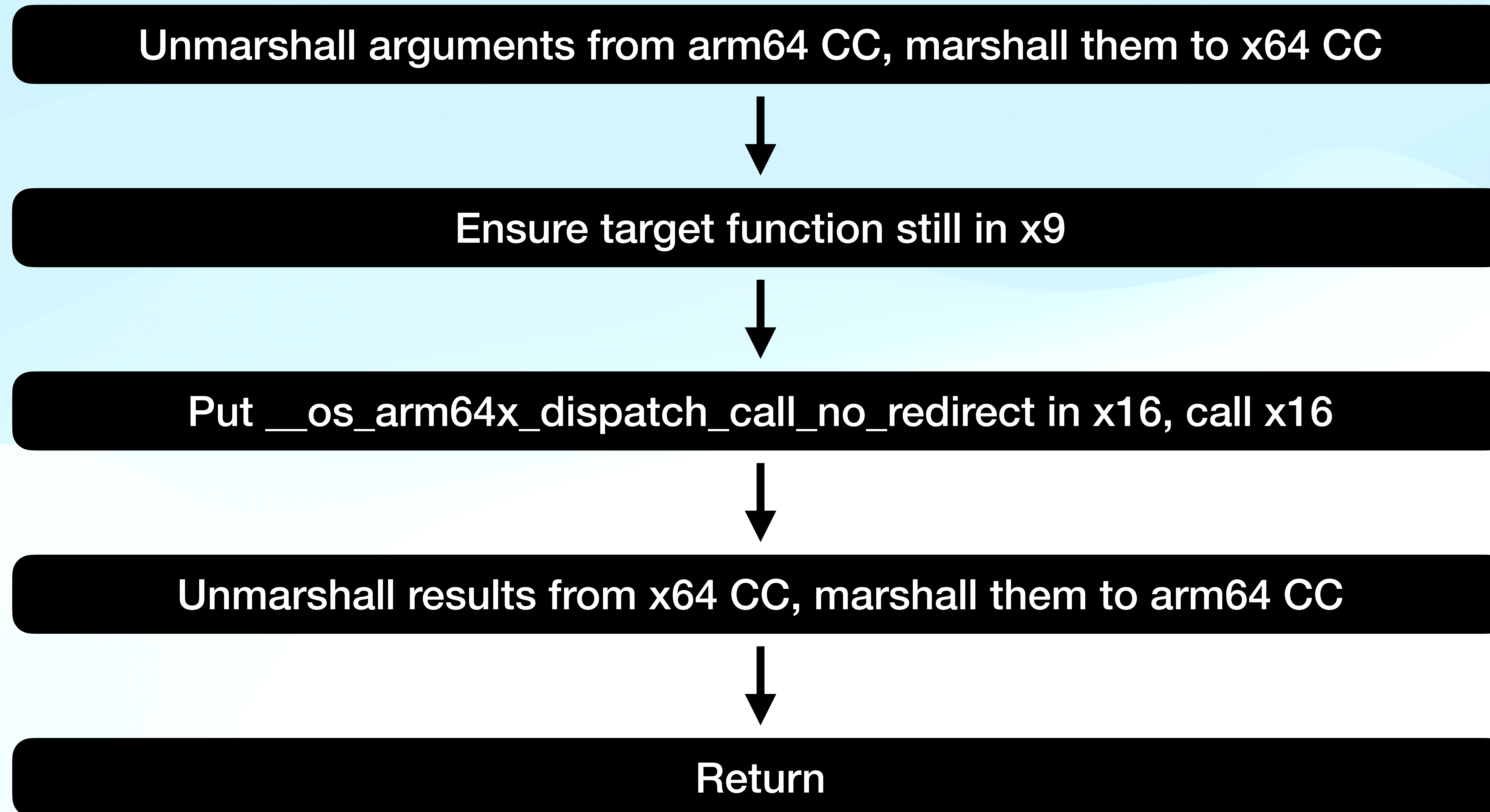
Function calls in Arm64EC code



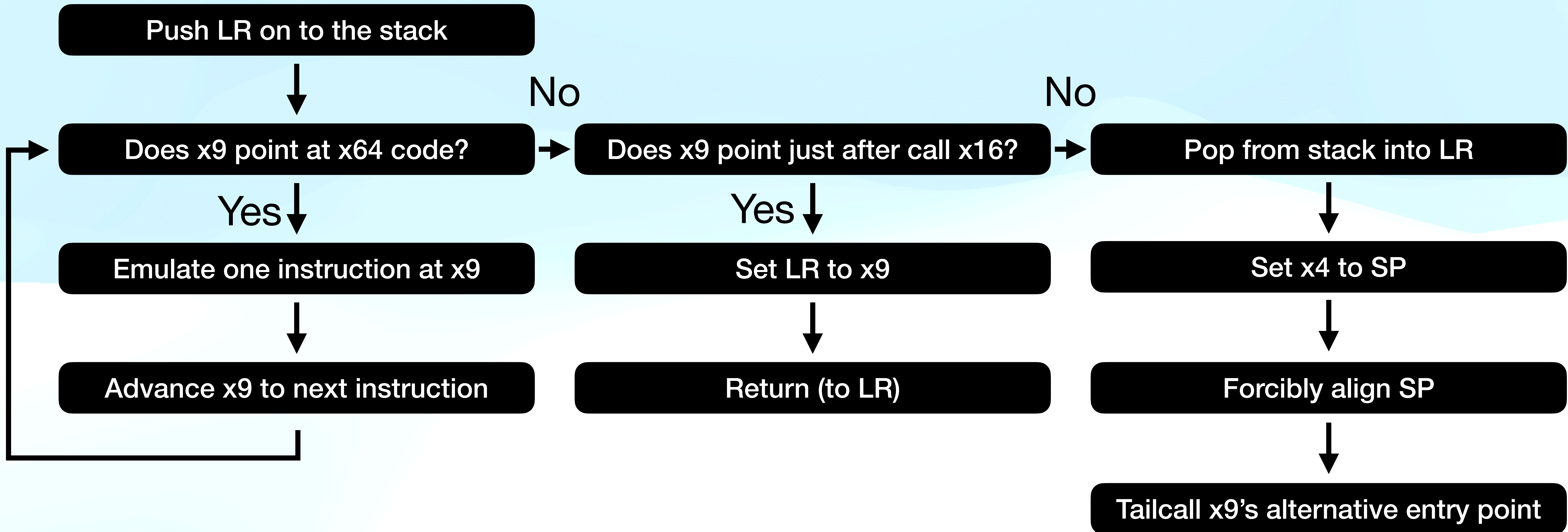
Function calls in Arm64EC code, with helper



Contents of an exit thunk function



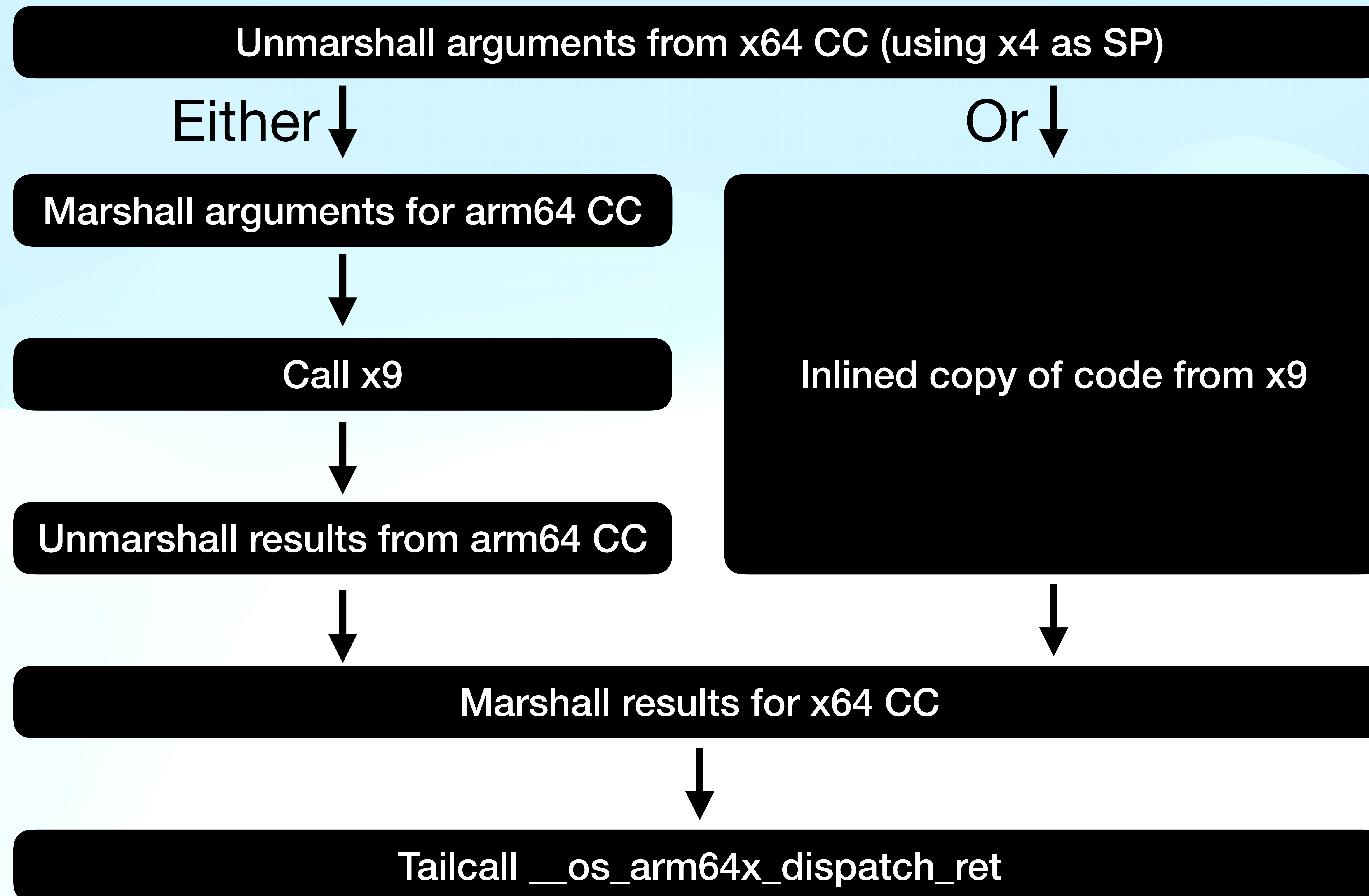
Contents of __os_arm64x_dispatch_call_no_redirect



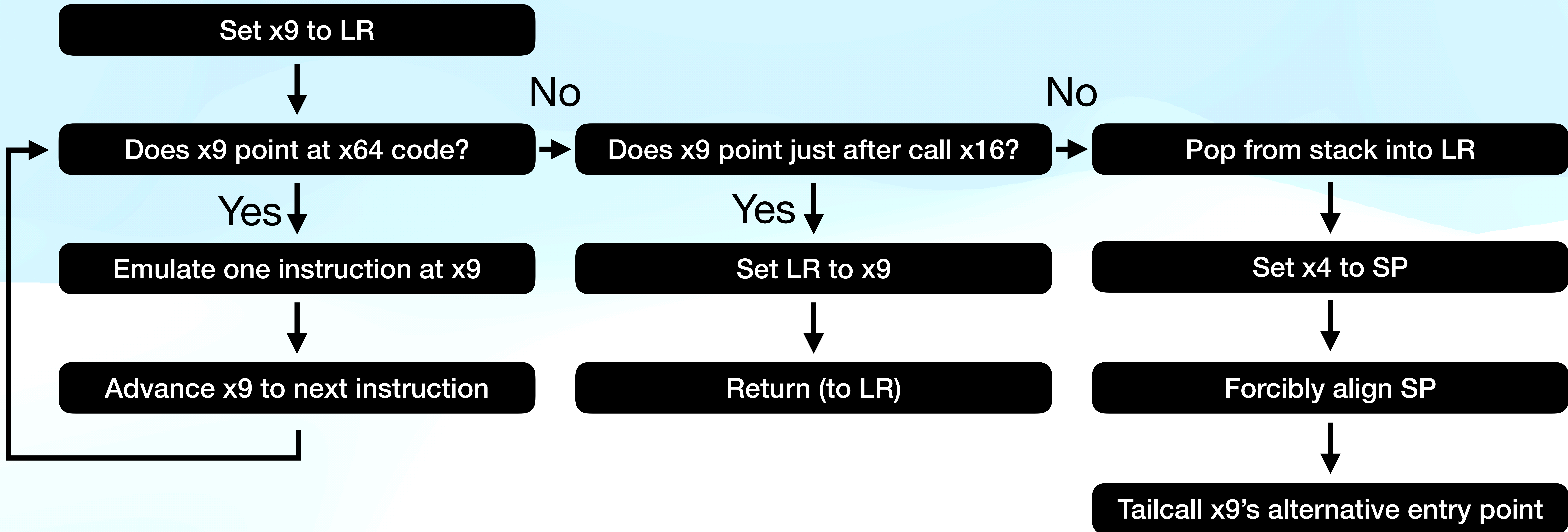
Alternative entry points?

- Every arm64 function that could be called from x64 code needs an alternative entry point, for when the caller was x64.
- The alternative entry point is arm64 code for handling the mode switch.
- Offset of alternative entry point specified as 32-bit int in the 32 bits immediately before the function.

Contents of an alternative entry point



Contents of __os_arm64x_dispatch_ret



Agenda

1. General landscape of emulating x64 on arm64.
2. What is Arm64EC?
3. Lessons learnt porting LuaJIT to Arm64EC. ←

Impact on LuaJIT: losing 5 GPRs & 16 FPRs

1. Interpreter doesn’t care about losing x13-14, v16-31.
2. Losing x23-24 mitigated via some ~zero cost tricks.
3. Losing x28 really annoying, requires spills/restores.
4. Easy to make JIT compiler avoid the lost registers, though likely impact on speed of generated code.

Impact on LuaJIT: mode switches

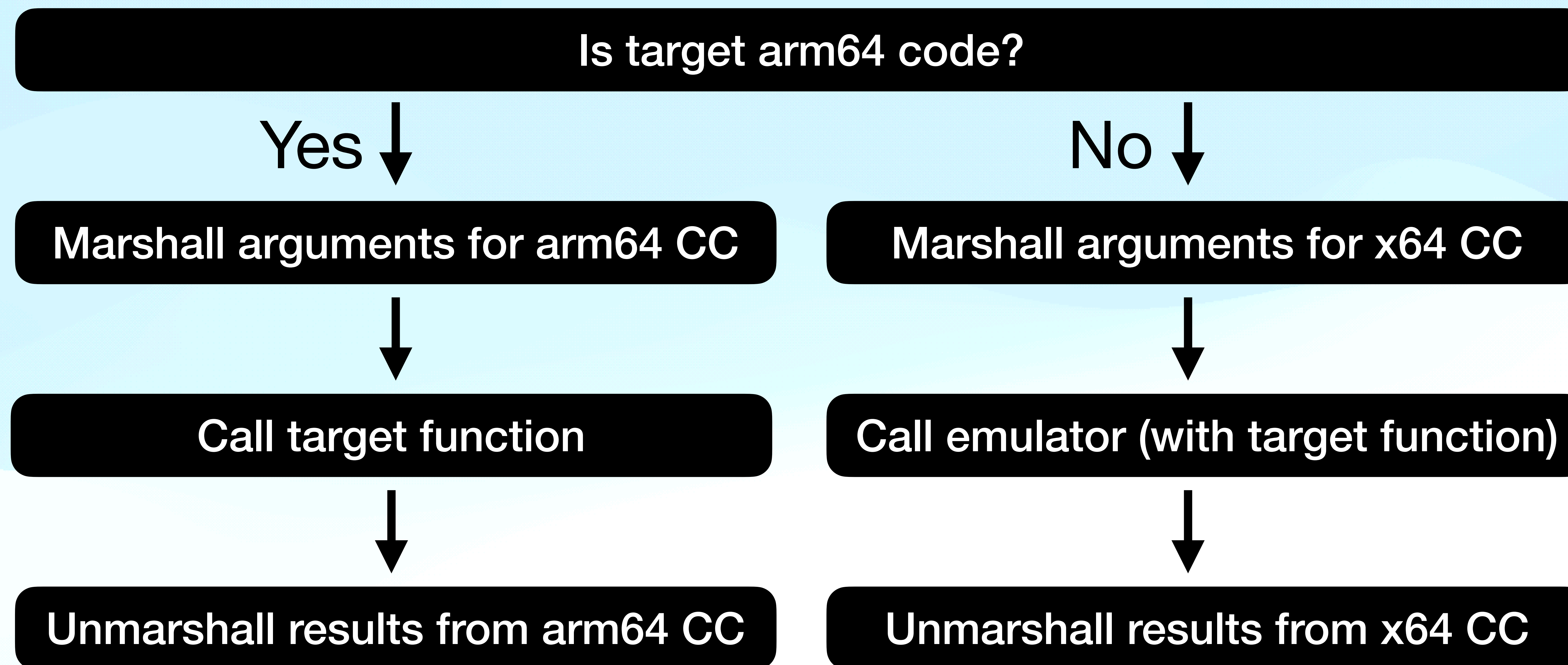
C compiler handles most of them. Three arm64 to x64 it doesn't:

1. Interpreter opcode for calling Lua API C functions.
2. Interpreted FFI calls to arbitrary functions.
3. JIT-compiled FFI calls to functions with "simple" types.

One x64 to arm64 it doesn't:

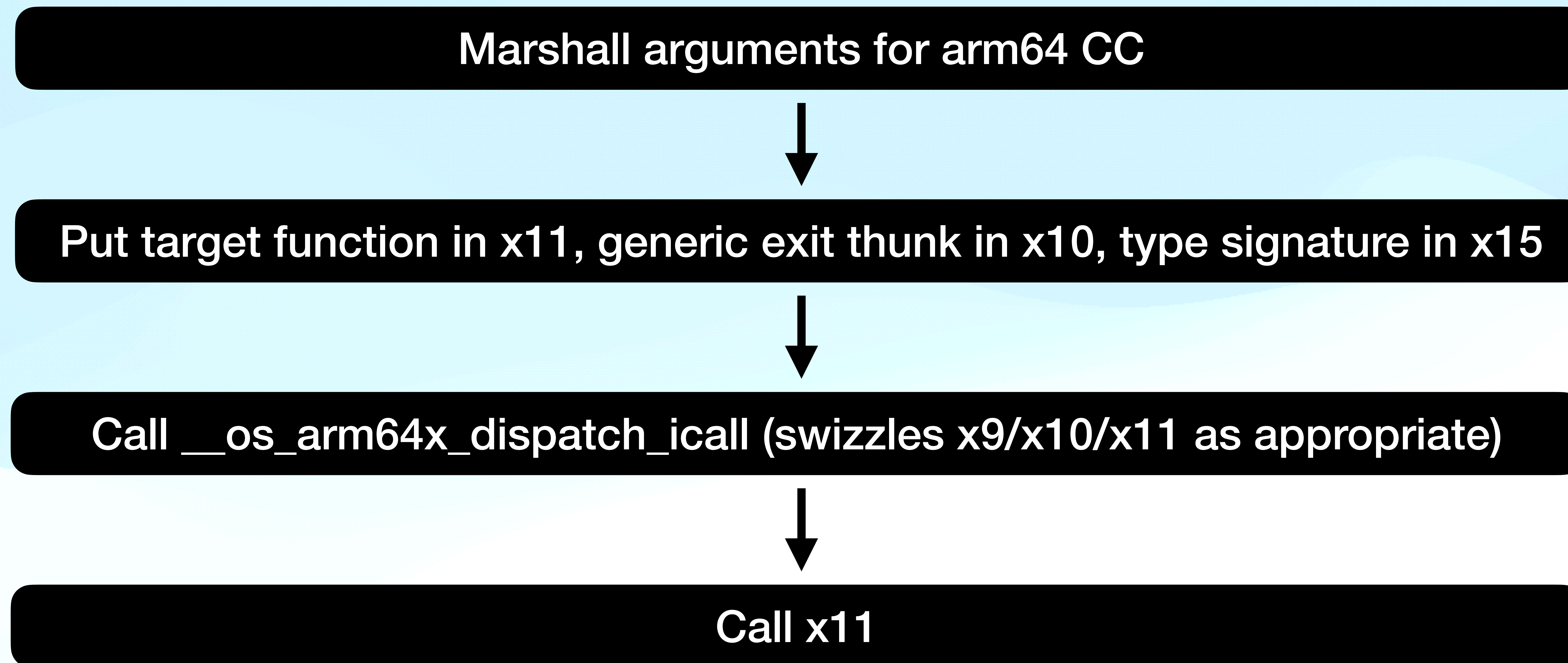
1. FFI callbacks with "simple" types.

How Arm64EC LuaJIT interprets FFI calls



Works fine in practice, unless target is a typeless arm64 function relying on presence of x10. Could fix this, but the need has not yet arisen.

How Arm64EC LuaJIT compiles FFI calls



Works fine in practice, unless target is a typeless arm64 function relying on presence of x10 that also happens to trash x15.

Impact on LuaJIT: performance

Windows on Arm VM under hypervisor:

- Windows native arm64: 37 seconds.
- Windows emulated x64: 106 seconds (+186%).
- Windows Arm64EC: 38 seconds (+3%).

Bonus problem: function hooking

1. Linux has LD_PRELOAD.
2. macOS / iOS have DYLD_INSERT_LIBRARIES.
3. Windows has … ad-hoc x64 machine code patching.

Casualty: Wrap public arm64 functions in an x64 shell that does nothing except a tail call to the arm64 code. __os_arm64x_dispatch_icall can skip over the shell.