LLVM IR and Transform Pipeline

David Chisnall
University of Cambridge
LLVM Summer School, Paris, June 12, 2017

What Is LLVM IR?

- Unlimited Single-Assignment Register machine instruction set
- Strongly typed
- Three common representations:
    - Human-readable LLVM assembly (.ll files)
    - Dense ‘bitcode’ binary representation (.bc files)
    - C++ classes

Unlimited Register Machine?

- Real CPUs have a fixed number of registers
- LLVM IR has an infinite number
- New registers are created to hold the result of every instruction
- CodeGen's register allocator determines the mapping from LLVM registers to physical registers
- Type legalisation maps LLVM types to machine types and so on (e.g. 128-element float vector to 32 SSE vectors or 16 AVX vectors, 1-bit integers to 32-bit values)

Static Single Assignment

- Registers may be assigned to only once
- Most (imperative) languages allow variables to be... variable
- This requires some effort to support in LLVM IR: SSA registers are not variables
- SSA form makes dataflow explicit: All consumers of the result of an instruction read the output register(s)

Multiple Assignment

```
int a = someFunction();
a++;
```

- One variable, assigned to twice.

Translating to LLVM IR

```
%a = call i32 @someFunction()
%a = add i32 %a, 1
```

```
error: multiple definition of local value named ’a’
  %a = add i32 %a, 1
    ^
```

Translating to *Correct* LLVM IR

```
%a = call i32 @someFunction()
%a2 = add i32 %a, 1
```

- Front end must keep track of which register holds the current value of `a` at any point in the code
- How do we track the new values?

Translating to LLVM IR The Easy Way

```
; int a
%a = alloca i32, align 4
; a = someFunction
%0 = call i32 @someFunction()
store i32 %0, i32* %a
; a++
%1 = load i32* %a
%2 = add i32 %1, 1
store i32 %2, i32* %a
```

- Numbered register are allocated automatically
- Each expression in the source is translated without worrying about data flow
- Memory is not SSA in LLVM

Isn't That Slow?

- Lots of redundant memory operations
- Stores followed immediately by loads
- The Scalar Replacement of Aggregates (SROA) or mem2reg pass cleans it up for us

```
%0 = call i32 @someFunction()
%1 = add i32 %0, 1
```

Important: SROA only works if the `alloca` is declared in the entry block to the function!

Sequences of Instructions

- A sequence of instructions that execute in order is a *basic block*
- Basic blocks must end with a terminator
- Terminators are *intraprocedural* flow control instructions.
- `call` is not a terminator because execution resumes at the same place after the call
- `invoke` is a terminator because flow either continues or branches to an exception cleanup handler
- This means that even "zero-cost" exceptions can have a cost: they complicate the control-flow graph (CFG) within a function and make optimisation harder.

Intraprocedural Flow Control

- Assembly languages typically manage flow control via jumps / branches (often the same instructions for inter- and intraprocedural flow)
- LLVM IR has conditional and unconditional branches
- Branch instructions are terminators (they go at the end of a basic block)
- Basic blocks are branch targets
- You can't jump into the middle of a basic block (by the definition of a basic block)

What About Conditionals?

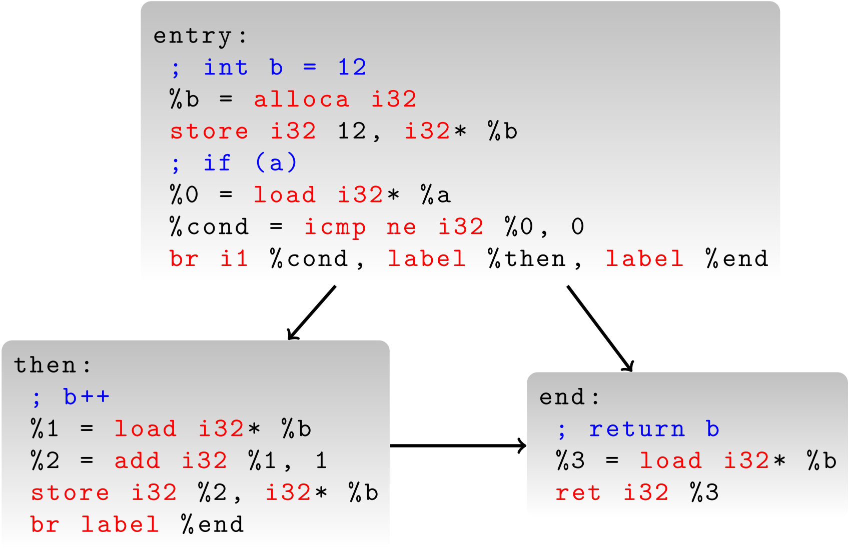
```c
int b = 12;
if (a)
    b++;
return b;
```

- Flow control requires one basic block for each path
- Conditional branches determine which path is taken

‘Phi, my lord, phi!’ - Lady Macbeth, Compiler Developer

- ϕ nodes are special instructions used in SSA construction
- Their value is determined by the preceding basic block
- ϕ nodes must come before any non-ϕ instructions in a basic block
- In code generation, ϕ nodes become a requirement for one basic block to leave a value in a specific register.
- Alternate representation: named parameters to basic blocks (used in Swift IR)

Easy Translation into LLVM IR

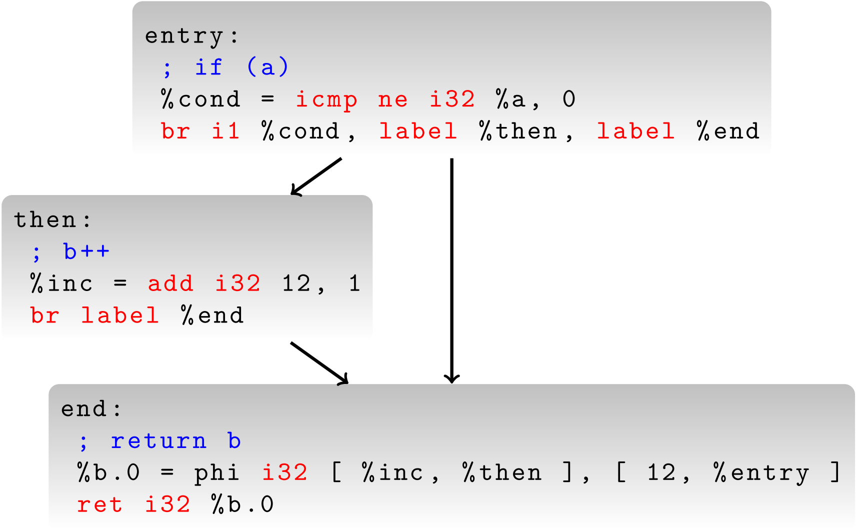


```
entry:
 ; int b = 12
 %b = alloca i32
 store i32 12, i32* %b
 ; if (a)
 %0 = load i32* %a
 %cond = icmp ne i32 %0, 0
 br i1 %cond, label %then, label %end
```

```
then:
 ; b++
 %1 = load i32* %b
 %2 = add i32 %1, 1
 store i32 %2, i32* %b
 br label %end
```

```
end:
 ; return b
 %3 = load i32* %b
 ret i32 %3
```

In SSA Form...

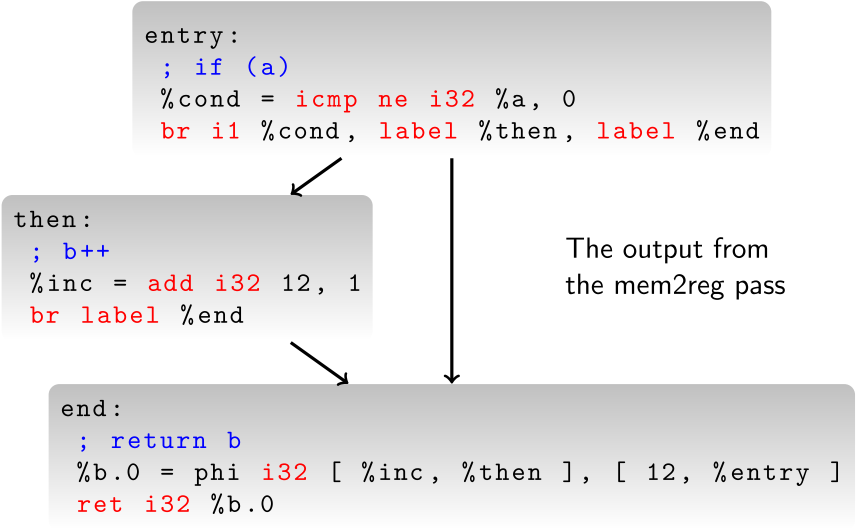


```
entry:
 ; if (a)
 %cond = icmp ne i32 %a, 0
 br i1 %cond, label %then, label %end
```

```
then:
 ; b++
 %inc = add i32 12, 1
 br label %end
```

```
end:
 ; return b
 %b.0 = phi i32 [ %inc, %then ], [ 12, %entry ]
 ret i32 %b.0
```

In SSA Form...



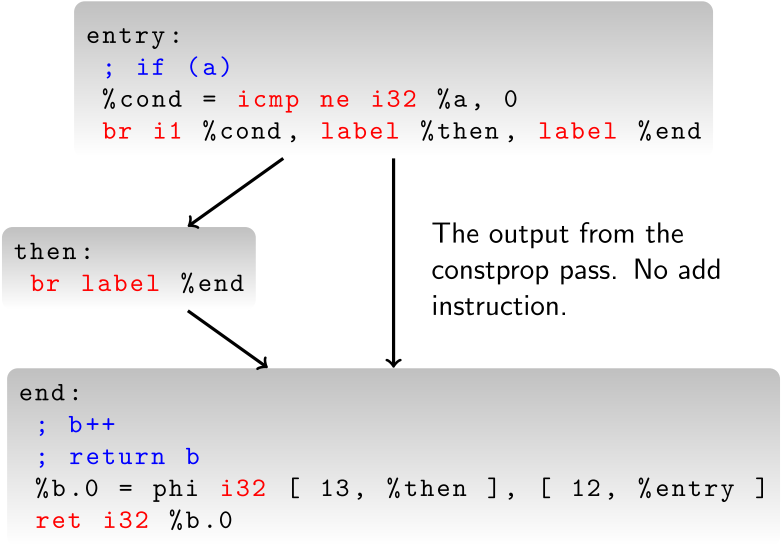
```
entry:
 ; if (a)
 %cond = icmp ne i32 %a, 0
 br i1 %cond, label %then, label %end
```

```
then:
 ; b++
 %inc = add i32 12, 1
 br label %end
```

```
end:
 ; return b
 %b.0 = phi i32 [ %inc, %then ], [ 12, %entry ]
 ret i32 %b.0
```

The output from the mem2reg pass

And After Constant Propagation...



```
entry:
 ; if (a)
 %cond = icmp ne i32 %a, 0
 br i1 %cond, label %then, label %end
```

```
then:
 br label %end
```

```
end:
 ; b++
 ; return b
 %b.0 = phi i32 [ 13, %then ], [ 12, %entry ]
 ret i32 %b.0
```

The output from the constprop pass. No add instruction.

And After CFG Simplification...

```
entry:
  %tobool = icmp ne i32 %a, 0
  %0 = select i1 %tobool, i32 13, i32 12
  ret i32 %0
```

- Output from the simplifycfg pass
- No flow control in the IR, just a `select` instruction

Why Select?

x86:

```
testl %edi, %edi
setne %al
movzbl %al, %eax
orl $12, %eax
ret
```

ARM:

```
mov r1, r0
mov r0, #12
cmp r1, #0
movne r0, #13
mov pc, lr
```

PowerPC:

```
cmplwi 0, 3, 0
beq 0, .LBB0_2
li 3, 13
blr
.LBB0_2:
li 3, 12
blr
```

Branch is only needed on some architectures.

Why Select?

x86:

```
testl %edi, %edi
setne %al
movzbl %al, %eax
orl $12, %eax
ret
```

ARM:

```
mov r1, r0
mov r0, #12
cmp r1, #0
movne r0, #13
mov pc, lr
```

PowerPC:

```
cmplwi 0, 3, 0
beq 0, .LBB0_2
li 3, 13
blr
.LBB0_2:
li 3, 12
blr
```

Branch is only needed on some architectures.
Would a predicated add instruction be better on ARM?

Canonical Form

- LLVM IR has a notion of canonical form
- High-level have a single canonical representation
- For example, loops:
  - Have a single entry block
  - Have a single back branch to the start of the entry block
  - Have induction variables in a specific form
- Some passes generate canonical form from non-canonical versions commonly generated by front ends
- All other passes can expect canonical form as input

Functions

- LLVM functions contain at least one basic block
- Arguments are registers and are explicitly typed
- Registers are valid only within a function scope

```
@hello = private constant [13 x i8] c"Hello
    world!\00"

define i32 @main(i32 %argc, i8** %argv) {
entry:
  %0 = getelementptr [13 x i8]* @hello, i32 0,
     i32 0
  call i32 @puts(i8* %0)
  ret i32 0
}
```

Get Element Pointer?

- Often shortened to GEP (in code as well as documentation)
- Represents pointer arithmetic
- Translated to complex addressing modes for the CPU
- Also useful for alias analysis: result of a GEP is the same object as the original pointer (or undefined)

In modern LLVM IR, on the way to typeless pointers, GEP instructions carry the pointee type. For brevity, we'll use the old form in the slides.

F!@£ing GEPs! HOW DO THEY WORK?!?

```
struct a {
    int c;
    int b[128];
} a;
int get(int i) { return a.b[i]; }
```

F!@£ing GEPs! HOW DO THEY WORK?!?

```c
struct a {
    int c;
    int b[128];
} a;
int get(int i) { return a.b[i]; }
```

```llvm
%struct.a = type { i32, [128 x i32] }
@a = common global %struct.a zeroinitializer,
   align 4

define i32 @get(i32 %i) {
entry:
  %arrayidx = getelementptr inbounds %struct.a*
      @a, i32 0, i32 1, i32 %i
  %0 = load i32* %arrayidx
  ret i32 %0
}
```

As x86 Assembly

```
define i32 @get(i32 %i) {
entry:
  %arrayidx = getelementptr inbounds %struct.a*
      @a, i32 0, i32 1, i32 %i
  %0 = load i32* %arrayidx
  ret i32 %0
}
```

```
get:
    movl    4(%esp), %eax         # load parameter
    movl    a+4(,%eax,4), %eax    # GEP + load
    ret
```

As ARM Assembly

```
define i32 @get(i32 %i) {
entry:
  %arrayidx = getelementptr inbounds %struct.a*
      @a, i32 0, i32 1, i32 %i
  %0 = load i32* %arrayidx
  ret i32 %0
}
```

```
get:
    ldr     r1, .LCPI0_0          // Load global address
    add     r0, r1, r0, lsl #2 // GEP
    ldr     r0, [r0, #4]          // load return value
    bx      lr
.LCPI0_0:
    .long     a
```

The Most Important LLVM Classes

- `Module` - A compilation unit.

The Most Important LLVM Classes

- `Module` - A compilation unit.
- `Function` - Can you guess?

The Most Important LLVM Classes

- `Module` - A compilation unit.
- `Function` - Can you guess?
- `BasicBlock` - a basic block

The Most Important LLVM Classes

- `Module` - A compilation unit.
- `Function` - Can you guess?
- `BasicBlock` - a basic block
- `GlobalVariable` (I hope it's obvious)

The Most Important LLVM Classes

- `Module` - A compilation unit.
- `Function` - Can you guess?
- `BasicBlock` - a basic block
- `GlobalVariable` (I hope it's obvious)
- `IRBuilder` - a helper for creating IR

The Most Important LLVM Classes

- `Module` - A compilation unit.
- `Function` - Can you guess?
- `BasicBlock` - a basic block
- `GlobalVariable` (I hope it's obvious)
- `IRBuilder` - a helper for creating IR
- `Type` - superclass for all LLVM concrete types

The Most Important LLVM Classes

- `Module` - A compilation unit.
- `Function` - Can you guess?
- `BasicBlock` - a basic block
- `GlobalVariable` (I hope it's obvious)
- `IRBuilder` - a helper for creating IR
- `Type` - superclass for all LLVM concrete types
- `ConstantExpr` - superclass for all constant expressions

The Most Important LLVM Classes

- `Module` - A compilation unit.
- `Function` - Can you guess?
- `BasicBlock` - a basic block
- `GlobalVariable` (I hope it's obvious)
- `IRBuilder` - a helper for creating IR
- `Type` - superclass for all LLVM concrete types
- `ConstantExpr` - superclass for all constant expressions
- `PassManagerBuilder` - Constructs optimisation pass sequences to run

The Most Important LLVM Classes

- `Module` - A compilation unit.
- `Function` - Can you guess?
- `BasicBlock` - a basic block
- `GlobalVariable` (I hope it's obvious)
- `IRBuilder` - a helper for creating IR
- `Type` - superclass for all LLVM concrete types
- `ConstantExpr` - superclass for all constant expressions
- `PassManagerBuilder` - Constructs optimisation pass sequences to run
- `ExecutionEngine` - Interface to the JIT compiler

Writing a New Pass

LLVM optimisations are self-contained classes:

- `ModulePass` subclasses modify a whole module
- `FunctionPass` subclasses modify a function
- `LoopPass` subclasses modify a function
- Lots of analysis passes create information your passes can use!

Example Language-specific Passes

ARC Optimisations:

- Part of LLVM
- Elide reference counting operations in Objective-C code when not required
- Makes heavy use of LLVM's flow control analysis

GNUstep Objective-C runtime optimisations:

- Distributed with the runtime.
- Can be used by clang (Objective-C) or LanguageKit (Smalltalk)
- Cache method lookups, turn dynamic into static behaviour if safe

Writing A Simple Pass

- Memoise an expensive library call
- Call maps a string to an integer (e.g. string intern function)
- Mapping can be expensive.
- Always returns the same result.

```
x = example("some␣string");
```

```
static int ._cache;
if (!._cache)
  ._cache = example("some␣string");
x = ._cache;
```

Declaring the Pass

```
struct MemoiseExample : ModulePass, InstVisitor<
    SimplePass>
{
  ... // Boilerplate, see SimplePass
  /// The function that we're going to memoise
  Function *exampleFn;
  /// The return type of the function
  Type *retTy;
  /// Call sites and their constant string
      arguments
  using ExampleCall = std::pair<CallInst&,std::
      string>;
  /// All of the call sites that we've found
  SmallVector<ExampleCall, 16> sites;
```

The Entry Point

```
/// Pass entry point
bool runOnModule(Module &Mod) override {
  sites.clear();
  // Find the example function
  exampleFn = Mod.getFunction("example");
  // If it isn't referenced, exit early
  if (!exampleFn)
    return false;
  // We'll use the return type later for the
      caches
  retTy = exampleFn->getFunctionType()->
      getReturnType();
  // Find all call sites
  visit(Mod);
  // Insert the caches
  return insertCaches(Mod);
}
```

Finding the Call

```
void visitCallInst(CallInst &CI) {
  if (CI.getCalledValue() == exampleFn)
    if (auto *arg = dyn_cast<GlobalVariable>(
         CI.getOperand(0)->stripPointerCasts()))
      if (auto *init = dyn_cast<
         ConstantDataSequential>(
            arg->getInitializer()))
        if (init->isString())
          sites.push_back({CI,
                          init->getAsString()});
}
```

Creating the Cache

- Once we've found all of the replacement points, we can insert the caches.
- Don't do this during the search - iteration doesn't like the collection being mutated...

```
StringMap<GlobalVariable*> statics;
for (auto &s : sites) {
  auto *lookup = &s.first;
  auto arg = s.second;
  GlobalVariable *cache = statics[arg];
  if (!cache) {
    cache = new GlobalVariable(M, retTy, false,
        GlobalVariable::PrivateLinkage ,
           Constant::getNullValue(retTy),
        "._cache");
    statics[arg] = cache;
  }
```

Restructuring the CFG

```
auto *preLookupBB = lookup->getParent();
auto *lookupBB =
   preLookupBB->splitBasicBlock(lookup);
BasicBlock::iterator iter(lookup);
auto *afterLookupBB =
   lookupBB->splitBasicBlock(++iter);
preLookupBB->getTerminator()->eraseFromParent();
lookupBB->getTerminator()->eraseFromParent();
auto *phi = PHINode::Create(retTy, 2, "cache",
   &*afterLookupBB->begin());
lookup->replaceAllUsesWith(phi);
```

Adding the Test

```
IRBuilder <> B(beforeLookupBB);
llvm::Value *cachedClass =
  B.CreateBitCast(B.CreateLoad(cache), retTy);
llvm::Value *needsLookup =
  B.CreateIsNull(cachedClass);
B.CreateCondBr(needsLookup , lookupBB ,
   afterLookupBB);
B.SetInsertPoint(lookupBB);
B.CreateStore(lookup , cache);
B.CreateBr(afterLookupBB);
phi->addIncoming(cachedClass, beforeLookupBB);
phi->addIncoming(lookup, lookupBB);
```

A Simple Test

```c
int example(char *foo) {
  printf("example(%s)\n", foo);
  int i=0;
  while (*foo)
    i += *(foo++);
  return i;
}
int main(void) {
  int a = example("a␣contrived␣example");
  a += example("a␣contrived␣example");
  a += example("a␣contrived␣example");
  a += example("a␣contrived␣example");
  a += example("a␣contrived␣example");
  return a;
}
```

Running the Test

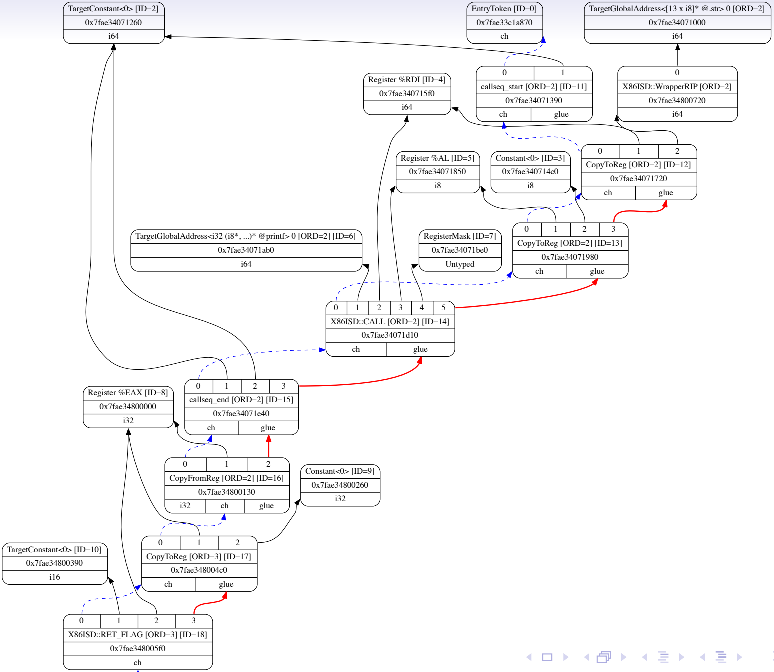
```
$ clang example.c -O2 ; ./a.out ; echo $?
example(a contrived example)
example(a contrived example)
example(a contrived example)
example(a contrived example)
example(a contrived example)
199
$ clang -Xclang -load -Xclang ./memo.so -O2
$ ./a.out ; echo $?
example(a contrived example)
199
```

How Does LLVM IR Become Native Code?

- Transformed to directed acyclic graph representation (SelectionDAG)
- Mapped to instructions (Machine IR)
- Streamed to assembly or object code writer

Selection DAG

- DAG defining operations and dependencies
- Legalisation phase lowers IR types to target types
  - Arbitrary-sized vectors to fixed-size
  - Float to integer and softfloat library calls
  - And so on
- DAG-to-DAG transforms simplify structure
- Code is still (more or less) architecture independent at this point
- Some peephole optimisations happen here


TargetConstant<0> [ID=2]
0x7fae34071260
i64
EntryToken [ID=0]
0x7fae33c1a870
ch
TargetGlobalAddress<[13 x i8]* @.str> 0 [ORD=2]
0x7fae34071000
i64
Register %RDI [ID=4]
0x7fae340715f0
i64
0
1
callseq_start [ORD=2] [ID=11]
0x7fae34071390
ch
glue
0
X86ISD::WrapperRIP [ORD=2]
0x7fae34800720
i64
Register %AL [ID=5]
0x7fae34071850
i8
Constant<0> [ID=3]
0x7fae340714c0
i8
0
1
2
CopyToReg [ORD=2] [ID=12]
0x7fae34071720
ch
glue
TargetGlobalAddress<i32 (i8*, ...)* @printf> 0 [ORD=2] [ID=6]
0x7fae34071ab0
i64
RegisterMask [ID=7]
0x7fae34071be0
Untyped
0
1
2
3
CopyToReg [ORD=2] [ID=13]
0x7fae34071980
ch
glue
0
1
2
3
4
5
X86ISD::CALL [ORD=2] [ID=14]
0x7fae34071d10
ch
glue
Register %EAX [ID=8]
0x7fae34800000
i32
0
1
2
3
callseq_end [ORD=2] [ID=15]
0x7fae34071e40
ch
glue
0
1
2
CopyFromReg [ORD=2] [ID=16]
0x7fae34800130
i32
ch
glue
Constant<0> [ID=9]
0x7fae34800260
i32
TargetConstant<0> [ID=10]
0x7fae34800390
i16
0
1
2
CopyToReg [ORD=3] [ID=17]
0x7fae348004c0
ch
glue
0
1
2
3
X86ISD::RET_FLAG [ORD=3] [ID=18]
0x7fae348005f0
ch

Instruction Selection

- Pattern matching engine maps subtrees to instructions and pseudo-ops
- Generates another SSA form: Machine IR (MIR)
- Real machine instructions
- Some (target-specific) pseudo instructions
- Mix of virtual and physical registers
- Low-level optimisations can happen here

Register allocation

- Maps virtual registers to physical registers
- Adds stack spills / reloads as required
- Can reorder instructions, with some constraints

MC Streamer

- Class with assembler-like interface
- Emits one of:
  - Textual assembly
  - Object code file (ELF, Mach-O, COFF)
  - In-memory instruction stream
- All generated from the same instruction definitions

Questions?