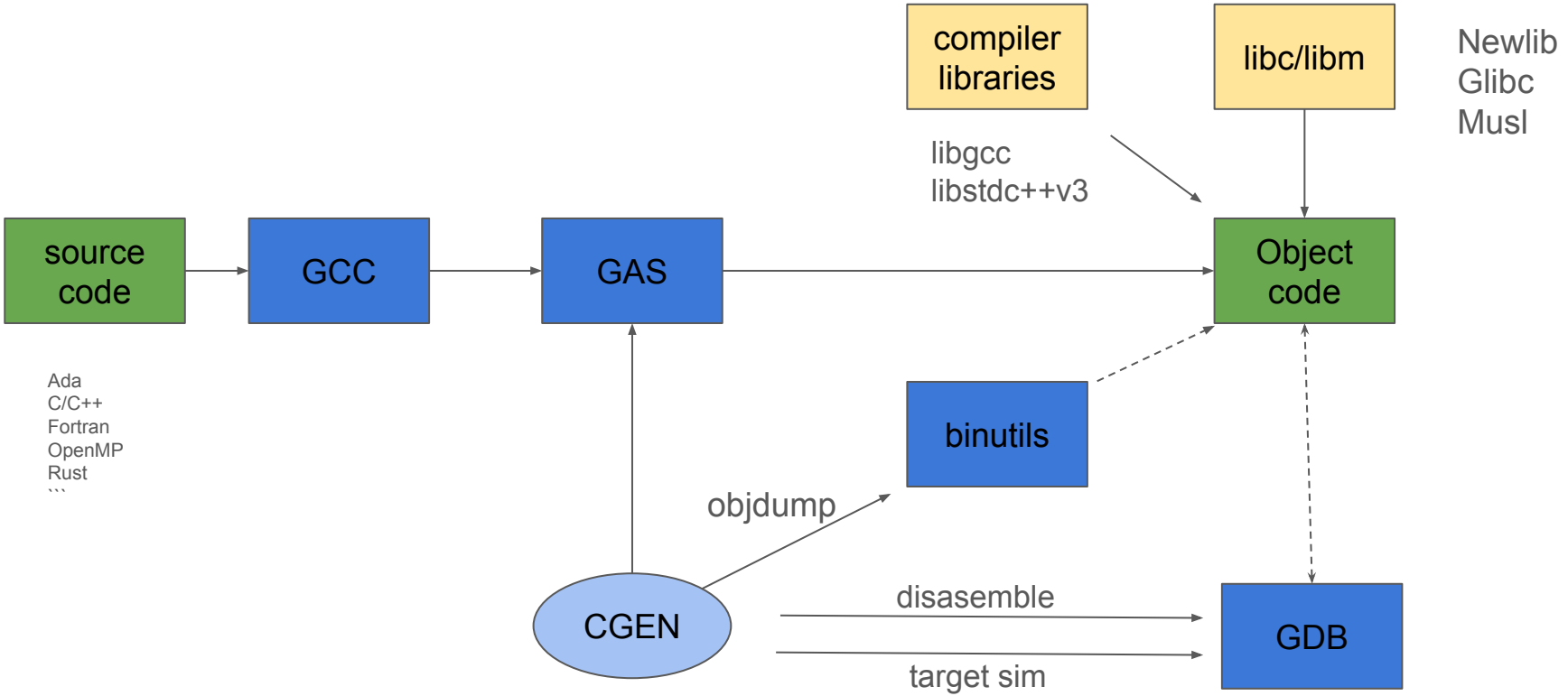
gnu gcc RISC-V 后端介绍

陈逸轩
wechat:XYenChi
github:https://github.com/XYenChi
blog:https://xyenchi.github.io/

参考书目和教程

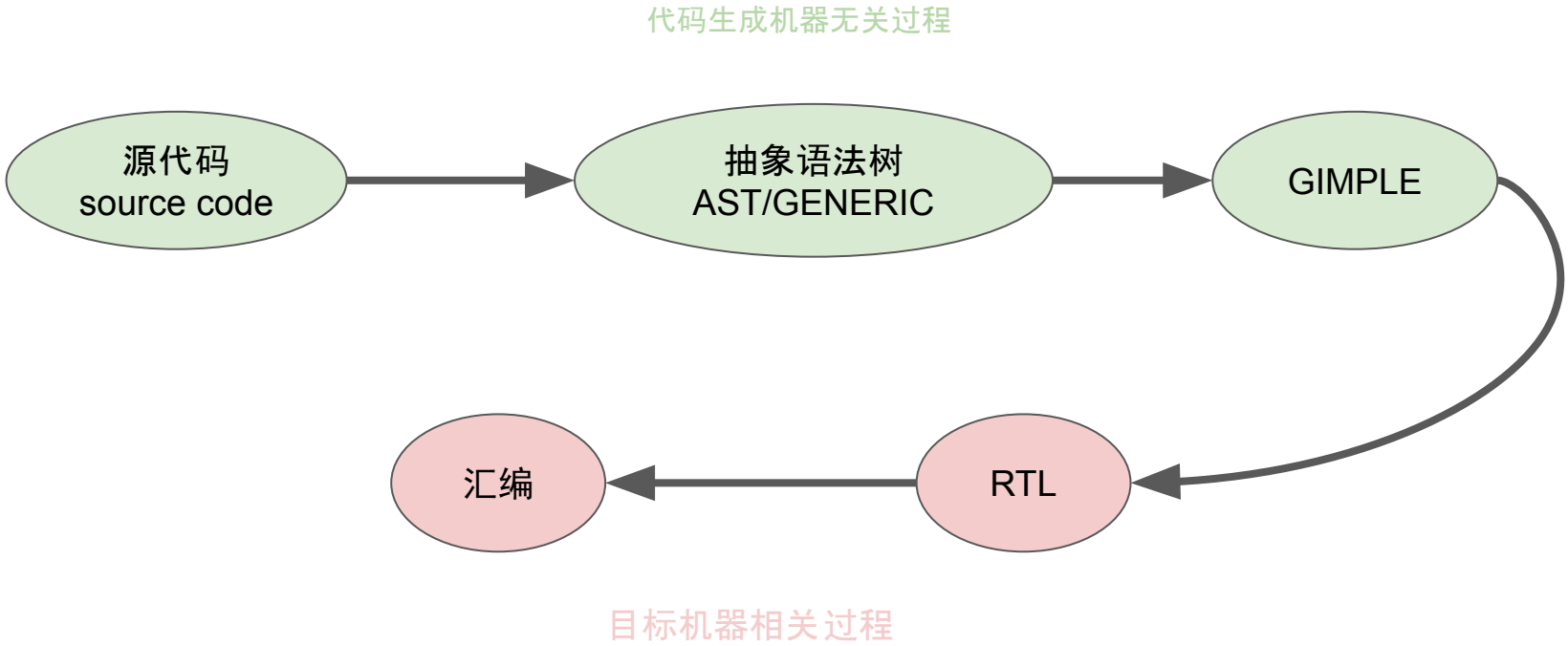
- 参考书目:《深入分析GCC》
- Jeremy Bennett在fosdem视频教程:https://fosdem.org/2024/schedule/event/fosdem-2024-2713-how-to-bring-up-gcc-for-your-new-chip/
- Jeremy Bennett在fosdem教程slides:https://fosdem.org/2024/events/attachments/fosdem-2024-2713-how-to-bring-up-gcc-for-your-new-chip/slides/22552/fosdem-2024-gcc-tutorial_qY63lhv.pdf
- vam架构的gcc实现源码仓库:https://github.com/embecosm/gcc-vam/commit/97863f37b4a844e596214f12d13f9b7e0b979b16#diff-9c966208fd0a0c8e24a1526da6904887c378283b9b645b9740c19339884174d9
- github仓库:https://github.com/gcc-mirror/gcc
- gcc manual:https://gcc.gnu.org/onlinedocs/gccint/
- gcc源码: https://gcc.gnu.org/git/gitweb.cgi?p=gcc.git

gnu toolchain 的组成



source:https://archive.fosdem.org/2024/events/attachments/fosdem-2024-2713-how-to-bring-up-gcc-for-your-new-chip/slides/22552/fosdem-2024-gcc-tutorial_qY63lhv.pdf

GCC 编译过程代码生成



GCC 中间表示（Intermediate Representations）

- GENERIC：上层语言对应的树
  - 中段转换开始的点
- GIMPLE：上层语言依赖的树
  - GENERIC的子集
  - 树转换（比如向量化，循环展开）
- RTL：底层三地址码
  - 三地址码转换

gcc文件结构

- gcc/
  - gcc/
    - config/
      - riscv/
      - …
  - lib*/
  - libstdc++-v3/
  - libgcc/

- gcc项目文件夹
  - 编译器所在目录
    - 架构相关代码配置文件夹
      - RISC-V相关代码
      - 其他架构相关代码
  - 一些gcc库
  - C++库
  - gcc底层运行时库

RISC-V相关代码路径

- gcc/gcc/config/riscv
  - riscv.h　　RISC-V相关宏定义。寄存器、字节顺序等。
  - riscv.cc　　生成RISC-V架构代码。
  - riscv.md　　RISC-V的机器描述（machine desicription）文件。
  - riscv.opt　　RISC-V选项定义文件。比如-march=
  - riscv-builtins.cc　　RISC-V内置函数
  - …

RISC-V 架构相关资料

spec releases：https://github.com/riscv/riscv-isa-manual/releases

查询工具网站：https://en.wikichip.org/wiki/risc-v/registers

riscv-gnu-toolchain：https://github.com/riscv-collab/riscv-gnu-toolchain

ps-abi：https://github.com/riscv-non-isa/riscv-elf-psabi-doc/releases/tag/v1.0

riscv.h (1)

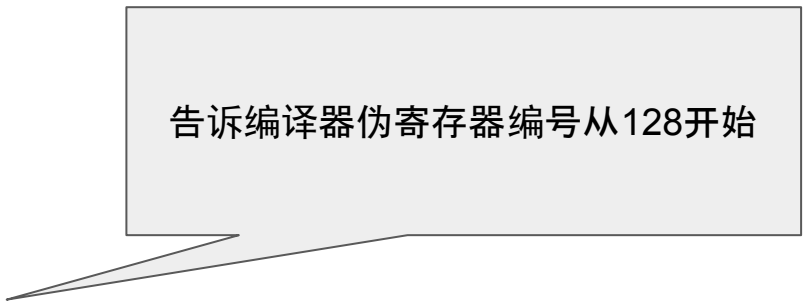
manual : https://gcc.gnu.org/onlinedocs/gccint/Register-Basics.html

```
/* Standard register usage.  */

/* Number of hardware registers.  We have:

   - 32 integer registers
   - 32 floating point registers
   - 2 fake registers:
        - ARG_POINTER_REGNUM
        - FRAME_POINTER_REGNUM
   - 1 vl register
   - 1 vtype register
   - 28 unused registers for future expansion
   - 32 vector registers  */

#define FIRST_PSEUDO_REGISTER 128
```



riscv.h (2)

告诉编译器有固定用途的寄存器是哪些，在一般情况下不要分配。

```
/* x0, ra, sp, gp, and tp are fixed.  */

#define FIXED_REGISTERS
{ /* General registers.  */
  1, 1, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
  0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
  /* Floating-point registers.  */
  0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
  0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
  /* Others.  */
  1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
  1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
  /* Vector registers.  */
  0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
  0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0
}
```

Register	ABI Name	Description
x0	zero	hardwired zero
x1	ra	return address
x2	sp	stack pointer
x3	gp	global pointer
x4	tp	thread pointer
x5	t0	temporary register 0
x6	t1	temporary register 1
x7	t2	temporary register 2
x8	s0 / fp	saved register 0 / frame pointer
x9	s1	saved register 1
x10	a0	function argument 0 / return value 0
x11	a1	function argument 1 / return value 1
x12	a2	function argument 2
x13	a3	function argument 3
x14	a4	function argument 4
x15	a5	function argument 5

riscv.h (3)

```
/* a0-a7, t0-t6, fa0-fa7, and ft0-ft11 are volatile across calls.
   The call RTLs themselves clobber ra.  */

#define CALL_USED_REGISTERS
{ /* General registers.  */
  1, 1, 1, 1, 1, 1, 1, 1, 0, 0, 1, 1, 1, 1, 1, 1,
  1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 1,
  /* Floating-point registers.  */
  1, 1, 1, 1, 1, 1, 1, 1, 0, 0, 1, 1, 1, 1, 1, 1,
  1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 1,
  /* Others.  */
  1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
  1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
  /* Vector registers.  */
  1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
  1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1
}
```

1表示不适合存函数 调用过程中仍需要保持不变的值。
0表示使用该寄存器的值编译器会自动在函数调用的入口保存，函数退出时恢复。

x8	s0 / fp	saved register 0 / frame pointer	-E
x9	s1	saved register 1	-E

riscv.opt(1)

文件路径：gcc/gcc/config/riscv/riscv.opt

- gcc 使用选项定义文件描述命令行选项，该文件以.opt结尾
- 不同选项内容之间用空行隔开，注释分号开头
- 选项名称都去掉了前面的“-”
- target mask 会在运行时自动给 target_flags 分配一个bit，并设置对应的 TARGET_x 宏，比如Mask(VECTOR)。

riscv.opt(2)

manual : https://gcc.gnu.org/onlinedocs/gccint/Option-properties.html

```
march=
Target RejectNegative Joined Negative(march=)
Var(riscv_arch_string) Save
-march=         Generate code for given RISC-V ISA (e.g. RV64IM).
ISA strings must be
lower-case.

march=help
Target RejectNegative
-march=help  Print supported -march extensions.
```

- Target表示该选项支持所有语言，但只适用于特定的目 标机器。
- RejectNegative表示该选项没有前面带“no-”的形式。
- Var(riscv_arch_string)会传给gcc/config/riscv/riscv.cc

riscv.cc
文件路径：gcc/gcc/config/riscv/riscv.cc

riscv.cc文件需要定义全局targetm变量，包含与机器相关的指向函数和数据的指针。为了所有的目标机器在默认情况下保持一致，gcc/gcc/target-def.h中预定义了一系列的宏，当与目标机器不一致时，在.cc文件中进行重新定义。

```
struct gcc_target targetm = TARGET_INITIALIZER;
```

```
/* Initialize the GCC target structure.  */
#undef TARGET_ASM_ALIGNED_HI_OP
#define TARGET_ASM_ALIGNED_HI_OP "\t.half\t"
#undef TARGET_ASM_ALIGNED_SI_OP
#define TARGET_ASM_ALIGNED_SI_OP "\t.word\t"
#undef TARGET_ASM_ALIGNED_DI_OP
```

gcc/gcc/target-def.h

```
#define TARGET_ASM_ALIGNED_HI_OP "\t.short\t"
#define TARGET_ASM_ALIGNED_SI_OP "\t.long\t"
#define TARGET_ASM_ALIGNED_DI_OP NULL
```

gcc machine modes

gcc/machmode.def

manual : https://gcc.gnu.org/onlinedocs/gcc-4.8.1/gccint/Machine-Modes.html

machine mode	全称	bit 数
qi/QI	quarter-integer	8
hi/HI	half-integer	16
si/SI	single-integer	32
di/DI	double-integer	64
ti/TI	tetra-integer	128
sf/SF	single-float	32
df/DF	double-float	64

把 32 bit 当成计量单位，四分之一（quarter）、二分之一（half）、双倍（double）、四倍（tetra）。

机器描述(machine description)文件

gcc 手册：https://gcc.gnu.org/onlinedocs/gccint/Machine-Desc.html

文件路径：gcc/gcc/config/riscv/

机器描述：定义指令模板、常量、属性、断言、约束、枚举器、流水线和优化信息

gcc/gcc/config/riscv/riscv.md

gcc/gcc/config/riscv/constraints.md

gcc/gcc/config/riscv/predicates.md

gcc/gcc/config/riscv/iterator.md

gcc/gcc/config/riscv/peephole.md

- 使用 Scheme-like 语言编写
  - 复用了 RTL 表达式、机器模式和语法
- 在编译时被解析
  - 生成 C 指令选择器和模版匹配器

指令模板

(define_insn 指令模板名称

RTL 模板

条件

输出模板

属性

)

```
(define_insn "addsi3"
  [(set (match_operand:SI 0 "register_operand" "=r,r")
        (plus:SI (match_operand:SI 1 "register_operand" " r,r")
                 (match_operand:SI 2 "arith_operand" " r,I")))]
  ""
  "add%i2%~\t%0,%1,%2"
  [(set_attr "type" "arith")
   (set_attr "mode" "SI")])
```

常见的约束(constraint)

文件路径：gcc/gcc/config/riscv/constraints.md

=:操作数只写

+:操作可读可写

I: 12 bits 有符号整型立即数

r:寄存器

f: 浮点寄存器

```
(define_constraint "I"
  "An I-type 12-bit signed immediate."
  (and (match_code "const_int")
       (match_test "SMALL_OPERAND (ival)")))
```

指令模板（Insn Pattern）定义

文件路径：gcc/gcc/config/riscv/riscv.md

```
(define_expand "addsi3"
 [(set (match_operand:SI          0 "register_operand"
"=r,r")
 (plus:SI (match_operand:SI 1 "register_operand" " r,r")
    (match_operand:SI 2 "arith_operand"    " r,I")))]
 ""
{
 if (TARGET_64BIT)
   {
     rtx t = gen_reg_rtx (DImode);
     emit_insn (gen_addsi3_extended (t, operands[1],
operands[2]));
     t = gen_lowpart (SImode, t);
     SUBREG_PROMOTED_VAR_P (t) = 1;
     SUBREG_PROMOTED_SET (t, SRP_SIGNED);
     emit_move_insn (operands[0], t);
     DONE;
   }
})
```

```
(define_insn "add<mode>3"
 [(set (match_operand:ANYF            0 "register_operand" "=f")
 (plus:ANYF (match_operand:ANYF 1 "register_operand" " f")
      (match_operand:ANYF 2 "register_operand" " f")))]
 "TARGET_HARD_FLOAT || TARGET_ZFINX"
 "fadd.<fmt>\t%0,%1,%2"
 [(set_attr "type" "fadd")
  (set_attr "mode" "<UNITMODE>")])
```

查看编译过程的中间产物

在将.c程序编译成.s的汇编程序时带上 -fdump-rtl-expand

$ gcc -S hello.c -fdump-rtl-expand

会生成 hello.c.245r.expand 文件显示所有生成的 rtl

$ gcc -S hello.c -dp

会在生成的.s文件中显示注释

其他参数:

-fverbose-asm

-fdump-tree-all -fdump-ipa-all -fdump-rtl-all

谢谢大家收看

交流环节