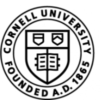
Global Instruction Selection for Scalable Vectors in LLVM

Jiahan Xie



Cornell Bowers C·IS
Computer Science

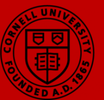
Why are we here?

- The purpose of the talk is to demonstrate the support for scalable vectors under the global instruction framework
- We would also like to give some tips and caveats for other targets if they also have similar plans

Agenda

1	Primers
1.1	Global Instruction Selection in LLVM
1.2	Scalable Vectors
2	Our Work
3	Demo

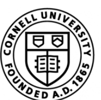
Primers on Global Instruction Selection and Scalable Vectors



Global Instruction Selection

Global Instruction Selection

- Global Instruction Selection (GISEL) is a framework that provides a set of reusable passes and utilities for instruction selection
- It is “global” because it operates on the whole function rather than a single basic block
- GISEL is intended to be a replacement for SelectionDAG
  - and we will give a demo to compare the results



GISEL has four passes

1. IRTranslator
2. Legalizer
3. Register Bank Selector
4. Instruction Select

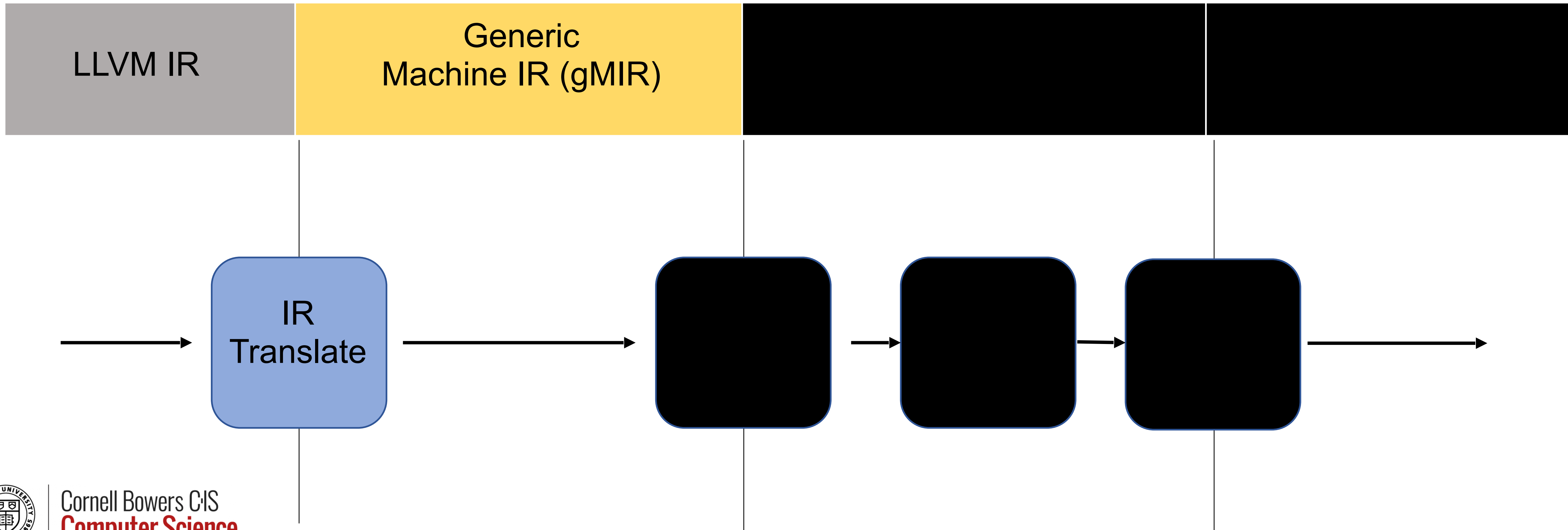
GISEL has four passes with some optional passes

- IRTranslator
- Combiner
- Legalizer
- Combiner
- Register Bank Selector
- Instruction Select

We will focus on the four main passes

GISEL's first pass is IRTranslator

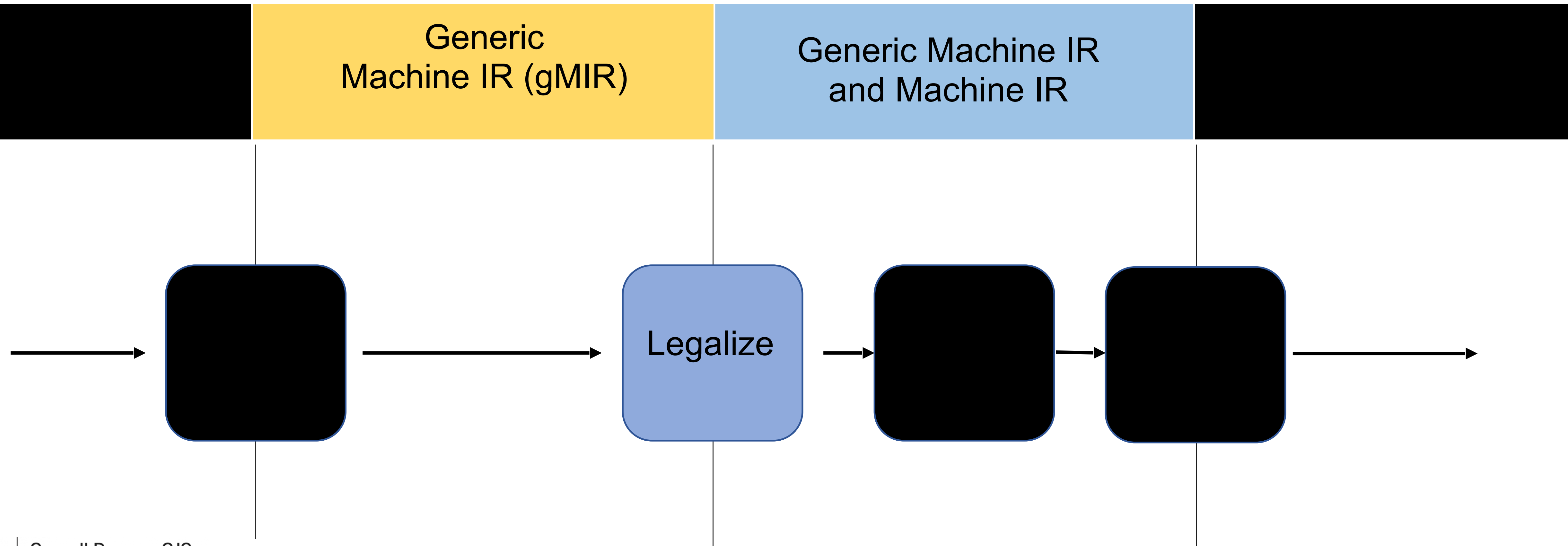
IRTranslator converts LLVM IR into Generic MachineIR (gMIR)



Cornell Bowers CIS
Computer Science

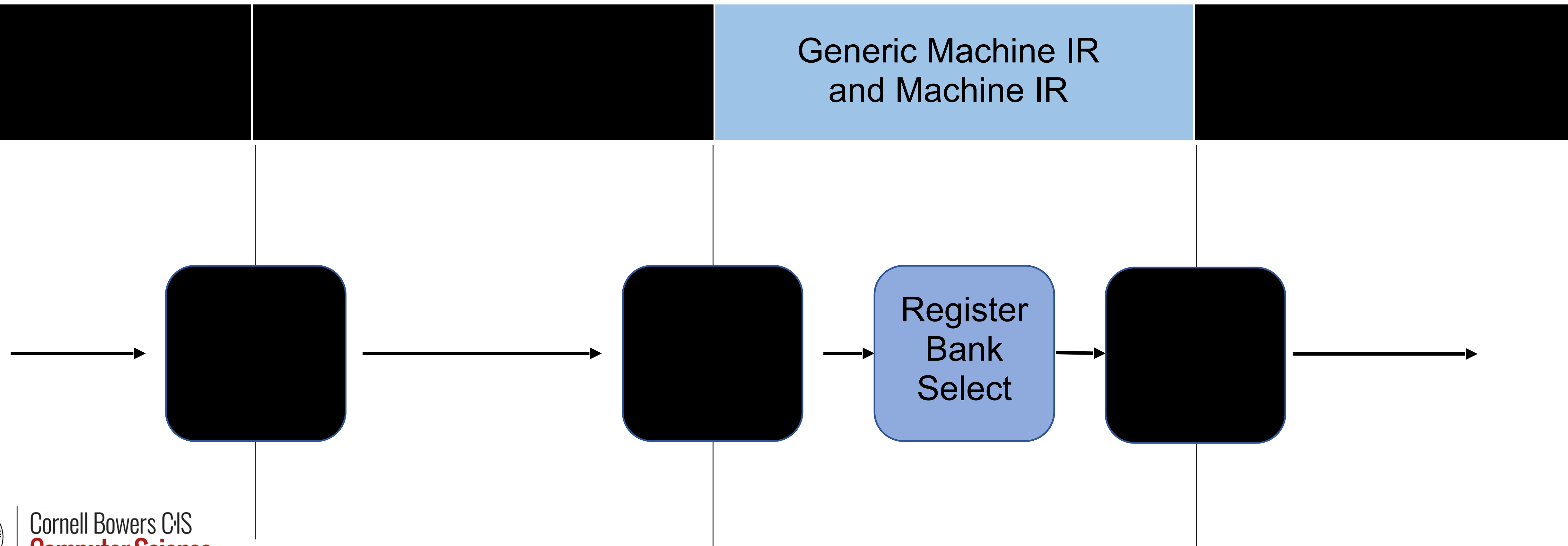
GISEL’s second pass is Legalizer

Legalizer replaces unsupported gMIRs with supported ones



GISEL's third pass is Register Bank Selector

Register Bank Selector binds virtual registers to register banks



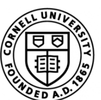
GISEL's fourth pass is Instruction Selector

Instruction Selector transforms gMIRs into equivalent target-specific instructions

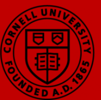
Generic Machine IR and Machine IR

Machine IR (MIR)

Instruction Select



Scalable Vectors

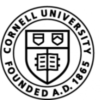


Scalable Vectors can scale by `vscale`

The total number of elements of a *scalable vector* is a constant multiple (`vscale`) of the specified # elements

- `vscale` is *unknown* at compile time;
- `vscale` is hardware-dependent and is fixed for all vectors across the program

`< vscale x N x elmty>`



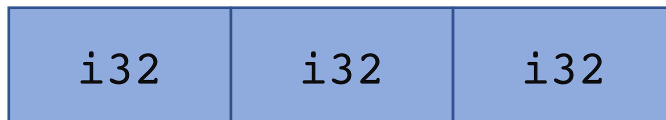
Scalable Vectors can scale by `vscale`

The total number of elements of a *scalable vector* is a constant multiple (`vscale`) of the specified # elements

`< vscale x N x elmty>`

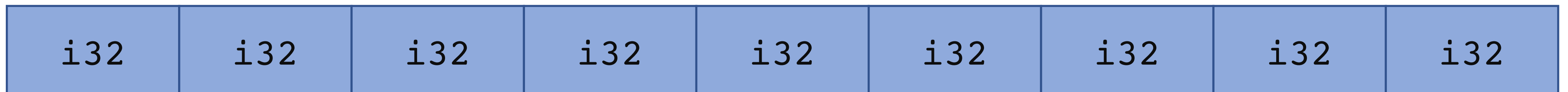
N = 3
elmty = i32

vscale=1

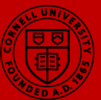


vscale=2

vscale=3

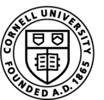


Our Work



Our Work

1	Pre-work	
	1.1	MIR parser and MachineVerifier enhancement
	1.2	Call lowering
2	Four passes	
	2.1	ALU operations
	2.2	Load/Store



Pre-work – *MIR Parser* and *Verifier* enhancements

- MIR parser is enhanced so that scalable vectors in MIR is now supported globally in LLVM GISEL

Pre-work – *MIR Parser* and *Verifier* enhancements

- The `MachineVerifier` is updated so that we can `COPY` from fixed-sized vectors (`Src`) to scalable types (`Dst`), as long as the minimum size of the destination register can hold the fixed size (`DstSize.getKnownMinValue() >= SrcSize.getFixedValue()`)
  - MIR files are still completely generic; and we can copy between physical registers to virtual registers, and vice versa

```
body:
  bb.0:
    liveins: $v8
    %0:_(<vscale x 1 x s8>) = COPY $v8
```

physical register

Pre-work – *Call lowering* needs to be supported for each target

Before working on the canonical four passes of GISEL, it's a requirement to support call lowering first:

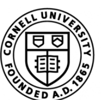
- Lower arguments: enables passing scalable vectors as function arguments;
- Lower return: enables returning scalable vectors from function calls;
- Lower call: enables lowering the call instruction according to target's ABI

IRTranslator

IRTranslation for ALU operations and Load/Stores are straightforward.

A sample result is shown below:

```
define <vscale x 2 x i8>  @vload_nx2i8(ptr %pa) {
  ; LABEL: name: vload_nx2i8
  ...
  ; NEXT:   [[LOAD:%[0-9]+]]:_(<vscale x 2 x s8>) = G_LOAD [[COPY]](p0) :: (load
(<vscale x 2 x s8>) from %ir.pa)
  ...
  %va = load <vscale x 2 x i8>, ptr %pa
  ret <vscale x 2 x i8> %va
}
```



IRTranslator

IRTranslation is usually applicable to all targets.

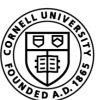
If you want to support more complicated `OpCodes`, `llvm/lib/CodeGen/GlobalISel/IRTranslator.cpp` has the corresponding `translateXX` to be changed

IRTranslator

If you want to support more complicated `OpCodes`, `llvm/lib/CodeGen/GlobalISel/IRTranslator.cpp` has the corresponding `translateXX` to be changed

- Most of the time, the things needed to be changed are related to types
    - migrate to use `TypeSize` explicitly, for example:

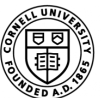
```
 bool IRTranslator::translateLoad(const User &U, MachineIRBuilder &MIRBuilder)
{
-
-   unsigned StoreSize = DL->getTypeStoreSize(LI.getType());
-   if (StoreSize == 0)
+   TypeSize StoreSize = DL->getTypeStoreSize(LI.getType());
+   if (StoreSize.isZero())
      return true;
```



Legalization for scalable vectors can be challenging

Legalization poses unique challenges due to *type* information:

- SelectionDAG uses `MVT`/`EVT`;
- GISEL uses `LowLevelTypes` (`LLTs`)



Legalization for scalable vectors can be challenging

Need to map `LowLevelTypes` to target-specific vector types

- Can use scalar types as the building blocks

scalar building blocks

- a scalable vector of scalars

```
const LLT nxv1s8 = LLT::scalable_vector(1, s8);
...
const LLT nxv8s64 = LLT::scalable_vector(8, s64);
auto AllVecTys = {nxv1s8, ..., nxv8s64};
```

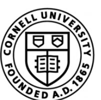
- a scalable vector of pointers

```
const LLT nxv1p0 = LLT::scalable_vector(1, p0);
...
auto PtrVecTys = {nxv1p0, ...};
```

Legalization for scalable vectors can be challenging

Implement the legalization logics using `LegalizerInfo` with `LegalityPredicates` and `LegalityQuerys`

- some simple ALU operations shares the same logic:
  - it’s legal only if the target supports vector instructions, followed by more custom predicates



Legalization for scalable vectors can be challenging

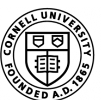
Implement the legalization logics using `LegalizerInfo` with

`LegalityPredicates` and `LegalityQuerys`

- need to implement custom logics using `customIf` for special cases:
  - for example, load/stores with non-standard alignments

```
LoadStoreActions.
    .lowerIf...()
    .customIf(LegalityPredicate(/* non-aligned memory load/store */ ))

bool RISCVLegalizerInfo::legalizeLoadStore(MachineInstr &MI,
                                           LegalizerHelper &Helper,
                                           MachineIRBuilder &MIB) const {
     some custom logic
}
```



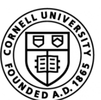
Our work chooses *legal* instructions/types that the RISC-V Vector Extension can support

RISC-V Vector Extension (RVV) has several important parameters:

- `VLEN`: vector register length, length of a single vector register in bits
- `LMUL`: vector register group multiplier (1/8 - 8)
- `SEW`: set element width, the bit width of each element in the vector (8 - 64)

Everything is calculated and legalized based on the following mapping:

LMUL / SEW	1/8	1/4	1/2	1	2	4	8
i64	N/A	N/A	N/A	nxv1i64	nxv2i64	nxv4i64	nxv8i64
i32	N/A	N/A	nxv1i32	nxv2i32	nxv4i32	nxv8i32	nxv16i32
i16	N/A	nxv1i16	nxv2i16	nxv4i16	nxv8i16	nxv16i16	nxv32i16
i8	nxv1i8	nxv2i8	nxv4i8	nxv8i8	nxv16i8	nxv32i8	nxv64i8



Cornell Bowers CIS
Computer Science

Register Bank Selection phase

- Each target needs to create its vector register banks:
    - `VRBRegBank` in RISC-V
- Need target-specific function to map operands values to register banks using the `ValueMapping` class based on the size of the operands

An example output from the Register Bank Selection phase

```
name:            vadd_vv_nxv2i8
    ; RV64I-LABEL: name: vadd_vv_nxv2i8
    ; RV64I: liveins: $v8, $v9
    ; RV64I-NEXT: {{  $}}
    ; RV64I-NEXT: [[COPY:%[0-9]+]]:vrb(<vscale x 2 x s8>) = COPY $v8
    ; RV64I-NEXT: [[COPY1:%[0-9]+]]:vrb(<vscale x 2 x s8>) = COPY $v9
    ; RV64I-NEXT: [[ADD:%[0-9]+]]:vrb(<vscale x 2 x s8>) = G_ADD [[COPY]], [[COPY1]]
    ; RV64I-NEXT: $v8 = COPY [[ADD]](<vscale x 2 x s8>)
    ; RV64I-NEXT: PseudoRET implicit $v8
    %0:_(<vscale x 2 x s8>) = COPY $v8
    %1:_(<vscale x 2 x s8>) = COPY $v9
    %2:_(<vscale x 2 x s8>) = G_ADD %0, %1
    $v8 = COPY %2(<vscale x 2 x s8>)
    PseudoRET implicit $v8
```

Instruction Selection phase

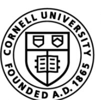
- `TableGen` patterns work out-of-the-box for the ALU instructions
- When `load`/`store` a vector of pointers, `TableGen` patterns might not be sufficient
  - pointer types are typically converted to integer `MVT`/`EVT`s before legalization in `SelectionDAG`
  - we defined custom logic to cast pointers to `sXLen`.

Demo – GISEL the SAXPY example to RVV

Demo – GISEL the SAXPY example to RVV

SAXPY in C:

```
void saxpy (size_t n, const int a, const int *restrict x, int *restrict y)
{
  for (size_t i = 0; i < n; ++i) {
    y[i] = a * x[i] + y[i];
  }
}
```



Demo – SAXPY in LLVM

```
define void @saxpy(i64 noundef %n, i32 noundef signext %a, ptr noalias nocapture noundef readonly %x,
ptr noalias nocapture noundef %y) {
entry:
  %cmp8.not = icmp eq i64 %n, 0
  br i1 %cmp8.not, label %for.cond.cleanup, label %for.body.preheader

for.body.preheader:                                  ; preds = %entry
  %0 = call i64 @llvm.vscale.i64()
  %1 = mul i64 %0, 4
  %min.iters.check = icmp ult i64 %n, %1
  br i1 %min.iters.check, label %scalar.ph, label %vector.ph

vector.ph:                                           ; preds = %for.body.preheader
  %2 = call i64 @llvm.vscale.i64()
  %3 = mul i64 %2, 4                                 ; sets the scalable vector stride length
  %n.mod.vf = urem i64 %n, %3
  %n.vec = sub i64 %n, %n.mod.vf                    ; computes the number of remaining scalar iterations
  %4 = call i64 @llvm.vscale.i64()
  %5 = mul i64 %4, 4                                 ; scalable stride = vscale x 4
  %broadcast.splatinsert = insertelement <vscale x 4 x i32> poison, i32 %a, i64 0
  %broadcast.splat = shufflevector <vscale x 4 x i32> %broadcast.splatinsert, <vscale x 4 x i32>
poison, <vscale x 4 x i32> zeroinitializer          ; splat scalar a to a vector
  br label %vector.body
```

Demo – SAXPY in LLVM

```
vector.body:                                                    ; preds = %vector.body, %vector.ph
  %index phi i64 [ 0, %vector.ph ], [ %index.next, %vector.body ]
  %6 = add i64 = %index, 0
  %7 = getelementptr inbounds i32, ptr %x, i64 %6
  %8 = getelementptr inbounds i32, ptr %7, i32 0
  %wide.load = load <vscale x 4 x i32>, ptr %8, align 4         ; load x vector
  %9 = mul nsw <vscale x 4 x i32> %wide.load, %broadcast.splat  ; x * a scalable vector multiplication
  %10 = getelementptr inbounds i32, ptr %y, i64 %6
  %11 = getelementptr inbounds i32, ptr %10, i32 0
  %wide.load10 = load <vscale x 4 x i32>, ptr %11, align 4
  %12 = add nsw <vscale x 4 x i32> %9, %wide.load10             ; x*a + y scalable vector add
  store <vscale x 4 x i32> %12, ptr %11, align 4
  %index.next = add nuw i64 %index, %5
  %13 = icmp eq i64 %index.next, %n.vec
  br i1 %13, label %middle.block, label %vector.body

middle.block:                                                   ; preds = %vector.body
  %cmp.n = icmp eq i64 %n, %n.vec
  br i1 %cmp.n, label %for.cond.cleanup.loopexit, label %scalar.ph
```

Demo – SAXPY in LLVM

```
scalar.ph:                                        ; preds = %middle.block, %for.body.preheader
  %bc.resume.val = phi i64 [ %n.vec, %middle.block ], [ 0, %for.body.preheader ]
  br label %for.body

for.cond.cleanup.loopexit:                        ; preds = %middle.block, %for.body
  br label %for.cond.cleanup

for.cond.cleanup:                                 ; preds = %for.cond.cleanup.loopexit, %entry
  ret void

for.body:                                         ; preds = %scalar.ph, %for.body
  %i.09 = phi i64 [ %inc, %for.body ], [ %bc.resume.val, %scalar.ph ]
  %arrayidx = getelementptr inbounds nuw i32, ptr %x, i64 %i.09
  %14 = load i32, ptr %arrayidx, align 4
  %mul = mul nsw i32 %14, %a
  %arrayidx1 = getelementptr inbounds nuw i32, ptr %y, i64 %i.09
  %15 = load i32, ptr %arrayidx1, align 4
  %add = add nsw i32 %mul, %15
  store i32 %add, ptr %arrayidx1, align 4
  %inc = add nuw i64 %i.09, 1
  %exitcond.not = icmp eq i64 %inc, %n
  br i1 %exitcond.not, label %for.cond.cleanup.loopexit, label %for.body
}
```

Demo – lower SAXPY to RISC-V using GISEL

```
.text
.globl saxpy                          # -- Begin function
saxpy
.p2align 2
.type saxpy,@function
saxpy:                                 # @saxpy
.cfi_startproc
# %bb.0:                               # %entry
beqz a0, .LBB0_8
# %bb.1:                               # %for.body.preheader
csrr a4, vlenb
srli a7, a4, 3
slli a6, a7, 2
bgeu a0, a6, .LBB0_3
# %bb.2:
li a4, 0
j .LBB0_6
.LBB0_3:                               # %vector.ph
remu a5, a0, a6
sub a4, a0, a5
slli a7, a7, 4
vsetvli t0, zero, e32, m2, ta, ma
vmv.v.x v8, a1
mv t0, a2
mv t1, a3
mv t2, a4
.LBB0_4:                               # %vector.body
                                       # =>This Inner Loop
Header: Depth=1
vl2re32.v v10, (t0)
vl2re32.v v12, (t1)
vmul.vv v10, v10, v8
vadd.vv v10, v10, v12
vs2r.v v10, (t1)
sub t2, t2, a6
add t1, t1, a7
add t0, t0, a7
bnez t2, .LBB0_4
# %bb.5:                               # %middle.block
beqz a5, .LBB0_8
.LBB0_6:                               # %scalar.ph
slli a5, a4, 2
add a4, a3, a5
add a2, a2, a5
slli a0, a0, 2
add a3, a3, a0
.LBB0_7:                               # %for.body
                                       # =>This Inner Loop
Header: Depth=1
lw a0, 0(a2)
lw a5, 0(a4)
mul a0, a0, a1
add a0, a0, a5
sw a0, 0(a4)
addi a4, a4, 4
addi a2, a2, 4
bne a4, a3, .LBB0_7
.LBB0_8:                               # %for.cond.cleanup
ret
.Lfunc_end0:
.size saxpy, .Lfunc_end0-saxpy
.cfi_endproc                           # -- End function
```

Demo – lower SAXPY to RISC-V using SelectionDAG

```
.text
.globl saxpy                          # -- Begin function saxpy
.p2align 2
.type saxpy,@function
saxpy:                                 # @saxpy
.cfi_startproc
# %bb.0:                               # %entry
beqz a0, .LBB0_8
# %bb.1:                               # %for.body.preheader
csrr a6, vlenb
srli a5, a6, 1
bgeu a0, a5, .LBB0_3
# %bb.2:
li a4, 0
j .LBB0_6
.LBB0_3:                               # %vector.ph
neg a4, a5
and a4, a0, a4
slli a6, a6, 1
mv a7, a2
mv t0, a3
mv t1, a4
vsetvli t2, zero, e32, m2, ta, ma
```

```
.LBB0_4:                               # %vector.body
                                       # =>This Inner Loop
Header: Depth=1
vl2re32.v v8, (a7)
vl2re32.v v10, (t0)
vmacc.vx v10, a1, v8
vs2r.v v10, (t0)
sub t1, t1, a5
add t0, t0, a6
add a7, a7, a6
bnez t1, .LBB0_4
# %bb.5:                               # %middle.block
beq a0, a4, .LBB0_8
.LBB0_6:                               # %scalar.ph
slli a5, a4, 2
add a4, a3, a5
add a2, a2, a5
slli a0, a0, 2
add a3, a3, a0
.LBB0_7:                               # %for.body
                                       # =>This Inner Loop
Header: Depth=1
lw a0, 0(a2)
lw a5, 0(a4)
mul a0, a0, a1
add a0, a0, a5
sw a0, 0(a4)
addi a4, a4, 4
addi a2, a2, 4
bne a4, a3, .LBB0_7
.LBB0_8:                               # %for.cond.cleanup
ret
.Lfunc_end0:
.size saxpy, .Lfunc_end0-saxpy
.cfi_endproc                           # -- End function
```

Cornell Bowers C·IS
Computer Science

Demo – Comparing GISEL and SelectionDAG

```
vector.ph
  %4 = call i64 @llvm.vscale.i64()
  ; scalable stride
  %5 = mul i64 %4, 4
  %splatinsert = insertelement <vscale x 4 x i32> poison, i32 %a, i64 0
  ; splat `a` to a vector
  %splat = shufflevector <vscale x 4 x i32> %splatinsert, <vscale x 4 x i32> poison, <vscale x 4 x i32> zeroinitializer
```

followed by:

```
vector.body:
  %7 = getelementptr inbounds i32, ptr %x, i64 %6
  ; x * a scalable vector multiplication
  %9 = mul nsw <vscale x 4 x i32> %wide.load, %broadcast.splat
  %10 = getelementptr inbounds i32, ptr %y, i64 %6
  ; x*a + y scalable vector add
  %12 = add nsw <vscale x 4 x i32> %9, %wide.load10
  store <vscale x 4 x i32> %12, ptr %11, align 4
```

SelectionDAG has an optimization on vector accumulation (mult and add)

```
.LBB0_3:                               # %vector.ph
# some operations, no splat
vsetvli t2, zero, e32, m2, ta, ma

.LBB0_4:                               # %vector.body
                                       # =>This Inner Loop

vl2re32.v v8, (a7)
vl2re32.v v10, (t0)
vmacc.vx v10, a1, v8
vs2r.v v10, (t0)
```

GISEL translates it literally

```
.LBB0_3:                               # %vector.ph
# some operations
vsetvli t0, zero, e32, m2, ta, ma
vmv.v.x v8, a1                         # splat `a`
# some operations

.LBB0_4:                               # %vector.body
                                       # =>This Inner Loop

vl2re32.v v10, (t0)
vl2re32.v v12, (t1)
vmul.vv v10, v10, v8
vadd.vv v10, v10, v12
vs2r.v v10, (t1)
```

Demo – Comparing GISEL and SelectionDAG

```
vector.ph:                                         ; preds = %for.body.preheader
  %2 = call i64 @llvm.vscale.i64()
  %3 = mul i64 %2, 4                               ; sets the scalable vector stride length
  %n.mod.vf = urem i64 %n, %3
  %n.vec = sub i64 %n, %n.mod.vf                   ; computes the number of remaining scalar iterations
  %4 = call i64 @llvm.vscale.i64()
  %5 = mul i64 %4, 4                               ; scalable stride = vscale x 4
  %broadcast.splatinsert = insertelement <vscale x 4 x i32> poison, i32 %a, i64 0
  %broadcast.splat = shufflevector <vscale x 4 x i32> %broadcast.splatinsert, <vscale x 4 x i32>
poison, <vscale x 4 x i32> zeroinitializer         ; splat scalar a to a vector
  br label %vector.body
```

SelectionDAG has an optimization on remainder calculation

```
.LBB0_3:                          # %vector.ph
neg a4, a5
and a4, a0, a4
slli a6, a6, 1
mv a7, a2
mv t0, a3
mv t1, a4
vsetvli t2, zero, e32, m2, ta, ma
```

GISEL translates it literally again

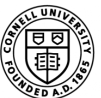
```
.LBB0_3:                          # %vector.ph
remu a5, a0, a6
sub a4, a0, a5
slli a7, a7, 4
vsetvli t0, zero, e32, m2, ta, ma
vmv.v.x v8, a1
mv t0, a2
mv t1, a3
mv t2, a4
```

Miscellaneous Tips

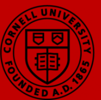
Dealing with *Type* information in GISEL

- use `LLT` wherever you can
- `unsigned => TypeSize` in `TargetRegisterInfo`
  - `== 0` ➔ `isZero; >` ➔ `TypeSize::isKnownGT`
  - `getKnownMinValue()` if want to compare with some fixed size values
  - `getNumElements()` ➔ `getElementCount()` to get *N*

< *vscale x N x elmty*>



Future Work



Future Work

- Improve the GISEL framework to exploit more optimizations;
  - especially global/inter-basic-block optimizations
- Support more `OpCodes`

Thank you!