编译器开发入门指南

GNU ToolChain 为例

通常的选择 --GNU ToolChain

- 事实上的工业标准，绝大部分 *NIX 软件都用 GCC 编译
- 久经考验，代码质量还不错， bug 少！
- 完善的生态环境， binutils gdb glibc 以及众多针对 GCC 开发的软件
- 广泛的用户群，巨大的影响
- IBM AdaCore CodeSourcery RedHat Intel Google 等有实力的厂商支持
- 开发维护成本低，对中小型企业非常划算

其他的选择 --LLVM

- 非常先进的设计，非常清晰的结构，非常好懂的代码！
- iOS 众多 app 检验质量很好
- OpenCL 新版 CUDA 等最新技术都基于 LLVM 去开发
- 生态环境还不成熟
- 对 GCC 兼容还不是 100%
- Apple NVIDIA AMD Xilinx 等有实力的厂商自己开发维护得起

移植就是抄抄改改

- 明确自己处理器的需求
- 找现有的相似的实现去改
- 基本上就是 copy and modify
- 结合 gccint 去抄
- 边抄边看边学边改

binutils 移植的要点

- 符号表， tc-machine 后端， bfd ，基本上就改这几个地方，没什么文档，自己看代码吧，不难

- gas 就一个工作， text2binary ，牢记这个，因为现在 gas 有很多地方好像要做正确性检查，容易让人多想

- text2binary 最重要的就是 position ！ bit 的 position

bfd 部分修改

- bfd/archures.c 中 #define bfd_mach_mips_XXX 的定义
- bfd/bfd-in2.h 中 #define bfd_mach_mips_XXX 的定义
- bfd/cpu-mips.c 中匿名 enum 增加 I-mipsXXX 项， arch_info_struct 数组增加对应描述
- bfd/elfxx-mips.c 中 _bfd_elf_mips_mach 和 mips_set_isa_flag 函数增加对应 case 选择分支，并在 mips_mach_extensions 数组中增加你的描述项

elf 部分修改

- binutils/readelf.c 中 get_machine_flags 函数增加一个对应的 case 描述选项

- include/elf/mips.h 中增加 #define E_MIPS_MACH_XXX 的定义

反汇编支持

- opcodes/mips-dis.c 中增加对应开关选项即可

一个完整的 opcode 分析

- {"baddu", "d,v,t", 0x70000028, 0xfc0007ff, WR_d|RD_s|RD_t, 0, I1}

- baddu 就是指令名字，汇编代码里面的助记符

- 0x70000028 是假设 add 指令的操作数都是 0 的情况下整条指令的编码

- 0xfc0007ff 是用来区分操作码的， mask 和 match 与出来的结果就是操作码

- #define IOCT INSN_OCTEON 让我们找到了 #define INSN_OCTEON 0x00000800
- 值得解释的是 membership 是一个 32 位的 bit 表示，每一位代表一种体系，而 INSN_CHIP_MASK 恰恰 mask 出来有效的位

- d v t 这些 args 是核心所在， #define OP_MASK_RD 0x1f 代表了 d 这个 arg 描述的操作数占 5 位
- 而 #define OP_SH_RD 11 则表示 d 这个 arg 描述的操作数是从 11 位开始的

- pinfo 域跟 args 的定义是一样的， pinfo 是对操作数的一个额外描述，可以用来检查操作数是否合法也可以用做自己想要的意图
- pinfo 如果是 MACRO 的话，这个指令就会先被 macro 函数展开成实际指令

gas 后端支持代码

- machine_ip 处理一个指令，主要工作是通过 args 匹配操作数并 INSERT_OPRAND
- validate_mips_insn 从语法上检查指令是否合法
- macro 通过 macro_build 来把宏指令分解
- 指令表里面写的 args 和 pinfo 在这里被当做处理操作数的依据

binutils 的调试

- 出错提示仅仅是一个提示，造成这个错误的往往是你修改的 opcode 或者 tc-machine.c
- gdb -q $AS $ARGS
- 虽然有人很不屑， print 确实是个简单直观有效的手段

binutils 的验证

- objdump -S 一条指令一条指令的查看二进制是否正确，当然，有模拟器的话，就轻松多了

gcc 移植要点

- gcc 就是 text2text，很多地方就是字符串替换，没有过多的含义
- gcc 的 port 其实是一个逆向的过程，从指令去匹配 rtl，而不是从 rtl 来匹配指令
- gccint 是目前最好的文档，可以看 gccint-zh，理解代码是唯一的出路，文档太少了

gcc 外围支持代码

- gcc/config/mips/driver-native.c 中添加 -march=XXX 的开关代码
- gcc/config/mips/mips.h 中 processor_type 添加 PROCESSOR_XXX，同时定义 #define TARGET_XXX (mips_arch == PROCESSOR_XXX) 和 #define TUNE_XXX (mips_tune == PROCESSOR_XXX)
- gcc/config/mips/mips.h 中还有 N 多宏，根据你的处理器的情况添加相应的开关
- gcc/config/mips/mips.c 中的 mips_cpu_info_table 添加 { "orion", PROCESSOR_R4600, 3, 0 } 的描述项
- gcc/config/mips/mips.c 中的 mips_rtx_cost_data 添加对应的 cost 描述项

mips.h 中几个重要的宏

- #define FIRST_PSEUDO_REGISTER 188
- #define FIXED_REGISTERS
- #define CALL_USED_REGISTERS
- #define CALL_REALLY_USED_REGISTERS
- #define GP_REG_FIRST 0
- #define GP_REG_LAST 31
- #define GP_REG_NUM   (GP_REG_LAST - GP_REG_FIRST + 1)

- enum reg_class
- #define REG_CLASS_NAMES
- #define REG_CLASS_CONTENTS
- #define REG_ALLOC_ORDER
- #define REGISTER_NAMES

一个完整的 rtl 例子

```
(define_insn "add<mode>3"
  [(set (match_operand:VWHB 0 "register_operand" "=f")
        (plus:VWHB (match_operand:VWHB 1 "register_operand" "f")
                   (match_operand:VWHB 2 "register_operand" "f")))]
  "TARGET_HARD_FLOAT && TARGET_LOONGSON_VECTORS"
  "padd<V_suffix>\t%0,%1,%2"
  [(set_attr "type" "fadd")])
```

- define_insn "add<mode>3"
- 是这个模式的名字，总要有个名字吧，匿名的一般用来组成其他模式，比如 define_expand 定义的模式
- 可以映射到一个指令的 define_insn，不能需要分解的 define_expand

- match_operand:VWHB 是用来匹配操作数的
- 三个操作数，分别是 0 1 2
- = 表示这个操作数是被赋值，或者说被写入的
- 很明显呢，都是寄存器操作数
- f 代表浮点数，由 constraints.md 中定义，用来约束操作数的
- plus ，明显的加模式，可以认为是 RTL 的内部操作
- VWHB 是模式，这条指令匹配的机器模式

- padd<V_suffix>\t%0,%1,%2 就是指令摸版，还记得前面的 0 1 2 么？

- set_attr 就算指令的一个分类吧

- HARD_REG 那个条件宏是判断有浮点寄存器并且是龙芯向量指令才会匹配这个模式，就是说满足这个宏才匹配下面的指令摸版

迭代器的使用

- <mode 是一个迭代器 > ，使用迭代器可以大量减少重复代码，迭代器内容定义如下，最终匹配的模式被替换成对应的字符串
- (define_mode_iterator VB [V8QI])
- (define_mode_iterator VH [V4HI])
- (define_mode_iterator VW [V2SI])
- (define_mode_iterator VHB [V4HI V8QI])
- (define_mode_iterator VWH [V2SI V4HI])
- (define_mode_iterator VWHB [V2SI V4HI V8QI])
- (define_mode_iterator VWHBDI [V2SI V4HI V8QI DI])

另一个字符串替换

- ;; The Loongson instruction suffixes corresponding to the modes in the
- ;; VWHBDI iterator.
- (define_mode_attr V_suffix [(V2SI "w") (V4HI "h") (V8QI "b") (DI "d")])
- 这个显然是根据模式，来替换指令模板内容的
- gcc 的工作就是 text2text，不要想太多

gcc 的调试

- gcc -save-temps 是个有用的参数，我们可以在无法正确编译代码的时候，通过这个参数保留 i 和 s 文件以供查找线索

- 出错提示仅仅是提示，不要怀疑公共代码，去你修改的 md 里面找原因，有些东西 print 比 gdb 要方便，自己打印 log 出来吧

- xgcc -### 是个好参数，让我们知道分别给 cc1 as collect2 的参数是什么，就可以 gdb 对应的参数了

- gcc -fdump-tree-all 是个输出所有中间 pass 处理的 IR 供我们找线索的好参数

- make 的时候指定 CFLAGS+="-Wno-error" 避免一些 Warning 当 Error 处理带来的不必要麻烦

gcc 的验证

- gcc -S 是个人尽皆知的蠢办法，但是这足够有效
- 模拟器，模拟器是个好东西

爱好者该怎么入门

- 在 vendor 工作会有一些资源，做起来难度会比较小，但是呢，一般 vendor 只要求 port 即可，想提高也很难
- 专门做编译的 CodeSourcery PathScale 等公司没有相当的实力又无法融入
- 你真的想做这个么？你确定你想做，后面的内容或许会对你有一丁点儿帮助

GCC 社区的开发形式

- committee 是每年 summit 上开黑会的，比较高端， GCC 的 " 命运 " 由他们来满足 FSF 对赞助商的有偿赞助
- maintainer 也是有 level 的， global 的 level 比较高，middle 的也很强悍， x86 这种复杂 BE 的也有实力。没有 review 权限和非主流 Arch 的 port maintainer 相对就不是很 powerful
- 剩下的人就在 maillist 里面发 patch 并且等待 review accept 被 commit ，或者中间再让你改几次，要不就不理你或者拒绝你的代码

学习 gcc 的建议

- 有个明确的目标，针对目标去边做边学，解决问题的原理和方法很重要，而不是变化非常快的 gcc 代码，有了解决问题的方法之后再去查代码实现
- 找一个你感兴趣的模块，从目录名，文件名，函数名，变量名上理解代码才是好办法，找不到的可以 grep
- 代码的流程用 gdb 跟踪几遍，用笔和纸画下来

如何融入社区开发？

- 订阅 gcc 和 hellogcc 的 maillist，加入 gcc 和 hellogcc 的 irc
- svn 最新的代码， make check 发现并修正错误
- grep FIXME 或 TODO，根据自己的意愿和能力去做
- 很多 pass 实现的不够好，阅读对应 paper 去完善算法
- 做插件，邢明杰的 gcc-vcg-plugin 是一个很好的例子

如何去解 gcc 的 bug

- 发现了编译错误， internal error ，或者编译 OK 运行出错，怎么定位 bug ？
- 首先确定源代码没有问题，用其他编译器编译运行都 OK
- internal error 就根据信息去 gdb 吧，一般会给出代码退出在哪个文件的多少行，根据这个信息可以推测错误的原因

编译 OK 运行出错或者性能骤降或者其他诡异的问提如何定位 bug

- 对比排除， O1 O2 O3 的排除，发现问题在 O?
- 通过下面命令确定 O? 具体由多少 pass 组成
- touch empty.c
- gcc -O1 -S -fverbose-asm empty.c
- cat empty.s
- 最后一个小参数一个小参数的指定，确定问题的具体 pass

需要你来一起完善文档

- 完善 gccint 或者参与 gccint-zh 的翻译
- gccint 是个 manual，爱好者需要 tutorial 来入门
- 开发中的经验和 tips，希望你写出来投稿给 HelloGcc，或者在 ChinaUnix 的 CPU 与编译器版发帖

Link Time

- LTO 不是一种在 link time 的优化，而是在 link time 去进行一些优化， IPO 一般会设想在 High Level 实现，但是现在人们都是模块化编程， LLVM 的 LTO 是把 IR 放在 o 文件里面以供分析和转换

我们明年
HelloGcc2011 见