Palmer Dabbelt—August 14, 2017

https://www.sifive.com/blog/all-aboard-part-1-compiler-args

All Aboard, Part 1: The -march, -mabi, and -mtune arguments to RISC-V Compilers

← IntroductionRelocations in ELF Toolchains →

Before we can board the RISC-V train, we'll have to take a stop at the metaphorical ticket office: our machine-specific GCC command-line arguments. These arguments all begin with **-m**, and are all specific to the RISC-V architecture port. In general, we've tried to match existing conventions for these arguments, but like pretty much everything else there are enough quirks to warrant a blog post. This blog discusses the arguments most fundamental to the RISC-V ISA: the **-march**, **-mabi**, and **-mtune** arguments.

One advantage of having a functional GCC port for a long time before we stabilized SiFive's interfaces is that we can have a well-thought-out command-line interface to the RISC-V C and C++ compilers. This allows us to expose the same command-line interface from both the GNU tools (GCC and binutils) as well as the LLVM tools, as well as avoid the need for users to directly pass flags to the assembler or linker via the compiler's **-Wa** and **-Wl** arguments.

To ensure that the RISC-V compiler command-line interface is easy to extend in the future, we decided on a scheme where users describe the RISC-V target they are trying to compile for using three arguments:

- **-march=ISA** selects the architecture to target. This controls which instructions and registers are available for the compiler to use.
- **-mabi=ABI** selects the ABI to target. This controls the calling convention (which arguments are passed in which registers) and the layout of data in memory.
- **-mtune=CODENAME** selects the microarchitecture to target. This informs GCC about the performance of each instruction, allowing it to perform target-specific optimizations.

The -march Argument

The **-march** argument is essentially defined by the RISC-V user-level ISA manual. **-march** controls instruction set from which the compiler is allowed to generate instructions. This argument determines the set of implementations that a program will run on: any RISC-V compliant system that subsumes the **-march** value used to compile a program should be able to run that program.

To get a bit more specific: Version 2.2 of the RISC-V User-Level ISA defines three base ISAs that are currently supported by the toolchain:

- RV32I: A load-store ISA with 32, 32-bit general-purpose integer registers.
- RV32E: An embedded flavor of RV32I with only 16 integer registers.
- RV64I: A 64-bit flavor of RV32I where the general-purpose integer registers are 64-bits wide.

In addition to these base ISAs, a handful of extensions have been specified. The extensions that have been specified and are supported by the toolchain are:

- M: Integer Multiplication and Division
- A: Atomic Instructions
- F: Single-Precision Floating-Point
- D: Double-Precision Floating-Point
- C: Compressed Instructions

RISC-V ISA strings are defined by appending the supported extensions to the base ISA in the order listed above. For example, the RISC-V ISA with 32, 32-bit integer registers and the instructions for multiplication would be denoted as **RV32IM**. Users can control the set of instructions that GCC uses when generating assembly code by passing the lower-case ISA string to the **-march** GCC argument: for example **-march=rv32im**.

On RISC-V systems that don't support particular operations, emulation routines may be used to provide the missing functionality. For example the following C code

```
double dmul(double a, double b) {
  return a * b;
}
```

will compile directly to a FP multiplication instruction when compiled with the D extension

```
$ riscv64-unknown-elf-gcc test.c -march=rv64imafdc -mabi=lp64d -o- -S -O3
dmul:
  fmul.d  fa0,fa0,fa1
  ret
```

but will compile to an emulation routine without the D extension

```
$ riscv64-unknown-elf-gcc test.c -march=rv64i -mabi=lp64 -o- -S -O3
dmul:
  add     sp,sp,-16
  sd      ra,8(sp)
  call    __muldf3
  ld      ra,8(sp)
  add     sp,sp,16
  jr      ra
```

Similar emulation routines exist for the C intrinsics that are trivially implemented by the M and F extensions. As of this writing, there are no A routine emulations because they were rejected as part of the Linux upstreaming process -- this might change in the future, but - for now - we plan to mandate that Linux-capable machines subsume the A extension as part of the RISC-V platform specification.

The -mabi Argument

The **-mabi** argument to GCC specifies both the integer and floating-point ABIs to which the generated code complies. Much like how the **-march** argument specifies which hardware generated code can run on, the **-mabi** argument specifies which software generated code can link against. We use the standard naming scheme for integer ABIs (**ilp32** or **lp64**), with an argumental single letter appended to select the floating-point registers used by the ABI

(**ilp32** vs **ilp32f** vs **ilp32d**). In order for objects to be linked together, they must follow the same ABI.

RISC-V defines two integer ABIs and three floating-point ABIs, which together are treated as a single ABI string. The integer ABIs follow the standard ABI naming scheme:

- **ilp32**: **int**, **long**, and pointers are all 32-bits long. **long** is a 64-bit type, **char** is 8-bit, and **short** is 16-bit.
- **lp64**: **long** and pointers are 64-bits long, while **int** is a 32-bit type. The other types remain the same as ilp32.
while the floating-point ABIs are a RISC-V specific addition:
- "" (the empty string): No floating-point arguments are passed in registers.
- **f**: 32-bit and smaller floating-point arguments are passed in registers. This ABI requires the F extension, as without F there are no floating-point registers.
- **d**: 64-bit and smaller floating-point arguments are passed in registers. This ABI requires the D extension.
Just like ISA strings, ABI strings are concatenated together and passed via the **-mabi** argument to GCC. In order to explain why the ISA and ABI should be treated as two separate arguments, let's examine a handful of **-march**/**-mabi** combinations:

- **-march=rv32imafdc -mabi=ilp32d**: Hardware floating-point instructions can be generated and floating-point arguments are passed in registers. This is like the **-mfloat-abi=hard** argument to ARM's GCC.
- **-march=rv32imac -mabi=ilp32**: No floating-point instructions can be generated and no floating-point arguments are passed in registers. This is like the **-mfloat-abi=soft** argument to ARM's GCC.
- **-march=rv32imafdc -mabi=ilp32**: Hardware floating-point instructions can be generated, but no floating-point arguments will be passed in registers. This is like the **-mfloat-abi=softfp** argument to ARM's GCC, and is usually used when interfacing with soft-float binaries on a hard-float system.
- **-march=rv32imac -mabi=ilp32d**: Illegal, as the ABI requires floating-point arguments are passed in registers but the ISA defines no floating-point registers to pass them in.
As a more concrete example, let's examine a simple C function that takes two double-precision arguments and returns their product. In order to make argument location explicit in all cases, we'll reverse the order of the arguments between the function call and the multiplication:

```
double dmul(double a, double b) {
  return b * a;
}
```

The first argument is the simplest one: if neither the ABI, nor the ISA, contains the concept of floating-point hardware, then the C compiler cannot emit any floating-point-specific instructions. In this case,emulation routines are used to perform the computation and the arguments are passed in integer registers. As you can see, the double-precision arguments are passed in 32-bit integer register pairs, the order of arguments is swapped, **ra** is saved (as it's callee saved), the emulation routine is called, the stack is unwound, and the result is returned (which is already in **a0,a1** from **__muldf3**).

```
$ riscv64-unknown-elf-gcc test.c -march=rv32imac -mabi=ilp32 -o- -S -O3
dmul:
```

```
        mv      a4,a2
        mv      a5,a3
        add     sp,sp,-16
        mv      a2,a0
        mv      a3,a1
        mv      a0,a4
        mv      a1,a5
        sw      ra,12(sp)
        call    __muldf3
        lw      ra,12(sp)
        add     sp,sp,16
        jr      ra
```

The second case is the exact opposite of this one: everything is supported in hardware. In this case we can emit a single **fmul.d** instruction to perform the computation, which when register allocated correctly handles reversing the input arguments and producing the return value.

```
    $ riscv64-unknown-elf-gcc test.c -march=rv32imafdc -mabi=ilp32d -o- -S -O3
    dmul:
        fmul.d  fa0,fa1,fa0
        ret
```

The last case exposes why there is a split between the **-march** and **-mabi** arguments to RISC-V compilers: users may want to generate code that can be linked with code designed for systems that don't subsume a particular extension while still taking advantage of the extra instructions present in a particular extension. This is a common problem when dealing with legacy libraries that need to be integrated into newer systems so we've designed our compiler arguments and multilib paths to cleanly integrate with this workflow.

The generated code is essentially a mix between the two above outputs: the arguments are passed in the registers specified by the **ilp32** ABI (as opposed to the **ilp32d** ABI, which could pass these arguments in registers) but then once inside the function the compiler is free to use the full power of the **rv32imafdc** ISA to actually compute the result. As a result, the compiler generates the double-precision arguments in memory (the only way to construct a double on **rv32**), loads them into **F** registers, performs the computation, stores the **F**-register result back out to the stack, and loads the result into the ABI-compliant return value registers (**a0** and **a1**). While this is less efficient than the code the compiler could generate if it was allowed to take full advantage of the D-extension registers, it's a lot more efficient than computing the floating-point multiplication without the D-extension instructions

```
    $ riscv64-unknown-elf-gcc test.c -march=rv32imafdc -mabi=ilp32 -o- -S -O3
    dmul:
        add     sp,sp,-16
        sw      a0,8(sp)
        sw      a1,12(sp)
        fld     fa5,8(sp)
        sw      a2,8(sp)
        sw      a3,12(sp)
        fld     fa4,8(sp)
        fmul.d  fa5,fa5,fa4
```

```
    fsd     fa5,8(sp)
    lw      a0,8(sp)
    lw      a1,12(sp)
    add     sp,sp,16
    jr      ra
```

The final possibly ABI/ISA combination is easy: it's illegal. There's no way the compiler could generate code for an ISA that requires passing arguments in **F** registers if it doesn't have access to the instructions required to access those registers. As this must be user error, we bail out right away.

```
$ riscv64-unknown-elf-gcc test.c -march=rv32imac -mabi=ilp32d -o- -S -O3
cc1: error: requested ABI requires -march to subsume the 'D' extension
```

The -mtune Argument

The last compiler argument that's involved in specifying a target is the simplest of the bunch. While the **-march** argument can cause systems to be unable to execute code and the **-mabi** argument can cause objects to be incompatible with each other, the **-mtune** argument should only change the performance of the generated code. As it currently stands we really don't have any tuning models for RISC-V systems. Unless you've just added a new tuning parameter to our GCC port, you probably shouldn't bother doing anything with this argument.