



How to Add a Calling Convention RISC-V Vector as an Example

Brandon Wu <brandon.wu@sifive.com>

Kito Cheng <kito.cheng@sifive.com>

LLVM developers meeting, October, 2024



Outline

- **Brief Recap of Calling Convention**
- **RISC-V Vector ABI and Stack Layout**
- **Implementation Details**

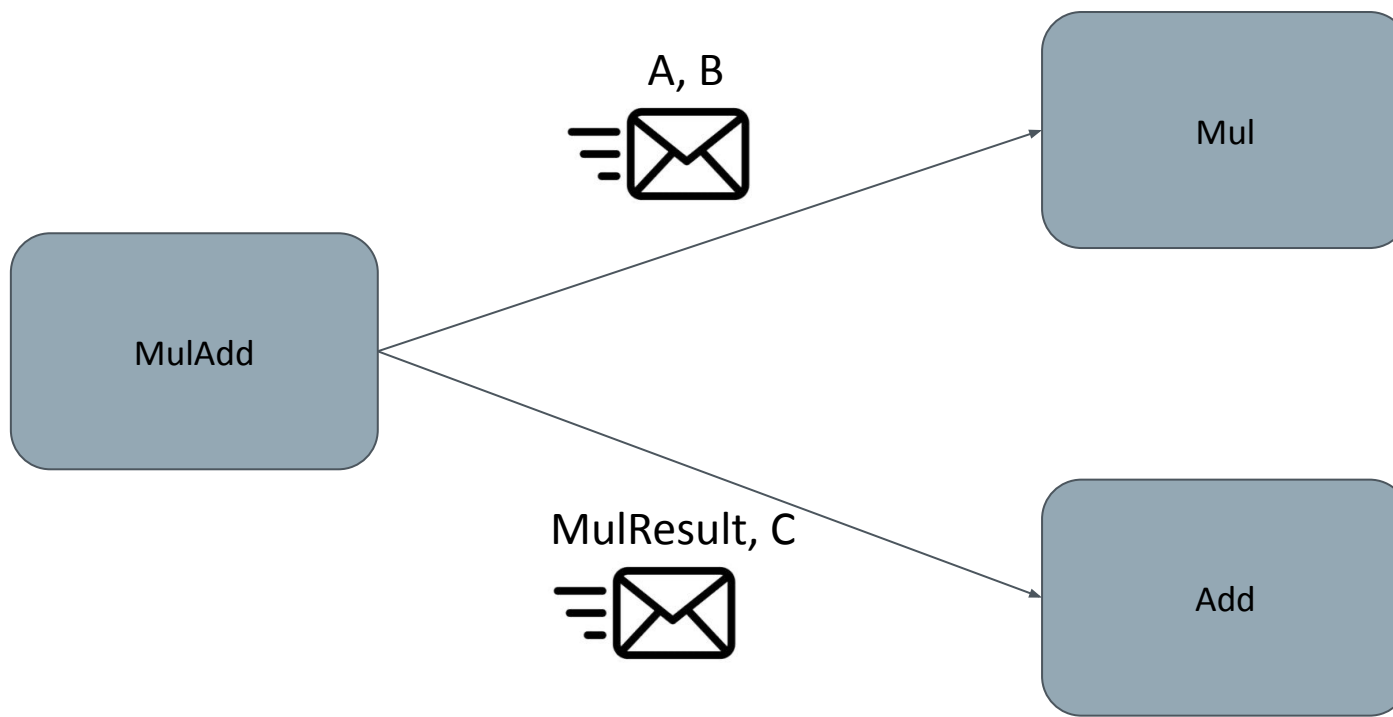


Brief Recap of Calling Convention



What's the Calling Convention?

- Calling Convention unifies the registers for **arguments** between caller and callee.
 - Registers for A and B need to be consistent between MulAdd and Mul.
 - Registers for MulResult and C need to be consistent between MulAdd and Add.

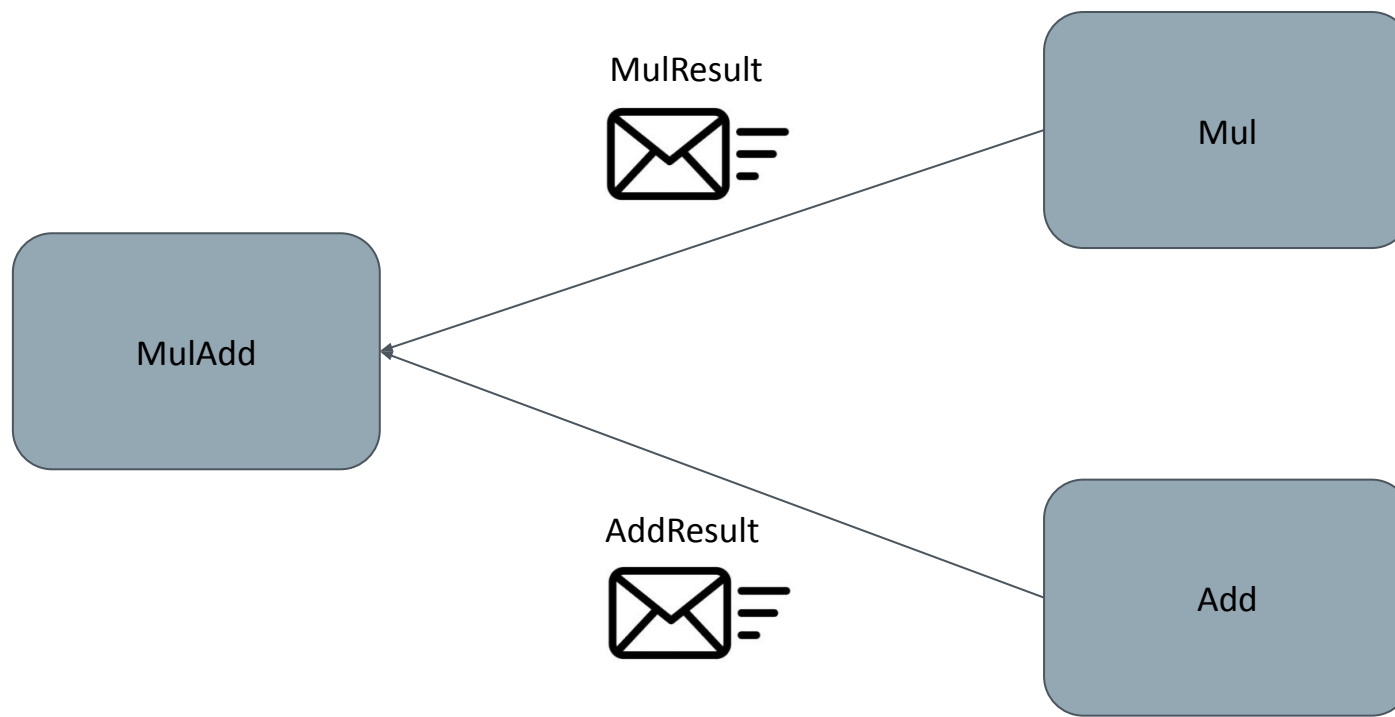


```
function Multiply(Input1, Input2);  
function Add(Input1, Input2);  
function MulAdd {  
    A = 2;  
    B = 3;  
    C = 4;  
    MulResult = call Multiply(A, B);  
    AddResult = call Add(MulResult, C);  
    return AddResult;  
}
```



What's the Calling Convention?

- Calling Convention unifies the registers for **return value** between caller and callee.
 - Registers for A and B need to be consistent between MulAdd and Mul.
 - Registers for MulResult and C need to be consistent between MulAdd and Add.



```
function Mul(Input1, Input2);  
function Add(Input1, Input2);  
function MulAdd {  
    A = 2;  
    B = 3;  
    C = 4;  
    MulResult = call Mul(A, B);  
    AddResult = call Add(MulResult, C);  
    return AddResult;  
}
```



What's the Calling Convention?

- **Calling Convention defines the responsibility of maintaining register states during the call.**
 - Register states are guaranteed to be consistent before and after the call.
 - Calling Convention specifies whether caller or callee is in charge of it.
 - e.g.

```
func {  
  def reg0  
  call callee()  
  use reg0  
}
```

who needs to keep reg0's state during the call?



RISC-V ABI and Stack Layout



RISC-V Scalar Registers

Integer Register Convention

Table 1. Integer register convention

Name	ABI Mnemonic	Meaning	Preserved across calls?
x0	zero	Zero	— (Immutable)
x1	ra	Return address	No
x2	sp	Stack pointer	Yes
x3	gp	Global pointer	— (Unallocatable)
x4	tp	Thread pointer	— (Unallocatable)
x5 - x7	t0 - t2	Temporary registers	No
x8 - x9	s0 - s1	Callee-saved registers	Yes
x10 - x17	a0 - a7	Argument registers	No
x18 - x27	s2 - s11	Callee-saved registers	Yes
x28 - x31	t3 - t6	Temporary registers	No

Ref:

<https://github.com/riscv-non-isa/riscv-elf-psabi-doc/blob/master/riscv-cc.adoc#integer-register-convention>



RISC-V Scalar Registers

Float point Register Convention

Table 2. Floating-point register convention

Name	ABI Mnemonic	Meaning	Preserved across calls?
f0 - f7	ft0 - ft7	Temporary registers	No
f8 - f9	fs0 - fs1	Callee-saved registers	Yes*
f10 - f17	fa0 - fa7	Argument registers	No
f18 - f27	fs2 - fs11	Callee-saved registers	Yes*
f28 - f31	ft8 - ft11	Temporary registers	No

Ref:

<https://github.com/riscv-non-isa/riscv-elf-psabi-doc/blob/master/riscv-cc.adoc#floating-point-register-convention>



RISC-V Vector Introduction

- RISC-V vector is a scalable vector.
- RISC-V has 32 vector registers, each of them is VLEN bits.
- VLENB CSR holds the vector length in bytes, e.g. 128 bits vector length -> VLENB = 16.
- vtype CSR and the corresponding instructions for configuration.



RISC-V Vector Registers

Standard calling convention v.s. Vector calling convention

Standard calling convention

Table 3. Standard vector register calling convention

Name	ABI Mnemonic	Meaning	Preserved across calls?
v0-v31		Temporary registers	No
vl		Vector length	No
vtype		Vector data type register	No
vxrm		Vector fixed-point rounding mode register	No
vxsat		Vector fixed-point saturation flag register	No

Calling convention variant

Table 4. Variant vector register calling convention*

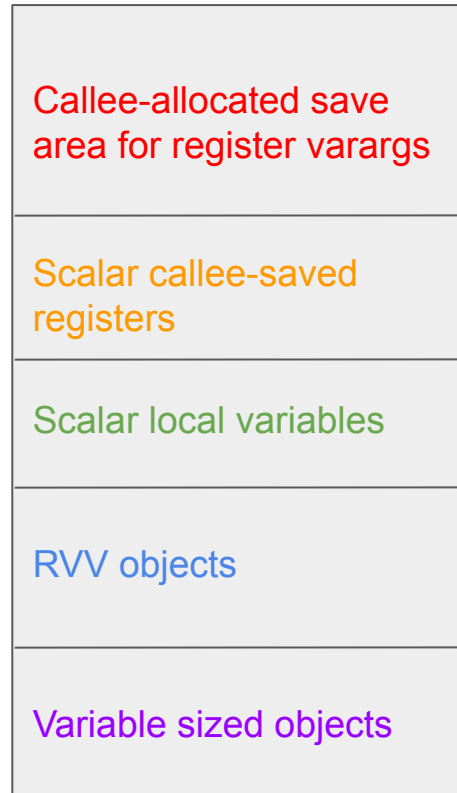
Name	ABI Mnemonic	Meaning	Preserved across calls?
v0		Argument register	No
v1-v7		Callee-saved registers	Yes
v8-v23		Argument registers	No
v24-v31		Callee-saved registers	Yes
vl		Vector length	No
vtype		Vector data type register	No
vxrm		Vector fixed-point rounding mode register	No
vxsat		Vector fixed-point saturation flag register	No

Ref:

<https://github.com/riscv-non-isa/riscv-elf-psa-bi-doc/blob/master/riscv-cc.adoc#vector-register-convention>



Stack Layout in RISC-V

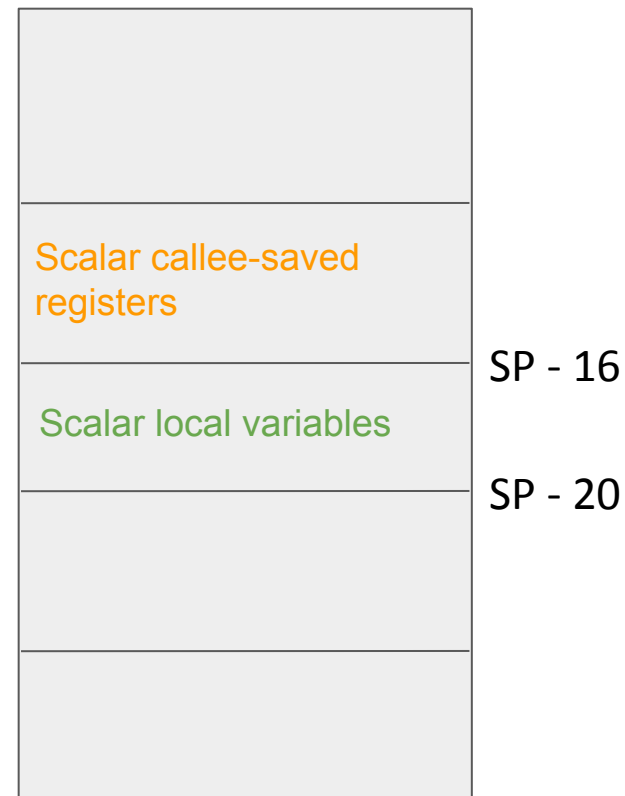




Stack Layout in RISC-V

```
void test() {  
    int a = 2024;  
}
```

```
test:  
    addi    sp, sp, -32  
    sd     ra, 24(sp)  
    sd     s0, 16(sp)  
    addi    s0, sp, 32  
    li     a0, 2024  
    sw     a0, -20(s0)  
    addi    sp, s0, -32  
    ld     ra, 24(sp)  
    ld     s0, 16(sp)  
    addi    sp, sp, 32  
    ret
```





Stack Layout in RISC-V

```
void test() {  
    int a = 2024;  
    vint32m1_t b;  
}
```

test:

```
addi    sp, sp, -32
```

```
sd      ra, 24(sp)
```

```
sd      s0, 16(sp)
```

```
addi    s0, sp, 32
```

```
csrr    a0, vlenb
```

```
slli    a0, a0, 1
```

```
sub     sp, sp, a0
```

```
li      a0, 2024
```

```
sw     a0, -20(s0)
```

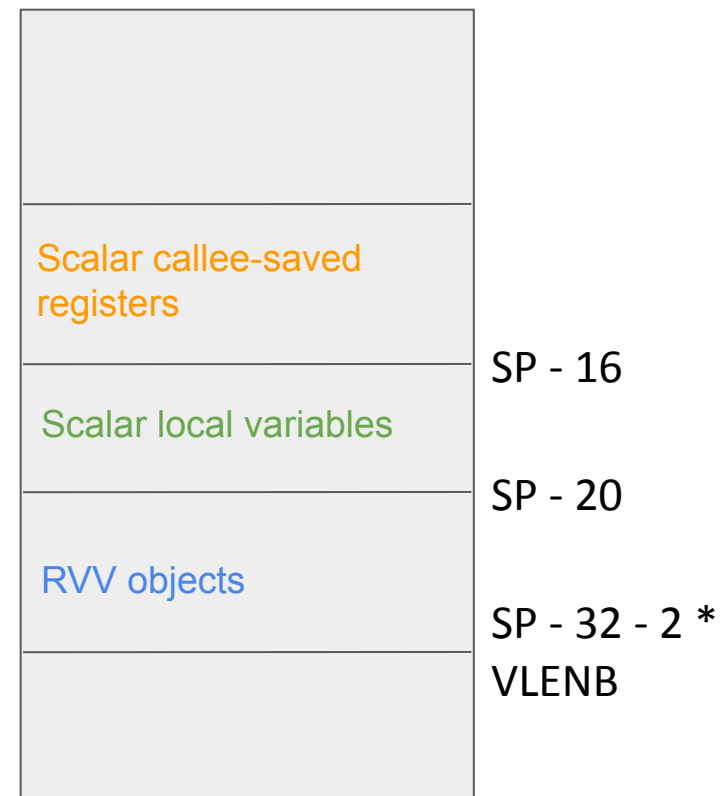
```
addi    sp, s0, -32
```

```
ld      ra, 24(sp)
```

```
ld      s0, 16(sp)
```

```
addi    sp, sp, 32
```

```
ret
```

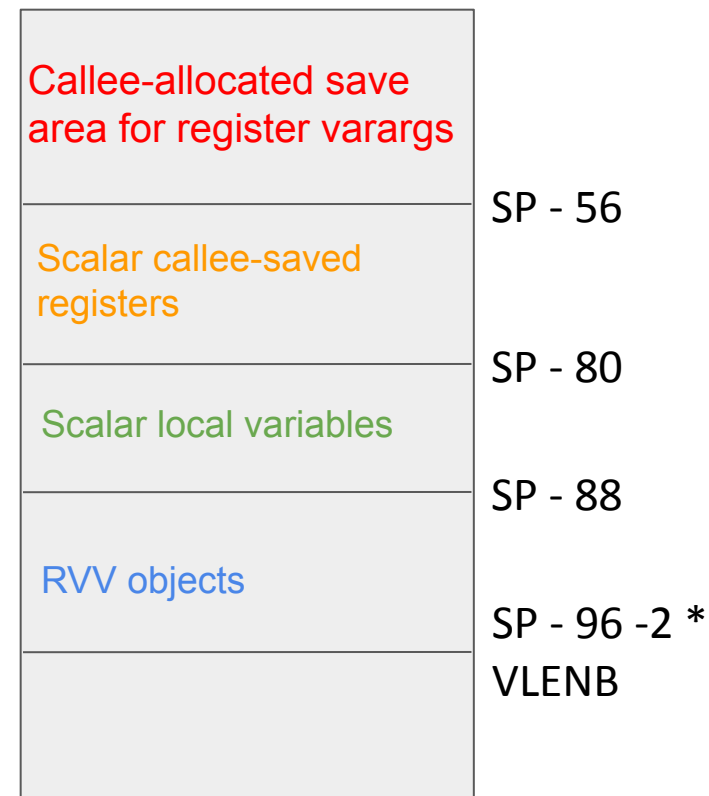




Stack Layout in RISC-V

```
void test(int
count, ...) {
    int a = 2024;
    vint32m1_t b;
}
```

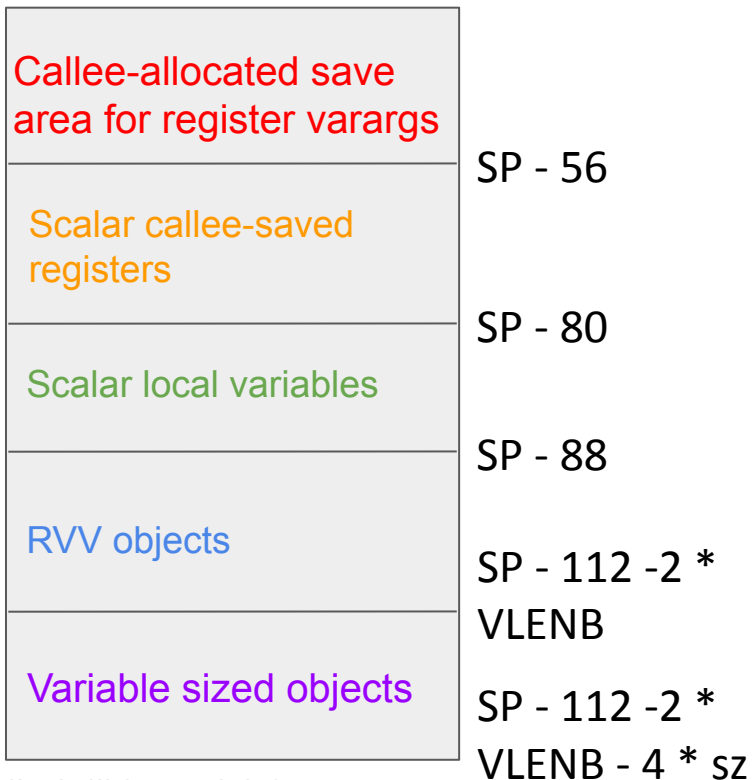
```
test:
    addi    sp, sp, -96
    sd     ra, 24(sp)
    sd     s0, 16(sp)
    addi    s0, sp, 32
    csrr    t0, vlenb
    slli    t0, t0, 1
    sub     sp, sp, t0
    sd     a7, 56(s0)
    sd     a6, 48(s0)
    sd     a5, 40(s0)
    sd     a4, 32(s0)
    sd     a3, 24(s0)
    sd     a2, 16(s0)
    sd     a1, 8(s0)
    sw     a0, -20(s0)
    li     a0, 2024
    sw     a0, -24(s0)
    addi    sp, s0, -32
    ld     ra, 24(sp)
    ld     s0, 16(sp)
    addi    sp, sp, 96
    ret
```





Stack Layout in RISC-V

```
extern int sz;
void test(int
count, ...) {
    int a = 2024;
    vint32m1_t b;
    int arr[sz];
}
```



```
test:
    addi    sp, sp, -112
    sd     ra, 40(sp)
    sd     s0, 32(sp)
    addi    s0, sp, 48
    csrr    t0, vlenb
    slli    t0, t0, 1
    sub     sp, sp, t0
    sd     a7, 56(s0)
    sd     a6, 48(s0)
    sd     a5, 40(s0)
    sd     a4, 32(s0)
    sd     a3, 24(s0)
    sd     a2, 16(s0)
    sd     a1, 8(s0)
    sw     a0, -20(s0)
    li     a0, 2024
    sw     a0, -24(s0)
```

```
lui     a0, %hi(sz)
lwu     a0, %lo(sz)(a0)
mv      a1, sp
sd      a1, -32(s0)
slli    a1, a0, 2
addi    a1, a1, 15
andi    a2, a1, -16
mv      a1, sp
sub     a1, a1, a2
mv      sp, a1
sd      a0, -40(s0)
ld      a0, -32(s0)
mv      sp, a0
addi    sp, s0, -48
ld      ra, 40(sp)
ld      s0, 32(sp)
addi    sp, sp, 112
ret
```




Implementation Details



Add a Function Attribute(Clang Side)

- clang/include/clang/Basic/Attr.td

```
def RISCVectorCC: DeclOrTypeAttr, TargetSpecificAttr<TargetRISCV> {  
  let Spellings = [CXX11<"riscv", "vector_cc">,  
                  C23<"riscv", "vector_cc">,  
                  Clang<"riscv_vector_cc">];  
  let Documentation = [RISCVectorCCDocs];  
}
```

- clang/include/clang/Basic/Specifiers.h

```
enum CallingConv {  
  CC_C, // __attribute__((cdecl))  
  CC_X86StdCall, // __attribute__((stdcall))  
  .,  
  .,  
  .,  
  CC_RISCVectorCall, // __attribute__((riscv_vector_cc))  
};
```



Add a Function Attribute (Clang Side)

- clang/lib/CodeGen/CGCall.cpp

```
unsigned CodeGenTypes::ClangCallConvToLLVMCallConv(CallingConv CC) {  
    switch (CC) {  
        default: return llvm::CallingConv::C;  
        case CC_X86StdCall: return llvm::CallingConv::X86_StdCall;  
        .  
        .  
        .  
        case CC_RISCVVectorCall: return llvm::CallingConv::RISCV_VectorCall;  
    }  
}
```



Add a Function Attribute(Clang Side)

- llvm/include/llvm/IR/CallingConv.h

```
namespace CallingConv {  
  
  /// LLVM IR allows to use arbitrary numbers as calling convention identifiers.  
  using ID = unsigned;  
  
  /// A set of enums which specify the assigned numeric values for known llvm  
  /// calling conventions.  
  /// LLVM Calling Convention Representation  
  enum {  
    /// The default llvm calling convention, compatible with C. This convention  
    /// is the only one that supports varargs calls. As with typical C calling  
    /// conventions, the callee/caller have to tolerate certain amounts of  
    /// prototype mismatch.  
    C = 0,  
  
    /// stdcall is mostly used by the Win32 API. It is basically the same as the  
    /// C convention with the difference in that the callee is responsible for  
    /// popping the arguments from the stack.  
    X86_StdCall = 64,  
    .  
    .  
    .  
    /// Calling convention used for RISC-V V-extension.  
    RISCV_VectorCall = 110,  
  
    /// The highest possible ID. Must be some 2^k - 1.  
    MaxID = 1023  
  };  
} // end namespace CallingConv
```



Add a Function Attribute(Clang Side)

- llvm/include/llvm/AsmParser/LLToken.h

```
namespace lltok {
enum Kind {
    .
    .
    .
    kw_cc,
    kw_x86_stdcallcc,
    kw_riscv_vector_cc,
    .
    .
    .
};
} // end namespace lltok
```

- llvm/lib/AsmParser/LLParser.cpp

```
bool LLParser::parseOptionalCallingConv(unsigned &CC) {
    switch (Lex.getKind()) {
    default:          CC = CallingConv::C; return false;
    case lltok::kw_x86_stdcallcc: CC = CallingConv::X86_StdCall; break;
    .
    .
    .
    case lltok::kw_riscv_vector_cc:
        CC = CallingConv::RISCV_VectorCall;
        break;
    }
    Lex.Lex();
    return false;
}
```



Add a Function Attribute (Clang Side)

- llvm/lib/IR/AsmWriter.cpp

```
static void PrintCallingConv(unsigned cc, raw_ostream &Out) {  
    switch (cc) {  
    default:          Out << "cc" << cc; break;  
    case CallingConv::X86_StdCall:  Out << "x86_stdcallcc"; break;  
    .  
    .  
    .  
    case CallingConv::RISCV_VectorCall:  
        Out << "riscv_vector_cc";  
        break;  
    }  
}
```



Add a Function Attribute (Clang Side)

- test.c

```
vint32m1_t __attribute__((riscv_vector_cc)) bar(vint32m1_t input);  
// [[riscv::vector_cc]] vint32m1_t bar(vint32m1_t input);  
vint32m1_t test_vector_cc_attr(vint32m1_t input, int32_t *base, size_t vl) {  
    vint32m1_t val = __riscv_vle32_v_i32m1(base, vl);  
    vint32m1_t ret = bar(input);  
    __riscv_vse32_v_i32m1(base, val, vl);  
    return ret;  
}
```

- clang -target riscv64 -S -emit-llvm test.c

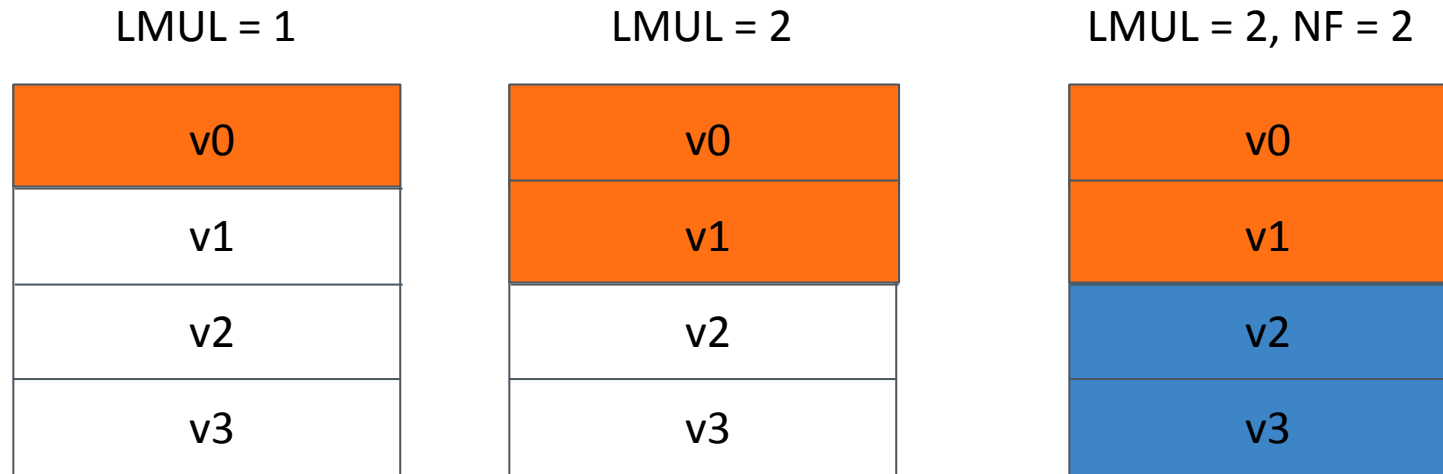
```
declare riscv_vector_cc <vscale x 2 x i32> @bar(<vscale x 2 x i32>)  
  
define <vscale x 2 x i32> @test_vector_cc_attr(<vscale x 2 x i32> %input, ptr %base, i64 %vl) {  
entry:  
    %0 = tail call <vscale x 2 x i32> @llvm.riscv.vle.nxv2i32.i64(<vscale x 2 x i32> poison, ptr %base, i64 %vl)  
    %call = tail call riscv_vector_cc <vscale x 2 x i32> @bar(<vscale x 2 x i32> %input)  
    tail call void @llvm.riscv.vse.nxv2i32.i64(<vscale x 2 x i32> %0, ptr %base, i64 %vl)  
    ret <vscale x 2 x i32> %call  
}
```



Handling Vector Type Arguments

- **RISC-V Vector has 2 concepts**






- LMUL: Vector register group multiplier.
- NF: Number of Fields(vector register group) in a segment(This is used for segment load/store)
- e.g.



- **The type for representing NF is called “Vector Tuple Type”.**



Handling Vector Type Arguments

- **Constraints for RVV(normal vector) Type:**
 - 1. Starting register should be multiple of LMUL.
 - 2. All LMUL registers should be consecutive.
 - example1:  vint64m2 -> V2,V3
 - example2:  vint64m2 -> V1,V2
 - example3:  vint64m4 -> V4,V6,V8,V24
- **Constraints for Vector Tuple Type:**
 - 1. All of constraints in RVV type.
 - 2. All NF register group should be consecutive.
 - example4:  vint64m2x2 -> V2,V3,V4,V5
 - example5:  vint64m2x2 -> V2,V3,V8,V9



Handling Vector Type Arguments

- **RVV type can be model using “vscale” in llvm**
 - Bits per block = 64
 - `vint64m1` -> `<vscale x 1 x i64>`
- **In RISC-V, if we use the same approach to model tuple type, it would be ambiguous.**
 - `vint64m2x4` -> `<vscale x 8 x i64>???`
 - `vint64m4x2` -> `<vscale x 8 x i64>???`
 - So we need another type to model the vector tuple type correctly.
- **AArch64 has NF but not LMUL, the AArch64 vector tuple type can be represented using existing scalable vector.**
 - Bits per block = 128
 - `svint64_t` -> `<vscale x 2 x i64>`
 - `svint64x2_t` -> `<vscale x 4 x i64>`



Handling Vector Type Arguments

- Previously we used “struct” for tuple type, but it would be flattened during selection dag construction, so the “NF” information is gone.
 - `vint64m2x4` ->

`{<vscale x 2 x 64>, <vscale x 2 x 64>, <vscale x 2 x 64>, <vscale x 2 x 64>} ->`

`MVT::nxv2i64, MVT::nxv2i64, MVT::nxv2i64, MVT::nxv2i64`
- **RISC-V models the vector tuple type in LLVM IR by using TargetExtType.**
 - `vint64m2x4` -> `target("riscv.vector.tuple", <vscale x 16 x i8>, 4)`
 - The type argument represents the LMUL with regularized element type i8.
 - The integer argument represents the NF.



Handling Vector Type Arguments

- **MVT(Machine Value Type) for RISC-V Vector**
 - RVV: `<vscale x 2 x i64>` -> `MVT::nxv2i64`
 - Vector Tuple Type: `target("riscv.vector.tuple", <vscale x 16 x i8>, 4)` -> `MVT::riscv_nxv16i8x4`
- **Machine Register class for RISC-V Vector**
 - RVV: `<vscale x 2 x i64>` -> `VRM2`
 - Registers in `VRM2` class: `V0M2, V2M2, V4M2, V6M2... V30M2`
 - Vector Tuple Type: `target("riscv.vector.tuple", <vscale x 16 x i8>, 4)` -> `VRN4M2`
 - Registers in `VRN4M2` class: `V0M2_V2M2_V4M2_V6M2, V2M2_V4M2_V6M2_V8M2... V24M2_V26M2_V28M2_V30M2`



Caller saved and Callee saved Registers

- llvm/lib/Target/RISCV/RISCVCallingConv.td

```
defvar CSR_V = (add (sequence "V%u", 1, 7), (sequence "V%u", 24, 31),
                   V2M2, V4M2, V6M2, V24M2, V26M2, V28M2, V30M2,
                   V4M4, V24M4, V28M4, V24M8);

def CSR_ILP32_LP64_V
  : CalleeSavedRegs<(add CSR_ILP32_LP64, CSR_V)>;

def CSR_ILP32F_LP64F_V
  : CalleeSavedRegs<(add CSR_ILP32F_LP64F, CSR_V)>;

def CSR_ILP32D_LP64D_V
  : CalleeSavedRegs<(add CSR_ILP32D_LP64D, CSR_V)>;
```



Caller saved and Callee saved Registers

- llvm/lib/Target/RISCV/RISCVRegisterInfo.cpp

```
const MPhysReg *
RISCVRegisterInfo::getCalleeSavedRegs(const MachineFunction *MF) const {
    .
    .
    .
    bool HasVectorCSR =
        MF->getFunction().getCallingConv() == CallingConv::RISCV_VectorCall &&
        Subtarget.hasVInstructions();

    switch (Subtarget.getTargetABI()) {
    default:
        llvm_unreachable("Unrecognized ABI");
    case RISCVABI::ABI_ILP32:
    case RISCVABI::ABI_LP64:
        if (HasVectorCSR)
            return CSR_ILP32_LP64_V_SaveList;
        return CSR_ILP32_LP64_SaveList;
    case RISCVABI::ABI_ILP32F:
    case RISCVABI::ABI_LP64F:
        if (HasVectorCSR)
            return CSR_ILP32F_LP64F_V_SaveList;
        return CSR_ILP32F_LP64F_SaveList;
    case RISCVABI::ABI_ILP32D:
    case RISCVABI::ABI_LP64D:
        if (HasVectorCSR)
            return CSR_ILP32D_LP64D_V_SaveList;
        return CSR_ILP32D_LP64D_SaveList;
    }
}
```



Caller saved and Callee saved Registers

- `llvm/lib/Target/RISCV/RISCVFrameLowering.cpp`
 - Utility function to get all of RVV callee saved information.

```
static SmallVector<CalleeSavedInfo, 8>
getRVVCalleeSavedInfo(const MachineFunction &MF,
                      const std::vector<CalleeSavedInfo> &CSI) {
    const MachineFrameInfo &MFI = MF.getFrameInfo();
    SmallVector<CalleeSavedInfo, 8> RVVCSI;

    for (auto &CS : CSI) {
        int FI = CS.getFrameIdx();
        if (FI >= 0 && MFI.getStackID(FI) == TargetStackID::ScalableVector)
            RVVCSI.push_back(CS);
    }

    return RVVCSI;
}
```



Caller saved and Callee saved Registers

- llvm/lib/Target/RISCV/RISCVFrameLowering.cpp

```
std::pair<int64_t, Align>
RISCVFrameLowering::assignRVVStackObjectOffsets(MachineFunction &MF) const {
    .
    .
    .
    SmallVector<int, 8> ObjectsToAllocate;
    auto pushRVVObjects = [&](int FIBegin, int FIEnd) {
        for (int I = FIBegin, E = FIEnd; I != E; ++I) {
            unsigned StackID = MFI.getStackID(I);
            if (StackID != TargetStackID::ScalableVector)
                continue;
            if (MFI.isDeadObjectIndex(I))
                continue;

            ObjectsToAllocate.push_back(I);
        }
    };
    // First push RVV Callee Saved object, then push RVV stack object
    std::vector<CalleeSavedInfo> &CSI = MF.getFrameInfo().getCalleeSavedInfo();
    const auto &RVVCSI = getRVVCalleeSavedInfo(MF, CSI);
    if (!RVVCSI.empty())
        pushRVVObjects(RVVCSI[0].getFrameIdx(),
                      RVVCSI[RVVCSI.size() - 1].getFrameIdx() + 1);
    .
    .
    .
}
```

```
.
.
.
// Allocate all RVV locals and spills
int64_t Offset = 0;
for (int FI : ObjectsToAllocate) {
    // ObjectSize in bytes.
    int64_t ObjectSize = MFI.getObjectSize(FI);
    auto ObjectAlign = std::max(Align(8), MFI.getObjectAlign(FI));
    // If the data type is the fractional vector type, reserve one vector
    // register for it.
    if (ObjectSize < 8)
        ObjectSize = 8;
    Offset = alignTo(Offset + ObjectSize, ObjectAlign);
    MFI.setObjectOffset(FI, -Offset);
    // Update the maximum alignment of the RVV stack section
    RVVStackAlign = std::max(RVVStackAlign, ObjectAlign);
}
.
.
.
}
```




Caller saved and Callee saved Registers

- llvm/lib/Target/RISCV/RISCVFrameLowering.cpp
 - Emit spills for RVV.

```
bool RISCVFrameLowering::spillCalleeSavedRegisters(
    MachineBasicBlock &MBB, MachineBasicBlock::iterator MI,
    ArrayRef<CalleeSavedInfo> CSI, const TargetRegisterInfo *TRI) const {
    .
    .
    .
    const auto &RVVCSI = getRVVCalleeSavedInfo(*MF, CSI);

    auto storeRegToStackSlot = [&](decltype(UnmanagedCSI) CSInfo) {
        for (auto &CS : CSInfo) {
            // Insert the spill to the stack frame.
            Register Reg = CS.getReg();
            const TargetRegisterClass *RC = TRI->getMinimalPhysRegClass(Reg);
            TII.storeRegToStackSlot(MBB, MI, Reg, !MBB.isLiveIn(Reg),
                CS.getFrameIdx(), RC, TRI, Register());
        }
    };

    storeRegToStackSlot(RVVCSI);

    return true;
}
```



Caller saved and Callee saved Registers

- llvm/lib/Target/RISCV/RISCVFrameLowering.cpp
 - Emit reloads for RVV.

```
bool RISCVFrameLowering::restoreCalleeSavedRegisters(
    MachineBasicBlock &MBB, MachineBasicBlock::iterator MI,
    MutableArrayRef<CalleeSavedInfo> CSI, const TargetRegisterInfo *TRI) const {
    .
    .
    .
    const auto &RVVCSI = getRVVCalleeSavedInfo(*MF, CSI);

    auto loadRegFromStackSlot = [&](decltype(UnmanagedCSI) CSInfo) {
        for (auto &CS : CSInfo) {
            Register Reg = CS.getReg();
            const TargetRegisterClass *RC = TRI->getMinimalPhysRegClass(Reg);
            TII.loadRegFromStackSlot(MBB, MI, Reg, CS.getFrameIdx(), RC, TRI,
                Register());
            assert(MI != MBB.begin() &&
                "loadRegFromStackSlot didn't insert any code!");
        }
    };
    loadRegFromStackSlot(RVVCSI);
    .
    .
    .
    return true;
}
```



Caller saved and Callee saved Registers

- Test without RVV calling convention
 - v1 is not callee-saved register, v8 is not callee-saved register

```
define <vscale x 1 x i32> @test_vector_std(<vscale x 1 x i32> %va) nounwind {  
; SPILL-LABEL: test_vector_std:  
; SPILL:      # %bb.0: # %entry  
; SPILL-NEXT:  vmv1r.v v9, v8  
; SPILL-NEXT:  #APP  
; SPILL-NEXT:  #NO_APP  
; SPILL-NEXT:  vmv1r.v v8, v9  
; SPILL-NEXT:  ret  
entry:  
  call void asm sideeffect "", "~{v1},~{v8}"()  
  
  ret <vscale x 1 x i32> %va  
}
```



Caller saved and Callee saved Registers

- Test with RVV calling convention
 - v1 is not callee-saved register, v8 is callee-saved register

```
define riscv_vector_cc <vscale x 1 x i32> @test_vector_callee(<vscale x 1 x i32> %va) nounwind {
; SPILL-LABEL: test_vector_callee:
; SPILL:      # %bb.0: # %entry
; SPILL-NEXT:  addi sp, sp, -16
; SPILL-NEXT:  csrr a0, vlenb
; SPILL-NEXT:  slli a0, a0, 1
; SPILL-NEXT:  sub sp, sp, a0
; SPILL-NEXT:  addi a0, sp, 16
; SPILL-NEXT:  vs1r.v v1, (a0) # Unknown-size Folded Spill
; SPILL-NEXT:  vmv1r.v v9, v8
; SPILL-NEXT:  #APP
; SPILL-NEXT:  #NO_APP
; SPILL-NEXT:  vmv1r.v v8, v9
; SPILL-NEXT:  vl1r.v v1, (a0) # Unknown-size Folded Reload
; SPILL-NEXT:  csrr a0, vlenb
; SPILL-NEXT:  slli a0, a0, 1
; SPILL-NEXT:  add sp, sp, a0
; SPILL-NEXT:  addi sp, sp, 16
; SPILL-NEXT:  ret
entry:
  call void @asm_sideeffect "", "~{v1},~{v8}"()

  ret <vscale x 1 x i32> %va
}
```



Handling Vector Type Arguments

- **llvm/lib/Target/RISCV/RISCVISelLowering.cpp**
 - Need to implement the register assign interface RISCVCCToAssignFn for target hooks
 - RISCVTargetLowering::LowerFormalArgument
 - RISCVTargetLowering::LowerReturn
 - RISCVTargetLowering::LowerCall
 - The RISCVCCToAssignFn for RISC-V Vector CC is called CC_RISCV



Handling Vector Type Arguments

- RISC-V Vector CC uses V8-V23 for passing arguments.
 - RVV argument registers:

```
static const MCPPhysReg ArgVRs[] = {
    RISCV::V8, RISCV::V9, RISCV::V10, RISCV::V11, RISCV::V12, RISCV::V13,
    RISCV::V14, RISCV::V15, RISCV::V16, RISCV::V17, RISCV::V18, RISCV::V19,
    RISCV::V20, RISCV::V21, RISCV::V22, RISCV::V23};
static const MCPPhysReg ArgVRM2s[] = {RISCV::V8M2, RISCV::V10M2, RISCV::V12M2,
    RISCV::V14M2, RISCV::V16M2, RISCV::V18M2,
    RISCV::V20M2, RISCV::V22M2};
static const MCPPhysReg ArgVRM4s[] = {RISCV::V8M4, RISCV::V12M4, RISCV::V16M4,
    RISCV::V20M4};
static const MCPPhysReg ArgVRM8s[] = {RISCV::V8M8, RISCV::V16M8};
```




Handling Vector Type Arguments

- Vector Tuple Type Registers:

```
static const MPhysReg ArgVRN2M1s[] = {
    RISCV::V8_V9,    RISCV::V9_V10,    RISCV::V10_V11,  RISCV::V11_V12,
    RISCV::V12_V13, RISCV::V13_V14,    RISCV::V14_V15,  RISCV::V15_V16,
    RISCV::V16_V17, RISCV::V17_V18,    RISCV::V18_V19,  RISCV::V19_V20,
    RISCV::V20_V21, RISCV::V21_V22,    RISCV::V22_V23};
static const MPhysReg ArgVRN3M1s[] = {
    RISCV::V8_V9_V10,    RISCV::V9_V10_V11,    RISCV::V10_V11_V12,
    RISCV::V11_V12_V13, RISCV::V12_V13_V14,    RISCV::V13_V14_V15,
    RISCV::V14_V15_V16, RISCV::V15_V16_V17,    RISCV::V16_V17_V18,
    RISCV::V17_V18_V19, RISCV::V18_V19_V20,    RISCV::V19_V20_V21,
    RISCV::V20_V21_V22, RISCV::V21_V22_V23};
static const MPhysReg ArgVRN4M1s[] = {
    RISCV::V8_V9_V10_V11,    RISCV::V9_V10_V11_V12,    RISCV::V10_V11_V12_V13,
    RISCV::V11_V12_V13_V14, RISCV::V12_V13_V14_V15,    RISCV::V13_V14_V15_V16,
    RISCV::V14_V15_V16_V17, RISCV::V15_V16_V17_V18,    RISCV::V16_V17_V18_V19,
    RISCV::V17_V18_V19_V20, RISCV::V18_V19_V20_V21,    RISCV::V19_V20_V21_V22,
    RISCV::V20_V21_V22_V23};
static const MPhysReg ArgVRN5M1s[] = {
    RISCV::V8_V9_V10_V11_V12,    RISCV::V9_V10_V11_V12_V13,
    RISCV::V10_V11_V12_V13_V14, RISCV::V11_V12_V13_V14_V15,
    RISCV::V12_V13_V14_V15_V16, RISCV::V13_V14_V15_V16_V17,
    RISCV::V14_V15_V16_V17_V18, RISCV::V15_V16_V17_V18_V19,
    RISCV::V16_V17_V18_V19_V20, RISCV::V17_V18_V19_V20_V21,
    RISCV::V18_V19_V20_V21_V22, RISCV::V19_V20_V21_V22_V23};
static const MPhysReg ArgVRN6M1s[] = {
    RISCV::V8_V9_V10_V11_V12_V13,    RISCV::V9_V10_V11_V12_V13_V14,
    RISCV::V10_V11_V12_V13_V14_V15, RISCV::V11_V12_V13_V14_V15_V16,
    RISCV::V12_V13_V14_V15_V16_V17, RISCV::V13_V14_V15_V16_V17_V18,
    RISCV::V14_V15_V16_V17_V18_V19, RISCV::V15_V16_V17_V18_V19_V20,
    RISCV::V16_V17_V18_V19_V20_V21, RISCV::V17_V18_V19_V20_V21_V22,
    RISCV::V18_V19_V20_V21_V22_V23};
```



Handling Vector Type Arguments

- Vector Tuple Type Registers:

```
static const MCPPhysReg ArgVRN7M1s[] = {
    RISCV::V8_V9_V10_V11_V12_V13_V14,    RISCV::V9_V10_V11_V12_V13_V14_V15,
    RISCv::V10_V11_V12_V13_V14_V15_V16,  RISCv::V11_V12_V13_V14_V15_V16_V17,
    RISCv::V12_V13_V14_V15_V16_V17_V18,  RISCv::V13_V14_V15_V16_V17_V18_V19,
    RISCv::V14_V15_V16_V17_V18_V19_V20,  RISCv::V15_V16_V17_V18_V19_V20_V21,
    RISCv::V16_V17_V18_V19_V20_V21_V22,  RISCv::V17_V18_V19_V20_V21_V22_V23};
static const MCPPhysReg ArgVRN8M1s[] = {RISCv::V8_V9_V10_V11_V12_V13_V14_V15,
    RISCv::V9_V10_V11_V12_V13_V14_V15_V16,
    RISCv::V10_V11_V12_V13_V14_V15_V16_V17,
    RISCv::V11_V12_V13_V14_V15_V16_V17_V18,
    RISCv::V12_V13_V14_V15_V16_V17_V18_V19,
    RISCv::V13_V14_V15_V16_V17_V18_V19_V20,
    RISCv::V14_V15_V16_V17_V18_V19_V20_V21,
    RISCv::V15_V16_V17_V18_V19_V20_V21_V22,
    RISCv::V16_V17_V18_V19_V20_V21_V22_V23};
static const MCPPhysReg ArgVRN2M2s[] = {RISCv::V8M2_V10M2,    RISCv::V10M2_V12M2,
    RISCv::V12M2_V14M2,    RISCv::V14M2_V16M2,
    RISCv::V16M2_V18M2,    RISCv::V18M2_V20M2,
    RISCv::V20M2_V22M2};
static const MCPPhysReg ArgVRN3M2s[] = {
    RISCv::V8M2_V10M2_V12M2,    RISCv::V10M2_V12M2_V14M2,
    RISCv::V12M2_V14M2_V16M2,    RISCv::V14M2_V16M2_V18M2,
    RISCv::V16M2_V18M2_V20M2,    RISCv::V18M2_V20M2_V22M2};
static const MCPPhysReg ArgVRN4M2s[] = {
    RISCv::V8M2_V10M2_V12M2_V14M2,    RISCv::V10M2_V12M2_V14M2_V16M2,
    RISCv::V12M2_V14M2_V16M2_V18M2,    RISCv::V14M2_V16M2_V18M2_V20M2,
    RISCv::V16M2_V18M2_V20M2_V22M2};
static const MCPPhysReg ArgVRN2M4s[] = {RISCv::V8M4_V12M4,    RISCv::V12M4_V16M4,
    RISCv::V16M4_V20M4};
```




Handling Vector Type Arguments

- **Run out of registers?**
 - The arguments that don't have enough register to pass would be passed by reference.
 - Store the value to the stack and pass the stack address through GPRs.



Handling Vector Type Arguments

- Test RVV

```
define <vscale x 8 x i64> @call_lmulo8_add(<vscale x 8 x i64> %x, <vscale x 8 x i64> %y) {  
; CHECK-LABEL: call_lmulo8_add:  
; CHECK:      # %bb.0:  
; CHECK-NEXT:  vsetvli a0, zero, e64, m8, ta, ma  
; CHECK-NEXT:  vadd.vv v8, v8, v16  
; CHECK-NEXT:  ret  
%a = add <vscale x 8 x i64> %x, %y  
ret <vscale x 8 x i64> %a  
}
```

- Test RVV out of registers

```
define <vscale x 8 x i64> @call_lmulo8_add_out_of_regs(<vscale x 8 x i64> %x1, <vscale x 8 x i64> %y1,  
                                                    <vscale x 8 x i64> %x2, <vscale x 8 x i64> %y2) {  
; CHECK-LABEL: call_lmulo8_add_out_of_regs:  
; CHECK:      # %bb.0:  
; CHECK-NEXT:  vl8re64.v v24, (a1)  
; CHECK-NEXT:  vl8re64.v v0, (a0)  
; CHECK-NEXT:  vsetvli a0, zero, e64, m8, ta, ma  
; CHECK-NEXT:  vadd.vv v8, v8, v16  
; CHECK-NEXT:  vadd.vv v16, v0, v24  
; CHECK-NEXT:  vadd.vv v8, v8, v16  
; CHECK-NEXT:  ret  
%a = add <vscale x 8 x i64> %x1, %y1  
%b = add <vscale x 8 x i64> %x2, %y2  
%c = add <vscale x 8 x i64> %a, %b  
ret <vscale x 8 x i64> %c  
}
```



Handling Vector Type arguments

- Test Vector Tuple Type

```
define void @test_vector_tuple(target("riscv.vector.tuple", <vscale x 32 x i8>, 2) %val, ptr %base, i64 %vl) {  
; CHECK-LABEL: test_vector_tuple:  
; CHECK:      # %bb.0: # %entry  
; CHECK-NEXT: vsetvli zero, a1, e8, m4, ta, ma  
; CHECK-NEXT: vsseg2e8.v v8, (a0)  
; CHECK-NEXT: ret  
entry:  
  tail call void @llvm.riscv.vsseg2.triscv.vector.tuple_nxv32i8_2t(  
    target("riscv.vector.tuple", <vscale x 32 x i8>, 2) %val, ptr %base, i64 %vl, i64 3)  
  ret void  
}
```



Handling Vector Type arguments

- Test Vector Tuple Type out of registers

```
define void @test_vector_tuple_out_of_registers(target("riscv.vector.tuple", <vscale x 32 x i8>, 2) %val1,  
                                               target("riscv.vector.tuple", <vscale x 32 x i8>, 2) %val2,  
                                               target("riscv.vector.tuple", <vscale x 32 x i8>, 2) %val3,  
                                               ptr %base1, ptr %base2, ptr %base3, i64 %vl) {  
; CHECK-LABEL: test_vector_tuple_out_of_registers:  
; CHECK:      # %bb.0: # %entry  
; CHECK-NEXT: vl8r.v v24, (a0)  
; CHECK-NEXT: vsetvli zero, a4, e8, m4, ta, ma  
; CHECK-NEXT: vsseg2e8.v v8, (a1)  
; CHECK-NEXT: vsseg2e8.v v16, (a2)  
; CHECK-NEXT: vsseg2e8.v v24, (a3)  
; CHECK-NEXT: ret  
entry:  
  tail call void @llvm.riscv.vsseg2.triscv.vector.tuple_nxv32i8_2t(  
    target("riscv.vector.tuple", <vscale x 32 x i8>, 2) %val1, ptr %base1, i64 %vl, i64 3)  
  tail call void @llvm.riscv.vsseg2.triscv.vector.tuple_nxv32i8_2t(  
    target("riscv.vector.tuple", <vscale x 32 x i8>, 2) %val2, ptr %base2, i64 %vl, i64 3)  
  tail call void @llvm.riscv.vsseg2.triscv.vector.tuple_nxv32i8_2t(  
    target("riscv.vector.tuple", <vscale x 32 x i8>, 2) %val3, ptr %base3, i64 %vl, i64 3)  
  ret void  
}
```



Thank you for your attention!!!
