编译技术入门与实战——RISCV-GNU-GCC (GCC指令添加)

PLCT实验室

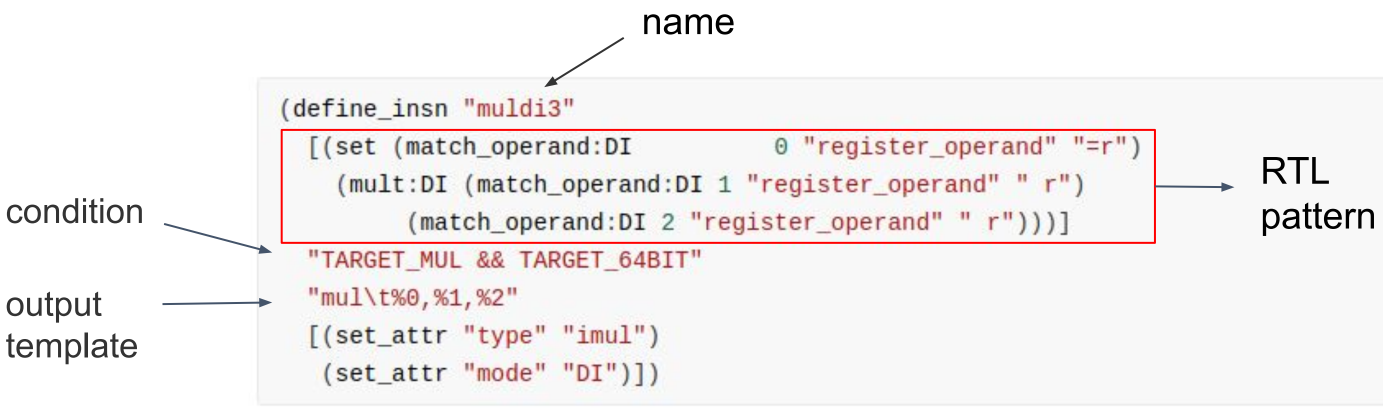
sinan@iscas.ac.cn

2021/10/13

GCC指令添加

1. intrincis接口添加
2. 迭代器使用
3. 标准模板名指令的使用
4. 自定义constraint, predicate
5. 寄存器限制
6. 添加一个条件跳转指令

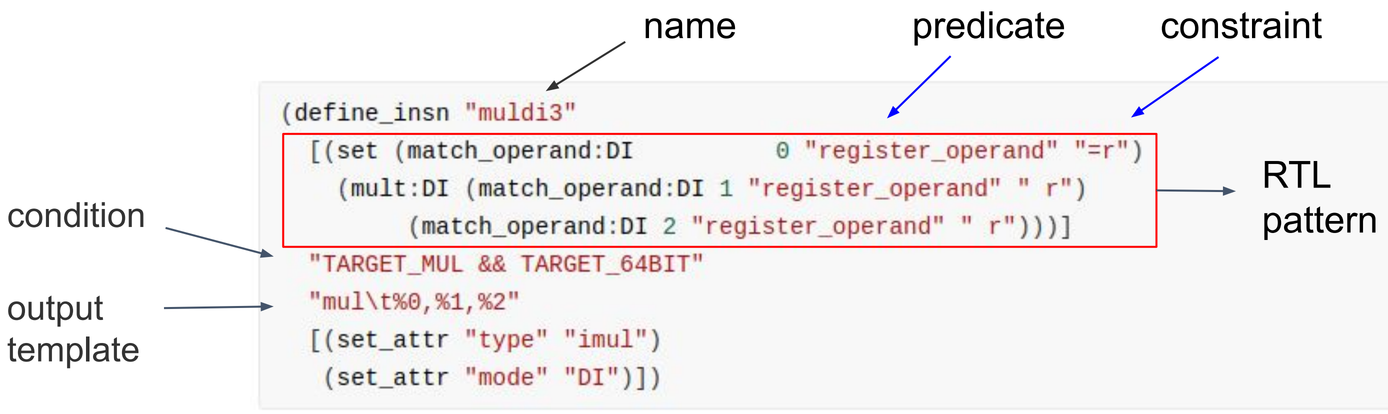
GCC指令添加: 回顾



```
(define_insn "muldi3"
  [(set (match_operand:DI          0 "register_operand" "=r")
    (mult:DI (match_operand:DI 1 "register_operand" " r")
         (match_operand:DI 2 "register_operand" " r")))]
  "TARGET_MUL && TARGET_64BIT"
  "mul\t%0,%1,%2"
  [(set_attr "type" "imul")
   (set_attr "mode" "DI")])
```

muldi3 pattern in riscv.md

GCC指令添加: 回顾



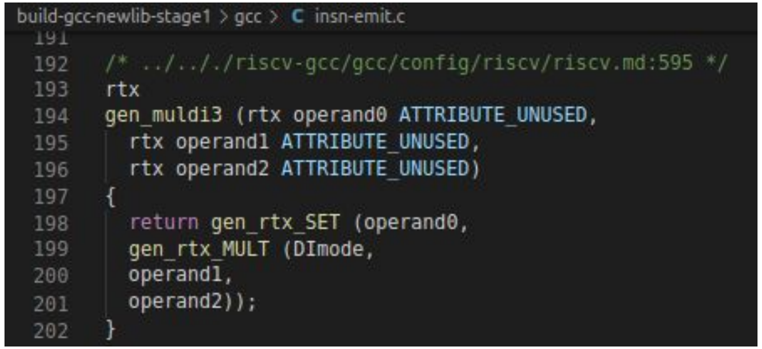
```
(define_insn "muldi3"
  [(set (match_operand:DI          0 "register_operand" "=r")
    (mult:DI (match_operand:DI 1 "register_operand" " r")
         (match_operand:DI 2 "register_operand" " r")))]
  "TARGET_MUL && TARGET_64BIT"
  "mul\t%0,%1,%2"
  [(set_attr "type" "imul")
   (set_attr "mode" "DI")])
```

muldi3 pattern in riscv.md

GCC指令添加: 回顾

```
(define_insn "muldi3"
  [(set (match_operand:DI          0 "register_operand" "=r")
    (mult:DI (match_operand:DI 1 "register_operand" " r")
         (match_operand:DI 2 "register_operand" " r")))]
  "TARGET_MUL && TARGET_64BIT"
  "mul\t%0,%1,%2"
  [(set_attr "type" "imul")
   (set_attr "mode" "DI")])
```

muldi3 pattern in riscv.md


build-gcc-newlib-stage1 > gcc > C insn-emit.c

```
/* ../.././riscv-gcc/gcc/config/riscv/riscv.md:595 */
rtx
gen_muldi3 (rtx operand0 ATTRIBUTE_UNUSED,
	rtx operand1 ATTRIBUTE_UNUSED,
	rtx operand2 ATTRIBUTE_UNUSED)
{
  return gen_rtx_SET (operand0,
	gen_rtx_MULT (DImode,
	operand1,
	operand2));
}
```

后端机器描述实现的指令模板会在GCC构建的时候生成相应的C函数

emit_insn (gen_muldi3 (dst, src1, src2));

通过emit_insn把对应的rtl表达式插入insn链表中
*gcc/emit-rtl.c

GCC指令添加:

mul rd, rs1, rs2

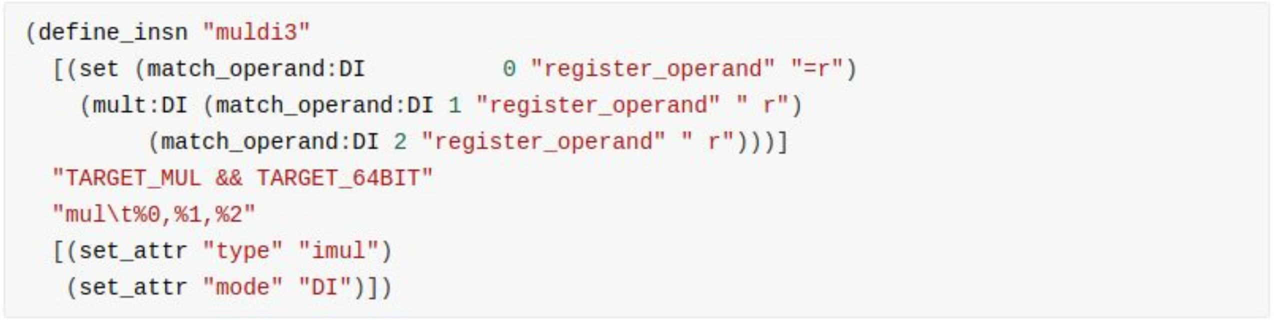
muli rd, rs1, imm

假如想要在ISA添加一个muli指令，对立即数的乘法进行支持

GCC指令添加: 添加一个新的指令

1. md文件中添加指令模板

Q: 如何根据mul在rv64im的指令pattern来修改得到想要的muli？



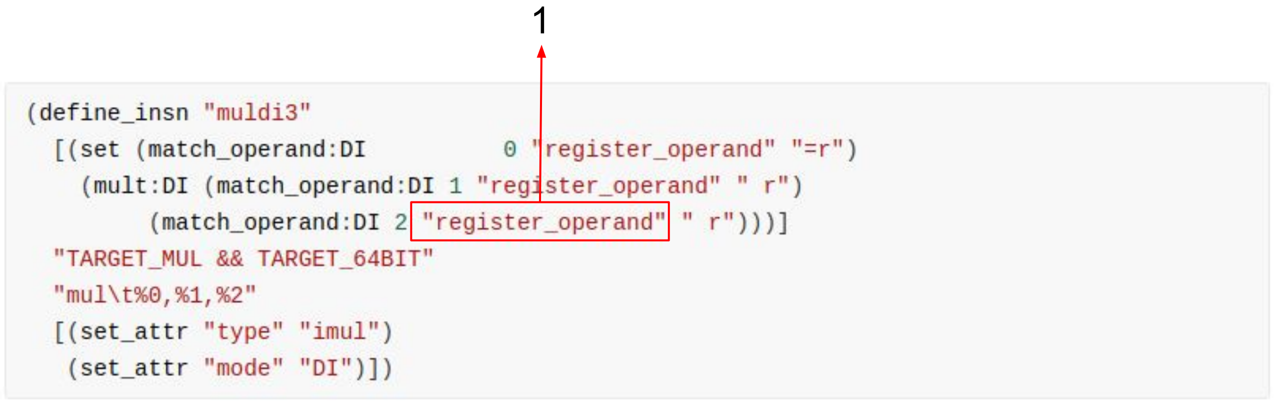
```
(define_insn "muldi3"
  [(set (match_operand:DI          0 "register_operand" "=r")
    (mult:DI (match_operand:DI 1 "register_operand" " r")
         (match_operand:DI 2 "register_operand" " r")))]
  "TARGET_MUL && TARGET_64BIT"
  "mul\t%0,%1,%2"
  [(set_attr "type" "imul")
   (set_attr "mode" "DI")])
```

mul指令pattern

GCC指令添加: 添加一个新的指令

1. md文件中添加指令模板

Predicate:
register_operand
const_int_operand
immediate_operand
memory_operand
… you can even customize one!
(predicates.md)



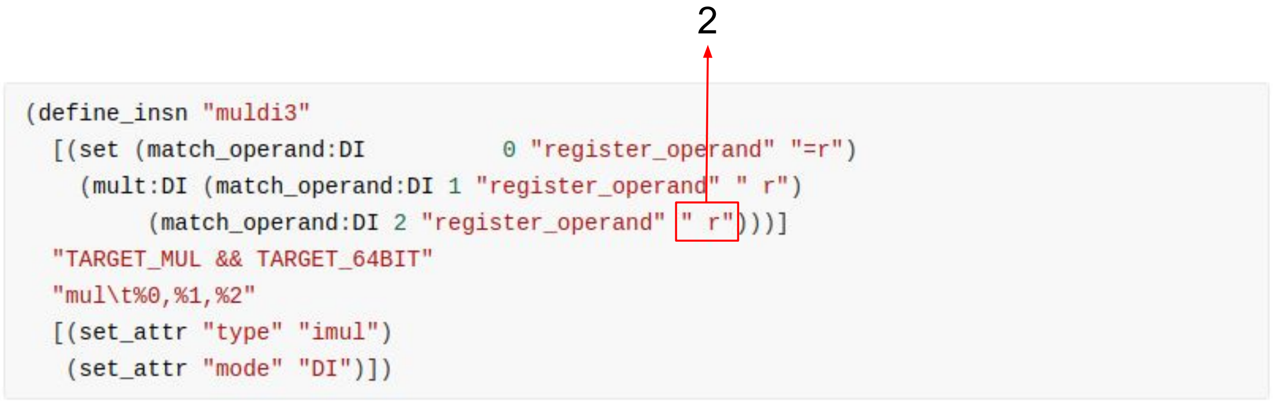
```
(define_insn "muldi3"
  [(set (match_operand:DI          0 "register_operand" "=r")
    (mult:DI (match_operand:DI 1 "register_operand" " r")
         (match_operand:DI 2 "register_operand" " r"))))]
  "TARGET_MUL && TARGET_64BIT"
  "mul\t%0,%1,%2"
  [(set_attr "type" "imul")
   (set_attr "mode" "DI")])
```

mul指令pattern

GCC指令添加: 添加一个新的指令

1. md文件中添加指令模板

Predicate:
r
m
I
… you also can customize it!



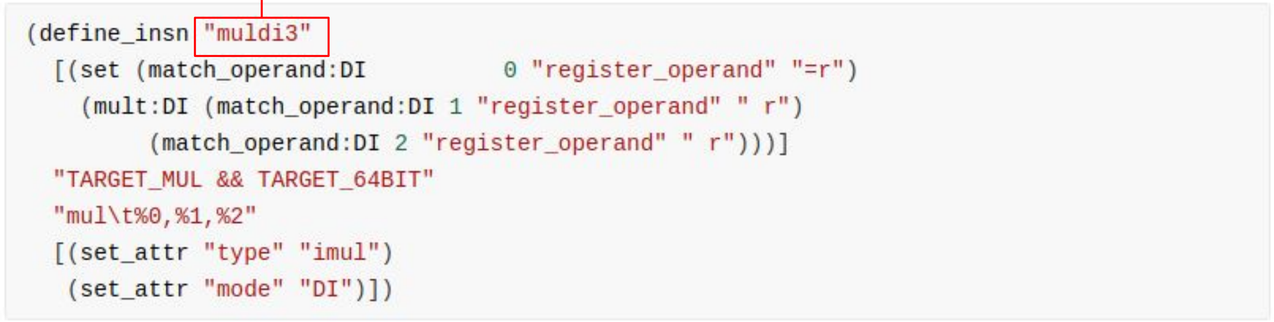
```
(define_insn "muldi3"
  [(set (match_operand:DI          0 "register_operand" "=r")
    (mult:DI (match_operand:DI 1 "register_operand" " r")
         (match_operand:DI 2 "register_operand" " r")))]
  "TARGET_MUL && TARGET_64BIT"
  "mul\t%0,%1,%2"
  [(set_attr "type" "imul")
   (set_attr "mode" "DI")])
```

mul指令pattern

GCC指令添加: 添加一个新的指令

1. md文件中添加指令模板

3



```
(define_insn "muldi3"
  [(set (match_operand:DI           0 "register_operand" "=r")
    (mult:DI (match_operand:DI 1 "register_operand" " r")
         (match_operand:DI 2 "register_operand" " r")))]
  "TARGET_MUL && TARGET_64BIT"
  "mul\t%0,%1,%2"
  [(set_attr "type" "imul")
   (set_attr "mode" "DI")])
```

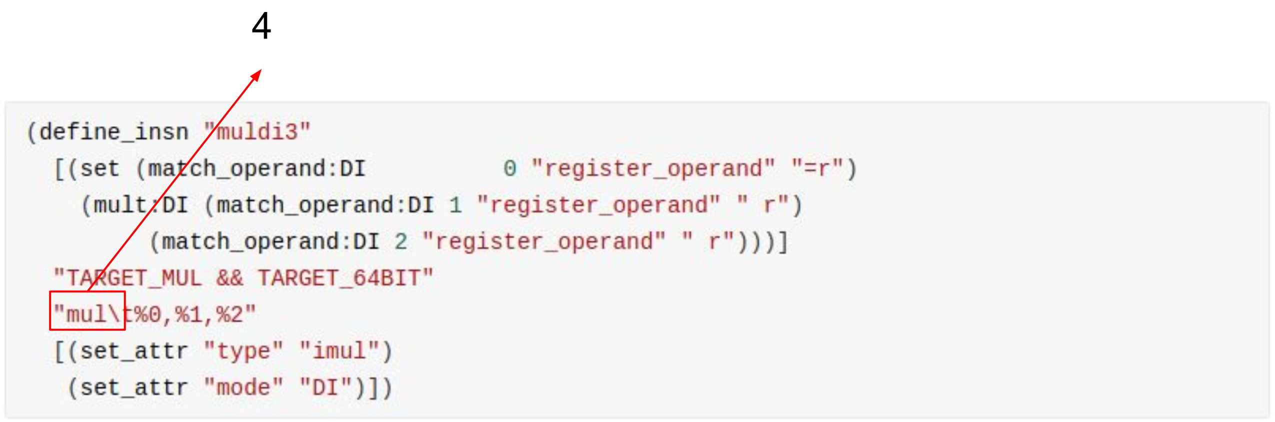
insn patterns cannot have duplicate name!

mul指令pattern

GCC指令添加: 添加一个新的指令

1. md文件中添加指令模板

generate muli when operand 2 is an immediate

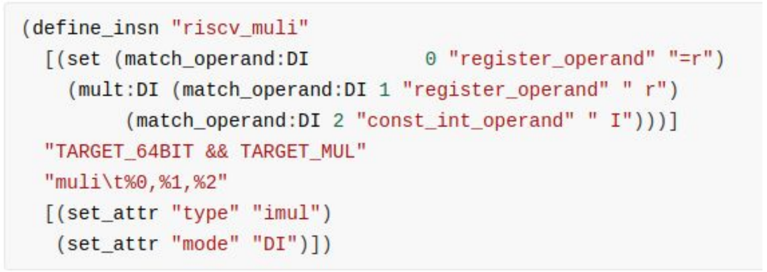


```
(define_insn "muldi3"
  [(set (match_operand:DI          0 "register_operand" "=r")
    (mult:DI (match_operand:DI 1 "register_operand" " r")
         (match_operand:DI 2 "register_operand" " r")))]
  "TARGET_MUL && TARGET_64BIT"
  "mul\t%0,%1,%2"
  [(set_attr "type" "imul")
   (set_attr "mode" "DI")])
```

mul指令pattern

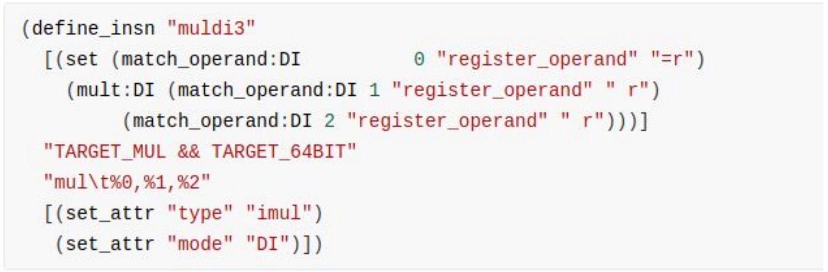
GCC指令添加: 添加一个新的指令

1. md文件中添加指令模板



```
(define_insn "riscv_muli"
  [(set (match_operand:DI          0 "register_operand" "=r")
    (mult:DI (match_operand:DI 1 "register_operand" " r")
          (match_operand:DI 2 "const_int_operand" " I")))]
  "TARGET_64BIT && TARGET_MUL"
  "muli\t%0,%1,%2"
  [(set_attr "type" "imul")
   (set_attr "mode" "DI")])
```

muli指令pattern

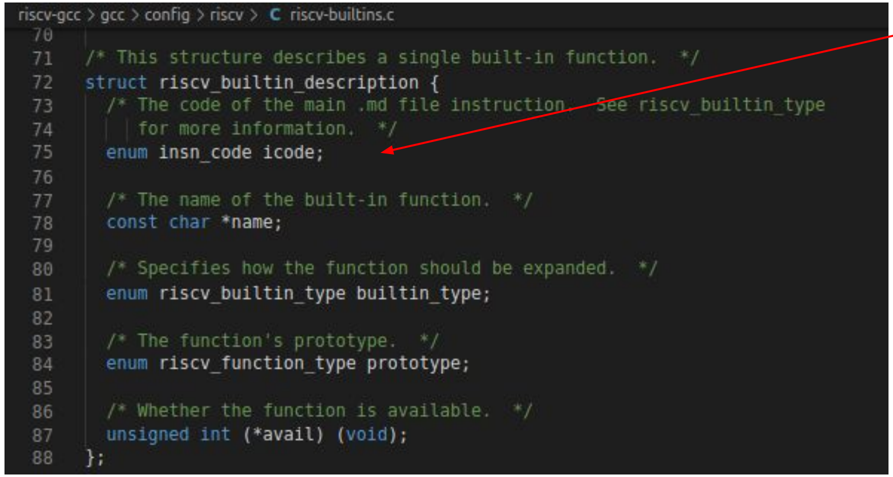


```
(define_insn "muldi3"
  [(set (match_operand:DI          0 "register_operand" "=r")
    (mult:DI (match_operand:DI 1 "register_operand" " r")
          (match_operand:DI 2 "register_operand" " r")))]
  "TARGET_MUL && TARGET_64BIT"
  "mul\t%0,%1,%2"
  [(set_attr "type" "imul")
   (set_attr "mode" "DI")])
```

mul指令pattern

GCC指令添加: 添加一个新的指令

2. 添加intrinsic接口(optional)

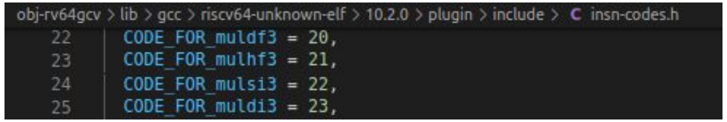


riscv-gcc > gcc > config > riscv > C riscv-builtins.c

```c

/* This structure describes a single built-in function.  */
struct riscv_builtin_description {
  /* The code of the main .md file instruction.  See riscv_builtin_type
     for more information.  */
  enum insn_code icode;

  /* The name of the built-in function.  */
  const char *name;

  /* Specifies how the function should be expanded.  */
  enum riscv_builtin_type builtin_type;

  /* The function's prototype.  */
  enum riscv_function_type prototype;

  /* Whether the function is available.  */
  unsigned int (*avail) (void);
};
```

riscv-builtin.c

CODE_FOR_riscv_muli

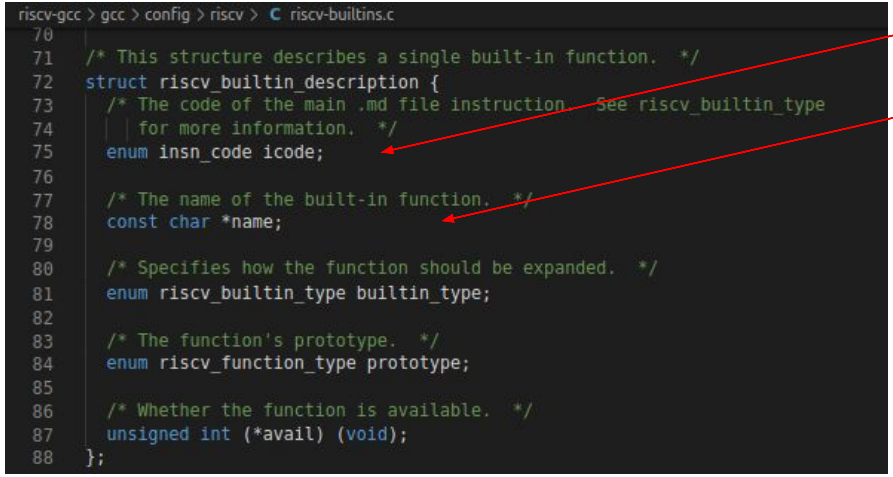


obj-rv64gcv > lib > gcc > riscv64-unknown-elf > 10.2.0 > plugin > include > C insn-codes.h

```c
  CODE_FOR_muldf3 = 20,
  CODE_FOR_mulhf3 = 21,
  CODE_FOR_mulsi3 = 22,
  CODE_FOR_muldi3 = 23,
```

GCC指令添加: 添加一个新的指令

2. 添加intrinsic接口(optional)



riscv-gcc > gcc > config > riscv > C riscv-builtins.c

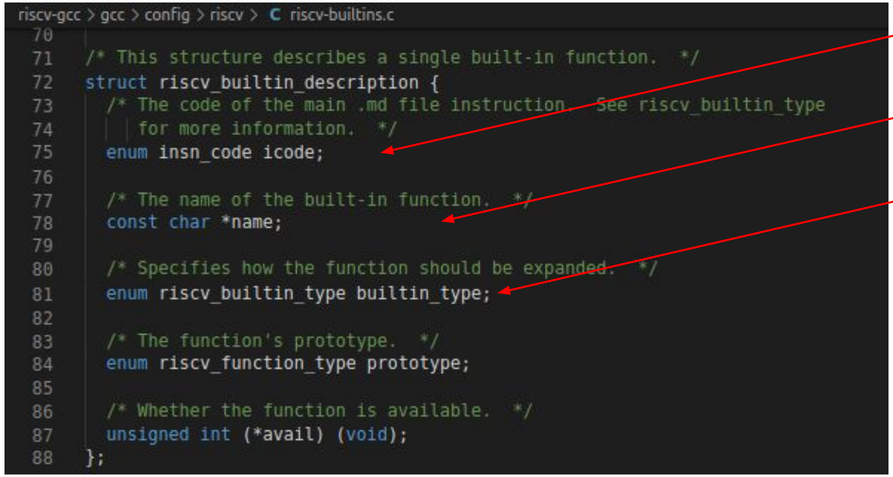
```

/* This structure describes a single built-in function.  */
struct riscv_builtin_description {
  /* The code of the main .md file instruction.  See riscv_builtin_type
     for more information.  */
  enum insn_code icode;

  /* The name of the built-in function.  */
  const char *name;

  /* Specifies how the function should be expanded.  */
  enum riscv_builtin_type builtin_type;

  /* The function's prototype.  */
  enum riscv_function_type prototype;

  /* Whether the function is available.  */
  unsigned int (*avail) (void);
};
```

CODE_FOR_riscv_muli

__builtin_riscv_muli (instrisic函数名)

GCC指令添加: 添加一个新的指令

2. 添加intrinsic接口(optional)



riscv-gcc > gcc > config > riscv > C riscv-builtins.c

```c

/* This structure describes a single built-in function.  */
struct riscv_builtin_description {
  /* The code of the main .md file instruction.  See riscv_builtin_type
     for more information.  */
  enum insn_code icode;

  /* The name of the built-in function.  */
  const char *name;

  /* Specifies how the function should be expanded.  */
  enum riscv_builtin_type builtin_type;

  /* The function's prototype.  */
  enum riscv_function_type prototype;

  /* Whether the function is available.  */
  unsigned int (*avail) (void);
};
```

CODE_FOR_riscv_muli

__builtin_riscv_muli (instrisic函数名)

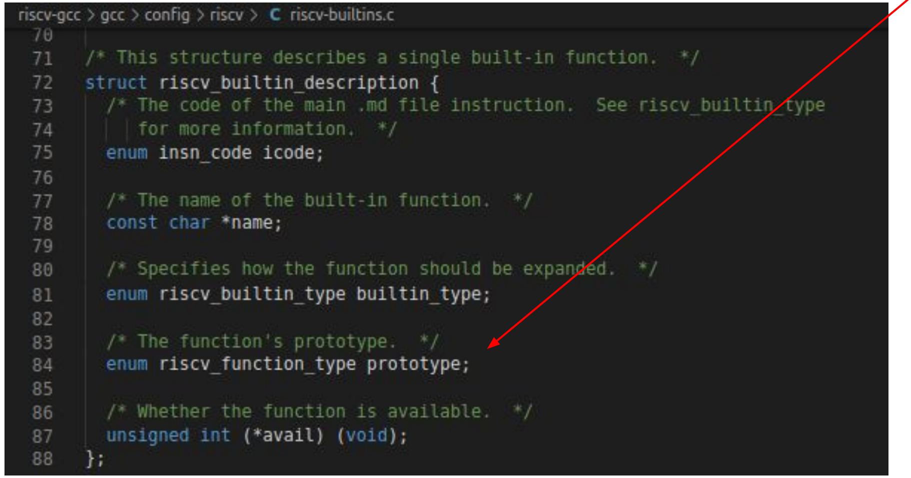
builtin函数类型

RISCV_BUILTIN_DIRECT
RISCV_BUILTIN_DIRECT_NO_TARGET
(.md中的指令pattern不发生赋值)

GCC指令添加: 添加一个新的指令

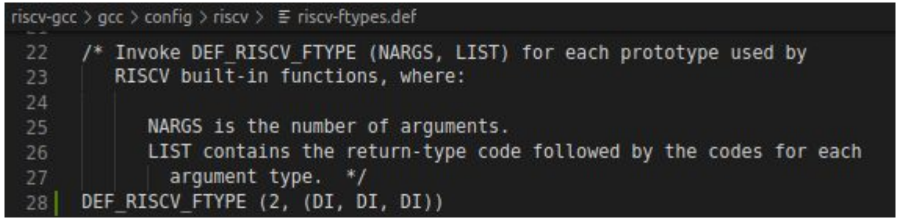
2. 添加intrinsic接口(optional)

insn pattern的函数签名



riscv-gcc > gcc > config > riscv > riscv-builtins.c

```
/* This structure describes a single built-in function.  */
struct riscv_builtin_description {
  /* The code of the main .md file instruction.  See riscv_builtin_type
     for more information.  */
  enum insn_code icode;

  /* The name of the built-in function.  */
  const char *name;

  /* Specifies how the function should be expanded.  */
  enum riscv_builtin_type builtin_type;

  /* The function's prototype.  */
  enum riscv_function_type prototype;

  /* Whether the function is available.  */
  unsigned int (*avail) (void);
};
```



riscv-gcc > gcc > config > riscv > riscv-ftypes.def

```
/* Invoke DEF_RISCV_FTYPE (NARGS, LIST) for each prototype used by
   RISCV built-in functions, where:

     NARGS is the number of arguments.
     LIST contains the return-type code followed by the codes for each
       argument type.  */
DEF_RISCV_FTYPE (2, (DI, DI, DI))
```

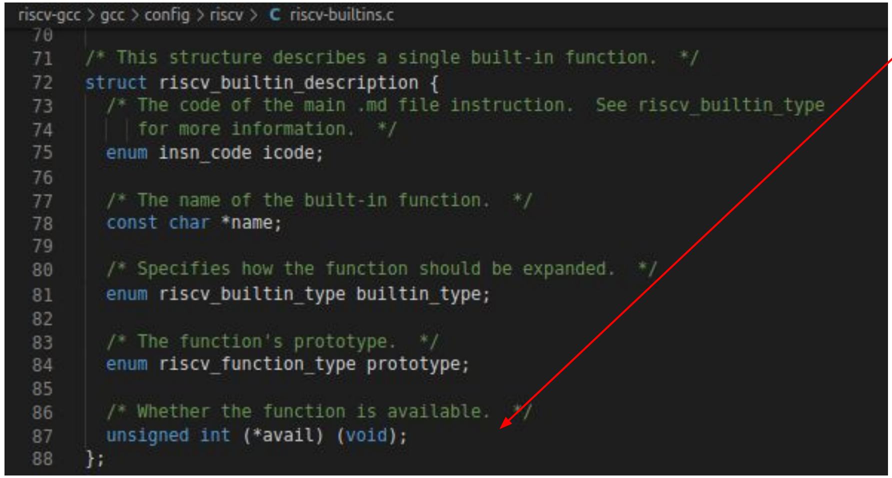
riscv-ftypes.def

DEF_RISCV_FTYPE (2, (DI, DI, DI))
expand to
RISCV_DI_FTYPE_DI_DI

, where DI = RISCV_ATYPE_DI = intDI_type_node(type node in tree.h)

GCC指令添加: 添加一个新的指令

2. 添加intrinsic接口(optional)

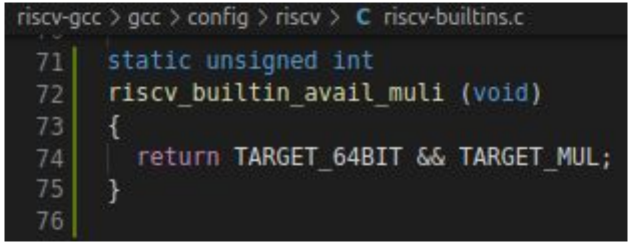


riscv-gcc > gcc > config > riscv > C riscv-builtins.c

```c
/* This structure describes a single built-in function.  */
struct riscv_builtin_description {
  /* The code of the main .md file instruction.  See riscv_builtin_type
     for more information.  */
  enum insn_code icode;

  /* The name of the built-in function.  */
  const char *name;

  /* Specifies how the function should be expanded.  */
  enum riscv_builtin_type builtin_type;

  /* The function's prototype.  */
  enum riscv_function_type prototype;

  /* Whether the function is available.  */
  unsigned int (*avail) (void);
};
```

riscv-builtin.c

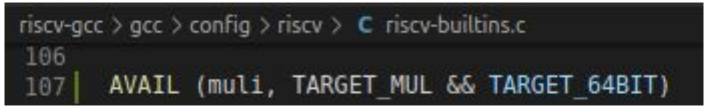
判别intrinsic是否可用的函数



riscv-gcc > gcc > config > riscv > C riscv-builtins.c

```c
static unsigned int
riscv_builtin_avail_muli (void)
{
  return TARGET_64BIT && TARGET_MUL;
}
```

AVAIL宏快速生成avail判别函数

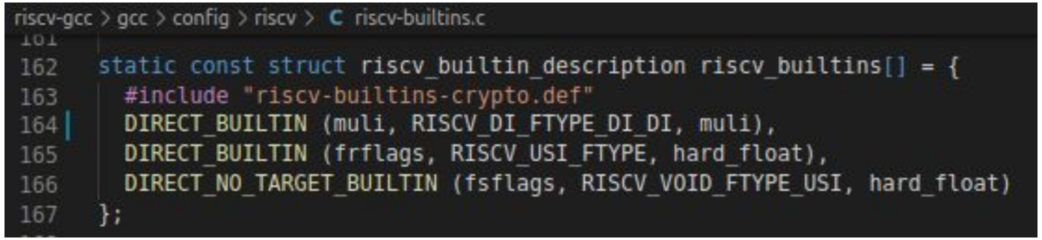


riscv-gcc > gcc > config > riscv > C riscv-builtins.c

```c
AVAIL (muli, TARGET_MUL && TARGET_64BIT)
```

GCC指令添加: 添加一个新的指令

2. 添加intrinsic接口(optional)



riscv-gcc > gcc > config > riscv > riscv-builtins.c

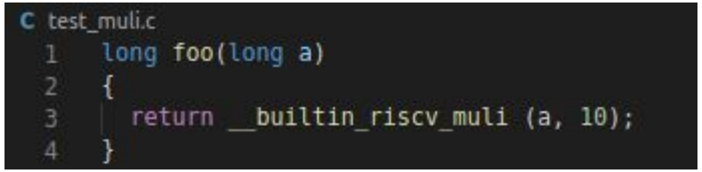
```c
static const struct riscv_builtin_description riscv_builtins[] = {
  #include "riscv-builtins-crypto.def"
  DIRECT_BUILTIN (muli, RISCV_DI_FTYPE_DI_DI, muli),
  DIRECT_BUILTIN (frflags, RISCV_USI_FTYPE, hard_float),
  DIRECT_NO_TARGET_BUILTIN (fsflags, RISCV_VOID_FTYPE_USI, hard_float)
};
```

riscv-builtin.c

*使用DIRECT_BUILTIN宏，默认指令模板名字前缀为riscv_

GCC指令添加: 添加一个新的指令

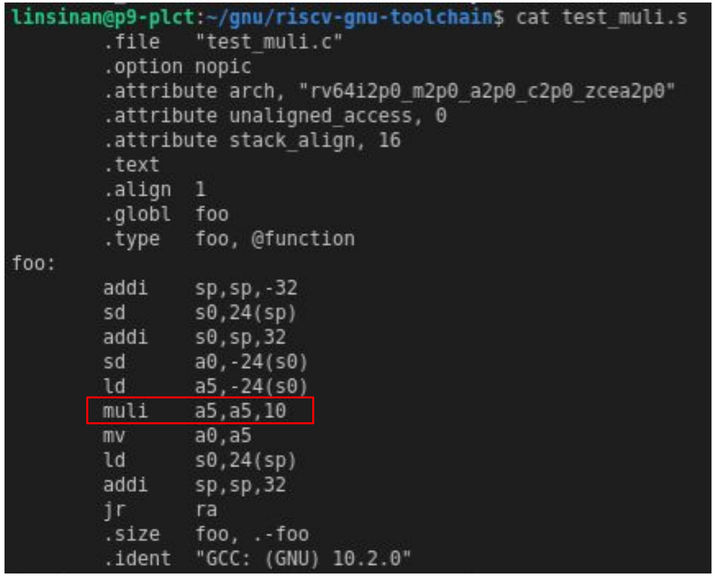
3. 测试一下muli指令



```c
test_muli.c
long foo(long a)
{
  return __builtin_riscv_muli (a, 10);
}
```

test_muli.c

./obj-tutorial/bin/riscv64-unknown-elf-gcc -S test_muli.c



```
linsinan@p9-plct:~/gnu/riscv-gnu-toolchain$ cat test_muli.s
        .file   "test_muli.c"
        .option nopic
        .attribute arch, "rv64i2p0_m2p0_a2p0_c2p0_zcea2p0"
        .attribute unaligned_access, 0
        .attribute stack_align, 16
        .text
        .align  1
        .globl  foo
        .type   foo, @function
foo:
        addi    sp,sp,-32
        sd      s0,24(sp)
        addi    s0,sp,32
        sd      a0,-24(s0)
        ld      a5,-24(s0)
        muli    a5,a5,10
        mv      a0,a5
        ld      s0,24(sp)
        addi    sp,sp,32
        jr      ra
        .size   foo, .-foo
        .ident  "GCC: (GNU) 10.2.0"
```

GCC指令添加: SPN (standard pattern name)

GCC提供了一系列的指令名称，叫 标准指令名称，后端通过实现带有标准指令名称的指令pattern，可以让GCC更好地实现指令生产。

```
movm  e.g. movsi   两个SI类型数据间的复制
mulm3 e.g. muldi3  两个DI类型数据的乘法和结果赋值
addm3 e.g. addsi3  两个SI类型数据间的加法和结果赋值
…(full list)
```

GCC指令添加: SPN (standard pattern name)

GCC提供了一系列的指令名称，叫 标准指令名称，后端通过实现带有标准指令名称的指令 pattern，可以让GCC更好地实现指令生成。

movm	e.g.	movsi	两个SI类型数据间的复制
mulm3	e.g.	muldi3	两个DI类型数据的乘法和结果赋值
addm3	e.g.	addsi3	两个SI类型数据间的加法和结果赋值

…(full list)

Q: 在后端中SPN是必须必须实现的吗？如果riscv.md没有实现muldi3会发生什么？

GCC指令添加: SPN (standard pattern name)

GCC提供了一系列的指令名称，叫 标准指令名称，后端通过实现带有标准指令名称的指令 pattern，可以让GCC更好地实现指令生成。

```
movm  e.g. movsi   两个SI类型数据间的复制
mulm3 e.g. muldi3  两个DI类型数据的乘法和结果赋值
addm3 e.g. addsi3  两个SI类型数据间的加法和结果赋值
…(full list)
```

Q: 在后端中SPN是必须必须实现的吗？如果riscv.md没有实现muldi3会发生什么？

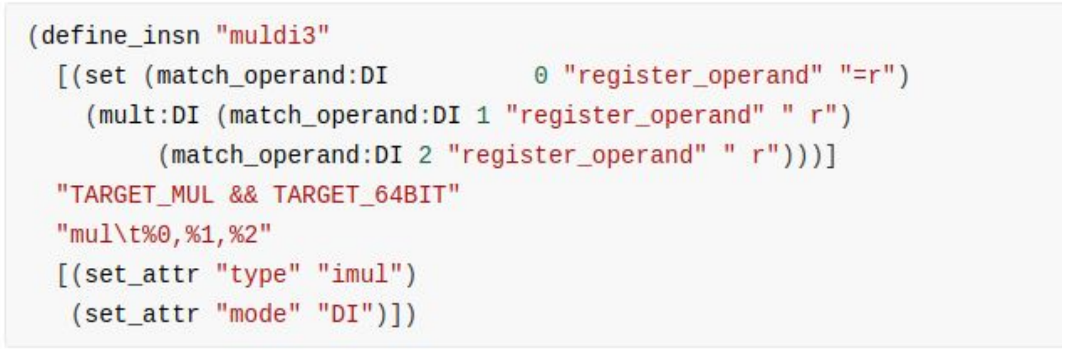
```
if implmented:
  emit_insn (gen_muldi3 (dst, src1, src2)); -> 直接生产mul指令
else:
  调用libgcc运行库中的__muldi函数
```

$\longleftarrow$ 更多的指令& 函数调用的overhead😅

GCC指令添加: 添加一个SPN的指令

1. 找到标准名muldi3并修改

1. predicate
2. contraints
3. output template



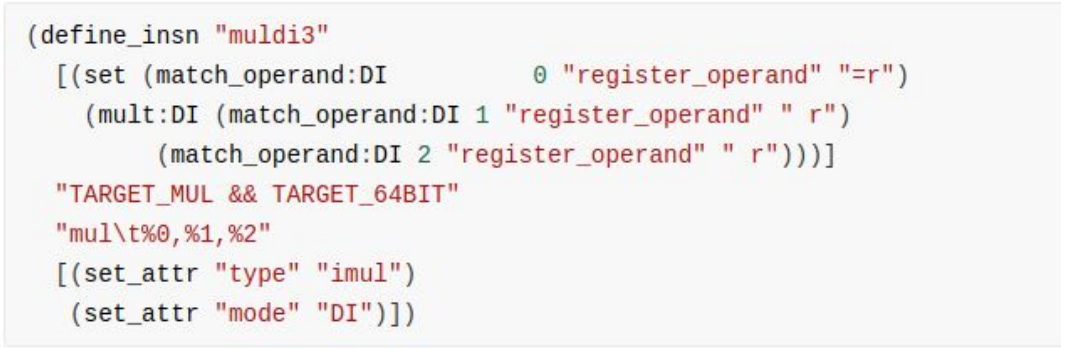
```
(define_insn "muldi3"
  [(set (match_operand:DI           0 "register_operand" "=r")
    (mult:DI (match_operand:DI 1 "register_operand" " r")
         (match_operand:DI 2 "register_operand" " r")))]
  "TARGET_MUL && TARGET_64BIT"
  "mul\t%0,%1,%2"
  [(set_attr "type" "imul")
   (set_attr "mode" "DI")])
```

mul指令pattern

GCC指令添加: 添加一个SPN的指令

1. 找到标准名muldi3并修改

output template:
- 字符串形式
- C语言形式



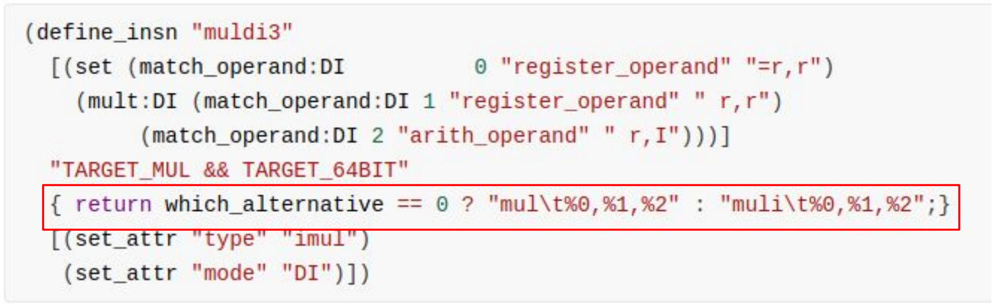
```
(define_insn "muldi3"
  [(set (match_operand:DI          0 "register_operand" "=r")
    (mult:DI (match_operand:DI 1 "register_operand" " r")
         (match_operand:DI 2 "register_operand" " r")))]
  "TARGET_MUL && TARGET_64BIT"
  "mul\t%0,%1,%2"
  [(set_attr "type" "imul")
   (set_attr "mode" "DI")])
```

mul指令pattern

GCC指令添加: 添加一个SPN的指令

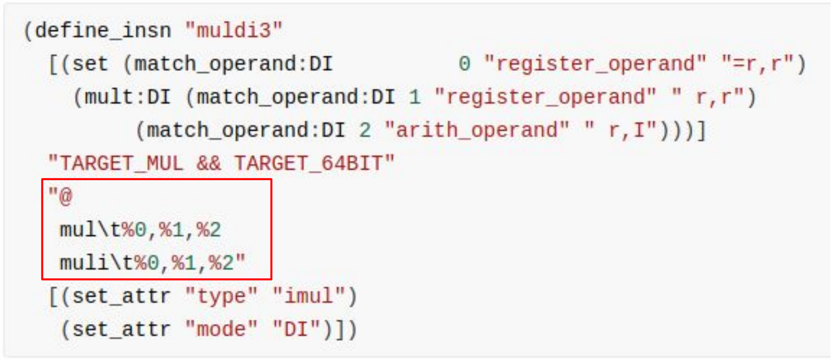
2. 实现指令模板对指令生成的选择

通过which_alternative 和@，基于constraint对指令生成进行微调



```
(define_insn "muldi3"
  [(set (match_operand:DI           0 "register_operand" "=r,r")
    (mult:DI (match_operand:DI 1 "register_operand" " r,r")
         (match_operand:DI 2 "arith_operand" " r,I")))]
  "TARGET_MUL && TARGET_64BIT"
  { return which_alternative == 0 ? "mul\t%0,%1,%2" : "muli\t%0,%1,%2";}
  [(set_attr "type" "imul")
   (set_attr "mode" "DI")])
```

mul指令pattern 1



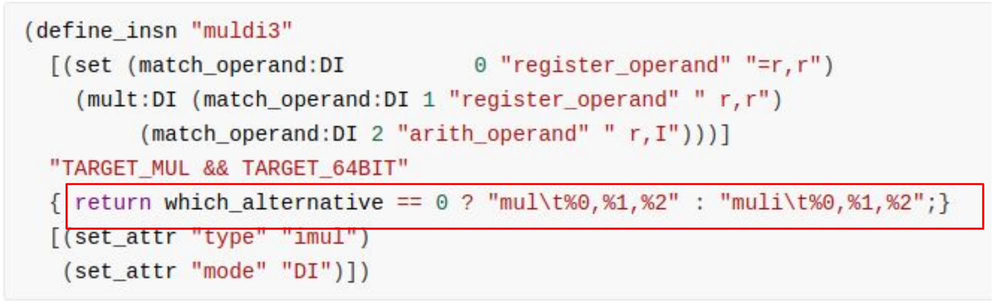
```
(define_insn "muldi3"
  [(set (match_operand:DI           0 "register_operand" "=r,r")
    (mult:DI (match_operand:DI 1 "register_operand" " r,r")
         (match_operand:DI 2 "arith_operand" " r,I")))]
  "TARGET_MUL && TARGET_64BIT"
  "@
   mul\t%0,%1,%2
   muli\t%0,%1,%2"
  [(set_attr "type" "imul")
   (set_attr "mode" "DI")])
```

mul指令pattern 2

GCC指令添加: 添加一个SPN的指令

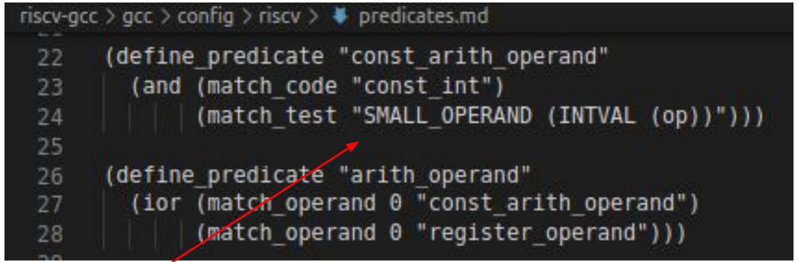
2. 实现指令模板对指令生成的选择

通过自定义arith_operand实现”12bits立即数或寄存器”的predicate



```
(define_insn "muldi3"
  [(set (match_operand:DI          0 "register_operand" "=r,r")
    (mult:DI (match_operand:DI 1 "register_operand" " r,r")
         (match_operand:DI 2 "arith_operand" " r,I")))]
  "TARGET_MUL && TARGET_64BIT"
  { return which_alternative == 0 ? "mul\t%0,%1,%2" : "muli\t%0,%1,%2";}
  [(set_attr "type" "imul")
   (set_attr "mode" "DI")])
```

mul指令pattern 1



riscv-gcc > gcc > config > riscv > predicates.md

```
(define_predicate "const_arith_operand"
  (and (match_code "const_int")
       (match_test "SMALL_OPERAND (INTVAL (op))")))

(define_predicate "arith_operand"
  (ior (match_operand 0 "const_arith_operand")
       (match_operand 0 "register_operand")))
```

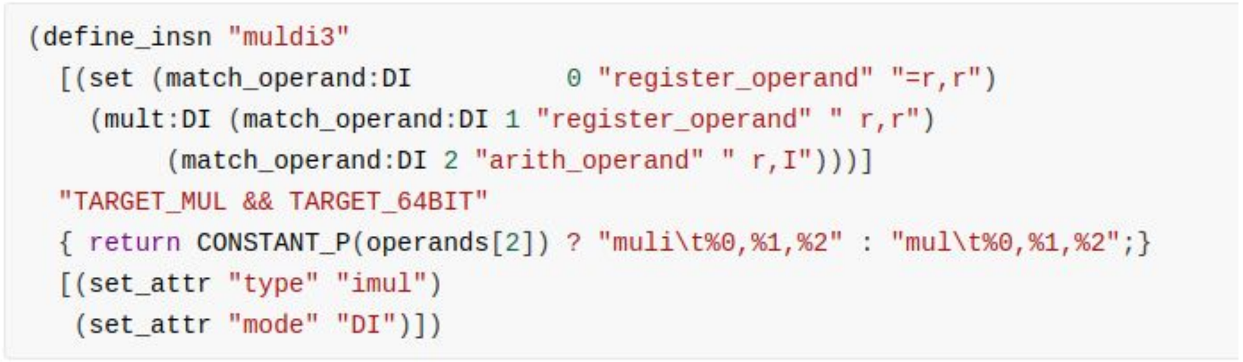
后端自定义predicate

有符号的12位长立即数

GCC指令添加: 添加一个SPN的指令

2. 实现指令模板对指令生成的选择

在output template的C语言形式中，可通过operands[idx]获得对应坐标的operand的rtx，进行更复杂的操作。

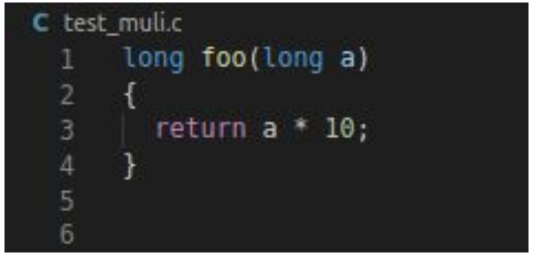


```
(define_insn "muldi3"
  [(set (match_operand:DI          0 "register_operand" "=r,r")
    (mult:DI (match_operand:DI 1 "register_operand" " r,r")
         (match_operand:DI 2 "arith_operand" " r,I")))]
  "TARGET_MUL && TARGET_64BIT"
  { return CONSTANT_P(operands[2]) ? "muli\t%0,%1,%2" : "mul\t%0,%1,%2";}
  [(set_attr "type" "imul")
   (set_attr "mode" "DI")])
```

muldi3 with operation on operand rtx

GCC指令添加: 添加一个SPN的指令

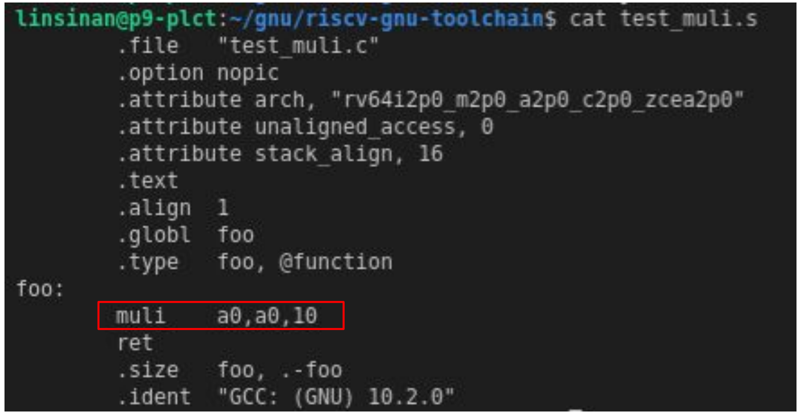
测试一下muli指令



```
test_muli.c
long foo(long a)
{
    return a * 10;
}


```

gcc -Os -S test_muli.c

test_muli.c



```
linsinan@p9-plct:~/gnu/riscv-gnu-toolchain$ cat test_muli.s
        .file   "test_muli.c"
        .option nopic
        .attribute arch, "rv64i2p0_m2p0_a2p0_c2p0_zcea2p0"
        .attribute unaligned_access, 0
        .attribute stack_align, 16
        .text
        .align  1
        .globl  foo
        .type   foo, @function
foo:
        muli    a0,a0,10
        ret
        .size   foo, .-foo
        .ident  "GCC: (GNU) 10.2.0"
```

test_muli.s

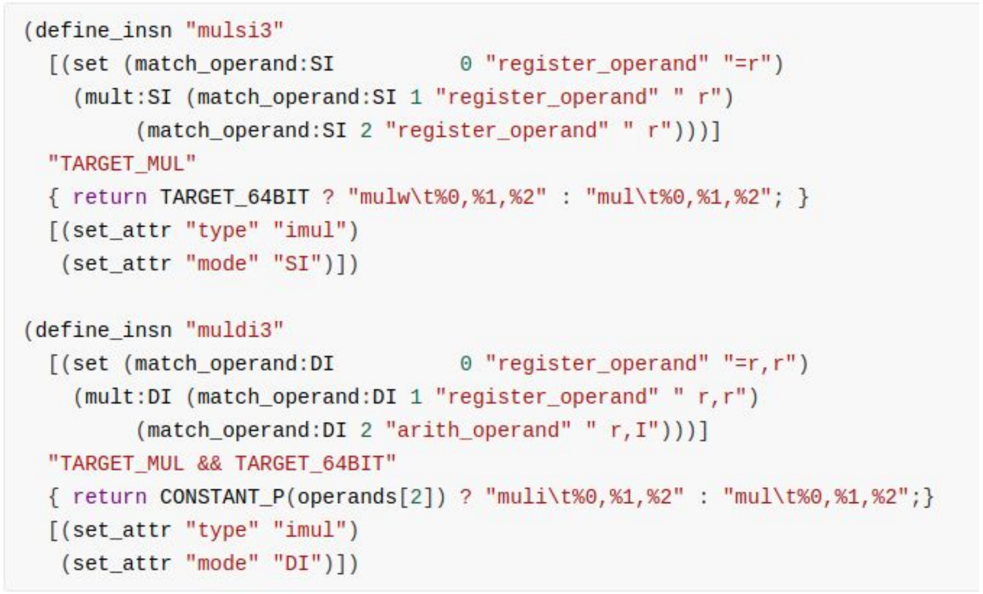
*在riscv中，因为乘法操作设定cost较高，会被优化成加法和位移运算，想自 动生成时需要使用-Os flag

GCC指令添加: 更好的指令模板

1. 使用迭代器对代码进行精简

   机器描述文件中很多指令模板存在共性，使用迭代器能让一个指令模板生成多个实例，来达到精简和省力的目的。

   arch rv64gcv_zpn的addm3标准名就有几十个：
   addsi3
   adddi3
   addv2hi3
   addv4hi3
   addv2si3
   addv4si3
   ....😅



```
(define_insn "mulsi3"
  [(set (match_operand:SI          0 "register_operand" "=r")
    (mult:SI (match_operand:SI 1 "register_operand" " r")
         (match_operand:SI 2 "register_operand" " r")))]
  "TARGET_MUL"
  { return TARGET_64BIT ? "mulw\t%0,%1,%2" : "mul\t%0,%1,%2"; }
  [(set_attr "type" "imul")
   (set_attr "mode" "SI")])

(define_insn "muldi3"
  [(set (match_operand:DI          0 "register_operand" "=r,r")
    (mult:DI (match_operand:DI 1 "register_operand" " r,r")
         (match_operand:DI 2 "arith_operand" " r,I")))]
  "TARGET_MUL && TARGET_64BIT"
  { return CONSTANT_P(operands[2]) ? "muli\t%0,%1,%2" : "mul\t%0,%1,%2";}
  [(set_attr "type" "imul")
   (set_attr "mode" "DI")])
```

mulsi3和muldi3的指令pattern

GCC指令添加: 更好的指令模板

1. 使用迭代器对代码进行精简

使用mode_iterator合并指令模板。

mode迭代器定义

指令名字使用"<mode>", 在构建时候会展开成mulsi3和muldi3

<>中会根据大写或小写得到 对应的mode

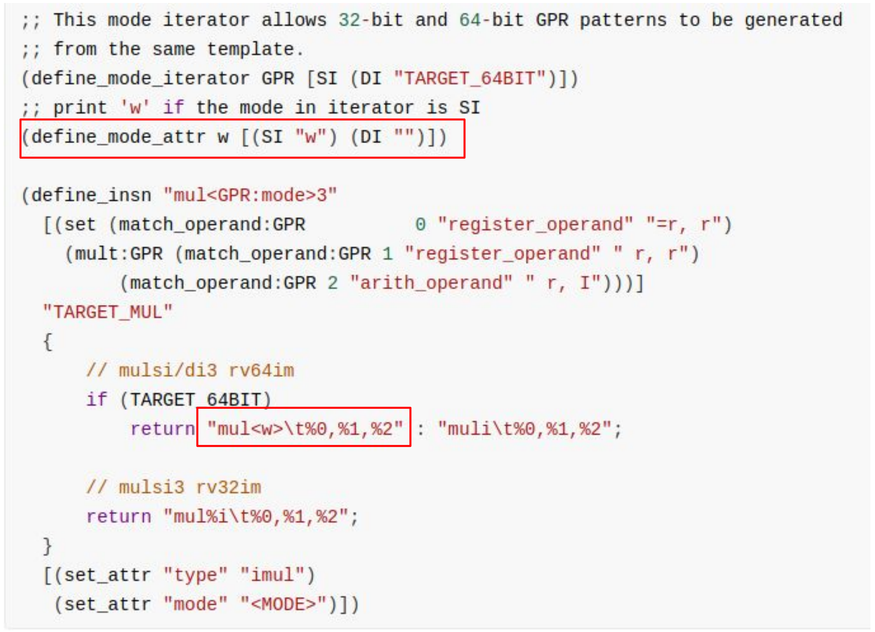
```
;; This mode iterator allows 32-bit and 64-bit GPR patterns to be generated
;; from the same template.
(define_mode_iterator GPR [SI (DI "TARGET_64BIT")])

(define_insn "mul<GPR:mode>3"
  [(set (match_operand:GPR           0 "register_operand" "=r, r")
    (mult:GPR (match_operand:GPR 1 "register_operand" " r, r")
         (match_operand:GPR 2 "arith_operand" " r, I")))]
  "TARGET_MUL"
  {
      // muldi3 rv64im
      if (GET_MODE (operands[2]) == DImode)
          return which_alternative == 0 ? "mul\t%0,%1,%2" : "muli\t%0,%1,%2";

      // mulsi3 rv64im
      if (TARGET_64BIT)
          return which_alternative == 0 ? "mulw\t%0,%1,%2" : "muli\t%0,%1,%2";

      // mulsi3 rv32im
      return which_alternative == 0 ? "mul\t%0,%1,%2" : "muli\t%0,%1,%2";
  }
  [(set_attr "type" "imul")
   (set_attr "mode" "<MODE>")])
```

利用mode迭代器合并的mulsi3和muldi3指令

GCC指令添加: 更好的指令模板

1. 使用迭代器对代码进行精简

使用mode_iterator和mode_attr合并指令模板。

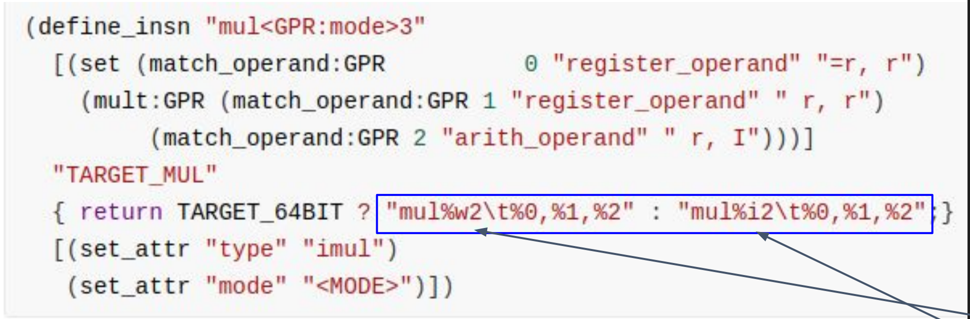


```
;; This mode iterator allows 32-bit and 64-bit GPR patterns to be generated
;; from the same template.
(define_mode_iterator GPR [SI (DI "TARGET_64BIT")])
;; print 'w' if the mode in iterator is SI
(define_mode_attr w [(SI "w") (DI "")])

(define_insn "mul<GPR:mode>3"
  [(set (match_operand:GPR          0 "register_operand" "=r, r")
    (mult:GPR (match_operand:GPR 1 "register_operand" " r, r")
         (match_operand:GPR 2 "arith_operand" " r, I")))]
  "TARGET_MUL"
  {
      // mulsi/di3 rv64im
      if (TARGET_64BIT)
          return "mul<w>\t%0,%1,%2" : "muli\t%0,%1,%2";

      // mulsi3 rv32im
      return "mul%i\t%0,%1,%2";
  }
  [(set_attr "type" "imul")
   (set_attr "mode" "<MODE>")])
```

利用mode_attr迭代器微调输出的汇编指令名称

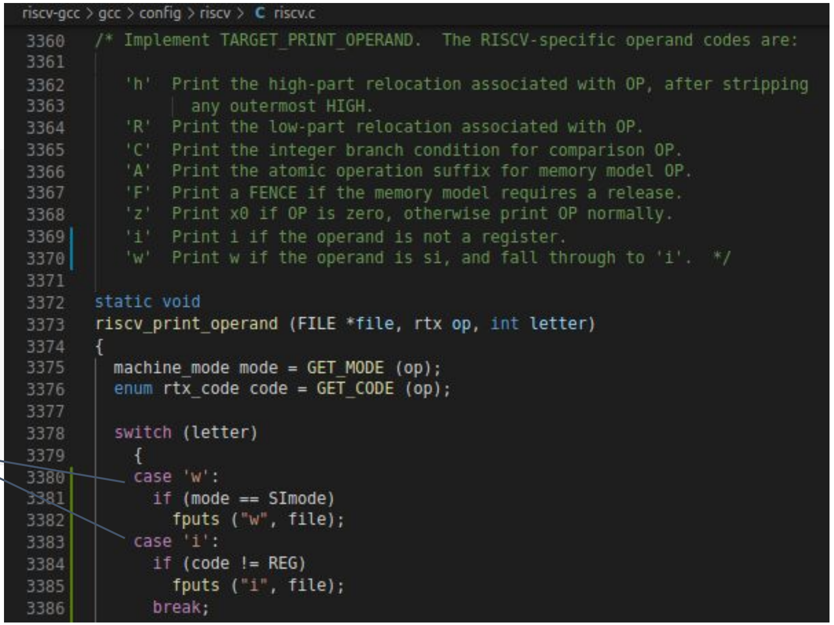
GCC指令添加: 更好的指令模板

2. 使用TARGET_PRINT_OPERAND后端钩子微调输出的汇编指令



```
(define_insn "mul<GPR:mode>3"
  [(set (match_operand:GPR          0 "register_operand" "=r, r")
    (mult:GPR (match_operand:GPR 1 "register_operand" " r, r")
         (match_operand:GPR 2 "arith_operand" " r, I")))]
  "TARGET_MUL"
  { return TARGET_64BIT ? "mul%w2\t%0,%1,%2" : "mul%i2\t%0,%1,%2" }
  [(set_attr "type" "imul")
   (set_attr "mode" "<MODE>")])
```

合并后的mulsi3, muldi3模板



riscv-gcc > gcc > config > riscv > C riscv.c

```
/* Implement TARGET_PRINT_OPERAND.  The RISCV-specific operand codes are:

   'h'  Print the high-part relocation associated with OP, after stripping
          any outermost HIGH.
   'R'  Print the low-part relocation associated with OP.
   'C'  Print the integer branch condition for comparison OP.
   'A'  Print the atomic operation suffix for memory model OP.
   'F'  Print a FENCE if the memory model requires a release.
   'z'  Print x0 if OP is zero, otherwise print OP normally.
   'i'  Print i if the operand is not a register.
   'w'  Print w if the operand is si, and fall through to 'i'.  */

static void
riscv_print_operand (FILE *file, rtx op, int letter)
{
  machine_mode mode = GET_MODE (op);
  enum rtx_code code = GET_CODE (op);

  switch (letter)
    {
    case 'w':
      if (mode == SImode)
        fputs ("w", file);
    case 'i':
      if (code != REG)
        fputs ("i", file);
      break;
```

riscv.c中实现了通过%letter+index对output指令微调的后端钩子

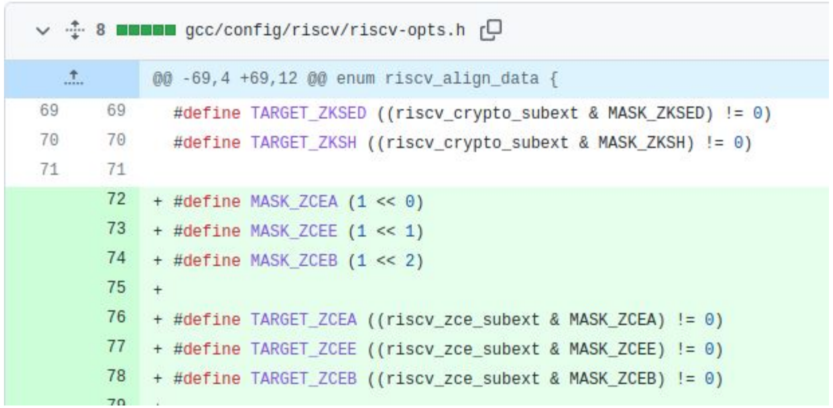
GCC指令添加: define_expand与SPN

有一天，小王正在努力工作着，突然 发现先前的muli指令在spec上被划分到了一个新的扩展，m扩展开启时会生成mul[w]，而新扩展zcea的muli指令只有新扩展的arch否来决定使用与否。

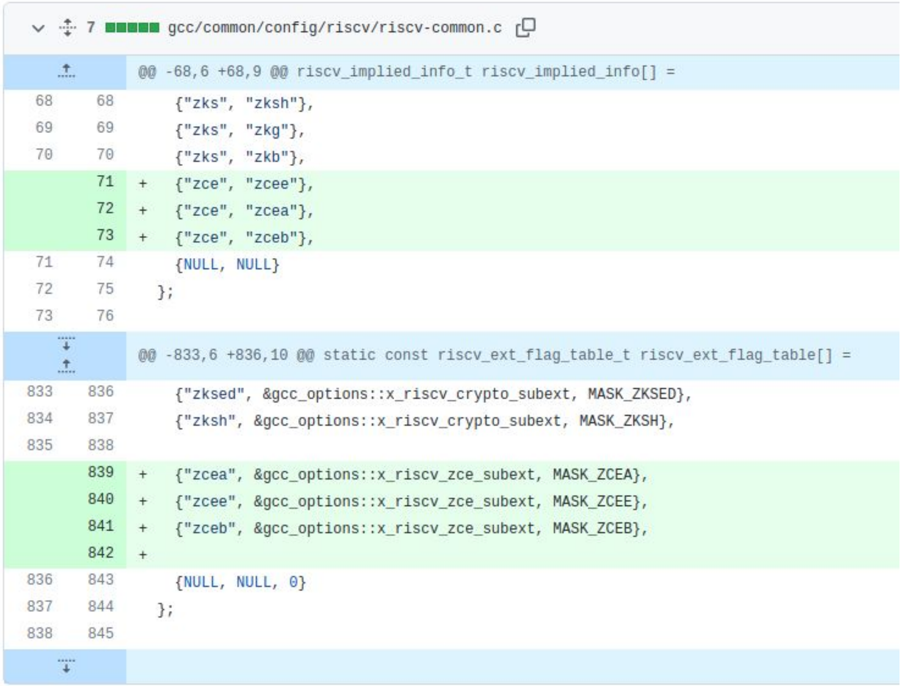
先前定义的迭代器好像不起作用了 ...

GCC指令添加: define_expand与SPN

1. 为riscv新增一个arch



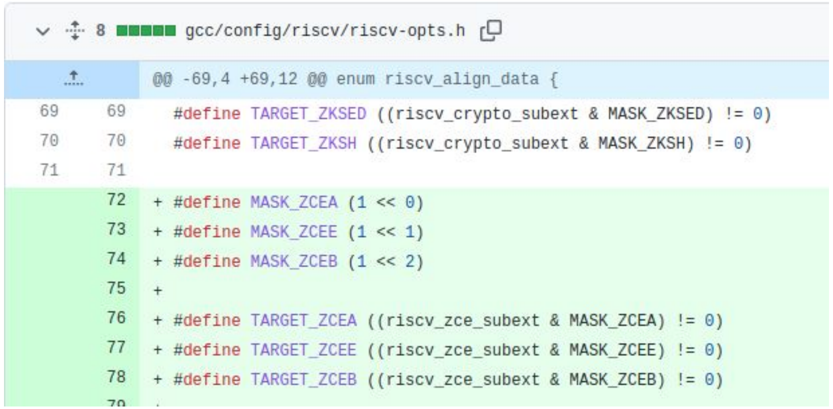
```
8 gcc/config/riscv/riscv-opts.h

@@ -69,4 +69,12 @@ enum riscv_align_data {
  #define TARGET_ZKSED ((riscv_crypto_subext & MASK_ZKSED) != 0)
  #define TARGET_ZKSH ((riscv_crypto_subext & MASK_ZKSH) != 0)

+ #define MASK_ZCEA (1 << 0)
+ #define MASK_ZCEE (1 << 1)
+ #define MASK_ZCEB (1 << 2)
+
+ #define TARGET_ZCEA ((riscv_zce_subext & MASK_ZCEA) != 0)
+ #define TARGET_ZCEE ((riscv_zce_subext & MASK_ZCEE) != 0)
+ #define TARGET_ZCEB ((riscv_zce_subext & MASK_ZCEB) != 0)
```



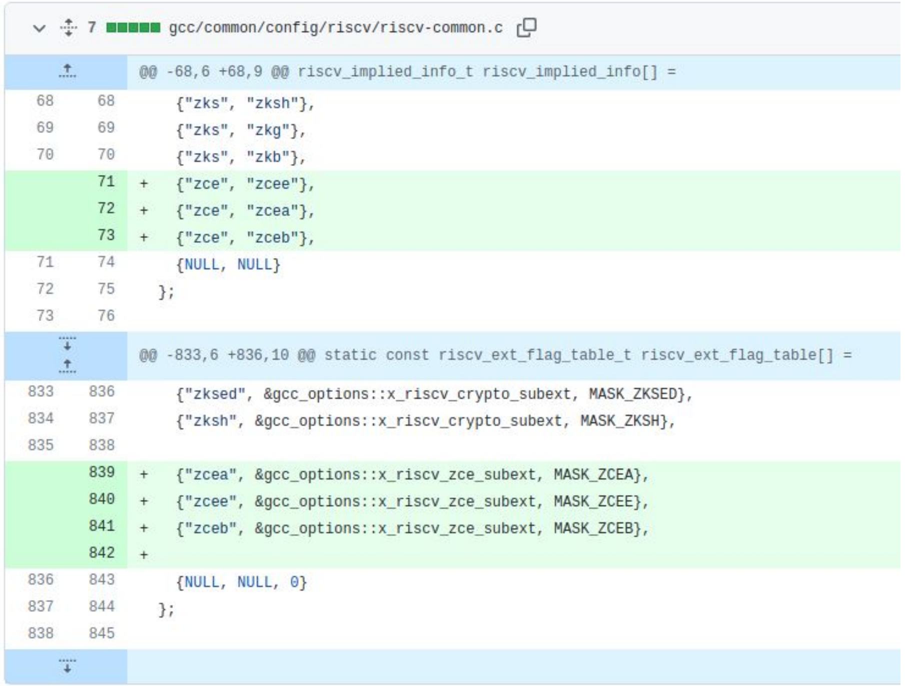
```
7 gcc/common/config/riscv/riscv-common.c

@@ -68,6 +68,9 @@ riscv_implied_info_t riscv_implied_info[] =
    {"zks", "zksh"},
    {"zks", "zkg"},
    {"zks", "zkb"},
+   {"zce", "zcee"},
+   {"zce", "zcea"},
+   {"zce", "zceb"},
    {NULL, NULL}
  };

@@ -833,6 +836,10 @@ static const riscv_ext_flag_table_t riscv_ext_flag_table[] =
    {"zksed", &gcc_options::x_riscv_crypto_subext, MASK_ZKSED},
    {"zksh", &gcc_options::x_riscv_crypto_subext, MASK_ZKSH},

+   {"zcea", &gcc_options::x_riscv_zce_subext, MASK_ZCEA},
+   {"zcee", &gcc_options::x_riscv_zce_subext, MASK_ZCEE},
+   {"zceb", &gcc_options::x_riscv_zce_subext, MASK_ZCEB},
+
    {NULL, NULL, 0}
  };
```

GCC指令添加: define_expand与SPN

1. 为riscv新增一个arch



```
8 gcc/config/riscv/riscv-opts.h
@@ -69,4 +69,12 @@ enum riscv_align_data {
  #define TARGET_ZKSED ((riscv_crypto_subext & MASK_ZKSED) != 0)
  #define TARGET_ZKSH ((riscv_crypto_subext & MASK_ZKSH) != 0)

+ #define MASK_ZCEA (1 << 0)
+ #define MASK_ZCEE (1 << 1)
+ #define MASK_ZCEB (1 << 2)
+
+ #define TARGET_ZCEA ((riscv_zce_subext & MASK_ZCEA) != 0)
+ #define TARGET_ZCEE ((riscv_zce_subext & MASK_ZCEE) != 0)
+ #define TARGET_ZCEB ((riscv_zce_subext & MASK_ZCEB) != 0)
```



```
7 gcc/common/config/riscv/riscv-common.c
@@ -68,6 +68,9 @@ riscv_implied_info_t riscv_implied_info[] =
    {"zks", "zksh"},
    {"zks", "zkg"},
    {"zks", "zkb"},
+   {"zce", "zcee"},
+   {"zce", "zcea"},
+   {"zce", "zceb"},
    {NULL, NULL}
  };

@@ -833,6 +836,10 @@ static const riscv_ext_flag_table_t riscv_ext_flag_table[] =
    {"zksed", &gcc_options::x_riscv_crypto_subext, MASK_ZKSED},
    {"zksh", &gcc_options::x_riscv_crypto_subext, MASK_ZKSH},

+   {"zcea", &gcc_options::x_riscv_zce_subext, MASK_ZCEA},
+   {"zcee", &gcc_options::x_riscv_zce_subext, MASK_ZCEE},
+   {"zceb", &gcc_options::x_riscv_zce_subext, MASK_ZCEB},
+
    {NULL, NULL, 0}
  };

```

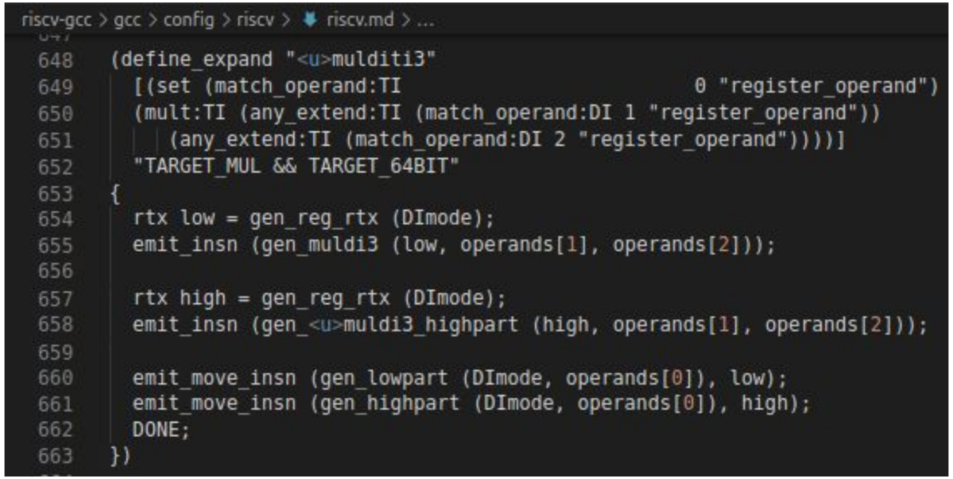
```
./configure --prefix="$PWD/obj-tutorial" --with-arch=rv64imac_zcea --with-abi=lp64
```

$\longrightarrow$ TARGET_ZCEA

GCC指令添加: define_expand与SPN

2. 通过define_expand的使SPN匹配多个指令模板

review: define_expand的使用



riscv-gcc > gcc > config > riscv > riscv.md > ...

```
(define_expand "<u>mulditi3"
  [(set (match_operand:TI                        0 "register_operand")
  (mult:TI (any_extend:TI (match_operand:DI 1 "register_operand"))
     (any_extend:TI (match_operand:DI 2 "register_operand"))))]
  "TARGET_MUL && TARGET_64BIT"
{
  rtx low = gen_reg_rtx (DImode);
  emit_insn (gen_muldi3 (low, operands[1], operands[2]));

  rtx high = gen_reg_rtx (DImode);
  emit_insn (gen_<u>muldi3_highpart (high, operands[1], operands[2]));

  emit_move_insn (gen_lowpart (DImode, operands[0]), low);
  emit_move_insn (gen_highpart (DImode, operands[0]), high);
  DONE;
})
```

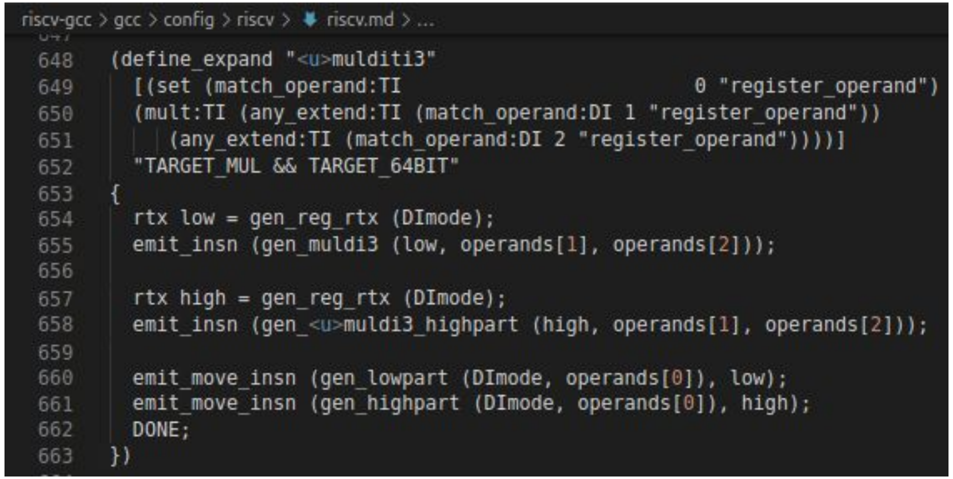
riscv.md中的mulditi3指令模板

1. 使用define_expand跳过RTL模板的生成。(define_insn如果被使用，其rtl模板会被插入到insn链表中)

GCC指令添加: define_expand与SPN

2. 通过define_expand的使SPN匹配多个指令模板

review: define_expand的使用



```
(define_expand "<u>mulditi3"
  [(set (match_operand:TI                    0 "register_operand")
  (mult:TI (any_extend:TI (match_operand:DI 1 "register_operand"))
      (any_extend:TI (match_operand:DI 2 "register_operand"))))]
  "TARGET_MUL && TARGET_64BIT"
{
  rtx low = gen_reg_rtx (DImode);
  emit_insn (gen_muldi3 (low, operands[1], operands[2]));

  rtx high = gen_reg_rtx (DImode);
  emit_insn (gen_<u>muldi3_highpart (high, operands[1], operands[2]));

  emit_move_insn (gen_lowpart (DImode, operands[0]), low);
  emit_move_insn (gen_highpart (DImode, operands[0]), high);
  DONE;
})
```

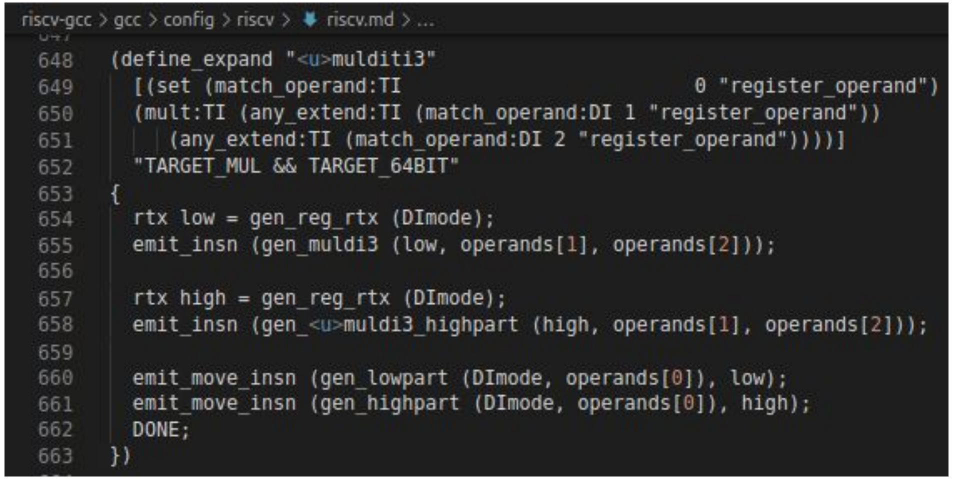
riscv.md中的mulditi3指令模板

1. 使用define_expand跳过RTL模板的生成

GCC指令添加: define_expand与SPN

2. 通过define_expand的使SPN匹配多个指令模板

review: define_expand的使用



riscv-gcc > gcc > config > riscv > riscv.md > ...

```
(define_expand "<u>mulditi3"
  [(set (match_operand:TI                    0 "register_operand")
  (mult:TI (any_extend:TI (match_operand:DI 1 "register_operand"))
      (any_extend:TI (match_operand:DI 2 "register_operand"))))]
  "TARGET_MUL && TARGET_64BIT"
{
  rtx low = gen_reg_rtx (DImode);
  emit_insn (gen_muldi3 (low, operands[1], operands[2]));

  rtx high = gen_reg_rtx (DImode);
  emit_insn (gen_<u>muldi3_highpart (high, operands[1], operands[2]));

  emit_move_insn (gen_lowpart (DImode, operands[0]), low);
  emit_move_insn (gen_highpart (DImode, operands[0]), high);
  DONE;
})
```

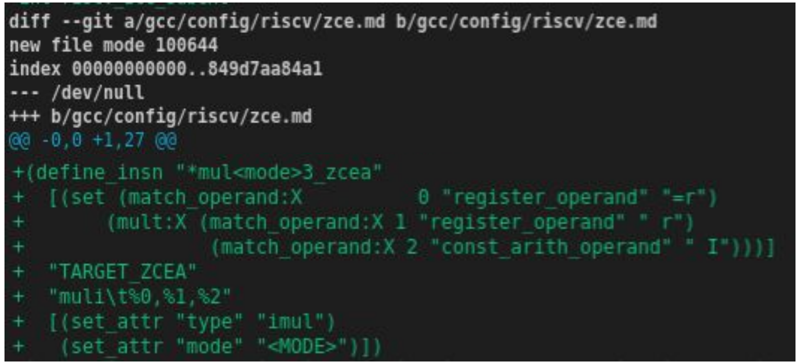
riscv.md中的mulditi3指令模板

1. 使用define_expand跳过RTL模板的生成
2. 使用define_expand让一个SPN对多个RTL模板进行匹配

GCC指令添加: define_expand与SPN

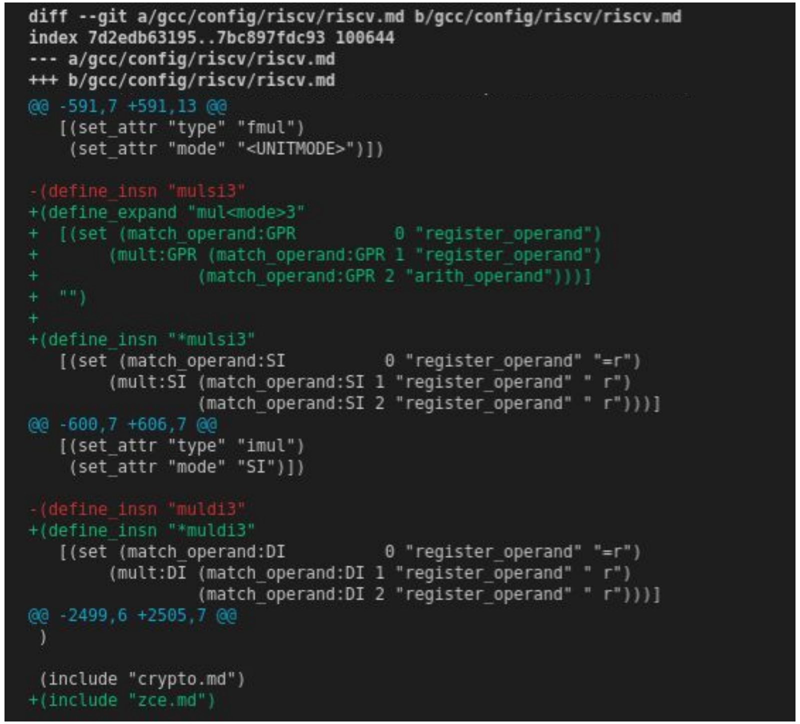
2. 通过define_expand的使SPN匹配多个指令模板

*开头的模板名 -> nameless pattern
.md文件可使用include包含其他.md文件



```
diff --git a/gcc/config/riscv/zce.md b/gcc/config/riscv/zce.md
new file mode 100644
index 00000000000..849d7aa84a1
--- /dev/null
+++ b/gcc/config/riscv/zce.md
@@ -0,0 +1,27 @@
+(define_insn "*mul<mode>3_zcea"
+  [(set (match_operand:X            0 "register_operand" "=r")
+        (mult:X (match_operand:X 1 "register_operand" " r")
+                  (match_operand:X 2 "const_arith_operand" " I")))]
+  "TARGET_ZCEA"
+  "muli\t%0,%1,%2"
+  [(set_attr "type" "imul")
+   (set_attr "mode" "<MODE>")])
```

使用define_expand，并给其他mul<mode>3指令模板名前加*



```
diff --git a/gcc/config/riscv/riscv.md b/gcc/config/riscv/riscv.md
index 7d2edb63195..7bc897fdc93 100644
--- a/gcc/config/riscv/riscv.md
+++ b/gcc/config/riscv/riscv.md
@@ -591,7 +591,13 @@
   [(set_attr "type" "fmul")
    (set_attr "mode" "<UNITMODE>")])

-(define_insn "mulsi3"
+(define_expand "mul<mode>3"
+  [(set (match_operand:GPR          0 "register_operand")
+        (mult:GPR (match_operand:GPR 1 "register_operand")
+                  (match_operand:GPR 2 "arith_operand")))]
+  "")
+
+(define_insn "*mulsi3"
   [(set (match_operand:SI          0 "register_operand" "=r")
         (mult:SI (match_operand:SI 1 "register_operand" " r")
                  (match_operand:SI 2 "register_operand" " r")))]
@@ -600,7 +606,7 @@
   [(set_attr "type" "imul")
    (set_attr "mode" "SI")])

-(define_insn "muldi3"
+(define_insn "*muldi3"
   [(set (match_operand:DI          0 "register_operand" "=r")
         (mult:DI (match_operand:DI 1 "register_operand" " r")
                  (match_operand:DI 2 "register_operand" " r")))]
@@ -2499,6 +2505,7 @@
 )

 (include "crypto.md")
+(include "zce.md")
```

使用define_expand，并给其他mul<mode>3指令模板名前加*

GCC指令添加: 自定义predicate和constraints

突然spec又变了，muli指令被拆分成了好几个😅:

muli6:   如果imm长度小于5bits
muli12: 如果imm长度在5bits到12bits之间
mulipow: 如果imm是1,2,4或者8

我们又该如何如何处理呢？

GCC指令添加: 自定义predicate和constraints

突然spec又变了，muli指令被拆分成了好几个😅:

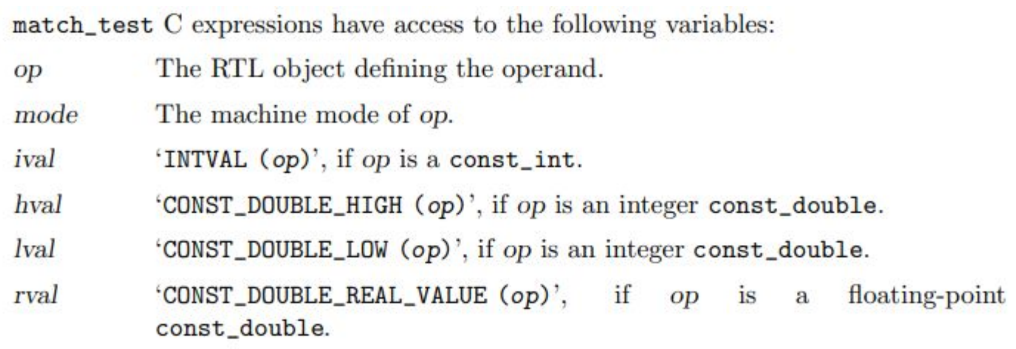
muli5:   如果imm长度小于5bits
muli12:   如果imm长度在5bits到12bits之间
mulipow: 如果imm是1,2,4或者8

我们又该如何如何处理呢？
方法之一：添加predicate和constraints

GCC指令添加: 自定义predicate和constraints

review: predicate和constraint的区别:

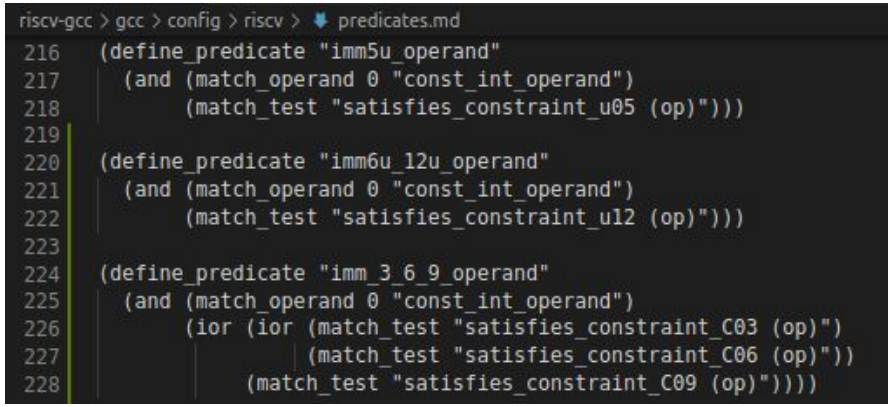
- constraint的匹配发生在匹配MD-RTL之后(predicate为True)
- constraint多用于对汇编指令输出格式的微调
- define_predicate得到函数可以在riscv.c中对rtx进行判断是否满足某种条件



`match_test` C expressions have access to the following variables:

op	The RTL object defining the operand.
mode	The machine mode of *op*.
ival	'`INTVAL (`*op*`)`', if *op* is a `const_int`.
hval	'`CONST_DOUBLE_HIGH (`*op*`)`', if *op* is an integer `const_double`.
lval	'`CONST_DOUBLE_LOW (`*op*`)`', if *op* is an integer `const_double`.
rval	'`CONST_DOUBLE_REAL_VALUE (`*op*`)`', if *op* is a floating-point `const_double`.

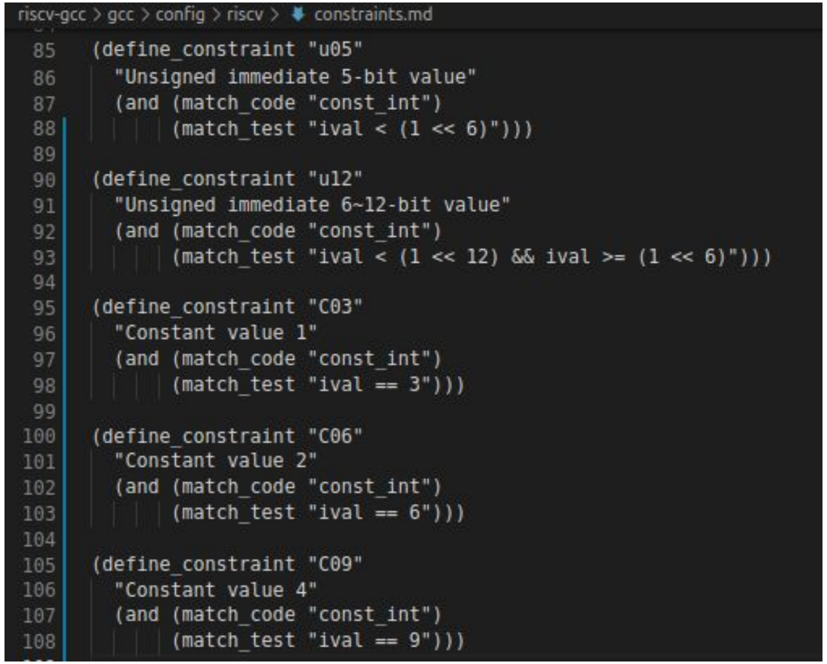
GCC指令添加: 自定义predicate和constraints


riscv-gcc > gcc > config > riscv > predicates.md

```
(define_predicate "imm5u_operand"
  (and (match_operand 0 "const_int_operand")
       (match_test "satisfies_constraint_u05 (op)")))

(define_predicate "imm6u_12u_operand"
  (and (match_operand 0 "const_int_operand")
       (match_test "satisfies_constraint_u12 (op)")))

(define_predicate "imm_3_6_9_operand"
  (and (match_operand 0 "const_int_operand")
       (ior (ior (match_test "satisfies_constraint_C03 (op)")
                 (match_test "satisfies_constraint_C06 (op)"))
            (match_test "satisfies_constraint_C09 (op)"))))
```

define_predicate中可以直接使用定义好的constraints,
或者直接对operand的rtx进行操作,
e.g. INTVAL (op) < (1 << 6) …

使用ior/and进行逻辑操作

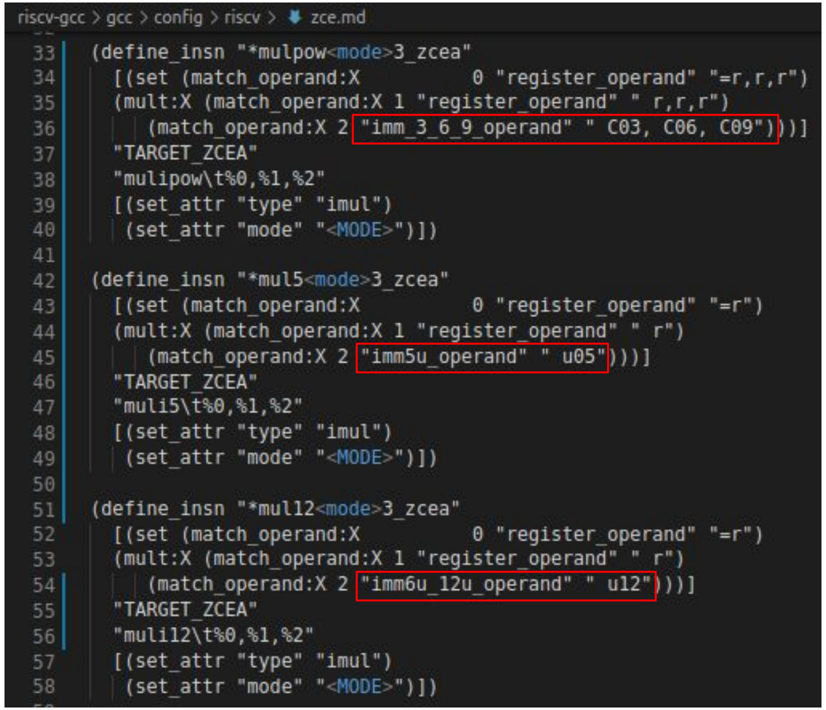

riscv-gcc > gcc > config > riscv > constraints.md

```
(define_constraint "u05"
  "Unsigned immediate 5-bit value"
  (and (match_code "const_int")
       (match_test "ival < (1 << 6)")))

(define_constraint "u12"
  "Unsigned immediate 6~12-bit value"
  (and (match_code "const_int")
       (match_test "ival < (1 << 12) && ival >= (1 << 6)")))

(define_constraint "C03"
  "Constant value 1"
  (and (match_code "const_int")
       (match_test "ival == 3")))

(define_constraint "C06"
  "Constant value 2"
  (and (match_code "const_int")
       (match_test "ival == 6")))

(define_constraint "C09"
  "Constant value 4"
  (and (match_code "const_int")
       (match_test "ival == 9")))
```

利用match_test根据条件实现constraint

*constraint的命名要遵循一定 规则: 同开头字母的名字 长度一样; 名字不能是其他 constraint的prefix等 ....

GCC指令添加: 自定义predicate和constraints



riscv-gcc > gcc > config > riscv > zce.md

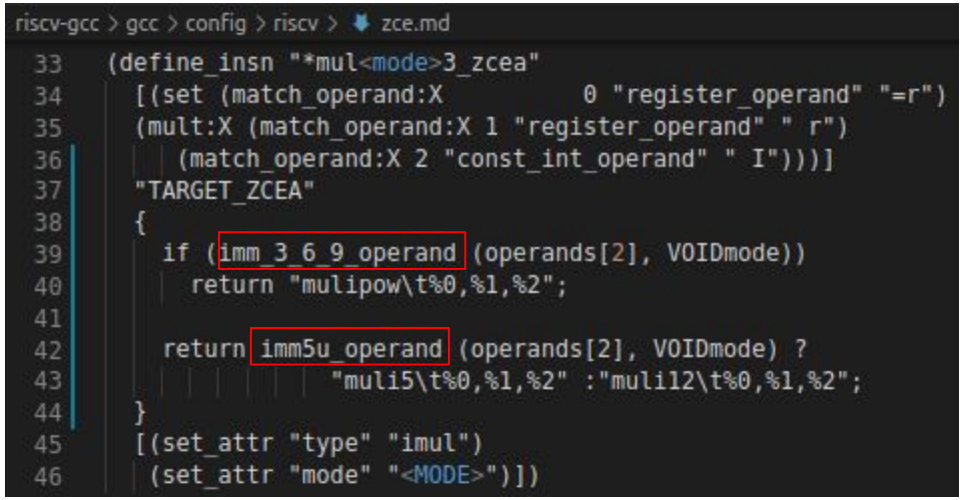
```
(define_insn "*mulpow<mode>3_zcea"
  [(set (match_operand:X          0 "register_operand" "=r,r,r")
  (mult:X (match_operand:X 1 "register_operand" " r,r,r")
     (match_operand:X 2 "imm_3_6_9_operand" " C03, C06, C09")))]
  "TARGET_ZCEA"
  "mulipow\t%0,%1,%2"
  [(set_attr "type" "imul")
   (set_attr "mode" "<MODE>")])

(define_insn "*mul5<mode>3_zcea"
  [(set (match_operand:X          0 "register_operand" "=r")
  (mult:X (match_operand:X 1 "register_operand" " r")
     (match_operand:X 2 "imm5u_operand" " u05")))]
  "TARGET_ZCEA"
  "muli5\t%0,%1,%2"
  [(set_attr "type" "imul")
   (set_attr "mode" "<MODE>")])

(define_insn "*mul12<mode>3_zcea"
  [(set (match_operand:X          0 "register_operand" "=r")
  (mult:X (match_operand:X 1 "register_operand" " r")
     (match_operand:X 2 "imm6u_12u_operand" " u12")))]
  "TARGET_ZCEA"
  "muli12\t%0,%1,%2"
  [(set_attr "type" "imul")
   (set_attr "mode" "<MODE>")])
```

三种muli的指令模板

*order masters.

GCC指令添加: 自定义predicate和constraints

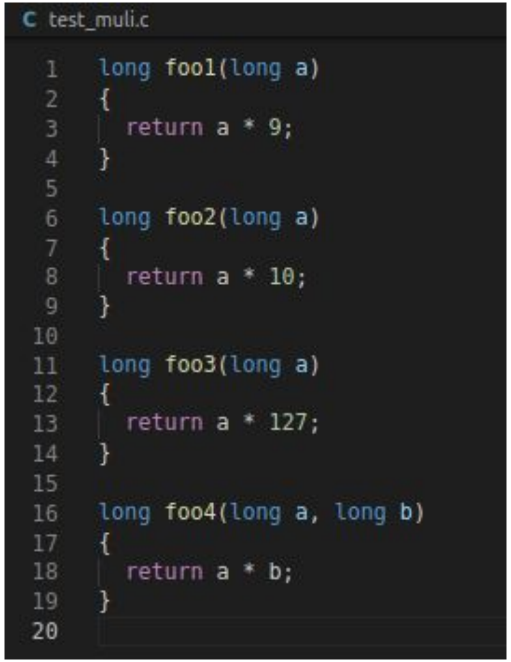
define_predicates的定义会被展开成函数，可以在 riscv.c或.md文件中使用。



riscv-gcc > gcc > config > riscv > zce.md

```
(define_insn "*mul<mode>3_zcea"
  [(set (match_operand:X          0 "register_operand" "=r")
  (mult:X (match_operand:X 1 "register_operand" " r")
     (match_operand:X 2 "const_int_operand" " I")))]
  "TARGET_ZCEA"
  {
    if (imm_3_6_9_operand (operands[2], VOIDmode))
      return "mulipow\t%0,%1,%2";

    return imm5u_operand (operands[2], VOIDmode) ?
                "muli5\t%0,%1,%2" :"muli12\t%0,%1,%2";
  }
  [(set_attr "type" "imul")
   (set_attr "mode" "<MODE>")])
```

GCC指令添加: 自定义predicate和constraints

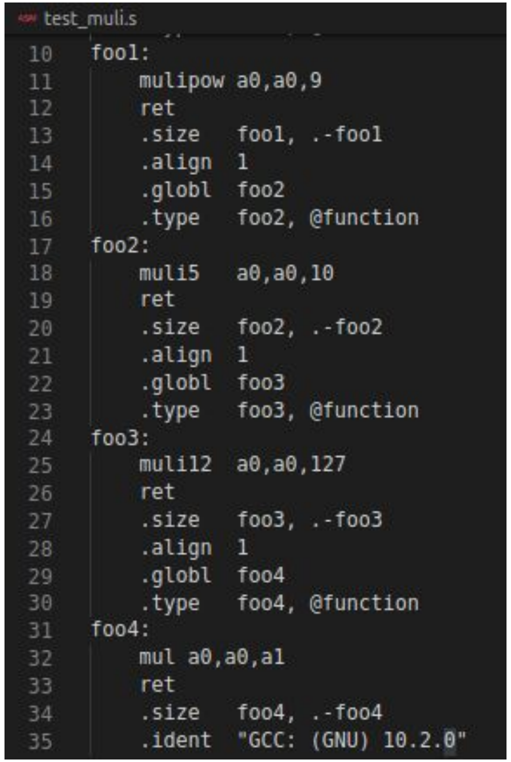


test_muli.c

```c
long foo1(long a)
{
    return a * 9;
}

long foo2(long a)
{
    return a * 10;
}

long foo3(long a)
{
    return a * 127;
}

long foo4(long a, long b)
{
    return a * b;
}

```

test_muli.c

gcc -Os -S test_muli.c

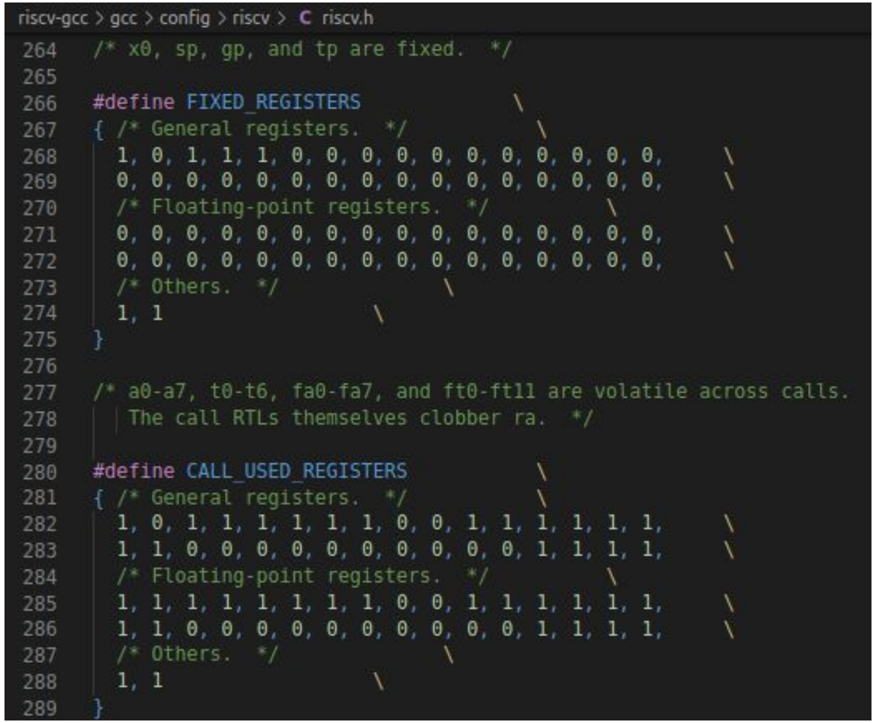


test_muli.s

```asm
foo1:
        mulipow a0,a0,9
        ret
        .size   foo1, .-foo1
        .align  1
        .globl  foo2
        .type   foo2, @function
foo2:
        muli5   a0,a0,10
        ret
        .size   foo2, .-foo2
        .align  1
        .globl  foo3
        .type   foo3, @function
foo3:
        muli12  a0,a0,127
        ret
        .size   foo3, .-foo3
        .align  1
        .globl  foo4
        .type   foo4, @function
foo4:
        mul a0,a0,a1
        ret
        .size   foo4, .-foo4
        .ident  "GCC: (GNU) 10.2.0"
```

test_muli.s

GCC指令添加：寄存器限制


riscv-gcc > gcc > config > riscv > C riscv.h

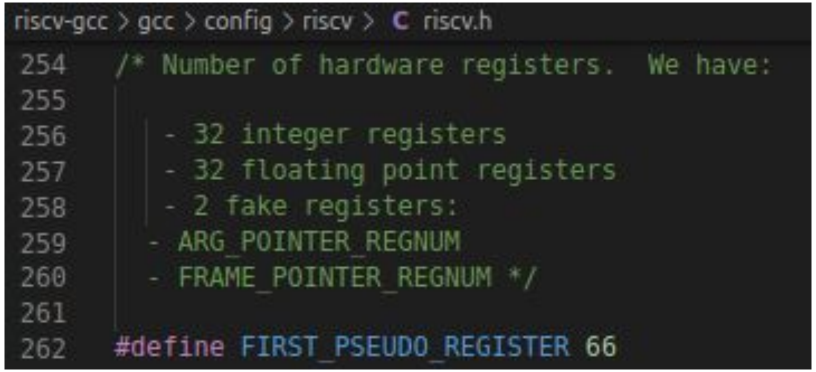
```
/* x0, sp, gp, and tp are fixed.  */

#define FIXED_REGISTERS                         \
{ /* General registers.  */                     \
  1, 0, 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                                  \
}

/* 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, 0, 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                                  \
}
```

riscv.h

FIXED_REGISTERS: 有特殊目的的寄存器，不参与到寄存器分配, e.g. gp.

CALL_USED_REGISTERS: 该宏标识寄存器 不可用于分配跨函数调用的值。

GCC指令添加：寄存器限制

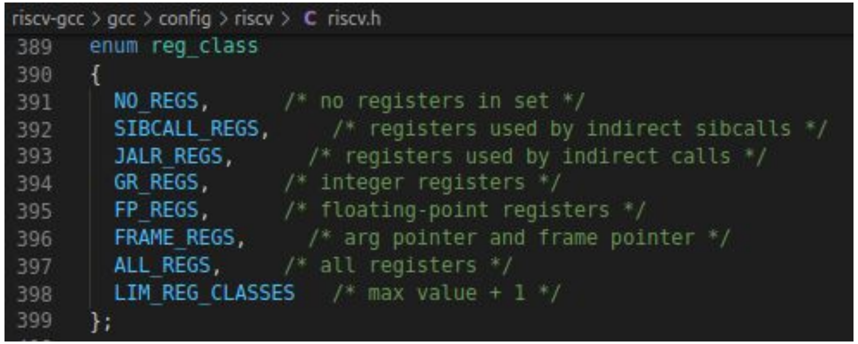

riscv-gcc > gcc > config > riscv > riscv.h

```
/* Number of hardware registers.  We have:

   - 32 integer registers
   - 32 floating point registers
   - 2 fake registers:
  - ARG_POINTER_REGNUM
  - FRAME_POINTER_REGNUM */

#define FIRST_PSEUDO_REGISTER 66
```

riscv.h

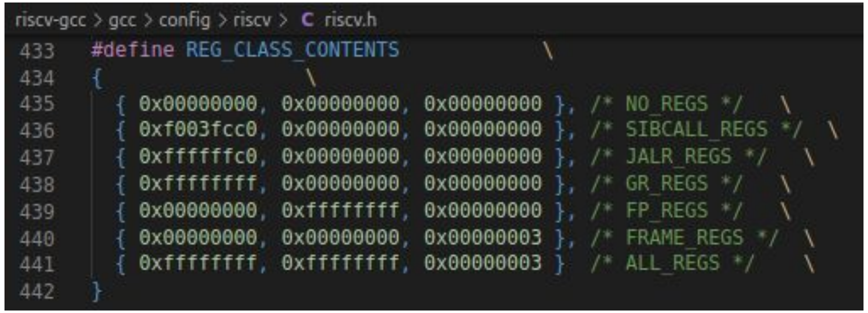
FIRST_PSEUDO_REGISTER:
编译器可知的寄存器数

GCC指令添加：寄存器限制


riscv-gcc > gcc > config > riscv > C riscv.h

```
enum reg_class
{
  NO_REGS,		/* no registers in set */
  SIBCALL_REGS,		/* registers used by indirect sibcalls */
  JALR_REGS,		/* registers used by indirect calls */
  GR_REGS,		/* integer registers */
  FP_REGS,		/* floating-point registers */
  FRAME_REGS,		/* arg pointer and frame pointer */
  ALL_REGS,		/* all registers */
  LIM_REG_CLASSES	/* max value + 1 */
};
```

enum reg_class: 寄存器类型

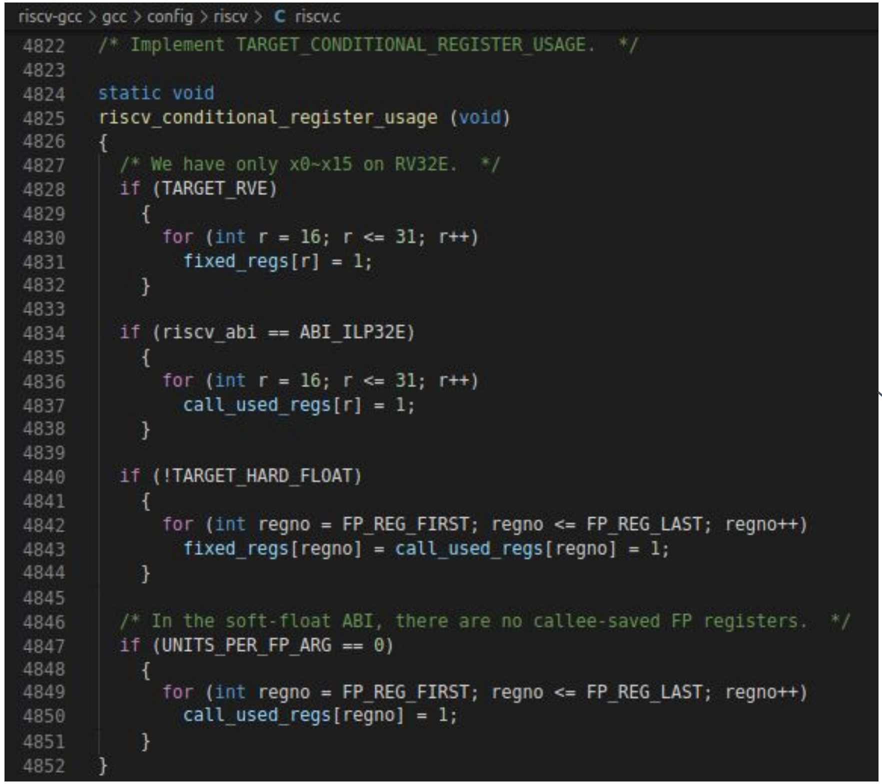

riscv-gcc > gcc > config > riscv > C riscv.h

```
#define REG_CLASS_CONTENTS						\
{									\
  { 0x00000000, 0x00000000, 0x00000000 },	/* NO_REGS */		\
  { 0xf003fcc0, 0x00000000, 0x00000000 },	/* SIBCALL_REGS */	\
  { 0xffffffc0, 0x00000000, 0x00000000 },	/* JALR_REGS */		\
  { 0xffffffff, 0x00000000, 0x00000000 },	/* GR_REGS */		\
  { 0x00000000, 0xffffffff, 0x00000000 },	/* FP_REGS */		\
  { 0x00000000, 0x00000000, 0x00000003 },	/* FRAME_REGS */	\
  { 0xffffffff, 0xffffffff, 0x00000003 }	/* ALL_REGS */		\
}
```

riscv.h

REG_CLASS_CONTENTS: 寄存器类型掩码

GCC指令添加：寄存器限制


riscv-gcc > gcc > config > riscv > riscv.c

```c
/* Implement TARGET_CONDITIONAL_REGISTER_USAGE.  */

static void
riscv_conditional_register_usage (void)
{
  /* We have only x0~x15 on RV32E.  */
  if (TARGET_RVE)
    {
      for (int r = 16; r <= 31; r++)
        fixed_regs[r] = 1;
    }

  if (riscv_abi == ABI_ILP32E)
    {
      for (int r = 16; r <= 31; r++)
        call_used_regs[r] = 1;
    }

  if (!TARGET_HARD_FLOAT)
    {
      for (int regno = FP_REG_FIRST; regno <= FP_REG_LAST; regno++)
        fixed_regs[regno] = call_used_regs[regno] = 1;
    }

  /* In the soft-float ABI, there are no callee-saved FP registers.  */
  if (UNITS_PER_FP_ARG == 0)
    {
      for (int regno = FP_REG_FIRST; regno <= FP_REG_LAST; regno++)
        call_used_regs[regno] = 1;
    }
}
```

riscv.c中对TARGET_CONDITIONAL_REGISTER_USAGE的实现

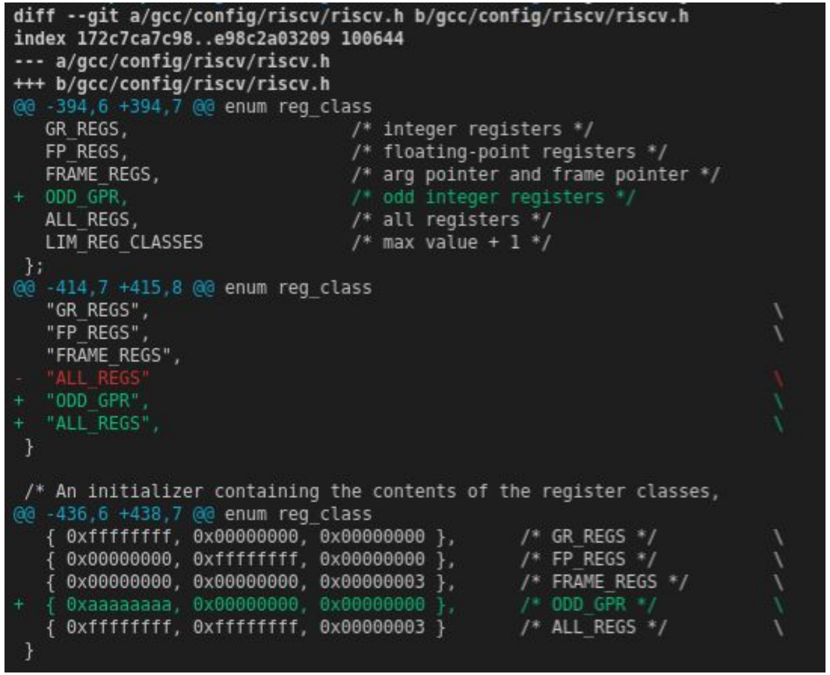
GCC还提供了一系列的后端钩子，来对寄存器继续限制。

根据arch或abi改变寄存器的用途

GCC指令添加：寄存器限制

spec进行了新一轮的更新，DI版本的muli指令可以在rv32imac_zcea下使用，但是DImode的变量必须存放在基数寄存器中。

GCC指令添加：寄存器限制

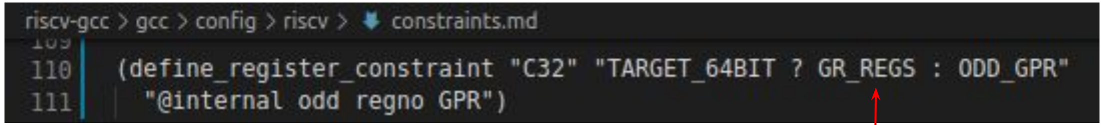


```
diff --git a/gcc/config/riscv/riscv.h b/gcc/config/riscv/riscv.h
index 172c7ca7c98..e98c2a03209 100644
--- a/gcc/config/riscv/riscv.h
+++ b/gcc/config/riscv/riscv.h
@@ -394,6 +394,7 @@ enum reg_class
   GR_REGS,                     /* integer registers */
   FP_REGS,                     /* floating-point registers */
   FRAME_REGS,                  /* arg pointer and frame pointer */
+  ODD_GPR,                     /* odd integer registers */
   ALL_REGS,                    /* all registers */
   LIM_REG_CLASSES              /* max value + 1 */
 };
@@ -414,7 +415,8 @@ enum reg_class
   "GR_REGS",                                                          \
   "FP_REGS",                                                          \
   "FRAME_REGS",
-  "ALL_REGS"                                                          \
+  "ODD_GPR",                                                          \
+  "ALL_REGS",                                                         \
 }

 /* An initializer containing the contents of the register classes,
@@ -436,6 +438,7 @@ enum reg_class
   { 0xffffffff, 0x00000000, 0x00000000 },      /* GR_REGS */          \
   { 0x00000000, 0xffffffff, 0x00000000 },      /* FP_REGS */          \
   { 0x00000000, 0x00000000, 0x00000003 },      /* FRAME_REGS */       \
+  { 0xaaaaaaaa, 0x00000000, 0x00000000 },      /* ODD_GPR */          \
   { 0xffffffff, 0xffffffff, 0x00000003 }       /* ALL_REGS */         \
 }
```

riscv.h

1. 在riscv.h中加入一个新的寄存器类型，ODD_GPR

GCC指令添加：寄存器限制


riscv-gcc > gcc > config > riscv > constraints.md

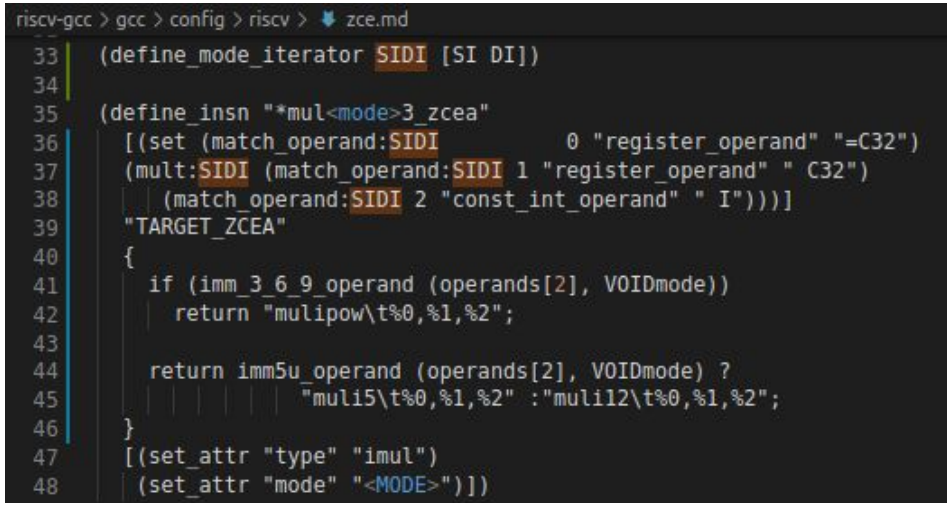
```
(define_register_constraint "C32" "TARGET_64BIT ? GR_REGS : ODD_GPR"
  "@internal odd regno GPR")
```

constraints.md

GR_REGS if TARGET_64BIT

1. 在riscv.h中加入一个新的寄存器类型，ODD_GPR
2. 添加一个新的constraint

GCC指令添加：寄存器限制



riscv-gcc > gcc > config > riscv > zce.md

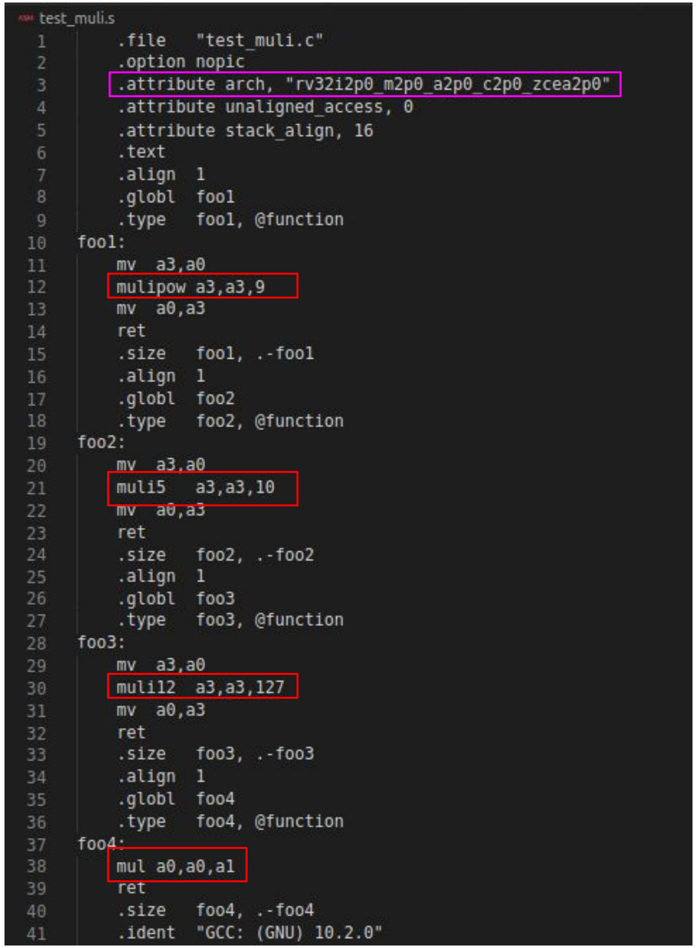
```
(define_mode_iterator SIDI [SI DI])

(define_insn "*mul<mode>3_zcea"
  [(set (match_operand:SIDI          0 "register_operand" "=C32")
  (mult:SIDI (match_operand:SIDI 1 "register_operand" " C32")
     (match_operand:SIDI 2 "const_int_operand" " I")))]
  "TARGET_ZCEA"
  {
    if (imm_3_6_9_operand (operands[2], VOIDmode))
      return "mulipow\t%0,%1,%2";

    return imm5u_operand (operands[2], VOIDmode) ?
                "muli5\t%0,%1,%2" :"muli12\t%0,%1,%2";
  }
  [(set_attr "type" "imul")
   (set_attr "mode" "<MODE>")])
```

muli指令模板

1. 在riscv.h中加入一个新的寄存器类型，ODD_GPR
2. 添加一个新的constraint
3. 修改原先的指令模板，rv32下允许DI，修改operand0和operand1的constraint

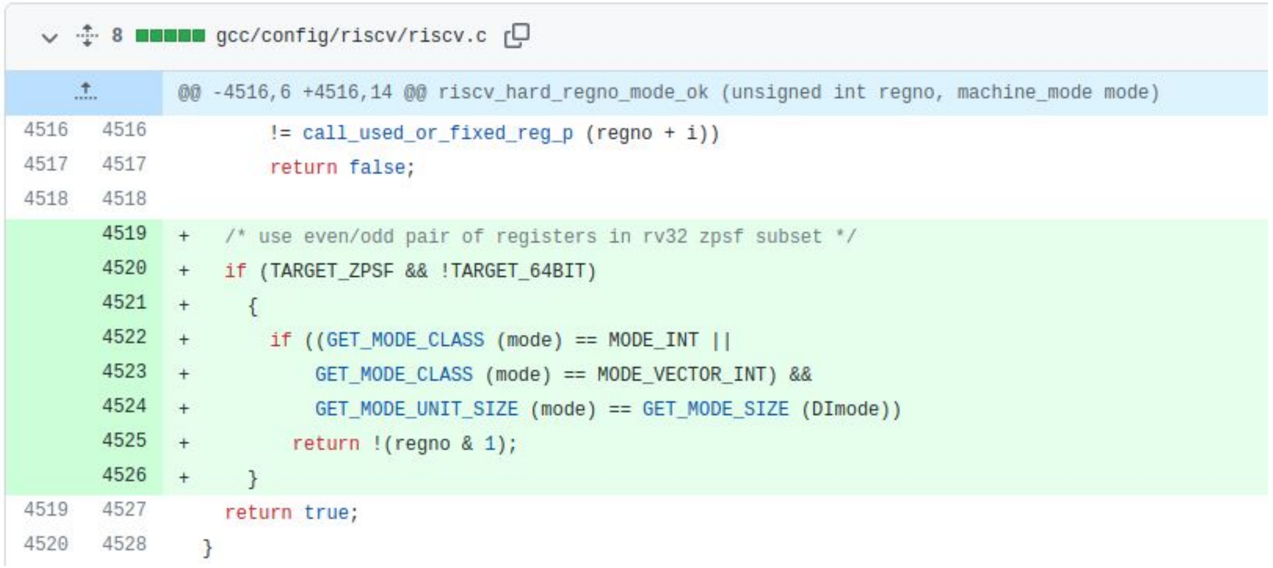
GCC指令添加：寄存器限制

rv32下muli指令的rd和rs1都是使用奇数regno的寄存器了！


test_muli.s

```
	.file	"test_muli.c"
	.option nopic
	.attribute arch, "rv32i2p0_m2p0_a2p0_c2p0_zcea2p0"
	.attribute unaligned_access, 0
	.attribute stack_align, 16
	.text
	.align	1
	.globl	foo1
	.type	foo1, @function
foo1:
	mv	a3,a0
	mulipow a3,a3,9
	mv	a0,a3
	ret
	.size	foo1, .-foo1
	.align	1
	.globl	foo2
	.type	foo2, @function
foo2:
	mv	a3,a0
	muli5	a3,a3,10
	mv	a0,a3
	ret
	.size	foo2, .-foo2
	.align	1
	.globl	foo3
	.type	foo3, @function
foo3:
	mv	a3,a0
	muli12	a3,a3,127
	mv	a0,a3
	ret
	.size	foo3, .-foo3
	.align	1
	.globl	foo4
	.type	foo4, @function
foo4:
	mul	a0,a0,a1
	ret
	.size	foo4, .-foo4
	.ident	"GCC: (GNU) 10.2.0"
```

GCC指令添加：奇偶寄存器对的分配实现



```
8 gcc/config/riscv/riscv.c

@@ -4516,6 +4516,14 @@ riscv_hard_regno_mode_ok (unsigned int regno, machine_mode mode)
         != call_used_or_fixed_reg_p (regno + i))
         return false;

+   /* use even/odd pair of registers in rv32 zpsf subset */
+   if (TARGET_ZPSF && !TARGET_64BIT)
+     {
+       if ((GET_MODE_CLASS (mode) == MODE_INT ||
+           GET_MODE_CLASS (mode) == MODE_VECTOR_INT) &&
+           GET_MODE_UNIT_SIZE (mode) == GET_MODE_SIZE (DImode))
+         return !(regno & 1);
+     }
    return true;
  }
```

TARGET_HARD_REGNO_MODE_OK：返回true, 如果允许在该编号寄存器下能储存machine_mode的值。

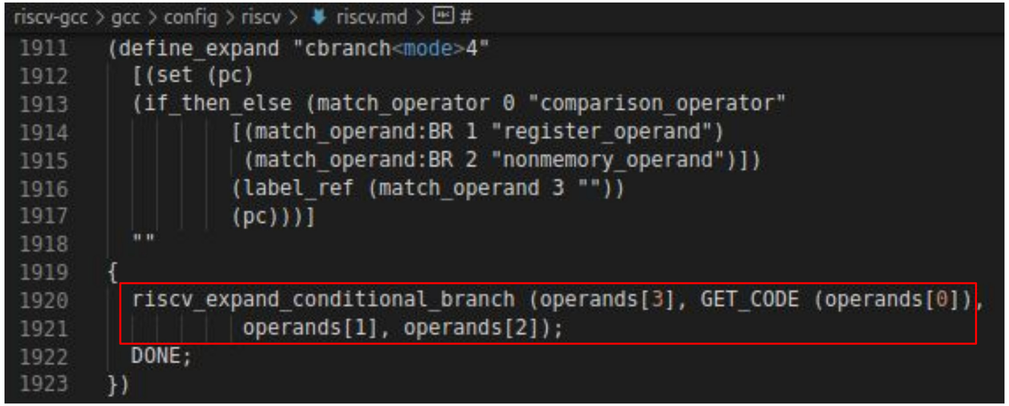
GCC指令添加：riscv-protos.h

riscv.h会在GCC构建的早期被引用，此 时rtl.h和tree.h内的类型都没有定义，所以若要.md中使用的函数，需要在 riscv-protos.h声明（是现在riscv.c）。

tm_p.h $\xrightarrow{\text{include}}$ tm_pred.h, riscv-protos.h

GCC指令添加：条件跳转指令模板的添加

bne rs1 rs2 label
beq rs1 rs2 label



riscv-gcc > gcc > config > riscv > riscv.md > #

```
(define_expand "cbranch<mode>4"
  [(set (pc)
  (if_then_else (match_operator 0 "comparison_operator"
          [(match_operand:BR 1 "register_operand")
           (match_operand:BR 2 "nonmemory_operand")])
          (label_ref (match_operand 3 ""))
          (pc)))]
  ""
{
  riscv_expand_conditional_branch (operands[3], GET_CODE (operands[0]),
                                   operands[1], operands[2]);
  DONE;
})
```

riscv.c中的条件跳转指令模板

GCC指令添加：条件跳转指令模板的添加

```
bne rs1,rs2,label
beq rs1,rs2,label

bnei rs1,imm,label
beqi rs1,imm,label
```

添加一组允许指令使用立即数的条件跳 转指令

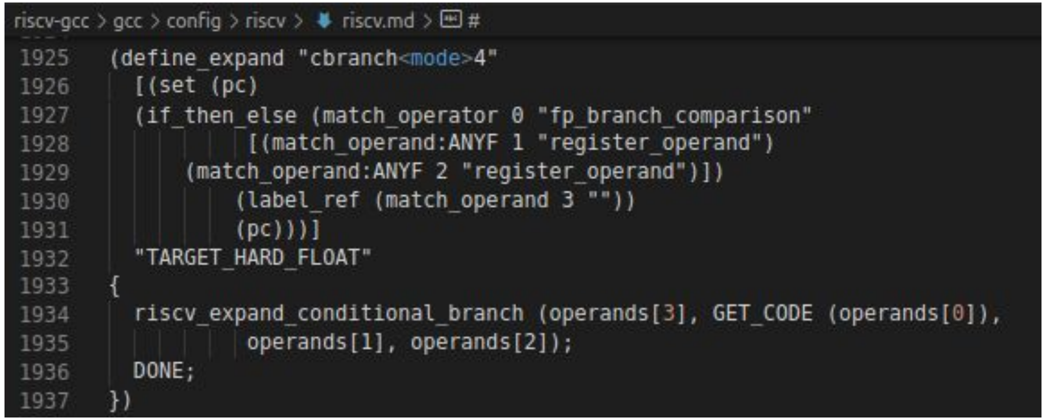
GCC指令添加：条件跳转指令模板的添加

bne rs1,rs2,label
beq rs1,rs2,label

bnei rs1,imm,label
beqi rs1,imm,label

添加一组允许指令使用立即数的条件跳 转指令

1. 查看是否存在SPN，是否riscv.md已有对应的实现



riscv-gcc > gcc > config > riscv > riscv.md > #

```
(define_expand "cbranch<mode>4"
  [(set (pc)
  (if_then_else (match_operator 0 "fp_branch_comparison"
                 [(match_operand:ANYF 1 "register_operand")
             (match_operand:ANYF 2 "register_operand")])
                (label_ref (match_operand 3 ""))
                (pc)))]
  "TARGET_HARD_FLOAT"
{
  riscv_expand_conditional_branch (operands[3], GET_CODE (operands[0]),
                                   operands[1], operands[2]);
  DONE;
})
```

条件跳转spn在riscv.md的指令模板

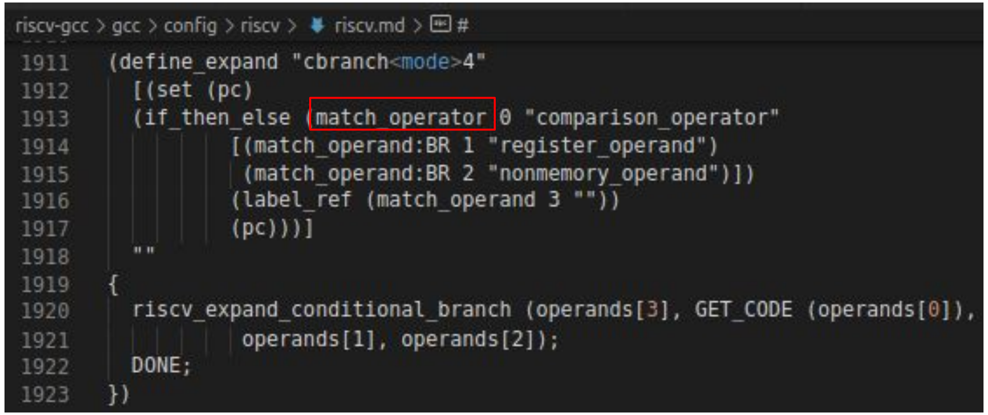
GCC指令添加：条件跳转指令模板的添加

添加一组允许指令使用立即数的条件跳 转指令

bne rs1,rs2,label
beq rs1,rs2,label

bnei rs1,imm,label
beqi rs1,imm,label

1. 查看是否存在SPN，是否riscv.md已有对应的实现

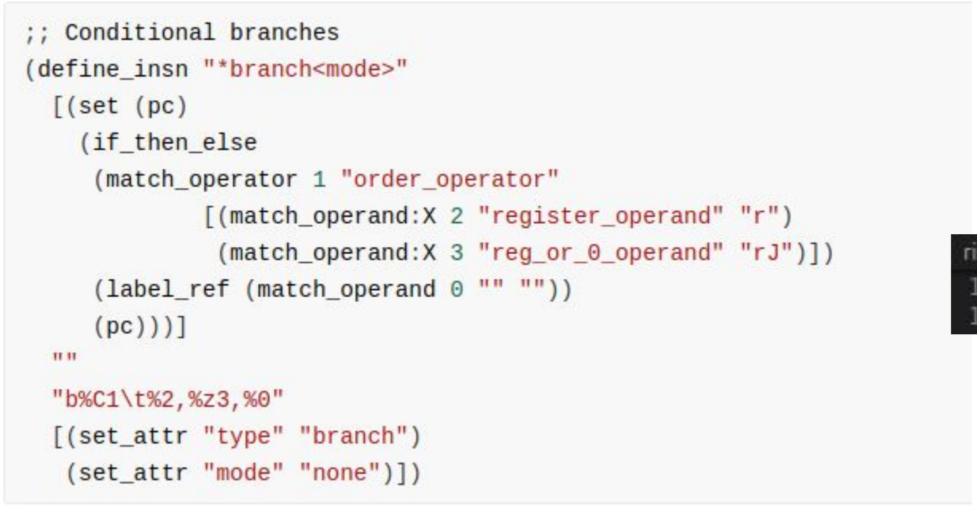


```
(define_expand "cbranch<mode>4"
  [(set (pc)
  (if_then_else (match_operator 0 "comparison_operator"
                [(match_operand:BR 1 "register_operand")
                 (match_operand:BR 2 "nonmemory_operand")])
                (label_ref (match_operand 3 ""))
                (pc)))]
  ""
{
  riscv_expand_conditional_branch (operands[3], GET_CODE (operands[0]),
                                   operands[1], operands[2]);
  DONE;
})
```

条件跳转spn在riscv.md的指令模板

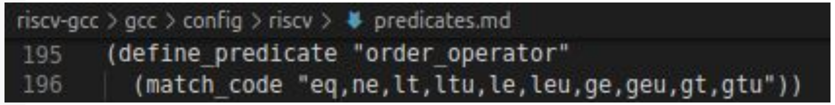
match_operator: 匹配rtl表达式code, NE,EQ,.... (见rtl.def)

GCC指令添加：条件跳转指令模板的添加



```
;; Conditional branches
(define_insn "*branch<mode>"
  [(set (pc)
    (if_then_else
     (match_operator 1 "order_operator"
             [(match_operand:X 2 "register_operand" "r")
              (match_operand:X 3 "reg_or_0_operand" "rJ")])
     (label_ref (match_operand 0 "" ""))
     (pc)))]
  ""
  "b%C1\t%2,%z3,%0"
  [(set_attr "type" "branch")
   (set_attr "mode" "none")])
```

riscv.md中条件跳转的指令模板

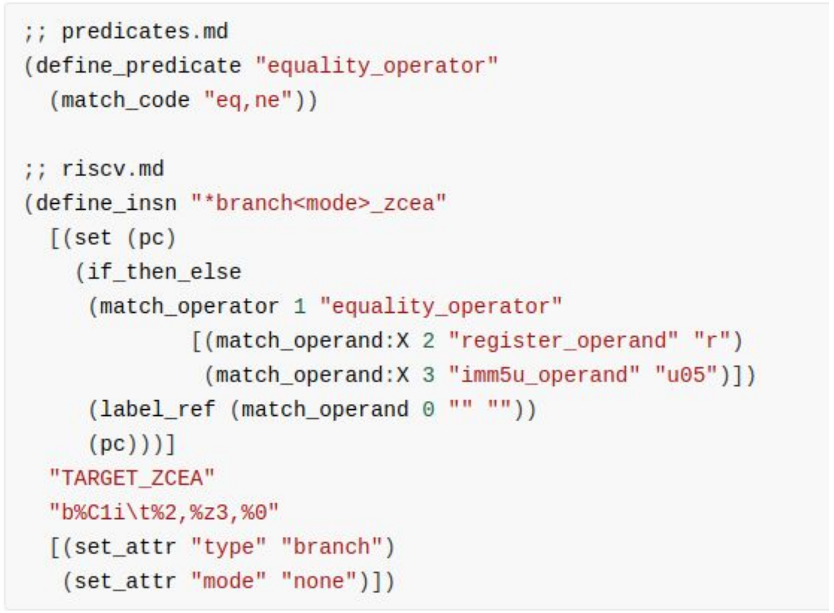


```
(define_predicate "order_operator"
  (match_code "eq,ne,lt,ltu,le,leu,ge,geu,gt,gtu"))
```

*可以通过写带条件跳转的c代码，加-da编译，查看任意一个expand pass之后的log文件,就能看到使用哪个指令模板得到的

GCC指令添加：条件跳转指令模板的添加

2. 添加beni/beqi指令模板



```
;; predicates.md
(define_predicate "equality_operator"
  (match_code "eq,ne"))

;; riscv.md
(define_insn "*branch<mode>_zcea"
  [(set (pc)
    (if_then_else
     (match_operator 1 "equality_operator"
             [(match_operand:X 2 "register_operand" "r")
              (match_operand:X 3 "imm5u_operand" "u05")])
     (label_ref (match_operand 0 "" ""))
     (pc)))]
  "TARGET_ZCEA"
  "b%C1i\t%2,%z3,%0"
  [(set_attr "type" "branch")
   (set_attr "mode" "none")])
```

bnei/beqi指令模板

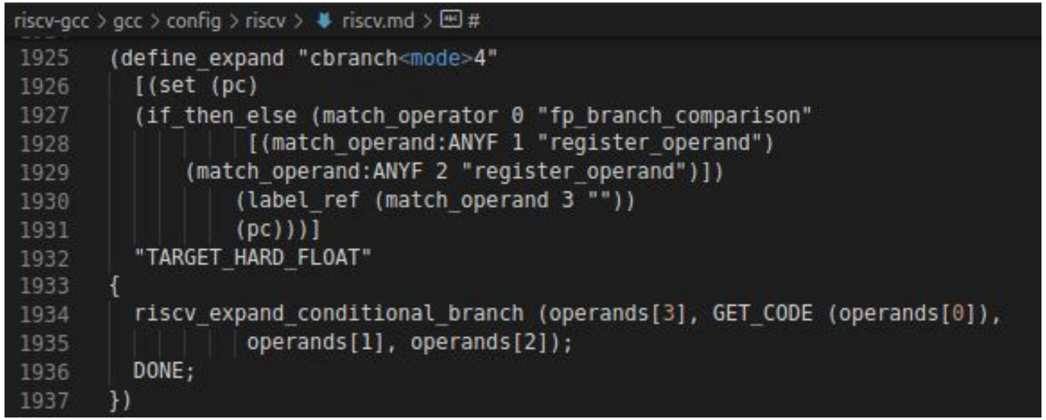
GCC指令添加：条件跳转指令模板的添加

bne rs1,rs2,label
beq rs1,rs2,label

bnei rs1,imm,label
beqi rs1,imm,label

添加一组允许指令使用立即数的条件跳 转指令

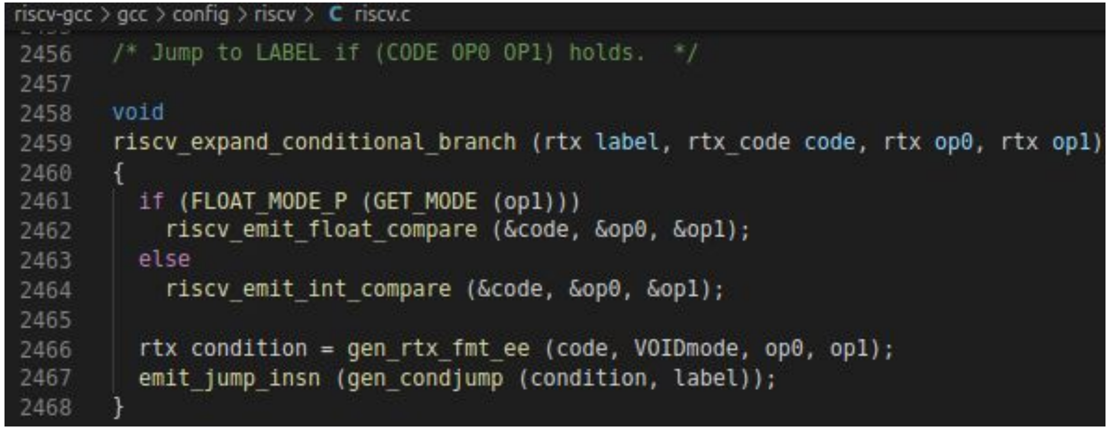
1. 查看是否存在SPN，是否riscv.md已有对应的实现



```
riscv-gcc > gcc > config > riscv > riscv.md > #
(define_expand "cbranch<mode>4"
  [(set (pc)
  (if_then_else (match_operator 0 "fp_branch_comparison"
                  [(match_operand:ANYF 1 "register_operand")
        (match_operand:ANYF 2 "register_operand")])
            (label_ref (match_operand 3 ""))
            (pc)))]
  "TARGET_HARD_FLOAT"
{
  riscv_expand_conditional_branch (operands[3], GET_CODE (operands[0]),
                  operands[1], operands[2]);
  DONE;
})
```

条件跳转spn在riscv.md的指令模板

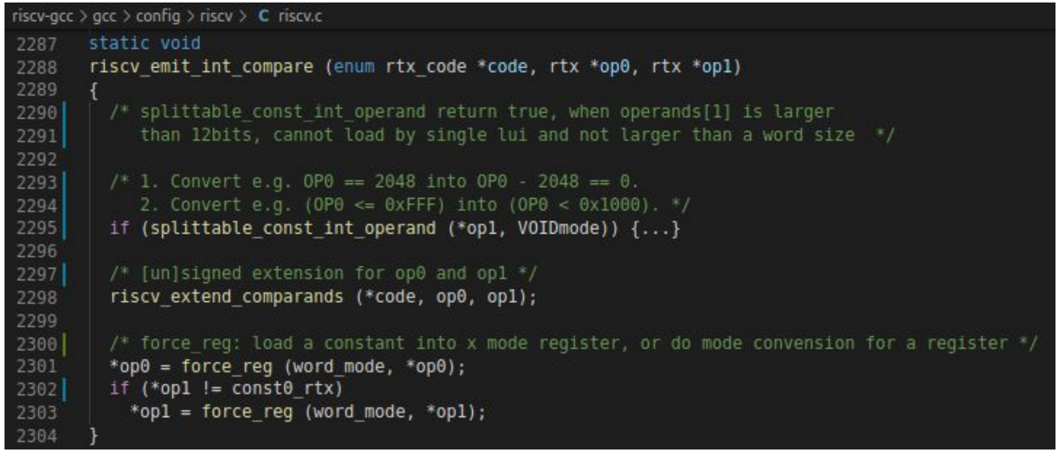
GCC指令添加：条件跳转指令模板的添加


riscv-gcc > gcc > config > riscv > C riscv.c

```c
/* Jump to LABEL if (CODE OP0 OP1) holds.  */

void
riscv_expand_conditional_branch (rtx label, rtx_code code, rtx op0, rtx op1)
{
  if (FLOAT_MODE_P (GET_MODE (op1)))
    riscv_emit_float_compare (&code, &op0, &op1);
  else
    riscv_emit_int_compare (&code, &op0, &op1);

  rtx condition = gen_rtx_fmt_ee (code, VOIDmode, op0, op1);
  emit_jump_insn (gen_condjump (condition, label));
}
```

riscv.c

GCC指令添加：条件跳转指令模板的添加

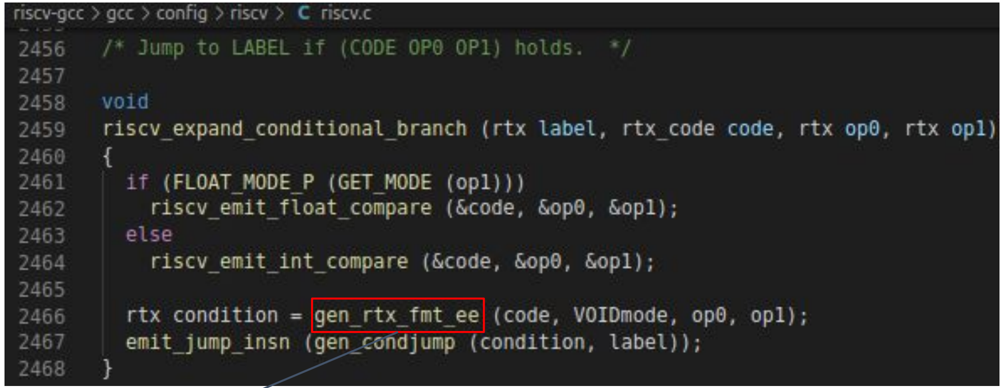

riscv-gcc > gcc > config > riscv > C riscv.c

```c
static void
riscv_emit_int_compare (enum rtx_code *code, rtx *op0, rtx *op1)
{
  /* splittable_const_int_operand return true, when operands[1] is larger
     than 12bits, cannot load by single lui and not larger than a word size  */

  /* 1. Convert e.g. OP0 == 2048 into OP0 - 2048 == 0.
     2. Convert e.g. (OP0 <= 0xFFF) into (OP0 < 0x1000). */
  if (splittable_const_int_operand (*op1, VOIDmode)) {...}

  /* [un]signed extension for op0 and op1 */
  riscv_extend_comparands (*code, op0, op1);

  /* force_reg: load a constant into x mode register, or do mode conversion for a register */
  *op0 = force_reg (word_mode, *op0);
  if (*op1 != const0_rtx)
    *op1 = force_reg (word_mode, *op1);
}
```

riscv_emit_int_compare

1. constant -> register
2. ?mode -> word mode

GCC指令添加：条件跳转指令模板的添加



riscv-gcc > gcc > config > riscv > riscv.c

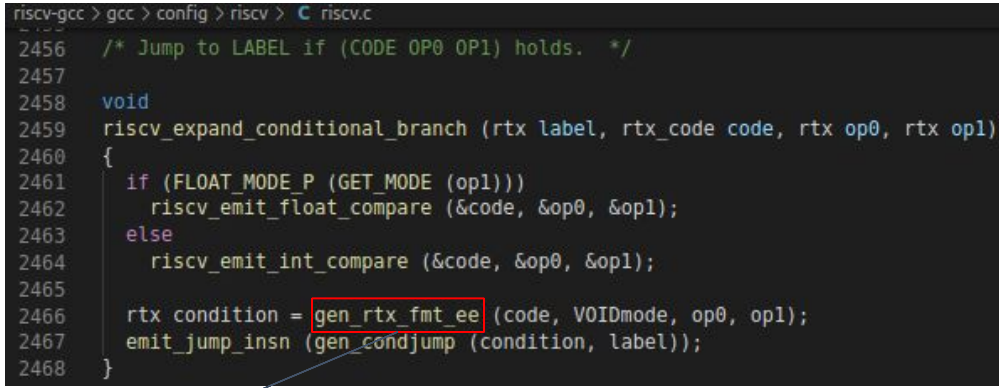
```
/* Jump to LABEL if (CODE OP0 OP1) holds.  */

void
riscv_expand_conditional_branch (rtx label, rtx_code code, rtx op0, rtx op1)
{
  if (FLOAT_MODE_P (GET_MODE (op1)))
    riscv_emit_float_compare (&code, &op0, &op1);
  else
    riscv_emit_int_compare (&code, &op0, &op1);

  rtx condition = gen_rtx_fmt_ee (code, VOIDmode, op0, op1);
  emit_jump_insn (gen_condjump (condition, label));
}
```

riscv.c

生成格式是“ee” rtl表达式

ISCAS 中国科学院软件研究所 Institute of Software Chinese Academy of Sciences | 智能软件研究中心 Intelligent Software Research Center

GCC指令添加：条件跳转指令模板的添加


riscv-gcc > gcc > config > riscv > C riscv.c

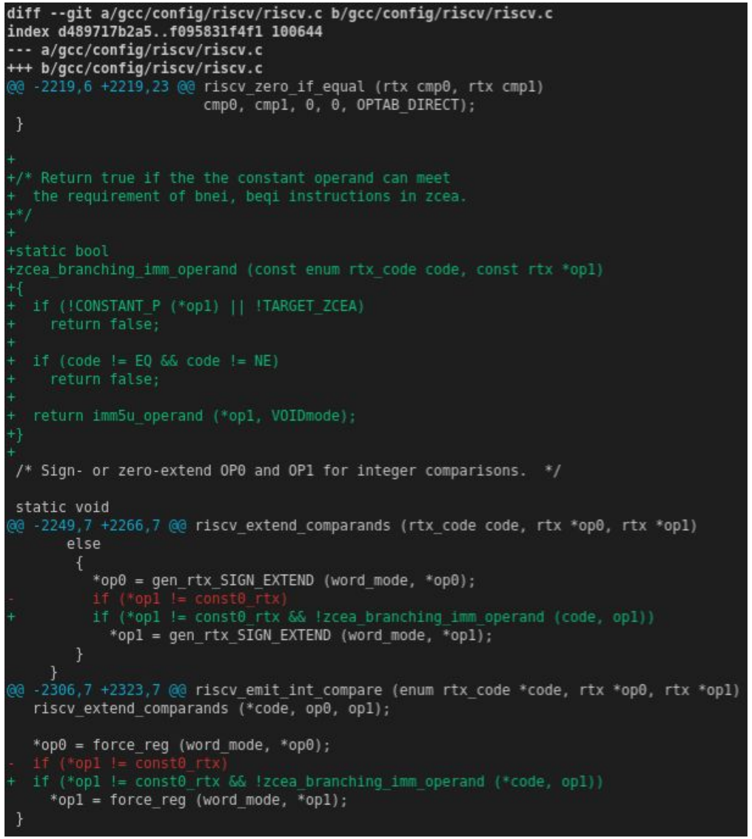
```
/* Jump to LABEL if (CODE OP0 OP1) holds.  */

void
riscv_expand_conditional_branch (rtx label, rtx_code code, rtx op0, rtx op1)
{
  if (FLOAT_MODE_P (GET_MODE (op1)))
    riscv_emit_float_compare (&code, &op0, &op1);
  else
    riscv_emit_int_compare (&code, &op0, &op1);

  rtx condition = gen_rtx_fmt_ee (code, VOIDmode, op0, op1);
  emit_jump_insn (gen_condjump (condition, label));
}
```

riscv.c

生成格式是“ee” rtl表达式
（check rtl.def for more information)

GCC指令添加：条件跳转指令模板的添加

3. 修改cbranch中准备阶段对operand3的强行加载到word mode寄存器的操作

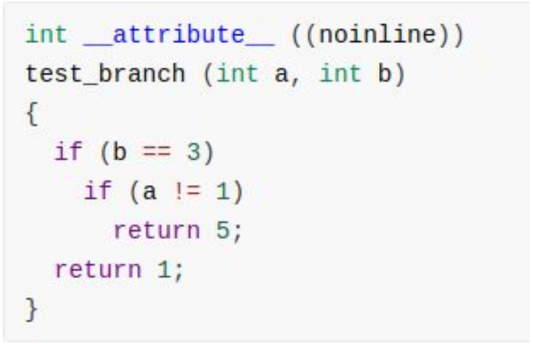


```
diff --git a/gcc/config/riscv/riscv.c b/gcc/config/riscv/riscv.c
index d489717b2a5..f095831f4f1 100644
--- a/gcc/config/riscv/riscv.c
+++ b/gcc/config/riscv/riscv.c
@@ -2219,6 +2219,23 @@ riscv_zero_if_equal (rtx cmp0, rtx cmp1)
                       cmp0, cmp1, 0, 0, OPTAB_DIRECT);
 }

+
+/* Return true if the the constant operand can meet
+  the requirement of bnei, beqi instructions in zcea.
+*/
+
+static bool
+zcea_branching_imm_operand (const enum rtx_code code, const rtx *op1)
+{
+  if (!CONSTANT_P (*op1) || !TARGET_ZCEA)
+    return false;
+
+  if (code != EQ && code != NE)
+    return false;
+
+  return imm5u_operand (*op1, VOIDmode);
+}
+
 /* Sign- or zero-extend OP0 and OP1 for integer comparisons.  */

 static void
@@ -2249,7 +2266,7 @@ riscv_extend_comparands (rtx_code code, rtx *op0, rtx *op1)
       else
        {
          *op0 = gen_rtx_SIGN_EXTEND (word_mode, *op0);
-         if (*op1 != const0_rtx)
+         if (*op1 != const0_rtx && !zcea_branching_imm_operand (code, op1))
            *op1 = gen_rtx_SIGN_EXTEND (word_mode, *op1);
        }
     }
@@ -2306,7 +2323,7 @@ riscv_emit_int_compare (enum rtx_code *code, rtx *op0, rtx *op1)
   riscv_extend_comparands (*code, op0, op1);

   *op0 = force_reg (word_mode, *op0);
-  if (*op1 != const0_rtx)
+  if (*op1 != const0_rtx && !zcea_branching_imm_operand (*code, op1))
     *op1 = force_reg (word_mode, *op1);
 }
```

允许operand1在给定的条件下不发生constant到register的转换

GCC指令添加：条件跳转指令模板的添加

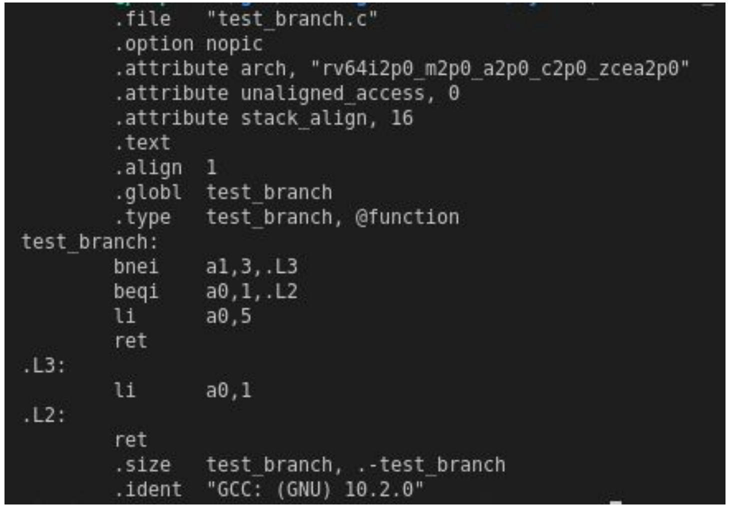
4. 编译测试



```c
int __attribute__ ((noinline))
test_branch (int a, int b)
{
  if (b == 3)
    if (a != 1)
      return 5;
  return 1;
}
```

test_branch.c

gcc -S



```
        .file   "test_branch.c"
        .option nopic
        .attribute arch, "rv64i2p0_m2p0_a2p0_c2p0_zcea2p0"
        .attribute unaligned_access, 0
        .attribute stack_align, 16
        .text
        .align  1
        .globl  test_branch
        .type   test_branch, @function
test_branch:
        bnei    a1,3,.L3
        beqi    a0,1,.L2
        li      a0,5
        ret
.L3:
        li      a0,1
.L2:
        ret
        .size   test_branch, .-test_branch
        .ident  "GCC: (GNU) 10.2.0"
```

谢 谢

欢迎交流合作