Head First into GlobalISel

Or: How to delete SelectionDAG in 100* easy commits

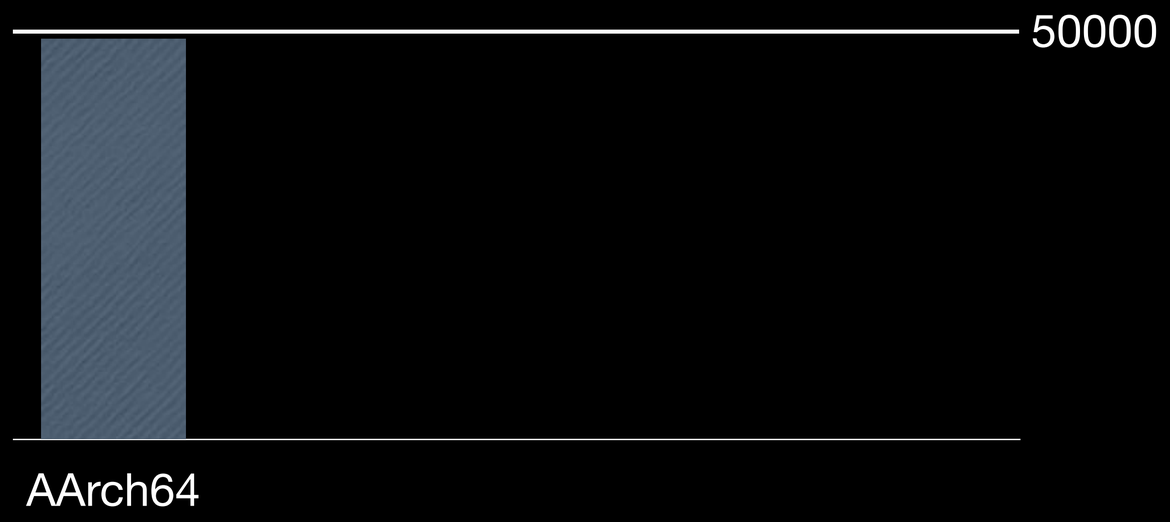
LLVM Dev Meeting 2017 • Justin Bogner, Aditya Nandakumar, Daniel Sanders • Apple

Porting to GlobalISel

- What is the structure of a GlobalISel backend?
- How can we test an implementation in progress?
- Where can we split up and parallelize work?
- What do we need to do next?

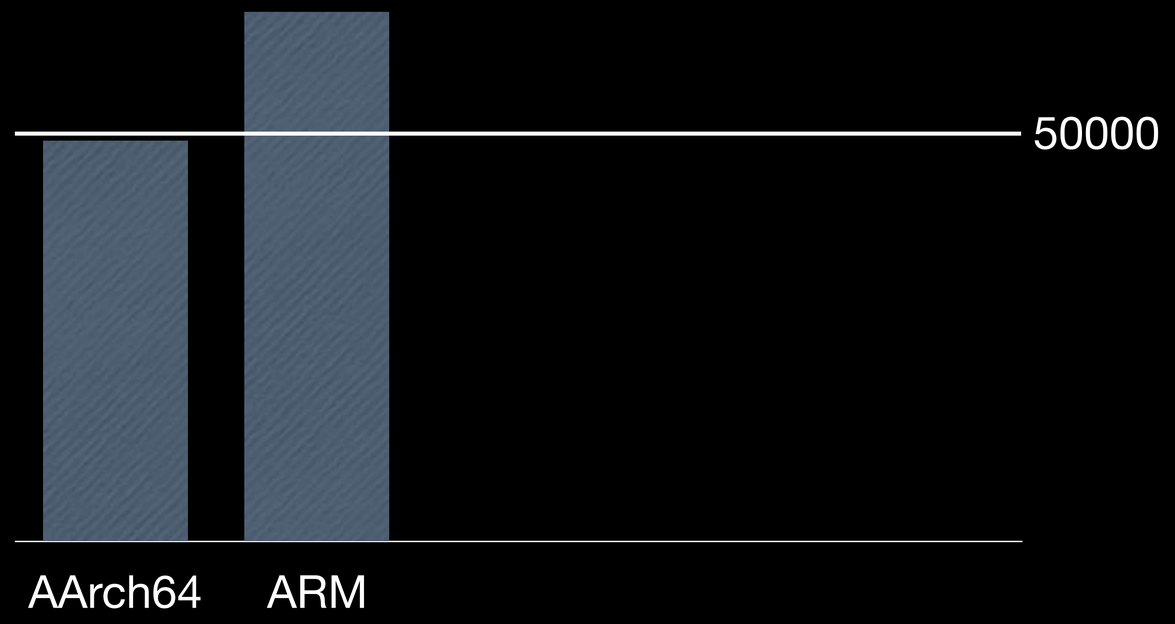
A Simple Backend

Lines of Code

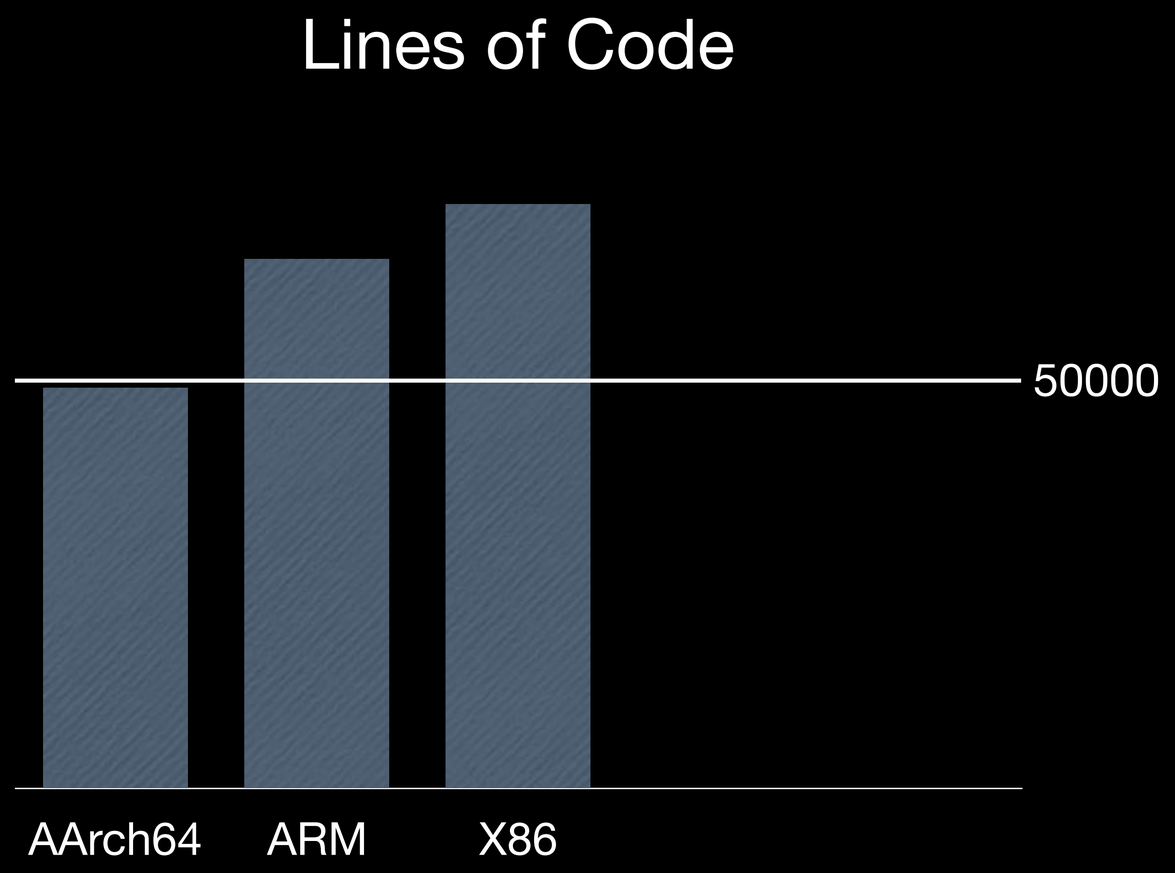


A Simple Backend

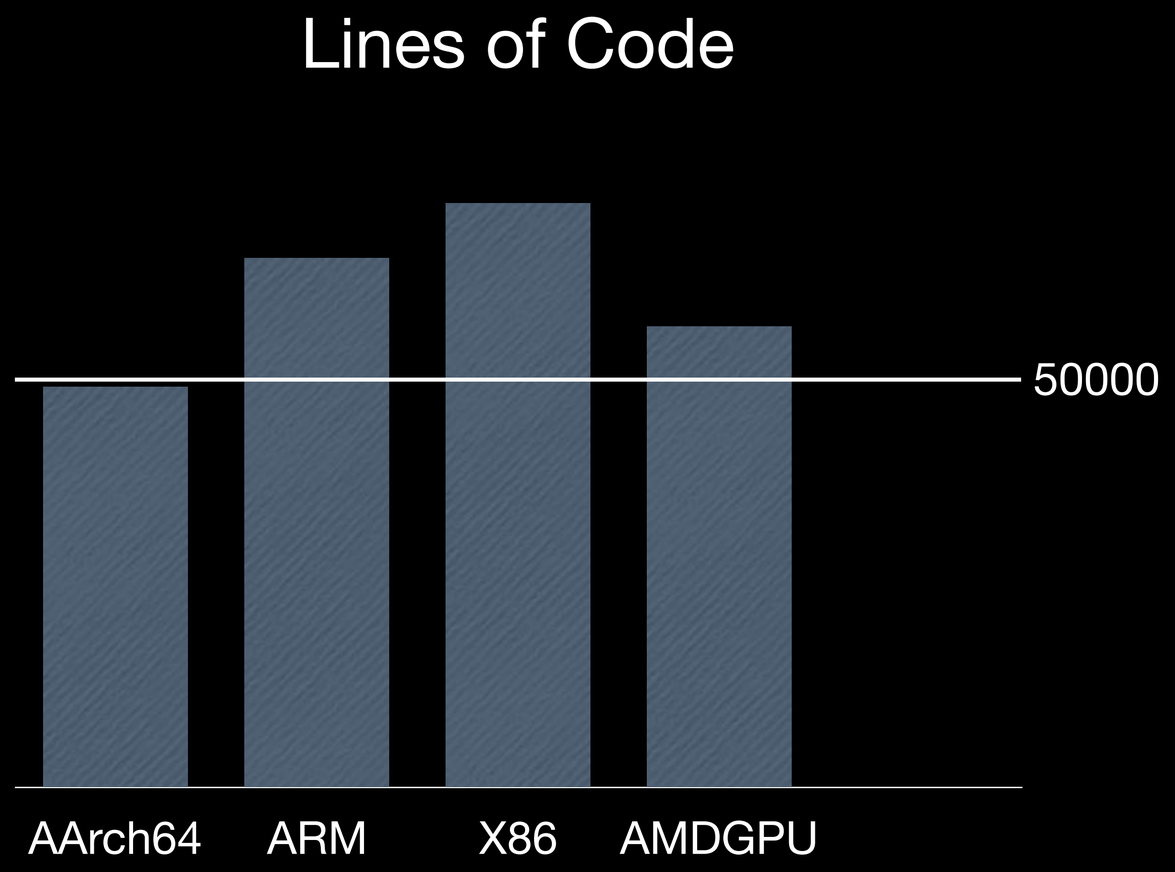
Lines of Code



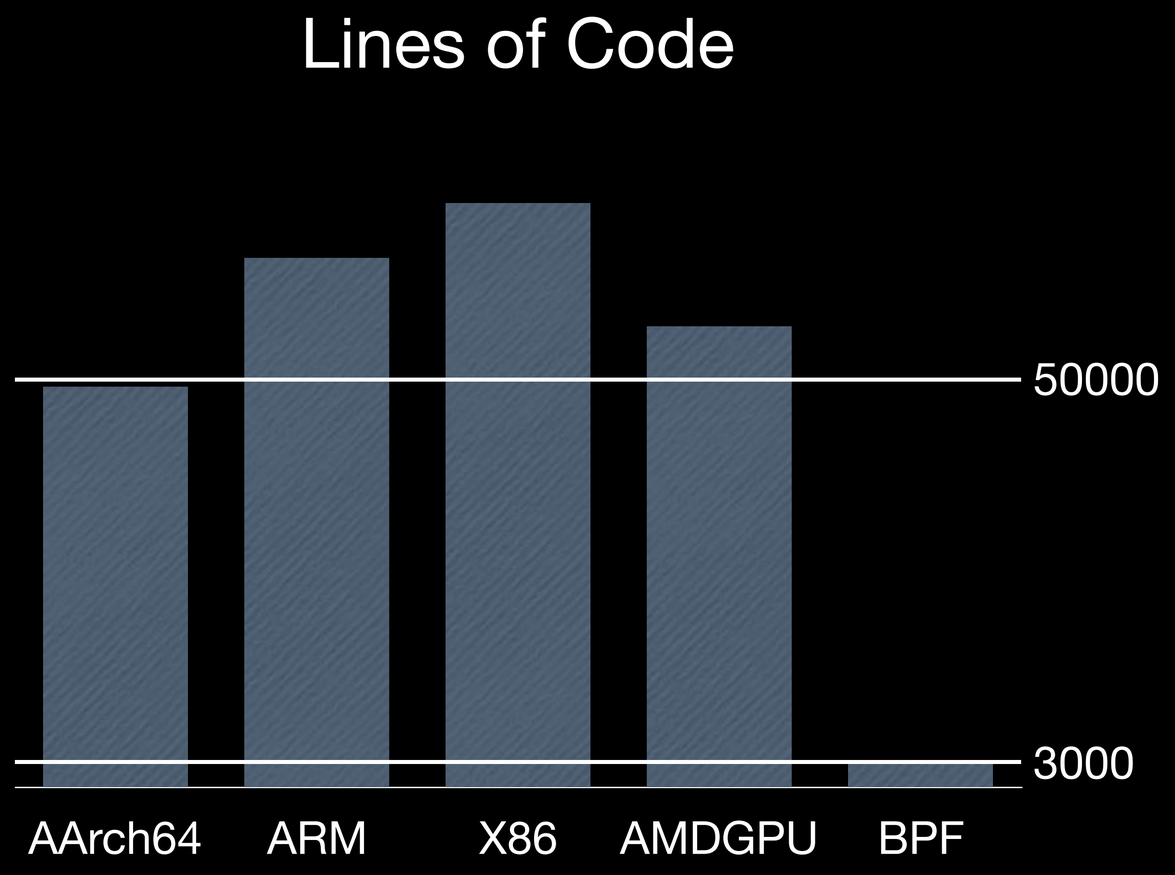
A Simple Backend



A Simple Backend

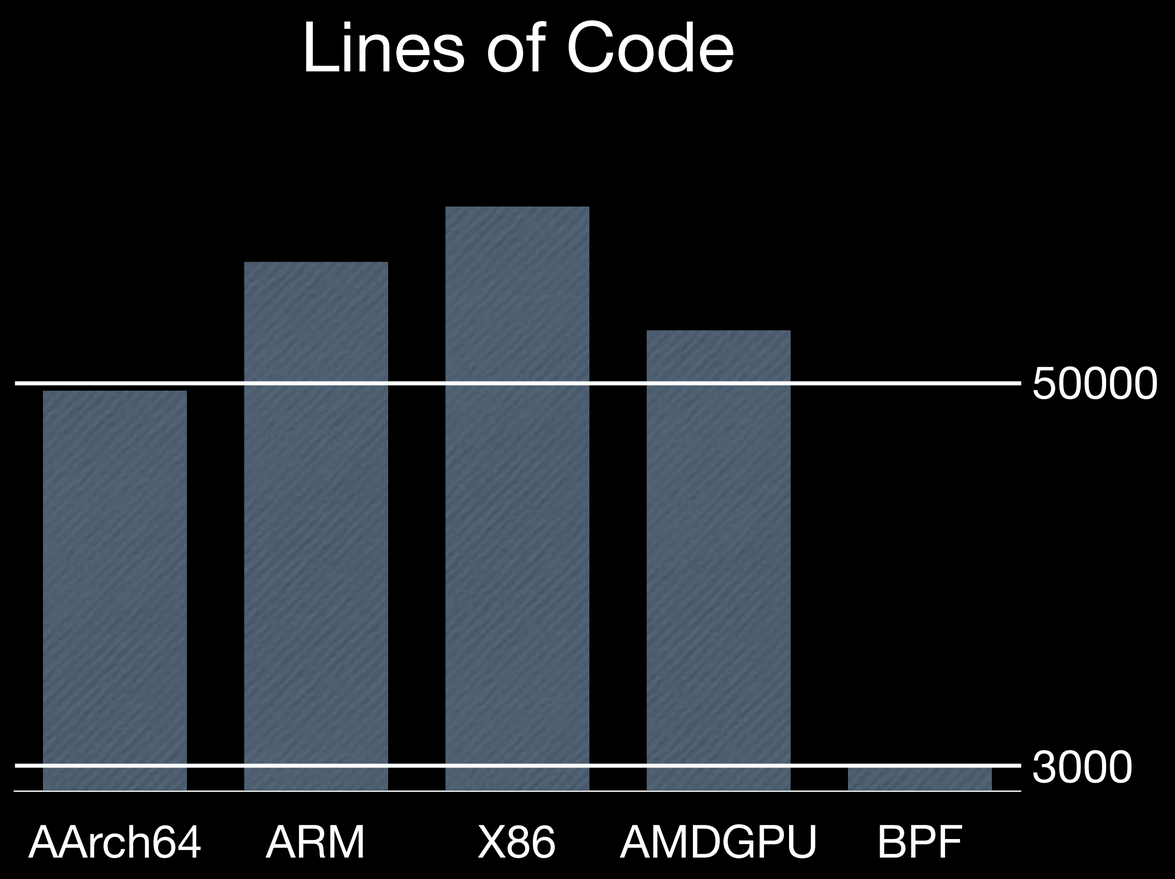


A Simple Backend

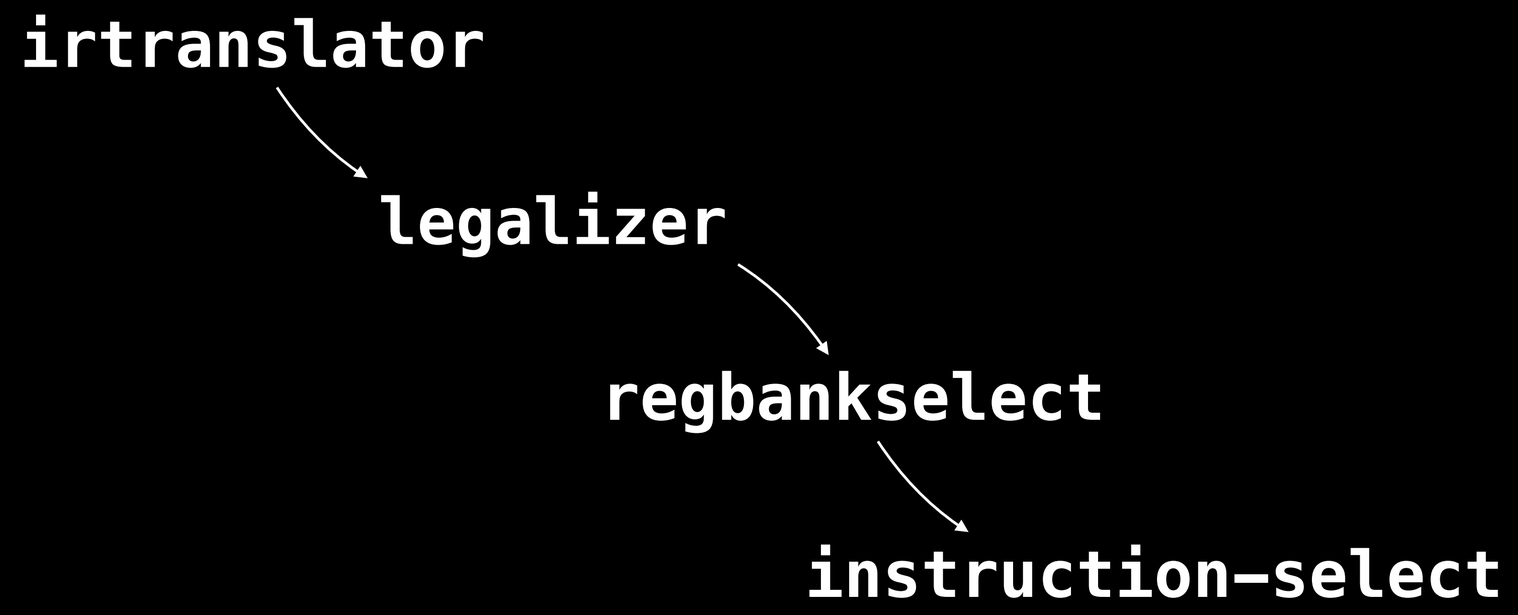


A Simple Backend

- Working through the BPF backend as a reference
- 1 Register class, 1 Legal type, 1 Calling convention
- Will refer to AArch64 as necessary for illustrating complexity



Anatomy of GlobalISel



Anatomy of GlobalISel

irtranslator	CallLowering
legalizer	LegalizerInfo
regbankselect	RegBankInfo
instruction-select	InstructionSelector

Implementing GlobalISel

Wire up GISel	`TargetMachine/SubTarget`
`irtranslator`	`CallLowering`
`regbankselect`	`RegBankInfo`
`legalizer`	`LegalizerInfo`
`instruction-select`	`InstructionSelector`

MIR Testing

- GISel tests target a single pass in the pipeline
- Test passes using `-run-pass` and `.mir` files
- `llc` emits MIR when told to `-stop-after` machine passes
- Use `-simplify-mir` to generate human-editable output

MIR Example

```
define i32 @double(i32 %x) {
  %y = add i32 %x, %x
  ret i32 %y
}
```

MIR Example

```
name: double
legalized: false
regBankSelected: false
body: |
  bb.0:
    liveins: %r1, %r2

    %1:_(s64) = COPY %r1
    %0:_(s32) = G_TRUNC %1(s64)
    %2:_(s32) = G_ADD %0, %0
    %r0 = COPY %2(s32)
    RET implicit %r0
```

```
llc -global-isel -march=bpf -stop-after=irtranslator -simplify-mir
```

Virtual Registers

```
%1:_(s64) = COPY %r1

%1:<bank>(s64) = COPY %r1

%1:<class> = COPY %r1
```

VReg constraints change throughout the pipeline

MIR Testing

```
; CHECK: [[CP:%[0-9]+]]:_(s64) = COPY %r1
; CHECK: [[TR:%[0-9]+]]:_(s32) = G_TRUNC [[CP]](s64)
; CHECK: [[ADD:%[0-9]+]]:_(s32) = G_ADD [[TR]], [[TR]]

%1:_(s64) = COPY %r1
%0:_(s32) = G_TRUNC %1(s64)
%2:_(s32) = G_ADD %0, %0
```

Don't match register numbers directly

Wire up GlobalISel

Wire up GISel	**`TargetMachine/SubTarget`**
`irtranslator`	`CallLowering`
`regbankselect`	`RegBankInfo`
`legalizer`	`LegalizerInfo`
`instruction-select`	`InstructionSelector`

Subtarget Setup

```
BPFSubtarget : BPFGenSubtargetInfo

    getCallLowering(...)
    getRegBankInfo(...)
    getLegalizerInfo(...)
    getInstructionSelector(...)
```

Override GlobalISel API getters by backing with `unique_ptrs`

Initialize GlobalISel

```
extern "C" void LLVMInitializeBPFTarget() {
  // ...
  auto PR = PassRegistry::getPassRegistry();
  initializeGlobalISel(*PR);
}
```

Initialize GlobalISel Passes

```
BPFPassConfig : TargetPassConfig

addIRTranslator(...)
addLegalizeMachineIR(...)
addRegBankSelect(...)
addGlobalInstructionSelect(...)
```

Implement the hooks to provide the required passes

Initialize GlobalISel Passes

```
BPFPassConfig : TargetPassConfig

addIRTranslator(...)
addLegalizeMachineIR(...)
addRegBankSelect(...)
addGlobalInstructionSelect(...)
addPreLegalizeMachineIR(...)
addPreRegBankSelect(...)
addPreGlobalInstructionSelect(...)
```

Optionally add extra passes in between

Build System Bookkeeping

- Update `CMakeLists.txt` with each `.cpp` file we add
- Add the GlobalISel dependency to `LLVMBuild.txt`
- When we add `.td` files, also add `tablegen` targets

Does Anything Work Yet?

```
define void @f() {
  ret void
}
```

```
llc -global-isel -march=bpf -stop-after=irtranslator
```

Does Anything Work Yet?

```
0  llc                      0x00000001076873f8 llvm::sys::PrintStackTrace(llvm::raw_ostream&) + 40
1  llc                      0x0000000107687ab6 SignalHandler(int) + 470
2  libsystem_platform.dylib 0x00007fff730f3f5a _sigtramp + 26
3  libsystem_platform.dylib 0x0000000111a8b250 _sigtramp + 2660856592
4  llc                      0x0000000106e1a2a4 llvm::MachineFunctionPass::runOnFunction(llvm::Function&) + 180
5  llc                      0x00000001070ecc5d llvm::FPPassManager::runOnFunction(llvm::Function&) + 509
6  llc                      0x00000001070eced3 llvm::FPPassManager::runOnModule(llvm::Module&) + 67
7  llc                      0x00000001070ed410 llvm::legacy::PassManagerImpl::run(llvm::Module&) + 944
8  llc                      0x0000000105e69813 compileModule(char**, llvm::LLVMContext&) + 10499
9  llc                      0x0000000105e66c5b main + 1419
10 libdyld.dylib            0x00007fff72e73145 start + 1
Stack dump:
0.      Program arguments: llc -o - -global-isel -march=bpf -stop-after=irtranslator test.ll
1.      Running pass 'Function Pass Manager' on module '/Users/bogner/tmp/t.ll'.
2.      Running pass 'IRTranslator' on function '@f'
segmentation fault
```

```
llc -global-isel -march=bpf -stop-after=irtranslator
```

The IR Translator

Wiring up GISel	`TargetMachine/SubTarget`
`irtranslator`	**`CallLowering`**
`regbankselect`	`RegBankInfo`
`legalizer`	`LegalizerInfo`
`instruction-select`	`InstructionSelector`

Call Lowering

```
BPFCallLowering : CallLowering

  lowerReturn(...)
  lowerFormalArguments(...)
  lowerCall(...)
```

Sketch Call Lowering

```
bool BPFCallLowering::lowerReturn(
    MachineIRBuilder &MIRBuilder,
    const Value *Val, unsigned VReg) const {
  if (VReg)
    return false;
  MIRBuilder.buildInstr(BPF::RET);
  return true;
}
```

Does Anything Work Yet?

```
define void @f() {
  ret void
}
```

```
llc -global-isel -march=bpf -stop-after=irtranslator
```

Does Anything Work Yet?

```
name: test_void
body: |
  bb.0:
    RET
```

```
llc -global-isel -march=bpf -stop-after=irtranslator
```

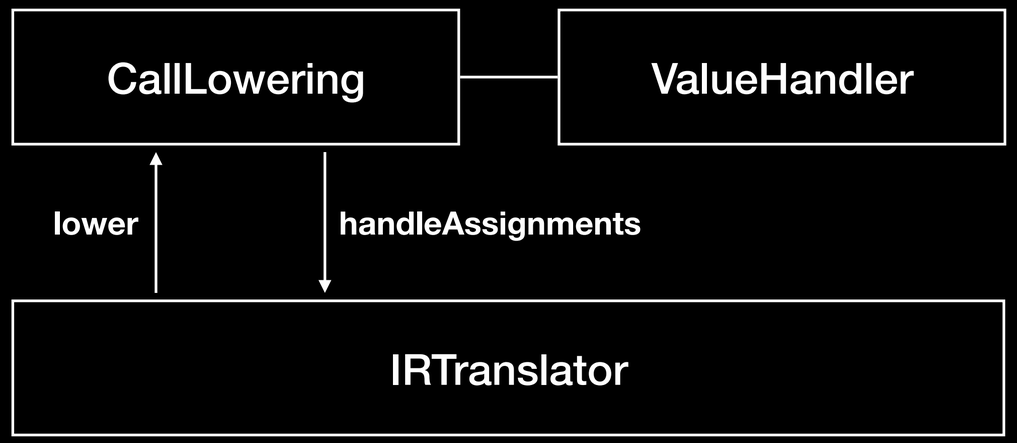
Does Anything Work Yet?

```
; CHECK: name: f
; CHECK: RET
define void @f() {
  ret void
}
```

```
llc -global-isel -march=bpf -stop-after=irtranslator
```

Value Handlers

- Arg handling is mostly uniform
- Decide where the arg/return goes
- Exact details vary
- ValueHandler abstracts differences



Value Handlers

```
BPFHandler : ValueHandler

getStackAddress(…)
assignValueToAddress(…)
assignValueToReg(…)
```

BPF Calling Convention

```
// Promote ints to i64.
CCIfType<[ i8, i16, i32 ], CCPromoteToType<i64>>,

// Pass args in registers.
CCIfType<[i64], CCAssignToReg<[ R1, R2, R3, R4, R5 ]>>,
```

We'll ignore stack handling

Argument Handling

```
FormalArgHandler : BPFHandler

assignValueToReg(...)
```

Argument Handling

```
void assignValueToReg(unsigned ValVReg, unsigned PhysReg,
                      CCValAssign &VA) override {
  switch (VA.getLocInfo()) {
  default:
    MIRBuilder.buildCopy(ValVReg, PhysReg);
    break;
  case CCValAssign::LocInfo::SExt:
  case CCValAssign::LocInfo::ZExt:
  case CCValAssign::LocInfo::AExt: {
    auto Copy = MIRBuilder.buildCopy(LLT{VA.getLocVT()}, PhysReg);
    MIRBuilder.buildTrunc(ValVReg, Copy);
    break;
  }
  }
  MIRBuilder.getMBB().addLiveIn(PhysReg);
}
```

Argument Handling

```
void assignValueToReg(unsigned ValVReg, unsigned PhysReg,
                      CCValAssign &VA) override {
  switch (VA.getLocInfo()) {
  default:
    MIRBuilder.buildCopy(ValVReg, PhysReg);
    break;
  case CCValAssign::LocInfo::SExt:
  case CCValAssign::LocInfo::ZExt:
  case CCValAssign::LocInfo::AExt: {
    auto Copy = MIRBuilder.buildCopy(LLT{VA.getLocVT()}, PhysReg);
    MIRBuilder.buildTrunc(ValVReg, Copy);
    break;
  }
  }
  MIRBuilder.getMBB().addLiveIn(PhysReg);
}
```

Sizes match

```
%ValVReg:_(s32) = COPY PhysReg
```

Argument Handling

```
void assignValueToReg(unsigned ValVReg, unsigned PhysReg,
                      CCValAssign &VA) override {
  switch (VA.getLocInfo()) {
  default:
    MIRBuilder.buildCopy(ValVReg, PhysReg);
    break;
  case CCValAssign::LocInfo::SExt:
  case CCValAssign::LocInfo::ZExt:
  case CCValAssign::LocInfo::AExt: {
    auto Copy = MIRBuilder.buildCopy(LLT{VA.getLocVT()}, PhysReg);
    MIRBuilder.buildTrunc(ValVReg, Copy);
    break;
  }
  }
  MIRBuilder.getMBB().addLiveIn(PhysReg);
}
```

Extended Assign

```
%tmp:_(s64) = COPY PhysReg
```

Argument Handling

```
void assignValueToReg(unsigned ValVReg, unsigned PhysReg,
                      CCValAssign &VA) override {
  switch (VA.getLocInfo()) {
  default:
    MIRBuilder.buildCopy(ValVReg, PhysReg);
    break;
  case CCValAssign::LocInfo::SExt:
  case CCValAssign::LocInfo::ZExt:
  case CCValAssign::LocInfo::AExt: {
    auto Copy = MIRBuilder.buildCopy(LLT{VA.getLocVT()}, PhysReg);
    MIRBuilder.buildTrunc(ValVReg, Copy);
    break;
  }
  }
  MIRBuilder.getMBB().addLiveIn(PhysReg);
}
```

Extended Assign

```
%tmp:_(s64) = COPY PhysReg
%ValVReg:_(s32) = G_TRUNC %tmp(s64)
```

Argument Handling

```
void assignValueToReg(unsigned ValVReg, unsigned PhysReg,
                      CCValAssign &VA) override {
  switch (VA.getLocInfo()) {
  default:
    MIRBuilder.buildCopy(ValVReg, PhysReg);
    break;
  case CCValAssign::LocInfo::SExt:
  case CCValAssign::LocInfo::ZExt:
  case CCValAssign::LocInfo::AExt: {
    auto Copy = MIRBuilder.buildCopy(LLT{VA.getLocVT()}, PhysReg);
    MIRBuilder.buildTrunc(ValVReg, Copy);
    break;
  }
  }
  MIRBuilder.getMBB().addLiveIn(PhysReg);
}
```

Extended Assign

```
%tmp:_(s64) = COPY PhysReg
%ValVReg:_(s32) = G_TRUNC %tmp(s64)
```

Lower Formal Args

```
bool BPFCallLowering::lowerFormalArguments(MachineIRBuilder &MIRBuilder,
                                           const Function &F,
                                           ArrayRef<unsigned> VRegs) {
  // …
  SmallVector<ArgInfo, 8> InArgs;
  unsigned i = 0;
  for (auto &Arg : F.args()) {
    ArgInfo OrigArg{VRegs[i], Arg.getType()};
    setArgFlags(OrigArg, i + AttributeList::FirstArgIndex, DL, F);
    InArgs.push_back(OrigArg);
    ++i;
  }
  FormalArgHandler Handler(MIRBuilder, MRI, CC_BPF64);
  return handleAssignments(MIRBuilder, InArgs, Handler);
}
```

Lower Formal Args

```
bool BPFCallLowering::lowerFormalArguments(MachineIRBuilder &MIRBuilder,
                                           const Function &F,
                                           ArrayRef<unsigned> VRegs) {
  // …
  SmallVector<ArgInfo, 8> InArgs;
  unsigned i = 0;
  for (auto &Arg : F.args()) {
    ArgInfo OrigArg{VRegs[i], Arg.getType()};
    setArgFlags(OrigArg, i + AttributeList::FirstArgIndex, DL, F);
    InArgs.push_back(OrigArg);
    ++i;
  }
  FormalArgHandler Handler(MIRBuilder, MRI, CC_BPF64);
  return handleAssignments(MIRBuilder, InArgs, Handler);
}
```

Lower Formal Args

```
bool BPFCallLowering::lowerFormalArguments(MachineIRBuilder &MIRBuilder,
                                           const Function &F,
                                           ArrayRef<unsigned> VRegs) {
  // …
  SmallVector<ArgInfo, 8> InArgs;
  unsigned i = 0;
  for (auto &Arg : F.args()) {
    ArgInfo OrigArg{VRegs[i], Arg.getType()};
    setArgFlags(OrigArg, i + AttributeList::FirstArgIndex, DL, F);
    InArgs.push_back(OrigArg);
    ++i;
  }
  FormalArgHandler Handler(MIRBuilder, MRI, CC_BPF64);
  return handleAssignments(MIRBuilder, InArgs, Handler);
}
```

Lower Formal Args

```
bool BPFCallLowering::lowerFormalArguments(MachineIRBuilder &MIRBuilder,
                                           const Function &F,
                                           ArrayRef<unsigned> VRegs) {
  // …
  SmallVector<ArgInfo, 8> InArgs;
  unsigned i = 0;
  for (auto &Arg : F.args()) {
    ArgInfo OrigArg{VRegs[i], Arg.getType()};
    setArgFlags(OrigArg, i + AttributeList::FirstArgIndex, DL, F);
    InArgs.push_back(OrigArg);
    ++i;
  }
  FormalArgHandler Handler(MIRBuilder, MRI, CC_BPF64);
  return handleAssignments(MIRBuilder, InArgs, Handler);
}
```

Lower Formal Args

```
bool BPFCallLowering::lowerFormalArguments(MachineIRBuilder &MIRBuilder,
                                           const Function &F,
                                           ArrayRef<unsigned> VRegs) {
  // …
  SmallVector<ArgInfo, 8> InArgs;
  unsigned i = 0;
  for (auto &Arg : F.args()) {
    ArgInfo OrigArg{VRegs[i], Arg.getType()};
    setArgFlags(OrigArg, i + AttributeList::FirstArgIndex, DL, F);
    InArgs.push_back(OrigArg);
    ++i;
  }
  FormalArgHandler Handler(MIRBuilder, MRI, CC_BPF64);
  return handleAssignments(MIRBuilder, InArgs, Handler);
}
```

Lower Formal Args

```
bool BPFCallLowering::lowerFormalArguments(MachineIRBuilder &MIRBuilder,
                                           const Function &F,
                                           ArrayRef<unsigned> VRegs) {
  // …
  SmallVector<ArgInfo, 8> InArgs;
  unsigned i = 0;
  for (auto &Arg : F.args()) {
    ArgInfo OrigArg{VRegs[i], Arg.getType()};
    setArgFlags(OrigArg, i + AttributeList::FirstArgIndex, DL, F);
    InArgs.push_back(OrigArg);
    ++i;
  }
  FormalArgHandler Handler(MIRBuilder, MRI, CC_BPF64);
  return handleAssignments(MIRBuilder, InArgs, Handler);
}
```

Test Lowering

```
; CHECK: [[IN:%[0-9]+]]:_(s64) = COPY %r1
; CHECK: RET
define void @f(i64 %a) {
  ret void
}
```

```
llc -global-isel -march=bpf -stop-after=irtranslator
```

Test Lowering

```
; CHECK: [[CP:%[0-9]+]]:_(s64) = COPY %r1
; CHECK: [[IN:%[0-9]+]]:_(s32) = G_TRUNC [[CP]]
; CHECK: RET
define void @f(i32 %a) {
  ret void
}
```

```
llc -global-isel -march=bpf -stop-after=irtranslator
```

Lower Return

```
struct OutgoingHandler : public BPFHandler {
  OutgoingHandler(…, MachineInstrBuilder &MIB)
      : BPFHandler(…), MIB(MIB) {}
  MachineInstrBuilder &MIB;
  // …
};
```

Lower Return

```
void assignValueToReg(unsigned ValVReg, unsigned PhysReg,
                      CCValAssign &VA) override {
  unsigned ExtReg = extendRegister(ValVReg, VA);
  MIRBuilder.buildCopy(PhysReg, ExtReg);
  MIB.addUse(PhysReg, RegState::Implicit);
}
```

Lower Return

```
void assignValueToReg(unsigned ValVReg, unsigned PhysReg,
                      CCValAssign &VA) override {
  unsigned ExtReg = extendRegister(ValVReg, VA);
  MIRBuilder.buildCopy(PhysReg, ExtReg);
  MIB.addUse(PhysReg, RegState::Implicit);
}
```

Lower Return

```
void assignValueToReg(unsigned ValVReg, unsigned PhysReg,
                      CCValAssign &VA) override {
  unsigned ExtReg = extendRegister(ValVReg, VA);
  MIRBuilder.buildCopy(PhysReg, ExtReg);
  MIB.addUse(PhysReg, RegState::Implicit);
}
```

Lower Return

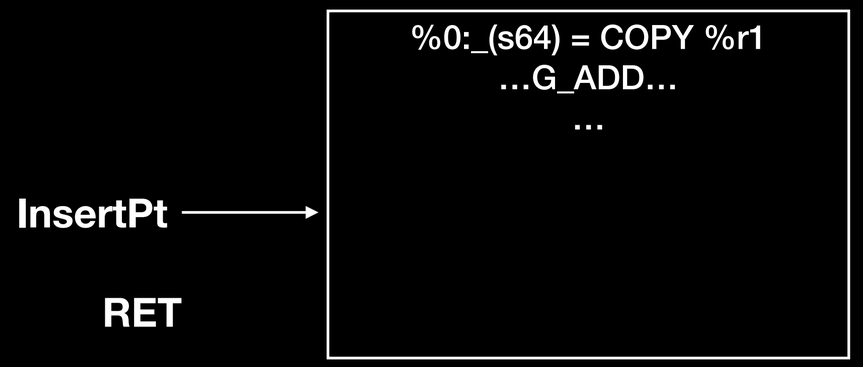
```
void assignValueToReg(unsigned ValVReg, unsigned PhysReg,
                      CCValAssign &VA) override {
  unsigned ExtReg = extendRegister(ValVReg, VA);
  MIRBuilder.buildCopy(PhysReg, ExtReg);
  MIB.addUse(PhysReg, RegState::Implicit);
}
```

Lower Return

```
bool BPFCallLowering::lowerReturn(MachineIRBuilder &MIRBuilder,
                                  const Value *Val, unsigned VReg) {
  auto MIB = MIRBuilder.buildInstrNoInsert(BPF::RET);
  bool Success = true;
  if (VReg) {
    // …
    ArgInfo OrigArg{VReg, Val->getType()};
    setArgFlags(OrigArg, AttributeList::ReturnIndex, DL, F);
    OutgoingHandler Handler(MIRBuilder, MRI, RetCC_BPF64, MIB);
    Success = handleAssignments(MIRBuilder, {OrigArg}, Handler);
  }
  MIRBuilder.insertInstr(MIB);
  return Success;
}
```

Lower Return

```
bool BPFCallLowering::lowerReturn(MachineIRBuilder &MIRBuilder,
                                  const Value *Val, unsigned VReg) {
  auto MIB = MIRBuilder.buildInstrNoInsert(BPF::RET);
  bool Success = true;
  if (VReg) {
    // …
    ArgInfo OrigArg{VReg, Val->getType()};
    setArgFlags(OrigArg, AttributeList::ReturnIndex, DL, F);
    OutgoingHandler Handler(MIRBuilder, MRI, RetCC_BPF64, MIB);
    Success = handleAssignments(MIRBuilder, {OrigArg}, Handler);
  }
  MIRBuilder.insertInstr(MIB);
  return Success;
}
```



Lower Return

```
bool BPFCallLowering::lowerReturn(MachineIRBuilder &MIRBuilder,
                                  const Value *Val, unsigned VReg) {
  auto MIB = MIRBuilder.buildInstrNoInsert(BPF::RET);
  bool Success = true;
  if (VReg) {
    // …
    ArgInfo OrigArg{VReg, Val->getType()};
    setArgFlags(OrigArg, AttributeList::ReturnIndex, DL, F);
    OutgoingHandler Handler(MIRBuilder, MRI, RetCC_BPF64, MIB);
    Success = handleAssignments(MIRBuilder, {OrigArg}, Handler);
  }
  MIRBuilder.insertInstr(MIB);
  return Success;
}
```

Lower Return

```
bool BPFCallLowering::lowerReturn(MachineIRBuilder &MIRBuilder,
                                  const Value *Val, unsigned VReg) {
  auto MIB = MIRBuilder.buildInstrNoInsert(BPF::RET);
  bool Success = true;
  if (VReg) {
    // …
    ArgInfo OrigArg{VReg, Val->getType()};
    setArgFlags(OrigArg, AttributeList::ReturnIndex, DL, F);
    OutgoingHandler Handler(MIRBuilder, MRI, RetCC_BPF64, MIB);
    Success = handleAssignments(MIRBuilder, {OrigArg}, Handler);
  }
  MIRBuilder.insertInstr(MIB);
  return Success;
}
```

InsertPt →

RET implicit %r0

```
%0:_(s64) = COPY %r1
…G_ADD…
…

%r0 = COPY %VReg
```

Lower Return

```
bool BPFCallLowering::lowerReturn(MachineIRBuilder &MIRBuilder,
                                  const Value *Val, unsigned VReg) {
  auto MIB = MIRBuilder.buildInstrNoInsert(BPF::RET);
  bool Success = true;
  if (VReg) {
    // …
    ArgInfo OrigArg{VReg, Val->getType()};
    setArgFlags(OrigArg, AttributeList::ReturnIndex, DL, F);
    OutgoingHandler Handler(MIRBuilder, MRI, RetCC_BPF64, MIB);
    Success = handleAssignments(MIRBuilder, {OrigArg}, Handler);
  }
  MIRBuilder.insertInstr(MIB);
  return Success;
}
```

InsertPt →
RET implicit %r0

%0:_(s64) = COPY %r1
…G_ADD…
…

%1:_(s64) = G_ANYEXT %VReg
%r0 = COPY %1

Lower Return

```
bool BPFCallLowering::lowerReturn(MachineIRBuilder &MIRBuilder,
                                  const Value *Val, unsigned VReg) {
  auto MIB = MIRBuilder.buildInstrNoInsert(BPF::RET);
  bool Success = true;
  if (VReg) {
    // …
    ArgInfo OrigArg{VReg, Val->getType()};
    setArgFlags(OrigArg, AttributeList::ReturnIndex, DL, F);
    OutgoingHandler Handler(MIRBuilder, MRI, RetCC_BPF64, MIB);
    Success = handleAssignments(MIRBuilder, {OrigArg}, Handler);
  }
  MIRBuilder.insertInstr(MIB);
  return Success;
}
```

InsertPt →

```
%0:_(s64) = COPY %r1
…G_ADD…
…

%1:_(s64) = G_ANYEXT %VReg
%r0 = COPY %1
BPF::RET implicit %r0
```

Test ArgLowering

```
; CHECK: [[IN:%[0-9]+]]:_(s64) = COPY %r1
; CHECK: %r0 = COPY [[IN]]
; CHECK: RET implicit %r0
define i64 @f(i64 %a) {
  ret i64 %a
}
```

```
llc -global-isel -march=bpf -stop-after=irtranslator
```

Test ArgLowering

```
; CHECK: [[TMP:%[0-9]+]]:_(s64) = COPY %r1
; CHECK: [[IN:%[0-9]+]]:_(s32) = G_TRUNC [[TMP]]
; CHECK: [[EXT:%[0-9]+]]:_(s64) = G_ANYEXT [[IN]]
; CHECK: %r0 = COPY [[EXT]]
define i32 @f(i32 %a) {
  ret i32 %a
}
```

```
llc -global-isel -march=bpf -stop-after=irtranslator
```

Register Banks

Wiring up GISel	`TargetMachine/SubTarget`
`irtranslator`	`CallLowering`
`regbankselect`	**`RegBankInfo`**
`legalizer`	`LegalizerInfo`
`instruction-select`	`InstructionSelector`

Register Banks

- Group register classes, ignoring size and type
- Different banks imply transferring values is costly
- A typical split is general purpose vs floating point

Define Register Banks

```
def AnyGPRRegBank : RegisterBank<"AnyGPR", [GPR]>;
```

Define a bank for BPF's GPRs in `BPFRegisterBanks.td`

More Register Classes

GPR32
FPR16
DDD
FPR8
GPR32sp
GPR32all
FPR32
GPR64sp
FPR128
DD
DDDD
GPR64sponly
GPR32sponly
GPR64common
CCR
tcGPR64
QQQ
GPR64
GPR32common
GPR64all
QQQQ
QQ
FPR64

AArch64 is a good example with more register classes

AArch64 Register Banks

GPR	FPR	CC
GPR32	FPR8	CCR
GPR32sp	FPR16	
GPR32common	FPR32	
GPR32sponly	FPR64	
GPR32all	FPR128	
GPR64	DD	
GPR64sp	DDD	
GPR64common	DDDD	
GPR64sponly	QQ	
GPR64all	QQQ	
tcGPR64	QQQQ	

AArch64 Register Banks

```
/// General Purpose Registers: W, X.
def GPRRegBank : RegisterBank<"GPR", [GPR64all]>;

/// Floating Point/Vector Registers: B, H, S, D, Q.
def FPRRegBank : RegisterBank<"FPR", [QQQQ]>;

/// Conditional register: NZCV.
def CCRegBank : RegisterBank<"CC", [CCR]>;
```

We only need to specify the largest register class to define each bank

Generated Bank Info

```
class BPFGenRegisterBankInfo : public RegisterBankInfo {
protected:
#define GET_TARGET_REGBANK_CLASS
#include "BPFGenRegisterBank.inc"
};
```

Register Bank Mapping

```
BPFRegisterBankInfo : BPFGenRegisterBankInfo

    getRegBankFromRegClass(...)
    getInstrMapping(...)
    getInstrAlternativeMappings(...)
```

Register Bank Mapping

```
BPFRegisterBankInfo : BPFGenRegisterBankInfo

    getRegBankFromRegClass(...)
    getInstrMapping(...)
    getInstrAlternativeMappings(...)
```

Given a register class, return the register bank

Register Bank Mapping

```
BPFRegisterBankInfo : BPFGenRegisterBankInfo

    getRegBankFromRegClass(...)
    getInstrMapping(...)
    getInstrAlternativeMappings(...)
```

getInstrMapping

```
const auto &Mapping = getInstrMappingImpl(MI);
if (Mapping.isValid())
  return Mapping;
```

Leverage `getInstrMappingImpl` to handle generic instructions

getInstrMapping

```
SmallVector<const ValueMapping *, 8> ValMappings(NumOperands);
for (unsigned Idx = 0; Idx < NumOperands; ++Idx) {
  if (MI.getOperand(Idx).isReg()) {
    LLT Ty = MRI.getType(MI.getOperand(Idx).getReg());
    auto Size = Ty.getSizeInBits();
    ValMappings[Idx] = &getValueMapping(0, Size, BPF::AnyGPRRegBank);
  }
}
```

Map each operand to an appropriate bank for target instructions

Test Bank Selection

```
; CHECK-LABEL: name: defaultMapping
; CHECK: %[[CP:[0-9]+]]:anygpr(s64) = COPY %r1
; CHECK: %[[ADD:[0-9]+]]:anygpr(s64) = G_ADD %[[CP]], %[[CP]]
%0:_(s64) = COPY %r1
%1:_(s64) = G_ADD %0, %0
```

```
llc -march=bpf -global-isel -run-pass=regbankselect
```

getInstrAlternativeMappings

```
case TargetOpcode::G_OR: {
  InstructionMappings AltMappings;
  AltMappings.push_back(getInstructionMapping(
      /*ID*/ 1, /*Cost*/ 1, getValueMapping(PMI_FirstGPR, Size),
      /*NumOperands*/ 3));
  AltMappings.push_back(getInstructionMapping(
      /*ID*/ 2, /*Cost*/ 1, getValueMapping(PMI_FirstFPR, Size),
      /*NumOperands*/ 3));
  return AltMappings;
}
```

In AArch64, 32 and 64-bit "`or`" map equally well on FPR or GPR

getInstrAlternativeMappings

```
case TargetOpcode::G_OR: {
  InstructionMappings AltMappings;
  AltMappings.push_back(getInstructionMapping(
      /*ID*/ 1, /*Cost*/ 1, getValueMapping(PMI_FirstGPR, Size),
      /*NumOperands*/ 3));
  AltMappings.push_back(getInstructionMapping(
      /*ID*/ 2, /*Cost*/ 1, getValueMapping(PMI_FirstFPR, Size),
      /*NumOperands*/ 3));
  return AltMappings;
}
```

In AArch64, 32 and 64-bit "`or`" map equally well on FPR or GPR

getInstrAlternativeMappings

```
case TargetOpcode::G_OR: {
  InstructionMappings AltMappings;
  AltMappings.push_back(getInstructionMapping(
      /*ID*/ 1, /*Cost*/ 1, getValueMapping(PMI_FirstGPR, Size),
      /*NumOperands*/ 3));
  AltMappings.push_back(getInstructionMapping(
      /*ID*/ 2, /*Cost*/ 1, getValueMapping(PMI_FirstFPR, Size),
      /*NumOperands*/ 3));
  return AltMappings;
}
```

In AArch64, 32 and 64-bit "`or`" map equally well on FPR or GPR

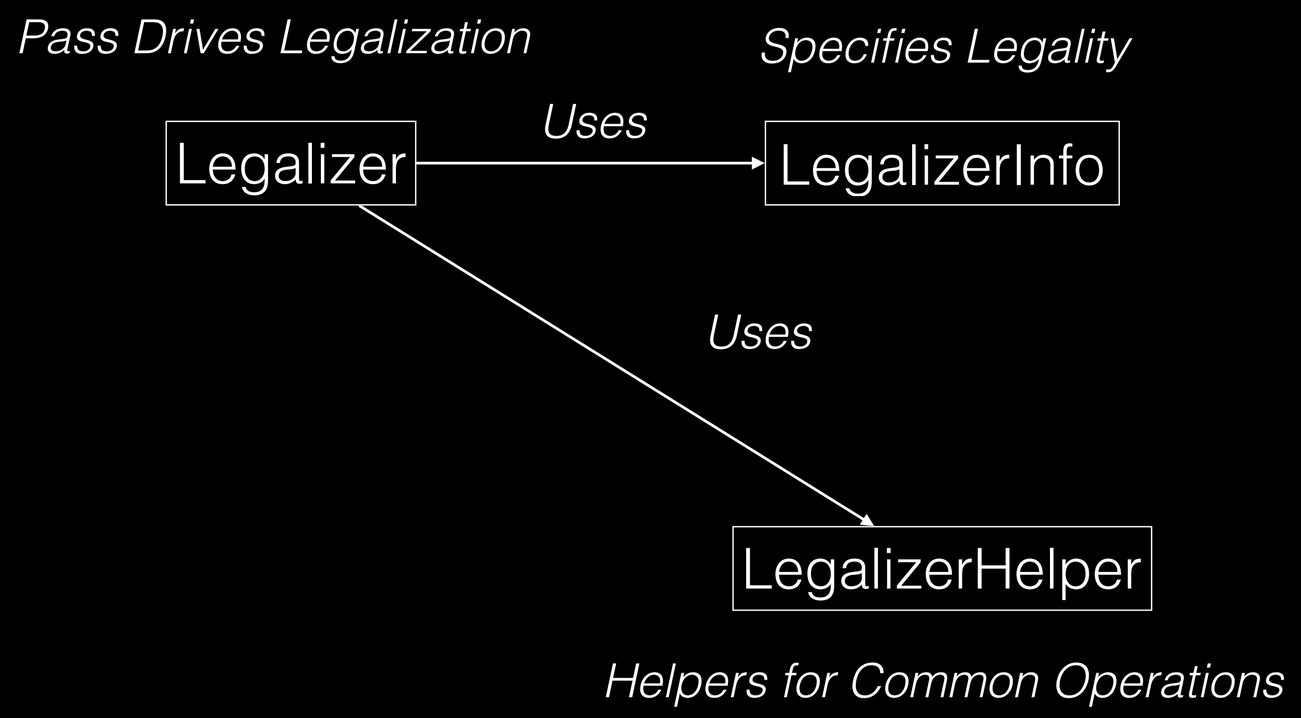
Legalizer

Wiring up GISel	`TargetMachine/SubTarget`
`irtranslator`	`CallLowering`
`regbankselect`	`RegBankInfo`
`legalizer`	**`LegalizerInfo`**
`instruction-select`	`InstructionSelector`

Legalizer

- Transform gMIR into ***legal*** instructions
- Legal is defined as
  - Selectable by target
  - Operating on vregs that can be loaded/stored
  - No SelectionDAG like type legalization

Legalization



Legality Specifiers

```
setAction({Opcode, [OperandIdx,] Ty}, LegalizeAction)
```

Legality Specifiers

```
setAction({Opcode, [OperandIdx,] Ty}, LegalizeAction)
```

```
setAction({G_ADD, s64}, Legal); // Want 64 bit Dest
```

```
%2:_(s64) = G_ADD %0, %1
```

```
setAction({G_ADD, s32}, WidenScalar);
```

```
%3:_(s32) = …
%4:_(s32) = G_ADD %3, %3
```

Legality Specifiers

```
setAction({Opcode, [OperandIdx,] Ty}, LegalizeAction)
```

```
setAction({G_ADD, s64}, Legal); // Want 64 bit Dest
```

```
%2:_(s64) = G_ADD %0, %1
```

```
setAction({G_ADD, s32}, WidenScalar);
```

```
%3:_(s32) = …
%4:_(s32) = G_ADD %3, %3
```

Legality Specifiers

```
setAction({Opcode, [OperandIdx,] Ty}, LegalizeAction)
```

```
setAction({G_ADD, s64}, Legal); // Want 64 bit Dest
```

```
setAction({G_ADD, s32}, WidenScalar);
```

```
%2:_(s64) = G_ADD %0, %1

%3:_(s32) = …
%5:_(s64) = G_ANYEXT %3(s32)
%6:_(s64) = G_ADD %5, %5
```

Legality Specifiers

```
setAction({Opcode, [OperandIdx,] Ty}, LegalizeAction)
```

```
setAction({G_ANYEXT, 0, s64}, Legal);
```

```
setAction({G_ANYEXT, 1, s32}, Legal);
```

```
%2:_(s64) = G_ADD %0, %1

%3:_(s32) = …
%5:_(s64) = G_ANYEXT %3(s32)
%6:_(s64) = G_ADD %5, %5
```

Legalization in BPF

- Set LegalizerAction in the constructor.
- Implement ***legalizeCustom*** for custom hook if required.

BPFLegalizerInfo

```
BPFLegalizerInfo::BPFLegalizerInfo() {
  using namespace TargetOpcode;
  const LLT p0 = LLT::pointer(0, 64);
  // …
  const LLT s32 = LLT::scalar(32);
  const LLT s64 = LLT::scalar(64);

  for (auto Ty : {p0, s1, s8, s16, s32, s64})
    setAction({G_IMPLICIT_DEF, Ty}, Legal);

  setAction({G_PHI, s64}, Legal);
  for (auto Ty : {s1, s8, s16, s32})
    setAction({G_PHI, Ty}, WidenScalar);

  // …

  computeTables();
}
```

BPFLegalizerInfo

```
BPFLegalizerInfo::BPFLegalizerInfo() {
  using namespace TargetOpcode;
  const LLT p0 = LLT::pointer(0, 64);
  // …
  const LLT s32 = LLT::scalar(32);
  const LLT s64 = LLT::scalar(64);

  for (auto Ty : {p0, s1, s8, s16, s32, s64})
    setAction({G_IMPLICIT_DEF, Ty}, Legal);

  setAction({G_PHI, s64}, Legal);
  for (auto Ty : {s1, s8, s16, s32})
    setAction({G_PHI, Ty}, WidenScalar);

  // …

  computeTables();
}
```

BPFLegalizerInfo

```
BPFLegalizerInfo::BPFLegalizerInfo() {
  using namespace TargetOpcode;
  const LLT p0 = LLT::pointer(0, 64);
  // …
  const LLT s32 = LLT::scalar(32);
  const LLT s64 = LLT::scalar(64);

  for (auto Ty : {p0, s1, s8, s16, s32, s64})
    setAction({G_IMPLICIT_DEF, Ty}, Legal);

  setAction({G_PHI, s64}, Legal);
  for (auto Ty : {s1, s8, s16, s32})
    setAction({G_PHI, Ty}, WidenScalar);

  // …

  computeTables();
}
```

BPFLegalizerInfo

```
BPFLegalizerInfo::BPFLegalizerInfo() {
  using namespace TargetOpcode;
  const LLT p0 = LLT::pointer(0, 64);
  // …
  const LLT s32 = LLT::scalar(32);
  const LLT s64 = LLT::scalar(64);

  for (auto Ty : {p0, s1, s8, s16, s32, s64})
    setAction({G_IMPLICIT_DEF, Ty}, Legal);

  setAction({G_PHI, s64}, Legal);
  for (auto Ty : {s1, s8, s16, s32})
    setAction({G_PHI, Ty}, WidenScalar);

  // …

  computeTables();
}
```

BPFLegalizerInfo

```
BPFLegalizerInfo::BPFLegalizerInfo() {
  using namespace TargetOpcode;
  const LLT p0 = LLT::pointer(0, 64);
  // …
  const LLT s32 = LLT::scalar(32);
  const LLT s64 = LLT::scalar(64);

  for (auto Ty : {p0, s1, s8, s16, s32, s64})
    setAction({G_IMPLICIT_DEF, Ty}, Legal);

  setAction({G_PHI, s64}, Legal);
  for (auto Ty : {s1, s8, s16, s32})
    setAction({G_PHI, Ty}, WidenScalar);

  // …

  computeTables();
}
```

Test Legalizer

```
; CHECK: [[IN:%[0-9]+]]:_(s64) = COPY %r1
; CHECK: %r0 = COPY [[IN]]
; CHECK: RET implicit %r0
%0:_(s64) = COPY %r1
%r0 = COPY %0(s64)
RET implicit %r0
```

llc -march=bpf -global-isel -run-pass=legalizer

Test Legalizer

```
%0:_(s64) = COPY %r1
%1:_(s64) = G_ADD %0, %0
%r0 = COPY %1(s64)
RET implicit %r0
```

```
LLVM ERROR: unable to legalize instruction
```

BPFLegalizerInfo

```
setAction({G_ADD, s64}, Legal);
for (const auto &Ty : {s1, s8, s16, s32})
  setAction({G_ADD, Ty}, WidenScalar);
```

BPFLegalizerInfo

```
setAction({G_ADD, s64}, Legal);
for (const auto &Ty : {s1, s8, s16, s32})
  setAction({G_ADD, Ty}, WidenScalar);
```

Test Legalizer

```
; CHECK: [[ADD:%[0-9]+]]:_(s64) = G_ADD
%0:_(s64) = COPY %r1
%1:_(s64) = G_ADD %0, %0
%r0 = COPY %1(s64)
RET implicit %r0
```

Test Legalizer

```
; CHECK: [[ADD:%[0-9]+]]:_(s64) = G_ADD
%0:_(s64) = COPY %r1
%1:_(s32) = G_TRUNC %0
%2:_(s32) = G_ADD %1, %1
; ...
```

Custom Legalization

- Implement legalizeCustom method
- Specify LegalizationAction as Custom
- Legalize G_SELECT into pseudo instruction (BPF_SELECT_CC)

Custom Legalization

```
%2:_(s64) = G_ICMP <pred>, %0(s64), %1(s64)
%3:_(s1) = G_TRUNC %2(s64)
%4:_(s64) = G_SELECT %3(s1), %5(s64), %6
```

```
%4:_(s64) = BPF_SELECT_CC %0, %1, <pred>, %5, %6
```

Custom Legalization

```
%2:_(s64) = G_ICMP <pred>, %0(s64), %1(s64)
%3:_(s1) = G_TRUNC %2(s64)
%4:_(s64) = G_SELECT %3(s1), %5(s64), %6
```

```
%4:_(s64) = BPF_SELECT_CC %0, %1, <pred>, %5, %6
```

Custom Legalization

```
def BPF_SELECT_CC
    : Pseudo<(outs type1:$dst),
             (ins type0:$lhs, type0:$rhs, i64imm:$cc,
                  type1:$true, type1:$false),
             “BPF_SELECT_CC\t$dst, $lhs, $rhs, $cc, $true, $false",
             []>;
```

Custom Legalization

```
setAction({G_SELECT, 1, s1}, Legal);
for (const auto &Ty : {s1, s8, s16, s32})
  setAction({G_SELECT, Ty}, WidenScalar);
setAction({G_SELECT, s64}, Custom);
```

legalizeCustom

```
bool BPFLegalizerInfo::legalizeCustom(…) {
  switch (MI.getOpcode()) {
  default:
    llvm_unreachable("Illegal Opcode");
  case TargetOpcode::G_SELECT:
    return legalizeCustomSelect(MI, MRI, MIRBuilder);
  }
}
```

legalizeCustom

```
bool BPFLegalizerInfo::legalizeCustomSelect(
    MachineInstr &MI, MachineRegisterInfo &MRI,
    MachineIRBuilder &MIRBuilder) const {

    // …

}
```

```
%2:_(s64) = G_ICMP <pred>, %0(s64), %1(s64)
%3:_(s1) = G_TRUNC %2(s64)
%4:_(s64) = G_SELECT %3(s1), %5(s64), %6
```

legalizeCustom

```
MachineOperand &Op0 = MI.getOperand(0);
LLT DstTy = MRI.getType(Op0.getReg());
MachineOperand &CmpOp = MI.getOperand(1);
MachineOperand &Res1 = MI.getOperand(2);
MachineOperand &Res2 = MI.getOperand(3);
```

```
%2:_(s64) = G_ICMP <pred>, %0(s64), %1(s64)
%3:_(s1) = G_TRUNC %2(s64)
%4:_(s64) = G_SELECT %3(s1), %5(s64), %6
```

legalizeCustom

```
MachineOperand &Op0 = MI.getOperand(0);
LLT DstTy = MRI.getType(Op0.getReg());
MachineOperand &CmpOp = MI.getOperand(1);
MachineOperand &Res1 = MI.getOperand(2);
MachineOperand &Res2 = MI.getOperand(3);
```

```
%2:_(s64) = G_ICMP <pred>, %0(s64), %1(s64)
%3:_(s1) = G_TRUNC %2(s64)
%4:_(s64) = G_SELECT %3(s1), %5(s64), %6
```

legalizeCustom

```
MachineInstr *DefMI = MRI.getVRegDef(CmpOp.getReg());
if (DefMI->getOpcode() == TargetOpcode::G_TRUNC)
  DefMI = MRI.getVRegDef(DefMI->getOperand(1).getReg());
if (DefMI->getOpcode() != TargetOpcode::G_ICMP)
  return false;
```

```
%2:_(s64) = G_ICMP <pred>, %0(s64), %1(s64)
%3:_(s1) = G_TRUNC %2(s64)
%4:_(s64) = G_SELECT %3(s1), %5(s64), %6
```

legalizeCustom

```
MachineInstr *DefMI = MRI.getVRegDef(CmpOp.getReg());
if (DefMI->getOpcode() == TargetOpcode::G_TRUNC)
  DefMI = MRI.getVRegDef(DefMI->getOperand(1).getReg());
if (DefMI->getOpcode() != TargetOpcode::G_ICMP)
  return false;
```

```
%2:_(s64) = G_ICMP <pred>, %0(s64), %1(s64)
%3:_(s1) = G_TRUNC %2(s64)
%4:_(s64) = G_SELECT %3(s1), %5(s64), %6
```

legalizeCustom

```
MachineInstr *DefMI = MRI.getVRegDef(CmpOp.getReg());
if (DefMI->getOpcode() == TargetOpcode::G_TRUNC)
  DefMI = MRI.getVRegDef(DefMI->getOperand(1).getReg());
if (DefMI->getOpcode() != TargetOpcode::G_ICMP)
  return false;
```

```
%2:_(s64) = G_ICMP <pred>, %0(s64), %1(s64)
%3:_(s1) = G_TRUNC %2(s64)
%4:_(s64) = G_SELECT %3(s1), %5(s64), %6
```

legalizeCustom

```
MachineInstr *DefMI = MRI.getVRegDef(CmpOp.getReg());
if (DefMI->getOpcode() == TargetOpcode::G_TRUNC)
  DefMI = MRI.getVRegDef(DefMI->getOperand(1).getReg());
if (DefMI->getOpcode() != TargetOpcode::G_ICMP)
  return false;
```

```
%2:_(s64) = G_ICMP <pred>, %0(s64), %1(s64)
%3:_(s1) = G_TRUNC %2(s64)
%4:_(s64) = G_SELECT %3(s1), %5(s64), %6
```

legalizeCustom

```
MachineInstr *DefMI = MRI.getVRegDef(CmpOp.getReg());
if (DefMI->getOpcode() == TargetOpcode::G_TRUNC)
  DefMI = MRI.getVRegDef(DefMI->getOperand(1).getReg());
if (DefMI->getOpcode() != TargetOpcode::G_ICMP)
  return false;
```

```
%2:_(s64) = G_ICMP <pred>, %0(s64), %1(s64)
%3:_(s1) = G_TRUNC %2(s64)
%4:_(s64) = G_SELECT %3(s1), %5(s64), %6
```

legalizeCustom

```
MachineOperand &C1 = DefMI->getOperand(2);
MachineOperand &C2 = DefMI->getOperand(3);
MachineOperand &PredOp = DefMI->getOperand(1);
```

```
%2:_(s64) = G_ICMP <pred>, %0(s64), %1(s64)
%3:_(s1) = G_TRUNC %2(s64)
%4:_(s64) = G_SELECT %3(s1), %5(s64), %6
```

legalizeCustom

```
MachineOperand &C1 = DefMI->getOperand(2);
MachineOperand &C2 = DefMI->getOperand(3);
MachineOperand &PredOp = DefMI->getOperand(1);
```

```
%2:_(s64) = G_ICMP <pred>, %0(s64), %1(s64)
%3:_(s1) = G_TRUNC %2(s64)
%4:_(s64) = G_SELECT %3(s1), %5(s64), %6
```

legalizeCustom

```
MachineOperand &C1 = DefMI->getOperand(2);
MachineOperand &C2 = DefMI->getOperand(3);
MachineOperand &PredOp = DefMI->getOperand(1);
```

```
%2:_(s64) = G_ICMP <pred>, %0(s64), %1(s64)
%3:_(s1) = G_TRUNC %2(s64)
%4:_(s64) = G_SELECT %3(s1), %5(s64), %6
```

legalizeCustom

```
// Get the dest reg
unsigned DstReg = MI.getOperand(0).getReg();
MIRBuilder.setInstr(MI);
```

```
%2:_(s64) = G_ICMP <pred>, %0(s64), %1(s64)
%3:_(s1) = G_TRUNC %2(s64)
%4:_(s64) = G_SELECT %3(s1), %5(s64), %6
```

legalizeCustom

```
// Get the dest reg
unsigned DstReg = MI.getOperand(0).getReg();
MIRBuilder.setInstr(MI);
```

```
%2:_(s64) = G_ICMP <pred>, %0(s64), %1(s64)
%3:_(s1) = G_TRUNC %2(s64)
%4:_(s64) = G_SELECT %3(s1), %5(s64), %6
```

legalizeCustom

```
// Get the dest reg
unsigned DstReg = MI.getOperand(0).getReg();
MIRBuilder.setInstr(MI);
```

```
%2:_(s64) = G_ICMP <pred>, %0(s64), %1(s64)
%3:_(s1) = G_TRUNC %2(s64)
%4:_(s64) = G_SELECT %3(s1), %5(s64), %6
```

InsertPt →

legalizeCustom

```
auto Cst = MIRBuilder.buildConstant(
    DstTy, (unsigned)PredKind);
```

```
%2:_(s64) = G_ICMP <pred>, %0(s64), %1(s64)
%3:_(s1) = G_TRUNC %2(s64)
%7:_(s64) = G_CONSTANT pred_kind
%4:_(s64) = G_SELECT %3(s1), %5(s64), %6
```

InsertPt →

legalizeCustom

```
MIRBuilder.buildInstr(BPF::BPF_SELECT_CC, DstReg)
        .addReg(Cmp1.getReg())
        .addReg(Cmp2.getReg())
        .addReg(Cst->getOperand(0).getReg())
        .addReg(Res1.getReg())
        .addReg(Res2.getReg());
```

```
%2:_(s64) = G_ICMP <pred>, %0(s64), %1(s64)
%3:_(s1) = G_TRUNC %2(s64)
%7:_(s64) = G_CONSTANT pred_kind
%4:_(s64) = G_SELECT %3(s1), %5(s64), %6
```

InsertPt →

legalizeCustom

```
MIRBuilder.buildInstr(BPF::BPF_SELECT_CC, DstReg)
        .addReg(Cmp1.getReg())
        .addReg(Cmp2.getReg())
        .addReg(Cst->getOperand(0).getReg())
        .addReg(Res1.getReg())
        .addReg(Res2.getReg());
```

```
%2:_(s64) = G_ICMP <pred>, %0(s64), %1(s64)
%3:_(s1) = G_TRUNC %2(s64)
%7:_(s64) = G_CONSTANT pred_kind
%4:_(s64) = BPF_SELECT_CC
%4:_(s64) = G_SELECT %3(s1), %5(s64), %6
```

InsertPt → %4:_(s64) = BPF_SELECT_CC

legalizeCustom

```
MIRBuilder.buildInstr(BPF::BPF_SELECT_CC, DstReg)
        .addReg(Cmp1.getReg())
        .addReg(Cmp2.getReg())
        .addReg(Cst->getOperand(0).getReg())
        .addReg(Res1.getReg())
        .addReg(Res2.getReg());
```

```
%2:_(s64) = G_ICMP <pred>, %0(s64), %1(s64)
%3:_(s1) = G_TRUNC %2(s64)
%7:_(s64) = G_CONSTANT pred_kind
%4:_(s64) = BPF_SELECT_CC, %0(s64)
%4:_(s64) = G_SELECT %3(s1), %5(s64), %6
```

InsertPt → `%4:_(s64) = BPF_SELECT_CC, %0(s64)`

legalizeCustom

```
MIRBuilder.buildInstr(BPF::BPF_SELECT_CC, DstReg)
        .addReg(Cmp1.getReg())
        .addReg(Cmp2.getReg())
        .addReg(Cst->getOperand(0).getReg())
        .addReg(Res1.getReg())
        .addReg(Res2.getReg());
```

```
%2:_(s64) = G_ICMP <pred>, %0(s64), %1(s64)
%3:_(s1) = G_TRUNC %2(s64)
%7:_(s64) = G_CONSTANT pred_kind
%4:_(s64) = BPF_SELECT_CC, %0(s64), %1
%4:_(s64) = G_SELECT %3(s1), %5(s64), %6
```

InsertPt → `%4:_(s64) = BPF_SELECT_CC, %0(s64), %1`

legalizeCustom

```
MIRBuilder.buildInstr(BPF::BPF_SELECT_CC, DstReg)
        .addReg(Cmp1.getReg())
        .addReg(Cmp2.getReg())
        .addReg(Cst->getOperand(0).getReg())
        .addReg(Res1.getReg())
        .addReg(Res2.getReg());
```

```
%2:_(s64) = G_ICMP <pred>, %0(s64), %1(s64)
%3:_(s1) = G_TRUNC %2(s64)
%7:_(s64) = G_CONSTANT pred_kind
%4:_(s64) = BPF_SELECT_CC, %0(s64), %1, %7(s64), %5, %6
%4:_(s64) = G_SELECT %3(s1), %5(s64), %6
```

InsertPt → %4:_(s64) = BPF_SELECT_CC, %0(s64), %1, %7(s64), %5, %6

legalizeCustom

```
MI.eraseFromParent();
```

```
%2:_(s64) = G_ICMP <pred>, %0(s64), %1(s64)
%3:_(s1) = G_TRUNC %2(s64)
%7:_(s64) = G_CONSTANT pred_kind
%4:_(s64) = BPF_SELECT_CC, %0(s64), %1, %7(s64), %5, %6
%4:_(s64) = G_SELECT %3(s1), %5(s64), %6
```

InsertPt → `%4:_(s64) = BPF_SELECT_CC, %0(s64), %1, %7(s64), %5, %6`

legalizeCustom

```
MI.eraseFromParent();
```

```
%2:_(s64) = G_ICMP <pred>, %0(s64), %1(s64)
%3:_(s1) = G_TRUNC %2(s64)
%7:_(s64) = G_CONSTANT pred_kind
%4:_(s64) = BPF_SELECT_CC, %0(s64), %1, %7(s64), %5, %6
```

InsertPt → `%4:_(s64) = BPF_SELECT_CC, %0(s64), %1, %7(s64), %5, %6`

legalizeCustom

```
auto &RB = MF.getSubtarget()
             .getRegBankInfo()
 ->getRegBankFromRegClass(BPF::GPRRegClass);
MRI.setRegBank(DstReg, RB);
```

```
%2:_(s64) = G_ICMP <pred>, %0(s64), %1(s64)
%3:_(s1) = G_TRUNC %2(s64)
%7:_(s64) = G_CONSTANT pred_kind
%4:_(s64) = BPF_SELECT_CC, %0(s64), %1, %7(s64), %5, %6
```

InsertPt →

legalizeCustom

```
auto &RB = MF.getSubtarget()
             .getRegBankInfo()
 ->getRegBankFromRegClass(BPF::GPRRegClass);
MRI.setRegBank(DstReg, RB);
```

```
%2:_(s64) = G_ICMP <pred>, %0(s64), %1(s64)
%3:_(s1) = G_TRUNC %2(s64)
%7:_(s64) = G_CONSTANT pred_kind
%4:anygpr(s64) = BPF_SELECT_CC, %0(s64), %1, %7(s64), %5, %6
```

InsertPt

legalizeCustom

```
return true;
```

```
%2:_(s64) = G_ICMP <pred>, %0(s64), %1(s64)
%3:_(s1) = G_TRUNC %2(s64)
%7:_(s64) = G_CONSTANT pred_kind
%4:anygpr(s64) = BPF_SELECT_CC, %0(s64), %1, %7(s64), %5, %6
```

InsertPt

Test Custom Legalization

```
; CHECK-LABEL: name: checkcmplegal
; CHECK: [[A:%[0-9]+]]:_(s64) = COPY %r1
; CHECK: [[B:%[0-9]+]]:_(s64) = COPY %r2
; CHECK: [[CST:%[0-9]+]]:_(s64) = G_CONSTANT
; CHECK: [[R:%[0-9]+]]:anygpr(s64) = BPF_SELECT_CC [[A]](s64), [[B]], [[CST]], [[A]], [[B]]
; CHECK: %r0 = COPY [[R]]

%2:_(s64) = COPY %r1
%3:_(s64) = COPY %r2
%4:_(s64) = G_ICMP intpred(sgt), %2(s64), %3
%5:_(s1)  = G_TRUNC %4(s64)
%6:_(s64) = G_SELECT %5(s1), %2(s64), %3
%r0 = COPY %5(s64)
RET implicit %r0
```

Good Practices

- ✓ Incrementally add instructions and write small targeted tests
- ✓ Always use MachineIRBuilder for building instructions
- ✗ Don’t erase any other instruction besides the current
- ✗ Do not allow legality to be conditional

Instruction Selector

Wiring up GISel	`TargetMachine/SubTarget`
`irtranslator`	`CallLowering`
`regbankselect`	`RegBankInfo`
`legalizer`	`LegalizerInfo`
`instruction-select`	**`InstructionSelector`**

Instruction Selection

- Importing rules from SelectionDAG
  - Converting PatLeaf and ComplexPattern
  - Handling custom SDNodes
- What to do when the importer fails
  - Custom Selection

Instruction Selection

```
BPFInstructionSelector : InstructionSelector

        select(...)
        selectImpl(...)
```

Implement `select` and use the tablegen-erated `selectImpl`

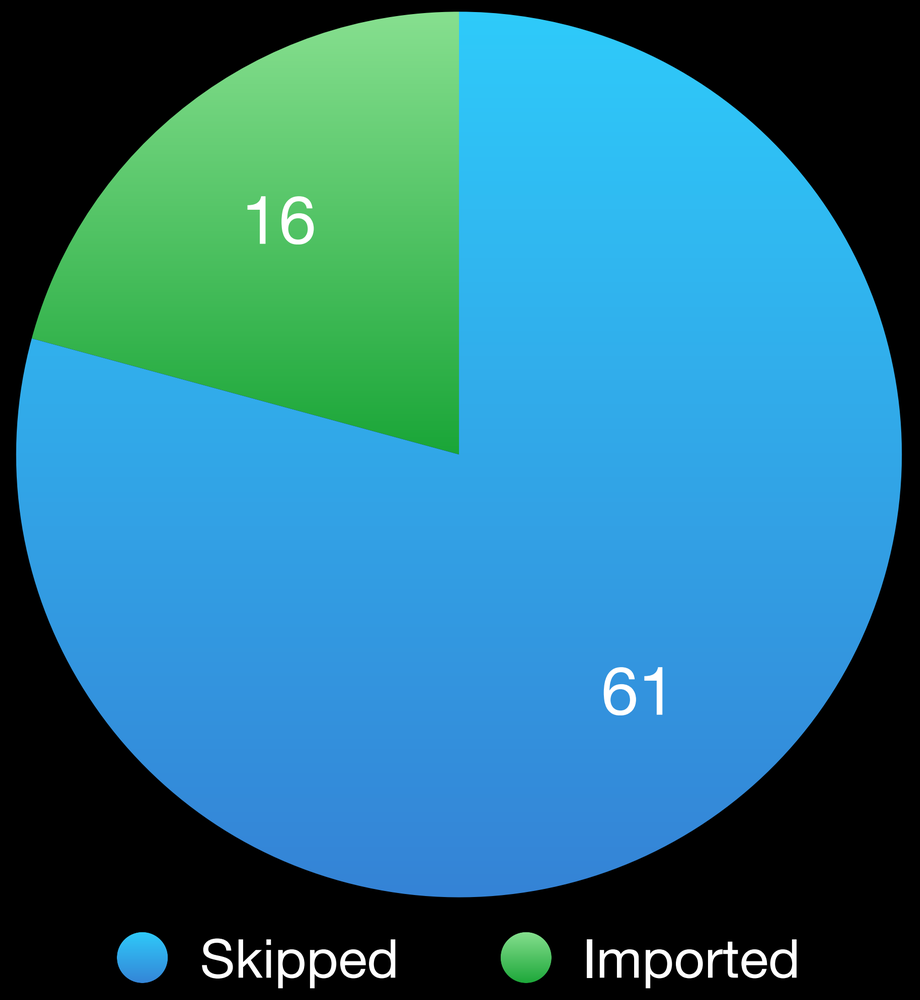
Implement select

```
bool BPFInstructionSelector::select(MachineInstr &I) const {
  // Ignore COPY's: the register allocator will handle them.
  if (Opcode == TargetOpcode::COPY)
    return true;
  if (selectImpl(I))
    return true;
  return false;
}
```

Imported Rule Statistics

```
$ rm lib/Target/BPF/BPFGenGlobalISel.inc*
$ ninja -v lib/Target/BPF/BPFGenGlobalISel.inc
$ llvm-tblgen -gen-global-isel ... --stats

===-------------------------------------------------------------------------===
                          ... Statistics Collected ...
===-------------------------------------------------------------------------===

16 gisel-emitter - Number of patterns emitted
16 gisel-emitter - Number of patterns imported from SelectionDAG
61 gisel-emitter - Number of SelectionDAG imports skipped
77 gisel-emitter - Total number of patterns
```

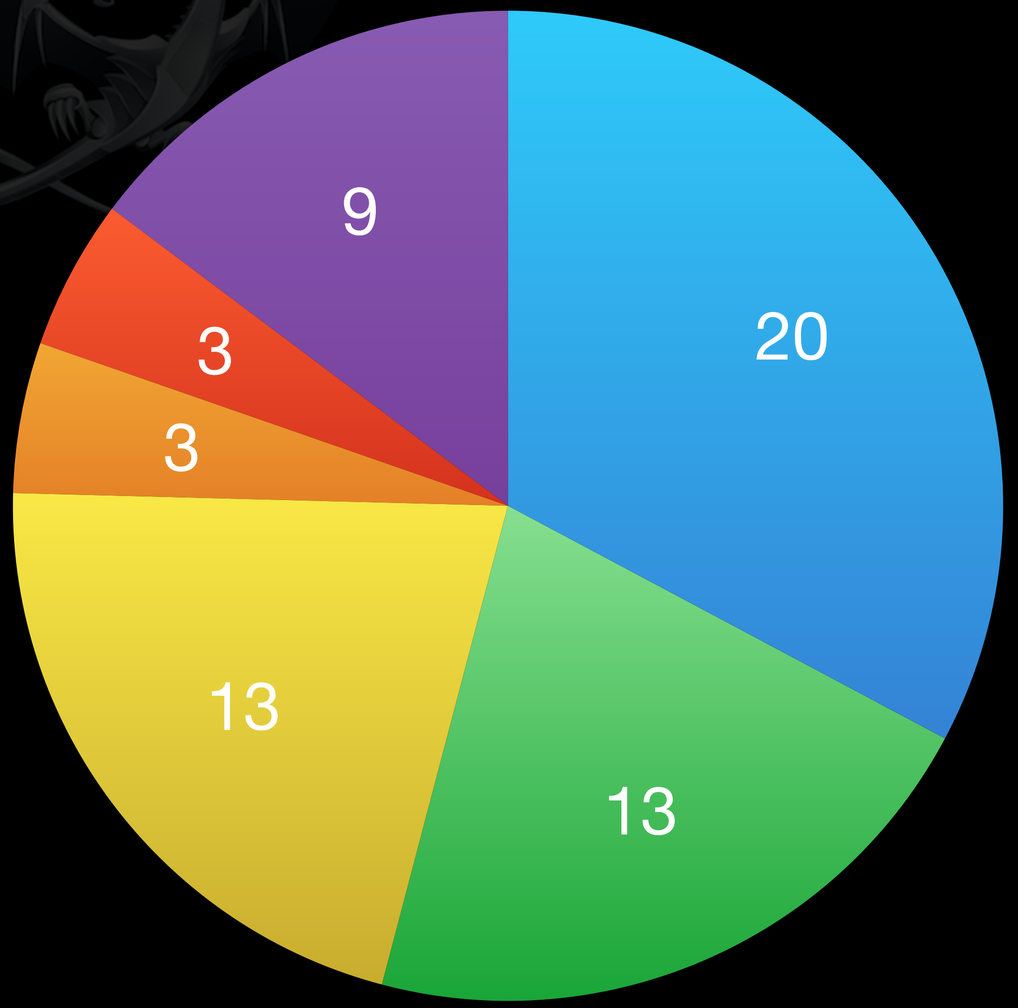
Initial Imports



Import Failures

```
$ rm lib/Target/BPF/BPFGenGlobalISel.inc*
$ ninja -v lib/Target/BPF/BPFGenGlobalISel.inc
$ llvm-tblgen -gen-global-isel ... --warn-on-skipped-patterns

Included from lib/Target/BPF/BPF.td:15:
lib/Target/BPF/BPFInstrInfo.td:352:1: warning: Skipped pattern: Src pattern
root isn't a trivial operator ...
def STW : STOREi64<0x0, "u32", truncstorei32>;
^
```

Import Failures



- No equivalent Instruction (BPFISD::BR_CC)
- Src pattern child has predicate
- Src pattern root isn't a trivial operator
- Src pattern results and Dst MI defs differ
- No equivalent Instruction (BPFISD::CALL)
- Others

Import PatLeaf

```
def i64immSExt32 : PatLeaf<i64, [{return isInt<32>(N->getSExtValue());}]>;
```

Src pattern child has predicate (i64immSExt32)

Import PatLeaf

```
def i64immSExt32 : PatLeaf<i64, [{return isInt<32>(N->getSExtValue());}]>;
```

↓

```
def i64immSExt32 : ImmLeaf<i64, [{return isInt<32>(Imm);}]>;
```

Use int64_t instead of SDNode

Import PatLeaf

```
def BPF_CC_EQ : PatLeaf<i64, [{return N->getZExtValue() == ISD::SETEQ;}]>;
```

Src pattern child has predicate (BPF_CC_EQ)

Import PatLeaf

```
def BPF_CC_EQ : PatLeaf<i64, [{return N->getZExtValue() == ISD::SETEQ;}]>;
```

↓

```
def BPF_CC_EQ : IntImmLeaf<i64, [{return Imm->getZExtValue() == ISD::SETEQ;}]>;
```

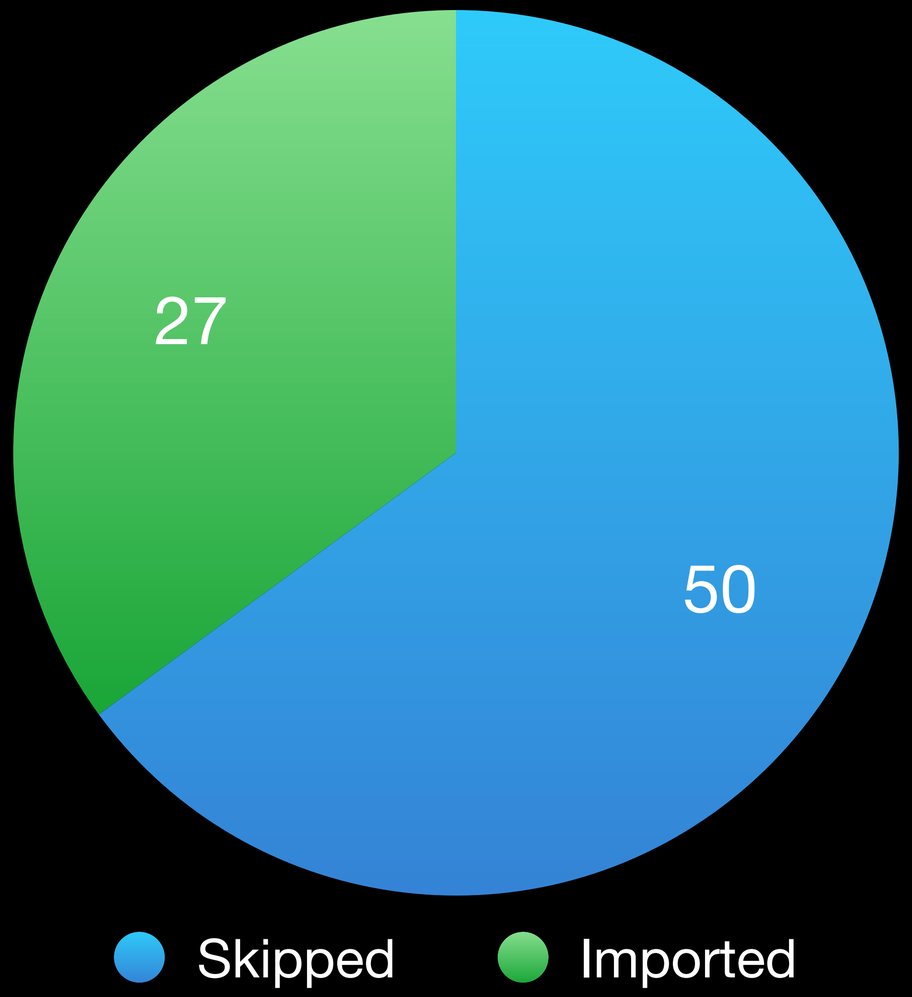
Use APInt instead of SDNode

Test Instruction Selection

```
; CHECK: %{{[0-9]}}:gpr = LD_imm64 1234
%0:anygpr(s64) = G_CONSTANT i64 1234
%r0 = COPY %0(s64)
```

llc -march=bpf -global-isel -run-pass=instruction-select

Import PatLeaf



Map Custom ISD Nodes

```
def BPF_BR_CC :
      Pseudo<(outs),
             (ins i64imm:$cc, type0:$lhs, type0:$rhs, brtarget:$target),
             "BR_CC_PSEUDO\t$cc, $lhs, $rhs, $target", []>;
def : GINodeEquiv<BPF_BR_CC, BPFbrcc>;
```

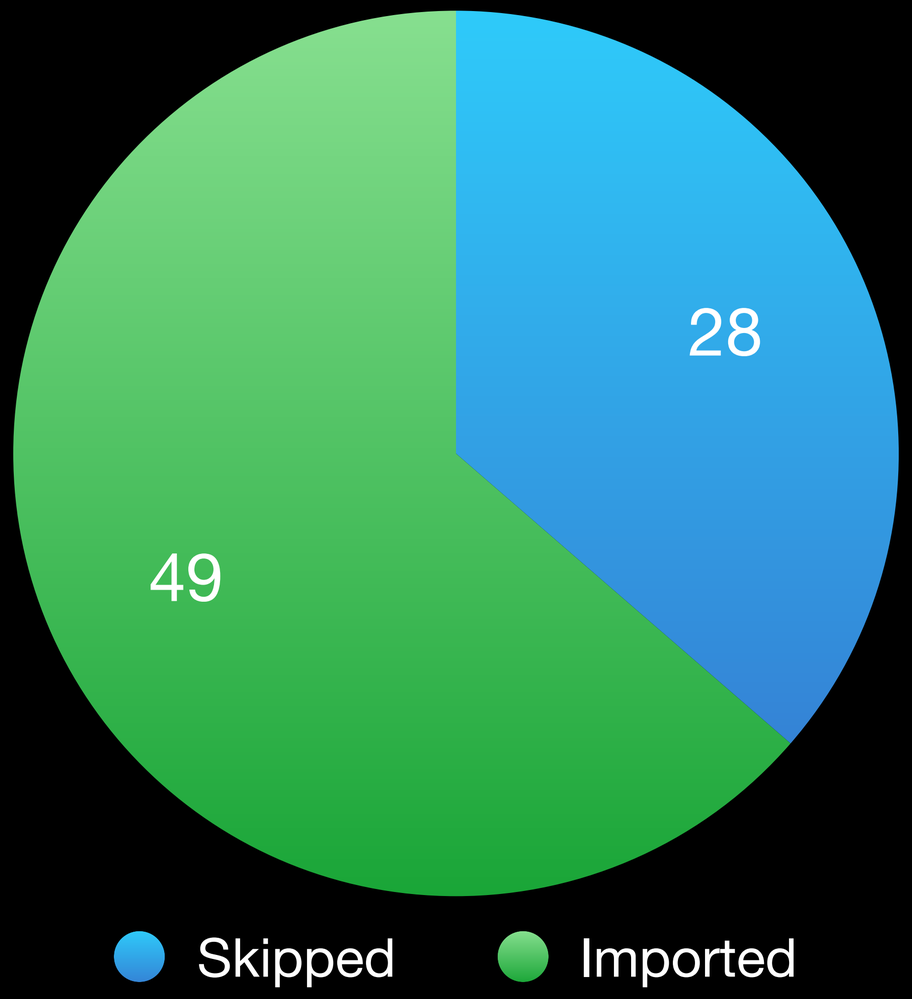
Pattern operator lacks an equivalent Instruction (BPFISD::BR_CC)

Map Custom ISD Nodes

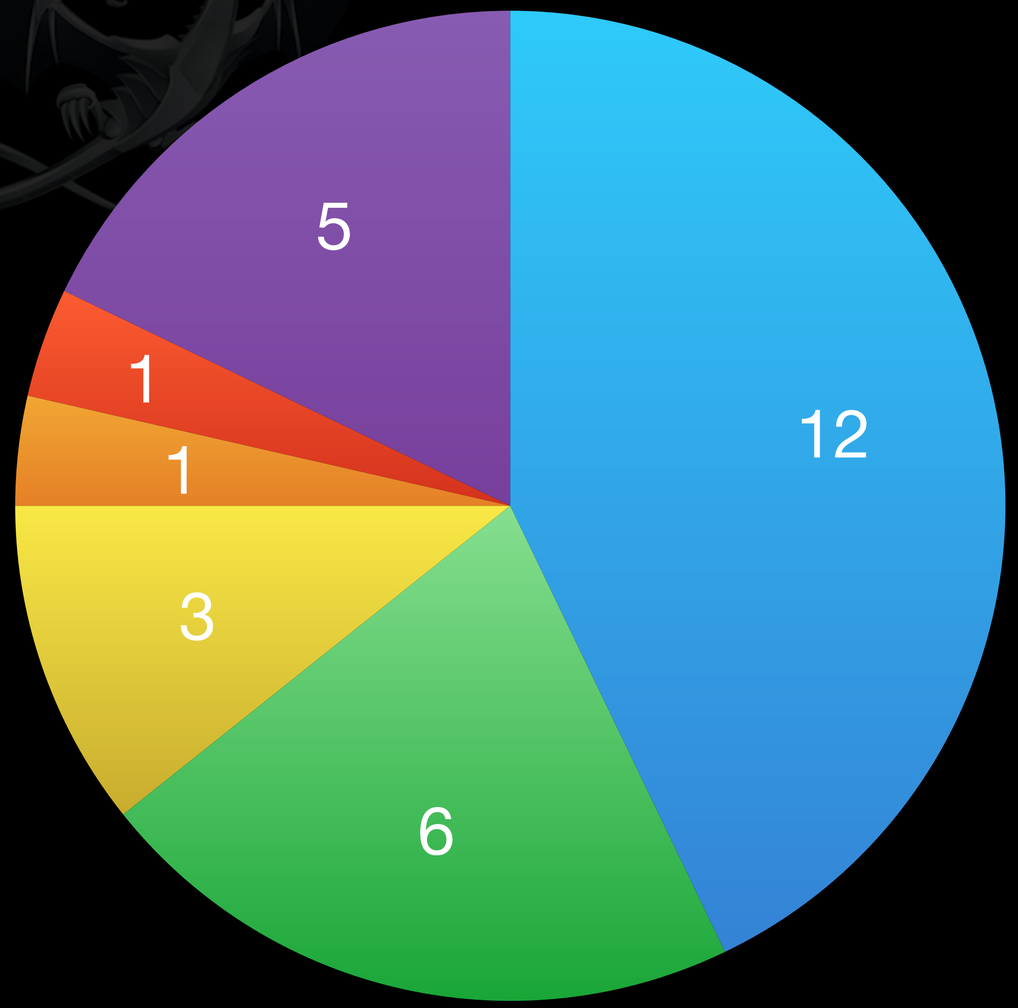
```
def BPF_SELECT_CC
    : Pseudo<(outs type1:$dst),
             (ins type0:$lhs, type0:$rhs, i64imm:$cc,
                  type1:$true, type1:$false),
             "BPF_SELECT_CC\t$dst, $lhs, $rhs, $cc, $true, $false",
             []>;
def : GINodeEquiv<BPF_SELECT_CC, BPFselectcc>;
```

Pattern operator lacks an equivalent Instruction (BPFISD::SELECT_CC)

Map Custom ISD Nodes



Import Failures



- Src pattern root isn't a trivial operator
- Src pattern results and dst MI defs are different
- No equivalent Instruction (BPFISD::CALL)
- Unable to deduce gMIR opcode to handle Src
- ComplexPattern (ADDRri) not mapped
- Others

Import ComplexPattern

```
def ADDRri : ComplexPattern<i64, 2, "SelectAddr", [], []>;

def gi_ADDRri : GIComplexOperandMatcher<s64, "selectAddr">,
                GIComplexPatternEquiv<ADDRri>;
```

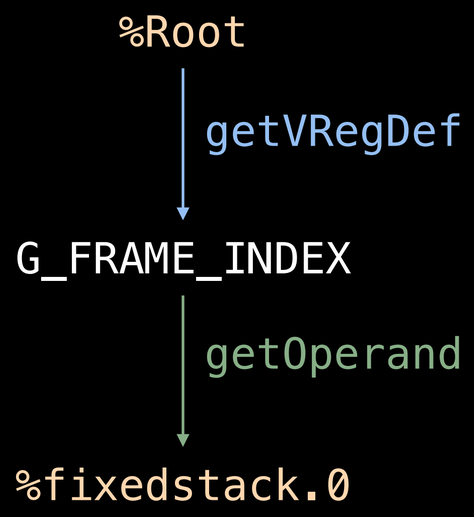
Patterns To Match

```
%Root = G_FRAME_INDEX %fixedstack.0

%0    = G_CONSTANT i64 simm16
%Root = G_GEP %1, %0

%0    = G_FRAME_INDEX %fixedstack.0
%1    = G_CONSTANT i64 simm16
%Root = G_GEP %0, %1

%Root = Any Register
```

Cases To Handle

```
%Root = G_FRAME_INDEX %fixedstack.0

%0    = G_CONSTANT i64 simm16
%Root = G_GEP %1, %0

%Root = G_GLOBAL_VALUE ...

%0    = G_FRAME_INDEX %fixedstack.0
%1    = G_CONSTANT i64 simm16
%Root = G_GEP %0, %1

%Root = Any Register
```

G_FRAME_INDEX

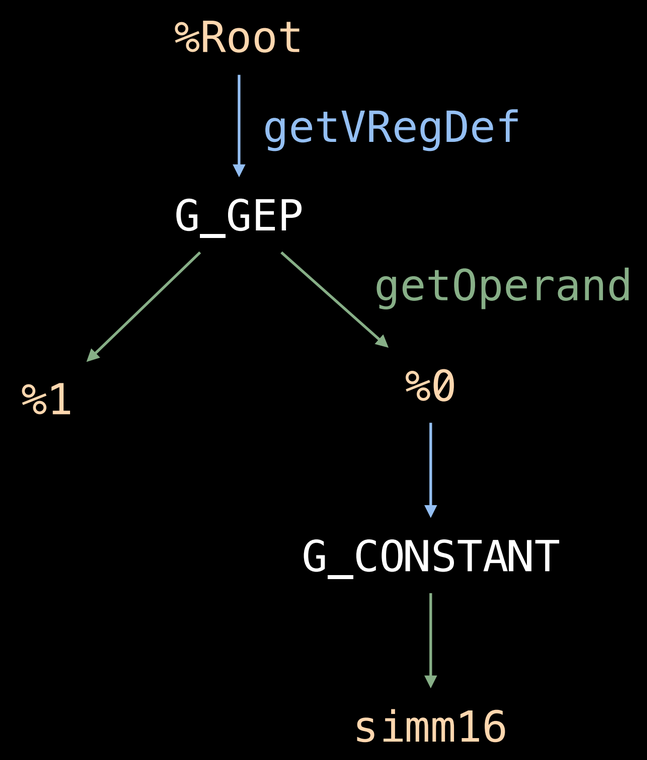


G_FRAME_INDEX

```
%0 = INST <Ops>, %fixedstack.0, i64 0
```

```
return {{ [=](auto &MIB) { MIB.add(FrameIndex->getOperand(1)); },
          [=](auto &MIB) { MIB.addImm(0); } }};
```

Base + Offset



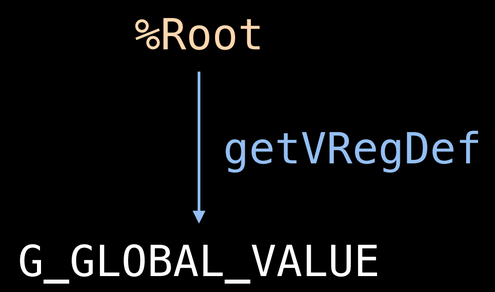

```
isBaseWithConstantOffset(Root, MRI)
```

Base + Offset

```
%0 = INST <Ops>, %1, i64 simm16
```

```
return {{ [=](auto &MIB) { MIB.add(Gep->getOperand(1)); },
          [=](auto &MIB) { MIB.addImm(Constant->getOperand(1).getImm()); }
       }};
```

G_GLOBAL_VALUE




```
return None;
```

Any Pointer

`%Root`

Any Pointer

```
%0 = INST <Ops>, %Root, i64 0
```

```
return {{ [=](auto &MIB) { MIB.add(Root); },
          [=](auto &MIB) { MIB.addImm(0); } }};
```

Common Predicates

```
isBaseWithConstantOffset(...)
isOperandImmEqual(...)
isObviouslySafeToFold(...)
```

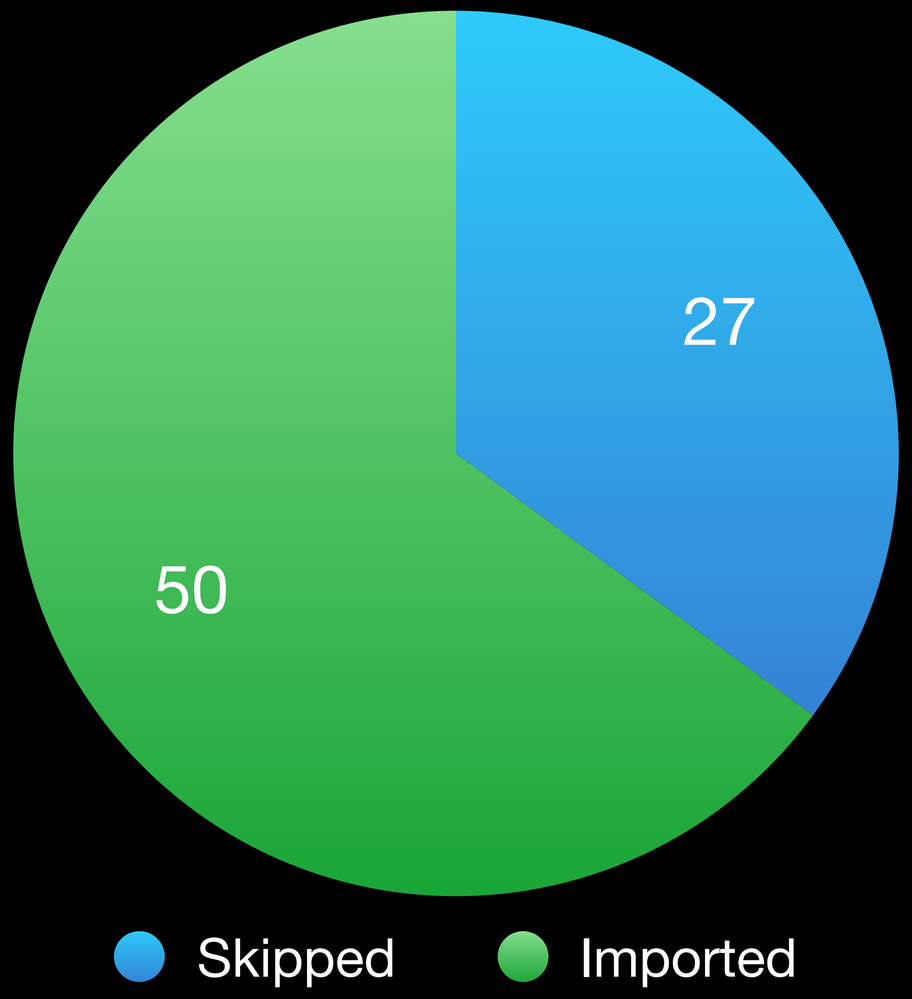
Good Practices

✗ Don't modify MIR in the predicate

✓ Capture locals by value in a renderer lambda

✓ Only create instructions in a renderer lambda

Import ComplexPattern



Known Issues

```
warning: Src pattern root isn't a trivial operator
         (Has a predicate (unindexedload unindexed-loadextload
                           extloadextloadi16 LoadMemVT=i16),
          first-failing:extload)
```

[sz]extload/extload/truncstore/atomic-load/atomic-store in development

Known Issues

```
def : Pat<(i64 (and (i64 GPR:$src), 0xffffffff)),
          (SRL_ri (SLL_ri (i64 GPR:$src), 32), 32)>;

warning: Dst pattern child isn't a leaf node or an MBB
```

Development for multiple instruction emission will start soon

Custom Selection

```
def : Pat<(i64 (and (i64 GPR:$src), 0xffffffff)),
          (SRL_ri (SLL_ri (i64 GPR:$src), 32), 32)>;
```

Rule Priority

```
bool BPFInstructionSelector::select(MachineInstr &I) const {
  // ...



  if (selectImpl(I))
    return true;



  // ...
  return false;
}
```

Rule Priority

```
bool BPFInstructionSelector::select(MachineInstr &I) const {
  // ...



  if (selectImpl(I))
    return true;

  // Implement (G_AND $dst, $src1, (G_CONSTANT i64 0xffffffff) here?

  // ...
  return false;
}
```

Rule Priority

```
bool BPFInstructionSelector::select(MachineInstr &I) const {
  // ...

  // Implement (G_AND $dst, $src1, (G_CONSTANT i64 0xffffffff) here

  if (selectImpl(I)) // Contains (G_AND $dst, $src1, $src2)
    return true;



  // ...
  return false;
}
```

Test Custom Selection

```
; CHECK: %[[T0:[0-9]+]]:gpr = COPY %r0
; CHECK: %[[T1:[0-9]+]]:gpr = SLL_ri %[[T0]], 32
; CHECK: %[[T2:[0-9]+]]:gpr = SRL_ri %[[T1]], 32
; CHECK: %r0 = COPY %[[T2]]
liveins: %r0
%0:anygpr(s64) = COPY %r0
%1:anygpr(s64) = G_CONSTANT i64 4294967295
%2:anygpr(s64) = G_AND %0, %1
%r0 = COPY %2(s64)
```

```
llc -march=bpf -global-isel -run-pass=instruction-select
```

Custom Selection

```
if (Opcode == TargetOpcode::G_AND) {
  MachineOperand *Dst = &I.getOperand(0);
  MachineOperand *LHS = &I.getOperand(1);
  MachineOperand *RHS = &I.getOperand(2);
```

```
%0:anygpr   = G_CONSTANT i64 0xffffffff
%dst:anygpr = G_AND %src:anygpr, %0
```

Custom Selection

```
if (MRI.getType(Dst->getReg()) == LLT::scalar(64) &&
    RBI.getRegBank(Dst->getReg(), MRI, TRI)->getID()
        == BPF::AnyGPRRegBankID) {
```

```
%0:anygpr   = G_CONSTANT i64 0xffffffff
%dst:anygpr = G_AND %src:anygpr, %0
```

Custom Selection

```
bool LHSIsMask = isOperandImmEqual(*LHS, 0xffffffff, MRI);
if (LHSIsMask)
  std::swap(LHS, RHS);
if (LHSIsMask || isOperandImmEqual(*RHS, 0xffffffff, MRI)) {
```

```
%0:anygpr   = G_CONSTANT i64 0xffffffff
%dst:anygpr = G_AND %src:anygpr, %0
```

Custom Selection

```
MachineIRBuilder Builder(I);
auto MIB = Builder.buildInstr(BPF::SLL_ri, &BPF::GPRRegClass)
              .add(*LHS)
              .addImm(32);

constrainSelectedInstRegOperands(*MIB, TII, TRI, RBI);
```

```
%0:anygpr   = G_CONSTANT i64 0xffffffff
%dst:anygpr = G_AND %src:anygpr, %0
%1:gpr      = SLL_ri %src:gpr, i64 32
```

Custom Selection

```
MIB = Builder.buildInstr(BPF::SRL_ri)
            .add(*Dst)
            .addUse(MIB->getOperand(0).getReg())
            .addImm(32);

constrainSelectedInstRegOperands(*MIB, TII, TRI, RBI);
```

```
%0:anygpr   = G_CONSTANT i64 0xffffffff
%dst:anygpr = G_AND %src:anygpr, %0
%1:gpr      = SLL_ri %src:gpr, i64 32
%dst:gpr    = SRL_ri %1,       i64 32
```

Custom Selection

```
I.eraseFromParent();
return true;
```

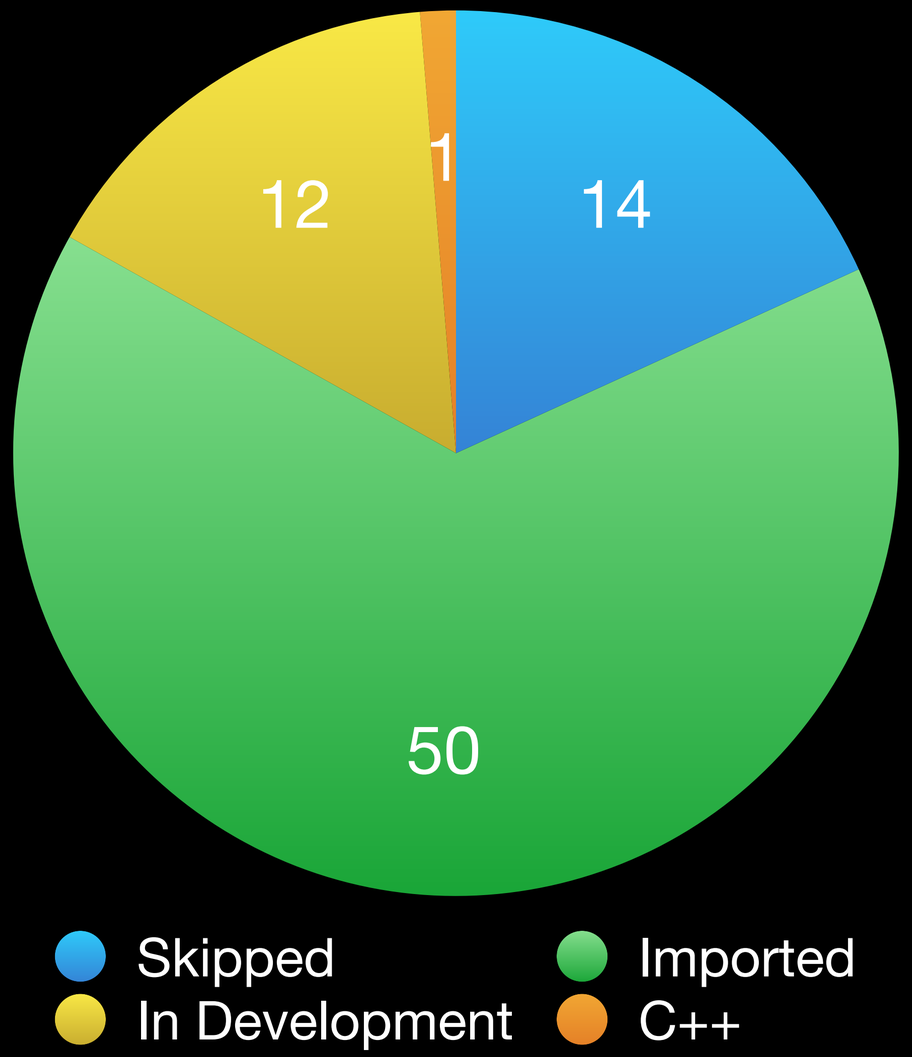
```
%0:anygpr   = G_CONSTANT i64 0xffffffff
%dst:anygpr = G_AND %src:anygpr, %0
%1:gpr      = SLL_ri %src:gpr, i64 32
%dst:gpr    = SRL_ri %1,       i64 32
```

Custom Selection

Dead code is automatically cleared away

```
%0:anygpr   = G_CONSTANT i64 0xffffffff

%1:gpr      = SLL_ri %src:gpr, i64 32
%dst:gpr    = SRL_ri %1,       i64 32
```

What's Left?



Questions?