11 OCTOBER 2017 BY PHILLIP JOHNSTON

Demystifying ARM Floating Point Compiler Options

When I first started bringing up new ARM platforms, I was pretty confused by the various floating point options such as `-mfloat-abi=softfp` or `-mfpu=fpv4-sp-d16`. I imagine this is confusing to other developers as well, so I'd like to share my ARM floating-point cheat sheet with the world.

An Overview of the ARM Floating-Point Architecture

Before we dive into compiler options, there are a few ARM floating-point details we should familiarize ourselves with: the ARM EABI, VFP, and NEON.

ARM EABI

An ABI is a specification which defines the rules that a generated program must follow to work with a specific platform or interface. The ARM EABI defines the rules for an ARM platform, and your compiler will build your program according to those rules.

The ARM EABI specification defines two incompatible ABIs: one which uses floating-point registers for function arguments, and another which does not. If we are not using hardware floating-point operations, we can simply build our program without using the floating-point compatible ABI.

Since ARM defines a standard floating-point instruction set, we can still utilize the floating-point ABI even if our chip does not support the actual hardware. If floating-point hardware is not present, the instructions will be trapped and executed by a floating-point

Since the ABI defines interfaces for our programs, we must compile and link all of our components and libraries using the same ABI.

Vector Floating Point (VFP)

Vector Floating Point (VFP) is the name for ARM's floating-point extension. Prior to ARMv8, VFP was implemented as a coprocessor extension. The VFP coprocessor supports both single and double-precision floating point operations according to the IEEE 754 standard. For practical purposes, VFP is not useful for vector operations and should be considered a normal scalar floating-point unit (FPU). VFP has been replaced with NEON as of ARMv8.

The VFP extensions are optional parts of the ARM architecture, though the majority of Cortex-A processors do provide a floating-point unit. Some Cortex-A8 devices may utilize a reduced VFPLite module instead of a full VFP module. This VFPLite module requires roughly a 10x increase in clock cycles per floating-point operation.

Some devices such as the ARM Cortex-A8 have a cut-down VFPLite module instead of a full VFP module, and require roughly ten times more clock cycles per float operation.[81]

VFP Versions

Here's a high-level summary of the different VFP versions that have been released throughout the years.

- **VFPv1**
  - Obsoleted by ARM
- **VFPv2**
  - 16 64-bit FPU registers
  - Optional extension to the ARM instruction set in the ARMv5TE, ARMv5TEJ, ARMv6, and ARMv6K architectures
  - Optional extension to the ARM and Thumb instruction set in the ARMv6T2 architecture
  - Supports standard FPU arithmetic (`add`, `sub`, `neg`, `mul`, `div`), full square root
- **VFPv3**
  - Backwards compatible with VFPv2, except that it cannot trap floating-point exceptions
  - Adds `VCVT` instructions to convert between scalar, `float` and `double`

    - Adds immediate mode to `VMOV` such that constants can be loaded into FPU registers.
    - **VFPv3-D32**
        - 32 64-bit FPU registers
        - Implemented on most Cortex-A8 and A9 ARMv7 processors
    - **VFPv3-D16**
        - 16 64-bit FPU registers
        - Implemented on Cortex-R4 and R5 processors and the Tegra 2 (Cortex-A9).
    - **VFPv3-F16**
        - Uncommon
        - Supports IEEE754-2008 half-precision (16-bit) floating point as a storage format
    - **VFPv3U**
        - A variant of VFPv3 that supports the trapping of floating-point exceptions to support code.
        - Can support single- or half-precision floating point
- **VFPv4**
    - Built on VFPv3
    - Adds half-precision support as a storage format
    - Adds fused multiply-accumulate instructions
    - **VFPv4-D32**
        - 32 64-bit FPU registers
        - Implemented on the Cortex-A12 and A15 ARMv7 processors
        - Cortex-A7 optionally has VFPv4-D32 (in the case of an FPU with NEON)
    - **VFPv4-D16**
        - 16 64-bit FPU registers
        - Implemented on Cortex-A5 and A7 processors (in case of an FPU without NEON)
    - **VFPv4U**
        - A variant of VFPv4 that supports the trapping of floating-point exceptions to support code
    - Can support single- or half-precision floating point
- **VFPv5**
    - Implemented on Cortex-M7 when single and double-precision floating-point core option exists

---

NEON

NEON, the "Advanced Single Instruction Multiple Data (SIMD) Extension", is ARM's successor to the VFP coprocessor. NEON is a VFP extension which allows for efficient matrix and vector data manipulation and is commonly used in signal-processing applications. Prior to ARMv8, the ARM architecture distinguished between VFP and NEON floating-point support. NEON was not fully IEEE 754 compliant, and there were instructions that VFP supported which NEON did not. These issues have been resolved with ARMv8.

NEON sports a combined 64- and 128-bit SIMD instruction set and shares the same floating-pointer registers as used in VFP. Some devices, such as the Cortex-A8 and Cortex-A9 lines, support 128-bit vectors but operate on 64 bits at a time. Newer processors such as the Cortex-A15 can operate on 128 bits at a time.

NEON remains an optional part of the ARM architecture. However, NEON is included in all Cortex-A8 devices.

SVE

The Scalable Vector Extension (SVE) is the next-generation ARM SIMD instruction set. Currently it is only targeting ARMv8-A and the `aarch64` ISA.

Compiler Options

Now that we have a high level understanding of ARM floating-point technologies, let's take a look at the compiler options we can use. I will be providing information relevant to the GNU and clang toolchains. For more information on the ARM compiler options, please see this reference documentation.

Let's dive into the two major compilation options: `-mfloat-abi` and `-mfpu`.

`float-abi`

The `-mfloat-abi=<name>` option is used to select which ARM ABI is used. This option also controls whether floating-point instructions may be used.

Here are your `float-abi` options:

- `soft`: full software floating-point support
- `softfp`: Allows use of floating-point instructions but maintains compatibility with the soft-float ABI
- `hard`: Uses floating-point instructions and the floating-point ABI.

Each target architecture has a default value which is used if no option is supplied.

Note well: the two ARM ABIs (hard-float and soft-float) are not link-compatible. Your entire program must be compiled using the same ABIs. If a pre-compiled library is not supplied with your target floating-point ABI, you will need to recompile it for your own purposes.

`soft`

The `soft` option enables full software floating-point support. The compiler will not generate FPU instructions in `soft` mode. Instead, the compiler generates library calls to handle floating point operations. The compiler also generates prologue and epilogue functions to pass floating-point arguments (`float`, `double`) into integer registers (one for `float`, `two for`double`).

When using the `soft` option, the `-mfpu` flag is ignored.

`softfp`

The `softfp` option is a hybrid between `hard` and `soft`. The compiler is allowed to generate hardware floating-point instructions, but it still uses the `soft-float` ABI. Like with `soft`, the compiler generates functions to pass floating-point arguments to integer registers. Depending on the chosen FPU (`-mfpu`), the compiler can choose when to use emulated or hardware floating-point instructions.

Since both `soft` and `softfp` use the same `soft-float` ABI, code built with either option can be linked together. However, when copying data from integer to floating-point registers, a pipeline stall is incurred for every copy. This additional overhead can impact the performance of your application, since data is being copied back-and-forth from the FPU registers when using floating-point arguments.

hard

The `hard` option enables full hardware floating-point support. The compiler generates floating-point instructions and uses the floating-point ABI. Floating-point function arguments are passed directly into FPU registers. Since there are no function prologue or epilogue requirements, no pipeline stalls are incurred with floating-point arguments. The `hard` float option will provide you with the highest performance, but does limit your compiled binary to the selected FPU.

When using the `hard` option, you must define an FPU using `-mfpu`.

fpu

When using the `hard` or `softfp` `float-abi`, you should specify the FPU type using the `-mfpu` compiler flag. This flag specifies what floating-point hardware (or emulation) is available on your target architecture. When using the `soft-float` ABI, `fpu` determines the format of the floating-point values.

The `-mfpu=<name>` option supports the following FPU types: `vfp`, `vfpv3`, `vfpv3-fp16`, `vfpv3-d16`, `vfpv3-d16-fp16`, `vfpv3xd`, `vfpv3xd-fp16`, `neon`, `neon-fp16`, `vfpv4`, `vfpv4-d16`, `fpv4-sp-d16`, `neon-vfpv4`, `fp-armv8`, `neon-fp-armv8`, and `crypto-neon-fp-armv8`.

Each of the FPU options corresponds to the floating-point architectures described above, and some options represent supersets. If you don't care about the specific VFP type, you can select supersets (`vfp`, `neon`). You can also generalize VFP versions as supersets (`vfpv3`, `vfpv4`).

fp16-format

The`-mfp16-format=<name>` option allows you to specify the format of the half-precision floating-point type (`__fp16`). Valid options are `none`, `ieee`, and `alternative`. The default option is `none`, meaning `__fp16` is not defined.

For more information, see the GNU Half-Precision Floating Point documentation.

Performance Impacts

In general, applications relying on floating-point operations will benefit from using the `hard-float` ABI.

Debian has some notes on VFP performance improvements and cite a proof-of-concept Ubuntu build which noted significant performance improvements with floating-point heavy libraries.

Further Reading

- ARM Technologies: Floating Point
- ARM Technologies: NEON
- ARM: Scalable Vector Extension
- Wikipedia: ARM Architecture
    - NEON
    - Floating Point (VFP)
- GNU ARM Options
    - Half-Precision Floating Point
    - Routines for Floating Point Emulation
- ARM Infocenter: Floating Point Build Options
- Debian: VFP Comparison
- VFP Instruction Set Quick Reference
- Floating Point Instructions
- IEEE Standard Floating-Point Arithmetic

Related

Prefer gcc-ar to ar In Your Buildsystems

compiler-rt

Implementing Stack Smashing Protection for Microcontrollers (and Embedded Artistry's libc)

UNCATEGORIZED

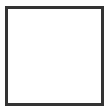
ARM, CLANG, COMPILERS, FEATURED, GCC

2 Replies to “Demystifying ARM Floating Point Compiler Options”

Stephen McCaul

13 JULY 2020 AT 12:45

How do you specify vfpv5? it is not listed in your -mfpu argument list. Also vfpv5 includes more features than just double support (VRINTZ for example).



Phillip Johnston

13 JULY 2020 AT 13:35

I will update the article with clarifications around vfpv5. Looking at the GCC manual I now see “fpv5-sp-d16” and “fpv5-d16” arguments. It looks like some of the -march options take extensions like “+fp” or “+fp.dp” to enable vfpv5. For example:

For armv8-m.main:
+dsp: Enables DSP Extension. +fp: Enables single-precision only VFPv5 instructions with 16 double-word registers. +fp.dp: Enables VFPv5 instructions with 16 double-word registers.+nofp: Disables all FPU instructions. +nodsp: Disables DSP Extension.

More info here: https://sourceware.org/binutils/docs/as/ARM-Options.html