LLVM 中的 XTHeadVector 拓展

—— 实战 Intrinsic 的编译流程

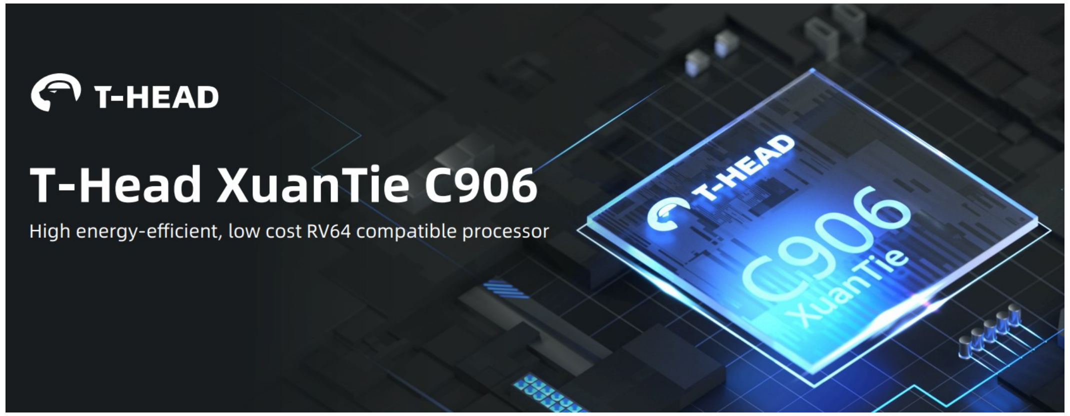
2023/12/15

总览

- XTHeadVector 介绍
- **如何向 LLVM 中添加一个拓展?**
- 目前进度 & 未来计划

什么是 XTHeadVector？

- 是 T-Head 现存的部分 RISC-V 芯片中的向量拓展
- 实质上是 RISC-V Vector 拓展 0.7.1 版本的修改版 [1]



XTHeadVector 对比 RISC-V Vector 0.7.1

- 版本号：1.0
- 指令重命名：`th.vmv.v.v` 而不是 `vmv.v.v`
- 寄存器重命名：`th.vtype` 而不是 `vtype`

- Zvmo 改名 XTHeadZvamo
- Zvlsseg 合并 XTHeadVector
- Zvediv 删除

为了让下面的函数工作，我们做了什么？

```c
#include <riscv_vector.h>
#include <stdint.h>
#include <stdio.h>

void memcpy_v(uint8_t *dst, const uint8_t *src, size_t n) {
  for (size_t vl; n > 0; n -= vl, src += vl, dst += vl) {
    vl = __riscv_vsetvl_e8m4(n);
    vuint8m4_t vec_src = __riscv_th_vle8_v_u8m4(src, vl);
    __riscv_th_vse8_v_u8m4(dst, vec_src, vl);
  }
}
```

如何向 LLVM 中添加一个拓展?

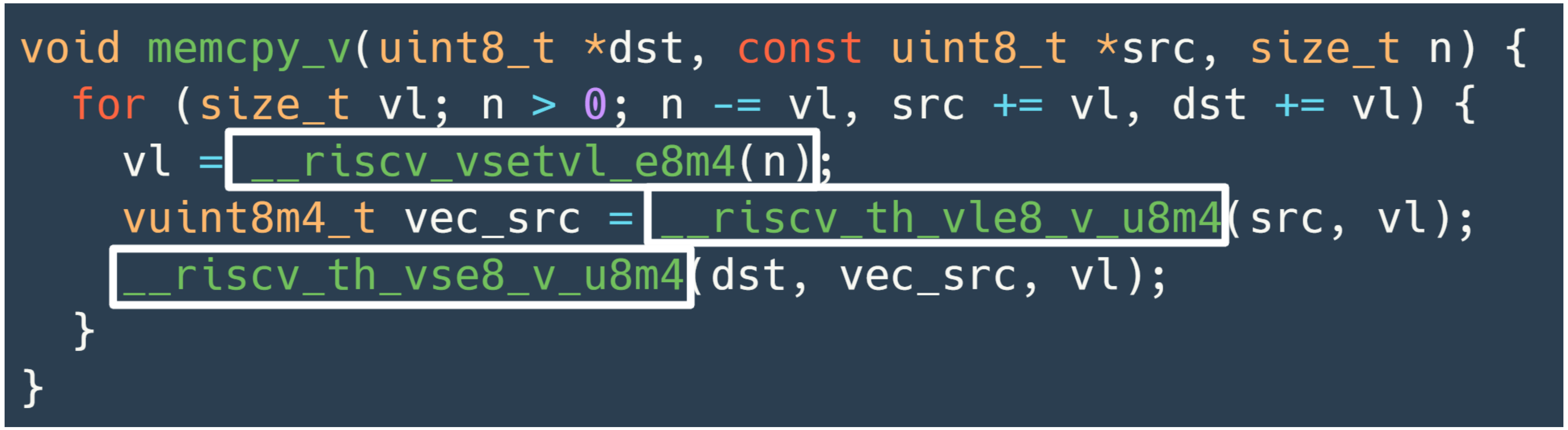
1. 注册拓展
2. 冲突检测（如果有）

3. **汇编器**
4. **LLVM 内建函数**
5. **Clang 内建函数**

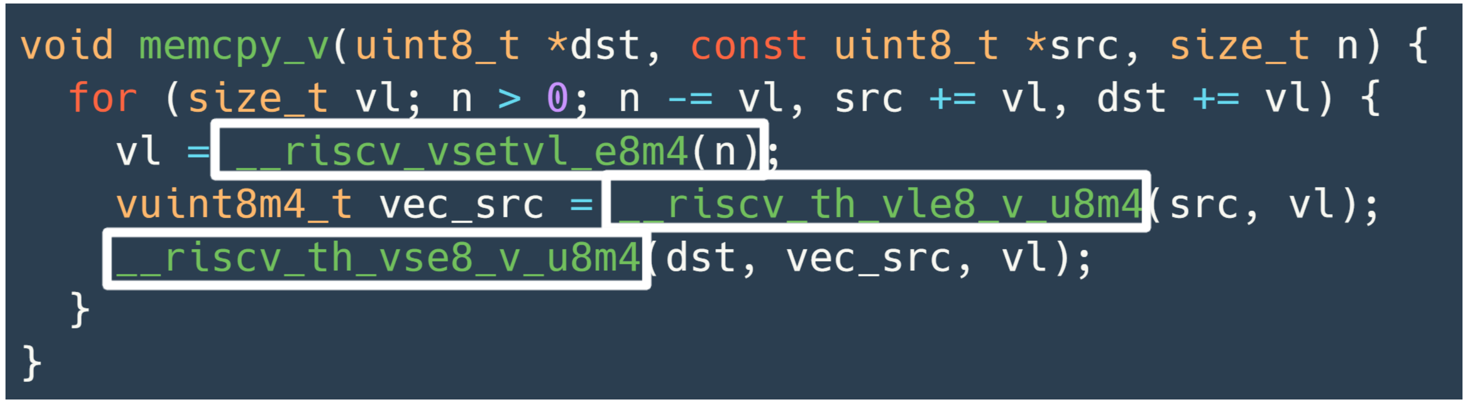
6. Pass 调整（如果有）

内建函数?

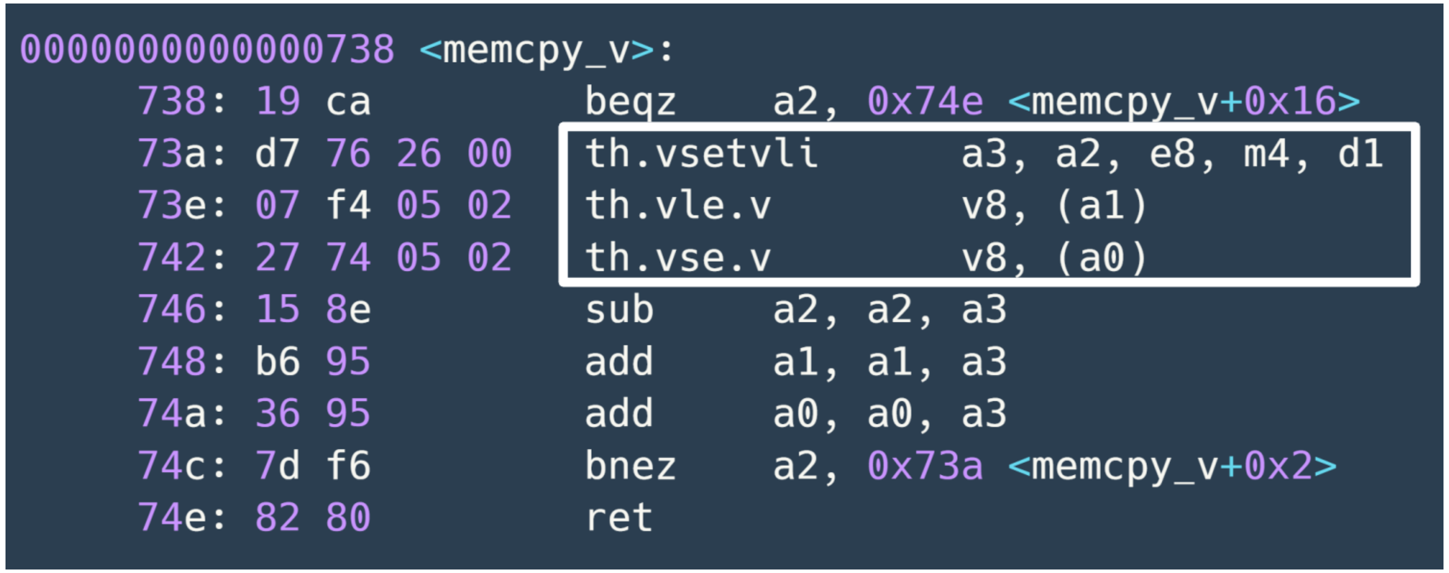
- 在 LLVM 框架下称为 **Intrinsic**
- 向**高级语言**暴露**低级操作**的函数



```c
void memcpy_v(uint8_t *dst, const uint8_t *src, size_t n) {
  for (size_t vl; n > 0; n -= vl, src += vl, dst += vl) {
    vl = __riscv_vsetvl_e8m4(n);
    vuint8m4_t vec_src = __riscv_th_vle8_v_u8m4(src, vl);
    __riscv_th_vse8_v_u8m4(dst, vec_src, vl);
  }
}
```



```
void memcpy_v(uint8_t *dst, const uint8_t *src, size_t n) {
  for (size_t vl; n > 0; n -= vl, src += vl, dst += vl) {
    vl = __riscv_vsetvl_e8m4(n);
    vuint8m4_t vec_src = __riscv_th_vle8_v_u8m4(src, vl);
    __riscv_th_vse8_v_u8m4(dst, vec_src, vl);
  }
}
```



```
0000000000000738 <memcpy_v>:
     738: 19 ca          beqz    a2, 0x74e <memcpy_v+0x16>
     73a: d7 76 26 00    th.vsetvli      a3, a2, e8, m4, d1
     73e: 07 f4 05 02    th.vle.v        v8, (a1)
     742: 27 74 05 02    th.vse.v        v8, (a0)
     746: 15 8e          sub     a2, a2, a3
     748: b6 95          add     a1, a1, a3
     74a: 36 95          add     a0, a0, a3
     74c: 7d f6          bnez    a2, 0x73a <memcpy_v+0x2>
     74e: 82 80          ret
```

通常被编译器特殊处理，在代码生成阶段**直接生成汇编指令**

理论讲完了，该实践了！

拓展定义、基础设施

- **llvm/lib/Support/RISCVISAInfo.cpp**
- XTHeadVector 和 V 拓展冲突

```
// NOTE: This table should be sorted alphabetically by extension name.
static const RISCVSupportedExtension SupportedExtensions[] = {
  {"a", RISCVExtensionVersion{2, 1}},
  {"c", RISCVExtensionVersion{2, 0}},
  // ...
  {"xtheadvector", RISCVExtensionVersion{1, 0}},
  {"xtheadzvamo", RISCVExtensionVersion{1, 0}},
  {"xventanacondops", RISCVExtensionVersion{1, 0}},
  // ...
}
```

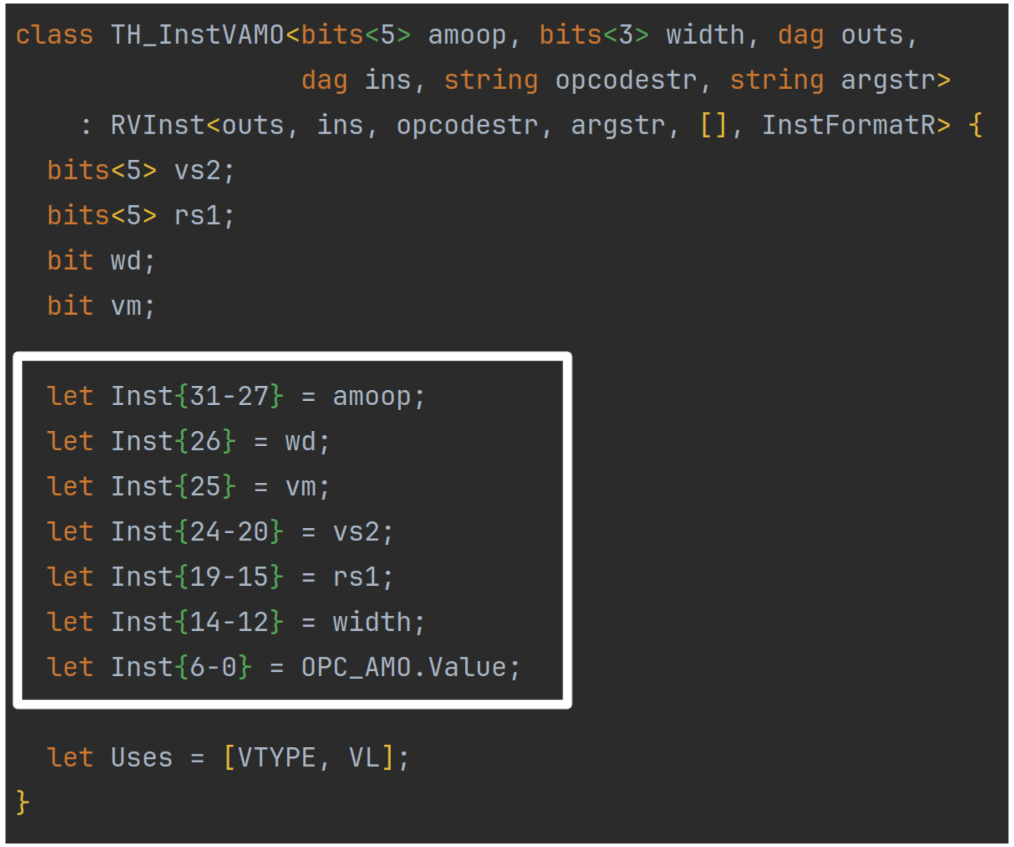
MC 汇编器

- **llvm/lib/Target/RISCV/RISCVInstrInfo.td**
- 使用 TableGen [2] 同时生成 AsmParser、AsmPrinter、Disassembler

- TableGen: **A DSL for meta-programming**.

如果有新的指令格式……

```
Format for Vector AMO Instructions under AMO major opcode
31      27 26   25   24        20 19         15 14    12 11        7 6      0
 amoop    |wd| vm |    vs2      |     rs1     | width | vs3/vd   |0101111|
   5        1    1      5             5           3        5         7
```



```
class TH_InstVAMO<bits<5> amoop, bits<3> width, dag outs,
                  dag ins, string opcodestr, string argstr>
    : RVInst<outs, ins, opcodestr, argstr, [], InstFormatR> {
  bits<5> vs2;
  bits<5> rs1;
  bit wd;
  bit vm;

  let Inst{31-27} = amoop;
  let Inst{26} = wd;
  let Inst{25} = vm;
  let Inst{24-20} = vs2;
  let Inst{19-15} = rs1;
  let Inst{14-12} = width;
  let Inst{6-0} = OPC_AMO.Value;

  let Uses = [VTYPE, VL];
}
```

根据规范定义指令格式，方便复用代码。

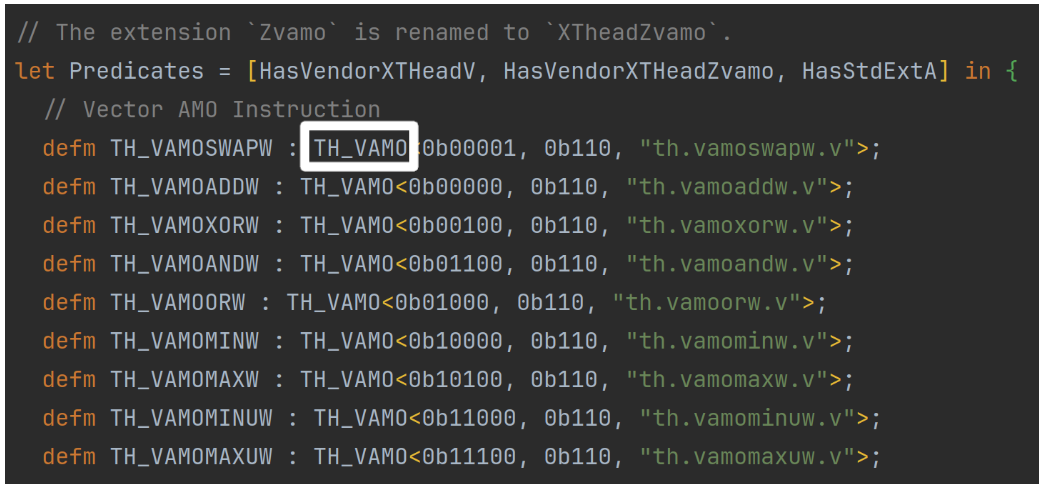
其他相关文件：
llvm/lilb/Target/RISCV/RISCVInstrFormats.td
llvm/include/llvm/Target/Target.td

TableGen，启动！

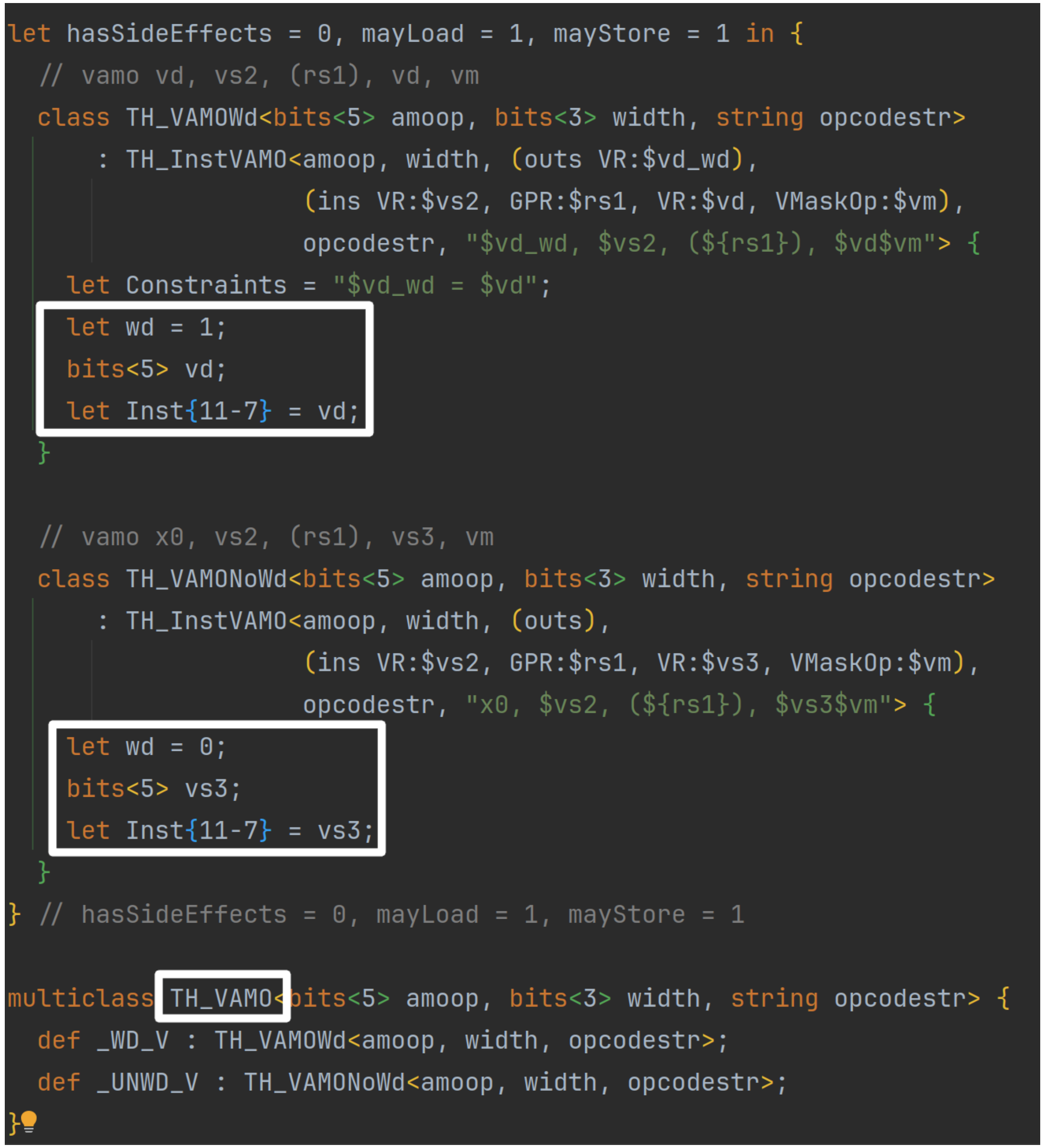
仅仅定义 Vamo 的指令格式远远不够：

1. 存在使用同种指令格式，但指令的**操作数**、**汇编语法**并不相同的情况

2. 同一条指令拥有多个**派生指令**



```
// The extension `Zvamo` is renamed to `XTheadZvamo`.
let Predicates = [HasVendorXTHeadV, HasVendorXTHeadZvamo, HasStdExtA] in {
  // Vector AMO Instruction
  defm TH_VAMOSWAPW : TH_VAMO<0b00001, 0b110, "th.vamoswapw.v">;
  defm TH_VAMOADDW : TH_VAMO<0b00000, 0b110, "th.vamoaddw.v">;
  defm TH_VAMOXORW : TH_VAMO<0b00100, 0b110, "th.vamoxorw.v">;
  defm TH_VAMOANDW : TH_VAMO<0b01100, 0b110, "th.vamoandw.v">;
  defm TH_VAMOORW : TH_VAMO<0b01000, 0b110, "th.vamoorw.v">;
  defm TH_VAMOMINW : TH_VAMO<0b10000, 0b110, "th.vamominw.v">;
  defm TH_VAMOMAXW : TH_VAMO<0b10100, 0b110, "th.vamomaxw.v">;
  defm TH_VAMOMINUW : TH_VAMO<0b11000, 0b110, "th.vamominuw.v">;
  defm TH_VAMOMAXUW : TH_VAMO<0b11100, 0b110, "th.vamomaxuw.v">;
```



```
let hasSideEffects = 0, mayLoad = 1, mayStore = 1 in {
  // vamo vd, vs2, (rs1), vd, vm
  class TH_VAMOWd<bits<5> amoop, bits<3> width, string opcodestr>
      : TH_InstVAMO<amoop, width, (outs VR:$vd_wd),
                    (ins VR:$vs2, GPR:$rs1, VR:$vd, VMaskOp:$vm),
                    opcodestr, "$vd_wd, $vs2, (${rs1}), $vd$vm"> {
    let Constraints = "$vd_wd = $vd";
    let wd = 1;
    bits<5> vd;
    let Inst{11-7} = vd;
  }

  // vamo x0, vs2, (rs1), vs3, vm
  class TH_VAMONoWd<bits<5> amoop, bits<3> width, string opcodestr>
      : TH_InstVAMO<amoop, width, (outs),
                    (ins VR:$vs2, GPR:$rs1, VR:$vs3, VMaskOp:$vm),
                    opcodestr, "x0, $vs2, (${rs1}), $vs3$vm"> {
    let wd = 0;
    bits<5> vs3;
    let Inst{11-7} = vs3;
  }
} // hasSideEffects = 0, mayLoad = 1, mayStore = 1

multiclass TH_VAMO<bits<5> amoop, bits<3> width, string opcodestr> {
  def _WD_V : TH_VAMOWd<amoop, width, opcodestr>;
  def _UNWD_V : TH_VAMONoWd<amoop, width, opcodestr>;
}
```

来点测试

```
th.vsetvli a2, a1, e8
# CHECK: error: operand must be e[8|16|32|64|128|256|512|1024],m[1|2|4|8],d[1|2|4|8]
```

```
        th.vamoaddw.v v4, v8, (a1), v4
        # CHECK-INST: th.vamoaddw.v v4, v8, (a1), v4
        # CHECK-ENCODING: [0x2f,0xe2,0x85,0x06]

        th.vamoaddw.v zero, v8, (a1), v4
        # CHECK-INST: th.vamoaddw.v x0, v8, (a1), v4
        # CHECK-ENCODING: [0x2f,0xe2,0x85,0x02]

        th.vamoaddd.v v4, v8, (a1), v4
        # CHECK-INST: th.vamoaddd.v v4, v8, (a1), v4
        # CHECK-ENCODING: [0x2f,0xf2,0x85,0x06]

        th.vamoaddd.v zero, v8, (a1), v4
        # CHECK-INST: th.vamoaddd.v x0, v8, (a1), v4
        # CHECK-ENCODING: [0x2f,0xf2,0x85,0x02]
```

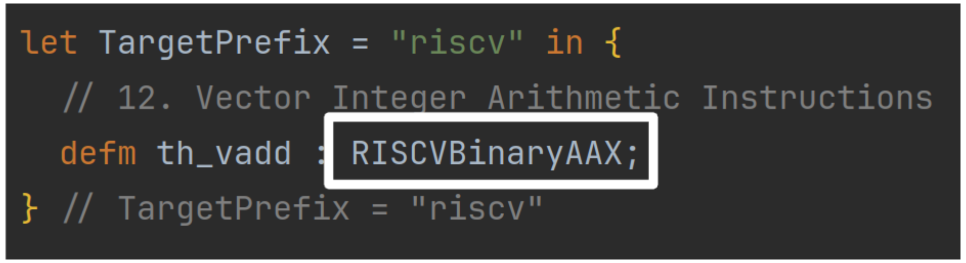
LLVM Intrinsic

1. 添加 Intrinsic 定义：名字、类型等：
   llvm/include/llvm/IR/IntrinsicsRISCV.td
2. 给出展开方式：
   - **使用 Pattern 自动匹配并展开：全自动的 TableGen！**
   - 参数过多（如 Load/Store），或更细粒度的自动化 Pseudo 展开
   - 展开过程中需要更多信息，拥抱 usesCustomInserter

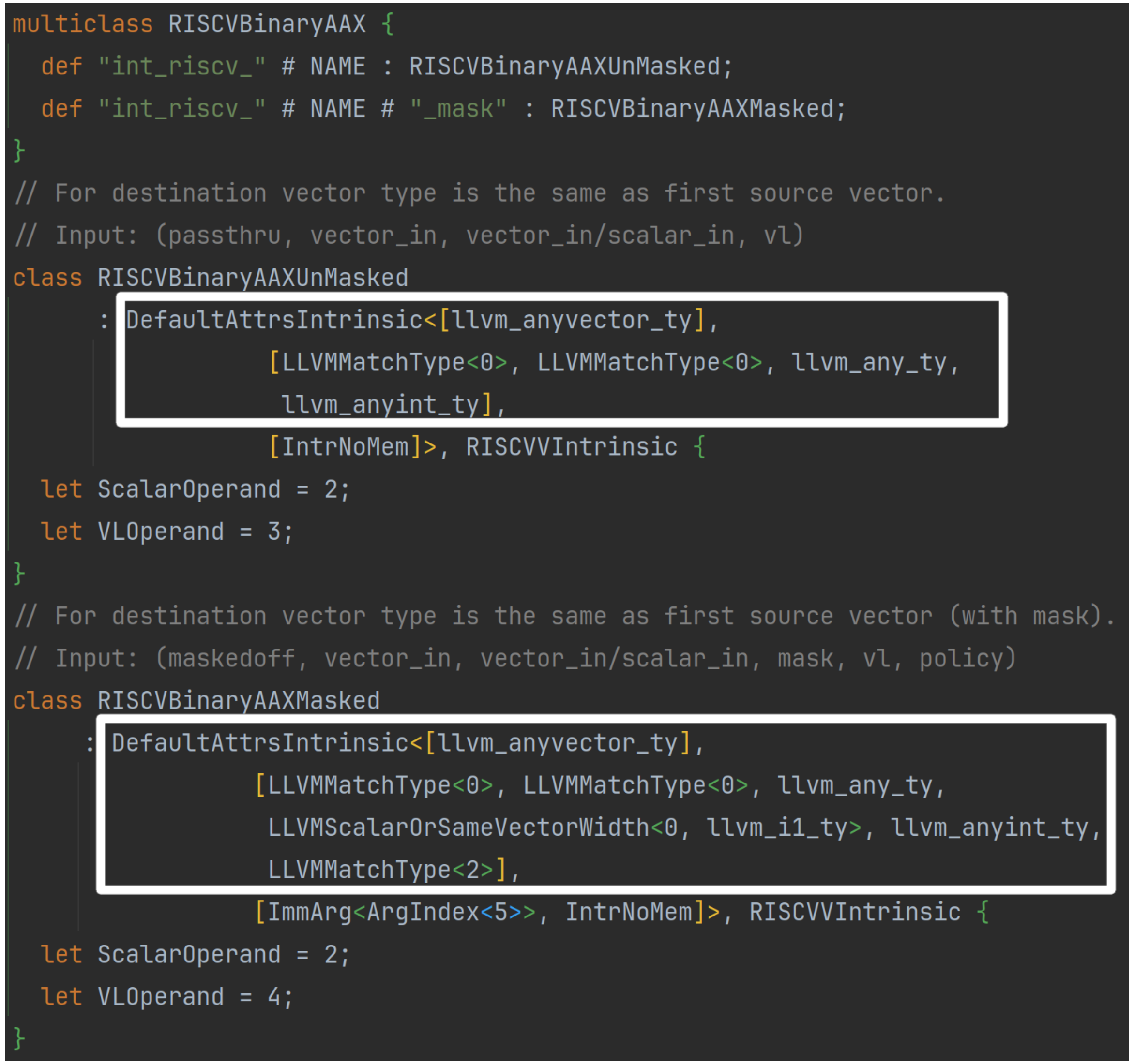
LLVM Intrinsic： TableGen 全自动展开

给出 vadd 指令的 intrinsic 的符号:

- TableGen 中名为 `int_riscv_th_vadd`
- IR 中名为 `llvm.riscv.th.vadd`
- 类型如右图所示

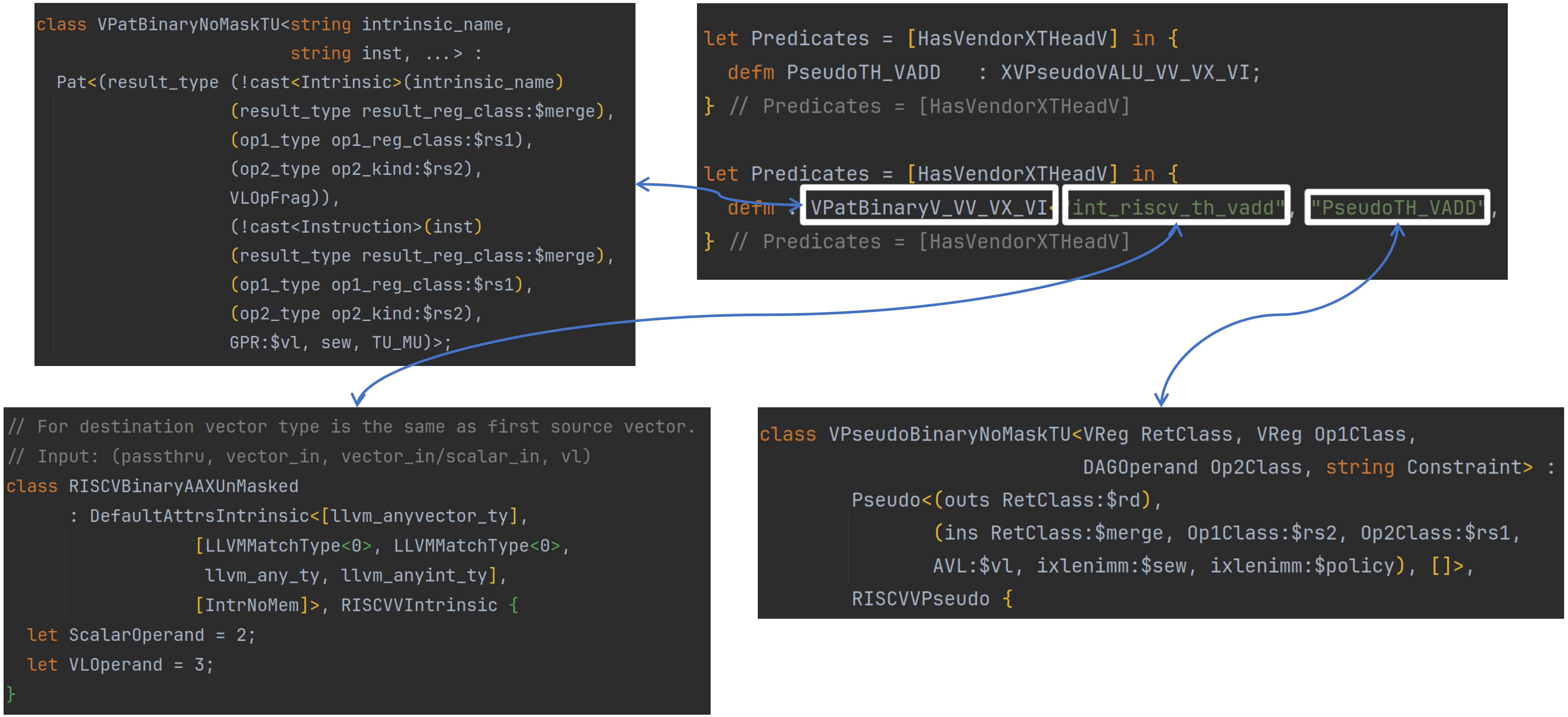


```
let TargetPrefix = "riscv" in {
  // 12. Vector Integer Arithmetic Instructions
  defm th_vadd : RISCVBinaryAAX;
} // TargetPrefix = "riscv"
```



```
multiclass RISCVBinaryAAX {
  def "int_riscv_" # NAME : RISCVBinaryAAXUnMasked;
  def "int_riscv_" # NAME # "_mask" : RISCVBinaryAAXMasked;
}
// For destination vector type is the same as first source vector.
// Input: (passthru, vector_in, vector_in/scalar_in, vl)
class RISCVBinaryAAXUnMasked
      : DefaultAttrsIntrinsic<[llvm_anyvector_ty],
                  [LLVMMatchType<0>, LLVMMatchType<0>, llvm_any_ty,
                   llvm_anyint_ty],
                  [IntrNoMem]>, RISCVVIntrinsic {
  let ScalarOperand = 2;
  let VLOperand = 3;
}
// For destination vector type is the same as first source vector (with mask).
// Input: (maskedoff, vector_in, vector_in/scalar_in, mask, vl, policy)
class RISCVBinaryAAXMasked
      : DefaultAttrsIntrinsic<[llvm_anyvector_ty],
                  [LLVMMatchType<0>, LLVMMatchType<0>, llvm_any_ty,
                   LLVMScalarOrSameVectorWidth<0, llvm_i1_ty>, llvm_anyint_ty,
                   LLVMMatchType<2>],
                  [ImmArg<ArgIndex<5>>, IntrNoMem]>, RISCVVIntrinsic {
  let ScalarOperand = 2;
  let VLOperand = 4;
}
```

思想：匹配 Intrinsic 的参数，传递给 Instruction



```
class VPatBinaryNoMaskTU<string intrinsic_name,
                         string inst, ...> :
  Pat<(result_type (!cast<Intrinsic>(intrinsic_name)
                   (result_type result_reg_class:$merge),
                   (op1_type op1_reg_class:$rs1),
                   (op2_type op2_kind:$rs2),
                   VLOpFrag)),
                   (!cast<Instruction>(inst)
                   (result_type result_reg_class:$merge),
                   (op1_type op1_reg_class:$rs1),
                   (op2_type op2_kind:$rs2),
                   GPR:$vl, sew, TU_MU)>;
```

```
let Predicates = [HasVendorXTHeadV] in {
  defm PseudoTH_VADD   : XVPseudoVALU_VV_VX_VI;
} // Predicates = [HasVendorXTHeadV]

let Predicates = [HasVendorXTHeadV] in {
  defm : VPatBinaryV_VV_VX_VI<"int_riscv_th_vadd", "PseudoTH_VADD",
} // Predicates = [HasVendorXTHeadV]
```

```
// For destination vector type is the same as first source vector.
// Input: (passthru, vector_in, vector_in/scalar_in, vl)
class RISCVBinaryAAXUnMasked
      : DefaultAttrsIntrinsic<[llvm_anyvector_ty],
                  [LLVMMatchType<0>, LLVMMatchType<0>,
                   llvm_any_ty, llvm_anyint_ty],
                  [IntrNoMem]>, RISCVVIntrinsic {
  let ScalarOperand = 2;
  let VLOperand = 3;
}
```

```
class VPseudoBinaryNoMaskTU<VReg RetClass, VReg Op1Class,
                            DAGOperand Op2Class, string Constraint> :
      Pseudo<(outs RetClass:$rd),
             (ins RetClass:$merge, Op1Class:$rs2, Op2Class:$rs1,
             AVL:$vl, ixlenimm:$sew, ixlenimm:$policy), []>,
      RISCVVPseudo {
```

还是来点测试

```
define <vscale x 8 x i8> @test(<vscale x 8 x i8> %0, <vscale x 8 x i8> %1, iXLen %2) nounwind {
; CHECK-LABEL: test:
; CHECK:       # %bb.0: # %entry
; CHECK-NEXT:    th.vsetvli zero, a0, e8, m1, d1
; CHECK-NEXT:    th.vadd.vv v8, v8, v9
; CHECK-NEXT:    ret
entry:
  %a = call <vscale x 8 x i8> @llvm.riscv.th.vadd.nxv8i8.nxv8i8(
    <vscale x 8 x i8> undef,
    <vscale x 8 x i8> %0,
    <vscale x 8 x i8> %1,
    iXLen %2)

  ret <vscale x 8 x i8> %a
}
```

LLVM Intrinsic：其他展开方式

- 参数过多（如 Load/Store），或更细粒度的自动化 Pseudo 展开：
  llvm/lib/Target/RISCV/RISCVExpandPseudoInsts.cpp
  llvm/lib/Target/RISCV/RISCVISelDAGToDAG.cpp

- 展开过程中需要更多信息，拥抱 usesCustomInserter：
  llvm/lib/Target/RISCV/RISCVISelLowering.cpp

- TableGen 老矣！

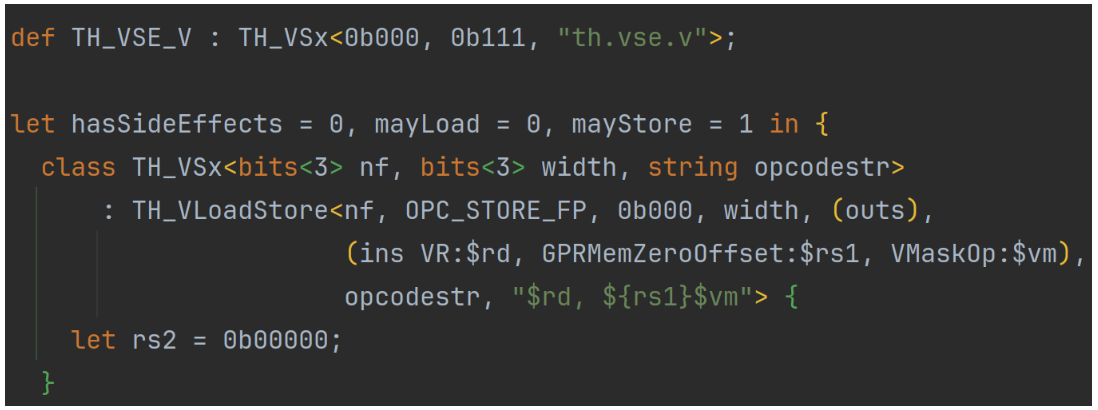
Clang Intrinsic

- 最后一步！ 目标：`riscv_vector.td`

- 同样存在多种将 Clang intrinsic 展开为 LLVM intrinsic 的方式！
- 同样介绍全自动的 TableGen！

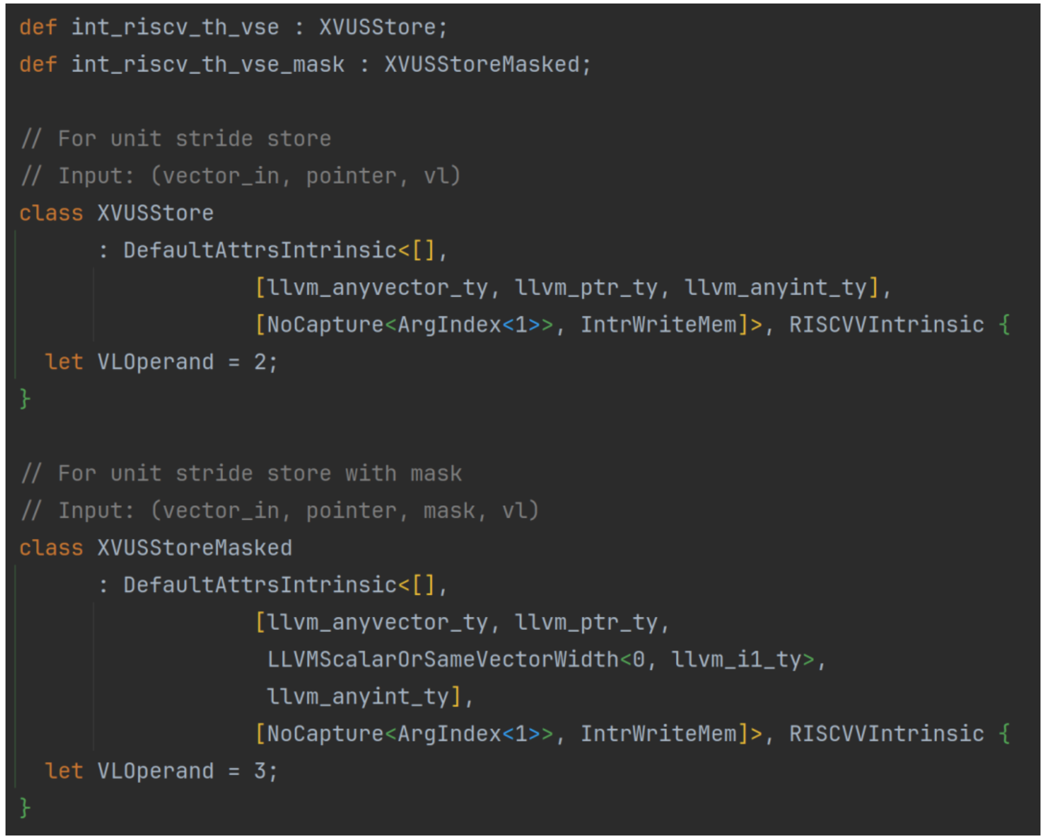
Clang Intrinsic：以 vse 为例

首先我们要有 vse 的 MC 指令和 LLVM Intrinsic：



```
def TH_VSE_V : TH_VSx<0b000, 0b111, "th.vse.v">;

let hasSideEffects = 0, mayLoad = 0, mayStore = 1 in {
  class TH_VSx<bits<3> nf, bits<3> width, string opcodestr>
      : TH_VLoadStore<nf, OPC_STORE_FP, 0b000, width, (outs),
                      (ins VR:$rd, GPRMemZeroOffset:$rs1, VMaskOp:$vm),
                      opcodestr, "$rd, ${rs1}$vm"> {
    let rs2 = 0b00000;
  }
```



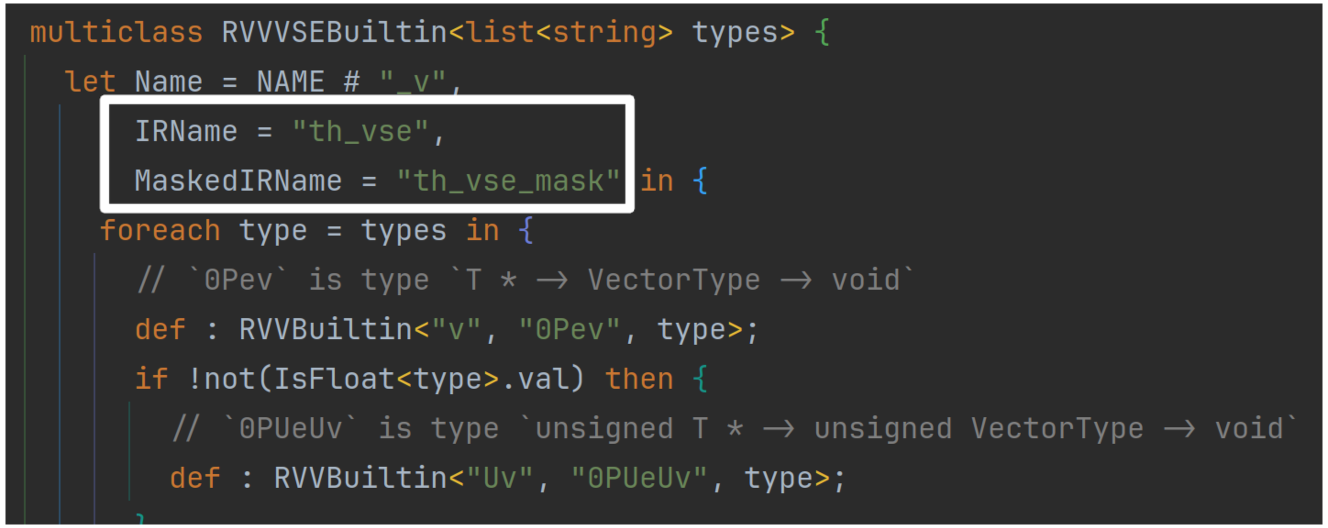
```
def int_riscv_th_vse : XVUSStore;
def int_riscv_th_vse_mask : XVUSStoreMasked;

// For unit stride store
// Input: (vector_in, pointer, vl)
class XVUSStore
      : DefaultAttrsIntrinsic<[],
                  [llvm_anyvector_ty, llvm_ptr_ty, llvm_anyint_ty],
                  [NoCapture<ArgIndex<1>>, IntrWriteMem]>, RISCVVIntrinsic {
  let VLOperand = 2;
}

// For unit stride store with mask
// Input: (vector_in, pointer, mask, vl)
class XVUSStoreMasked
      : DefaultAttrsIntrinsic<[],
                  [llvm_anyvector_ty, llvm_ptr_ty,
                   LLVMScalarOrSameVectorWidth<0, llvm_i1_ty>,
                   llvm_anyint_ty],
                  [NoCapture<ArgIndex<1>>, IntrWriteMem]>, RISCVVIntrinsic {
  let VLOperand = 3;
}
```

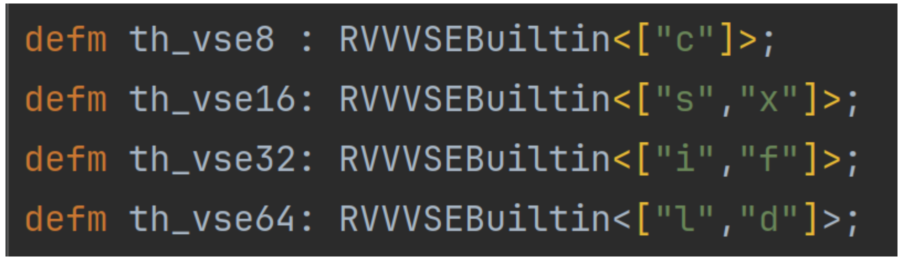
Clang Intrinsic：以 vse 为例

然后给出 Clang Intrinsic 的定义：



```
multiclass RVVVSEBuiltin<list<string> types> {
  let Name = NAME # "_v",
    IRName = "th_vse",
    MaskedIRName = "th_vse_mask" in {
    foreach type = types in {
      // `0Pev` is type `T * → VectorType → void`
      def : RVVBuiltin<"v", "0Pev", type>;
      if !not(IsFloat<type>.val) then {
        // `0PUeUv` is type `unsigned T * → unsigned VectorType → void`
        def : RVVBuiltin<"Uv", "0PUeUv", type>;
```

给出对应的 LLVM intrinsic，并指定 Clang intrinsic 的类型



```
defm th_vse8 : RVVVSEBuiltin<["c"]>;
defm th_vse16: RVVVSEBuiltin<["s","x"]>;
defm th_vse32: RVVVSEBuiltin<["i","f"]>;
defm th_vse64: RVVVSEBuiltin<["l","d"]>;
```

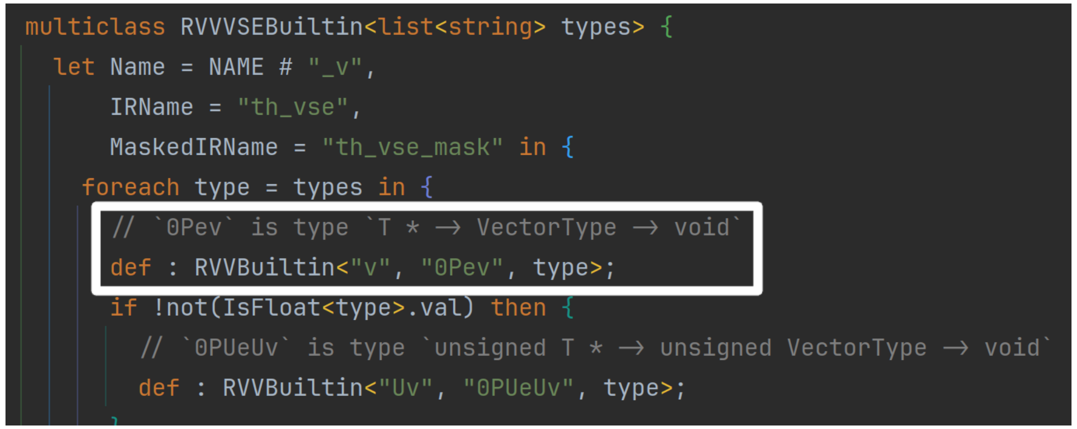
最后，使用上面的结构生成 Clang intrinsic 的符号：

- 在 C 语言中名为：`__riscv_th_vse8` 等等

Clang Intrinsic：以 vse 为例

刚刚是不是有什么问题？类型不一致？

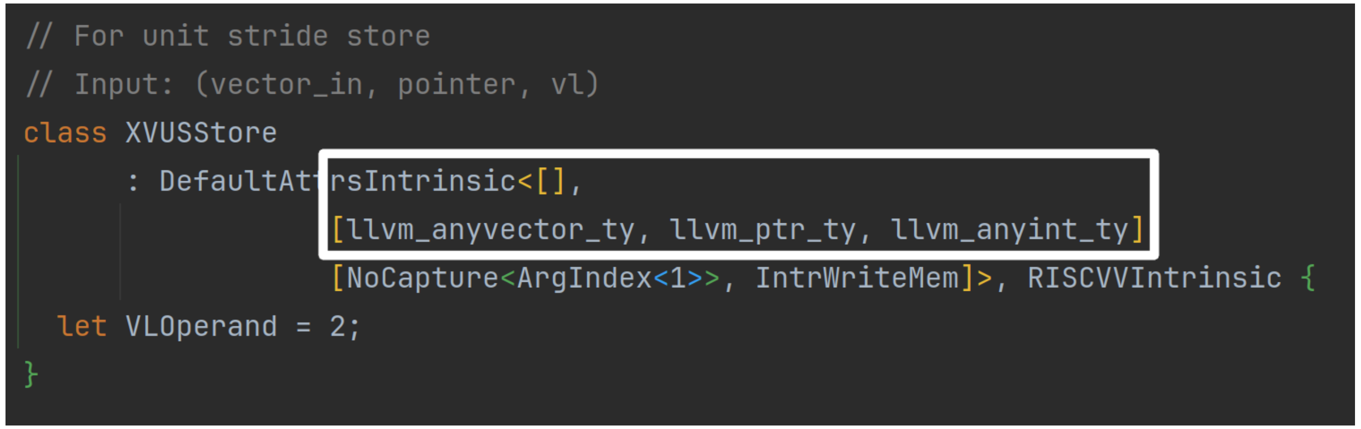
`T * → VectorType → VL → void`



```
multiclass RVVVSEBuiltin<list<string> types> {
  let Name = NAME # "_v",
      IRName = "th_vse",
      MaskedIRName = "th_vse_mask" in {
    foreach type = types in {
      // `0Pev` is type `T * -> VectorType -> void`
      def : RVVBuiltin<"v", "0Pev", type>;
      if !not(IsFloat<type>.val) then {
        // `0PUeUv` is type `unsigned T * -> unsigned VectorType -> void`
        def : RVVBuiltin<"Uv", "0PUeUv", type>;
```

Clang Intrinsic

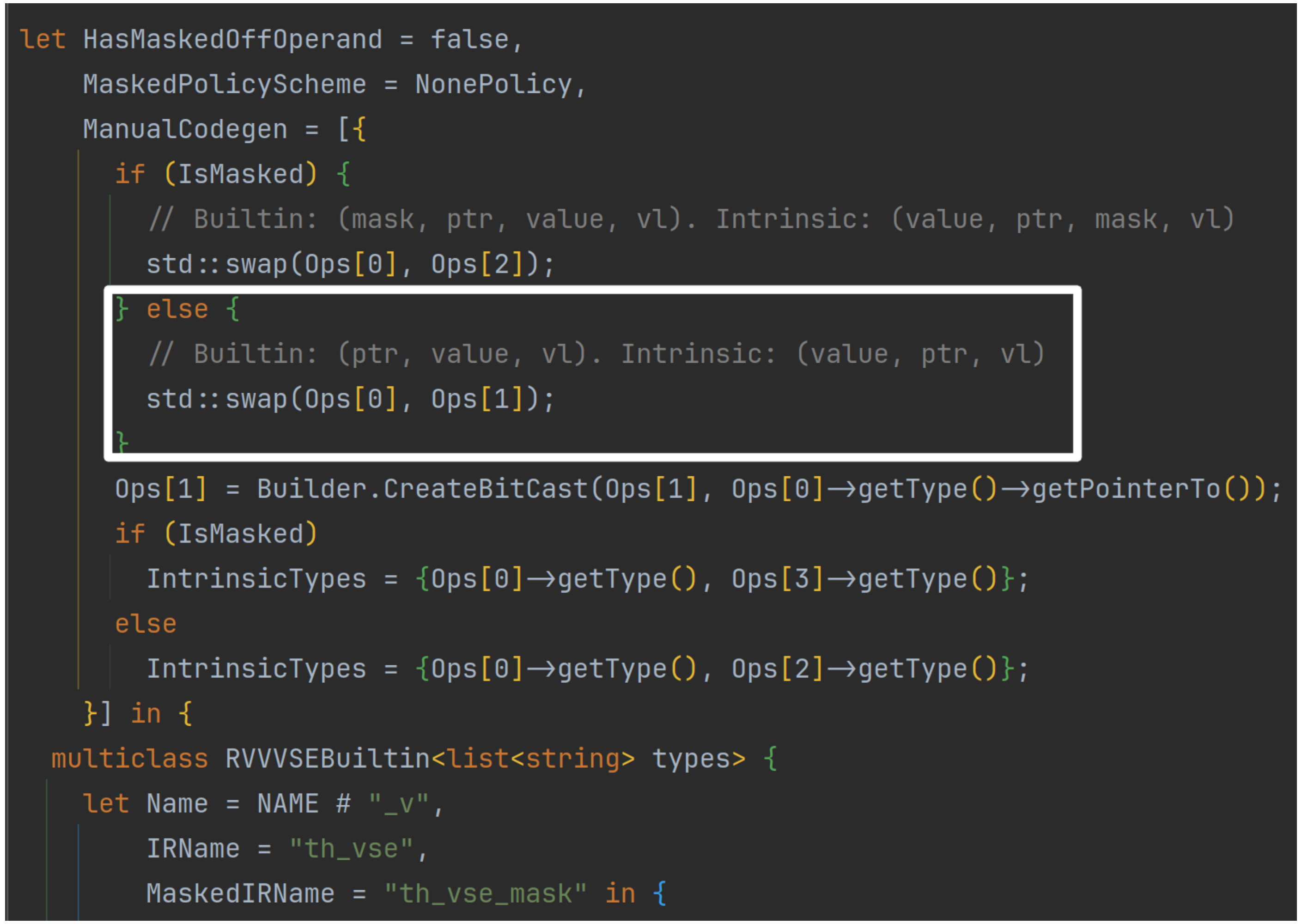
`VectorType → T * → VL → void`



```
// For unit stride store
// Input: (vector_in, pointer, vl)
class XVUSStore
      : DefaultAttrsIntrinsic<[],
                              [llvm_anyvector_ty, llvm_ptr_ty, llvm_anyint_ty],
                              [NoCapture<ArgIndex<1>>, IntrWriteMem]>, RISCVVIntrinsic {
  let VLOperand = 2;
}
```

LLVM Intrinsic

Clang Intrinsic：以 vse 为例



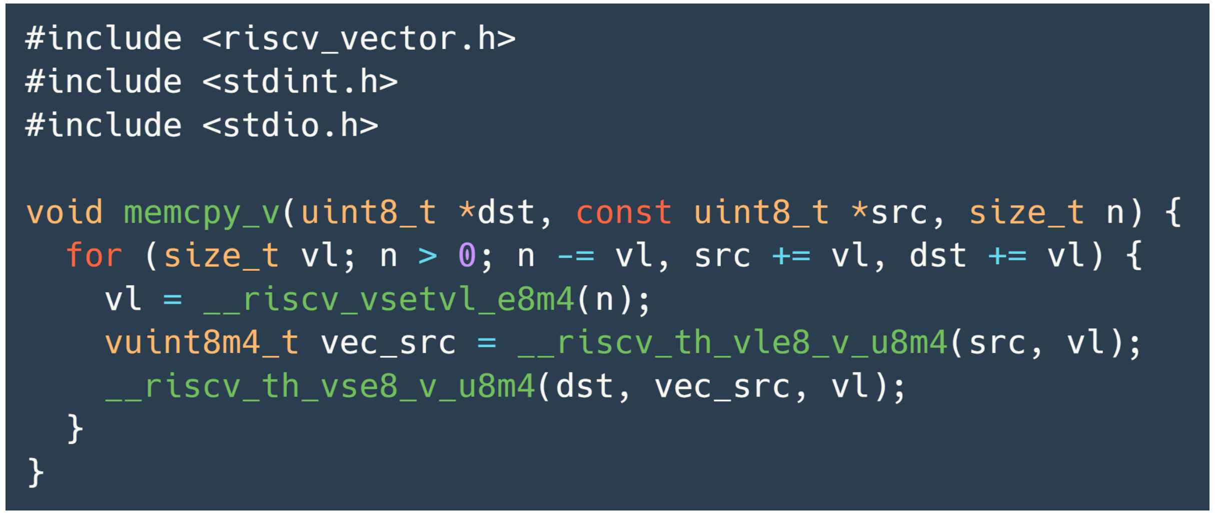
```
let HasMaskedOffOperand = false,
    MaskedPolicyScheme = NonePolicy,
    ManualCodegen = [{
      if (IsMasked) {
        // Builtin: (mask, ptr, value, vl). Intrinsic: (value, ptr, mask, vl)
        std::swap(Ops[0], Ops[2]);
      } else {
        // Builtin: (ptr, value, vl). Intrinsic: (value, ptr, vl)
        std::swap(Ops[0], Ops[1]);
      }
      Ops[1] = Builder.CreateBitCast(Ops[1], Ops[0]->getType()->getPointerTo());
      if (IsMasked)
        IntrinsicTypes = {Ops[0]->getType(), Ops[3]->getType()};
      else
        IntrinsicTypes = {Ops[0]->getType(), Ops[2]->getType()};
    }] in {
  multiclass RVVVSEBuiltin<list<string> types> {
    let Name = NAME # "_v",
        IRName = "th_vse",
        MaskedIRName = "th_vse_mask" in {
```

只拘泥于 TableGen 本身，往往会陷入意想不到的境地……

除非，超越 TableGen

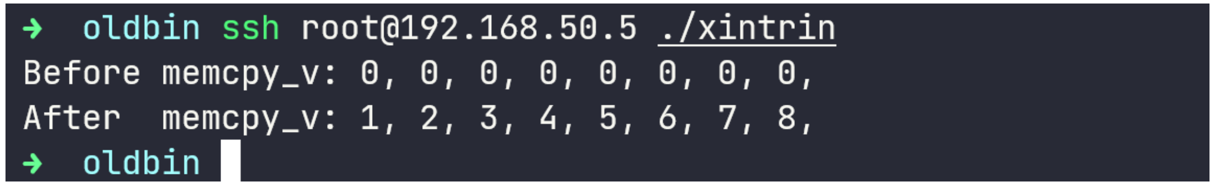
继续来点测试

回到最开始的代码，完整用例见：
https://github.com/ruyisdk/llvm-project/pull/33#issuecomment-1840167856



```c
#include <riscv_vector.h>
#include <stdint.h>
#include <stdio.h>

void memcpy_v(uint8_t *dst, const uint8_t *src, size_t n) {
  for (size_t vl; n > 0; n -= vl, src += vl, dst += vl) {
    vl = __riscv_vsetvl_e8m4(n);
    vuint8m4_t vec_src = __riscv_th_vle8_v_u8m4(src, vl);
    __riscv_th_vse8_v_u8m4(dst, vec_src, vl);
  }
}
```



```
➜  oldbin ssh root@192.168.50.5 ./xintrin
Before memcpy_v: 0, 0, 0, 0, 0, 0, 0, 0,
After  memcpy_v: 1, 2, 3, 4, 5, 6, 7, 8,
➜  oldbin
```

Pass 调优：以“插入 vsetvli” 为例

- 背景：VL 和 VType 决定**大部分**向量指令操作的**向量操作数的“范围”**。

1. 计算每个 BasicBlock（简称 **BB**）中对 VL 和 VType 的修改
   - 例如：已有的 vsetvli、Fault-Only-First Load 等指令
2. 数据流分析，得到每个 **BB** 进入时的 VL 和 Vtype
3. 如果 **BB** 之间存在 VL 和 VType 改变，在需要时插入一条 vsetvli

Pass 调优： 以“插入 vsetvli” 为例

- XTHeadVector 的 vtype 寄存器：

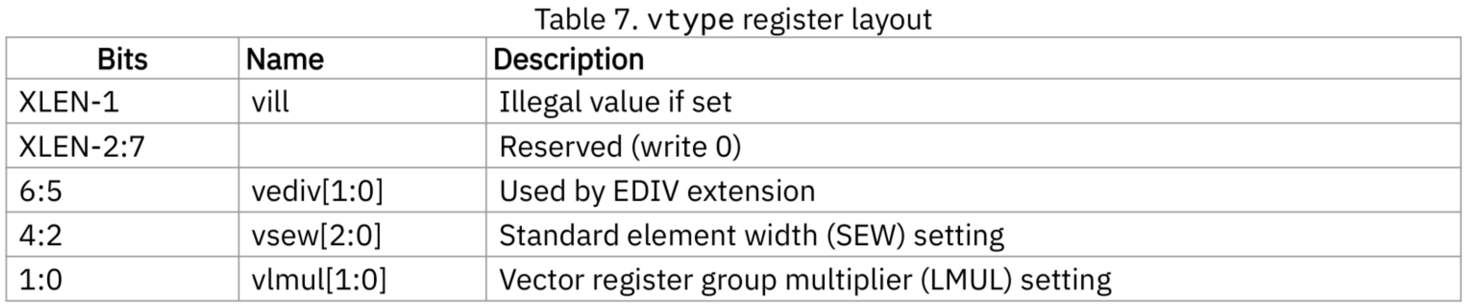


Table 7. vtype register layout

Bits	Name	Description
XLEN-1	vill	Illegal value if set
XLEN-2:7		Reserved (write 0)
6:5	vediv[1:0]	Used by EDIV extension
4:2	vsew[2:0]	Standard element width (SEW) setting
1:0	vlmul[1:0]	Vector register group multiplier (LMUL) setting

```
vsetvli t0, a0, e32,m2,d4
```

- RISC-V Vector 1.0 的 vtype 寄存器

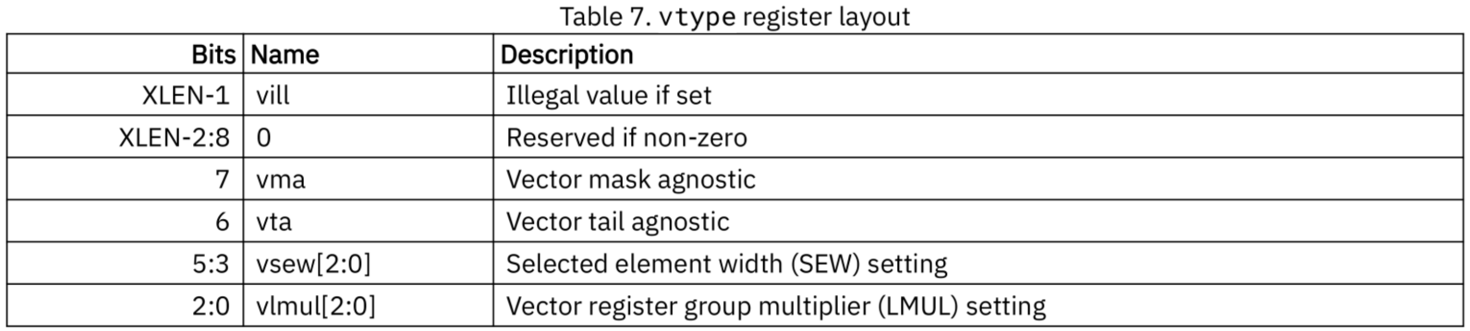
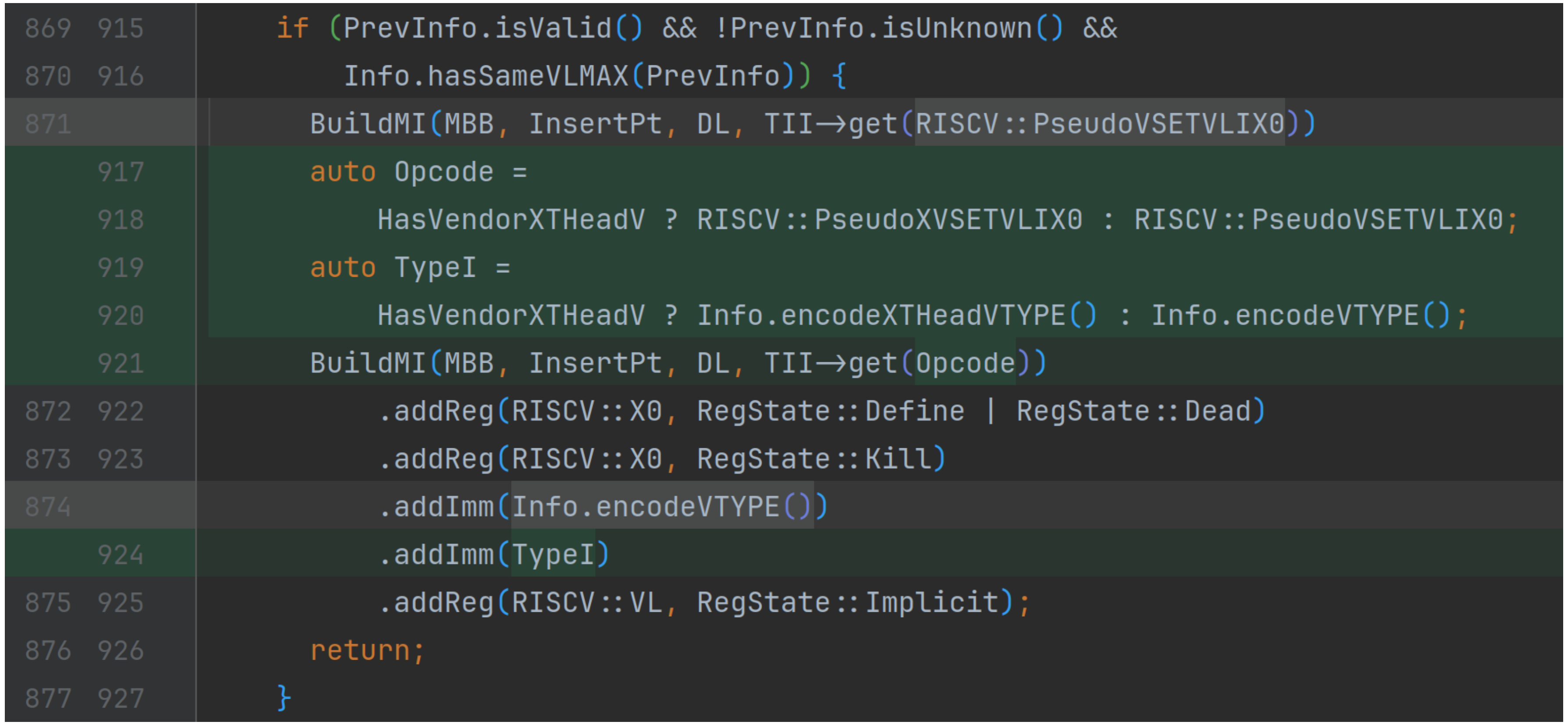


Table 7. vtype register layout

Bits	Name	Description
XLEN-1	vill	Illegal value if set
XLEN-2:8	0	Reserved if non-zero
7	vma	Vector mask agnostic
6	vta	Vector tail agnostic
5:3	vsew[2:0]	Selected element width (SEW) setting
2:0	vlmul[2:0]	Vector register group multiplier (LMUL) setting

```
vsetvli s1, zero, e8,mf2,ta,mu
```

Pass 调优： 以“插入 vsetvli” 为例



```
  if (PrevInfo.isValid() && !PrevInfo.isUnknown() &&
      Info.hasSameVLMAX(PrevInfo)) {
    BuildMI(MBB, InsertPt, DL, TII->get(RISCV::PseudoVSETVLIX0))
    auto Opcode =
        HasVendorXTHeadV ? RISCV::PseudoXVSETVLIX0 : RISCV::PseudoVSETVLIX0;
    auto TypeI =
        HasVendorXTHeadV ? Info.encodeXTHeadVTYPE() : Info.encodeVTYPE();
    BuildMI(MBB, InsertPt, DL, TII->get(Opcode))
        .addReg(RISCV::X0, RegState::Define | RegState::Dead)
        .addReg(RISCV::X0, RegState::Kill)
        .addImm(Info.encodeVTYPE())
        .addImm(TypeI)
        .addReg(RISCV::VL, RegState::Implicit);
    return;
  }
```

看着不怎么优雅，但确实复用了代码（大雾）

目前进度 & 未来计划

- 已经完成：全部 MC 汇编器相关

- 近期：支持全部 Intrinsic
- 未来：Auto Vectorization

- 关！注！我！们！
- https://github.com/ruyisdk/llvm-project

感 谢 聆 听

References

1. T-Head-Semi. (n.d.). *Vector implementation of THEAD*. GitHub. https://github.com/T-head-Semi/thead-extension-spec/blob/master/xtheadvector.adoc
2. LLVM Developers. (n.d.). *TableGen Overview*. LLVM Compiler Infrastructure. https://llvm.org/docs/TableGen/