

# LLVM IR and Transform Pipeline

L25: Modern Compiler Design

# 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, 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

# 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 %0, 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

# 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?

```
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

- $\phi$  nodes are special instructions used in SSA construction
- Their value is determined by the preceding basic block
- $\phi$  nodes must come before any non- $\phi$  instructions in a basic block

## 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*  
%2 = add i32 %  
store i32 %2,  
br label %end
```

```
end:  
; ret  
%3 =  
ret
```

## 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
```

The output from the  
constprop pass. No add  
instruction.

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

## 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?

# 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)

## F!@£ing GEPs! HOW DO THEY WORK?!?

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

```
%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:

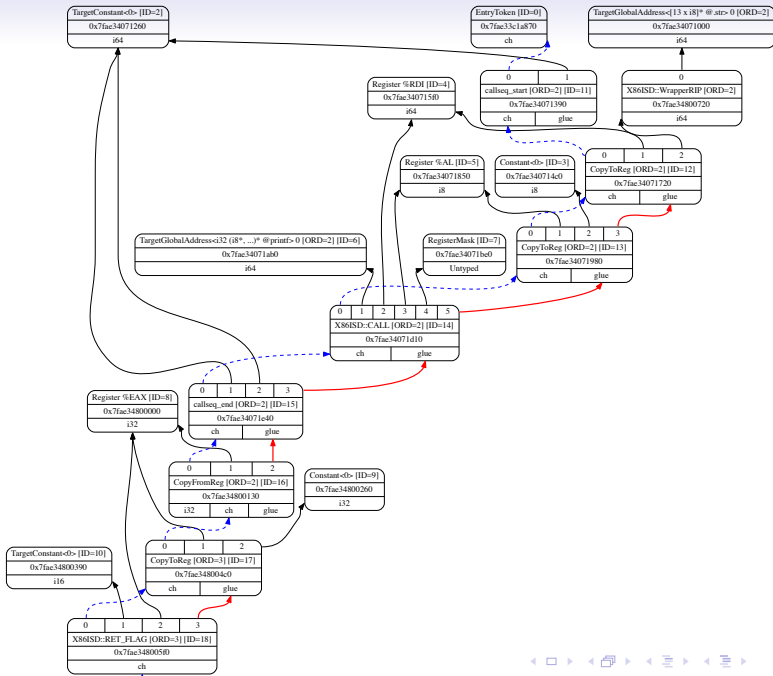
```
.long  a
```

# How Does This Become Native Code?

- Transformed to directed acyclic graph representation
- Mapped to instructions
- 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
- Code is still (more or less) architecture independent at this point
- Some peephole optimisations happen here



# Instruction Selection

- Pattern matching engine maps subtrees to instructions and pseudo-ops
- Another SSA form
- 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

# 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 passes 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 passes to run
- `ExecutionEngine` - The thing that drives the JIT



# 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.

## Declaring the Pass

```
class MemoiseExample : public ModulePass {
    /// Module that we're currently optimising
    Module *M;
    /// Static cache.
    llvm::StringMap<GlobalVariable*> statics;
    // Lookup - call plus its argument
    typedef std::pair<CallInst*,std::string>
        ExampleCall;
    bool runOnFunction(Function &F);
public:
    static char ID;
    MemoiseExample() : ModulePass(ID) {}
    virtual bool runOnModule(Module &Mod);
};
RegisterPass<MemoiseExample> X("example-pass",
    "Memoise_ example_pass");
```

## The Entry Point

```
bool MemoiseExample::runOnModule(Module &Mod) {
    statics.empty();
    M = &Mod;
    bool modified = false;

    for (auto &F : Mod) {

        if (F.isDeclaration()) { continue; }

        modified |= runOnFunction(F);
    }

    return modified;
};
```

## Finding the Call

```
for (auto &i : F) {
  for (auto &b : i) {
    if (CallInst *c = dyn_cast<CallInst>(&b)) {
      if (Function *func = c->getCalledFunction()){
        if (func->getName() == "example") {
          ExampleCall lookup;
          GlobalVariable *arg =
            dyn_cast<GlobalVariable>(
              c->getOperand(0)->stripPointerCasts());
          if (0 == arg) { continue; }
          ConstantDataArray *init =
            dyn_cast<ConstantDataArray>(
              arg->getInitializer());
```

## 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...

```
GlobalVariable *cache = statics[arg];
if (!cache) {
    cache = new GlobalVariable(*M, retTy, false,
        GlobalVariable::PrivateLinkage,
        Constant::getNullValue(retTy),
        "_cache");
    statics[arg] = cache;
}
```

## Restructuring the CFG

```
BasicBlock *beforeLookupBB=lookup->getParent();
BasicBlock *lookupBB =
    SplitBlock(beforeLookupBB, lookup, this);
BasicBlock::iterator iter = lookup;
iter++;
BasicBlock *afterLookupBB =
    SplitBlock(iter->getParent(), iter, this);
removeTerminator(beforeLookupBB);
removeTerminator(lookupBB);
PHINode *phi = PHINode::Create(retTy, 2, arg,
    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

```
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 ; ./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 'llvm-config --cxxflags --ldflags ' memo.cc \  
-std=c++11 -fPIC -shared -o memo.so
```

```
$ clang example.c -c -emit-llvm
```

```
$ opt -load ./memo.so -example-pass example.o | llc > e.s
```

```
$ clang e.s ; ./a.out ; echo $?
```

```
example(a contrived example)
```

```
199
```

# 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