Documentation

Article

Writing ARM64 code for Apple platforms

Create 64-bit ARM assembly language instructions that adhere to the application binary interface (ABI) that Apple platforms support.

Overview

The ARM architecture defines rules for how to call functions, manage the stack, and perform other operations. If part of your code includes ARM assembly instructions, you must adhere to these rules in order for your code to interoperate correctly with compiler-generated code. Similarly, if you write a compiler, the machine instructions you generate must adhere to these rules. If you don't adhere to them, your code may behave unexpectedly or even crash.

Apple platforms diverge from the standard 64-bit ARM architecture in a few specific ways. Apart from these small differences, iOS, tvOS, and macOS adhere to the rest of the 64-bit ARM specification. For information about the ARM64 specification, including the Procedure Call Standard for the ARM 64-bit Architecture (AArch64), go to https://developer.arm.com.

Respect the purpose of specific CPU registers

The ARM standard delegates certain decisions to platform designers. Apple platforms adhere to the following choices:

- The platforms reserve register x18. Don't use this register.
- The frame pointer register (x29) must always address a valid frame record. Some functions — such as leaf functions or tail calls — may opt not to create an entry in this list. As a result, stack traces are always meaningful, even without debug information.

Handle data types and data alignment properly

Some fundamental types of the C language have slightly different implementations:

- The `wchar_t` type is 32 bit and is a signed type.
- The `char` type is a signed type.
- The `long` type is 64 bit.
- The `__fp16` type uses the IEEE754-2008 format, where applicable.
- The `long double` type is a double precision IEEE754 binary floating-point type, which makes it identical to the `double` type. This behavior contrasts to the standard specification, in which a `long double` is a quad-precision, IEEE754 binary, floating-point type.

The following table lists the integer data types, their sizes, and their natural alignments on Apple platforms.

Data type	Size (in bytes)	Natural alignment (in bytes)
`BOOL`, `bool`	1	1
`char`	1	1
`short`	2	2
`int`	4	4
`long`	8	8
`long long`	8	8
pointer	8	8
`size_t`	8	8
`NSInteger`	8	8
`CFIndex`	8	8
`fpos_t`	8	8
`off_t`	8	8

Respect the stack's red zone

The ARM64 red zone consists of the 128 bytes immediately below the stack pointer. Apple platforms don't modify these bytes during exceptions. User-mode programs can rely on the bytes below the stack pointer to not change unexpectedly, and can potentially make use of the space for local variables.

> **Note**
>
> If a function calls itself, the caller must assume that the callee modifies the contents of its red zone. The caller must therefore create a proper stack frame.

Pass arguments to functions correctly

The stack pointer on Apple platforms follows the ARM64 standard ABI and requires 16-byte alignment. When passing arguments to functions, Apple platforms diverge from the ARM64 standard ABI in the following ways:

- Function arguments may consume slots on the stack that are not multiples of 8 bytes. If the total number of bytes for stack-based arguments is not a multiple of 8 bytes, insert padding on the stack to maintain the 8-byte alignment requirements.
- When passing an argument with 16-byte alignment in integer registers, Apple platforms allow the argument to start in an odd-numbered `xN` register. The standard ABI requires it to begin in an even-numbered `xN` register.
- The caller of a function is responsible for signing or zero-extending any argument with fewer than 32 bits. The standard ABI expects the callee to sign or zero-extend those arguments.
- Functions may ignore parameters that contain empty struct types. This behavior applies to the GNU extension in C and, where permitted by the language, in C++. The AArch64 documentation doesn't address the issue of empty structures as parameters, but Apple chose this path for its implementation.

The following example illustrates how Apple platforms specify stack-based arguments that are not multiples of 8 bytes. On entry to the function, `s0` occupies one byte at the current stack pointer (`sp`), and `s1` occupies one byte at `sp+1`. The compiler still adds padding after `s1` to satisfy the stack's 16-byte alignment requirements.

```
void two_stack_args(char w0, char w1, char w2, char w3, char w4, char w5, char w
```

The following example shows a function whose second argument requires 16-byte alignment. The standard ABI requires placing the second argument in the `x2` and `x3` registers, but Apple platforms allow it to be in the `x1` and `x2` registers.

```
void large_type(int x0, __int128 x1_x2) {}
```

Update code that passes arguments to variadic functions

For functions that contain a variable number of parameters, Apple initializes the relevant registers (Stage A) and determines how to pad or extend arguments (Stage B) as usual. When it's time to assign arguments to registers and stack slots, Apple platforms use the following rules for each variadic argument:

1. Round up the Next SIMD and Floating-point Register Number (NSRN) to the next multiple of 8 bytes.
2. Assign the variadic argument to the appropriate number of 8-byte stack slots.

Because of these changes, the type `va_list` is an alias for `char*`, and not for the struct type in the generic procedure call standard. The type also isn't in the `std` namespace when compiling C++ code.

> **Note**
>
> The C language requires the promotion of arguments smaller than int before a call. Beyond that, the Apple platforms ABI doesn't add unused bytes to the stack.

Handle C++ differences

The generic ARM64 C++ ABI mirrors the Itanium C++ ABI, which many UNIX-like systems use. Apple's C++ ABI differs from this ABI in the following ways:

- The mangled name of the `va_list` type is `Pc`, and not `St9__va_list`. This difference occurs because `va_list` is an alias for `char *`, and uses the same name-mangling conventions.
- The mangled names for NEON vector types match their 32-bit ARM counterparts, rather than using the 64-bit scheme. For example, Apple platforms use `17__simd128_int32_t` instead of the generic `11_int32x4_t`.

- When passing parameters to a function, Apple platforms ignore empty structures unless those structures have a nontrivial destructor or copy constructor. When passing such nontrivial structures, treat them as aggregates with one byte member in the generic manner.
- The ABI requires the complete object (C1) and base-object (C2) constructors to return this to their callers. Similarly, the complete object (D1) and base-object (D2) destructors return this. This behavior matches the ARM 32-bit C++ ABI.
- The ABI provides a fixed layout of two `size_t` words for array cookies, with no extra alignment requirements. This behavior matches the ARM 32-bit C++ ABI.
- Object initialization guards are nominally `uint64_t`, rather than `int64_t`. This behavior affects the prototypes of functions `__cxa_guard_acquire`, `__cxa_guard_release`, and `__cxa_guard_abort`.
- A pointer to a function declared as `extern "C"` isn't interchangeable with a function declared as `extern "c++"`. This behavior differs from the ARM64 ABI, in which the functions are interchangeable.

For more information about the generic ARM64 C++ ABI, see "C++ Application Binary Interface Standard for the ARM 64-bit architecture" at developer.arm.com.

Enable DIT for constant-time cryptographic operations

Certain instructions on ARM64, including but not limited to those described in Arm Architecture Registers for Future Architecture Technologies, may take a different amount of time to run depending on the data values on which they operate. Malicious code running on the device might use this property to infer information about the data the CPU processes, such as cryptographic keys, or other sensitive data.

Apple silicon provides *data-independent timing* (DIT), in which the processor completes certain instructions in a constant amount of time. With DIT enabled, the processor uses the longer, worst-case amount of time to complete the instruction, regardless of the input data. When you write software specifically to avoid leaking internal information and to run code in constant time, enabling DIT — and restricting your code to instructions that support DIT — before loading cryptographic key material, performing cryptographic operations, or processing sensitive data ensures the timing of specific instructions doesn't reveal information about the data being processed.

Enable DIT in specialized situations, such as cryptographic routines. Because DIT can slow down your code, only enable it in situations where you write software to run in constant time with respect to sensitive data, and to avoid leaking sensitive information. Apple cryptographic

routines implemented in the operating system, and which are made available in APIs such as Apple CryptoKit, enable DIT internally.

> **Important**
>
> While DIT ensures the timing of certain instructions don't reveal information about the data, you need additional programming practices to prevent other changes to the processor's microarchitectural state from providing an adversary with signals about secret values. For example, avoid conditional branches and memory access locations based on the value of the secret data.

In iOS 18.2, iPadOS 18.2, macOS 15.2, tvOS 18.2, watchOS 11.2, and visionOS 2.2 and later, two new function calls are available to control and optimize DIT for Apple devices. The functions are available on all devices regardless of whether they support DIT, but only turn on DIT on supported devices.

You protect a set of cryptographic operations by calling `timingsafe_enable_if_supported` unconditionally to turn on DIT. Internally, the function uses safeguards adequate for the device CPU to limit the speculative computation; for example, *speculation barriers* (SB). The API function is optimized so the device runs the most efficient safeguards automatically and you avoid several compile and runtime checks. If the timing-safe features aren't supported they are ignored and the function has no effect.

The function returns an opaque token you use to restore the previous state. Store the returned token and when the timing sensitive operations are done, pass the token to `timingsafe_restore_if_supported` to restore the CPU to its previous state.

If DIT is on prior to calling `timingsafe_enable_if_supported`, it remains on after calling `timingsafe_restore_if_supported`; otherwise, it's turned off.

To turn on timing-safe mode before performing constant-time cryptographic operations, and subsequently restore the CPU's previous state, use the following:

```
#include <timingsafe.h>

int cryptographic_routine() {
    timingsafe_token_t restore_token = timingsafe_enable_if_supported();
    // Perform constant-time cryptographic operations here.
    timingsafe_restore_if_supported(restore_token);
    return 0;
}
```

Alternatively, to turn on DIT with older SDKs or platforms where the APIs aren't available, you need to check whether DIT is defined and supported by the CPU and then use low-level kernel and CPU registers. To test whether the APIs are available, use the compiler's `__builtin_available` directive.

To determine whether a device's CPU supports DIT, test the `hw.optional.arm.FEAT_DIT` system control using `sysctlbyname`:

```
#include <sys/sysctl.h>
#include <stdbool.h>


bool is_DIT_supported(void) {
    static int has_DIT = -1;
    if (has_DIT == -1) {
        size_t has_DIT_size = sizeof(has_DIT);
        if (sysctlbyname("hw.optional.arm.FEAT_DIT", &has_DIT, &has_DIT_size, NU
            has_DIT = 0;
        }
    }
    return has_DIT;
}
```

This function returns `1` if DIT is available; `0` otherwise.

To check whether DIT is turned on for the current thread, test the `dit` bit (bit 24) of the processor state register. The `dit` bit is only defined on certain devices and you can set an attribute for the function to avoid errors when compiling for unsupported devices:

```
#ifdef __arm64__
__attribute__((target("dit")))
bool get_DIT_enabled(void) {
    return (__builtin_arm_rsr64("dit") >> 24) & 1;
}
#endif
```

Turn on DIT for the current thread by setting the `dit` processor state register's value to `1`. To ensure that subsequent instruction timing reflects the updated DIT processor state, add a speculation barrier after turning on DIT. Use the `sb` instruction to add a speculation barrier:

```
#ifdef __arm64__
__attribute__((target("sb")))
void inst_dit_speculation_barrier(void) {
  __asm__ __volatile__("sb" ::: "memory");
}
#endif
```

To restore the state of DIT after the sensitive operation, read and return its current status before turning on DIT:

```
#ifdef __arm64__
__attribute__((target("dit")))
bool set_DIT_enabled(void) {
    bool was_DIT_enabled = get_DIT_enabled();
    __asm__ __volatile__("msr dit, #1");
    inst_dit_speculation_barrier();
    return was_DIT_enabled;
}
#endif
```

To turn off DIT for the current thread, set the `dit` processor state register's value to `0`. To prevent turning off DIT inadvertently in nested calls, restore DIT to its previous value instead of unconditionally turning it off. Store the initial value in thread-local storage, as you can turn DIT on or off separately on different threads. If the DIT value record before turning on DIT is `0`, set the DIT value to `0`:

```
#ifdef __arm64__
__attribute__((target("dit")))
void restore_DIT(bool was_DIT_enabled) {
    if (was_DIT_enabled == false) {
        __asm__ __volatile__("msr dit, #0");
    }
}
#endif
```

To test the availability of speculation barriers, use a mechanism similar to DIT, above:

```
bool is_SB_supported(void) {
    static int has_SB = -1;
```

```
    if (has_SB == -1) {
        size_t has_SB_size = sizeof(has_SB);
        if (sysctlbyname("hw.optional.arm.FEAT_SB", &has_SB, &has_SB_size, NULL,
            has_SB = 0;
        }
    }
    return has_SB;
}
```

When speculation barriers aren't supported, use:

```
#ifdef __arm64__
void inst_dit_speculation_barrier_unsupported(void) {
  __asm__ __volatile__ ("dsb nsh" ::: "memory");
  __asm__ __volatile__ ("isb sy" ::: "memory");
}
#endif
```

To create functions that are available across all devices — whether they support DIT and speculation barriers — define function pointers. If the device supports DIT, set the function pointers to functions that test and change the processor state register. When DIT isn't supported, set the function pointers to functions that do nothing, and follow the same pattern with speculation barriers:

```
bool(* set_DIT_enabled_if_supported)(void);
void(* restore_DIT_if_supported)(bool);
bool(* get_DIT_enabled_if_supported)(void);
void(* inst_dit_speculation_barrier_if_supported)(void);

bool set_DIT_enabled(void);
void restore_DIT(bool);
bool get_DIT_enabled(void);
void inst_dit_speculation_barrier(void);

bool set_DIT_enabled_unsupported(void);
void restore_DIT_unsupported(__unused bool was_DIT_enabled);
bool get_DIT_enabled_unsupported(void);
void inst_dit_speculation_barrier_unsupported(void);

void init_DIT_control(void);
```

```
#ifdef __arm64__
__attribute__((target("dit")))
bool set_DIT_enabled(void) {
    bool was_DIT_enabled = get_DIT_enabled();
    __asm__ __volatile__("msr dit, #1");
    inst_dit_speculation_barrier_if_supported();
    return was_DIT_enabled;
}
#endif

bool get_DIT_enabled_unsupported(void) {
    return false;
}

bool set_DIT_enabled_unsupported(void) {
    return false;
}

void restore_DIT_unsupported(__unused bool was_DIT_enabled) {
    return;
}

void init_DIT_control(void) {
#ifdef __arm64__
    if (is_SB_supported()) {
        inst_dit_speculation_barrier_if_supported = inst_dit_speculation_barrie
    }
    else {
        inst_dit_speculation_barrier_if_supported = inst_dit_speculation_barrie
    }

    if (is_DIT_supported()) {
        set_DIT_enabled_if_supported = set_DIT_enabled;
        restore_DIT_if_supported = restore_DIT;
        get_DIT_enabled_if_supported = get_DIT_enabled;
    } else
#endif
    {
        set_DIT_enabled_if_supported = set_DIT_enabled_unsupported;
        restore_DIT_if_supported = restore_DIT_unsupported;
```

```
        get_DIT_enabled_if_supported = get_DIT_enabled_unsupported;
    }
#ifdef __arm64__
    inst_dit_speculation_barrier_if_supported();
#endif
```

Altogether, you protect a set of cryptographic instructions by calling the abstracted DIT enablement function prior to the cryptographic operation, and restore the previous DIT state after:

```
#if __has_include(<timingsafe.h>)
#include <timingsafe.h>
#endif

int main(void) {
    init_DIT_control();
    // Run the rest of your program.
    return 0;
}

int cryptographic_routine(void) {
#if __has_include(<timingsafe.h>)
    if (__builtin_available(macOS 15.2, iOS 18.2, visionOS 2.2, tvOS 18.2, watcl
        // API-based DIT control.
        timingsafe_token_t token = timingsafe_enable_if_supported();
        // Perform constant time cryptographic operations here.
        timingsafe_restore_if_supported(token);
    } else
#endif
    {
        // Fallback on earlier DIT control versions.
        bool was_DIT_enabled = set_DIT_enabled_if_supported();
        // Perform constant time cryptographic operations here.
        restore_DIT_if_supported(was_DIT_enabled);
    }
    return 0;
}
```

See Also

64-bit interfaces

Writing 64-bit Intel code for Apple Platforms

Create 64-bit Intel assembly language instructions that adhere to the application binary interface (ABI) that Apple platforms support.