

Low Level Virtual Machine (LLVM)

Chris Lattner*, Vikram Adve et al
(UIUC, *Apple)

Prashanth Radhakrishnan
CS7968 - Program Analysis
February 1, 2007

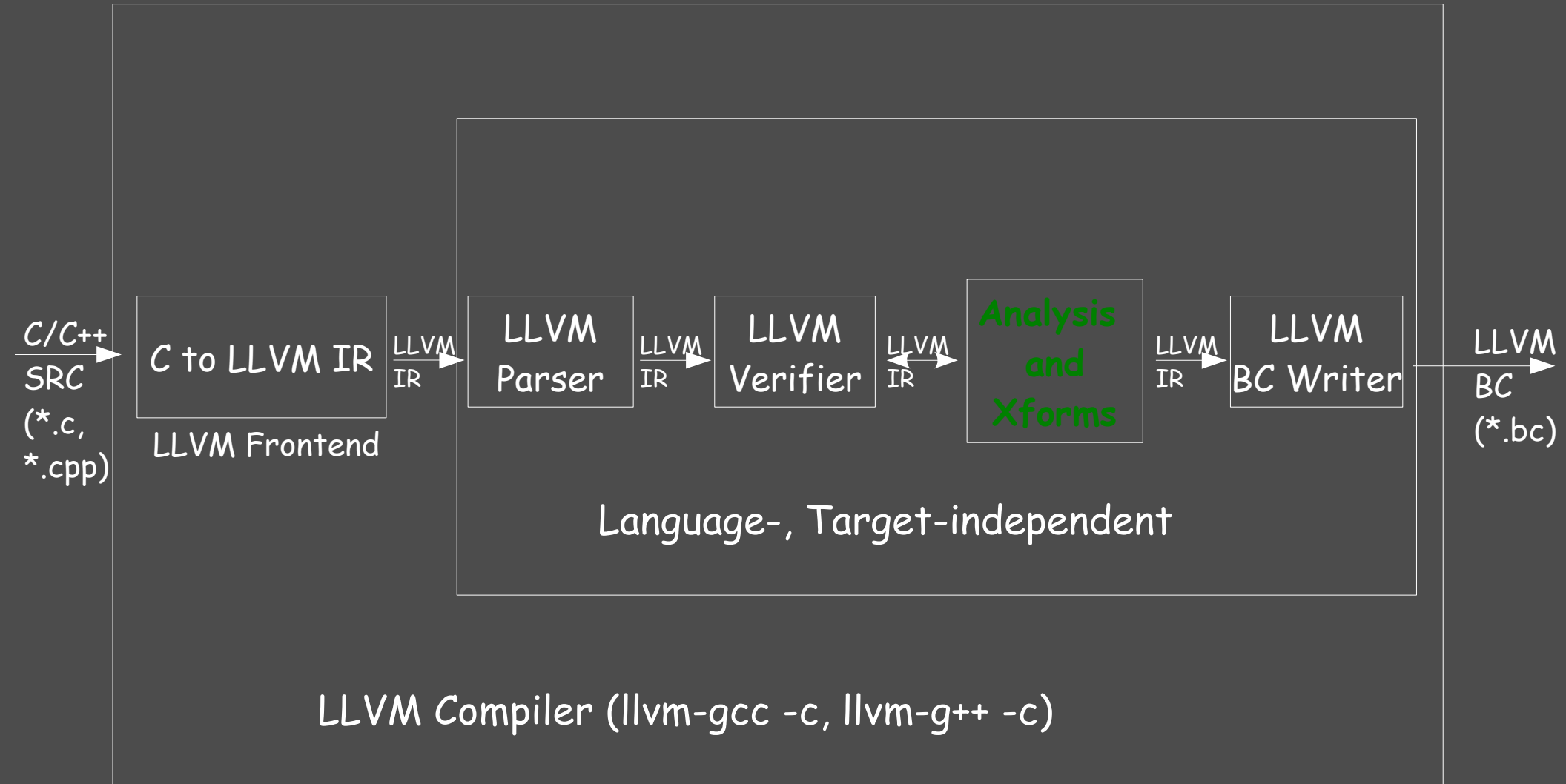
What's LLVM? ...

- Compiler Infrastructure
 - Language-, target-independent reusable components for analysis & xforms
 - Language-, target-dependent front-end & back-end
 - “Life-long” Program analysis and Xform
 - compile-time, link-time, (install-time, run-time, idle-time)

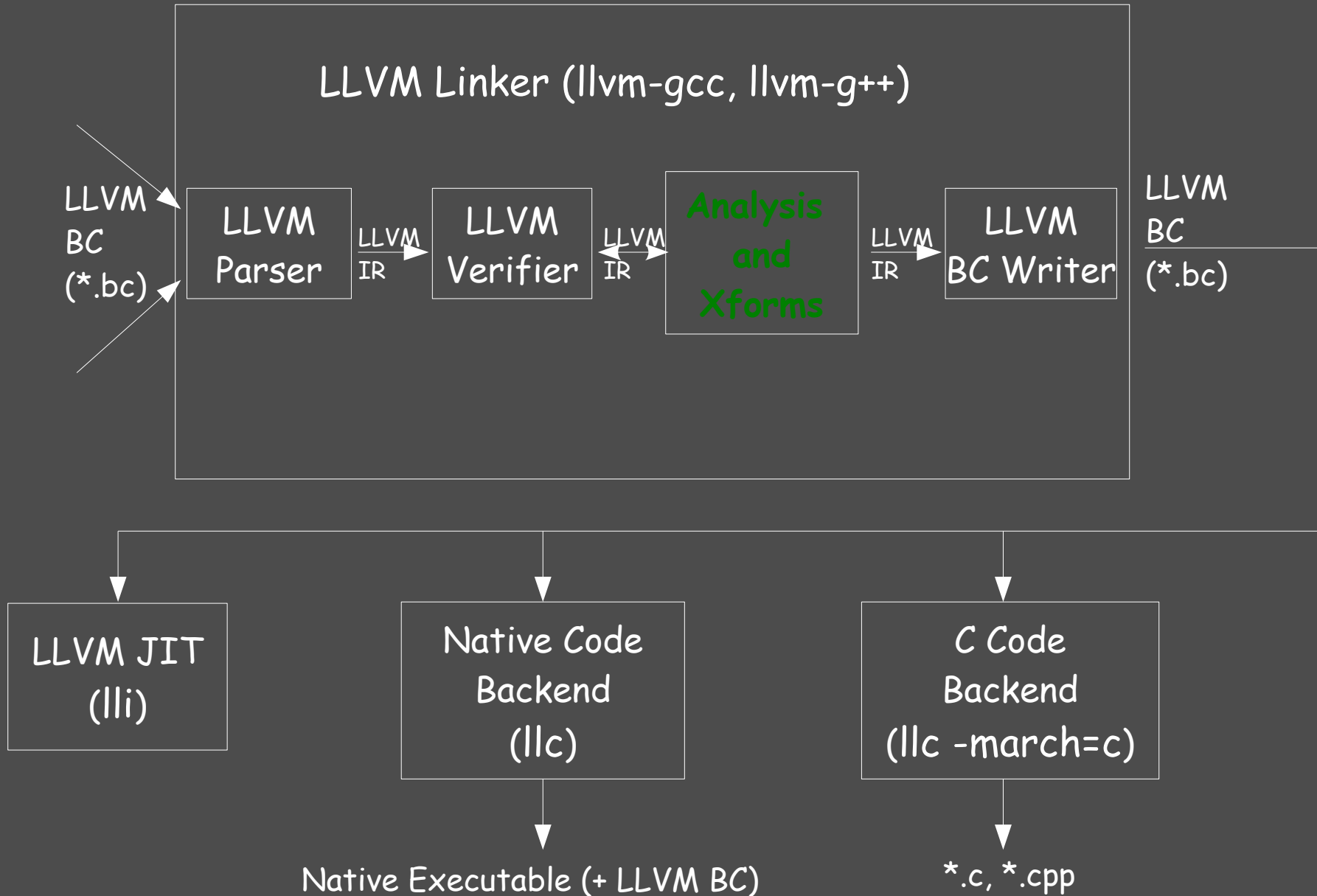
... What's LLVM?

- LLVM Intermediate Representation is the glue
 - Language-, target-independent
 - Type Info, SSA
 - Persistent
 - Support different optimizations

LLVM - Compile-time ...



LLVM - Link-time ...



LLVM - Post-Install (defunct?)

- Runtime path profiling & Re-optimization
 - Codegen inserts light-weight instrumentation
 - LLVM BC in executable
 - Runtime library hacks
 - Re-optimizes "hot" paths using LLVM IR
 - Does online codegen
- Offline Re-optimization
 - More intrusive optimizations at idle-time

Play around with Opts.

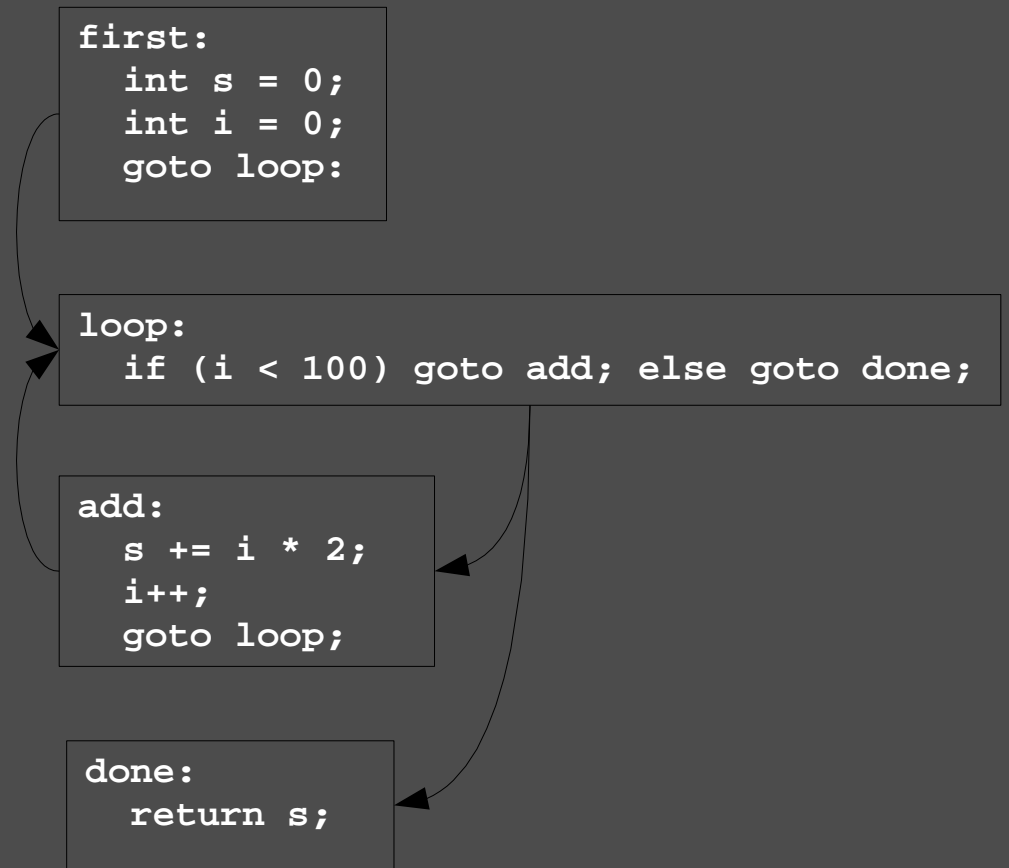
- Several Analysis & Xforms are run by default (45 @ compile-time, 23 @ link-time)
- To selectively run opts
 - Run only the front-end to produce LLVM BC
 - "llvm-gcc -S" or "llvm-g++ -S"
 - Use the "opt" tool to run your opts
 - "opt -help" lists all opts available with LLVM
 - "opt -load" can be used to dynamically load new opt libraries

Static Single Assignment

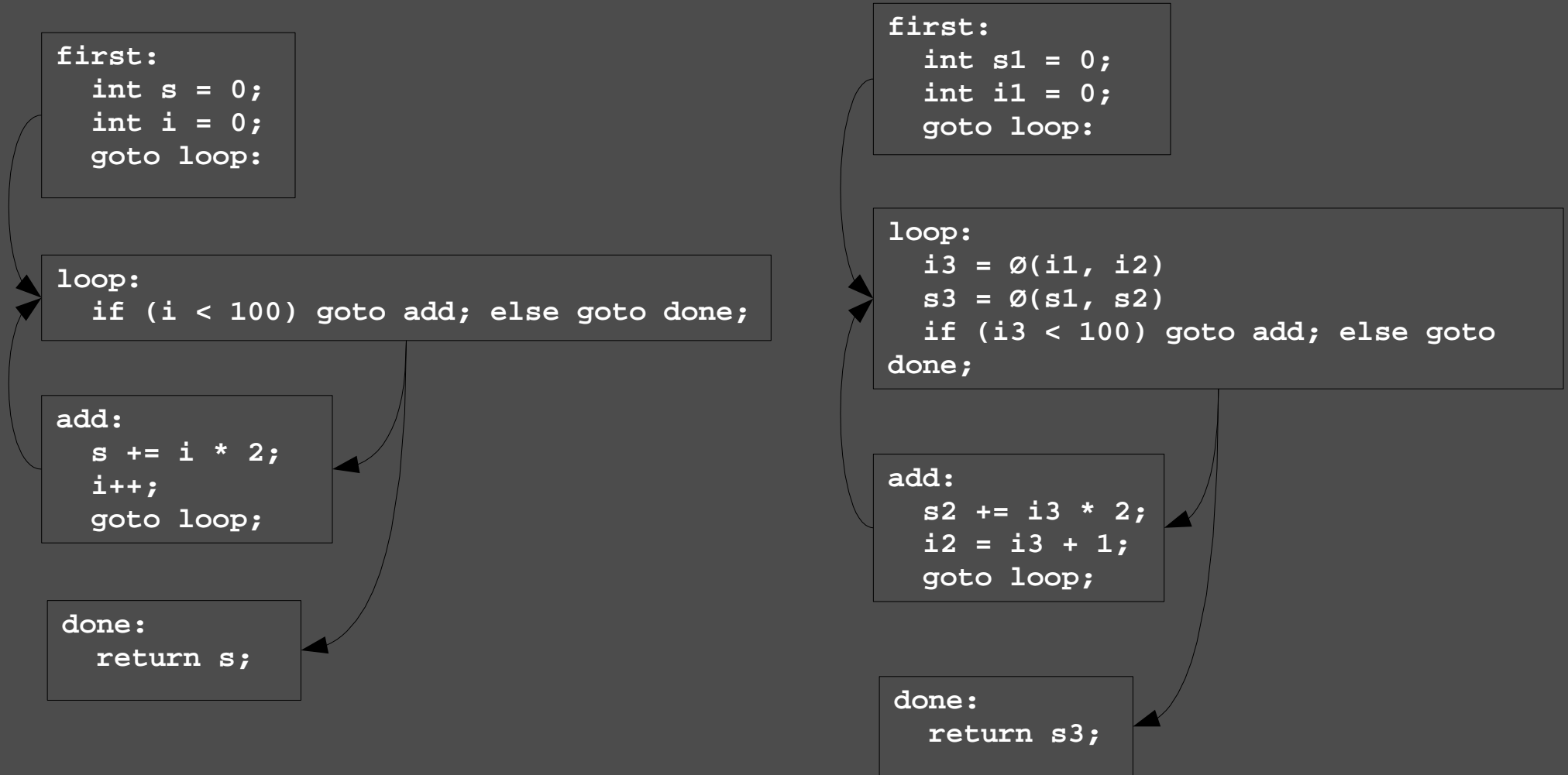
- A program is in SSA form if each variable is a target of exactly one assignment stmt.
- Conversion to SSA (from a CFG) involves:
 - Adding ϕ -functions $V = \phi(R, S, \dots)$ at "some" join nodes, (R, S, \dots are the control flow predecessors of the join node)
 - Rename each variable to satisfy single-assignment
- Variable Reaching defn. trivial (sans ptrs)

e.g. Src -> CFG

```
int f()
{
  int i;
  int s = 0;
  for (i = 0; i < 100; i++) {
    s += i * 2;
  }
  return s;
}
```



e.g. CFG → SSA



LLVM IR - Overview ...

- RISC-like 3 instruction code
- Scalar values in infinite register set SSA
 - Memory NOT in SSA
- Explicit type information
- Explicit CFG info
- Explicit \emptyset -function
 - PHI type [value1, label1] [value2, label2] . . .

... LLVM IR - Overview

- LLVM IR's isomorphic formats
 - In memory IR
 - On-disk bytecode
 - On-disk text
- Claimed to have a small memory footprint
 - "For 200K LOC, where GCC takes multi GB, LLVM uses ~50MB"

LLVM - Program Structure

- Module => Functions/GlobalVariables
 - Module - unit of compilation/analysis
- Functions => BasicBlocks/Args
- BasicBlock => Instrs
 - Identified by labels
 - Ends in a terminator instr (br, ret, unwind, invoke, switch) <= call?
 - Terminator instr transfers control to another BasicBlock i.e. label

LLVM IR - Type-info ...

- Every SSA reg & memory obj. explicitly typed
- Type + Opcode => Instruction Semantics (eg. FP add, Int add)

... LLVM IR - Type-info

- Primitives:
 - void, int(1,8-64), FP, label (BasicBlock)
- Derived:
 - pointer, array, structure, function
- Allows arbitrary casts
 - To express non-type-safe languages, like c

... LLVM IR - Type-info

- Typed pointer arithmetic
 - Machine-independent
 - Preserves array subscript, struct index
 - Key for field-sensitive analysis

... LLVM IR - Type-info

```
fieldtype* getelementptr(aggrrtype* aggr, int index, int field#)
```

```
struct a {  
    int b;  
    float d;  
} A, B[2];
```

```
A.d      -> float* getelementptr (%struct.a* %A, int 0, uint 1)
```

```
B[1].b   -> int* getelementptr ([2 x %struct.a]* %B, int 0, int 1, uint 0)
```

LLVM IR - Memory Model

- Addressable objects are explicitly alloc'd
 - Heap -> malloc
 - Stack -> alloca
- Global variables & Func defn => symbols provides addr. of object
- Stores don't respect SSA
 - This simplifies FE

LLVM IR - Memory vs. Reg.

- FE produces simple non-optimized IR
- Locals are all "alloca"d
- Non-SSA writes to such locals are OK
- Promoted to register by the "mem2reg", "scallarrepl" passes
- Cannot be promoted to reg if its address is used

e.g. LLVM IR from FE

```
entry: ;i = 0; s = 0;

      %result = alloca int
      %i = alloca int
      %s = alloca int
      store int 0, int* %s
      store int 0, int* %i
      br label %loopentry
```

```
int f()
{
    int i;
    int s = 0;
    for (i = 0; i < 100; i++) {
        s += i * 2;
    }
    return s;
}
```

```
loopentry: ;if (i <= 99) {}

      %tmp.0 = load int* %i
      %tmp.1 = setle int %tmp.0, 99
      %tmp.2 = cast bool %tmp.1 to int
      br bool %tmp.1, label %no_exit, label %loopexit
```

```
no_exit: ;s += i * 2; i++

      %tmp.3 = load int* %i
      %tmp.4 = mul int %tmp.3, 2
      %tmp.5 = load int* %s
      %tmp.6 = add int %tmp.4, %tmp.5
      store int %tmp.6, int* %s
      %tmp.7 = load int* %i
      %inc = add int %tmp.7, 1
      store int %inc, int* %i
      br label %loopentry
```

```
loopexit:

      %tmp.8 = load int* %s
      store int %tmp.8, int* %result
      br label %return
```

```
return:

      %tmp.9 = load int* %result
      ret int %tmp.9
```

e.g. LLVM IR post opt.

```
entry:  
  br label %no_exit
```

```
no_exit:      ; preds = %no_exit, %entry  
  %indvar = phi uint [ 0, %entry ], [ %indvar.next, %no_exit ]  
  %s.0.0 = phi int [ 0, %entry ], [ %tmp.6, %no_exit ]  
  %i.0.0 = cast uint %indvar to int  
  %tmp.4 = shl int %i.0.0, ubyte 1  
  %tmp.6 = add int %tmp.4, %s.0.0  
  %indvar.next = add uint %indvar, 1  
  %exitcond = seteq uint %indvar.next, 100  
  br bool %exitcond, label %loopexit, label %no_exit
```

```
loopexit:   ; preds = %no_exit  
  ret int %tmp.6
```

- No "alloca"s
- \emptyset s for index & sum variables
- Strength Reduction

Exception Support ...

- Lang.-independent representation of exceptions
- Two instructions:
 - Invoke - call to a function needing an exception handler
 - Unwind - unwind stack frames until an invoke is reached
- Optimizers benefit from knowing the exception control-flow

... Exception Support

- Language-specific details isolated in runtime libraries (for e.g. verifying exception types)
- Front-end generates these libcalls (for e.g. before a 'throw', at the 'catch')
- Support for C setjmp/longjmp & C++ exception handling

Lowering C++ to LLVM IR

- References -> Pointers
- Base classes -> nested struct types
- Implicit calls made explicit (ctors)
- Virtual Functions -> vtable pointer included in struct

... Lowering C++ to LLVM IR

- Virtual Function Table -> global, constant array of typed function ptr
- try/catch -> invoke/unwind + libcalls
 - func-calls within try become invoke
 - within try block, unwind -> br

e.g. C++ -> LLVM IR

```
class base {
    float X;
public:
    int Y;
    int virtual v() { return 1; }
};

class inherit : public base {
public:
    int Z;
    int v() { return 0; }
};

int main()
{
    class inherit A;
    return A.v();
}
```

Output from Front-End

```
%struct.base = type { int (...)**, float, int }

%struct.inherit =
    type [{ int (...)**, float, int }, int ]

int %main() {
entry:
    %result = alloca int
    %A = alloca %struct.inherit
    call void ()* %__main()
    call void (%struct.inherit)*
        %_ZN7inheritC1Ev(%struct.inherit* %A)
    %tmp.0 = call int (%struct.inherit)*
        %_ZN7inherit1vEv(%struct.inherit* %A)
    store int %tmp.0, int* %result
    br label %return

    ....
}
```

LLVM Pass Infrastructure ...

- Provides access to LLVM program to passes in various ways
 - ModulePass
 - CallGraphSCCPass
 - FunctionPass
 - BasicBlockPass

... LLVM Pass Infrastructure

- A pass can specify its interactions
 - What it requires
 - What it preserves
- Pass Management
 - Track Lifetime of analysis results (e.g. based on what a pass requires or invalidates)
 - Pipeline execution of passes (e.g. run the same func for all compatible FunctionPass)

LLVM - Dataflow Analysis

- No traditional data-flow analysis framework implemented
- Apparently, SSA representation simplifies most analysis & xforms
 - def/use chains readily available
- Illustrated by the constant propagation example

e.g. Constant Propagation

- Intraprocedural
- Simple since, SSA already captures data-flow
- Iterate over the worklist to find constant assignments
 - Once found
 - add the uses to worklist (easy, given SSA)
 - propagate the constant to them
 - remove the identified constant assignment

e.g. Constant Propagation

```
namespace {
    Statistic<> NumInstKilled("constprop",
                             "Number of instructions killed");

    struct ConstantPropagation : public FunctionPass {
        bool runOnFunction(Function &F);

        virtual void getAnalysisUsage(AnalysisUsage &AU) const {
            AU.setPreservesCFG();
        }
    };

    RegisterPass<ConstantPropagation> X("constprop",
                                       "Simple constant propagation");
}

FunctionPass *llvm::createConstantPropagationPass() {
    return new ConstantPropagation();
}

bool ConstantPropagation::runOnFunction(Function &F) {
    // Initialize the worklist to all of the instructions ready to process...
    std::set<Instruction*> WorkList;

    for(inst_iterator i = inst_begin(F), e = inst_end(F); i != e; ++i) {
        WorkList.insert(&*i);
    }
    bool Changed = false;
```

e.g. Constant Propagation

```
namespace {
    Statistic<> NumInstKilled("constprop",
                             "Number of instructions killed");

    struct ConstantPropagation : public FunctionPass {
        bool runOnFunction(Function &F);

        virtual void getAnalysisUsage(AnalysisUsage &AU) const {
            AU.setPreservesCFG();
        }
    };

    RegisterPass<ConstantPropagation> X("constprop",
                                       "Simple constant propagation");
}

FunctionPass *llvm::createConstantPropagationPass() {
    return new ConstantPropagation();
}

bool ConstantPropagation::runOnFunction(Function &F) {
    // Initialize the worklist to all of the instructions ready to process...
    std::set<Instruction*> WorkList;

    for(inst_iterator i = inst_begin(F), e = inst_end(F); i != e; ++i) {
        WorkList.insert(&*i);
    }
    bool Changed = false;
```

e.g. Constant Propagation

```
namespace {
    Statistic<> NumInstKilled("constprop",
                             "Number of instructions killed");

    struct ConstantPropagation : public FunctionPass {
        bool runOnFunction(Function &F);

        virtual void getAnalysisUsage(AnalysisUsage &AU) const {
            AU.setPreservesCFG();
        }
    };

    RegisterPass<ConstantPropagation> X("constprop",
                                       "Simple constant propagation");
}

FunctionPass *llvm::createConstantPropagationPass() {
    return new ConstantPropagation();
}

bool ConstantPropagation::runOnFunction(Function &F) {
    // Initialize the worklist to all of the instructions ready to process...
    std::set<Instruction*> WorkList;

    for(inst_iterator i = inst_begin(F), e = inst_end(F); i != e; ++i) {
        WorkList.insert(&*i);
    }
    bool Changed = false;
```

e.g. Constant Propagation

```
namespace {
  Statistic<> NumInstKilled("constprop",
                           "Number of instructions killed");

  struct ConstantPropagation : public FunctionPass {
    bool runOnFunction(Function &F);

    virtual void getAnalysisUsage(AnalysisUsage &AU) const {
      AU.setPreservesCFG();
    }
  };

  RegisterPass<ConstantPropagation> X("constprop",
                                     "Simple constant propagation");
}

FunctionPass *llvm::createConstantPropagationPass() {
  return new ConstantPropagation();
}

bool ConstantPropagation::runOnFunction(Function &F) {
  // Initialize the worklist to all of the instructions ready to process...
  std::set<Instruction*> WorkList;

  for(inst_iterator i = inst_begin(F), e = inst_end(F); i != e; ++i) {
    WorkList.insert(&*i);
  }
  bool Changed = false;
```

e.g. Constant Propagation

```
while (!WorkList.empty()) {
    Instruction *I = *WorkList.begin();

    if (!I->use_empty() // Don't muck with dead instructions...
        if (Constant *C = ConstantFoldInstruction(I)) {
            // Add all of the users of this instruction to the worklist, they might
            // be constant propagatable now...
            for (Value::use_iterator UI = I->use_begin(), UE = I->use_end();
                UI != UE; ++UI)
                WorkList.insert(cast<Instruction>(*UI));

            // Replace all of the uses of a variable with uses of the constant.
            I->replaceAllUsesWith(C);

            // Remove the dead instruction.
            WorkList.erase(I);
            I->getParent()->getInstList().erase(I);

            // We made a change to the function...
            Changed = true;
            ++NumInstKilled;
        }
    }
    return Changed;
}
```

e.g. Constant Propagation

```
while (!WorkList.empty()) {
    Instruction *I = *WorkList.begin();

    if (!I->use_empty() // Don't muck with dead instructions...
        if (Constant *C = ConstantFoldInstruction(I)) {
            // Add all of the users of this instruction to the worklist, they might
            // be constant propagatable now...
            for (Value::use_iterator UI = I->use_begin(), UE = I->use_end();
                UI != UE; ++UI)
                WorkList.insert(cast<Instruction>(*UI));

            // Replace all of the uses of a variable with uses of the constant.
            I->replaceAllUsesWith(C);

            // Remove the dead instruction.
            WorkList.erase(I);
            I->getParent()->getInstList().erase(I);

            // We made a change to the function...
            Changed = true;
            ++NumInstKilled;
        }
    }
    return Changed;
}
```

In Conclusion..

- A tool for compiler development/research
- Well Documented
- Small, but active Dev Community
- Lead Developer took LLVM with him to Apple
 - LLVM optimizer & JIT used in Apple's OpenGL Stack
- Efforts on to integrate with GCC4 (for link-time opt benefits)