riscv交叉编译指北

本文档主要是一些在 ubuntu 上安装 riscv-gnu-toolchain 并将c语言代码编译为可以直接上板的机器码的踩坑记录

我们使用的电脑大部分为x86、arm等架构，普通编译器（例如gcc）只能生成运行在本指令集上的代码，为了在本机生成 riscv 指令集的汇编/机器码甚至运行，需要安装交叉编译工具链 riscv-gnu-toolchain

riscv-gnu-toolchain 的安装建议在 ubuntu / wsl 环境下进行

安装 riscv-gnu-toolchain 需要将自行编译代码，由于 riscv-gnu-toolchain 包含了gcc等子模块，一共需要下载大约7GB的代码，部分代码所在的仓库从国内访问较为困难（托管在google上），只能通过设置代理访问

step1 下载riscv-gnu-toolchain

step 1.1 首先安装必要的 packages：

```
sudo apt-get install autoconf automake autotools-dev curl python3 python3-pip
libmpc-dev libmpfr-dev libgmp-dev gawk build-essential bison flex texinfo gperf
libtool patchutils bc zlib1g-dev libexpat-dev ninja-build git cmake libglib2.0-
dev
```

step 1.2 下载代码：

```
git clone git@github.com:riscv-collab/riscv-gnu-toolchain.git
cd riscv-gnu-toolchain
```

step 1.3 这份代码包含了很多 submodule，需要继续下载：

```
git submodule update --init --recursive --progress
```

由于网络环境问题，step 1.3 有可能会出错，或者这一步会执行半小时甚至更久

step 1.3.1 git submodule 需要下载的子模块的连接储存在 .gitmodules 下，可以进行换源，但 git submodule 存在嵌套，只对根目录的 .gitmodules 进行修改可能不能解决问题

step 1.3.2 找一个下载成功的同学复制一下，如果你用的是wsl，请打开目录 `\\wsl.localhost\` 以访问文件系统

step 2 编译riscv-gnu-toolchain

step 2.1 设置环境变量

```
vi ~/.bashrc
```

在 ~/.bashrc 中增加两行用于设置环境变量：

```
export RISCV='/opt/riscv'
export PATH=$PATH:$RISCV/bin
```

保存修改后更新修改

```
source ~/.bashrc
```

step 2.2 编译代码

riscv-gnu-toolchain 工具链分为 elf-gcc，linux-gnu-gcc 两个版本，以及他们对应的32位和64位版本。两个的主要区别在于：

riscv32-unknown-elf-gcc，riscv64-unknown-elf-gcc：使用的riscv-newlib库(面向嵌入式的C库)，而且只支持静态链接，不支持动态链接。riscv32-unknown-linux-gnu-gcc，riscv64-unknown-linux-gnu-gcc：使用的是glibc标准库，支持动态链接。如果是编译简单，较小的elf程序，使用elf-gcc版本即可，如果编译比较大的程序或者需要动态库（比如编译linux，或opencv库等），推荐使用linux-gnu-gcc版本。

由于大家的CPU是32位的，并且不需要使用c库，这里我们选择riscv32-unknown-elf-gcc，我们通过以下命令进行选择：

```
../configure --prefix=$RISCV --with-arch=rv32i --with-abi=ilp32
```

--prefix：设置路径

--with-arch：设置为你所需的架构，这里我们选择RV32I指令集，RV32I包含47条指令，除去SCALL/SBREAK/CSRR*等关于系统的指令后均为实验CPU支持的指令

--with-abi：设置为你所需的调用环境Application binary interface，包括数据类型的大小、布局和对齐，这里我们选择32位整数的ilp32

通过以下命令进行编译（可能需要sudo）：

```
make -j4      #wait...
make install  #No action
```

可能需要运行半小时以上，运行成功后可以在 $RISCV 目录看到安装的内容，可以通过以下命令检测安装是否成功：

```
riscv32-unknown-elf-gcc -v
riscv32-unknown-elf-readelf -v
riscv32-unknown-elf-objdump -v
```

step3 交叉编译

和一般的 gcc 一样，riscv32-unknown-elf-gcc 可以通过命令行生成汇编代码 (-S) 和机器码，但为了在实验CPU上运行，需要进行一些修改：

```
riscv32-unknown-elf-gcc test.c -o test -nostartfiles -nostdlib -e main
```

默认编译的时候包含了c标准库，所以需要通过 -nostdlib 参数取消，并加入 -nostartfiles -e main 将程序入口设置为 main 函数

此时 gcc 生成了一个ELF文件，可以通过 riscv32-unknown-elf-readelf 进行观察：

```
# riscv32-unknown-elf-readelf -h test
ELF Header:
  Magic:   7f 45 4c 46 01 01 01 00 00 00 00 00 00 00 00 00
  Class:                             ELF32
  Data:                              2's complement, little endian
  Version:                           1 (current)
  OS/ABI:                            UNIX - System V
  ABI Version:                       0
  Type:                              EXEC (Executable file)
  Machine:                           RISC-V
  Version:                           0x1
  Entry point address:               0x10074
  ...
```

-h 表示查看 ELF header，这里能看到文件的一些信息，其中比较重要的是 Entry point address

```
# riscv32-unknown-elf-readelf -S test
There are 8 section headers, starting at offset 0x15b4:

Section Headers:
  [Nr] Name              Type            Addr     Off    Size   ES Flg Lk Inf Al
  [ 0]                   NULL            00000000 000000 000000 00      0   0  0
  [ 1] .text             PROGBITS        00010074 000074 001200 00  AX  0   0  4
  [ 2] .rodata           PROGBITS        00011274 001274 000010 00   A  0   0  4
  [ 3] .comment          PROGBITS        00000000 001284 00001b 01  MS  0   0  1
  [ 4] .riscv.attributes RISCV_ATTRIBUTE 00000000 00129f 00001c 00      0   0  1
  [ 5] .symtab           SYMTAB          00000000 0012bc 0001f0 10      6   7  4
  [ 6] .strtab           STRTAB          00000000 0014ac 0000c1 00      0   0  1
  [ 7] .shstrtab         STRTAB          00000000 00156d 000044 00      0   0  1
Key to Flags:
  W (write), A (alloc), X (execute), M (merge), S (strings), I (info),
  L (link order), O (extra OS processing required), G (group), T (TLS),
  C (compressed), x (unknown), o (OS specific), E (exclude),
  D (mbind), p (processor specific)
```

-S 表示查看 ELF section，这里能看到 section 信息，注意我们的CPU使用哈弗架构，因此无法读取 .rodata 段的数据，这里笔者没有找到比较好的处理方法，只能通过修改程序写法使得不出现 .rodata 段，希望大家能找到更好的解决方案

接下来通过 objdump 可视化机器码

```
# riscv32-unknown-elf-objdump -S test

test:     file format elf32-littleriscv
Disassembly of section .text:
00010074 <main>:
   10074:       ff010113                add     sp,sp,-16
   10078:       00112623                sw      ra,12(sp)
   1007c:       00812423                sw      s0,8(sp)
   10080:       01010413                add     s0,sp,16
   10084:       40000113                li      sp,1024
   10088:       0f0000ef                jal     10178 <test1>
   ...
```

因为实验CPU的PC入口为0，这里需要保证main函数位于最前面，并且入口地址设置为0。由于编译器优化较难控制，这里笔者同样没有找到完美的解决方案，一种可行的操作是：将start函数写在最前面，设置O0优化，并设置入口为start

```
#pragma GCC push_options
#pragma GCC optimize ("O0")
void start() {
    asm("call main\n\t");
}
#pragma GCC pop_options
```

关于入口地址，一种可行的操作是：在 gcc 中用参数指定 .text 段地址：

```
riscv32-unknown-elf-gcc test.c -o test -O1 -nostartfiles -nostdlib -e main -Wl,-
Ttext-segment,-0x74
```

其中 0x74 为 .text 段 section header 大小，这样做会产生 warning: address of `text-segment' isn't multiple of maximum page size，但如果只是为了得到机器码这不会产生影响

这里提供一个可以生成 dat, coe 文件的 makefile 脚本和一个c语言上板示例代码：

makefile

```
main : *.*
    clear
    riscv32-unknown-elf-gcc test.c -o test -O1 -nostartfiles -nostdlib -e start
-Wl,-Ttext-segment,-0x74
    riscv32-unknown-elf-objdump -d test > test.s
    cat test.s | grep -E '^\s*[0-9a-f]+:.*$$' | grep -o -E '^\s*[0-9a-f]+:\s*[0-
9a-f]+' | grep -o -E '[0-9a-f]+$$' > test.dat
    cat test.dat | sed ':a;N;s/\n/,/g;ta' | sed
'1i;1.asm\nmemory_initialization_radix=16;\nmemory_initialization_vector=' | sed
'$$s/$$/;/g' > test.coe
```

test.c

```
#pragma GCC push_options
#pragma GCC optimize ("O0")
void start() {
  asm("li\tsp,1024\n\t"
      "call main");
}

__attribute__ ((noinline)) void wait(int instr_num) {
  while (instr_num--) ;
}

void main() {
  int x = 0x39C5BB00;
  (*((int*)0xE0000000)) = x;
  while (1) {
    wait(1000000);
    x = x ^ 0x39C5BB00 ^ 0x66CCFF00;
    (*((int*)0xE0000000)) = x;
  }
}
#pragma GCC pop_options
```

接下来的操作

如果有需要的话，可以尝试安装 qemu 模拟器，本地模拟 risc-v 指令集代码的运行，以及 gdb-multiarch 用于调试代码