How to add C intrinsic and code-gen it, using the RISC-V vector C intrinsics as an example

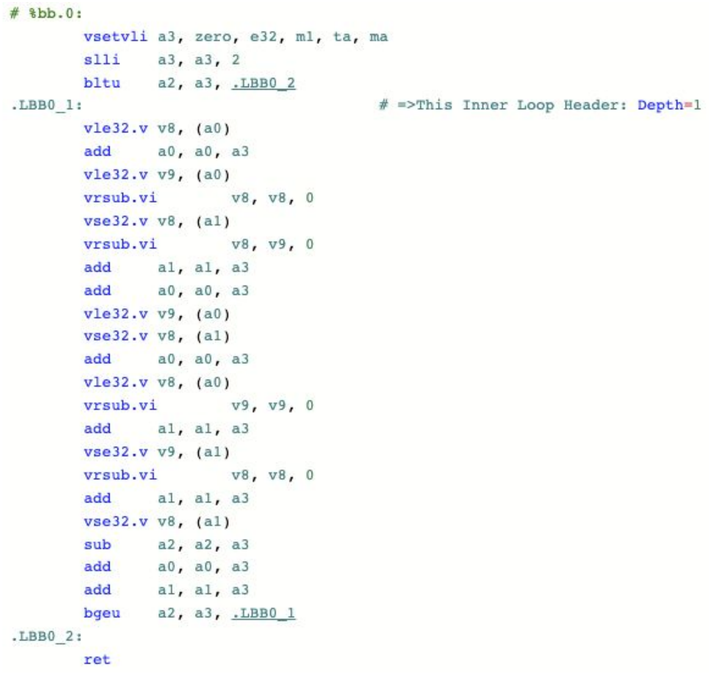
eop Chen, Kito Cheng
SiFive Compiler Team
LLVM Developers' Meeting 2023`

Table of contents

What are intrinsics?

Motivation - High performance

Users seek for better performance. In benchmarks and high performance libraries, hot kernels are investigated thoroughly and one less (or more) instruction in the loop may impact the performance.



```
# %bb.0:
        vsetvli a3, zero, e32, m1, ta, ma
        slli    a3, a3, 2
        bltu    a2, a3, .LBB0_2
.LBB0_1:                                # =>This Inner Loop Header: Depth=1
        vle32.v v8, (a0)
        add     a0, a0, a3
        vle32.v v9, (a0)
        vrsub.vi        v8, v8, 0
        vse32.v v8, (a1)
        vrsub.vi        v8, v9, 0
        add     a1, a1, a3
        add     a0, a0, a3
        vle32.v v9, (a0)
        vse32.v v8, (a1)
        add     a0, a0, a3
        vle32.v v8, (a0)
        vrsub.vi        v9, v9, 0
        add     a1, a1, a3
        vse32.v v9, (a1)
        vrsub.vi        v8, v8, 0
        add     a1, a1, a3
        vse32.v v8, (a1)
        sub     a2, a2, a3
        add     a0, a0, a3
        add     a1, a1, a3
        bgeu    a2, a3, .LBB0_1
.LBB0_2:
        ret
```

What are intrinsics?

Approach to improve performance

Upon identifying what causes the performance regression or when a potential performance improvement is observed, we have two possible ways of resolving the problem.

1. Source code performance tuning
2. Improve the optimization pass in the compiler

"Source code level" is the most straightforward approach in the short term.

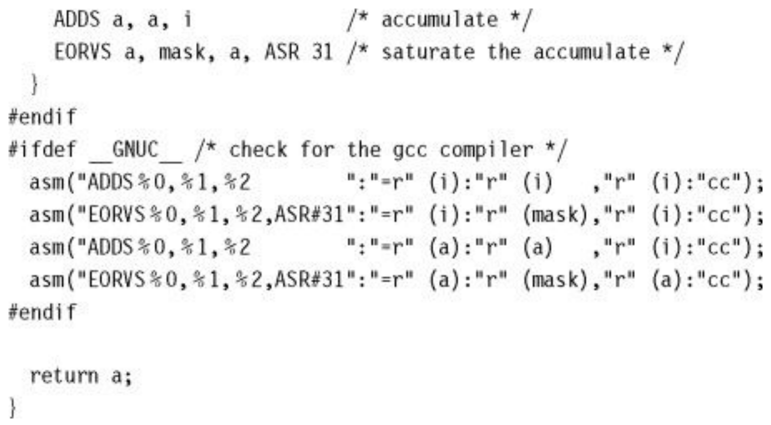
What are intrinsics?

Inline assembly

Using inline assembly allows users to control the exact code generated.

However we will be troubled by this approach.

- Users will have to handle register allocation
- Tailored inline code is platform specific



```
    ADDS a, a, i              /* accumulate */
    EORVS a, mask, a, ASR 31 /* saturate the accumulate */
  }
#endif
#ifdef __GNUC__ /* check for the gcc compiler */
  asm("ADDS %0,%1,%2         ":"=r" (i):"r" (i)   ,"r" (i):"cc");
  asm("EORVS %0,%1,%2,ASR#31":"=r" (i):"r" (mask),"r" (i):"cc");
  asm("ADDS %0,%1,%2         ":"=r" (a):"r" (a)   ,"r" (i):"cc");
  asm("EORVS %0,%1,%2,ASR#31":"=r" (a):"r" (mask),"r" (a):"cc");
#endif

  return a;
}
```

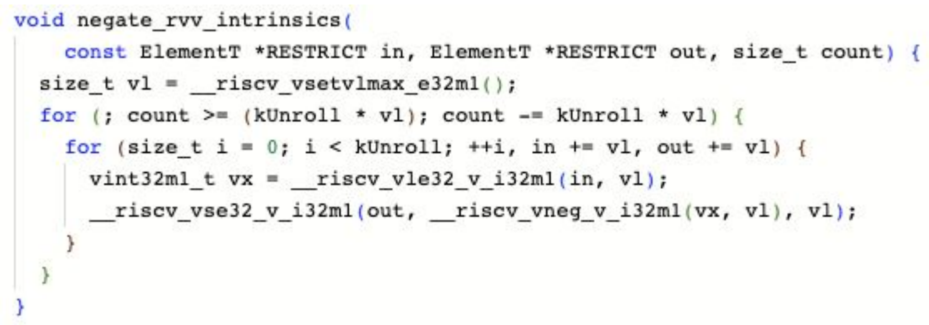
What are intrinsics?

Exposing intrinsic (built-in functions)

Hence, the compiler seeks to expose interfaces for users to assembly level control.

Intrinsics are interfaces to instruction level semantic control.

Compiler define values used by the intrinsics that suits the low level semantic and allows the compiler to handle the tedious tasks.

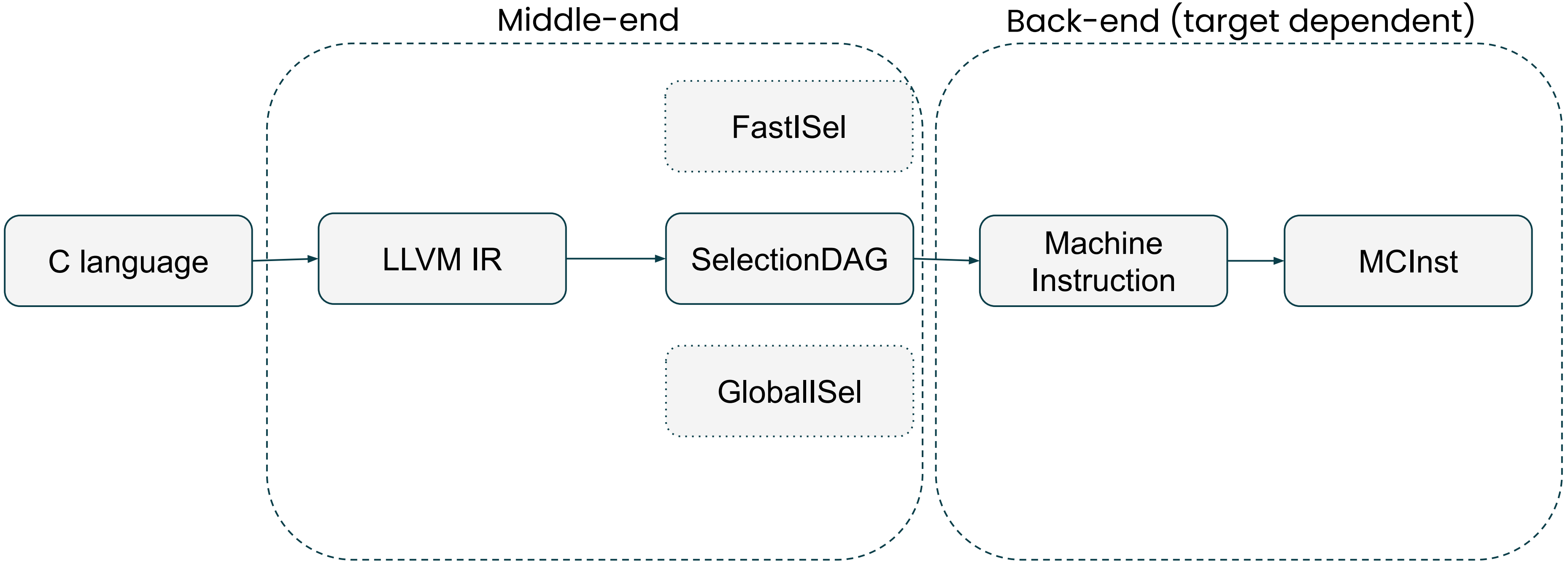


```
void negate_rvv_intrinsics(
    const ElementT *RESTRICT in, ElementT *RESTRICT out, size_t count) {
  size_t vl = __riscv_vsetvlmax_e32m1();
  for (; count >= (kUnroll * vl); count -= kUnroll * vl) {
    for (size_t i = 0; i < kUnroll; ++i, in += vl, out += vl) {
      vint32m1_t vx = __riscv_vle32_v_i32m1(in, vl);
      __riscv_vse32_v_i32m1(out, __riscv_vneg_v_i32m1(vx, vl), vl);
    }
  }
}
```

How are the intrinsics lowered in LLVM

Workflow in LLVM

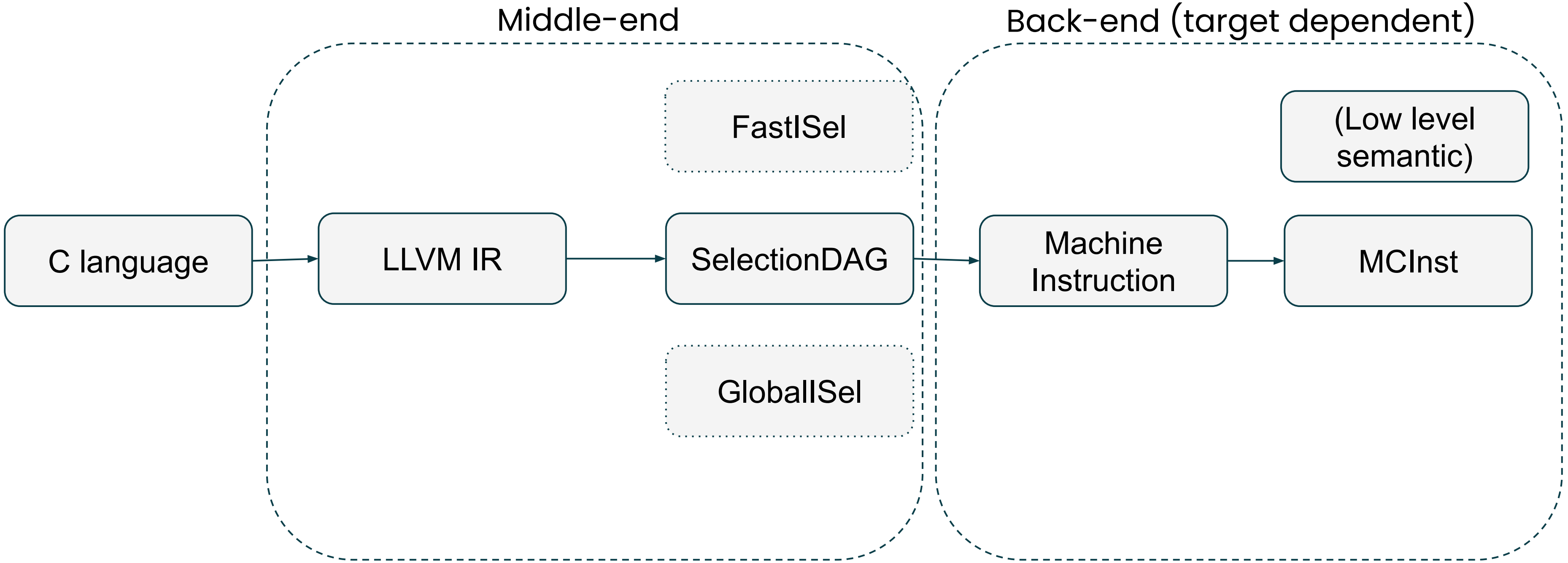
In this section, we introduce the infrastructures LLVM provide for users to represent the intrinsics.



How are the intrinsics lowered in LLVM

Workflow in LLVM

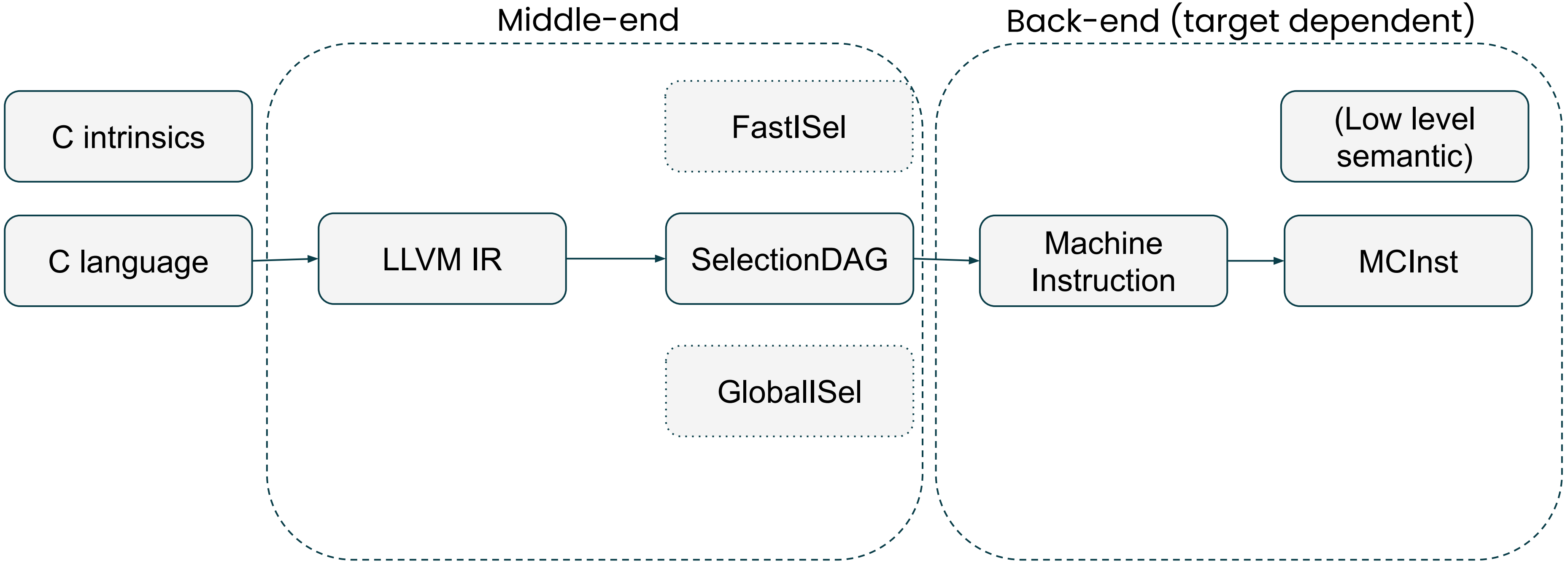
In this section, we introduce the infrastructures LLVM provide for users to represent the intrinsics.



How are the intrinsics lowered in LLVM

Workflow in LLVM

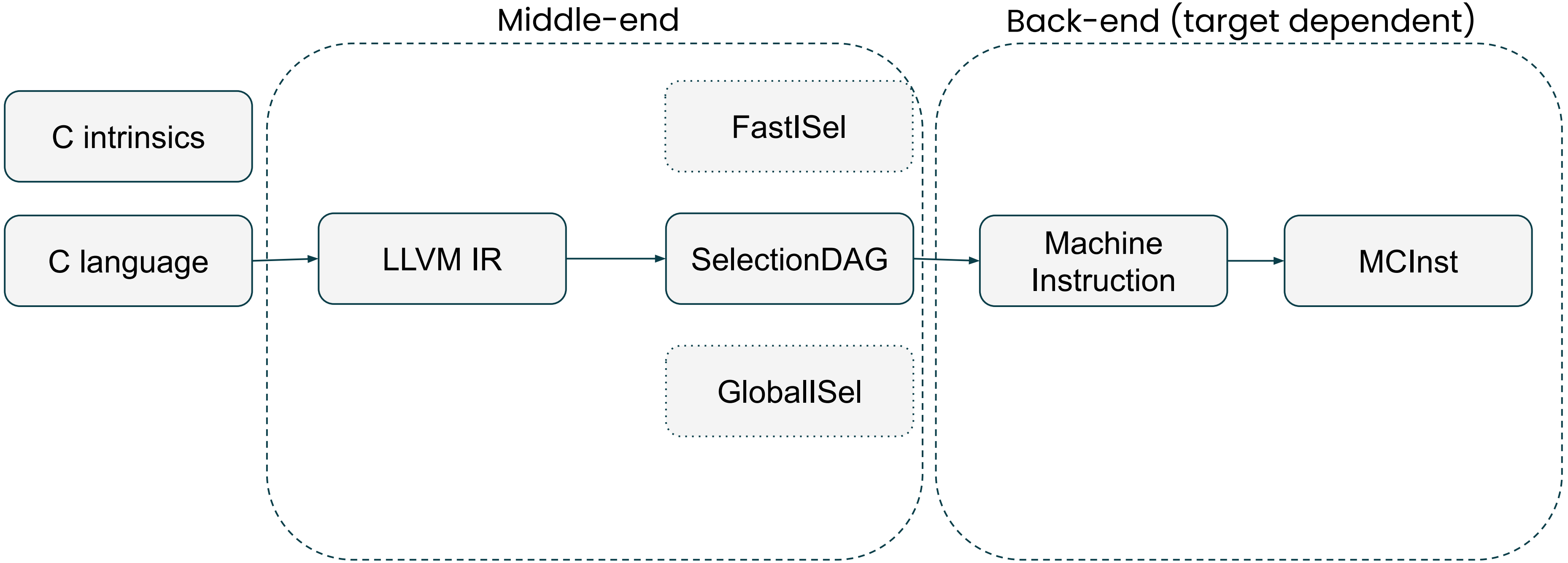
In this section, we introduce the infrastructures LLVM provide for users to represent the intrinsics.



How are the intrinsics lowered in LLVM

Workflow in LLVM

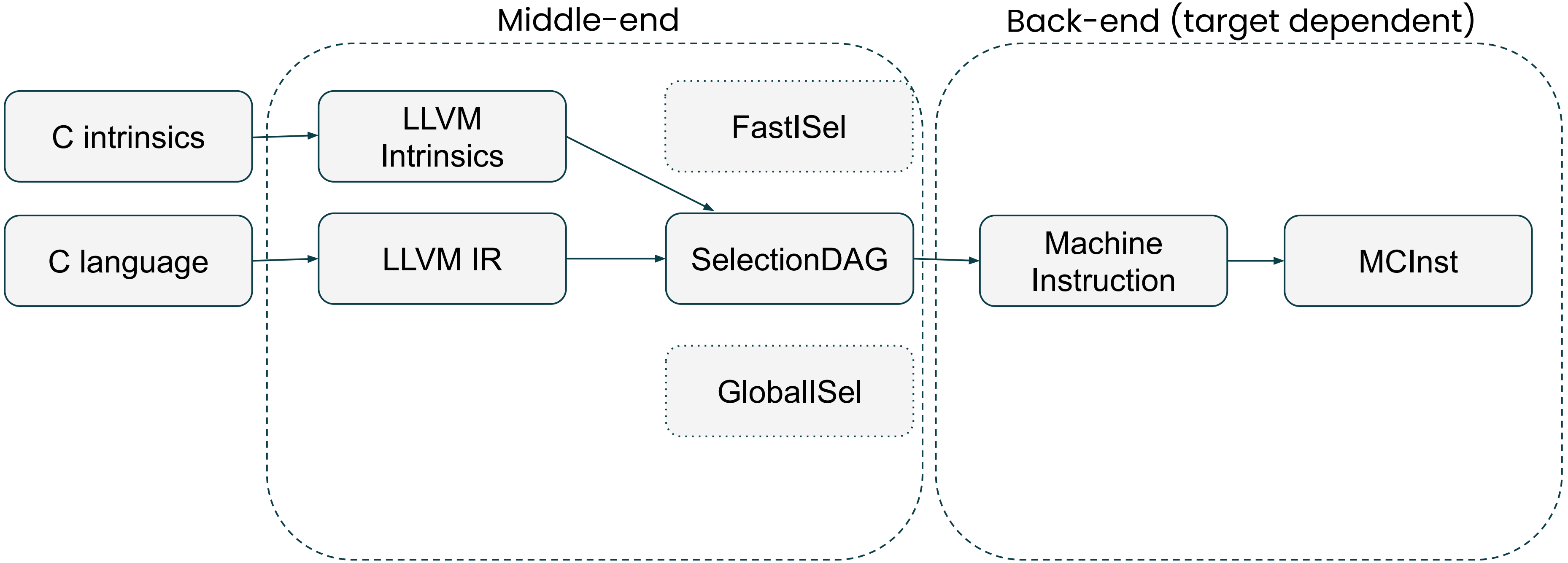
In this section, we introduce the infrastructures LLVM provide for users to represent the intrinsics.



How are the intrinsics lowered in LLVM

Workflow in LLVM

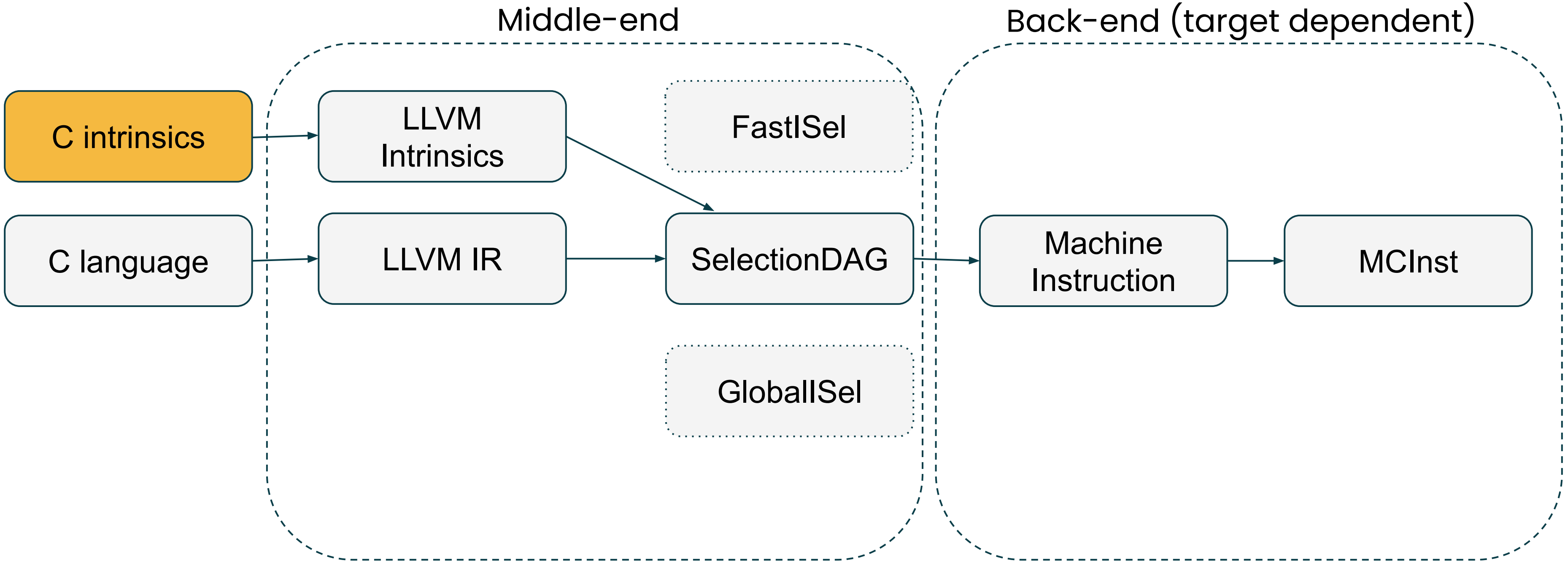
In this section, we introduce the infrastructures LLVM provide for users to represent the intrinsics.



How are the intrinsics lowered in LLVM

Workflow in LLVM

In this section, we introduce the infrastructures LLVM provide for users to represent the intrinsics.



How are the intrinsics lowered in LLVM

Defining intrinsic types in Clang - The `BuiltinType` class

```
/// clang/include/clang/AST/Type.h

/// This class is used for builtin types like 'int'. Builtin
/// types are always canonical and have a literal name field.
class BuiltinType : public Type {
public:
  enum Kind {
// OpenCL image types
#define IMAGE_TYPE(ImgType, Id, SingletonId, Access, Suffix) Id,
#include "clang/Basic/OpenCLImageTypes.def"






// All other builtin types
#define BUILTIN_TYPE(Id, SingletonId) Id,
#define LAST_BUILTIN_TYPE(Id) LastKind = Id
#include "clang/AST/BuiltinTypes.def"
  };
  /* ... */
}
```

How are the intrinsics lowered in LLVM

Defining intrinsic types in Clang - Registering singleton in `ASTContext::InitBuiltTypes`

```
/// clang/lib/AST/ASTContext.cpp

void ASTContext::InitBuiltinTypes(const TargetInfo &Target,
                                  const TargetInfo *AuxTarget) {

  // C99 6.2.5p19.
  InitBuiltinType(VoidTy, BuiltinType::Void);

  // C99 6.2.5p4.
  InitBuiltinType(SignedCharTy, BuiltinType::SChar);
  InitBuiltinType(ShortTy, BuiltinType::Short);
  InitBuiltinType(IntTy, BuiltinType::Int);
  InitBuiltinType(LongTy, BuiltinType::Long);
  InitBuiltinType(LongLongTy, BuiltinType::LongLong);








}
```

How are the intrinsics lowered in LLVM

Defining intrinsic types in Clang - Implement conversion to LLVM IR in `CodeGenTypes::ConvertTypes`

```
/// clang/lib/CodeGen/CodeGenTypes.cpp

/// ConvertType - Convert the specified type to its LLVM form.
llvm::Type *CodeGenTypes::ConvertType(QualType T) {
  T = Context.getCanonicalType(T);

  case BuiltinType::Bool:
    // Note that we always return bool as i1 for use as a scalar type.
    ResultType = llvm::Type::getInt1Ty(getLLVMContext());
    break;








}
```

How are the intrinsics lowered in LLVM

Defining intrinsics in Clang - Declare under `Builtins.def`

```
/// clang/include/clang/Basic/Builtins.def

// Standard libc/libm functions:
BUILTIN(__builtin_atan2 , "ddd" , "Fne")
BUILTIN(__builtin_atan2f, "fff" , "Fne")
BUILTIN(__builtin_atan2l, "LdLdLd", "Fne")
BUILTIN(__builtin_atan2f128, "LLdLLdLLd", "Fne")
BUILTIN(__builtin_abs , "ii" , "ncF")
```

```
/// clang/lib/Basic/Builtins.cpp

static constexpr Builtin::Info BuiltinInfo[] = {
{"not a builtin function", nullptr, nullptr, nullptr, HeaderDesc::NO_HEADER,
ALL_LANGUAGES},
#define BUILTIN(ID, TYPE, ATTRS) \
{#ID, TYPE, ATTRS, nullptr, HeaderDesc::NO_HEADER, ALL_LANGUAGES},
#define LANGBUILTIN(ID, TYPE, ATTRS, LANGS) \
{#ID, TYPE, ATTRS, nullptr, HeaderDesc::NO_HEADER, LANGS},
#define LIBBUILTIN(ID, TYPE, ATTRS, HEADER, LANGS) \
{#ID, TYPE, ATTRS, nullptr, HeaderDesc::HEADER, LANGS},
#include "clang/Basic/Builtins.def"
};
```

How are the intrinsics lowered in LLVM

Defining intrinsics in Clang - Declare under `Builtins.def`

```
/// clang/include/clang/Basic/Builtins.def

// Standard libc/libm functions:
BUILTIN(__builtin_atan2 , "ddd" , "Fne")
BUILTIN(__builtin_atan2f, "fff" , "Fne")
BUILTIN(__builtin_atan2l, "LdLdLd", "Fne")
BUILTIN(__builtin_atan2f128, "LLdLLdLLd", "Fne")
BUILTIN(__builtin_abs , "ii" , "ncF")
```

```
double __builtin_atan2(double, double);
float __builtin_atan2f(float, float);
long double __builtin_atan2l(long double, long double);
long long double __buildf128(long long double, long long double);
int __builtin_abs(int);
```

```
/// clang/include/clang/Basic/Builtins.def

// v -> void
// b -> boolean
// c -> char
// s -> short
// i -> int
// h -> half (__fp16, OpenCL)
// x -> half (_Float16)
// y -> half (__bf16)
// f -> float
// d -> double

// L -> long (e.g. Li for 'long int', Ld for 'long double')
// LL -> long long (e.g. LLi for 'long long int', LLd for __float128)
```

How are the intrinsics lowered in LLVM

Semantic checks - Check function call parameters

```
/// clang/lib/Sema/SemaChecking.cpp

bool Sema::CheckTSBuiltinFunctionCall(const TargetInfo &TI, unsigned BuiltinID,
                                      CallExpr *TheCall) {

  switch (TI.getTriple().getArch()) {
  default:
    // Some builtins don't require additional checking, so just consider these
    // acceptable.
    return false;








}
```

How are the intrinsics lowered in LLVM

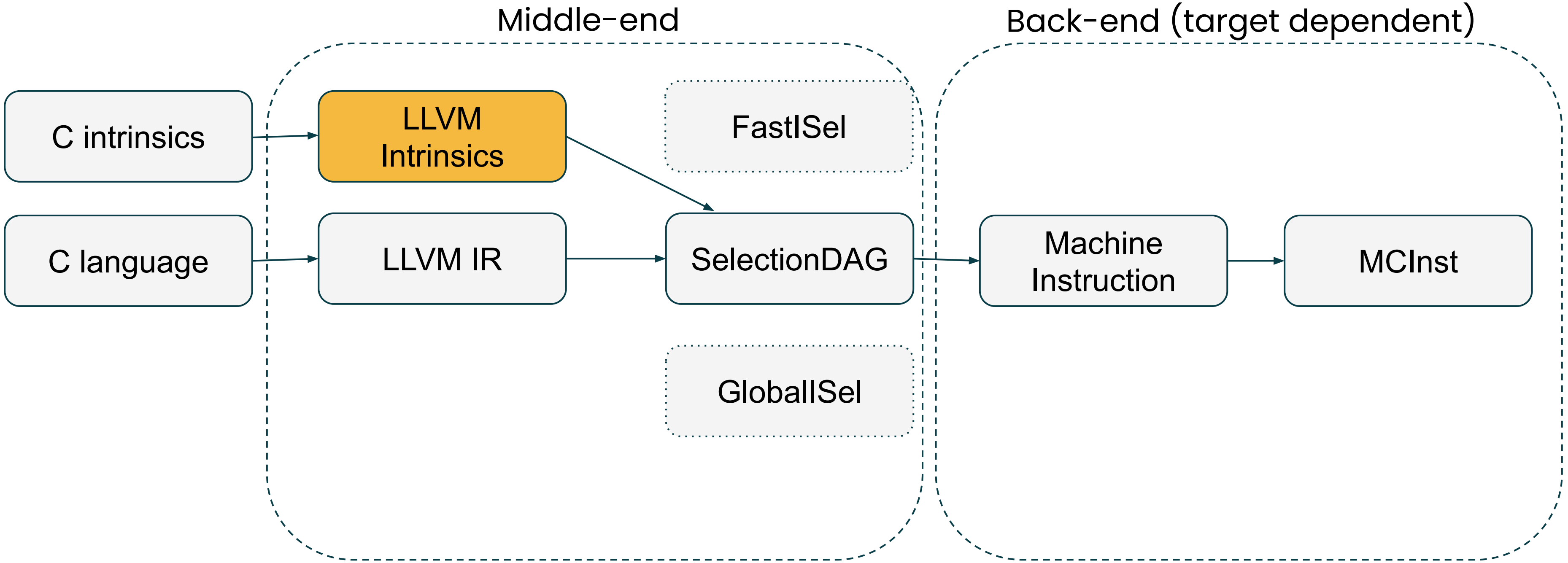
Semantic checks - Check type support for variable declaration

```
/// clang/lib/Sema/SemaDecl.cpp

void Sema::CheckVariableDeclarationType(VarDecl *NewVD) {

  QualType T = NewVD->getType();







}
```

How are the intrinsics lowered in LLVM

Workflow in LLVM

In this section, we introduce the infrastructures LLVM provide for users to represent the intrinsics.



How are intrinsics lowered in LLVM

Declaring the intrinsics in LLVM IR

```
/// llvm/include/llvm/IR/Intrinsics.td

// Intrinsic class - This is used to define one LLVM intrinsic. The name of the
// intrinsic definition should start with "int_", then match the LLVM intrinsic
// name with the "llvm." prefix removed, and all "."s turned into "_"s. For
// example, llvm.bswap.i16 -> int_bswap_i16.
class Intrinsic<list<LLVMType> ret_types,
                list<LLVMType> param_types = [],
                list<IntrinsicProperty> intr_properties = [],
                string name = "",
                list<SDNodeProperty> sd_properties = [],
                bit disable_default_attributes = true> : SDPatternOperator {
  string LLVMName = name;
  string TargetPrefix = ""; // Set to a prefix for target-specific intrinsics.
  list<LLVMType> RetTypes = ret_types;
  list<LLVMType> ParamTypes = param_types;
  list<IntrinsicProperty> IntrProperties = intr_properties;
  let Properties = sd_properties;

  /* ... */
}
```

How are the intrinsics lowered in LLVM

Code gen to LLVM IR under `CGBuiltin.cpp`

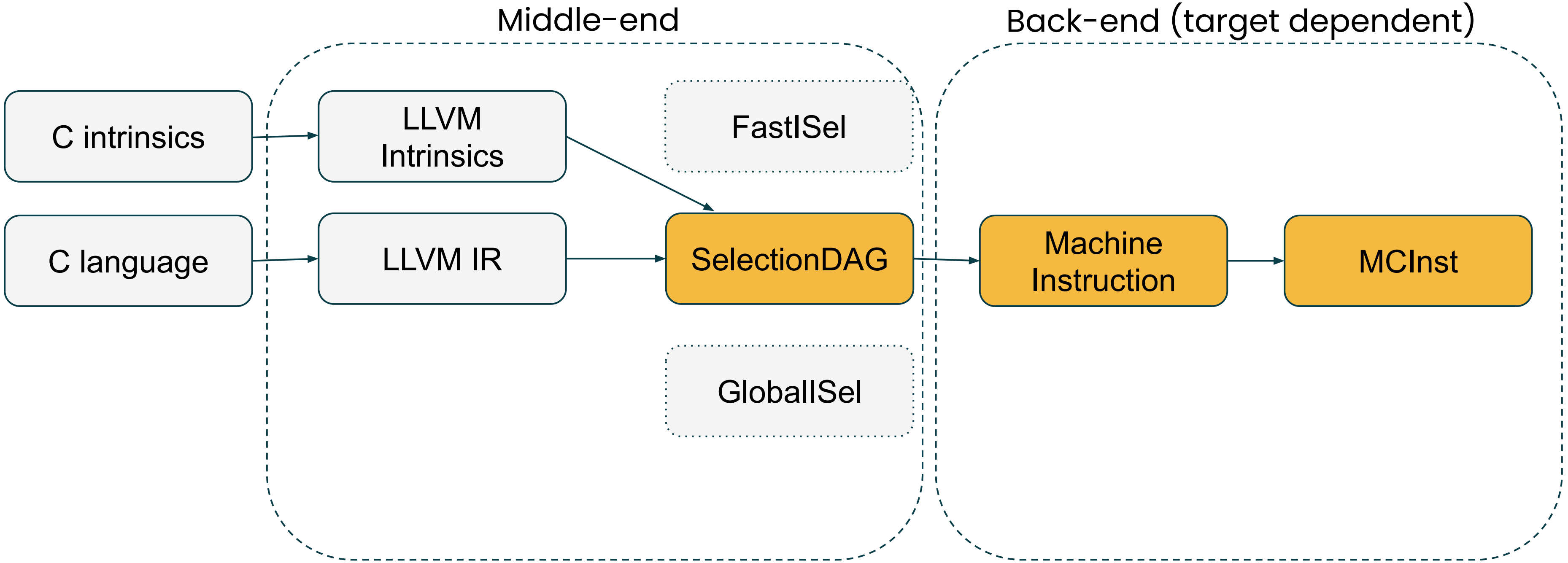
```
/// clang/lib/CodeGen/CGBuiltin.cpp

Value *CodeGenFunction::EmitRISCVBuiltinExpr(unsigned BuiltinID,
                                             const CallExpr *E,
                                             ReturnValueSlot ReturnValue) {

  SmallVector<Value *, 4> Ops;
  llvm::Type *ResultType = ConvertType(E->getType());


  Intrinsic::ID ID;
  llvm::SmallVector<llvm::Type *, 2> IntrinsicTypes;

  /* ... */

  llvm::Function *F = CGM.getIntrinsic(ID, IntrinsicTypes);
  return Builder.CreateCall(F, Ops, "");

}
```

```
llvm::Function *CodeGenModule::getIntrinsic(unsigned IID,
                                            ArrayRef<llvm::Type*> Tys) {
  return llvm::Intrinsic::getDeclaration(&getModule(), (llvm::Intrinsic::ID)IID,
                                         Tys);
}
```

How are the intrinsics lowered in LLVM

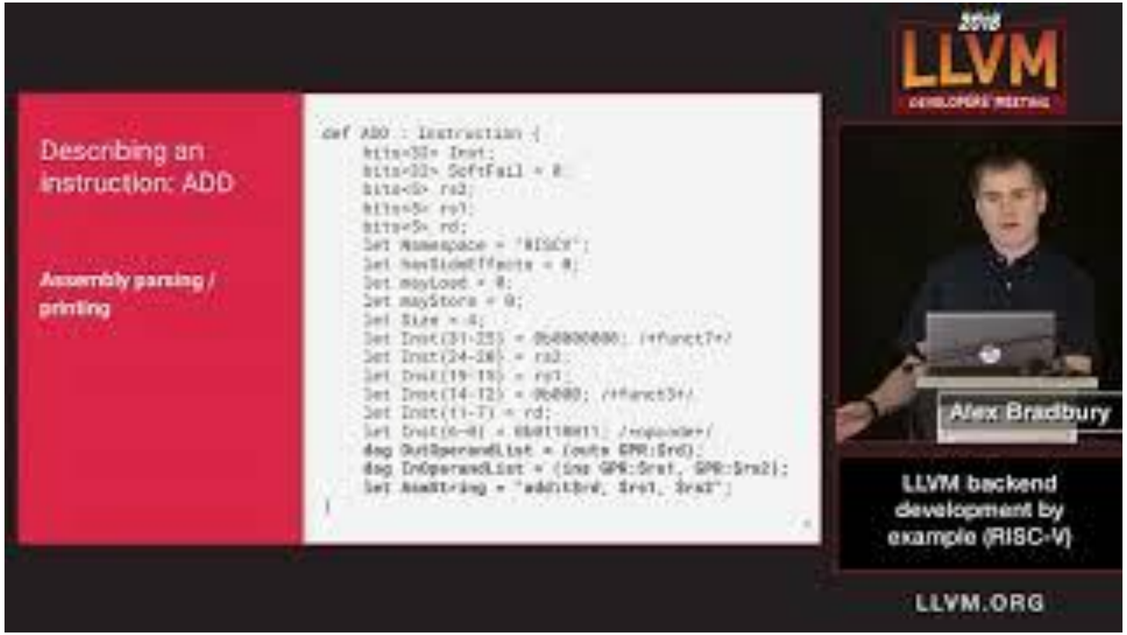
Workflow in LLVM

In this section, we introduce the infrastructures LLVM provide for users to represent the intrinsics.



How are the intrinsics lowered in LLVM

Alex Bradbury - “LLVM backend development by example (RISC-V)”



Describing an instruction: ADD

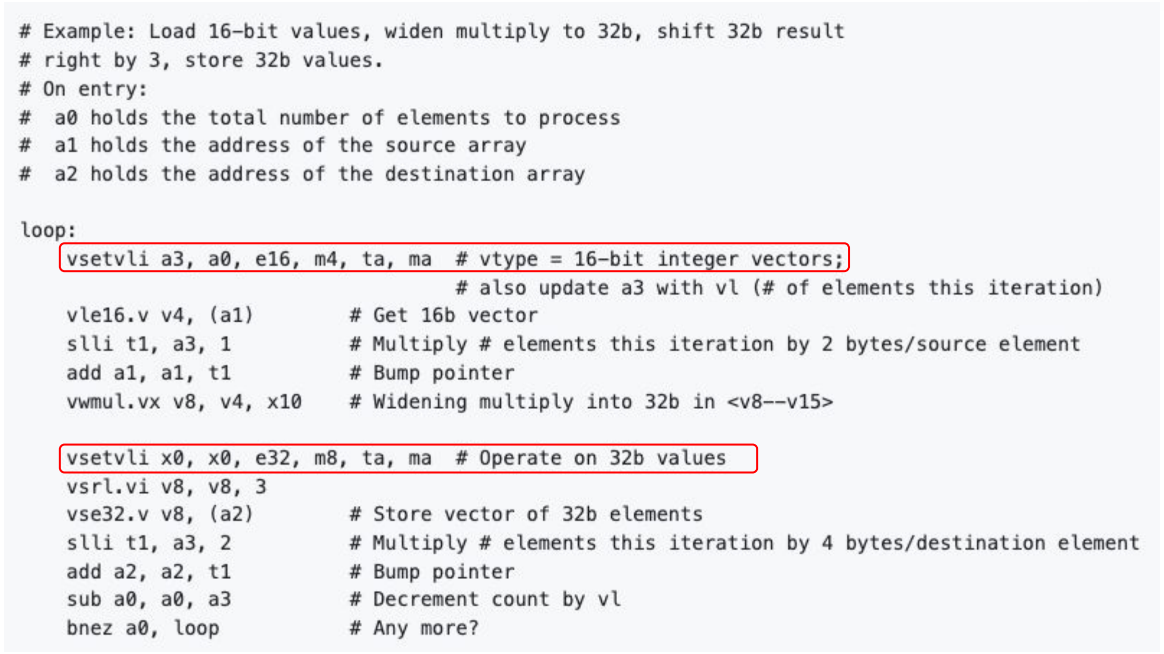
Assembly parsing / printing

```
def ADD : Instruction {
    bits<32> Inst;
    bits<32> SoftFail = 0;
    bits<5> rs2;
    bits<5> rs1;
    bits<5> rd;
    let Namespace = "RISCV";
    let hasSideEffects = 0;
    let mayLoad = 0;
    let mayStore = 0;
    let Size = 4;
    let Inst{31-25} = 0b0000000; /*funct7*/
    let Inst{24-20} = rs2;
    let Inst{19-15} = rs1;
    let Inst{14-12} = 0b000; /*funct3*/
    let Inst{11-7} = rd;
    let Inst{6-0} = 0b0110011; /*opcode*/
    dag OutOperandList = (outs GPR:$rd);
    dag InOperandList = (ins GPR:$rs1, GPR:$rs2);
    let AsmString = "add\t$rd, $rs1, $rs2";
}
```

[SelectionDAG and pattern matching starts from 30:14]

Case study: The RISC-V vector intrinsics

Introduction to the RISC-V "V" (RVV) extension



```
# Example: Load 16-bit values, widen multiply to 32b, shift 32b result
# right by 3, store 32b values.
# On entry:
#  a0 holds the total number of elements to process
#  a1 holds the address of the source array
#  a2 holds the address of the destination array

loop:
    vsetvli a3, a0, e16, m4, ta, ma  # vtype = 16-bit integer vectors;
                                     # also update a3 with vl (# of elements this iteration)
    vle16.v v4, (a1)        # Get 16b vector
    slli t1, a3, 1          # Multiply # elements this iteration by 2 bytes/source element
    add a1, a1, t1          # Bump pointer
    vwmul.vx v8, v4, x10    # Widening multiply into 32b in <v8--v15>

    vsetvli x0, x0, e32, m8, ta, ma  # Operate on 32b values
    vsrl.vi v8, v8, 3
    vse32.v v8, (a2)        # Store vector of 32b elements
    slli t1, a3, 2          # Multiply # elements this iteration by 4 bytes/destination element
    add a2, a2, t1          # Bump pointer
    sub a0, a0, a3          # Decrement count by vl
    bnez a0, loop           # Any more?
```

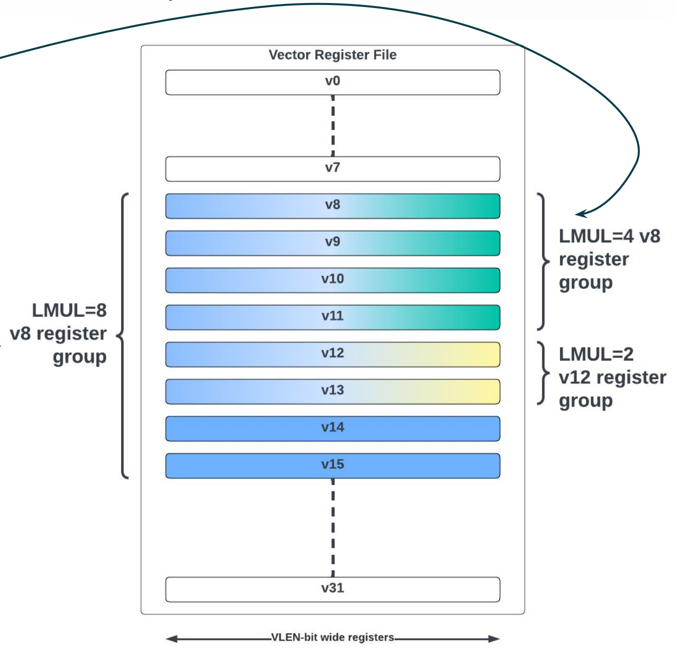
Code snippet is from the vector specification

Case study: The RISC-V vector intrinsics

Introduction to the RISC-V "V" (RVV) extension

The RISC-V vector extension allows vector register grouping with the `LMUL` parameter.

```
# Example: Load 16-bit values, widen multiply to 32b, shift 32b result
# right by 3, store 32b values.
# On entry:
#  a0 holds the total number of elements to process
#  a1 holds the address of the source array
#  a2 holds the address of the destination array

loop:
    vsetvli a3, a0, e16, m4, ta, ma  # vtype = 16-bit integer vectors;
                                     # also update a3 with vl (# of elements this i
    vle16.v v4, (a1)        # Get 16b vector
    slli t1, a3, 1          # Multiply # elements this iteration by 2 bytes/source
    add a1, a1, t1          # Bump pointer
    vwmul.vx v8, v4, x10    # Widening multiply into 32b in <v8--v15>

    vsetvli x0, x0, e32, m8, ta, ma  # Operate on 32b values
    vsrl.vi v8, v8, 3
    vse32.v v8, (a2)        # Store vector of 32b elements
    slli t1, a3, 2          # Multiply # elements this iteration by 4 bytes/destina
    add a2, a2, t1          # Bump pointer
    sub a0, a0, a3          # Decrement count by vl
    bnez a0, loop           # Any more?
```



Code snippet is from the vector specification
Picture is from blog post by Nicolas Brunie

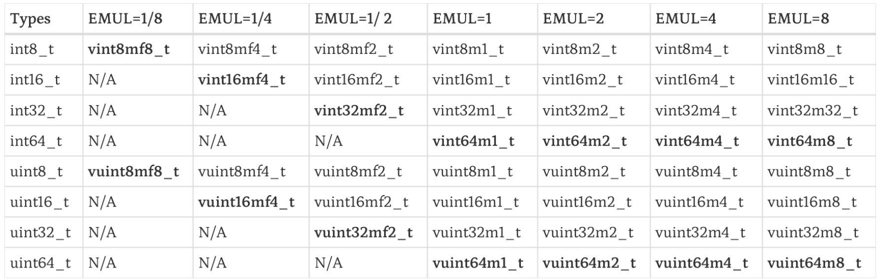
Case study: The RISC-V vector intrinsics

Introduction to the RISC-V "V" (RVV) extension

```
vadd.vv vd, vs2, vs1
```

SEW ∈ {8, 16, 32, 64} X LMUL ∈ {⅛, ¼, ½, 1, 2, 4, 8}

```
vint16m1_t __riscv_vadd_vv_i16m1 (vint16m1_t op1, vint16m1_t op2, size_t vl);
vint16m2_t __riscv_vadd_vv_i16m2 (vint16m2_t op1, vint16m2_t op2, size_t vl);
vint16m4_t __riscv_vadd_vv_i16m4 (vint16m4_t op1, vint16m4_t op2, size_t vl);
vint16m8_t __riscv_vadd_vv_i16m8 (vint16m8_t op1, vint16m8_t op2, size_t vl);
vint32m1_t __riscv_vadd_vv_i32m1 (vint32m1_t op1, vint32m1_t op2, size_t vl);
vint32m2_t __riscv_vadd_vv_i32m2 (vint32m2_t op1, vint32m2_t op2, size_t vl);
vint32m4_t __riscv_vadd_vv_i32m4 (vint32m4_t op1, vint32m4_t op2, size_t vl);
vint32m8_t __riscv_vadd_vv_i32m8 (vint32m8_t op1, vint32m8_t op2, size_t vl);
```



Types	EMUL=1/8	EMUL=1/4	EMUL=1/ 2	EMUL=1	EMUL=2	EMUL=4	EMUL=8
int8_t	**vint8mf8_t**	vint8mf4_t	vint8mf2_t	vint8m1_t	vint8m2_t	vint8m4_t	vint8m8_t
int16_t	N/A	**vint16mf4_t**	vint16mf2_t	vint16m1_t	vint16m2_t	vint16m4_t	vint16m16_t
int32_t	N/A	N/A	**vint32mf2_t**	vint32m1_t	vint32m2_t	vint32m4_t	vint32m32_t
int64_t	N/A	N/A	N/A	**vint64m1_t**	**vint64m2_t**	**vint64m4_t**	**vint64m8_t**
uint8_t	**vuint8mf8_t**	vuint8mf4_t	vuint8mf2_t	vuint8m1_t	vuint8m2_t	vuint8m4_t	vuint8m8_t
uint16_t	N/A	**vuint16mf4_t**	vuint16mf2_t	vuint16m1_t	vuint16m2_t	vuint16m4_t	vuint16m8_t
uint32_t	N/A	N/A	**vuint32mf2_t**	vuint32m1_t	vuint32m2_t	vuint32m4_t	vuint32m8_t
uint64_t	N/A	N/A	N/A	**vuint64m1_t**	**vuint64m2_t**	**vuint64m4_t**	**vuint64m8_t**

Table 1. Integer types

Table from the RVV C intrinsics specification

Case study: The RISC-V vector intrinsics

Introduction to the RISC-V "V" (RVV) extension

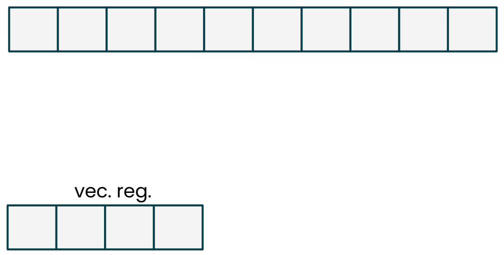
```c
#include <riscv_vector.h>

float reduce_max(const float *in, size_t n) {

  // VLMAX = Vector Length / element width
  size_t vlmax = __riscv_vsetvlmax_e32m1();
  vfloat32m1_t max_array = __riscv_vfmv_s_f_f32m1(in[0], vlmax);
  while (n > 0) {
    size_t vl =  __riscv_vsetvl_e32m1(n); // LMUL = 1
    // size_t vl = __riscv_vsetvl_e32m8(n); // LMUL = 8

    vfloat32m1_t vs2 = __riscv_vle32_v_f32m1(in, vl);
    max_array = __riscv_vfmax_vv_f32m1(max_array, vs2, vl);

    in += vl;
    n -= vl;
  }

  vfloat32m1_t reduce_max = __riscv_vfredmax_vs_f32m1_f32m1(max_array, max_array, vlmax);
  return __riscv_vfmv_f_s_f32m1_f32(reduce_max);
}
```

```c
float reduce_max(const float *in, size_t n) {
  float ret = in[0];
  for (int i=1; i<n; ++i)
    ret = max(ret, in[i]]);
  return ret;
}
```

Case study: The RISC-V vector intrinsics

Introduction to the RISC-V "V" (RVV) extension

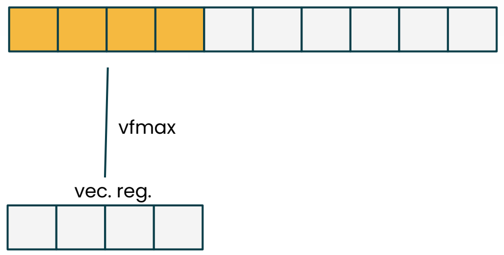
```
#include <riscv_vector.h>

float reduce_max(const float *in, size_t n) {

  // VLMAX = Vector Length / element width
  size_t vlmax = __riscv_vsetvlmax_e32m1();
  vfloat32m1_t max_array = __riscv_vfmv_s_f_f32m1(in[0], vlmax);
  while (n > 0) {
    size_t vl =  __riscv_vsetvl_e32m1(n); // LMUL = 1
    // size_t vl = __riscv_vsetvl_e32m8(n); // LMUL = 8

    vfloat32m1_t vs2 = __riscv_vle32_v_f32m1(in, vl);
    max_array = __riscv_vfmax_vv_f32m1(max_array, vs2, vl);

    in += vl;
    n -= vl;
  }

  vfloat32m1_t reduce_max = __riscv_vfredmax_vs_f32m1_f32m1(max_array, max_array, vlmax);
  return __riscv_vfmv_f_s_f32m1_f32(reduce_max);
}
```



Case study: The RISC-V vector intrinsics

Introduction to the RISC-V "V" (RVV) extension

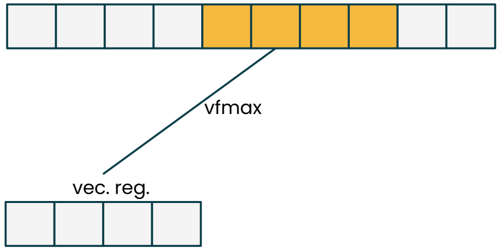
```c
#include <riscv_vector.h>

float reduce_max(const float *in, size_t n) {

  // VLMAX = Vector Length / element width
  size_t vlmax = __riscv_vsetvlmax_e32m1();
  vfloat32m1_t max_array = __riscv_vfmv_s_f_f32m1(in[0], vlmax);
  while (n > 0) {
    size_t vl =  __riscv_vsetvl_e32m1(n); // LMUL = 1
    // size_t vl = __riscv_vsetvl_e32m8(n); // LMUL = 8

    vfloat32m1_t vs2 = __riscv_vle32_v_f32m1(in, vl);
    max_array = __riscv_vfmax_vv_f32m1(max_array, vs2, vl);

    in += vl;
    n -= vl;
  }

  vfloat32m1_t reduce_max = __riscv_vfredmax_vs_f32m1_f32m1(max_array, max_array, vlmax);
  return __riscv_vfmv_f_s_f32m1_f32(reduce_max);
}
```



Case study: The RISC-V vector intrinsics

Introduction to the RISC-V "V" (RVV) extension

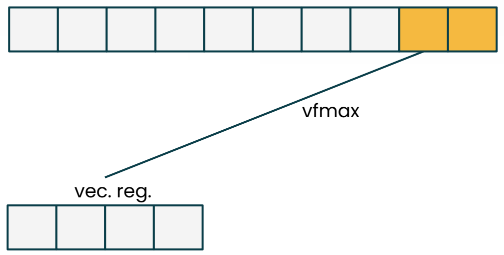
```c
#include <riscv_vector.h>

float reduce_max(const float *in, size_t n) {

  // VLMAX = Vector Length / element width
  size_t vlmax = __riscv_vsetvlmax_e32m1();
  vfloat32m1_t max_array = __riscv_vfmv_s_f_f32m1(in[0], vlmax);
  while (n > 0) {
    size_t vl =  __riscv_vsetvl_e32m1(n); // LMUL = 1
    // size_t vl = __riscv_vsetvl_e32m8(n); // LMUL = 8

    vfloat32m1_t vs2 = __riscv_vle32_v_f32m1(in, vl);
    max_array = __riscv_vfmax_vv_f32m1(max_array, vs2, vl);

    in += vl;
    n -= vl;
  }

  vfloat32m1_t reduce_max = __riscv_vfredmax_vs_f32m1_f32m1(max_array, max_array, vlmax);
  return __riscv_vfmv_f_s_f32m1_f32(reduce_max);
}
```



Case study: The RISC-V vector intrinsics

Introduction to the RISC-V "V" (RVV) extension

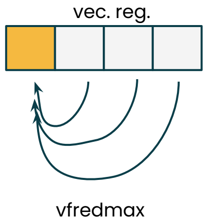
```c
#include <riscv_vector.h>

float reduce_max(const float *in, size_t n) {

  // VLMAX = Vector Length / element width
  size_t vlmax = __riscv_vsetvlmax_e32m1();
  vfloat32m1_t max_array = __riscv_vfmv_s_f_f32m1(in[0], vlmax);
  while (n > 0) {
    size_t vl =  __riscv_vsetvl_e32m1(n); // LMUL = 1
    // size_t vl = __riscv_vsetvl_e32m8(n); // LMUL = 8

    vfloat32m1_t vs2 = __riscv_vle32_v_f32m1(in, vl);
    max_array = __riscv_vfmax_vv_f32m1(max_array, vs2, vl);

    in += vl;
    n -= vl;
  }

  vfloat32m1_t reduce_max = __riscv_vfredmax_vs_f32m1_f32m1(max_array, max_array, vlmax);
  return __riscv_vfmv_f_s_f32m1_f32(reduce_max);
}
```



Case study: The RISC-V vector intrinsics

Introduction to the RISC-V "V" (RVV) extension

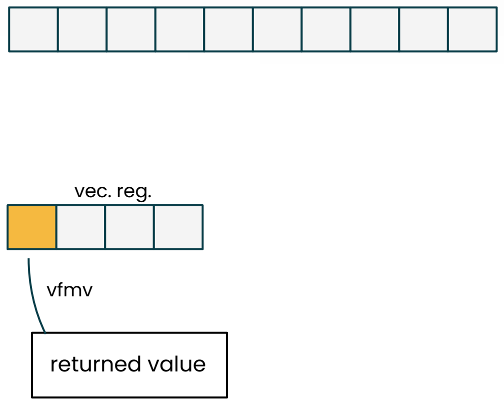
```c
#include <riscv_vector.h>

float reduce_max(const float *in, size_t n) {

  // VLMAX = Vector Length / element width
  size_t vlmax = __riscv_vsetvlmax_e32m1();
  vfloat32m1_t max_array = __riscv_vfmv_s_f_f32m1(in[0], vlmax);
  while (n > 0) {
    size_t vl =  __riscv_vsetvl_e32m1(n); // LMUL = 1
    // size_t vl = __riscv_vsetvl_e32m8(n); // LMUL = 8

    vfloat32m1_t vs2 = __riscv_vle32_v_f32m1(in, vl);
    max_array = __riscv_vfmax_vv_f32m1(max_array, vs2, vl);

    in += vl;
    n -= vl;
  }

  vfloat32m1_t reduce_max = __riscv_vfredmax_vs_f32m1_f32m1(max_array, max_array, vlmax);
  return __riscv_vfmv_f_s_f32m1_f32(reduce_max);
}
```



Case study: The RISC-V vector intrinsics

Introduction to the RISC-V "V" (RVV) extension

```c
#include <riscv_vector.h>

float reduce_max(const float *in, size_t n) {

  // VLMAX = Vector Length / element width
  size_t vlmax = __riscv_vsetvlmax_e32m1();
  vfloat32m1_t max_array = __riscv_vfmv_s_f_f32m1(in[0], vlmax);
  while (n > 0) {
    size_t vl =  __riscv_vsetvl_e32m1(n); // LMUL = 1
    // size_t vl = __riscv_vsetvl_e32m8(n); // LMUL = 8

    vfloat32m1_t vs2 = __riscv_vle32_v_f32m1(in, vl);
    max_array = __riscv_vfmax_vv_f32m1(max_array, vs2, vl);

    in += vl;
    n -= vl;
  }

  vfloat32m1_t reduce_max = __riscv_vfredmax_vs_f32m1_f32m1(max_array, max_array, vlmax);
  return __riscv_vfmv_f_s_f32m1_f32(reduce_max);
}
```



Case study: The RISC-V vector intrinsics

Pseudo instruction level for `vsetvl` insertion

```
# Example: Load 16-bit values, widen multiply to 32b, shift 32b result
# right by 3, store 32b values.
# On entry:
#  a0 holds the total number of elements to process
#  a1 holds the address of the source array
#  a2 holds the address of the destination array

loop:
    vsetvli a3, a0, e16, m4, ta, ma  # vtype = 16-bit integer vectors;
                                     # also update a3 with vl (# of elements this iteration)
    vle16.v v4, (a1)        # Get 16b vector
    slli t1, a3, 1          # Multiply # elements this iteration by 2 bytes/source element
    add a1, a1, t1          # Bump pointer
    vwmul.vx v8, v4, x10    # Widening multiply into 32b in <v8--v15>

    vsetvli x0, x0, e32, m8, ta, ma  # Operate on 32b values
    vsrl.vi v8, v8, 3
    vse32.v v8, (a2)        # Store vector of 32b elements
    slli t1, a3, 2          # Multiply # elements this iteration by 4 bytes/destination element
    add a2, a2, t1          # Bump pointer
    sub a0, a0, a3          # Decrement count by vl
    bnez a0, loop           # Any more?
```

Case study: The RISC-V vector intrinsics

Pseudo instruction level for `vsetvl` insertion

```
vint32m1_t foo(vint32m1_t va, vint32m1_t vb, size_t vl) {
  return __riscv_vadd_vv_i32m1(va, vb, vl);
}
```

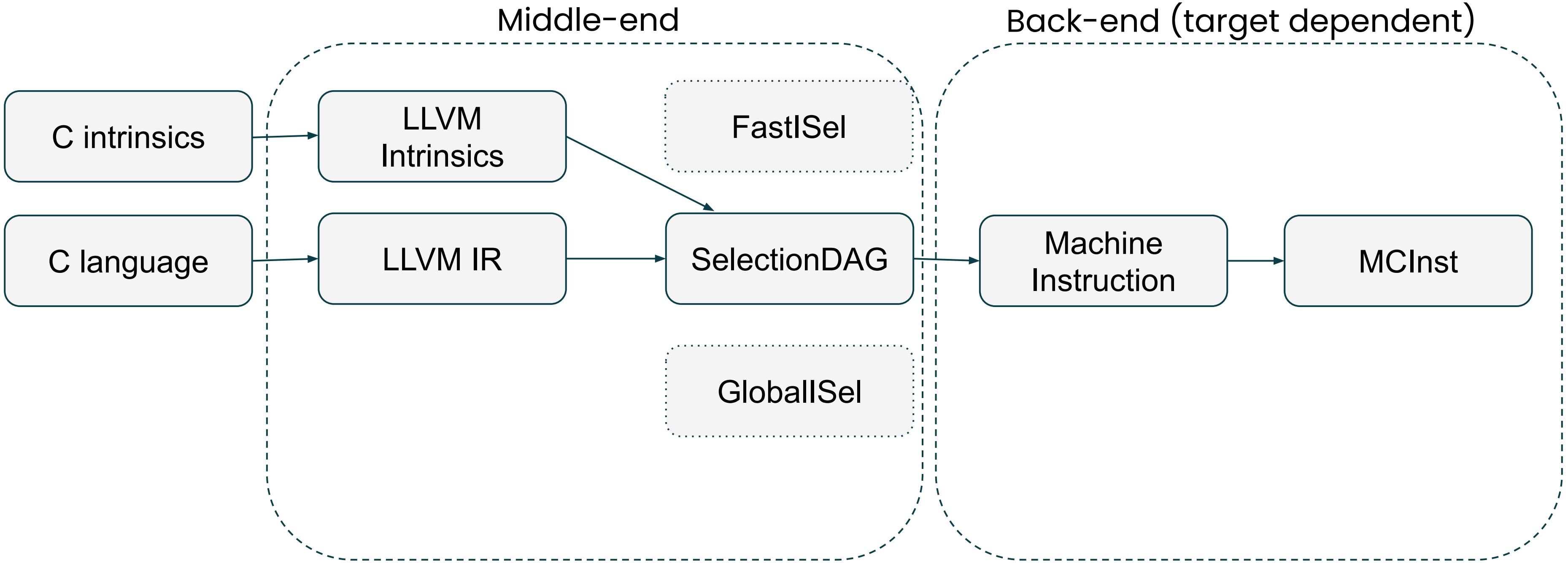
```
define <vscale x 2 x i32>
  @foo(<vscale x 2 x i32> %a, <vscale x 2 x i32> %b, i64 noundef %vl) {
entry:
  %0 = call <vscale x 2 x i32>
  @llvm.riscv.vadd.nxv2i32.nxv2i32.i64(<vscale x 2 x i32> poison,
                                       <vscale x 2 x i32> %a,
                                       <vscale x 2 x i32> %b,
                                       i64 %vl)

  ret <vscale x 2 x i32> %0
}
```

```
vsetvli    %vl, e32, m1
vadd.vv    %0, %a, %b
```

Case study: The RISC-V vector intrinsics

Pseudo instruction level for `vsetvl` insertion

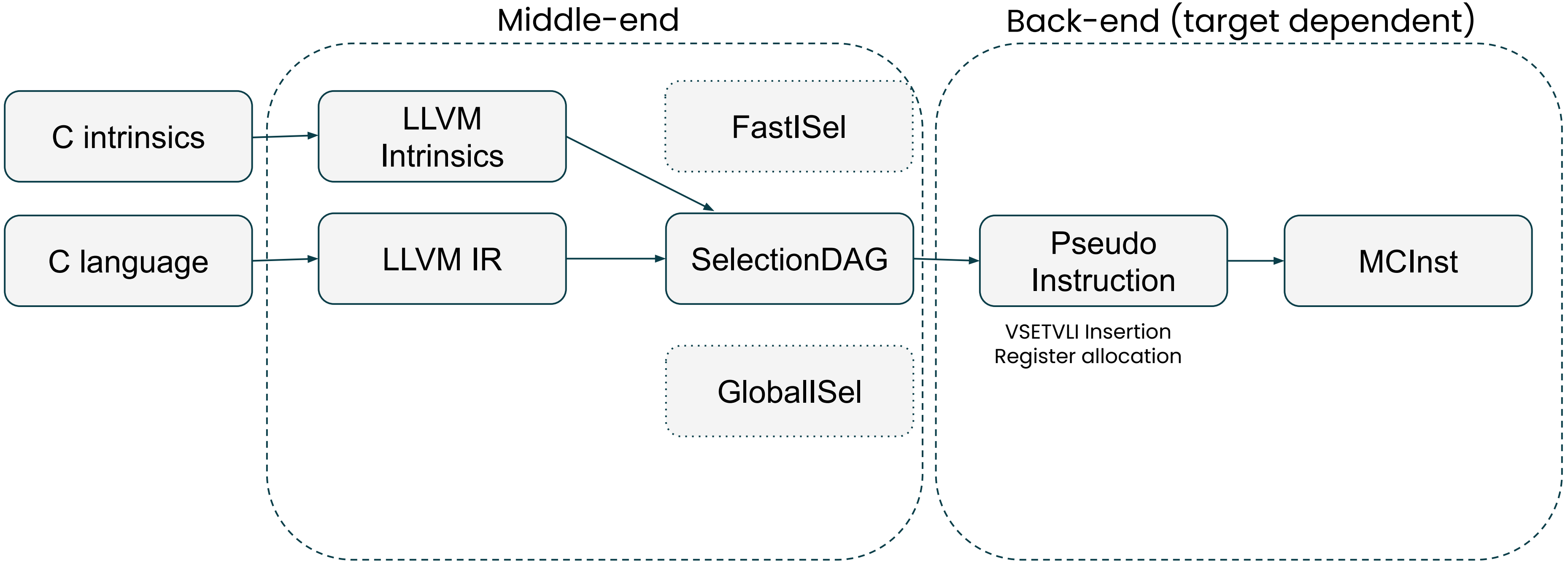
In this section, we introduce the infrastructures LLVM provide for users to represent the intrinsics.



Case study: The RISC-V vector intrinsics

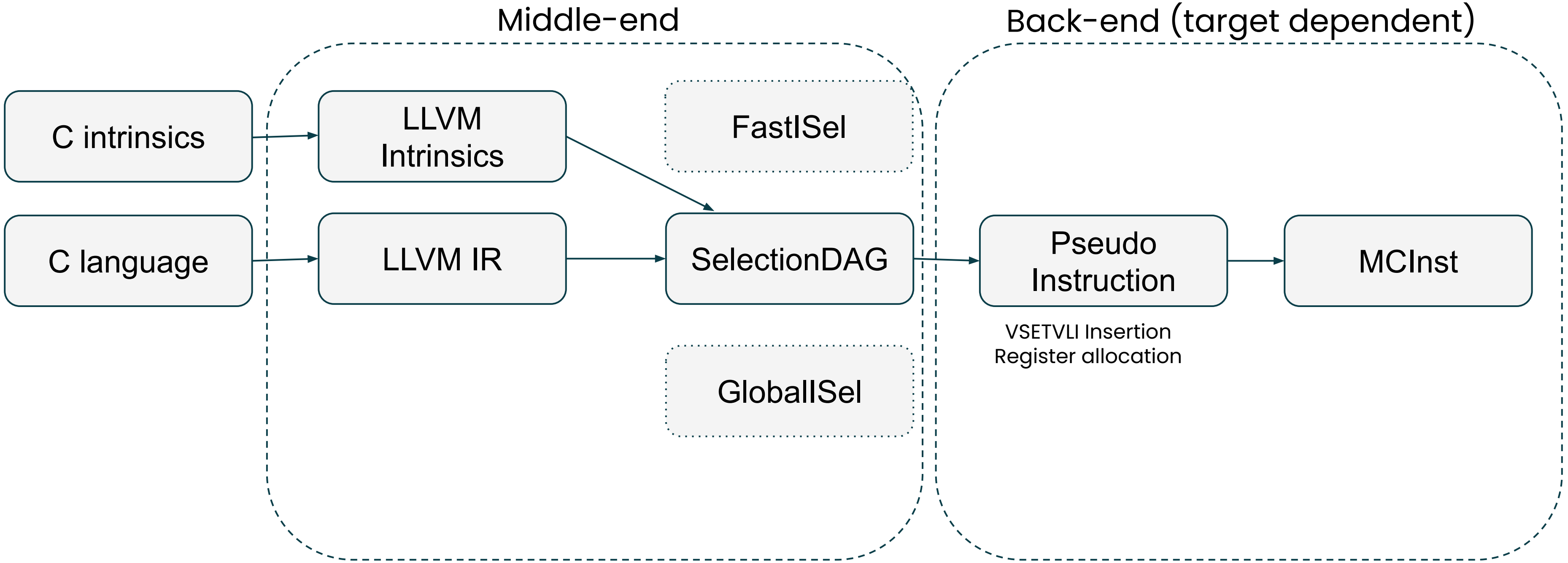
Pseudo instruction level for `vsetvl` insertion

In this section, we introduce the infrastructures LLVM provide for users to represent the intrinsics.



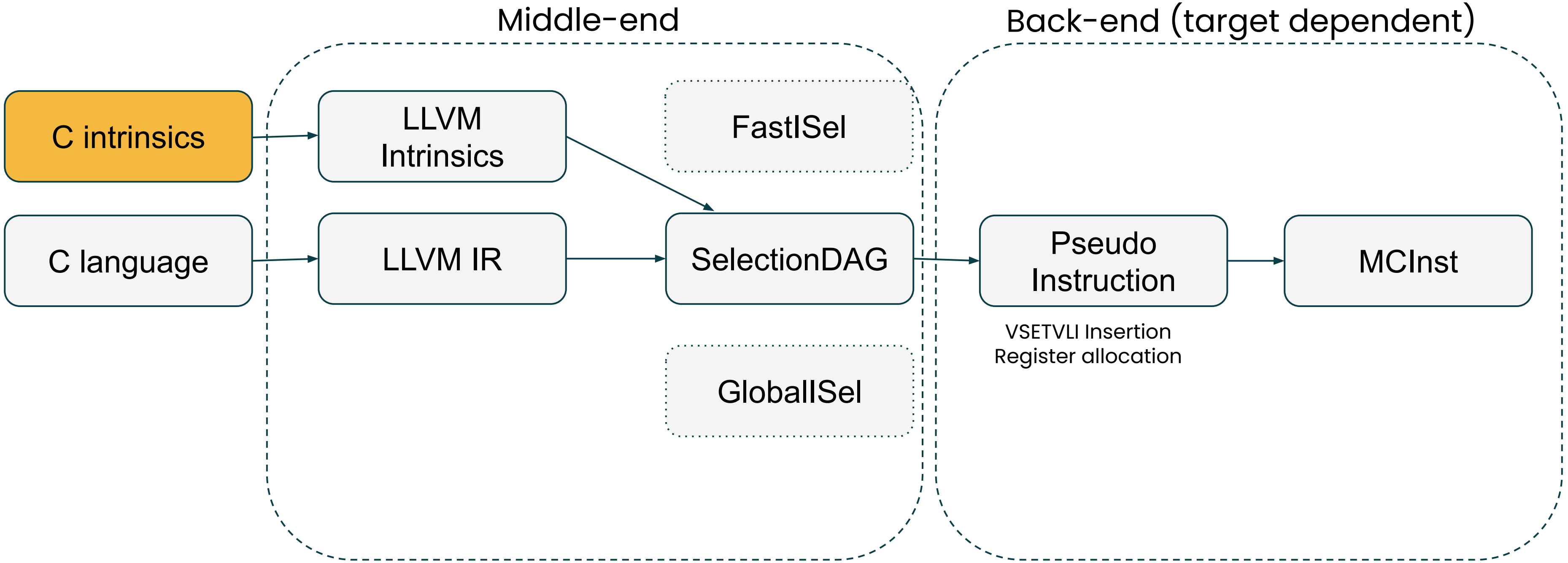
Case study: The RISC-V vector intrinsics

Workflow for RISC-V vector



Case study: The RISC-V vector intrinsics

Workflow for RISC-V vector



Case study: The RISC-V vector intrinsics

Defining intrinsic types for RISC-V vector - Creating own `RISCVVTypes.def`

```
/// clang/include/clang/Basic/RISCVVTypes.def
/// #define RVV_VECTOR_TYPE_INT(Name, Id, SingletonId, NumEls, ElBits, NF, IsSigned)

RVV_VECTOR_TYPE_INT("__rvv_int32mf2_t",RvvInt32mf2,RvvInt32mf2Ty,1, 32, 1, true)
RVV_VECTOR_TYPE_INT("__rvv_int32m1_t", RvvInt32m1, RvvInt32m1Ty, 2, 32, 1, true)
RVV_VECTOR_TYPE_INT("__rvv_int32m2_t", RvvInt32m2, RvvInt32m2Ty, 4, 32, 1, true)
RVV_VECTOR_TYPE_INT("__rvv_int32m4_t", RvvInt32m4, RvvInt32m4Ty, 8, 32, 1, true)
RVV_VECTOR_TYPE_INT("__rvv_int32m8_t", RvvInt32m8, RvvInt32m8Ty, 16, 32, 1, true)
```

```
#include <riscv_vector.h>

__rvv_int32m1_t foo(__rvv_int32m1_t va, __rvv_int32m1_t vb, size_t vl) {
return __riscv_vadd_vv_i32m1(va, vb, vl);
}
```

Case study: The RISC-V vector intrinsics

Defining intrinsic types for RISC-V vector - Creating own `RISCVVTypes.def`

```
/// clang/include/clang/Basic/RISCVVTypes.def
/// #define RVV_VECTOR_TYPE_INT(Name, Id, SingletonId, NumEls, ElBits, NF, IsSigned)

RVV_VECTOR_TYPE_INT("__rvv_int32mf2_t",RvvInt32mf2,RvvInt32mf2Ty,1, 32, 1, true)
RVV_VECTOR_TYPE_INT("__rvv_int32m1_t", RvvInt32m1, RvvInt32m1Ty, 2, 32, 1, true)
RVV_VECTOR_TYPE_INT("__rvv_int32m2_t", RvvInt32m2, RvvInt32m2Ty, 4, 32, 1, true)
RVV_VECTOR_TYPE_INT("__rvv_int32m4_t", RvvInt32m4, RvvInt32m4Ty, 8, 32, 1, true)
RVV_VECTOR_TYPE_INT("__rvv_int32m8_t", RvvInt32m8, RvvInt32m8Ty, 16, 32, 1, true)
```

Case study: The RISC-V vector intrinsics

Defining intrinsic types for RISC-V vector - Adding built-in type to `BuiltinType::Kind`

```
/// clang/include/clang/AST/Type.h

/// This class is used for builtin types like 'int'. Builtin
/// types are always canonical and have a literal name field.
class BuiltinType : public Type {
public:
  enum Kind {
// OpenCL image types
#define IMAGE_TYPE(ImgType, Id, SingletonId, Access, Suffix) Id,
#include "clang/Basic/OpenCLImageTypes.def"

// RVV Types
#define RVV_TYPE(Name, Id, SingletonId) Id,
#include "clang/Basic/RISCVVTypes.def"

// All other builtin types
#define BUILTIN_TYPE(Id, SingletonId) Id,
#define LAST_BUILTIN_TYPE(Id) LastKind = Id
#include "clang/AST/BuiltinTypes.def"
  };
  /* ... */
}
```

Case study: The RISC-V vector intrinsics

Defining intrinsic types for RISC-V vector - Registering singleton in `ASTContext::InitBuiltTypes`

```
/// clang/include/clang/Basic/RISCVVTypes.def
/// #define RVV_VECTOR_TYPE_INT(Name, Id, SingletonId, NumEls, ElBits, NF, IsSigned)

RVV_VECTOR_TYPE_INT("__rvv_int32mf2_t",RvvInt32mf2,RvvInt32mf2Ty,1, 32, 1, true)
RVV_VECTOR_TYPE_INT("__rvv_int32m1_t", RvvInt32m1, RvvInt32m1Ty, 2, 32, 1, true)
RVV_VECTOR_TYPE_INT("__rvv_int32m2_t", RvvInt32m2, RvvInt32m2Ty, 4, 32, 1, true)
RVV_VECTOR_TYPE_INT("__rvv_int32m4_t", RvvInt32m4, RvvInt32m4Ty, 8, 32, 1, true)
RVV_VECTOR_TYPE_INT("__rvv_int32m8_t", RvvInt32m8, RvvInt32m8Ty, 16, 32, 1, true)
```

Case study: The RISC-V vector intrinsics

Defining intrinsic types in Clang - Registering singleton in `ASTContext::InitBuiltTypes`

```
/// clang/lib/AST/ASTContext.cpp

void ASTContext::InitBuiltinTypes(const TargetInfo &Target,
                                  const TargetInfo *AuxTarget) {

  // C99 6.2.5p19.
  InitBuiltinType(VoidTy, BuiltinType::Void);

  // C99 6.2.5p4.
  InitBuiltinType(SignedCharTy, BuiltinType::SChar);
  InitBuiltinType(ShortTy, BuiltinType::Short);
  InitBuiltinType(IntTy, BuiltinType::Int);
  InitBuiltinType(LongTy, BuiltinType::Long);
  InitBuiltinType(LongLongTy, BuiltinType::LongLong);

  if (Target.hasRISCVVTypes()) {
#define RVV_TYPE(Name, Id, SingletonId) \
  InitBuiltinType(SingletonId, BuiltinType::Id);
#include "clang/Basic/RISCVVTypes.def"
  }

}
```

Case study: The RISC-V vector intrinsics

Defining intrinsic types in Clang - Implement conversion to LLVM IR in `CodeGenTypes::ConvertTypes`

```
/// clang/lib/CodeGen/CodeGenTypes.cpp

/// ConvertType - Convert the specified type to its LLVM form.
llvm::Type *CodeGenTypes::ConvertType(QualType T) {
  T = Context.getCanonicalType(T);

  case BuiltinType::Bool:
    // Note that we always return bool as i1 for use as a scalar type.
    ResultType = llvm::Type::getInt1Ty(getLLVMContext());
    break;
#define RVV_TYPE(Name, Id, SingletonId) case BuiltinType::Id:
#include "clang/Basic/RISCVVTypes.def"
      {
        ASTContext::BuiltinVectorTypeInfo Info =
            Context.getBuiltinVectorTypeInfo(cast<BuiltinType>(Ty));
        return llvm::ScalableVectorType::get(ConvertType(Info.ElementType),
                                             Info.EC.getKnownMinValue() *
                                             Info.NumVectors);
      }

}
```

```
/// clang/lib/CodeGen/ASTContext.cpp

ASTContext::BuiltinVectorTypeInfo
ASTContext::getBuiltinVectorTypeInfo(const BuiltinType *Ty) const {

#define RVV_VECTOR_TYPE_INT(Name, Id, SingletonId, NumEls, ElBits, NF, \
                            IsSigned) \
  case BuiltinType::Id: \
    return {getIntTypeForBitwidth(ElBits, IsSigned), \
            llvm::ElementCount::getScalable(NumEls), NF};
#define RVV_VECTOR_TYPE_FLOAT(Name, Id, SingletonId, NumEls, ElBits, NF) \
  case BuiltinType::Id: \
    return {ElBits == 16 ? Float16Ty : (ElBits == 32 ? FloatTy : DoubleTy), \
            llvm::ElementCount::getScalable(NumEls), NF};
#define RVV_PREDICATE_TYPE(Name, Id, SingletonId, NumEls) \
  case BuiltinType::Id: \
    return {BoolTy, llvm::ElementCount::getScalable(NumEls), 1};
#include "clang/Basic/RISCVVTypes.def"
}
```

Case study: The RISC-V vector intrinsics

Defining intrinsic types in Clang - Implement conversion to LLVM IR in `CodeGenTypes::ConvertTypes`

```
/// clang/lib/CodeGen/CodeGenTypes.cpp

/// ConvertType - Convert the specified type to its LLVM form.
llvm::Type *CodeGenTypes::ConvertType(QualType T) {
  T = Context.getCanonicalType(T);

  case BuiltinType::Bool:
    // Note that we always return bool as i1 for use as a scalar type.
    ResultType = llvm::Type::getInt1Ty(getLLVMContext());
    break;
#define RVV_TYPE(Name, Id, SingletonId) case BuiltinType::Id:
#include "clang/Basic/RISCVVTypes.def"
      {
        ASTContext::BuiltinVectorTypeInfo Info =
            Context.getBuiltinVectorTypeInfo(cast<BuiltinType>(Ty));
        return llvm::ScalableVectorType::get(ConvertType(Info.ElementType),
                                             Info.EC.getKnownMinValue() *
                                             Info.NumVectors);
      }

}
```

```
define <vscale x 2 x i32>
  @foo(<vscale x 2 x i32> %a, <vscale x 2 x i32> %b,
       i64 noundef %vl) {
entry:
  %0 = call <vscale x 2 x i32>
  @llvm.riscv.vadd.nxv2i32.nxv2i32.i64(<vscale x 2 x i32> poison,
                                       <vscale x 2 x i32> %a,
                                       <vscale x 2 x i32> %b,
                                       i64 %vl)

  ret <vscale x 2 x i32> %0
}
```

Case study: The RISC-V vector intrinsics

Defining intrinsics in Clang - Initial approach declaring builtins

```
/// clang/include/clang/Basic/Builtins.def

// Standard libc/libm functions:
BUILTIN(__builtin_atan2 , "ddd" , "Fne")
BUILTIN(__builtin_atan2f, "fff" , "Fne")
BUILTIN(__builtin_atan2l, "LdLdLd", "Fne")
BUILTIN(__builtin_atan2f128, "LLdLLdLLd", "Fne")
BUILTIN(__builtin_abs , "ii" , "ncF")
```

```
vint16m1_t __builtin_rvv_vadd_vv_i16m1_vl(vint16m1_t, vint16m1_t, size_t);
vint32m1_t __builtin_rvv_vadd_vv_i32m1_vl(vint32m1_t, vint32m1_t, size_t);
```

```
/// Initial approach in D93446
/// clang/include/clang/Basic/BuiltinsRISCV.def

RISCVV_BUILTIN(__builtin_rvv_vadd_vv_i16m1_vl, "q4Ssq4Ssq4Ssz", "n")
RISCVV_BUILTIN(__builtin_rvv_vadd_vv_i32m1_vl, "q2Siq2Siq2Siz", "n")
```

Case study: The RISC-V vector intrinsics

Defining intrinsics in Clang - Current approach to enumerate through variations

```
/// clang/include/clang/Basic/riscv_vector.td

defm vadd : RVVIntBinBuiltinSet;

multiclass RVVIntBinBuiltinSet
    : RVVSignedBinBuiltinSet,
      RVVUnsignedBinBuiltinSet;
```

```
multiclass RVVSignedBinBuiltinSet
    : RVVOutOp1BuiltinSet<NAME, "csil",
                          [["vv", "v", "vvv"],
                           ["vx", "v", "vve"]]>;


multiclass RVVUnsignedBinBuiltinSet
    : RVVOutOp1BuiltinSet<NAME, "csil",
                          [["vv", "Uv", "UvUvUv"],
                           ["vx", "Uv", "UvUvUe"]]>;
```

```
/// clang/utils/TableGen/RISCVVEmitter.cpp

void RVVEmitter::createRVVIntrinsics(

  // Create Intrinsics for each type and LMUL.
  for (char I : TypeRange) {
    for (int Log2LMUL : Log2LMULList) {
      BasicType BT = ParseBasicType(I);
      std::optional<RVVTypes> Types =
          TypeCache.computeTypes(BT, Log2LMUL, NF, Prototype);
      // Ignored to create new intrinsic if there are any illegal types.
      if (!Types)
        continue;
```

Case study: The RISC-V vector intrinsics

Semantic checks - Check function call parameters

```
/// clang/lib/Sema/SemaChecking.cpp

bool Sema::CheckTSBuiltinFunctionCall(const TargetInfo &TI, unsigned BuiltinID,
                                      CallExpr *TheCall) {

  switch (TI.getTriple().getArch()) {
  default:
    // Some builtins don't require additional checking, so just consider these
    // acceptable.
    return false;

  case llvm::Triple::riscv32:
  case llvm::Triple::riscv64:
    return CheckRISCVBuiltinFunctionCall(TI, BuiltinID, TheCall);


}
```

```
bool Sema::CheckRISCVBuiltinFunctionCall(const TargetInfo &TI,
                                         unsigned BuiltinID,
                                         CallExpr *TheCall) {

  switch (BuiltinID) {
  default:
    break;
  // Check if feature is missing for the builtin function call
  case RISCVVector::BI__builtin_rvv_vsmul_vv_tumu:
  case RISCVVector::BI__builtin_rvv_vsmul_vx_tumu: {
    bool RequireV = false;
    for (unsigned ArgNum = 0; ArgNum < TheCall->getNumArgs(); ++ArgNum)
      RequireV |= TheCall->getArg(ArgNum)->getType()->isRVVType(
          /* Bitwidth */ 64, /* IsFloat */ false);

    if (RequireV && !TI.hasFeature("v"))
      return Diag(TheCall->getBeginLoc(),
                  diag::err_riscv_builtin_requires_extension)
             << /* IsExtension */ false << TheCall->getSourceRange()
             << "v";

    break;
  }
}
```

Case study: The RISC-V vector intrinsics

Semantic checks - Check function call parameters

```
/// clang/lib/Sema/SemaChecking.cpp

bool Sema::CheckTSBuiltinFunctionCall(const TargetInfo &TI, unsigned BuiltinID,
                                       CallExpr *TheCall) {

  switch (TI.getTriple().getArch()) {
  default:
    // Some builtins don't require additional checking, so just consider these
    // acceptable.
    return false;

  case llvm::Triple::riscv32:
  case llvm::Triple::riscv64:
    return CheckRISCVBuiltinFunctionCall(TI, BuiltinID, TheCall);

}
```

```
bool Sema::CheckRISCVBuiltinFunctionCall(const TargetInfo &TI,
                                         unsigned BuiltinID,
                                         CallExpr *TheCall) {

  switch (BuiltinID) {
  default:
    break;

  // Check if parameters that require constants
  case RISCVVector::BI__builtin_rvv_vsm3c_vi_tu:
  case RISCVVector::BI__builtin_rvv_vsm3c_vi: {
    QualType Op1Type = TheCall->getArg(0)->getType();
    return CheckInvalidVLENandLMUL(TI, TheCall, *this, Op1Type, 256) ||
           SemaBuiltinConstantArgRange(TheCall, 2, 0, 31);
  }

}
```

Potential improvement: Reduce boilerplates by allowing constraints to be expressed in built-in definitions and handle them gracefully here.

Case study: The RISC-V vector intrinsics

Semantic checks - Check type support for variable declaration

```
/// clang/lib/Sema/SemaDecl.cpp

void Sema::CheckVariableDeclarationType(VarDecl *NewVD) {

  QualType T = NewVD->getType();


  if (T->isRVVType())
    checkRVVTypeSupport(T, NewVD->getLocation(), cast<Decl>(CurContext));


}
```

```
/// clang/lib/Sema/SemaDecl.cpp

void Sema::checkRVVTypeSupport(QualType Ty, SourceLocation Loc, Decl *D) {
  const TargetInfo &TI = Context.getTargetInfo();
  // (ELEN, LMUL) pairs of (8, mf8), (16, mf4), (32, mf2), (64, m1) requires at
  // least zve64x
  if ((Ty->isRVVType(/* Bitwidth */ 64, /* IsFloat */ false) ||
       Ty->isRVVType(/* ElementCount */ 1)) &&
      !TI.hasFeature("zve64x"))
    Diag(Loc, diag::err_riscv_type_requires_extension, D) << Ty << "zve64x";
```

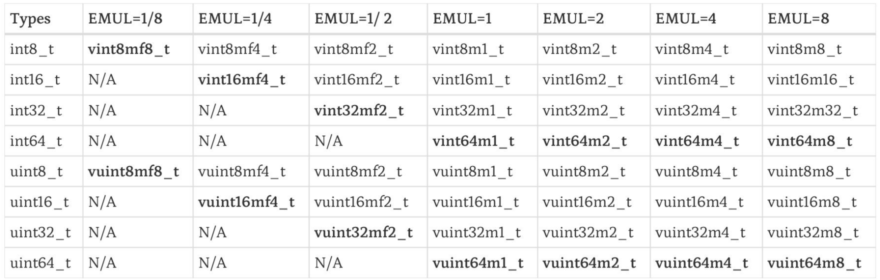
Case study: The RISC-V vector intrinsics

Speeding up overwhelming amounts of variants

```
vadd.vv vd, vs2, vs1
```

SEW ∈ {8, 16, 32, 64} X LMUL ∈ {⅛, ¼, ½, 1, 2, 4, 8}

```
vint16m1_t __riscv_vadd_vv_i16m1 (vint16m1_t op1, vint16m1_t op2, size_t vl);
vint16m2_t __riscv_vadd_vv_i16m2 (vint16m2_t op1, vint16m2_t op2, size_t vl);
vint16m4_t __riscv_vadd_vv_i16m4 (vint16m4_t op1, vint16m4_t op2, size_t vl);
vint16m8_t __riscv_vadd_vv_i16m8 (vint16m8_t op1, vint16m8_t op2, size_t vl);
vint32m1_t __riscv_vadd_vv_i32m1 (vint32m1_t op1, vint32m1_t op2, size_t vl);
vint32m2_t __riscv_vadd_vv_i32m2 (vint32m2_t op1, vint32m2_t op2, size_t vl);
vint32m4_t __riscv_vadd_vv_i32m4 (vint32m4_t op1, vint32m4_t op2, size_t vl);
vint32m8_t __riscv_vadd_vv_i32m8 (vint32m8_t op1, vint32m8_t op2, size_t vl);
```



Types	EMUL=1/8	EMUL=1/4	EMUL=1/ 2	EMUL=1	EMUL=2	EMUL=4	EMUL=8
int8_t	**vint8mf8_t**	vint8mf4_t	vint8mf2_t	vint8m1_t	vint8m2_t	vint8m4_t	vint8m8_t
int16_t	N/A	**vint16mf4_t**	vint16mf2_t	vint16m1_t	vint16m2_t	vint16m4_t	vint16m16_t
int32_t	N/A	N/A	**vint32mf2_t**	vint32m1_t	vint32m2_t	vint32m4_t	vint32m32_t
int64_t	N/A	N/A	N/A	**vint64m1_t**	**vint64m2_t**	**vint64m4_t**	**vint64m8_t**
uint8_t	**vuint8mf8_t**	vuint8mf4_t	vuint8mf2_t	vuint8m1_t	vuint8m2_t	vuint8m4_t	vuint8m8_t
uint16_t	N/A	**vuint16mf4_t**	vuint16mf2_t	vuint16m1_t	vuint16m2_t	vuint16m4_t	vuint16m8_t
uint32_t	N/A	N/A	**vuint32mf2_t**	vuint32m1_t	vuint32m2_t	vuint32m4_t	vuint32m8_t
uint64_t	N/A	N/A	N/A	**vuint64m1_t**	**vuint64m2_t**	**vuint64m4_t**	**vuint64m8_t**

Table 1. Integer types

Table from the RVV C intrinsics specification

Case study: The RISC-V vector intrinsics

Speeding up overwhelming amounts of variants

```
/// clang/include/clang/Basic/Builtins.def

// Standard libc/libm functions:
BUILTIN(__builtin_atan2 , "ddd" , "Fne")
BUILTIN(__builtin_atan2f, "fff" , "Fne")
BUILTIN(__builtin_atan2l, "LdLdLd", "Fne")
BUILTIN(__builtin_atan2f128, "LLdLLdLLd", "Fne")
BUILTIN(__builtin_abs , "ii" , "ncF")
```

```
vint16m1_t __builtin_rvv_vadd_vv_i16m1_vl(vint16m1_t, vint16m1_t, size_t);
vint32m1_t __builtin_rvv_vadd_vv_i32m1_vl(vint32m1_t, vint32m1_t, size_t);
```

```
/// Initial approach in D93446
/// clang/include/clang/Basic/BuiltinsRISCV.def

RISCVV_BUILTIN(__builtin_rvv_vadd_vv_i16m1_vl, "q4Ssq4Ssq4Ssz", "n")
RISCVV_BUILTIN(__builtin_rvv_vadd_vv_i32m1_vl, "q2Siq2Siq2Siz", "n")
```

Case study: The RISC-V vector intrinsics

Speeding up overwhelming amounts of variants

Aliasing different interfaces to the same built-in reduces number of built-ins needed.

```
/// Initial approach in D93446
/// clang/include/clang/Basic/BuiltinsRISCV.def

RISCVV_BUILTIN(__builtin_rvv_vadd_vv,"", "n")
```

```
static __inline__
__attribute__((__clang_builtin_alias__(__builtin_rvv_vadd_vv)))
vint32m1_t vadd_vv_i32m1(vint32m1_t, vint32m1_t, size_t);

static __inline__ __attribute__((__overloadable__))
__attribute__((__clang_builtin_alias__(__builtin_rvv_vadd_vv)))
vint8m1_t vadd(vint8m1_t, vint8m1_t, size_t);
```

However, we have **> 200K** intrinsic interfaces, this leads:

- Larger clang binary
- Require more run-time memory during compilation

Case study: The RISC-V vector intrinsics

Speeding up overwhelming amounts of variants

The latest approach in the compiler lazily constructs built-in function table at the first symbol lookup of the intrinsic.

```
/// clang/lib/Sema/SemaRISCVVectorLookup.cpp

struct RVVIntrinsicDef {
  /// Full function name with suffix, e.g. vadd_vv_i32m1.
  std::string Name;

  /// Overloaded function name, e.g. vadd.
  std::string OverloadName;

  /// Mapping to which clang built-in function,
  /// e.g. __builtin_rvv_vadd.
  std::string BuiltinName;

  /// Function signature, first element is return type.
  RVVTypes Signature;
};
```

```
/// clang/lib/Sema/SemaLookup.cpp

/// Lookup a builtin function, when name lookup would otherwise
/// fail.
bool Sema::LookupBuiltin(LookupResult &R) {
Sema::LookupNameKind NameKind = R.getLookupKind();
  // If we didn't find a use of this identifier, and if the identifier
  // corresponds to a compiler builtin, create the decl object for the builtin
  // now, injecting it into translation unit scope, and return it.
  if (NameKind == Sema::LookupOrdinaryName ||
      NameKind == Sema::LookupRedeclarationWithLinkage) {
    IdentifierInfo *II = R.getLookupName().getAsIdentifierInfo();
    if (II) {
      if (DeclareRISCVVBuiltins || DeclareRISCVSiFiveVectorBuiltins) {
        if (!RVIntrinsicManager)
          RVIntrinsicManager = CreateRISCVIntrinsicManager(*this);

        RVIntrinsicManager->InitIntrinsicList();

        if (RVIntrinsicManager->CreateIntrinsicIfFound(R, II, PP))
          return true;
      }
    }
  }
}
```

Lookup table is constructed in compile time

Case study: The RISC-V vector intrinsics

Speeding up overwhelming amounts of variants

The latest approach in the compiler lazily constructs built-in function table at the first symbol lookup of the intrinsic.

```
/// clang/lib/Sema/SemaRISCVVectorLookup.cpp

struct RVVIntrinsicDef {
  /// Full function name with suffix, e.g. vadd_vv_i32m1.
  std::string Name;

  /// Overloaded function name, e.g. vadd.
  std::string OverloadName;

  /// Mapping to which clang built-in function,
  /// e.g. __builtin_rvv_vadd.
  std::string BuiltinName;

  /// Function signature, first element is return type.
  RVVTypes Signature;
};
```

```
/// clang/lib/Sema/SemaRISCVVectorLookup.cpp

static const PrototypeDescriptor RVVSignatureTable[] = {
#define DECL_SIGNATURE_TABLE
#include "clang/Basic/riscv_vector_builtin_sema.inc"
#undef DECL_SIGNATURE_TABLE
};

static const RVVIntrinsicRecord RVVIntrinsicRecords[] = {
#define DECL_INTRINSIC_RECORDS
#include "clang/Basic/riscv_vector_builtin_sema.inc"
#undef DECL_INTRINSIC_RECORDS
};
```

Table built during building `clang`

```
/// $(BUILD)/tools/clang/include/clang/Basic/riscv_vector_builtin_sema.inc

{"vadd_vv",nullptr,957,23,0,3,1,0,0,15,127,1,1,1,1,1,1,0,0,1,2,},

// At [957] of RVVSignatureTable

PrototypeDescriptor(/* BaseTypeModifier = Vector */ 2, 0, 0),
PrototypeDescriptor(/* BaseTypeModifier = Vector */ 2, 0, 0),
PrototypeDescriptor(/* BaseTypeModifier = Vector */ 2, 0, 0),
```

Case study: The RISC-V vector intrinsics

Speeding up overwhelming amounts of variants

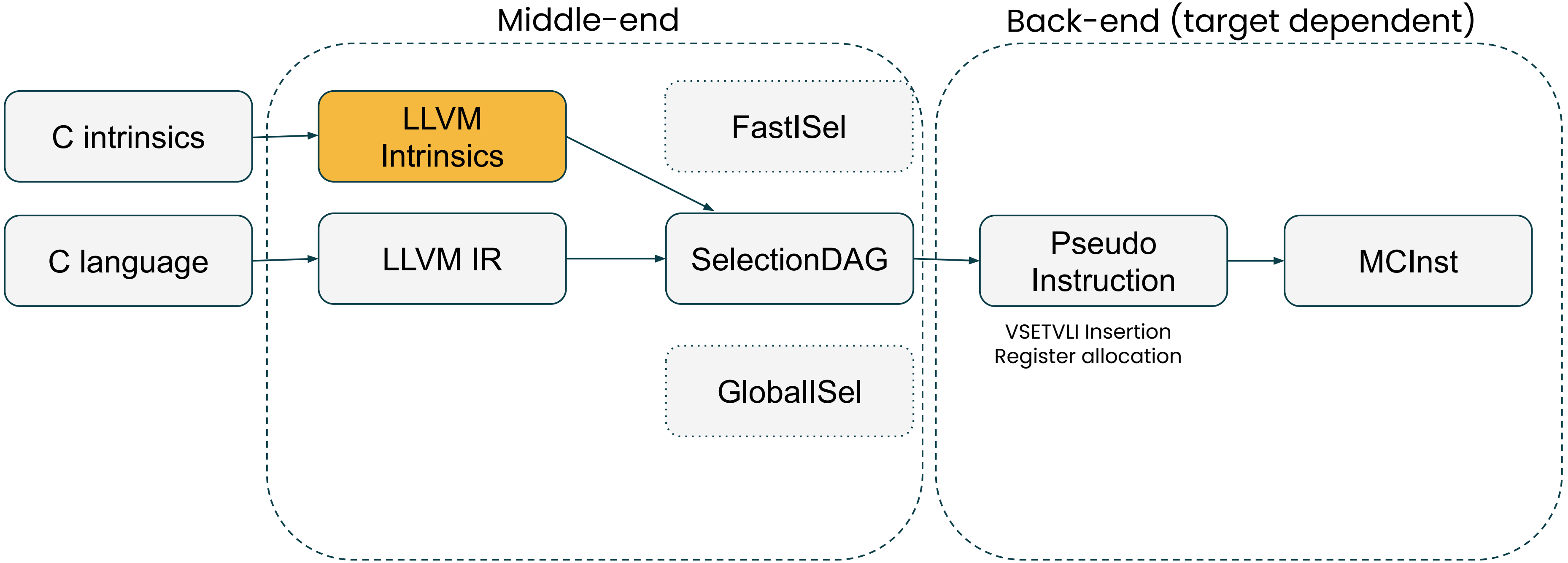
The latest approach in the compiler lazily constructs built-in function table at the first symbol lookup of the intrinsic.

```
/// clang/lib/Sema/SemaRISCVVectorLookup.cpp

struct RVVIntrinsicDef {
  /// Full function name with suffix, e.g. vadd_vv_i32m1.
  std::string Name;

  /// Overloaded function name, e.g. vadd.
  std::string OverloadName;

  /// Mapping to which clang built-in function,
  /// e.g. __builtin_rvv_vadd.
  std::string BuiltinName;

  /// Function signature, first element is return type.
  RVVTypes Signature;
};
```

```
// At [957] of RVVSignatureTable

PrototypeDescriptor(/* BaseTypeModifier = Vector */ 2, 0, 0),
PrototypeDescriptor(/* BaseTypeModifier = Vector */ 2, 0, 0),
PrototypeDescriptor(/* BaseTypeModifier = Vector */ 2, 0, 0),
```

Case study: The RISC-V vector intrinsics

Workflow for RISC-V vector



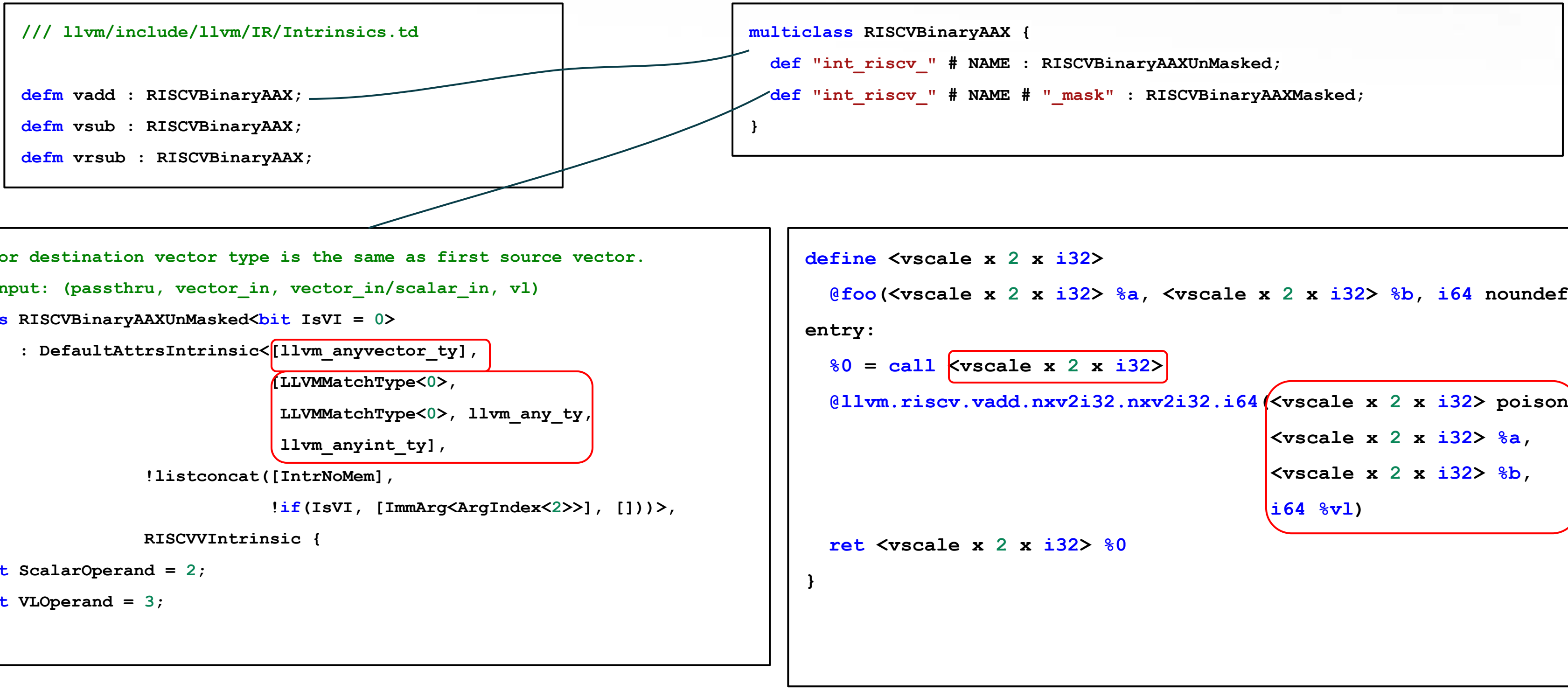
Case study: The RISC-V vector intrinsics

Declaring the intrinsics in LLVM IR

```
/// llvm/include/llvm/IR/Intrinsics.td

// Intrinsic class - This is used to define one LLVM intrinsic. The name of the
// intrinsic definition should start with "int_", then match the LLVM intrinsic
// name with the "llvm." prefix removed, and all "."s turned into "_"s. For
// example, llvm.bswap.i16 -> int_bswap_i16.
class Intrinsic<list<LLVMType> ret_types,
                list<LLVMType> param_types = [],
                list<IntrinsicProperty> intr_properties = [],
                string name = "",
                list<SDNodeProperty> sd_properties = [],
                bit disable_default_attributes = true> : SDPatternOperator {
  string LLVMName = name;
  string TargetPrefix = ""; // Set to a prefix for target-specific intrinsics.
  list<LLVMType> RetTypes = ret_types;
  list<LLVMType> ParamTypes = param_types;
  list<IntrinsicProperty> IntrProperties = intr_properties;
  let Properties = sd_properties;

  /* ... */
}
```

Case study: The RISC-V vector intrinsics

Declaring the intrinsics in LLVM IR - `IntrinsicsRISCV.td`



```
/// llvm/include/llvm/IR/Intrinsics.td


defm vadd : RISCVBinaryAAX;
defm vsub : RISCVBinaryAAX;
defm vrsub : RISCVBinaryAAX;
```

```
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<bit IsVI = 0>
      : DefaultAttrsIntrinsic<[llvm_anyvector_ty],
                              [LLVMMatchType<0>,
                               LLVMMatchType<0>, llvm_any_ty,
                               llvm_anyint_ty],
                  !listconcat([IntrNoMem],
                              !if(IsVI, [ImmArg<ArgIndex<2>>], [])),
                  RISCVVIntrinsic {
  let ScalarOperand = 2;
  let VLOperand = 3;
}
```

```
define <vscale x 2 x i32>
  @foo(<vscale x 2 x i32> %a, <vscale x 2 x i32> %b, i64 noundef %vl) {
entry:
  %0 = call <vscale x 2 x i32>
  @llvm.riscv.vadd.nxv2i32.nxv2i32.i64(<vscale x 2 x i32> poison,
                                       <vscale x 2 x i32> %a,
                                       <vscale x 2 x i32> %b,
                                       i64 %vl)

  ret <vscale x 2 x i32> %0
}
```

Case study: The RISC-V vector intrinsics

Code gen to LLVM IR under `CGBuiltin.cpp`

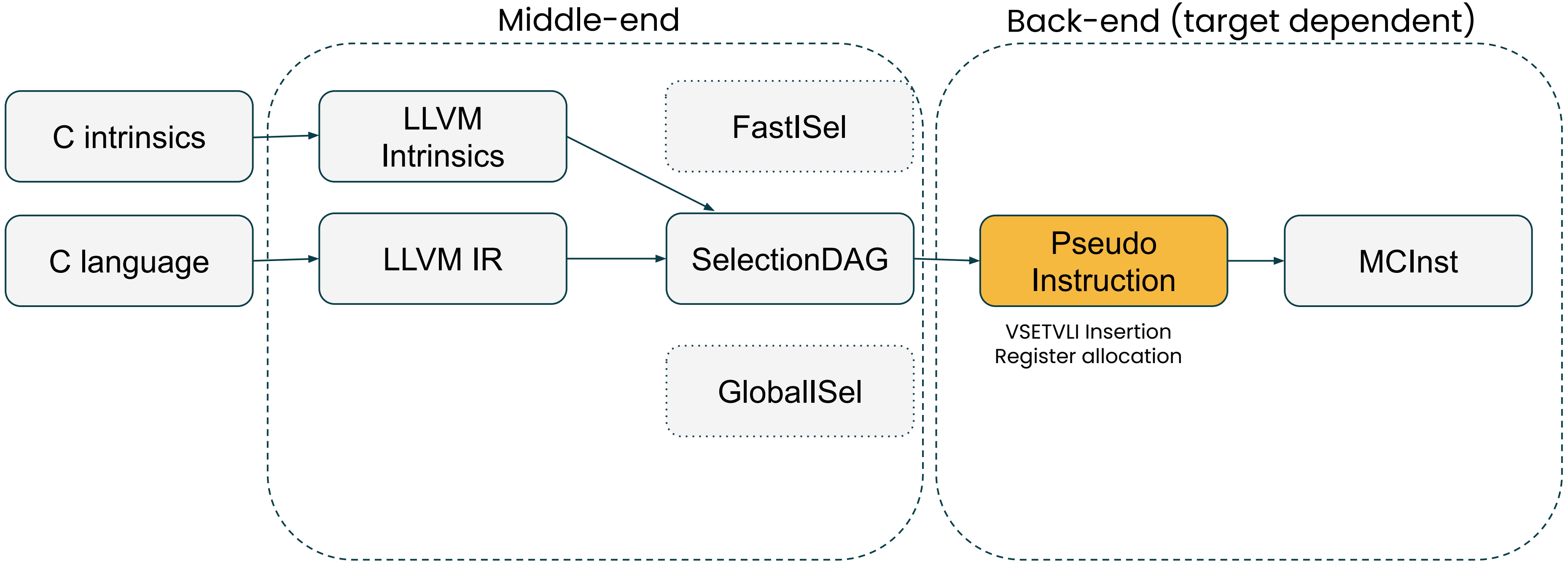
```
/// clang/lib/CodeGen/CGBuiltin.cpp

Value *CodeGenFunction::EmitRISCVBuiltinExpr(unsigned BuiltinID,
                                             const CallExpr *E,
                                             ReturnValueSlot ReturnValue) {

  SmallVector<Value *, 4> Ops;
  llvm::Type *ResultType = ConvertType(E->getType());


  Intrinsic::ID ID;
  llvm::SmallVector<llvm::Type *, 2> IntrinsicTypes;

  // Vector builtins are handled from here.
  #include "clang/Basic/riscv_vector_builtin_cg.inc"


  llvm::Function *F = CGM.getIntrinsic(ID, IntrinsicTypes);
  return Builder.CreateCall(F, Ops, "");

}
```

```
llvm::SmallVector<llvm::Type *, 2> IntrinsicTypes;
switch (BuiltinID) {

case RISCVVector::BI__builtin_rvv_vadd_vv_tu:
case RISCVVector::BI__builtin_rvv_vadd_vx_tu:
  ID = Intrinsic::riscv_vadd;
  PolicyAttrs = 2;
  IntrinsicTypes = {ResultType, Ops[2]->getType(), Ops.back()->getType()};
  break;
}
```

```
/// clang/utils/TableGen/RISCVVEmitter.cpp

void RVVEmitter::createCodeGen(raw_ostream &OS);
void emitCodeGenSwitchBody(const RVVIntrinsic *RVVI, raw_ostream &OS)
```

Case study: The RISC-V vector intrinsics

Workflow for RISC-V vector



Case study: The RISC-V vector intrinsics

Pseudo Instruction: Preserving the vtype status in the back-end

```
/// llvm/lib/Target/RISCV/RISCVInstrInfo.h

namespace RISCVVPseudosTable {

struct PseudoInfo {
  uint16_t Pseudo;
  uint16_t BaseInstr;
};

#define GET_RISCVVPseudosTable_DECL
#include "RISCVGenSearchableTables.inc"

} // end namespace RISCVVPseudosTable
```

```
{ PseudoVADD_VV_M1, VADD_VV, 0x0, 0x0 }, // 161
{ PseudoVADD_VV_M1_MASK, VADD_VV, 0x0, 0x0 }, // 162
{ PseudoVADD_VV_M2, VADD_VV, 0x1, 0x0 }, // 163
{ PseudoVADD_VV_M2_MASK, VADD_VV, 0x1, 0x0 }, // 164
{ PseudoVADD_VV_M4, VADD_VV, 0x2, 0x0 }, // 165
{ PseudoVADD_VV_M4_MASK, VADD_VV, 0x2, 0x0 }, // 166
{ PseudoVADD_VV_M8, VADD_VV, 0x3, 0x0 }, // 167
{ PseudoVADD_VV_M8_MASK, VADD_VV, 0x3, 0x0 }, // 168
{ PseudoVADD_VV_MF8, VADD_VV, 0x5, 0x0 }, // 169
{ PseudoVADD_VV_MF8_MASK, VADD_VV, 0x5, 0x0 }, // 170
```

```
/// llvm/lib/Target/RISCV/RISCVInstrInfoVPseudo.td

// This class holds the record of the RISCVVPseudoTable below.
// This represents the information we need in codegen for each pseudo.
// The definition should be consistent with `struct PseudoInfo` in
// RISCVInstrInfo.h.
class RISCVVPseudo {
  Pseudo Pseudo = !cast<Pseudo>(NAME); // Used as a key.
  Instruction BaseInstr = !cast<Instruction>(PseudoToVInst<NAME>.VInst);
  // SEW = 0 is used to denote that the Pseudo is not SEW specific (or unknown).
  bits<8> SEW = 0;
  bit NeedBeInPseudoTable = 1;
}

// The actual table.
def RISCVVPseudosTable : GenericTable {
  let FilterClass = "RISCVVPseudo";
  let FilterClassField = "NeedBeInPseudoTable";
  let CppTypeName = "PseudoInfo";
  let Fields = [ "Pseudo", "BaseInstr" ];
  let PrimaryKey = [ "Pseudo" ];
  let PrimaryKeyName = "getPseudoInfo";
  let PrimaryKeyEarlyOut = true;
}
```

Case study: The RISC-V vector intrinsics

Pseudo Instruction: Preserving the vtype status in the back-end

```
/// llvm/lib/Target/RISCV/RISCVInstrInfoVPseudo.td

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:$policy),
                                   []>,
                            RISCVVPseudo {
  let mayLoad = 0;
  let mayStore = 0;
  let hasSideEffects = 0;
  let Constraints = !interleave([Constraint, "$rd = $merge"], ",");
  let HasVLOp = 1;
  let HasSEWOp = 1;
  let HasVecPolicyOp = 1;
}
```

```
/// llvm/lib/Target/RISCV/RISCVInstrInfoVPseudo.td

// This class holds the record of the RISCVVPseudoTable below.
// This represents the information we need in codegen for each pseudo.
// The definition should be consistent with `struct PseudoInfo` in
// RISCVInstrInfo.h.
class RISCVVPseudo {
  Pseudo Pseudo = !cast<Pseudo>(NAME); // Used as a key.
  Instruction BaseInstr = !cast<Instruction>(PseudoToVInst<NAME>.VInst);
  // SEW = 0 is used to denote that the Pseudo is not SEW specific (or unknown).
  bits<8> SEW = 0;
  bit NeedBeInPseudoTable = 1;
}

// The actual table.
def RISCVVPseudosTable : GenericTable {
  let FilterClass = "RISCVVPseudo";
  let FilterClassField = "NeedBeInPseudoTable";
  let CppTypeName = "PseudoInfo";
  let Fields = [ "Pseudo", "BaseInstr" ];
  let PrimaryKey = [ "Pseudo" ];
  let PrimaryKeyName = "getPseudoInfo";
  let PrimaryKeyEarlyOut = true;
}
```

Case study: The RISC-V vector intrinsics

Pseudo Instruction: Preserving the vtype status in the back-end

```
/// llvm/lib/Target/RISCV/RISCVInstrInfo.h

namespace RISCVVPseudosTable {

struct PseudoInfo {
  uint16_t Pseudo;
  uint16_t BaseInstr;
};

#define GET_RISCVVPseudosTable_DECL
#include "RISCVGenSearchableTables.inc"

} // end namespace RISCVVPseudosTable
```

```
{ PseudoVADD_VV_M1, VADD_VV, 0x0, 0x0 }, // 161
{ PseudoVADD_VV_M1_MASK, VADD_VV, 0x0, 0x0 }, // 162
{ PseudoVADD_VV_M2, VADD_VV, 0x1, 0x0 }, // 163
{ PseudoVADD_VV_M2_MASK, VADD_VV, 0x1, 0x0 }, // 164
{ PseudoVADD_VV_M4, VADD_VV, 0x2, 0x0 }, // 165
{ PseudoVADD_VV_M4_MASK, VADD_VV, 0x2, 0x0 }, // 166
{ PseudoVADD_VV_M8, VADD_VV, 0x3, 0x0 }, // 167
{ PseudoVADD_VV_M8_MASK, VADD_VV, 0x3, 0x0 }, // 168
{ PseudoVADD_VV_MF8, VADD_VV, 0x5, 0x0 }, // 169
{ PseudoVADD_VV_MF8_MASK, VADD_VV, 0x5, 0x0 }, // 170
```

```
/// llvm/lib/Target/RISCV/RISCVAsmPrinter.cpp

static bool lowerRISCVVMachineInstrToMCInst(const MachineInstr *MI,
                                            MCInst &OutMI) {
  const RISCVVPseudosTable::PseudoInfo *RVV =
      RISCVVPseudosTable::getPseudoInfo(MI->getOpcode());
  if (!RVV)
    return false;

  OutMI.setOpcode(RVV->BaseInstr);

  /* ... */
```

Case study: The RISC-V vector intrinsics

Pseudo Instruction: Preserving the vtype status in the back-end

```
/// llvm/lib/Target/RISCV/RISCVInstrInfoVPseudo.td

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:$policy),
                                   []>,
                            RISCVVPseudo {
  let mayLoad = 0;
  let mayStore = 0;
  let hasSideEffects = 0;
  let Constraints = !interleave([Constraint, "$rd = $merge"], ",");
  let HasVLOp = 1;
  let HasSEWOp = 1;
  let HasVecPolicyOp = 1;
}
```

```
/// llvm/lib/Target/RISCV/RISCVInstrInfoVPseudo.td

multiclass VPseudoBinary<VReg RetClass,
                         VReg Op1Class,
                         DAGOperand Op2Class,
                         LMULInfo MInfo,
                         string Constraint = "",
                         int sew = 0> {
  let VLMul = MInfo.value, SEW=sew in {
    defvar suffix = !if(sew, "_" # MInfo.MX # "_E" # sew, "_" # MInfo.MX);
    def suffix : VPseudoBinaryNoMaskTU<RetClass, Op1Class, Op2Class,
                                       Constraint>;
    def suffix # "_MASK" : VPseudoBinaryMaskPolicy<RetClass, Op1Class, Op2Class,
                                                   Constraint>,
                           RISCVMaskedPseudo<MaskIdx=3>;
  }
}
```

Case study: The RISC-V vector intrinsics

Pseudo Instruction: Preserving the vtype status in the back-end

```
/// llvm/lib/Target/RISCV/RISCVInstrInfoVPseudo.td

multiclass VPseudoBinaryV_VV<LMULInfo m,
                             string Constraint = "",
                             int sew = 0> {
  defm _VV : VPseudoBinary<...>;
}
```

```
/// llvm/lib/Target/RISCV/RISCVInstrInfoVPseudo.td

multiclass VPseudoVSALU_VV_VX_VI<Operand ImmType = simm5,
                                 string Constraint = ""> {
  foreach m = MxList in {
    defvar mx = m.MX;
    defm "" : VPseudoBinaryV_VV<m, Constraint>,
              SchedBinary<...>;
    defm "" : VPseudoBinaryV_VX<m, Constraint>,
              SchedBinary<...>;
    defm "" : VPseudoBinaryV_VI<ImmType, m, Constraint>,
              SchedUnary<...>;
  }
}
```

```
/// llvm/lib/Target/RISCV/RISCVInstrInfoVPseudo.td

multiclass VPseudoBinary<VReg RetClass,
                         VReg Op1Class,
                         DAGOperand Op2Class,
                         LMULInfo MInfo,
                         string Constraint = "",
                         int sew = 0> {
  let VLMul = MInfo.value, SEW=sew in {
    defvar suffix = !if(sew, "_" # MInfo.MX # "_E" # sew, "_" # MInfo.MX);
    def suffix : VPseudoBinaryNoMaskTU<RetClass, Op1Class, Op2Class,
                                       Constraint>;
    def suffix # "_MASK" : VPseudoBinaryMaskPolicy<RetClass, Op1Class, Op2Class,
                                                   Constraint>,
                           RISCVMaskedPseudo<MaskIdx=3>;
  }
}
```

Case study: The RISC-V vector intrinsics

Pseudo Instruction: Preserving the vtype status in the back-end

```
/// llvm/lib/Target/RISCV/RISCVInstrInfoVPseudo.td

multiclass VPseudoBinaryV_VV<LMULInfo m,
                             string Constraint = "",
                             int sew = 0> {
  defm _VV : VPseudoBinary<...>;
}
```

```
/// llvm/lib/Target/RISCV/RISCVInstrInfoVPseudo.td

multiclass VPseudoVALU_VV_VX_VI<Operand ImmType = simm5,
                                string Constraint = ""> {
  foreach m = MxList in {
    defvar mx = m.MX;
    defm "" : VPseudoBinaryV_VV<m, Constraint>,
              SchedBinary<...>;
    defm "" : VPseudoBinaryV_VX<m, Constraint>,
              SchedBinary<...>;
    defm "" : VPseudoBinaryV_VI<ImmType, m, Constraint>,
              SchedUnary<...>;
  }
}
```

```
/// llvm/lib/Target/RISCV/RISCVInstrInfoVPseudo.td

//===----------------------------------------------------------------------===//
// 11.1. Vector Single-Width Integer Add and Subtract
//===----------------------------------------------------------------------===//
defm PseudoVADD  : VPseudoVALU_VV_VX_VI;
defm PseudoVSUB  : VPseudoVALU_VV_VX;
defm PseudoVRSUB : VPseudoVALU_VX_VI;
```

Case study: The RISC-V vector intrinsics

Pseudo Instruction: Preserving the vtype status in the back-end

```
/// llvm/lib/Target/RISCV/RISCVInstrInfoVPseudo.td

class VPatBinaryNoMaskTU<string intrinsic_name,
                         string inst,
                         ValueType result_type,
                         ValueType op1_type,
                         ValueType op2_type,
                         int sew,
                         VReg result_reg_class,
                         VReg op1_reg_class,
                         DAGOperand op2_kind> :
  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)>;
```

```
define <vscale x 2 x i32>
  @foo(<vscale x 2 x i32> %a, <vscale x 2 x i32> %b, i64 noundef %vl) {
entry:
  %0 = call <vscale x 2 x i32>
  @llvm.riscv.vadd.nxv2i32.nxv2i32.i64(<vscale x 2 x i32> poison,
                                       <vscale x 2 x i32> %a,
                                       <vscale x 2 x i32> %b,
                                       i64 %vl)
  ret <vscale x 2 x i32> %0
}
```

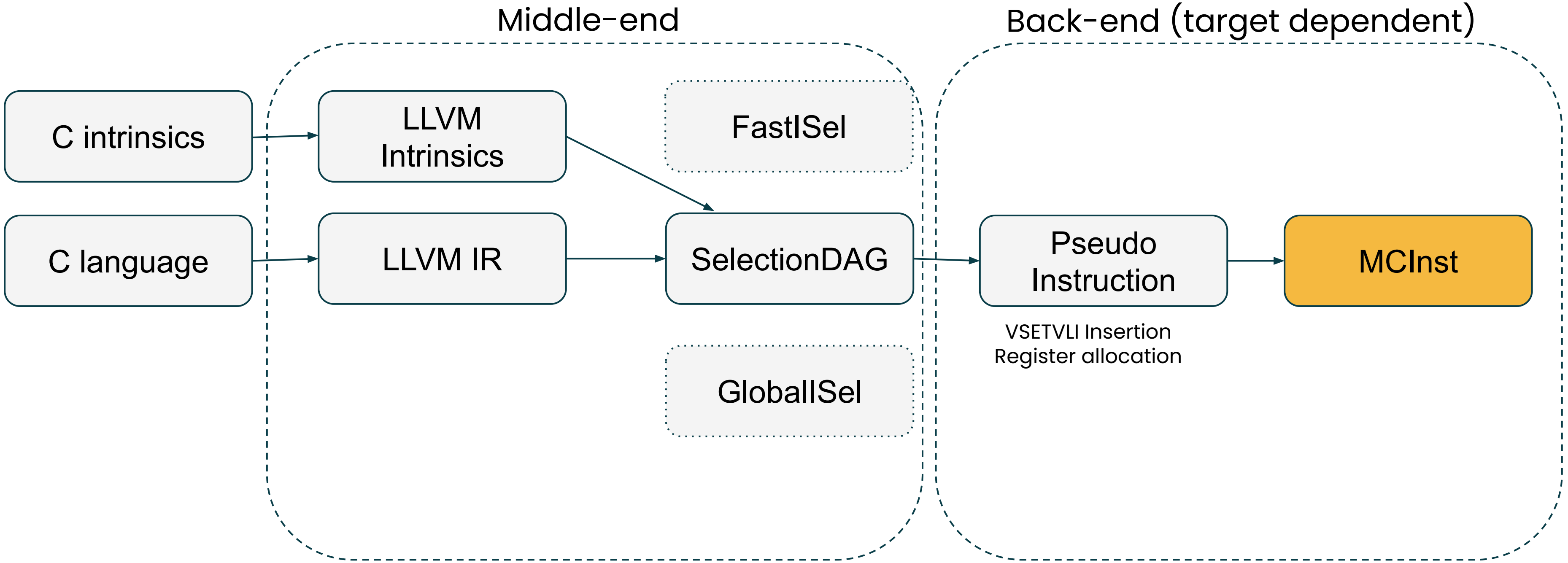
```
vr = PseudoVADD_VV_M1 undef %3:vr(tied-def 0), killed %0:vr, killed %1:vr,
                      $noreg, 5, 0, implicit $vl, implicit $vtype
```

SEW = 2^5 = 32

policy = TA_MA = 0

Case study: The RISC-V vector intrinsics

Workflow for RISC-V vector



Case study: The RISC-V vector intrinsics

Describing the machine instructions

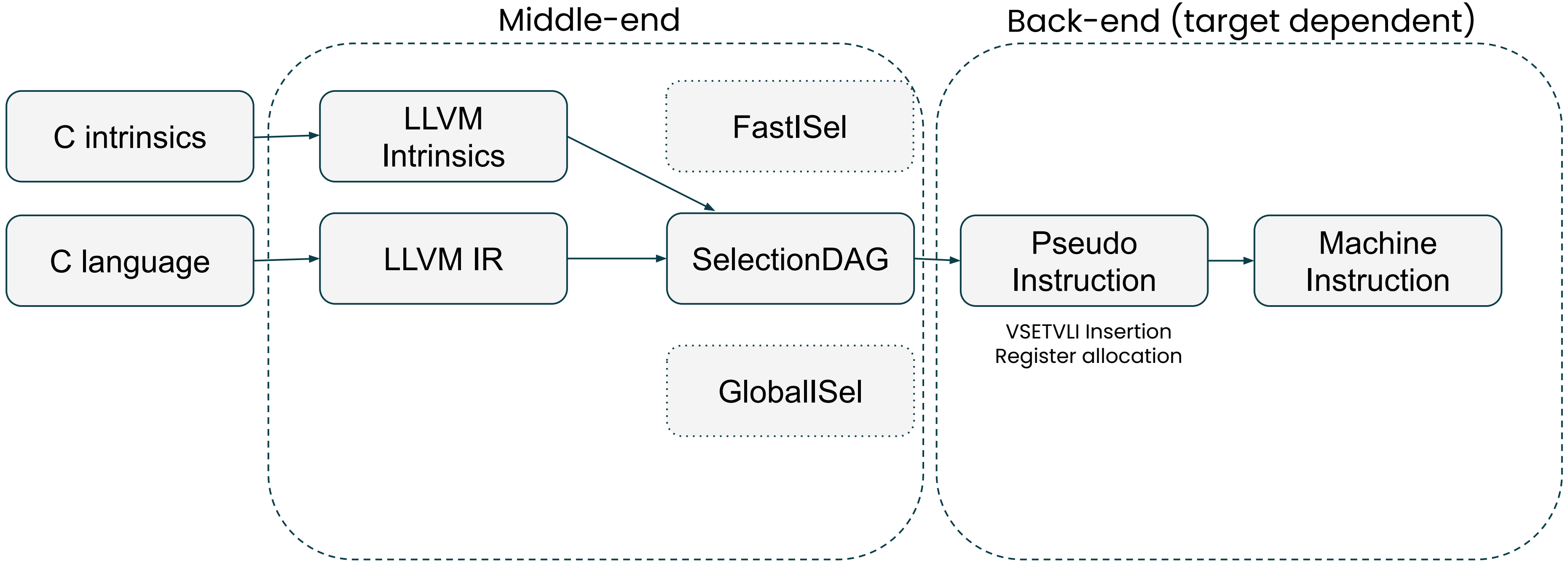
```
/// llvm/lib/Target/RISCV/RISCVInstrInfoV.td

// Vector Single-Width Integer Add and Subtract
defm VADD_V : VALU_IV_V_X_I<"vadd", 0b000000>;
defm VSUB_V : VALU_IV_V_X<"vsub", 0b000010>;
defm VRSUB_V : VALU_IV_X_I<"vrsub", 0b000011>;
```

```
/// llvm/lib/Target/RISCV/RISCVInstrInfoV.td

multiclass VALU_IV_V<string opcodestr, bits<6> funct6> {
  def V : VALUVV<funct6, OPIVV, opcodestr # ".vv">,
                SchedBinaryMC<"WriteVIALUV", "ReadVIALUV", "ReadVIALUV">;
}

multiclass VALU_IV_X<string opcodestr, bits<6> funct6> {
  def X : VALUVX<funct6, OPIVX, opcodestr # ".vx">,
                SchedBinaryMC<"WriteVIALUX", "ReadVIALUV", "ReadVIALUX">;
}

multiclass VALU_IV_I<string opcodestr, bits<6> funct6> {
  def I : VALUVI<funct6, opcodestr # ".vi", simm5>,
                SchedUnaryMC<"WriteVIALUI", "ReadVIALUV">;
}
```

```
/// llvm/lib/Target/RISCV/RISCVInstrFormatsV.td

def OPIVV : RISCVVFormat<0b000>;
def OPIVX : RISCVVFormat<0b100>;
```

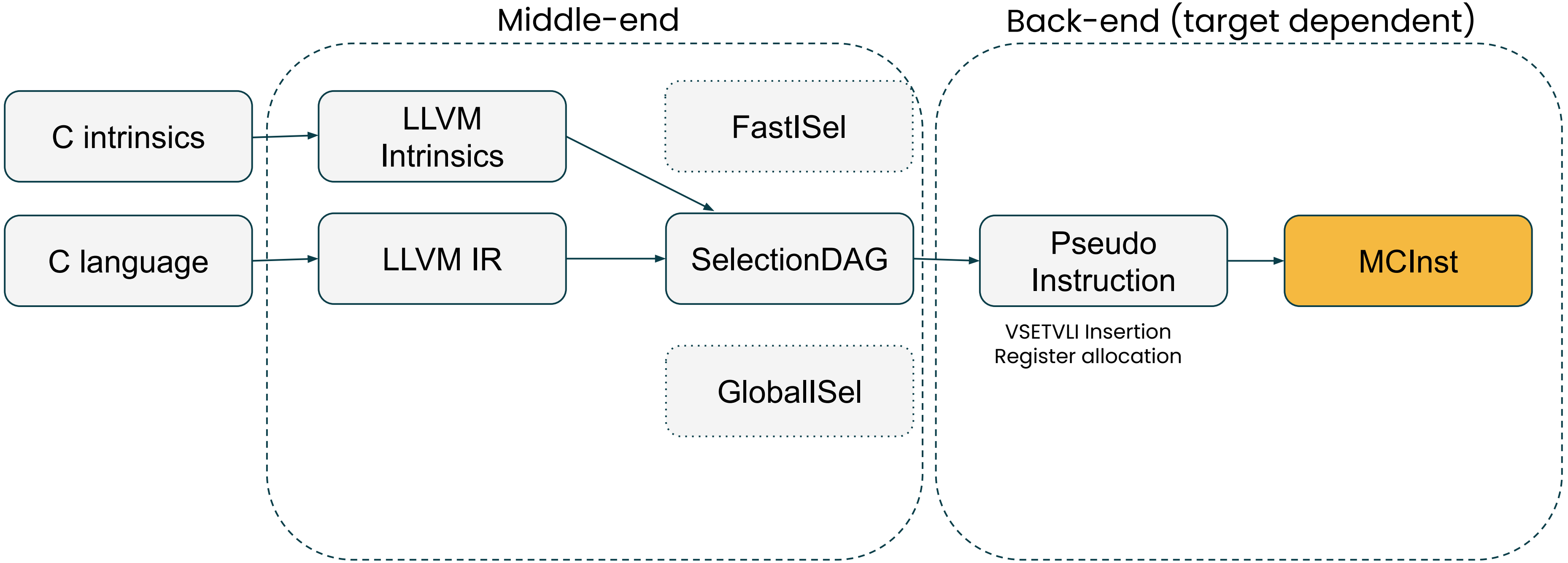
Demo: Supporting the vector bfloat16 intrinsics for RISC-V

Workflow for RISC-V vector



Case study: The RISC-V vector intrinsics

Workflow for RISC-V vector



Demo: Supporting the vector bfloat16 intrinsics for RISC-V

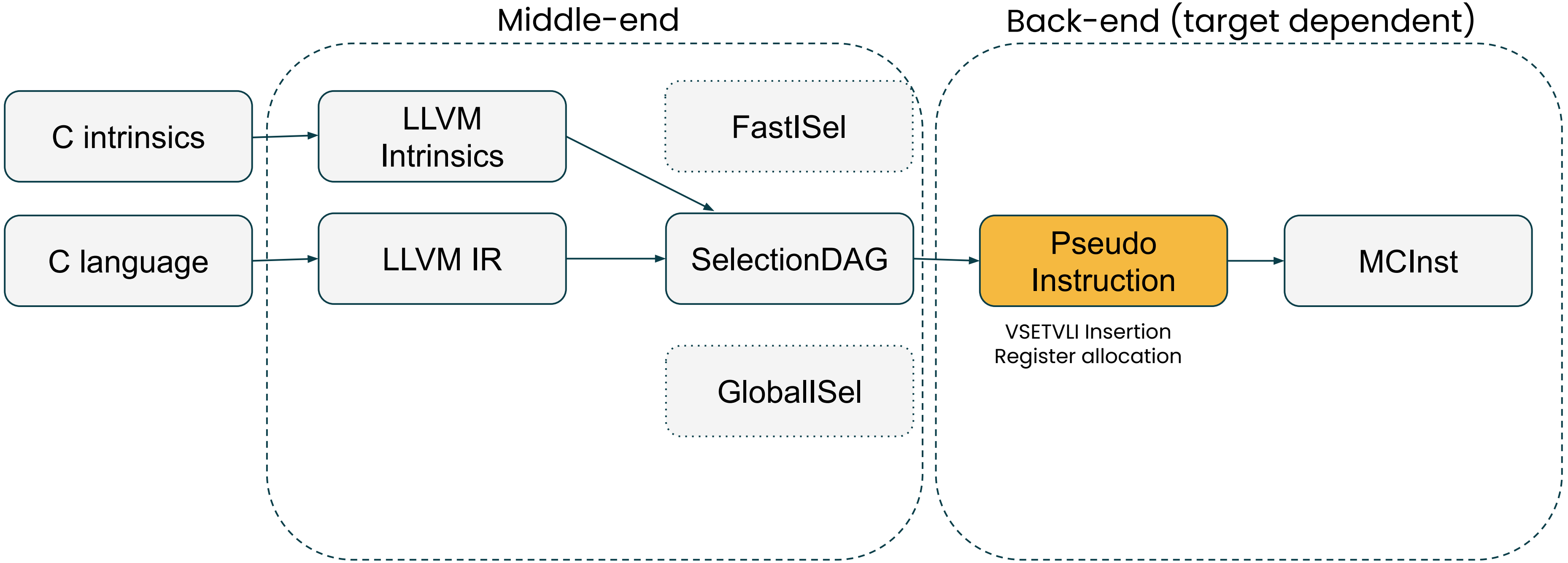
bfloat16 `vfwcvt.bf16.f.f.v` - Machine instruction

```
/// llvm/lib/Target/RISCV/RISCVInstrInfoZvfbf.td

let Predicates = [HasStdExtZvfbfmin], Constraints = "@earlyclobber $vd",
    mayRaiseFPException = true in {

let RVVConstraint = WidenCvt in
  defm VFWCVTBF16_F_F_V : VWCVTF_FV_VS2<"vfwcvtbf16.f.f.v", 0b010010, 0b01101>;

}
```

Patch in D147611 [RISCV][MC] Add support for experimental Zvfbfmin extension

Case study: The RISC-V vector intrinsics

Workflow for RISC-V vector



Demo: Supporting the vector bfloat16 intrinsics for RISC-V

bfloat16 `vfwcvt.bf16.f.f.v` - Pseudo Instruction

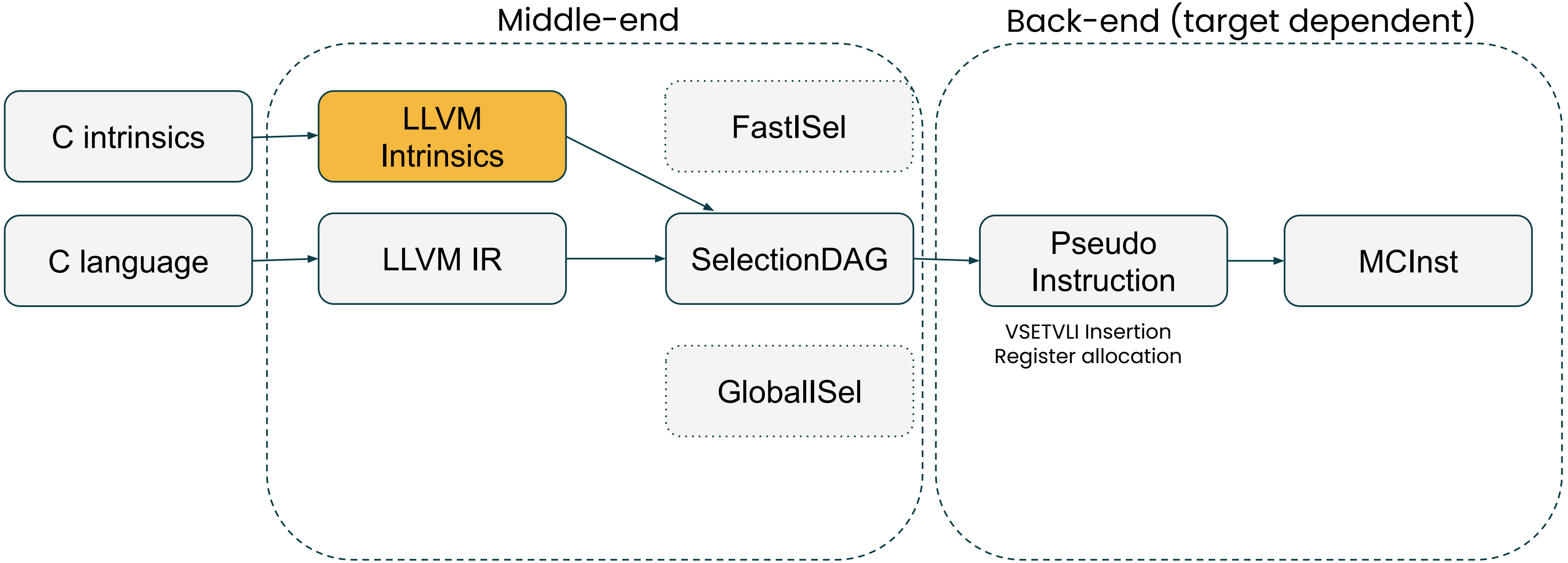
```
/// llvm/lib/Target/RISCV/RISCVInstrInfoVPseudo.td

let mayRaiseFPException = true in {
let hasSideEffects = 0, hasPostISelHook = 1 in {

defm PseudoVFWCVTBF16_F_F : VPseudoVWCVTD_V;

} // mayRaiseFPException = true
```

```
/// llvm/lib/Target/RISCV/RISCVInstrInfoVPseudo.td

defm : VPatConversionWF_VF_BF<"int_riscv_vfwcvtbf16_f_f_v",
                              "PseudoVFWCVTBF16_F_F">;
```

Patch in D156287 [RISCV] Add codegen support for bf16 vector

Case study: The RISC-V vector intrinsics

Workflow for RISC-V vector



Demo: Supporting the vector bfloat16 intrinsics for RISC-V

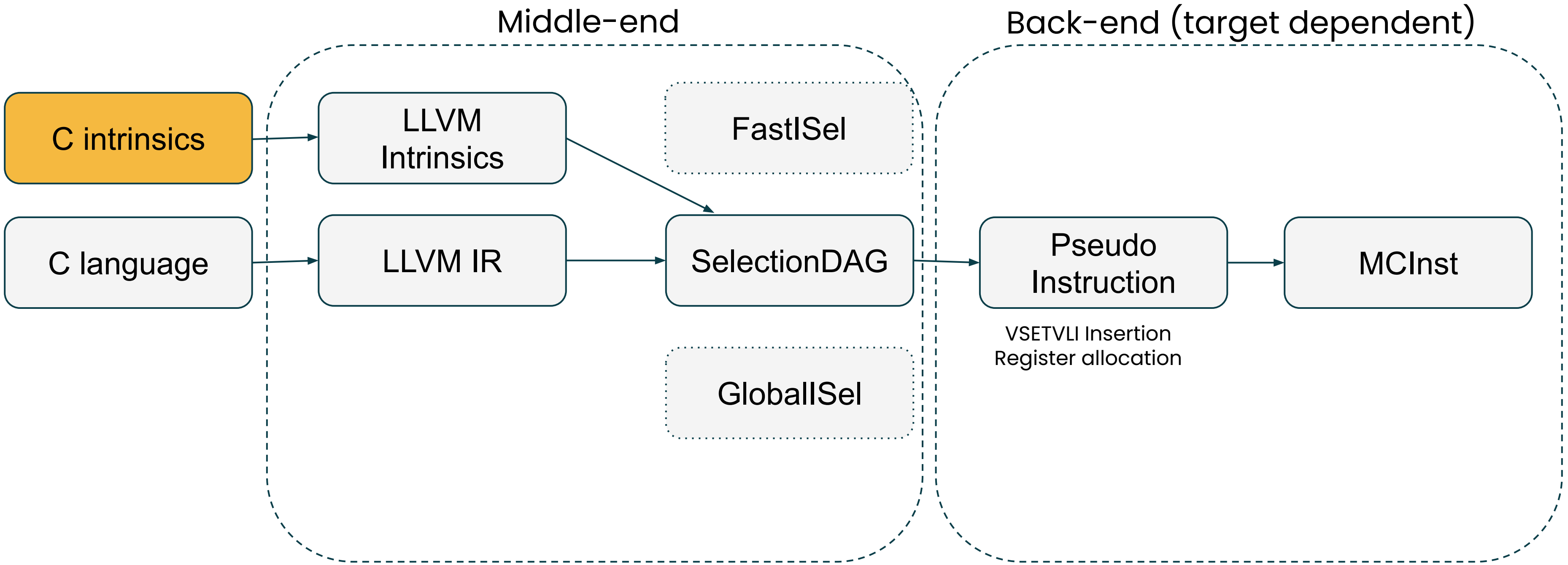
bfloat16 `vfwcvt.bf16.f.f.v` - LLVM IR

```
/// llvm/include/llvm/IR/IntrinsicsRISCV.td

defm vfwcvtbf16_f_f_v : RISCVConversion;
```

Patch in D156287 [RISCV] Add codegen support for bf16 vector

Case study: The RISC-V vector intrinsics

Workflow for RISC-V vector



Demo: Supporting the vector bfloat16 intrinsics for RISC-V

bfloat16 `vfwcvt.bf16.f.f.v` - C Intrinsics types

```
/// clang/include/clang/Basic/RISCVVTypes.def


RVV_VECTOR_TYPE_BFLOAT("__rvv_bfloat16mf4_t",RvvBFloat16mf4,RvvBFloat16mf4Ty,1, 16, 1)
RVV_VECTOR_TYPE_BFLOAT("__rvv_bfloat16mf2_t",RvvBFloat16mf2,RvvBFloat16mf2Ty,2, 16, 1)
RVV_VECTOR_TYPE_BFLOAT("__rvv_bfloat16m1_t", RvvBFloat16m1, RvvBFloat16m1Ty, 4, 16, 1)
RVV_VECTOR_TYPE_BFLOAT("__rvv_bfloat16m2_t", RvvBFloat16m2, RvvBFloat16m2Ty, 8, 16, 1)
RVV_VECTOR_TYPE_BFLOAT("__rvv_bfloat16m4_t", RvvBFloat16m4, RvvBFloat16m4Ty, 16, 16, 1)
RVV_VECTOR_TYPE_BFLOAT("__rvv_bfloat16m8_t", RvvBFloat16m8, RvvBFloat16m8Ty, 32, 16, 1)
```

Patch in D152498 [RISCV][Clang] Add bf16-type vector support for RVV

Demo: Supporting the vector bfloat16 intrinsics for RISC-V

bfloat16 `vfwcvt.bf16.f.f.v` - C Intrinsics

```
/// clang/include/clang/Basic/riscv_vector.td

let RequiredFeatures = ["Zvfbf"] in
  def vfwcvtbf16_f_f_v : RVVConvBuiltin<"w", "wv", "y", "vfwcvtbf16">;
```

Patch in D152498 [RISCV][Clang] Add bf16-type vector support for RVV

Learning resources

- Writing an LLVM Backend - LLVM documentation
- 2018 LLVM Developers' Meeting: A. Bradbury "LLVM backend development by example (RISC-V)"

SiFive

Thank you for your attention