RISC-V嵌入式开发入门篇1：RISC-VGCC工具链的介绍

> 随着国内第一本RISC-V中文书籍《手把手教你设计CPU——RISC-V处理器篇》正式上市，越来越多的爱好者开始使用开源的蜂鸟E203 RISC-V处理核，很多初学者留言询问有关RISC-V工具链使用的问题，因此本公众号将开始陆续发表若干篇有关RISC-V软件工具链使用的文章，包括：
>
> -
> -
> -
> - RISC-V嵌入式开发入门篇2：RISC-V汇编语言程序设计
> - RISC-V嵌入式开发上手篇：基于HBird-E-SDK平台的软件开发与运行
> - RISC-V嵌入式开发实践篇：运行开源蜂鸟E200 MCU更多示例程序
> - RISC-V嵌入式开发新奇篇：基于Windows Eclipse IDE的软件开发与运行
> - RISC-V嵌入式开发升华篇：基于开源蜂鸟E200 MCU移植RTOS

本文为RISC-V嵌入式开发入门篇1：RISC-V GCC工具链的介绍。

本文的目的是对RISC-V GCC工具链进行简单的中文科普与介绍。

注：本文力求通俗易懂，主要面向初学者，对RISC-V GCC工具链有所了解的读者可以忽略此文。

1 RISC-V GCC工具链种类

在本号上次发表的文章《编译过程简介》中已经介绍了通用的GCC工具链，RISC-V GCC工具链与普通的GCC工具链基本相同，用户可以遵照开源的riscv-gnu-toolchain项目（请在Github中搜索riscv-gnu-toolchain）中的说明自行生成全套的GCC工具链。

由于GCC工具链支持各种不同的处理器架构，因此不同处理器架构的GCC工具链会有不同的命名。遵循GCC工具链的命名规则，当前RISC-V GCC工具链有如下几个版本：

以“riscv64-unknown-linux-gnu-”为前缀的版本，譬如riscv64-unknown-linux-gnu-gcc、riscv64-unknown-linux-gnu-gdb、riscv64-unknown-linux-gnu-ar等。具体的后缀名称与《编译过程简介》中描述的GCC、GDB和Binutils工具相对应。

- “riscv64-unknown-linux-gnu-”前缀表示该版本的工具链是64位架构的Linux版本工具链。注意：此Linux不是指当前版本工具链一定要运行在Linux操作系统的电脑上，此Linux是指该GCC工具链会使用Linux的Glibc作为C运行库，请参见《编译过程简介》了解Glibc的更多信息。
- 同理，“riscv32-unknown-linux-gnu-”前缀的版本则是32位架构。
- 注意：此处的前缀riscv64（还有riscv32的版本）与运行在64位或者32位电脑上毫无关系，此处的64和32是指如果没有通过-march和-mabi选项指定RISC-V架构的位宽，默认将会按照64位还是32位的RISC-V架构来编译程序。有关-march和-mabi选项的含义，请参见第3节。

以“riscv64-unknown-elf-”为前缀的版本，则表示该版本为非Linux（Non-linux）版本的工具链。注意：

- 此Non-Linux不是指当前版本工具链一定不能运行在Linux操作系统的电脑上，此Non-Linux是指该GCC工具链会使用newlib作为C运行库，请参见本号上次发表的文章《嵌入式开发特点》中了解newlib的更多信息。
- 同上理，此处的前缀riscv64（还有riscv32的版本）与运行在64位或者32位电脑上毫无关系，此处的64和32是指如果没有通过-march和-mabi选项指定RISC-V架构的位宽，默认将会按照64位还是32位的RISC-V架构来编译程序。有关-march和-mabi选项的含义，请参见第3节。

以“riscv-none-embed-”为前缀的版本，则表示是最新为裸机（bare-metal）嵌入式系统而生成的交叉编译工具链，所谓裸机（bare-metal）是嵌入式领域的一个常见形态，表示不运行操作系统的系统。该版本使用新版本的newlib作为C运行库，并且支持newlib-nano，能够为嵌入式系统生成更加优化的代码体积（Code Size）。开源的蜂鸟E203 MCU系统是典型的嵌入式系统，因此将使用“riscv-none-embed-”为前缀的版本作为RISC-V GCC交叉工具链。注意：

- 此版本编译器由于使用newlib和newlib-nano作为C运行库，所以必须对 newlib底层的桩函数进行移植，否则无法正常使用调用底层桩函数的C函数（譬如printf会调用write桩函数）。
- 关于Newlib和newlib-nano及其桩函数，请参见本号上次发表的文章《嵌入式开发特点》中了解更多信息。

2 riscv-none-embed工具链下载

对于riscv-none-embed版本的工具链而言，为了方便用户直接使用预编译好的工具链，Eclipse开源社区会定期更新发布最新版本的预编译好的RISC-V嵌入式GCC工具链，包括Windows版本和Linux版本。请在谷歌中搜索“releases gnu-mcu-eclipse/riscv-none-gcc”进入网页下载Windows版本或者Linux版本，如图1中所示。对于Linux和Windows版本均只需在相应的操作系统中解压即可使用。

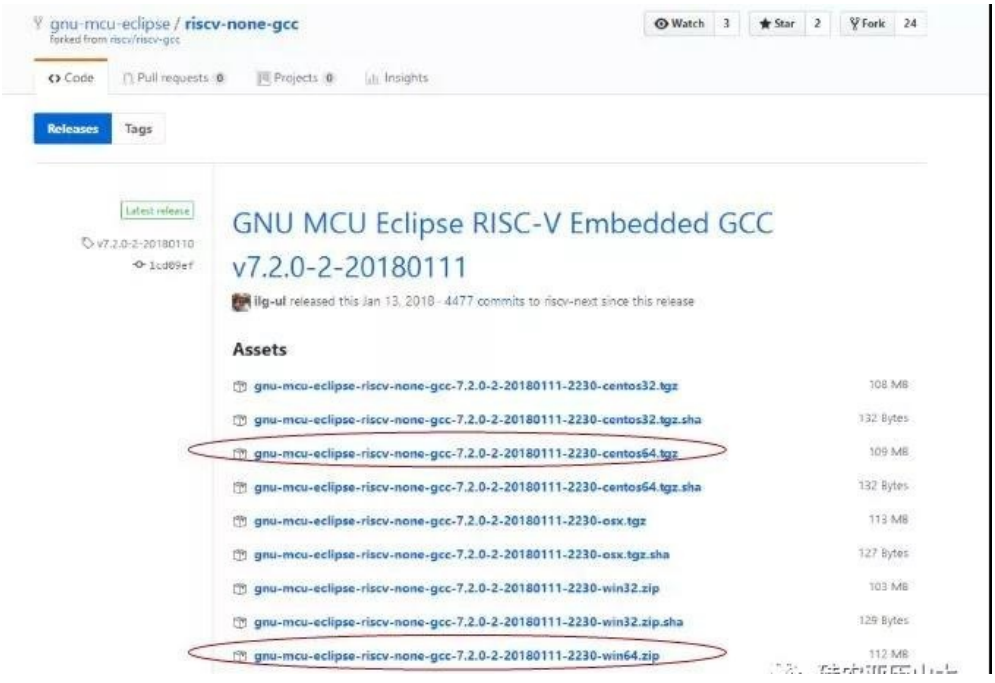

图1 riscv-none-embed工具链下载链接

3 RISC-V GCC工具链的（–march=）和（–mabi=）选项

3.1 （–march=）选项

由于RISC-V的指令集是模块化的指令集，因此在为目标RISC-V平台进行交叉编译之时，需要通过选项指定目标RISC-V平台所支持的模块化指令集组合，该选项为（-march=），有效的选项值如下：

- rv32i[m][a][f[d]][c]
- rv32g[c]
- rv64i[m][a][f[d]][c]
- rv64g[c]

注意：在上述选项中rv32表示目标平台是32位架构，rv64表示目标平台是64位架构，其他i/m/a/f/d/c/g分别代表了RISC-V模块化指令子集的字母简称。请参见RISC-V中文书籍《手把手教你设计CPU——RISC-V处理器篇》中附录A.1节中关于RISC-V架构指令集的详细中文介绍。

本节后文会介绍（-march=）选项使用的具体实例。

3.2 （–mabi=）选项

由于RISC-V的指令集是模块化的指令集，因此在为目标RISC-V平台进行交叉编译之时，需要通过选项指定嵌入式RISC-V目标平台所支持的ABI函数调用规则（有关ABI函数调用规则的相关信息请参见RISC-V中文书籍《手把手教你设计CPU——RISC-V处理器篇》中附录A的图A-1）。RISC-V定义了两种整数的ABI调用规则和三种浮点ABI调用规则，通过选项（-abi=）指明，有效的选项值如下：

- ilp32
- ilp32f
- ilp32d
- lp64
- lp64f
- lp64d

注意：

- 在上述选项中两种前缀（ilp32和lp64）表示的含义如下：
- 前缀ilp32表示目标平台是32位架构，在此架构下，C语言的“int”和“long”变量长度为32比特，“long long”变量为64位；
- 前缀lp64表示目标平台是64位架构，C语言的“int”变量长度为32比特，而“long”变量长度为64比特。
- RISC-V的32位和64位架构下更多的数据类型宽度如图2中所示。

C type	Description	Bytes in RV32	Bytes in RV64
`char`	Character value/byte	1	1
`short`	Short integer	2	2
`int`	Integer	4	4
`long`	Long integer	4	8
`long long`	Long long integer	8	8
`void*`	Pointer	4	8
`float`	Single-precision float	4	4
`double`	Double-precision float	8	8

图2 RISC-V的32位和64位架构下的数据类型宽度

- 上述选项中的三种后缀类型（无后缀、后缀f、后缀d）表示的含义如下：
  - 无后缀：在此架构下，如果使用了浮点类型的操作，直接使用RISC-V浮点指令进行支持。但是当浮点数作为函数参数进行传递之时，无论单精度浮点数还是双精度浮点数均需要通过存储器中的堆栈进行传递。
  - f：表示目标平台支持硬件单精度浮点指令。在此架构下，如果使用了浮点类型的操作，直接使用RISC-V浮点指令进行支持。但是当浮点数作为函数参数进行传递之时，单精度浮点数可以直接通过寄存器传递，而双精度浮点数需要通过存储器中的堆栈进行传递。
  - d：表示目标平台支持硬件双精度浮点指令。在此架构下，如果使用了浮点类型的操作，直接使用RISC-V浮点指令进行支持。当浮点数作为函数参数进行传递之时，无论单精度还是双精度浮点数都可以直接通过寄存器传递。

本节后文会介绍（-mabi=）选项使用的具体实例。

3.3 （–march=）和（–mabi=）不同选项编译实例

为了便于读者更加形象地理解（-march=）和（-mabi=）选项的具体含义，下面以一个实例加以介绍。

假设有一段C语言函数代码，如下所示：

```
//这是一个名为dmul的函数，其有两个参数，为double类型的双精度浮点数
 double dmul(double a, double b) {
  return b * a;
 }
```

- 如果使用-march=rv64imafdc -mabi=lp64d的选项组合进行编译，则会生成如下汇编代码：

```
$ riscv64-unknown-elf-gcc test.c -march=rv64imafdc -mabi=lp64d -o- -S -O3

//所生成的汇编代码如下，从中可以看出，浮点数乘法操作直接使用了RISC-V的fmul.d指令进行支持，且函数的两个double类型的参数直接使用浮点通用寄存器fa0和fa1进行传递。这是因为：
-    -march选项指明了目标平台支持的模块化指令子集为imafdc，其中包含了F和D指令子集，即支持单精度和双精度浮点指令，因此可以直接使用RISC-V的浮点指令来支持浮点数的操作。
-    -mabi选项指明了后缀“d”，表示当浮点数作为函数参数进行传递之时，无论单精度还是双精度浮点数都可以直接通过寄存器传递。

 dmul:
    fmul.d  fa0,fa0,fa1
    ret
```

- 如果使用-march=rv32imac -mabi=ilp32的选项组合进行编译，则会生成如下汇编代码：

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

//所生成的汇编代码如下，从中可以看出，浮点数乘法操作由C库函数（__muldf3）进行支持,这是因为：
-    -march选项指明了目标平台支持的模块化指令子集为I/M/A/C，其中未包含了F和D指令子集，即不支持单精度和双精度浮点指令，因此无法直接使用RISC-V的浮点指令来支持浮点数的操作。

  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
```

- 如果使用-march=rv32imafdc -mabi=ilp32的选项组合进行编译，则会生成如下汇编代码：

```
$ riscv64-unknown-elf-gcc test.c -march=rv32imafdc -mabi=ilp32 -o- -S -O3

//所生成的汇编代码如下，从中可以看出，浮点数乘法操作直接使用了RISC-V的fmul.d指令进行支持，但是函数的两个浮点类型的参数均通过堆栈进行的传递，这是因为：
-   -march选项指明了目标平台支持的模块化指令子集为I/M/A/F/D/C，其中包含了F和D指令子集，即支持单精度和双精度浮点指令，因此可以直接使用RISC-V的浮点指令来支持浮点数的操作。
-   -mabi选项指明了“无后缀”，表示当浮点数作为函数参数进行传递之时，无论单精度还是双精度浮点数都需要通过堆栈进行传递。

  dmul:
    add    sp,sp,-16   //对堆栈指针寄存器（sp）进行调整，分配堆栈空间
    sw     a0,8(sp)    //将函数参数寄存器a0中的值存入堆栈
    sw     a1,12(sp)   //将函数参数寄存器a1中的值存入堆栈
    fld    fa5,8(sp)   //从堆栈中取回双精度浮点操作数
    sw     a2,8(sp)    //将函数参数寄存器a2中的值存入堆栈
    sw     a3,12(sp)   //将函数参数寄存器a3中的值存入堆栈
    fld    fa4,8(sp)   //从堆栈中取回双精度浮点操作数
    fmul.d  fa5,fa5,fa4  //调用RISC-V的浮点指令进行运算
    fsd    fa5,8(sp)   //将计算结果存回堆栈
    lw     a0,8(sp)    //通过堆栈将结果赋值给函数结果返回寄存器a0
    lw     a1,12(sp)   //通过堆栈将结果赋值给函数结果返回寄存器a1
    add    sp,sp,16    //对堆栈指针寄存器（sp）进行调整，回收堆栈空间
    jr     ra
```

- 如果使用-march=rv32imac -mabi=ilp32d的选项组合进行编译，则会报非法错误：

```
$ riscv64-unknown-elf-gcc test.c -march=rv32imac -mabi=ilp32d -o- -S -O3

cc1: error: requested ABI requires -march to subsume the 'D' extension

//从中可以看出报非法错误，这是因为：
-   -march选项指明了目标平台支持的模块化指令子集为I/M/A/C，其中未包含了F和D指令子集，即不支持单精度和双精度浮点指令，因此无法直接使用RISC-V的浮点指令来支持浮点数的操作。
-   -mabi选项指明了后缀“d”，表示目标平台支持硬件浮点指令。这一点与-march选项中指明的指令集子集产生了冲突。
```

3.4 （–march=）和（–mabi=）选项合法组合

虽然（ - march=）和（ - mabi=）选项理论上可以组成很多种不同的组合，但是目前并不是所有的（ - march=）和（ - mabi=）选项组合都是合法，目前的riscv-none-embed工具所支持的组合如下：

- march=rv32i/mabi=ilp32
- march=rv32ic/mabi=ilp32
- march=rv32im/mabi=ilp32
- march=rv32imc/mabi=ilp32
- march=rv32iac/mabi=ilp32
- march=rv32imac/mabi=ilp32
- march=rv32imaf/mabi=ilp32f
- march=rv32imafc/mabi=ilp32f
- march=rv32imafdc/mabi=ilp32f
- march=rv32gc/mabi=ilp32f
- march=rv64imac/mabi=lp64
- march=rv64imafdc/mabi=lp64d
- march=rv64gc/mabi=lp64d

注意：上述有效组合来自于本文撰写时的信息。随着时间推移，新发布的riscv-none-embed工具可能会支持更多的组合，请读者以其最新的发布说明（Release Notes）为准。

4 RISC-V GCC工具链的（–mcmodel=）选项

目前RISC-V GCC工具链认为，在实际的情形中，一个程序的大小一般不会超过4GB的大小，因此在程序内部的寻址空间不能超过4GB的空间。而在64位的架构中，地址空间的大小远远的大于4GB的空间，因此针对RV64架构而言，RISC-V GCC工具链定义了（-mcmodel=）选项用于指定寻址范围的模式，使得编译器在编译阶段能够按照相应的策略编译生成代码。其有效的选项值如下：

- -mcmodel=medlow
- -mcmodel=medany

注意：

- 在RV32架构中，整个地址空间的大小就是4GB，因此（- mcmodel=）选项的任何值对于编译的结果都无影响。
- RISC-V GCC工具链在未来可能也会支持大于4GB的寻址空间。

medlow和medany两个选项的含义分别解释如下。

1. （-mcmodel=medlow）选项

> （-mcmodel==medlow）选项用于指示该程序的寻址范围固定只能在-2GB至+2GB的空间内。注意：地址区间没有负数可言，-2GB是指整个64位地址空间最高2GB地址区间。
> 由于此模式的寻址空间是固定的-2GB至+2GB的空间内，因此编译器能够相对生成比较高效的代码，但是由于寻址空间固定，对于整个64位的大多数地址空间都无法访问到，用户需小心使用。

2. （-mcmodel=medany）选项

> （-mcmodel==medlow）选项用于指示该程序的寻址范围可以在任意的一个4G空间内。由于此模式的寻址空间不是固定的，所以相对比较灵活。

5 RISC-V GCC工具链的其他选项

本章仅介绍了RISC-V GCC工具链几个特别的选项，有关RISC-V GCC工具链的完整选项列表和解释，感兴趣的读者可以在谷歌输入“gcc/RISC-V-Options”关键字进行搜索进入相关网页查询，如图3中所示。

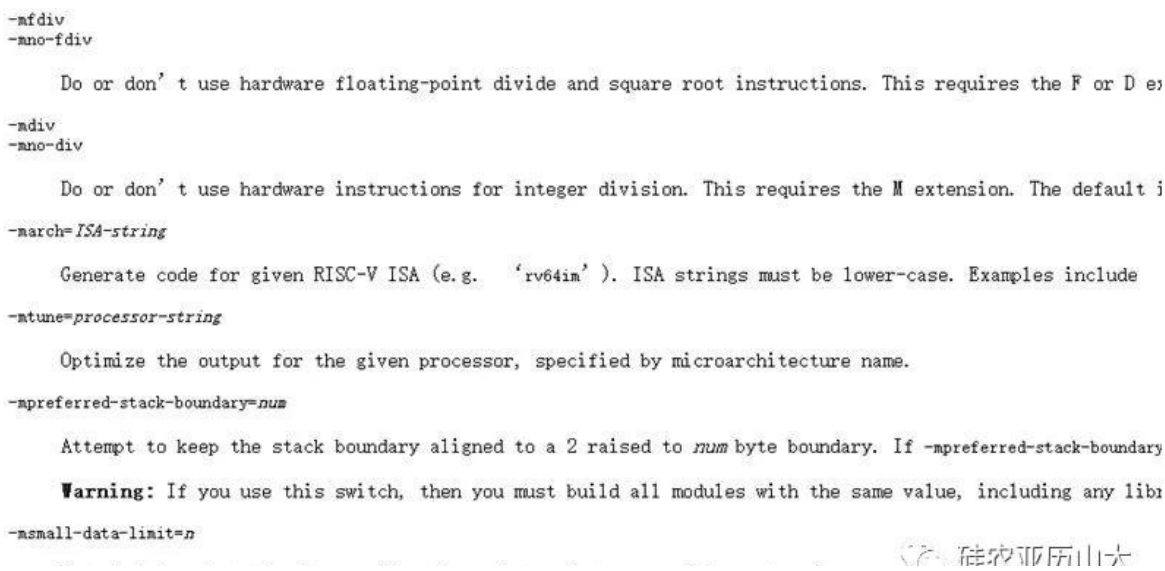

图3 RISC-V GCC工具链的完整选项列表和解释

6 RISC-V GCC工具链的预定义宏

RISC-V GCC会根据编译生成若干预定义的宏，在Linux操作环境中可以使用如下方法查看和RISC-V相关的宏：

```
//首先创建一个空文件
touch empty.h

//使用RISC-V GCC的-E选项对empty.h进行预处理，有关"预处理"的背景知识请参见本号文章《编译过程简介》
//通过grep命令对于处理后的文件搜索riscv的关键字

//如果使用-march=rv32imac -mabi=ilp32选项可以看出生成如下预定义宏
riscv-none-embed-gcc -march=rv32imac -mabi=ilp32 -E -dM empty.h | grep riscv

#define __riscv 1
#define __riscv_atomic 1
#define __riscv_cmodel_medlow 1
#define __riscv_float_abi_soft 1
#define __riscv_compressed 1
#define __riscv_mul 1
#define __riscv_muldiv 1
#define __riscv_xlen 32
#define __riscv_div 1

//如果使用-march=rv32imafdc -mabi=ilp32f选项可以看出生成如下预定义宏
riscv-none-embed-gcc -march=rv32imafdc -mabi=ilp32f -E -dM empty.h | grep riscv


#define __riscv 1
#define __riscv_atomic 1
#define __riscv_cmodel_medlow 1
#define __riscv_float_abi_single 1
#define __riscv_fdiv 1
#define __riscv_flen 64
#define __riscv_compressed 1
#define __riscv_mul 1
#define __riscv_muldiv 1
#define __riscv_xlen 32
#define __riscv_fsqrt 1
#define __riscv_div 1
```

7 RISC-V GCC工具链使用实例

本号后续发文《基于HBird-E-SDK平台的软件开发与运行》将结合HBird-E-SDK平台的实例了解如何使用RISC-V GCC工具链进行嵌入式程序的开发与编译。

更多信息

感兴趣的读者可以通过下面二维码关注公众号“硅农亚历山大”，了解Verilog、IC设计、CPU、RISC-V和人工智能AI相关的更多设计技巧和经验分享，注意：由于干货太多，请自备茶水。