Introduction to LLVM

Sthiti Deka

Jayashree Venkatesh

Agenda

Time	Elapsed Time	Topic	Description
5 min.	5 min.	Welcome, Course Agenda and Introductions	Welcome participants Cover Goals and Objectives Review Agenda, Logistics & Take-away
10 min.	15 min.	What is LLVM & Why LLVM	Define the scope and motivation behind LLVM
45 min.	60 min.	LLVM Compiler Pipeline Overview	Provide a design overview of the LLVM Compiler Pipeline stages – the frontend, the mid-end and the back-end
15 min.	75 min.	Hands-On Ex1: Hello World in Clang	Learn how to compile, diagnose errors with clang, lli, llc and the command line options
10 min.	85 min.	Hands-On Ex2: LLVM C API to generate an LLVM IR Module	Learn how to use the LLVM API and to generate a module in LLVM IR
5 min.	90 min.	Hands-On Ex3: Print & View Program State & Flow	Learn how to use LLVM tools to view & print program flow and program states
10 min.	100 min.	Hands-On Ex4: Analysis Pass in LLVM API	Learn how to write an analysis pass in LLVM API
5min.	105 min.	Hands-On Ex5: Transformation Pass in LLVM API	Learn how to write a transformation pass in LLVM API
5 min.	110 min.	Hands-On Ex6: Debug tools in LLVM IR	Learn how to use print before and after transformation to debug
10 min	120 min.	Q/A and wrap up	Final questions and conclusions

Content

- What is LLVM?
- Why LLVM?
- LLVM Compiler Pipeline
  - LLVM Front-end
  - LLVM Mid-end
  - LLVM Back-end
- Hands-On

What is LLVM?

What is LLVM

- Open Source Project

What is LLVM

- Open Source Project
  - Compiler Framework

What is LLVM

- Open Source Project
  - Compiler Framework
  - Infrastructure

What is LLVM

- Open Source Project
  - Compiler Framework
    - Optimizations
  - Infrastructure

What is LLVM

- Open Source Project
  - Compiler Framework
    - Optimizations
    - API + IR
  - Infrastructure

What is LLVM

- Open Source Project
  - Compiler Framework
    - Optimizations
    - API + IR
    - Retargetable: m languages x n targets
  - Infrastructure

What is LLVM

- Open Source Project
  - Compiler Framework
    - Optimizations
    - API + IR
    - Retargetable: m languages x n targets
  - Infrastructure
    - Reusable libraries
    - Tool Chain

Why LLVM?

Why LLVM

Why LLVM

- Reduced development time

Why LLVM

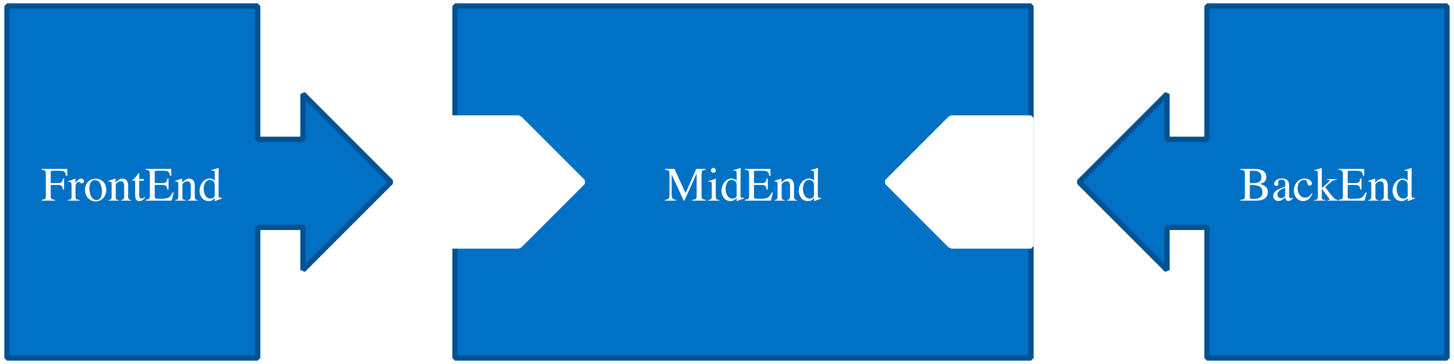
- Reduced development time
- Performance
    - Speed
    - Low Memory Footprint

Why LLVM

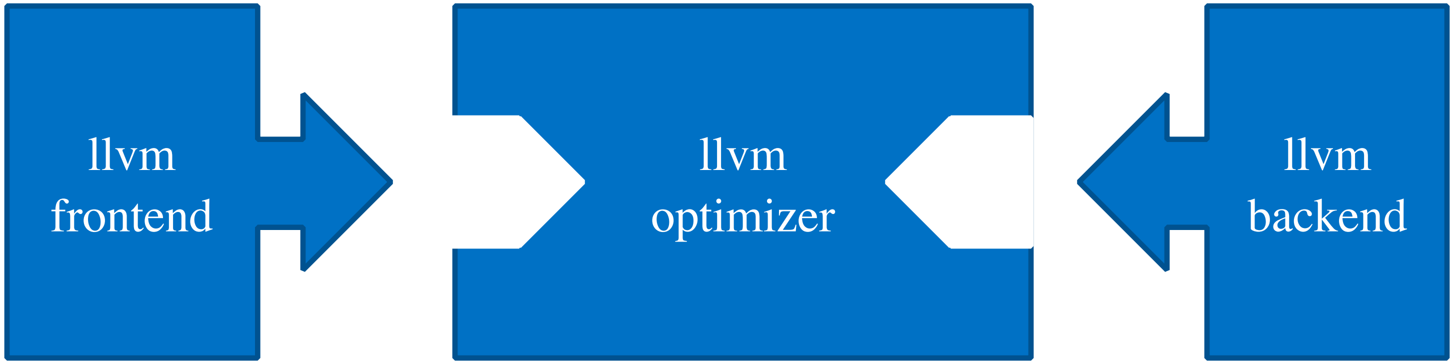
- Reduced development time
- Performance
  - Speed
  - Low Memory Footprint
- Ease of use
  - Toolchain, Libraries

LLVM Compiler Framework

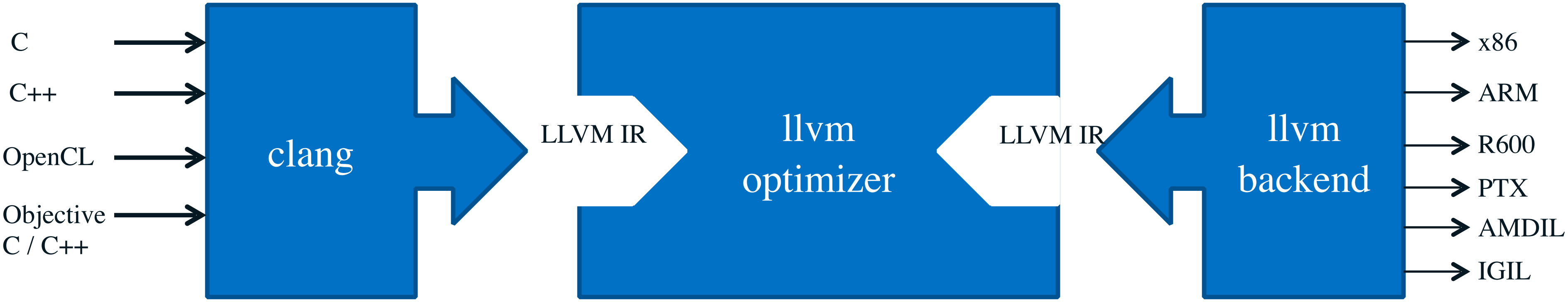
LLVM Compiler Pipeline



LLVM Compiler Pipeline

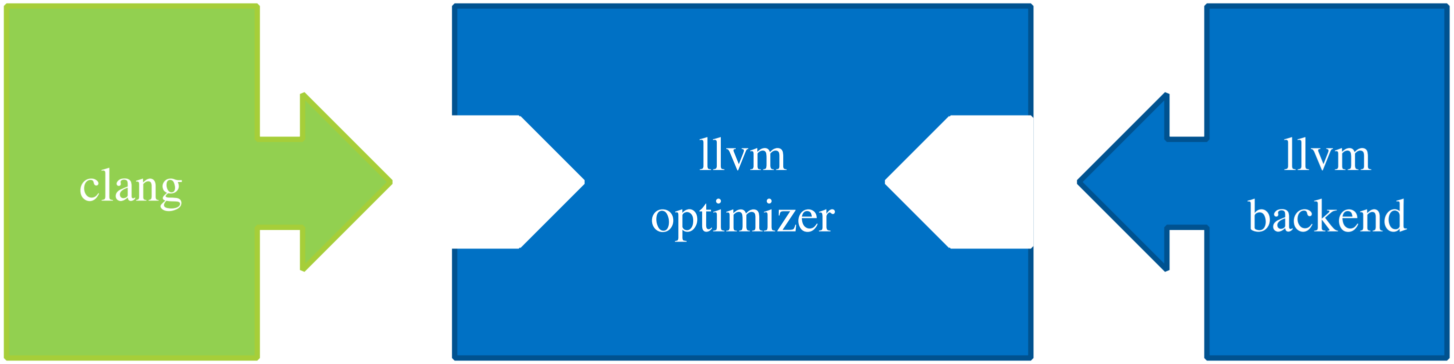


LLVM Compiler Pipeline

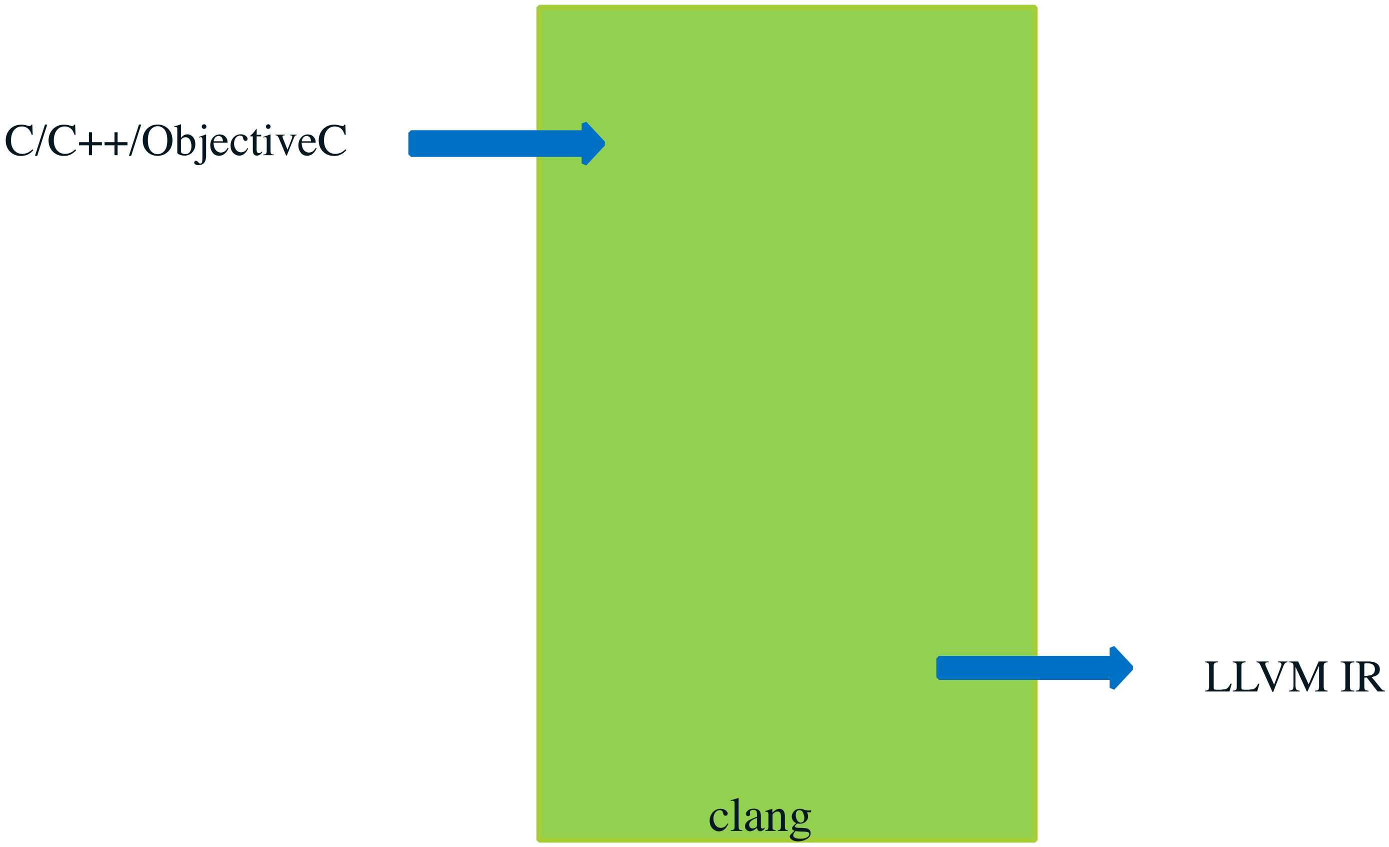


LLVM Front End

LLVM Pipeline



Clang – Design Overview

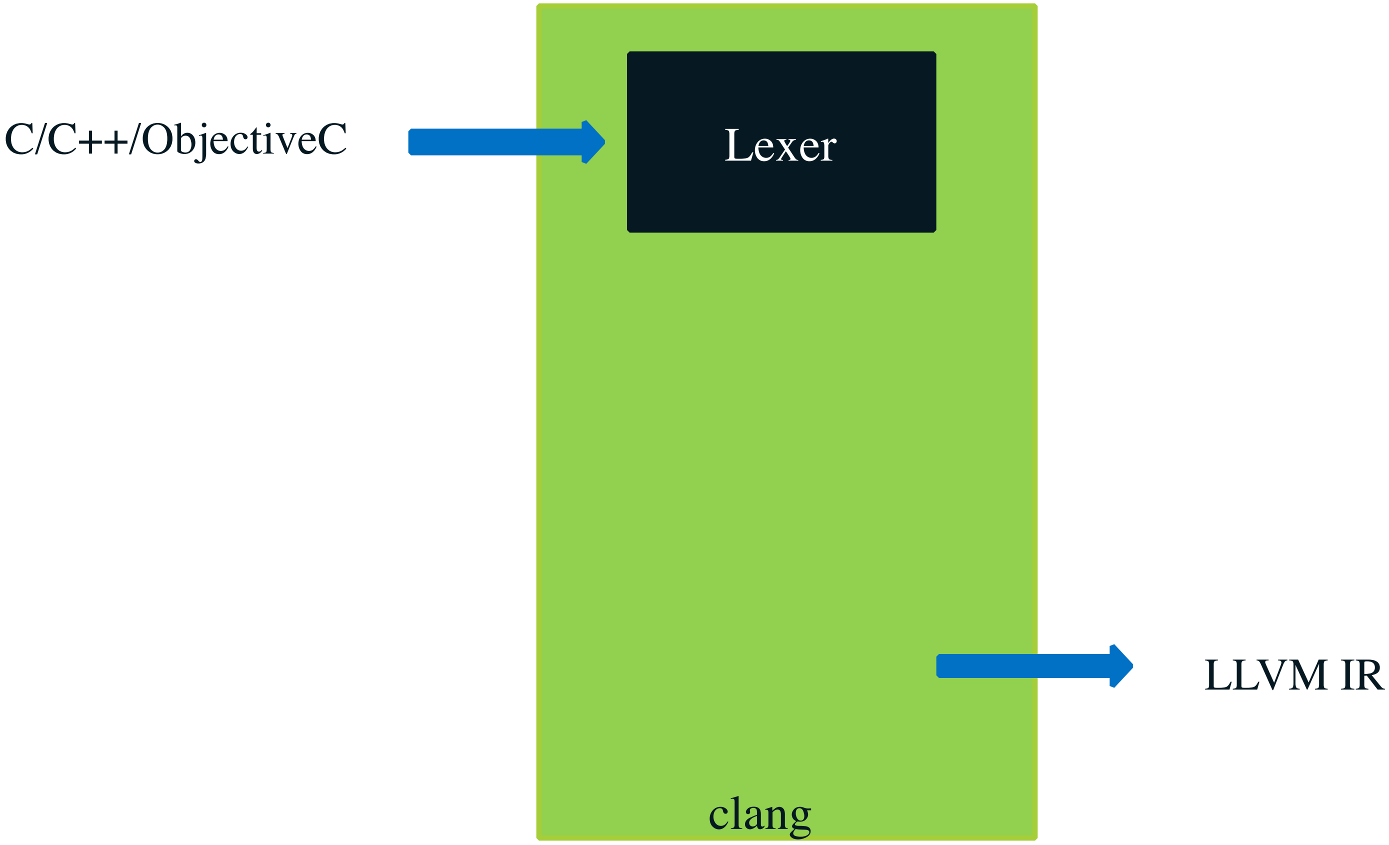


C/C++/ObjectiveC

clang

LLVM IR

Clang – Design Overview



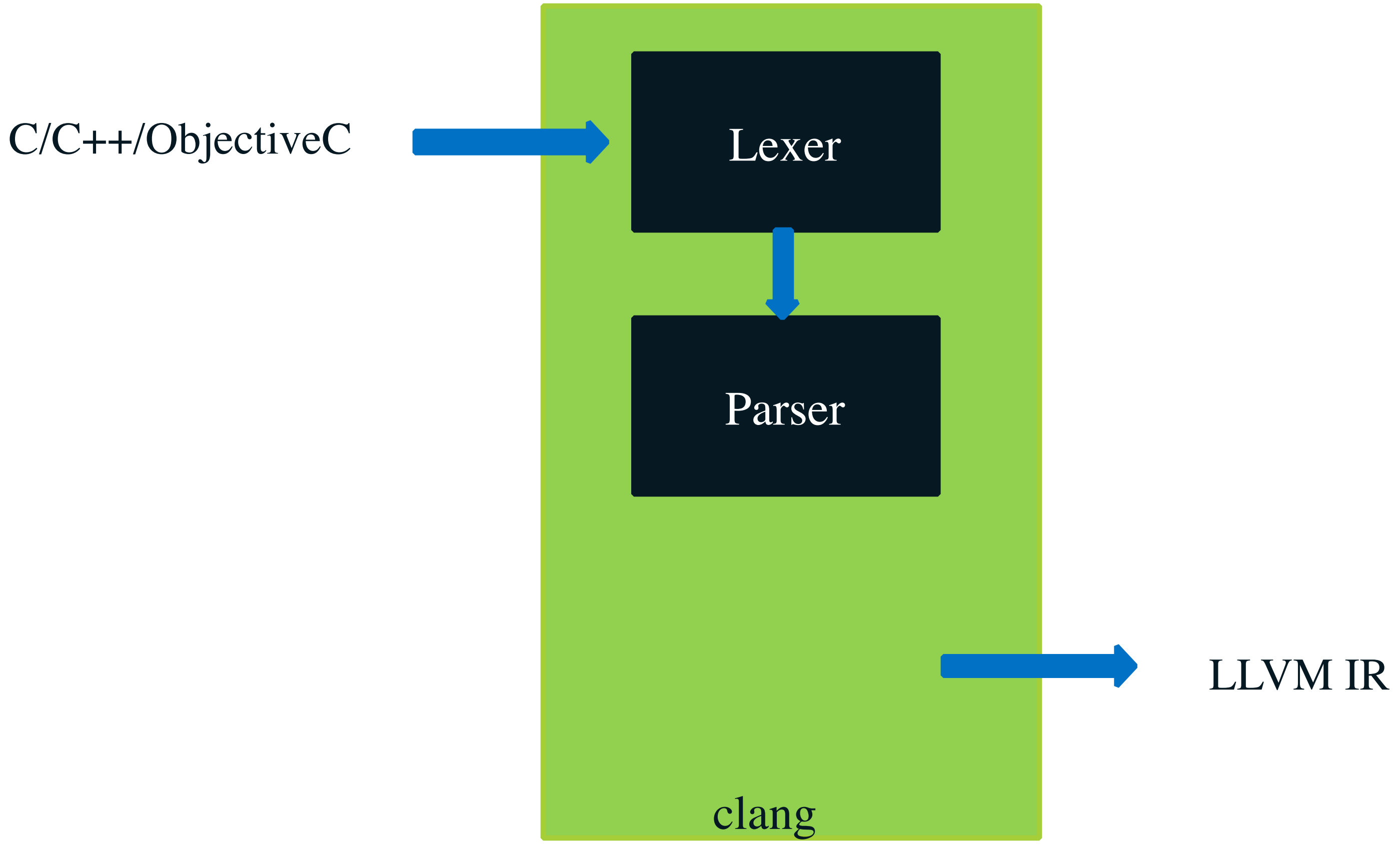
C/C++/ObjectiveC

Lexer

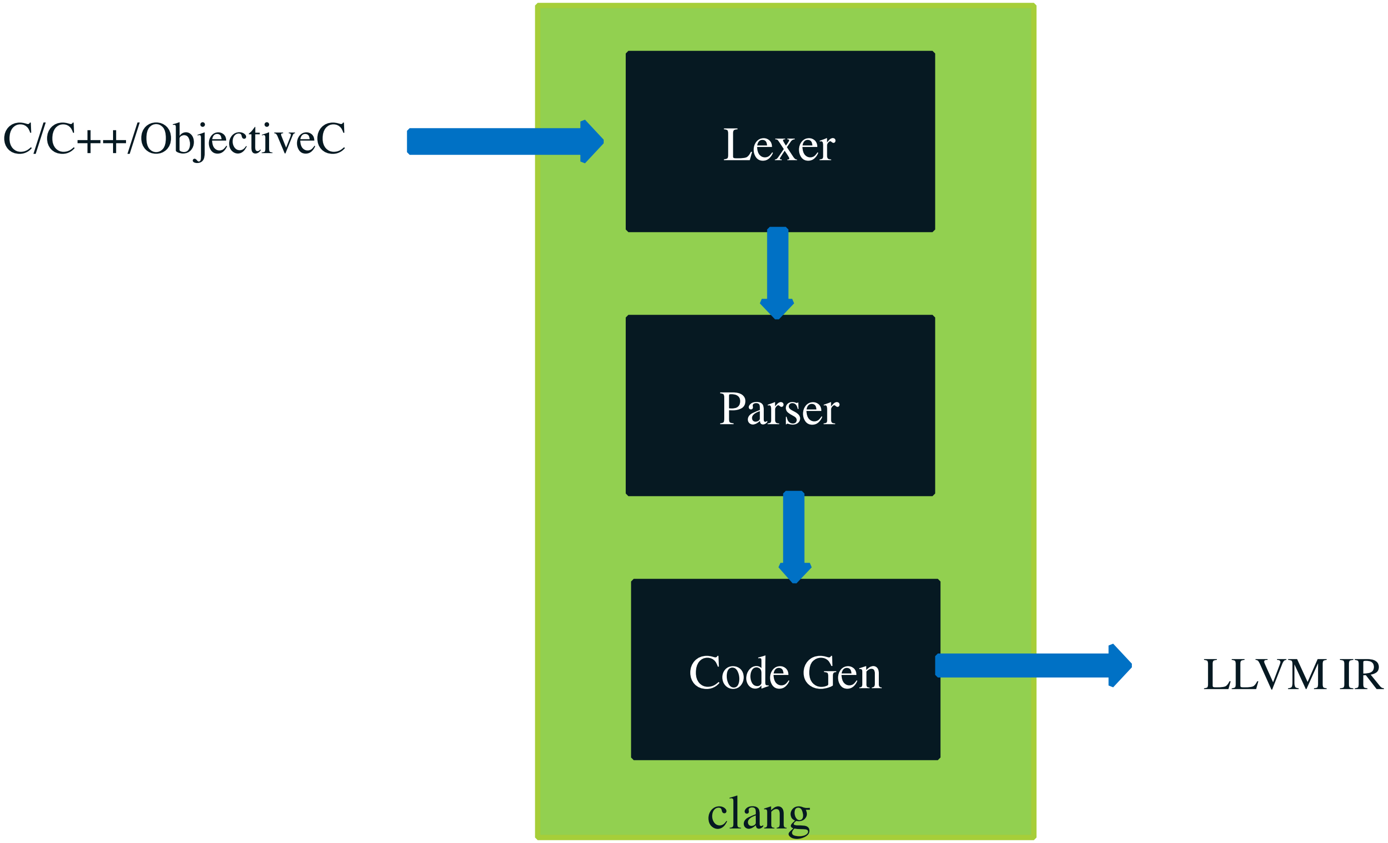
LLVM IR

clang

Clang – Design Overview



Clang – Design Overview


C/C++/ObjectiveC

Lexer

Parser

Code Gen

LLVM IR

clang

Example LLVM IR

```
int sum(int a, int b) {
  return a+b;
}
```

C function

Example LLVM IR

```c
int sum(int a, int b) {
  return a+b;
}
```

C function

```llvm
define i32 @sum(i32 %a, i32 %b)
{
entry:
  %add = add i32 %b, %a
  ret i32 %add
}
```

LLVM IR function

Example LLVM IR SSA

```
int mul_add(
int a,
int b,
int c)
{
  b = a * b;
  b = c + b;
  return b;
}
```

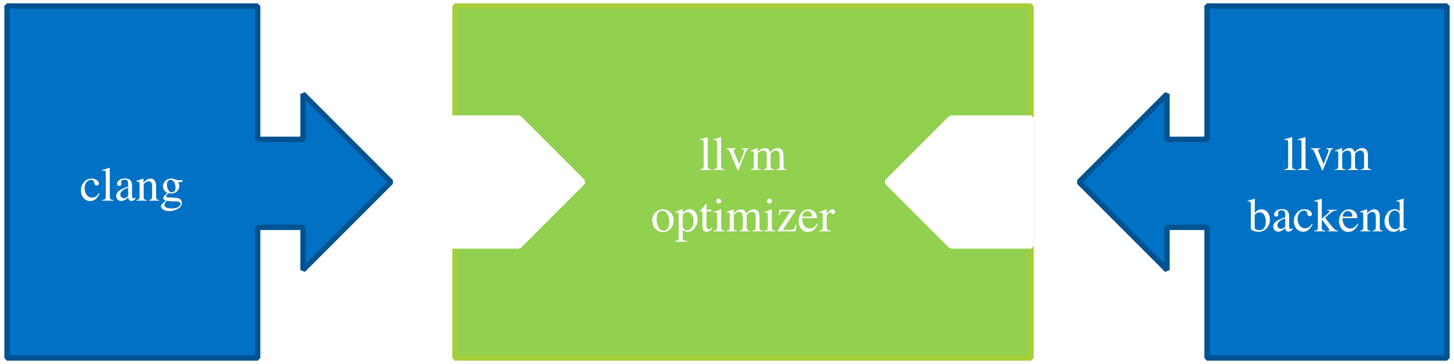
C function

```
define i32 @mul_add(
i32 %a,
i32 %b,
i32 %c)
{
entry:
  %mul = mul i32 %b, %a
  %add = add i32 %mul, %c
  ret i32 %add
}
```

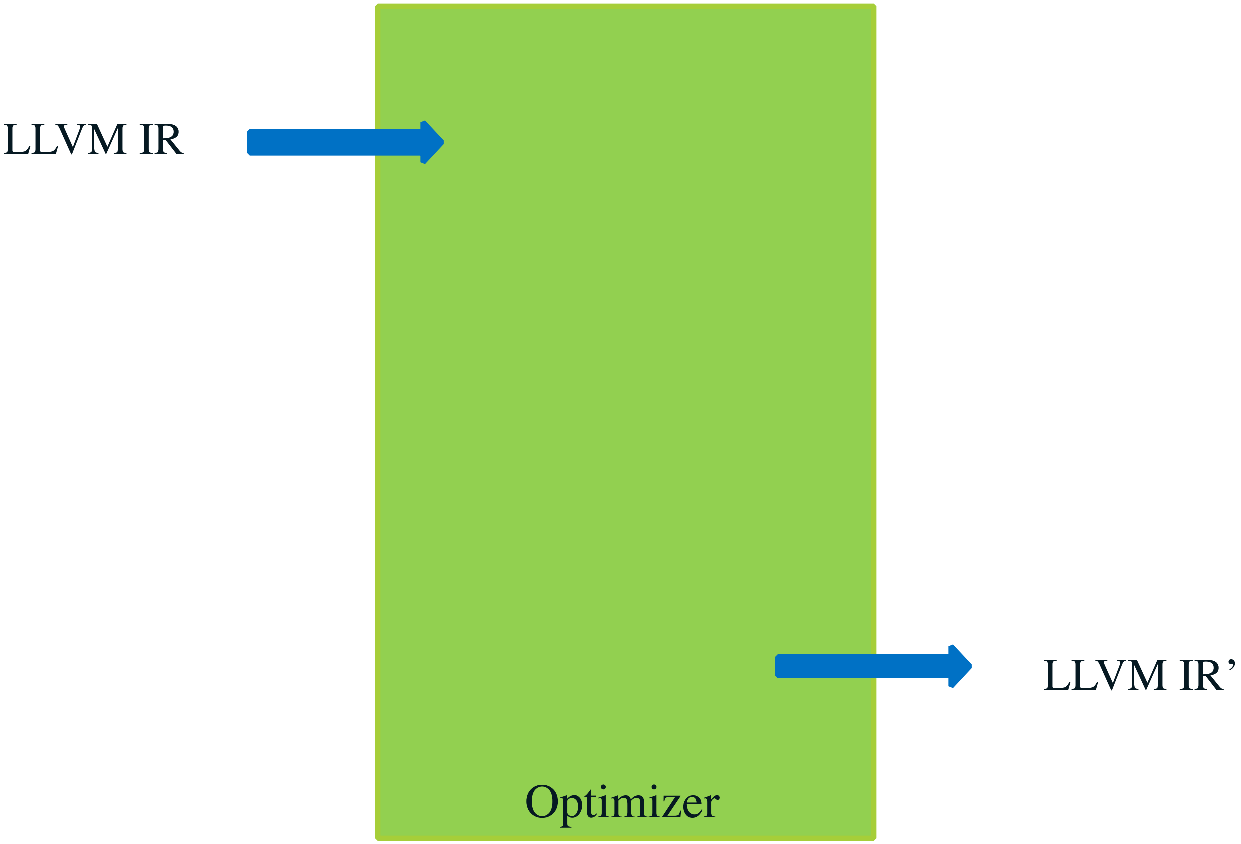
LLVM IR function

LLVM Mid End Optimizer

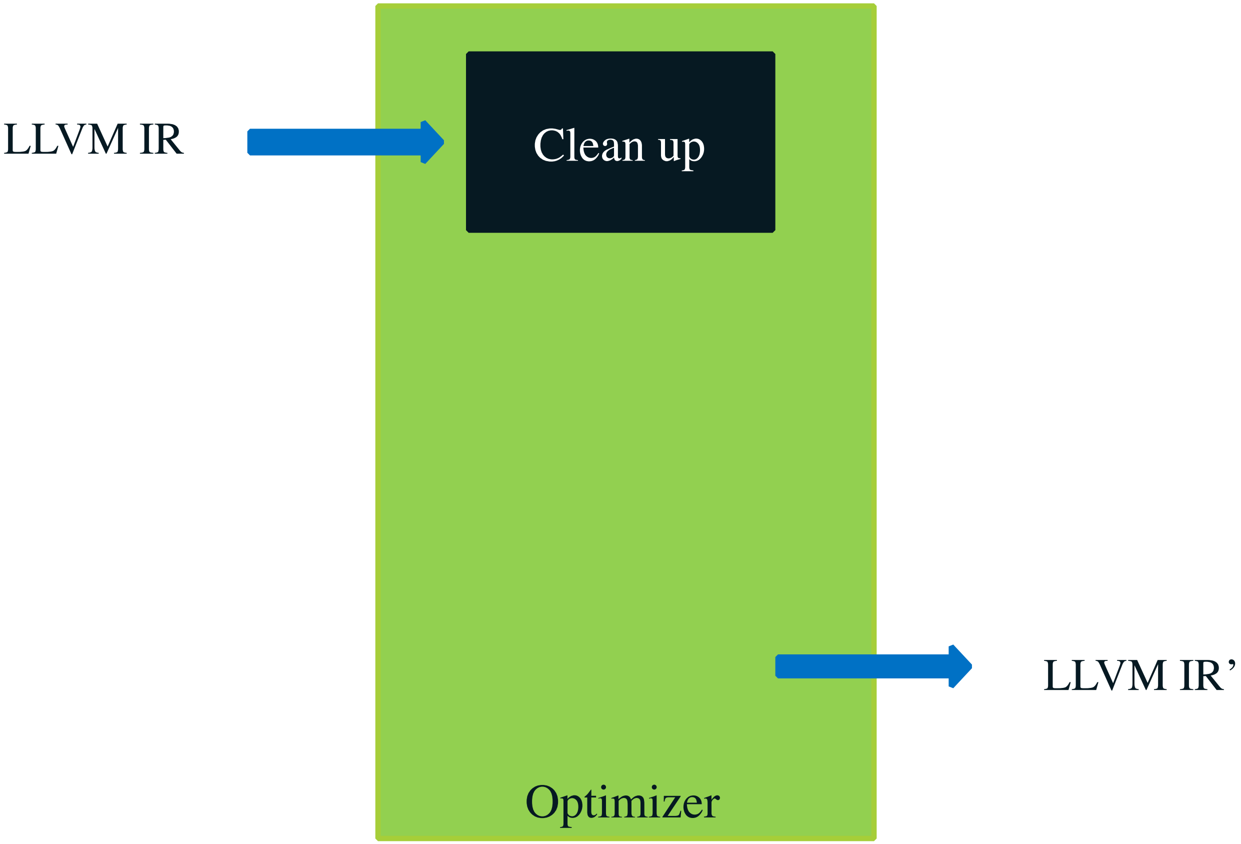
LLVM Pipeline



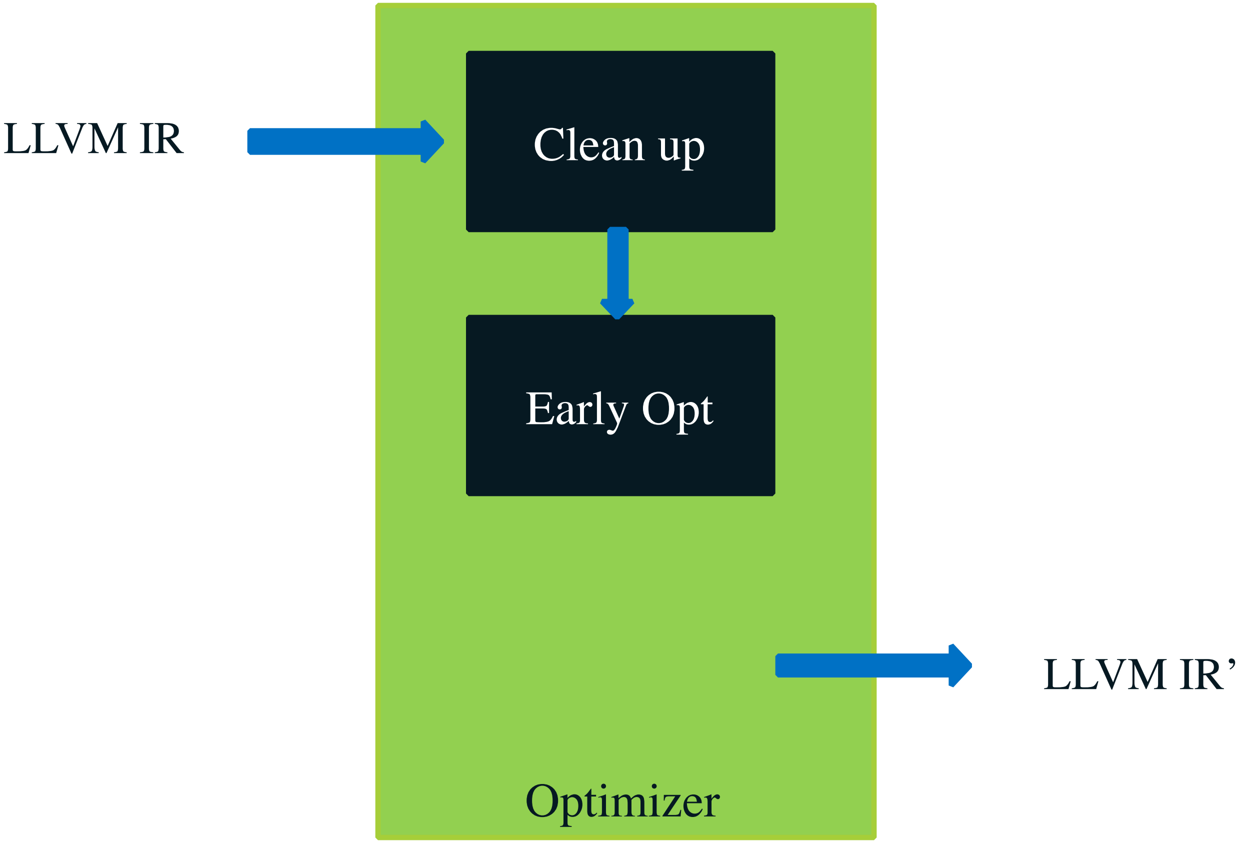
LLVM Mid End Optimizer – Design Overview



LLVM Mid End Optimizer – Design Overview



LLVM Mid End Optimizer – Design Overview



LLVM IR

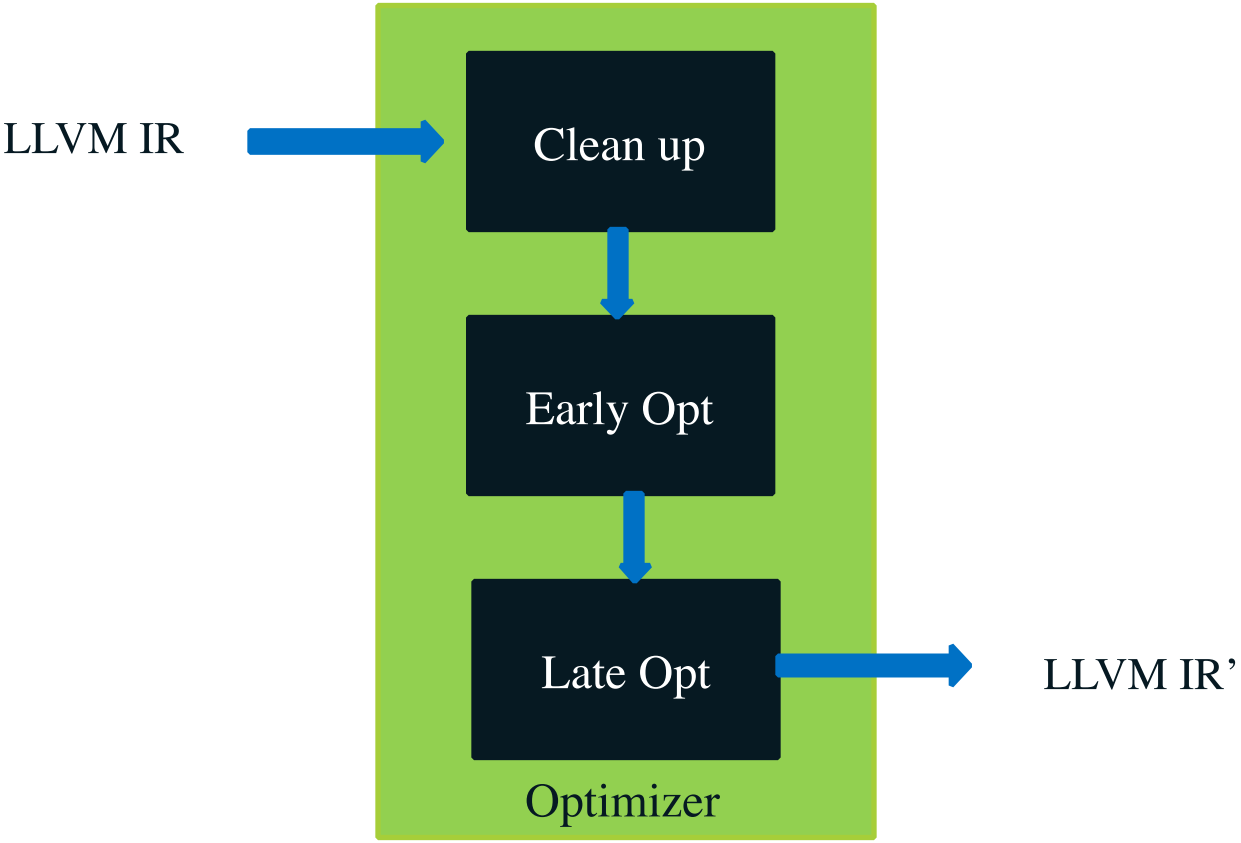
Clean up

Early Opt

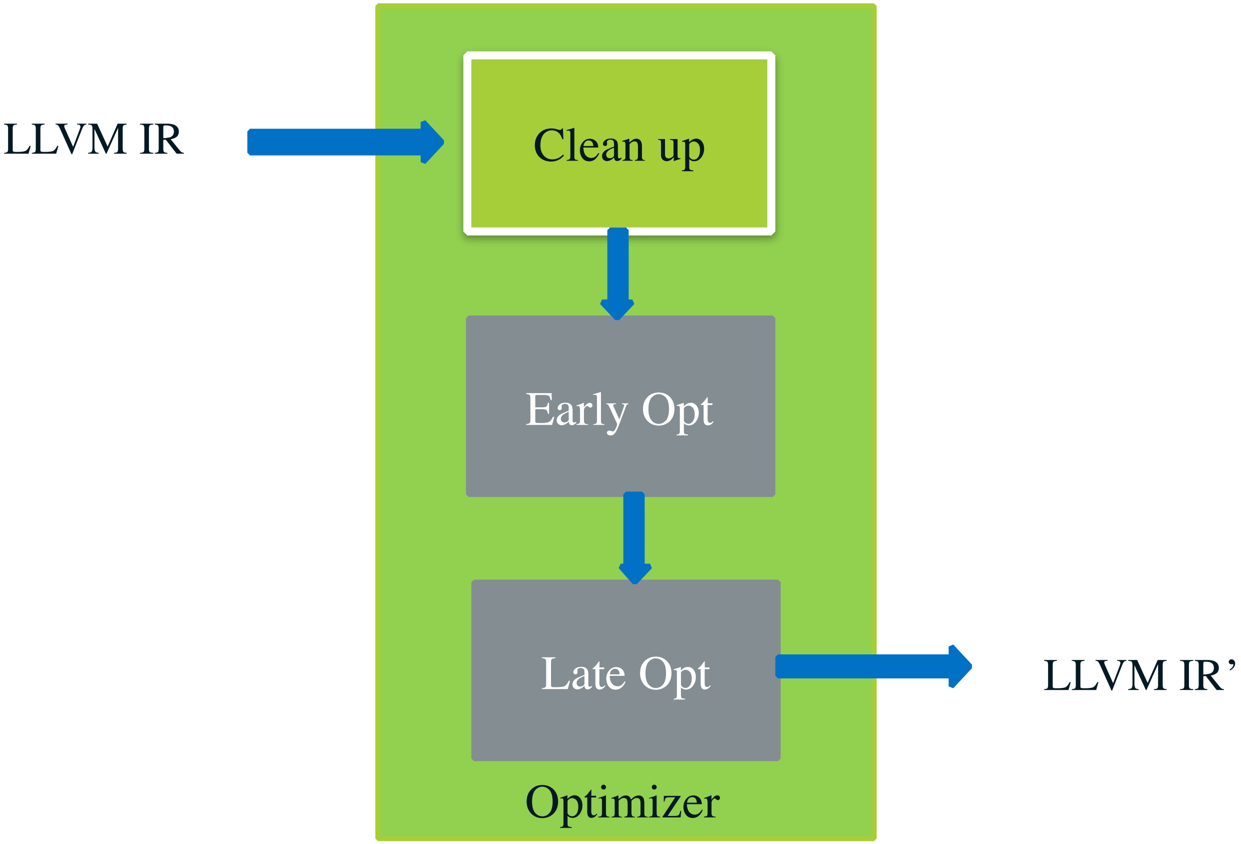
Optimizer

LLVM IR'

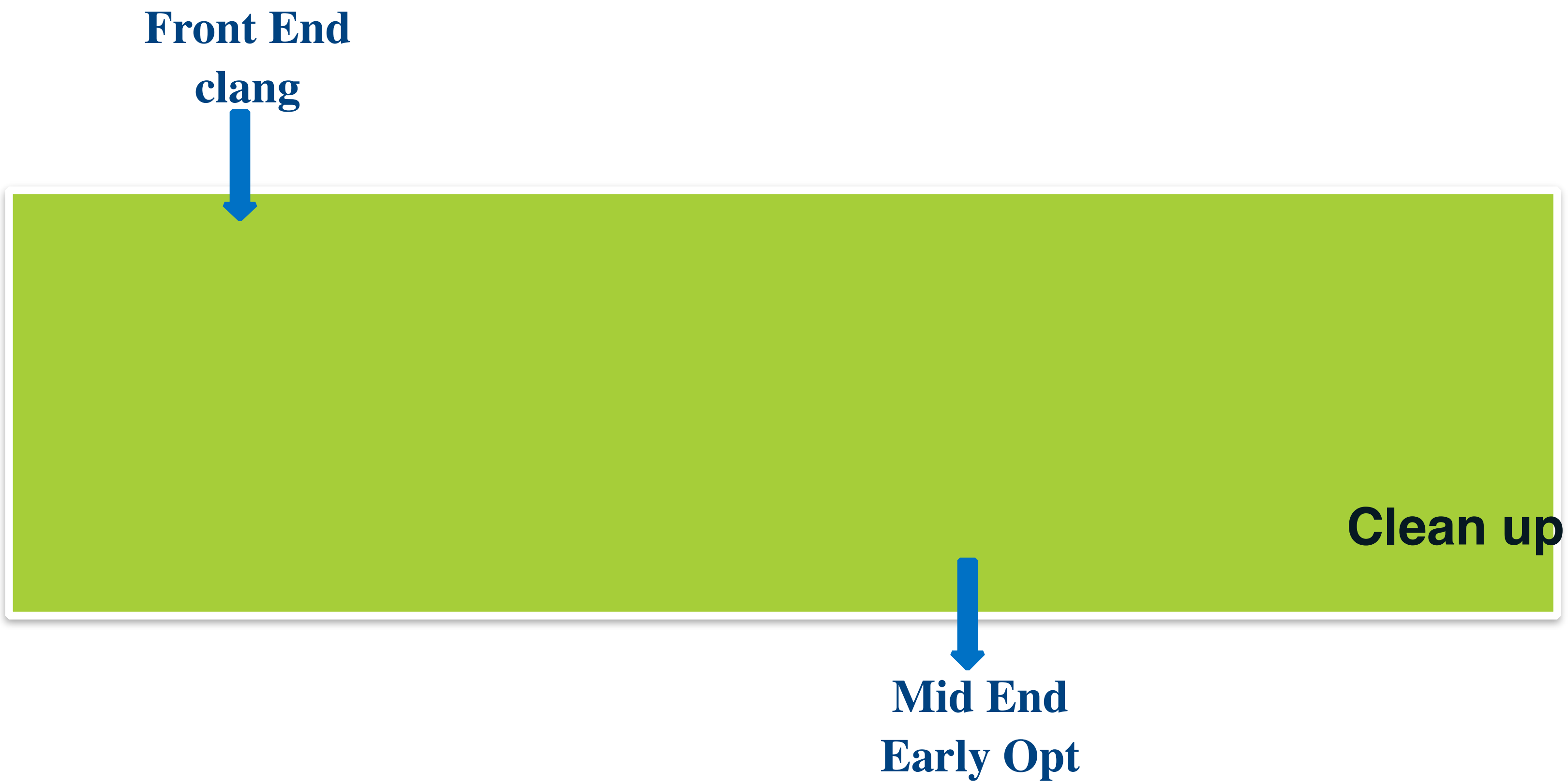
LLVM Mid End Optimizer – Design Overview



Mid End Optimizer – Clean up



Mid End Optimizer – Clean up

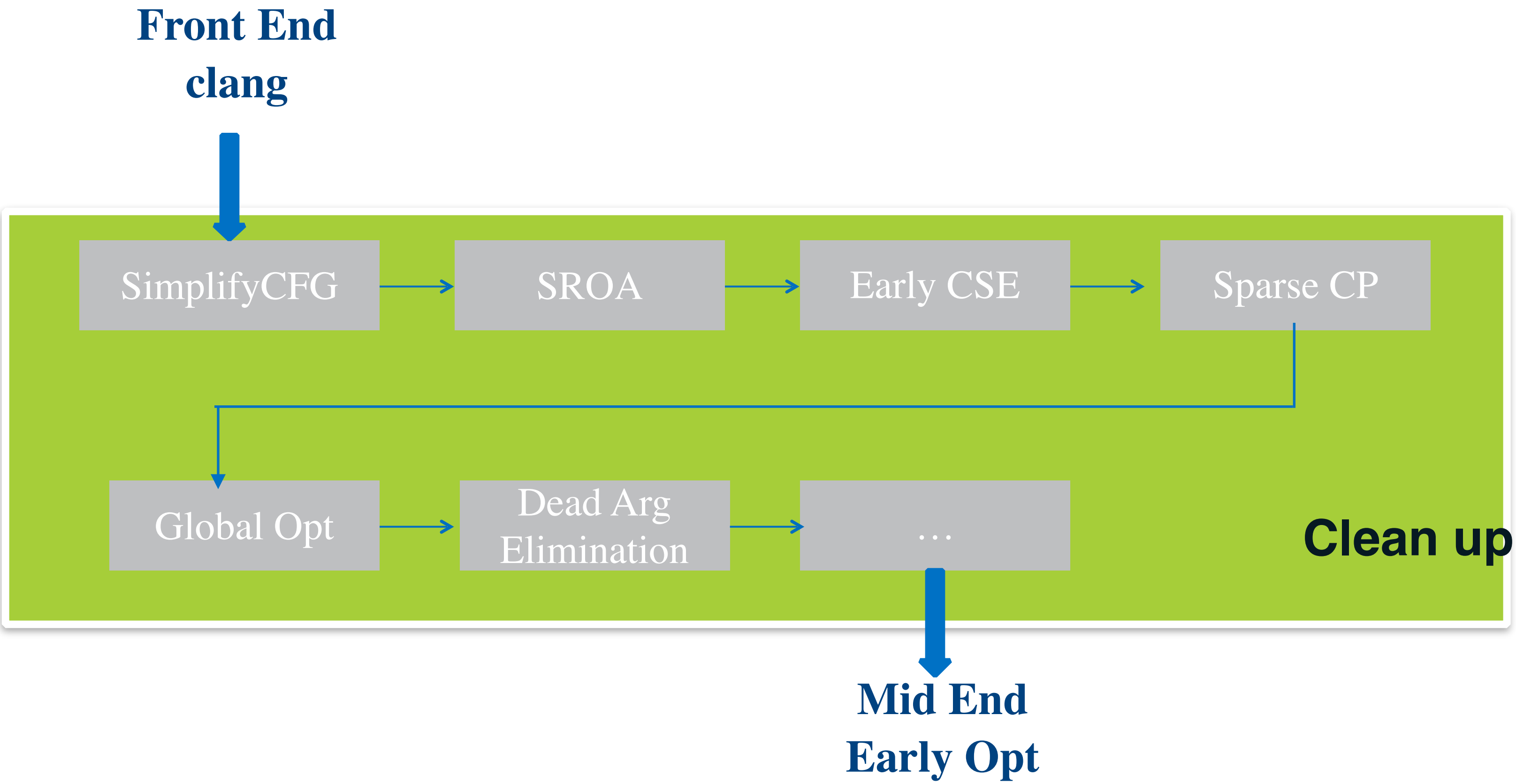


Front End
clang

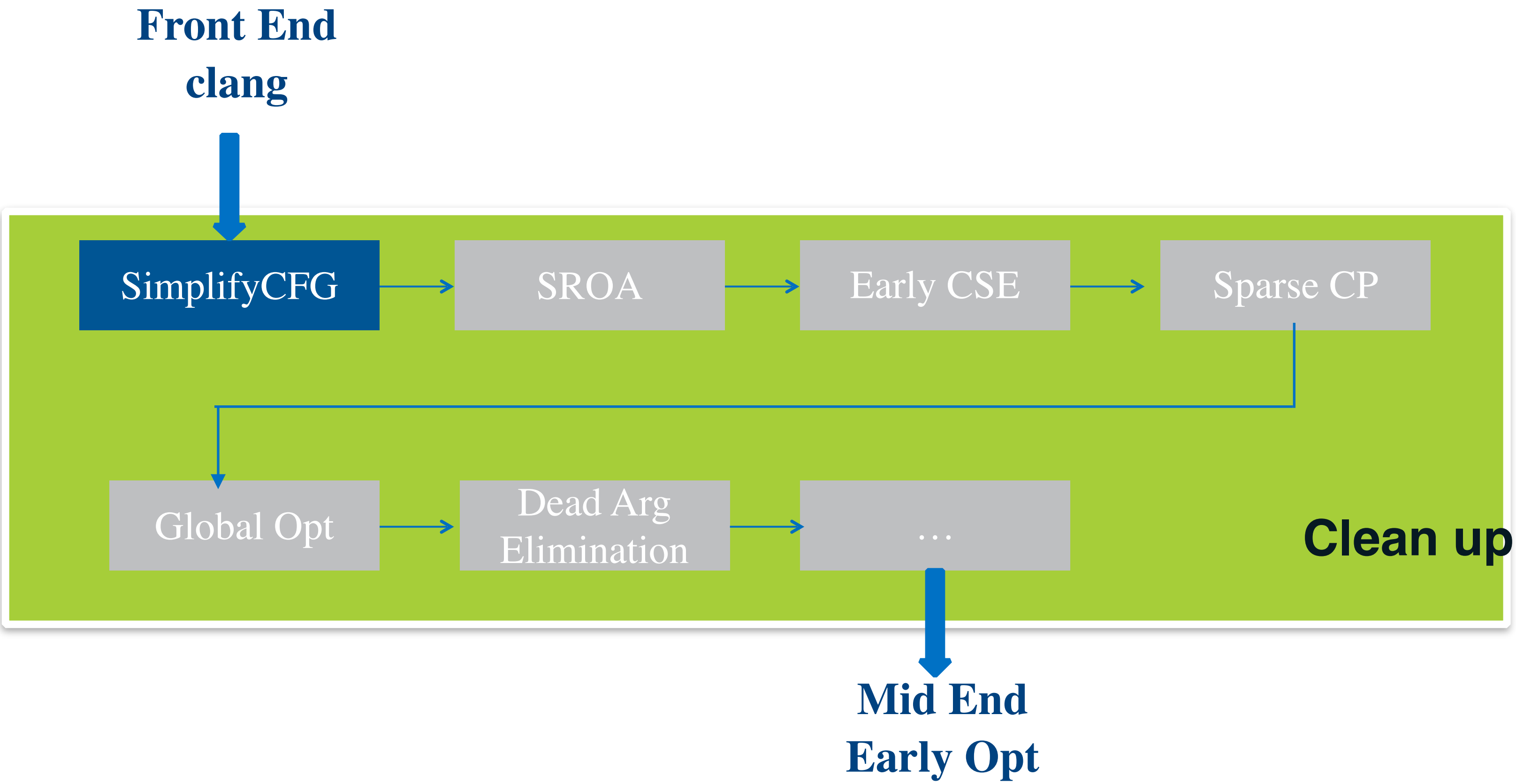
Clean up

Mid End
Early Opt

Mid End Optimizer – Clean up

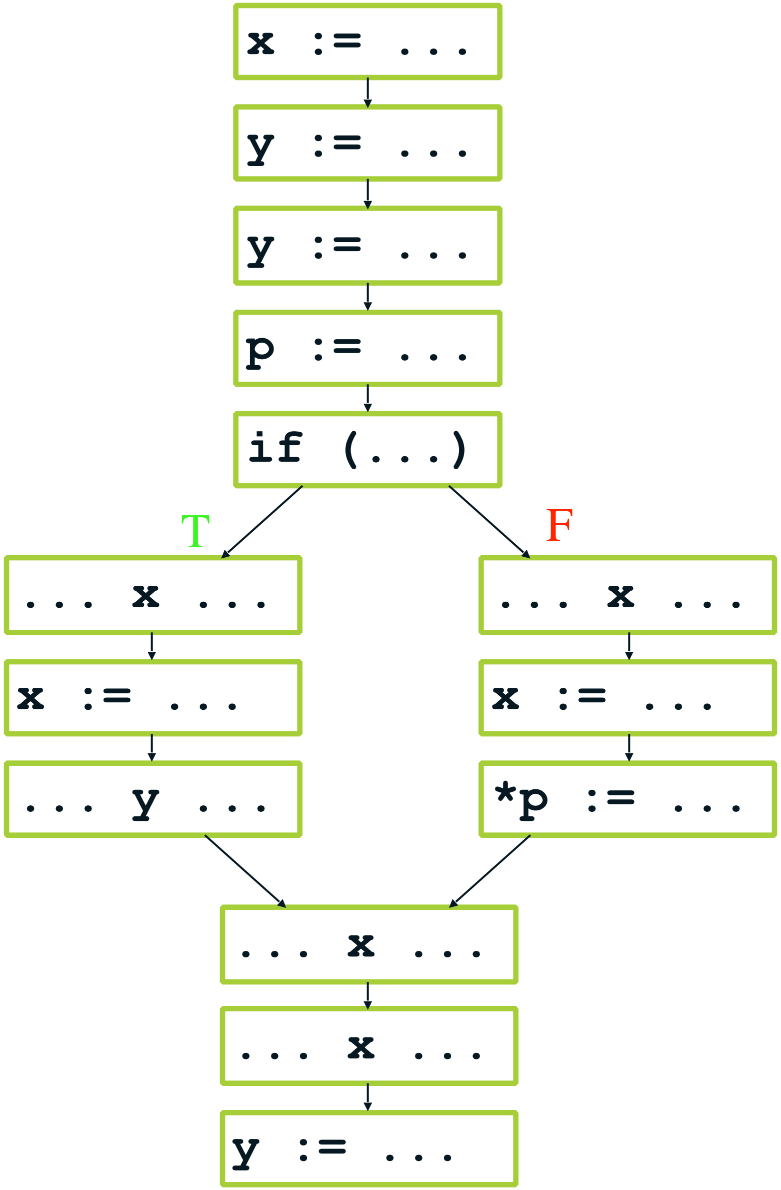


Mid End Optimizer – Clean up



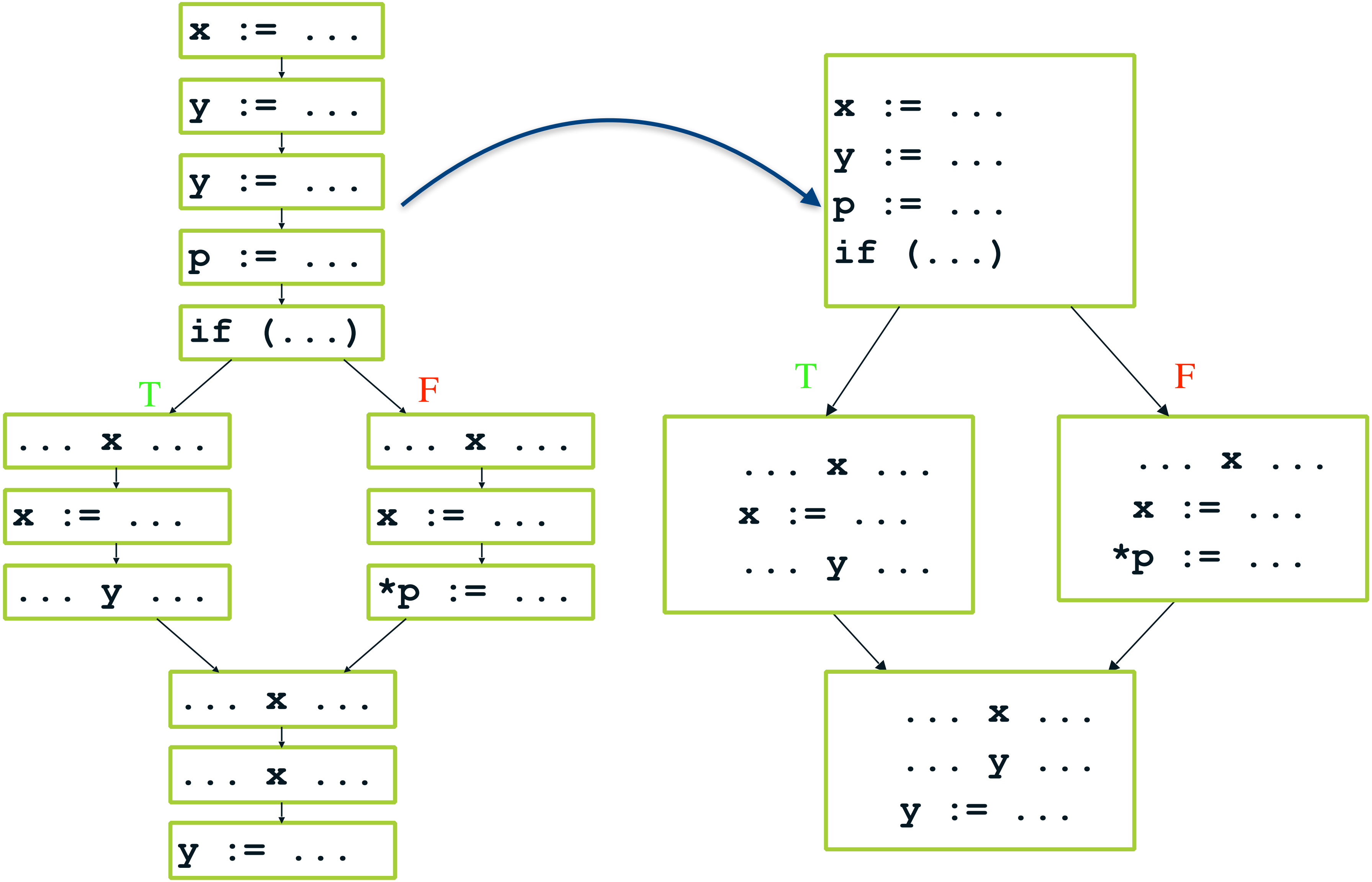
Example CFG

```
x := ...
y := ...
y := ...
p := ...
if (...) {
   ... x ...
   x := ...
   ... y ...
}
else {
   ... x ...
   x := ...
   *p := ...
}
... x ...
... y ...
y := ...
```

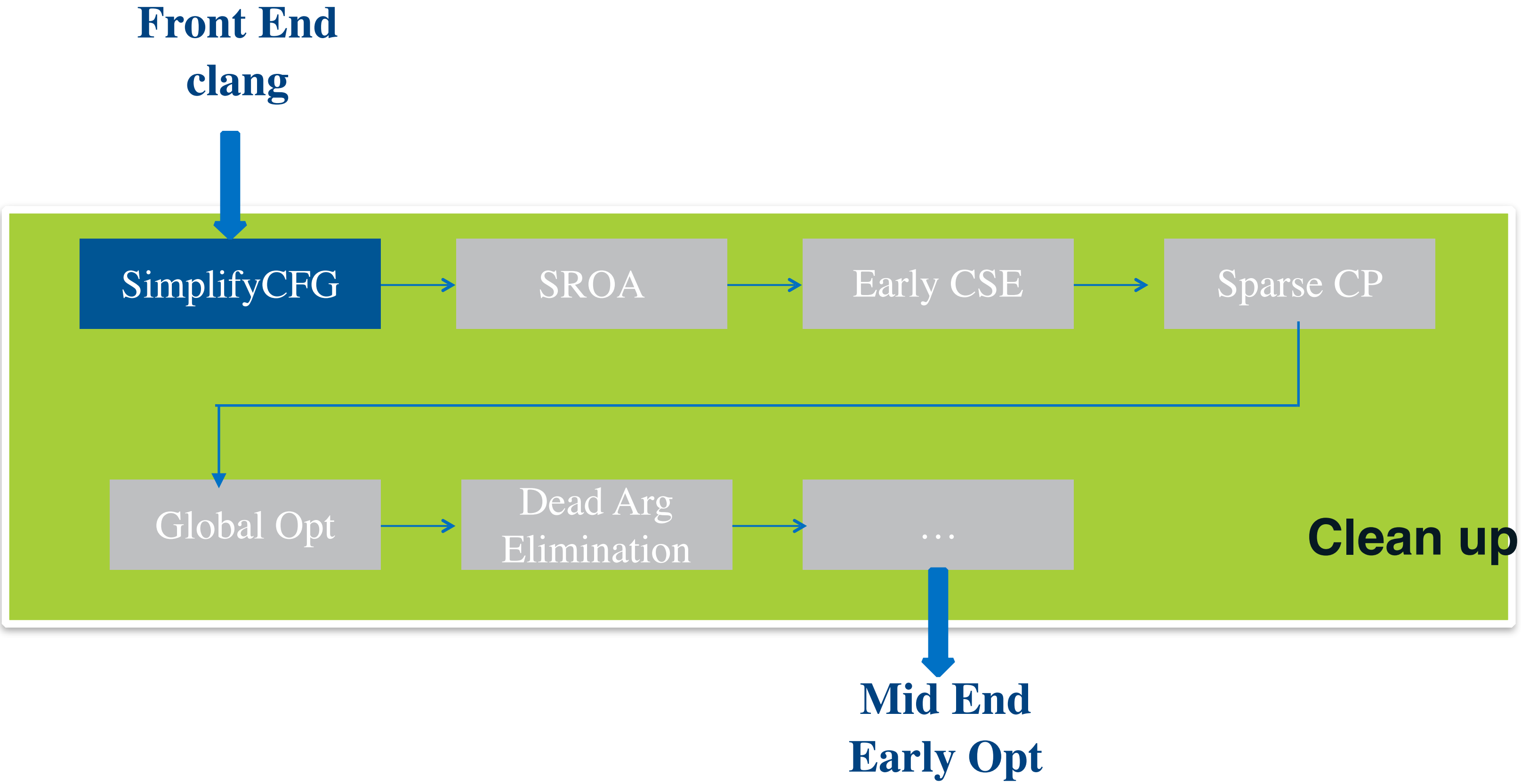


```
x := ...
y := ...
y := ...
p := ...
if (...)
T
... x ...
x := ...
... y ...
F
... x ...
x := ...
*p := ...
... x ...
... x ...
y := ...
```

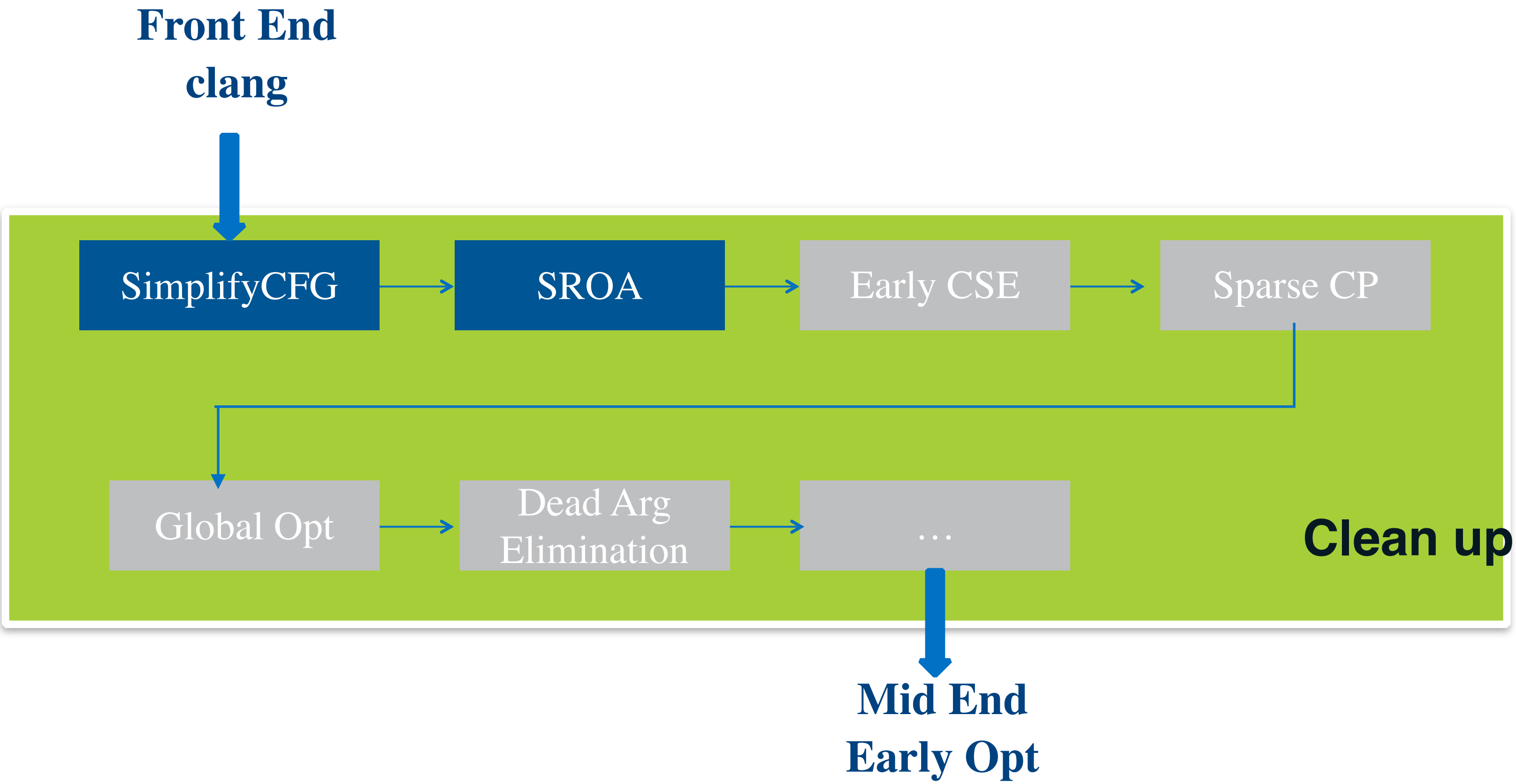
Simplified CFG



Mid End Optimizer – Clean up



Mid End Optimizer – Clean up



Front End
clang

SimplifyCFG

SROA

Early CSE

Sparse CP

Global Opt

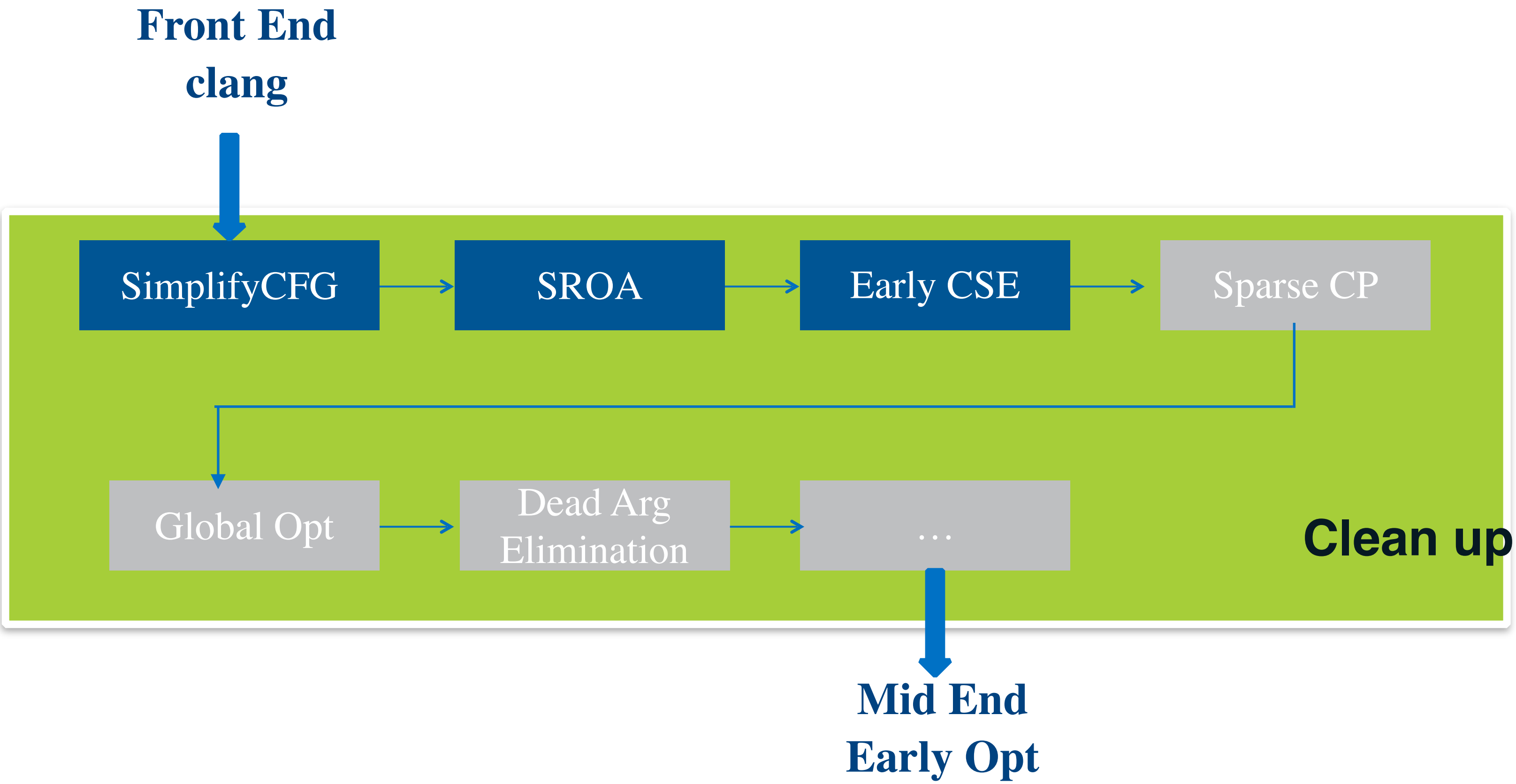
Dead Arg Elimination

…

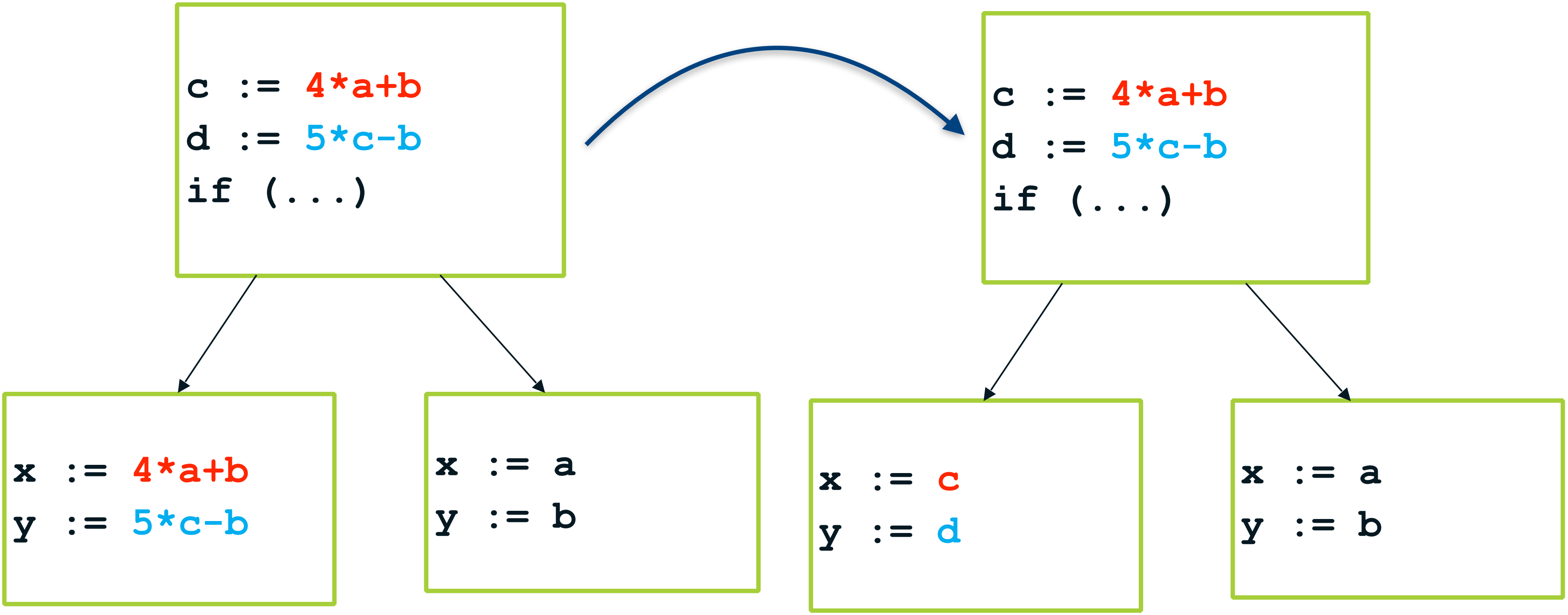
Clean up

Mid End
Early Opt

Mid End Optimizer – Clean up



Common Subexpression Elimination



```
c := 4*a+b
d := 5*c-b
if (...)
```

```
x := 4*a+b
y := 5*c-b
```

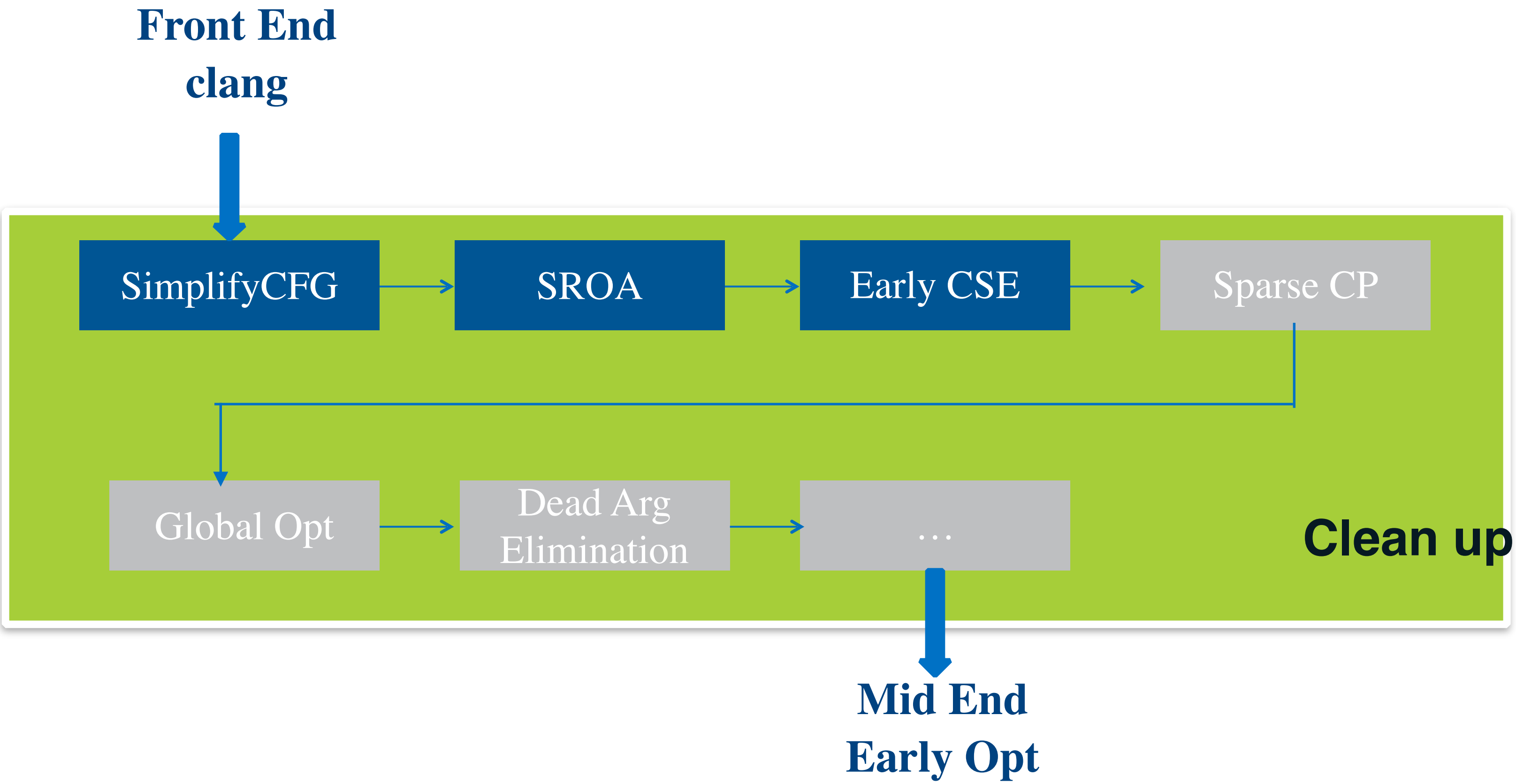
```
x := a
y := b
```

```
c := 4*a+b
d := 5*c-b
if (...)
```

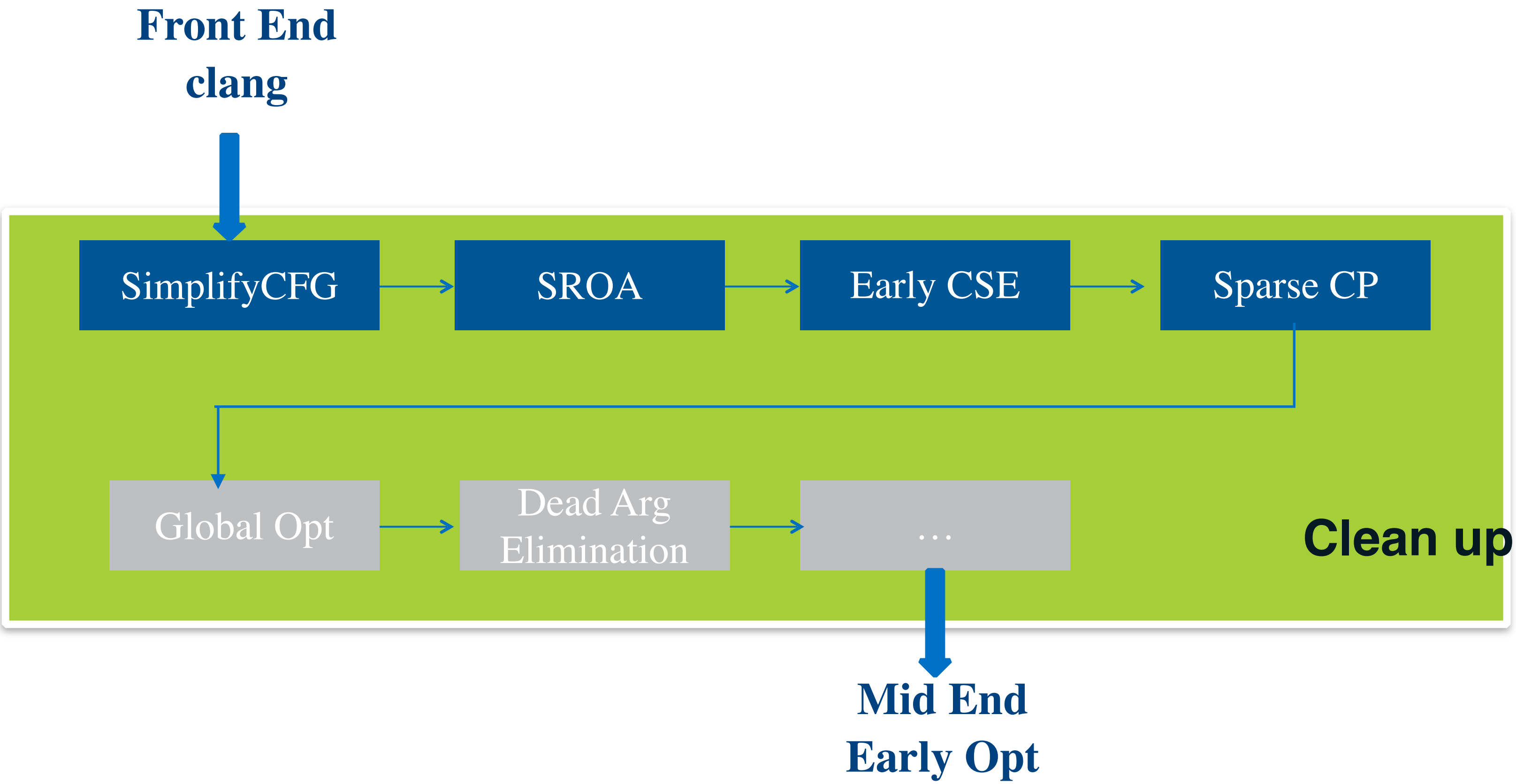
```
x := c
y := d
```

```
x := a
y := b
```

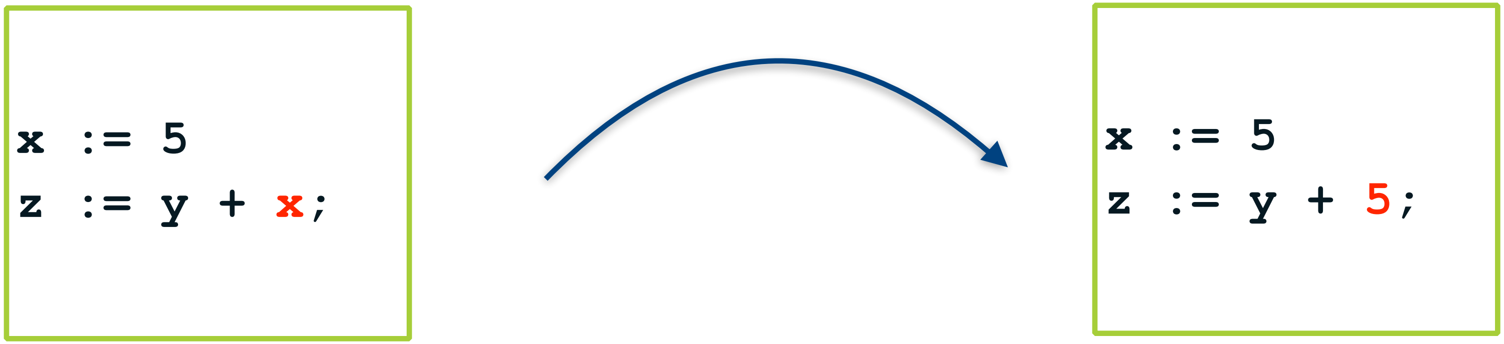
Mid End Optimizer – Clean up



Mid End Optimizer – Clean up



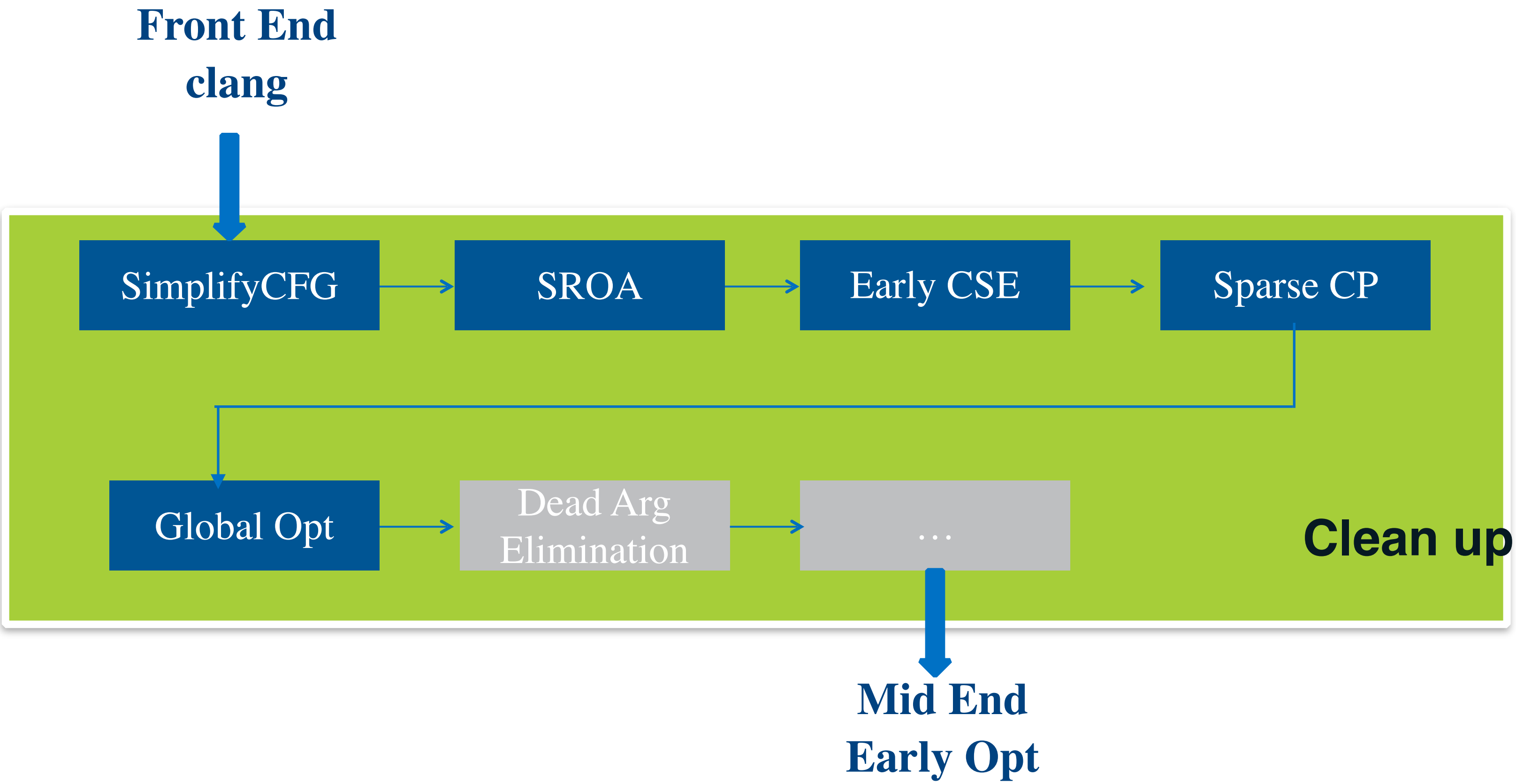
Constant Propagation



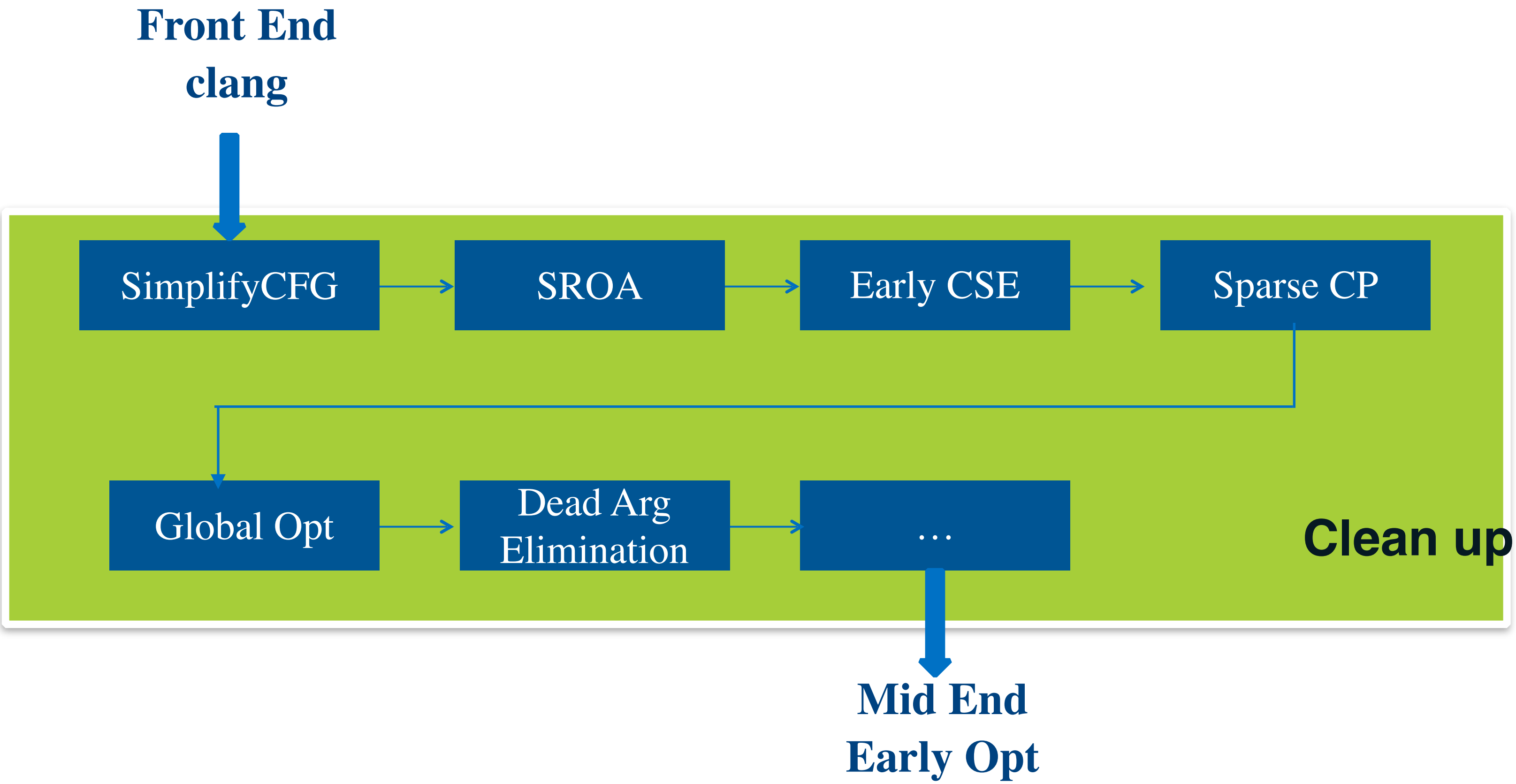
```
x := 5
z := y + x;
```

```
x := 5
z := y + 5;
```

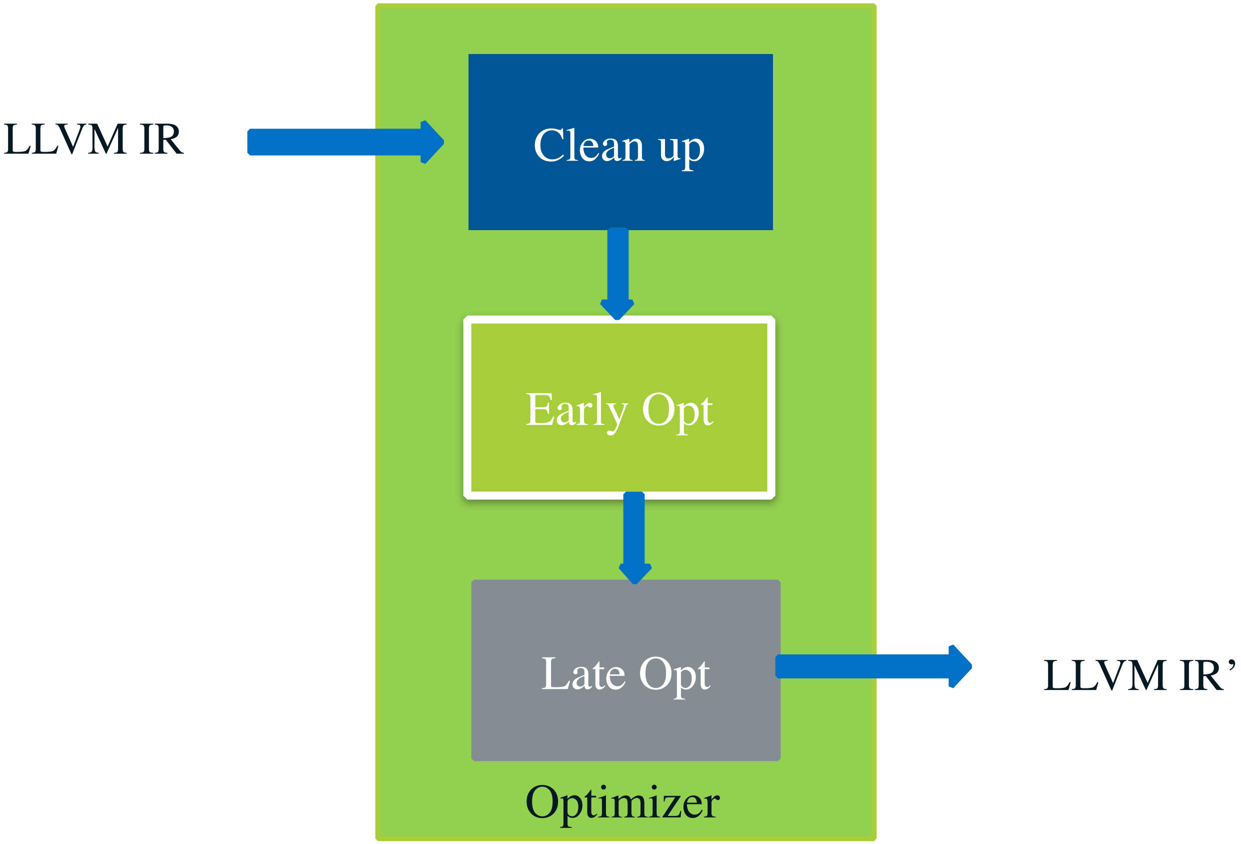
Mid End Optimizer – Clean up



Mid End Optimizer – Clean up



Mid End Optimizer – Early Opt



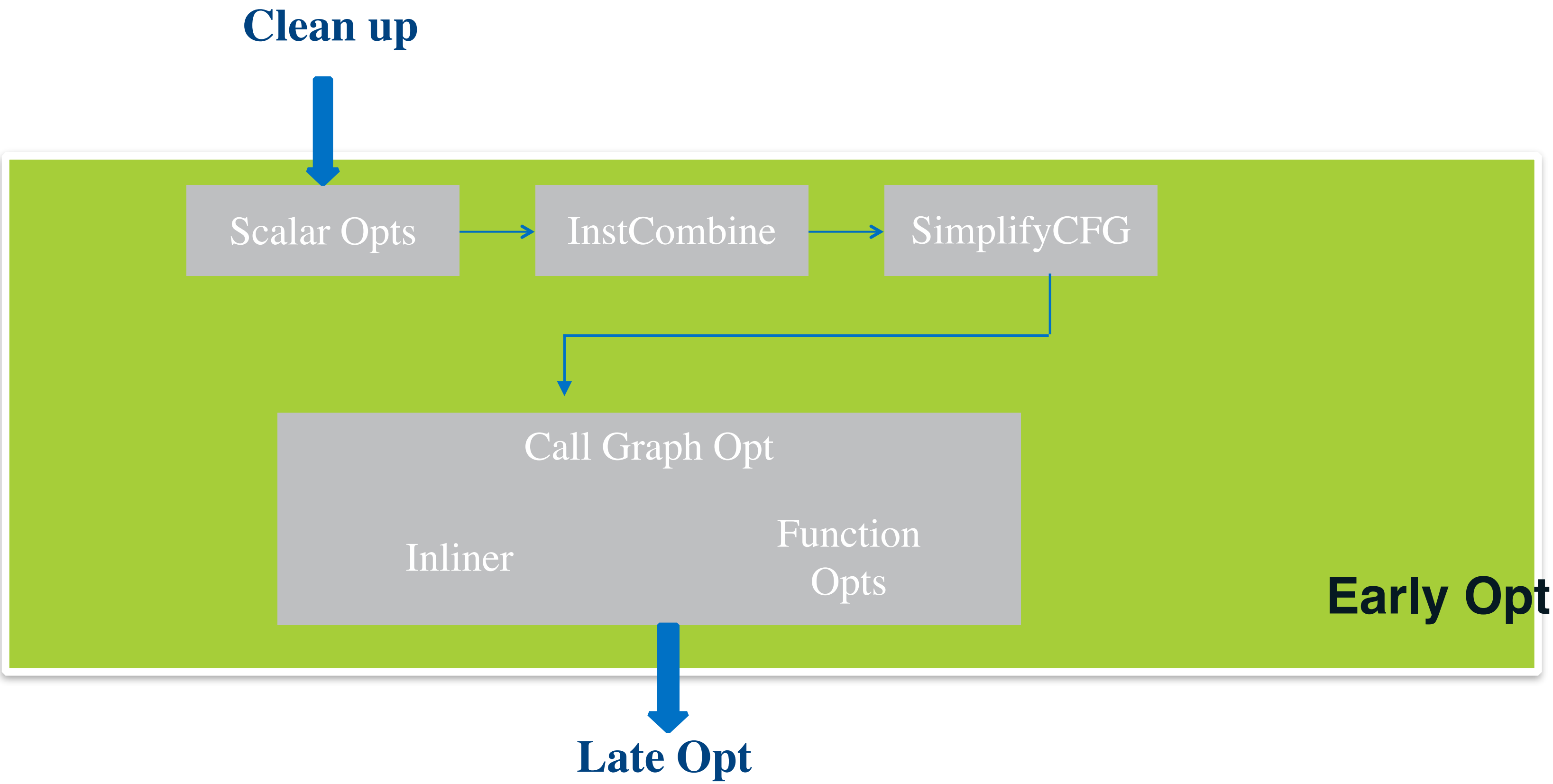
Mid End Optimizer – Early Opt

Clean up

Early Opt

Late Opt

Mid End Optimizer – Early Opt



Clean up

Scalar Opts

InstCombine

SimplifyCFG

Call Graph Opt

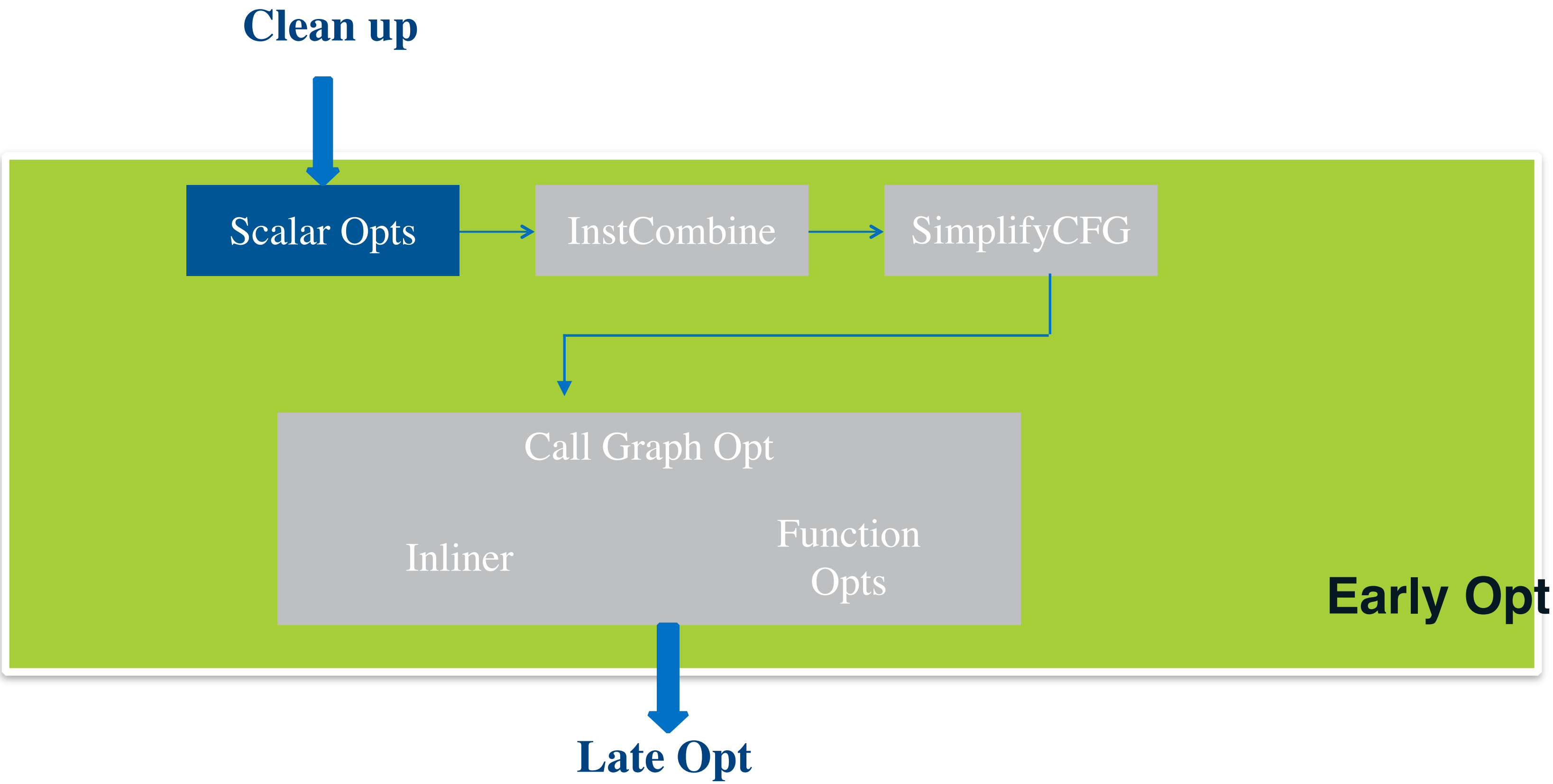
Inliner

Function Opts

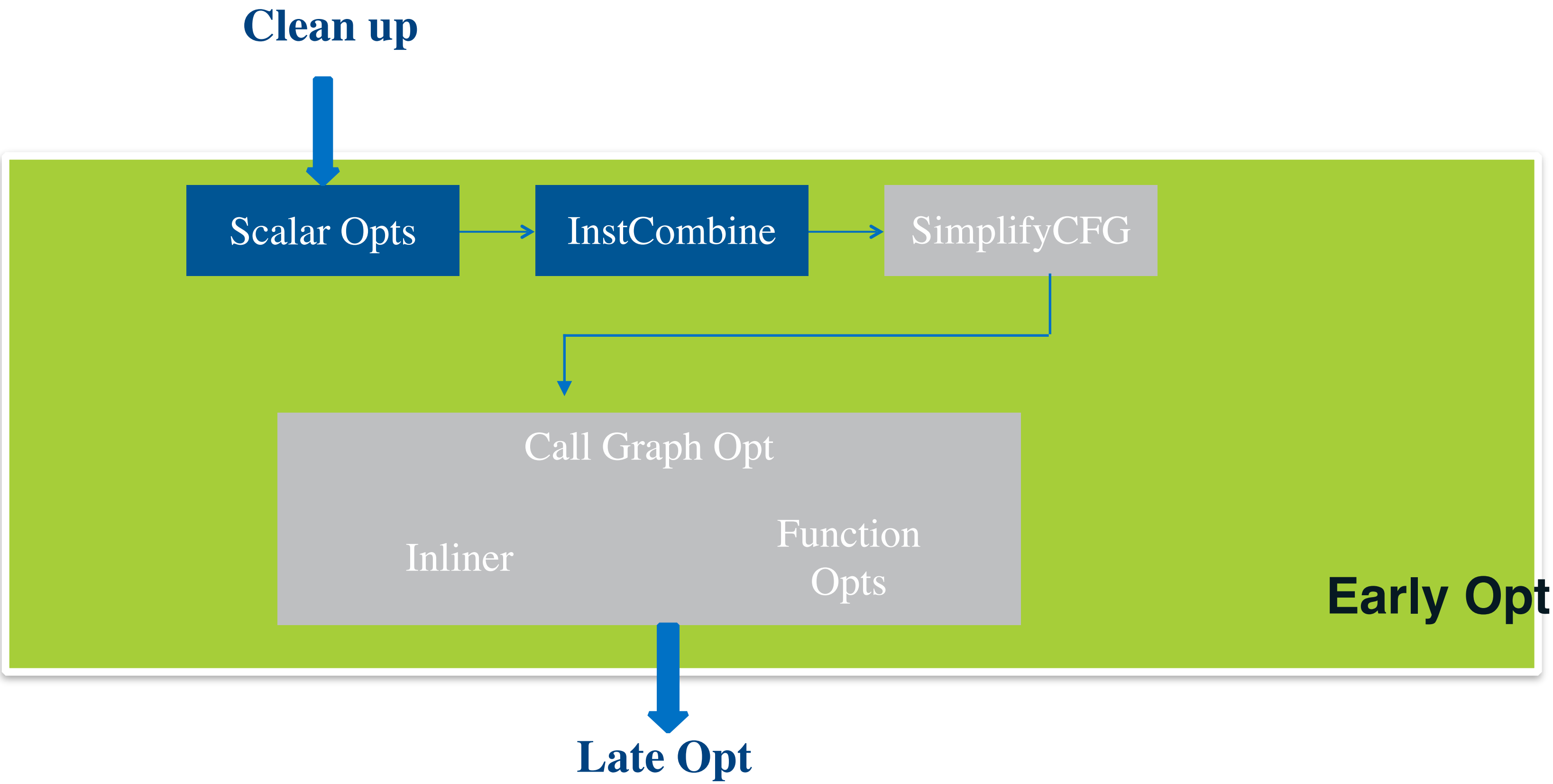
Early Opt

Late Opt

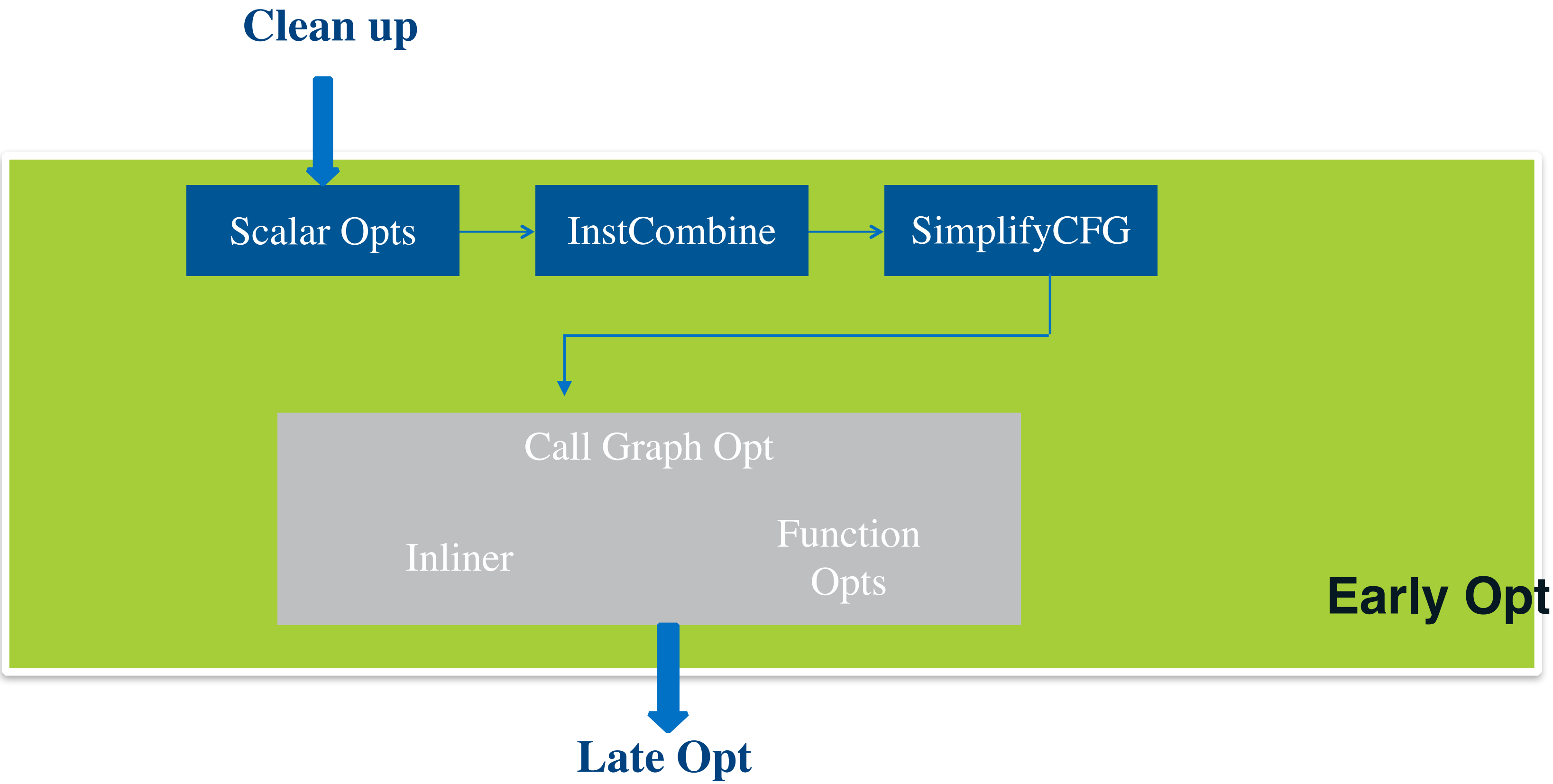
Mid End Optimizer – Early Opt



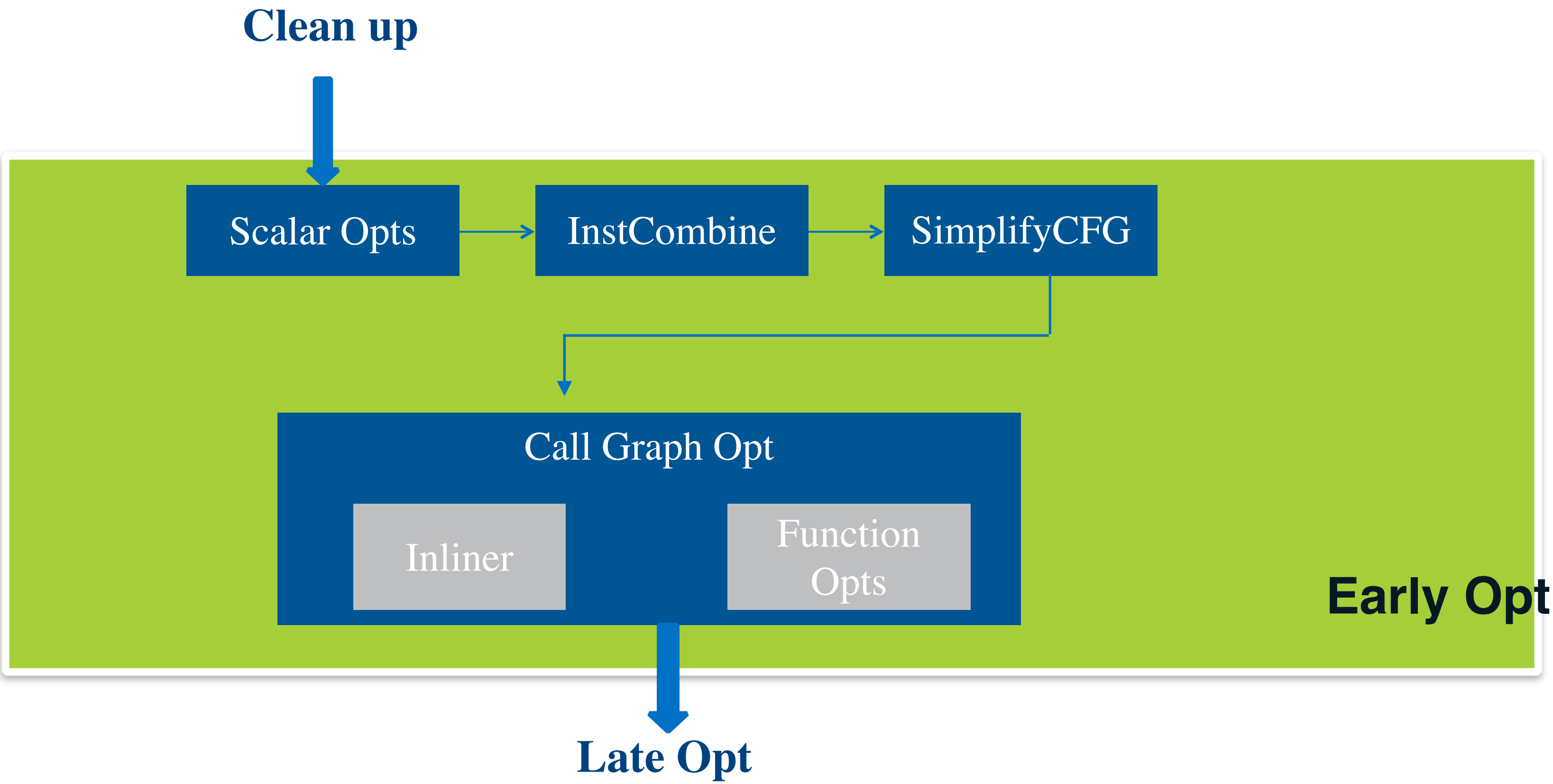
Mid End Optimizer – Early Opt



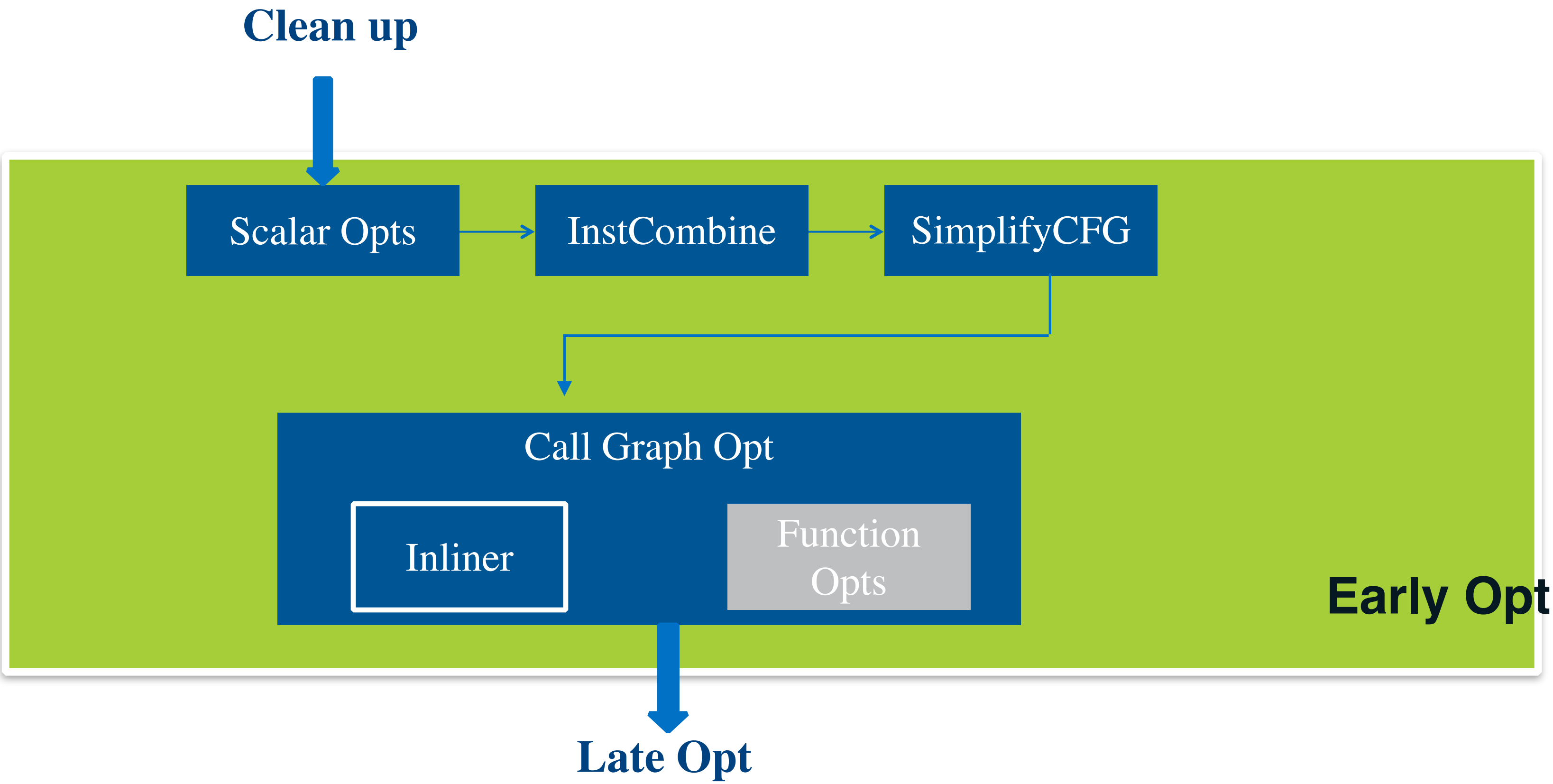
Mid End Optimizer – Early Opt



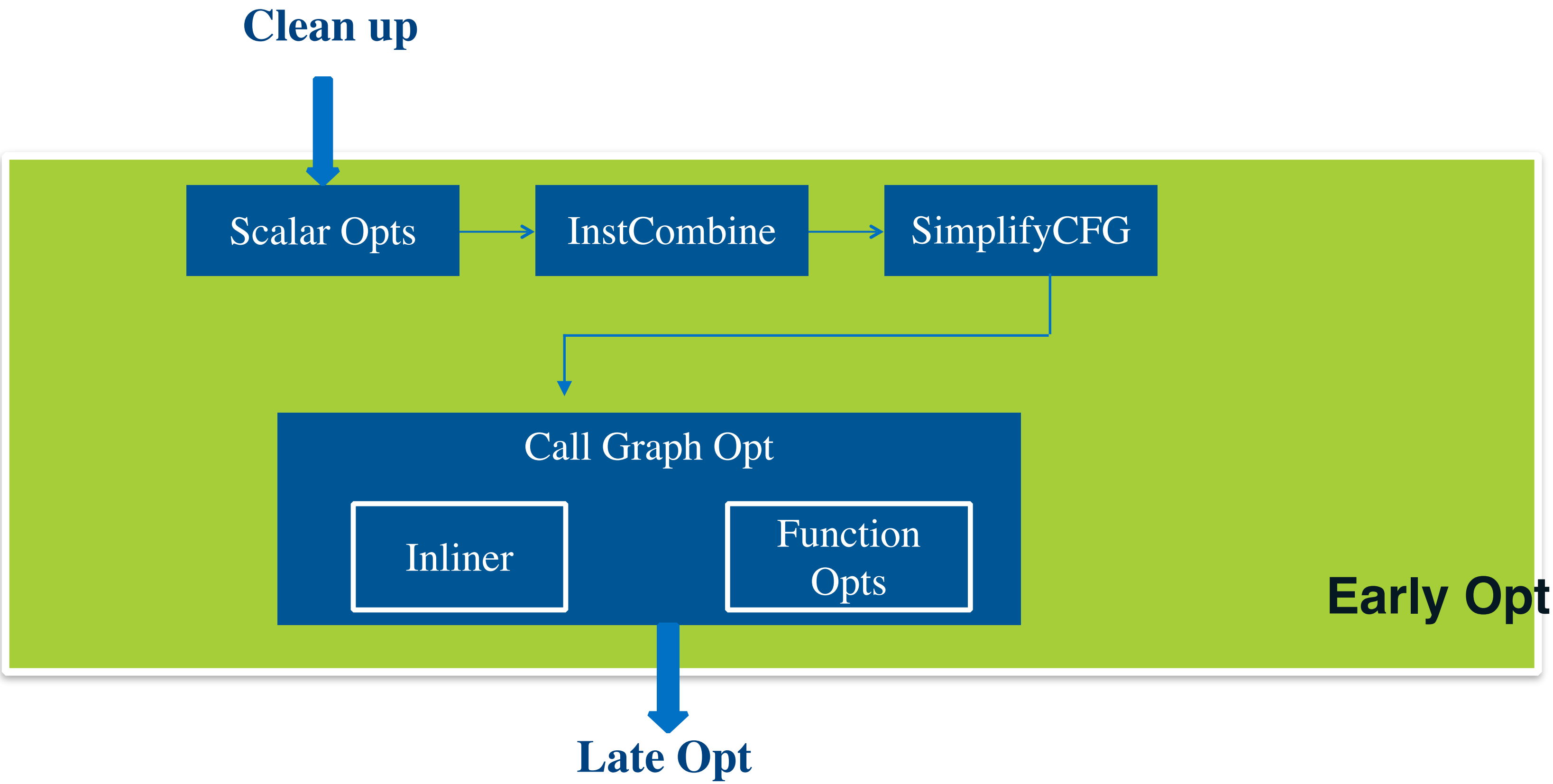
Mid End Optimizer – Early Opt



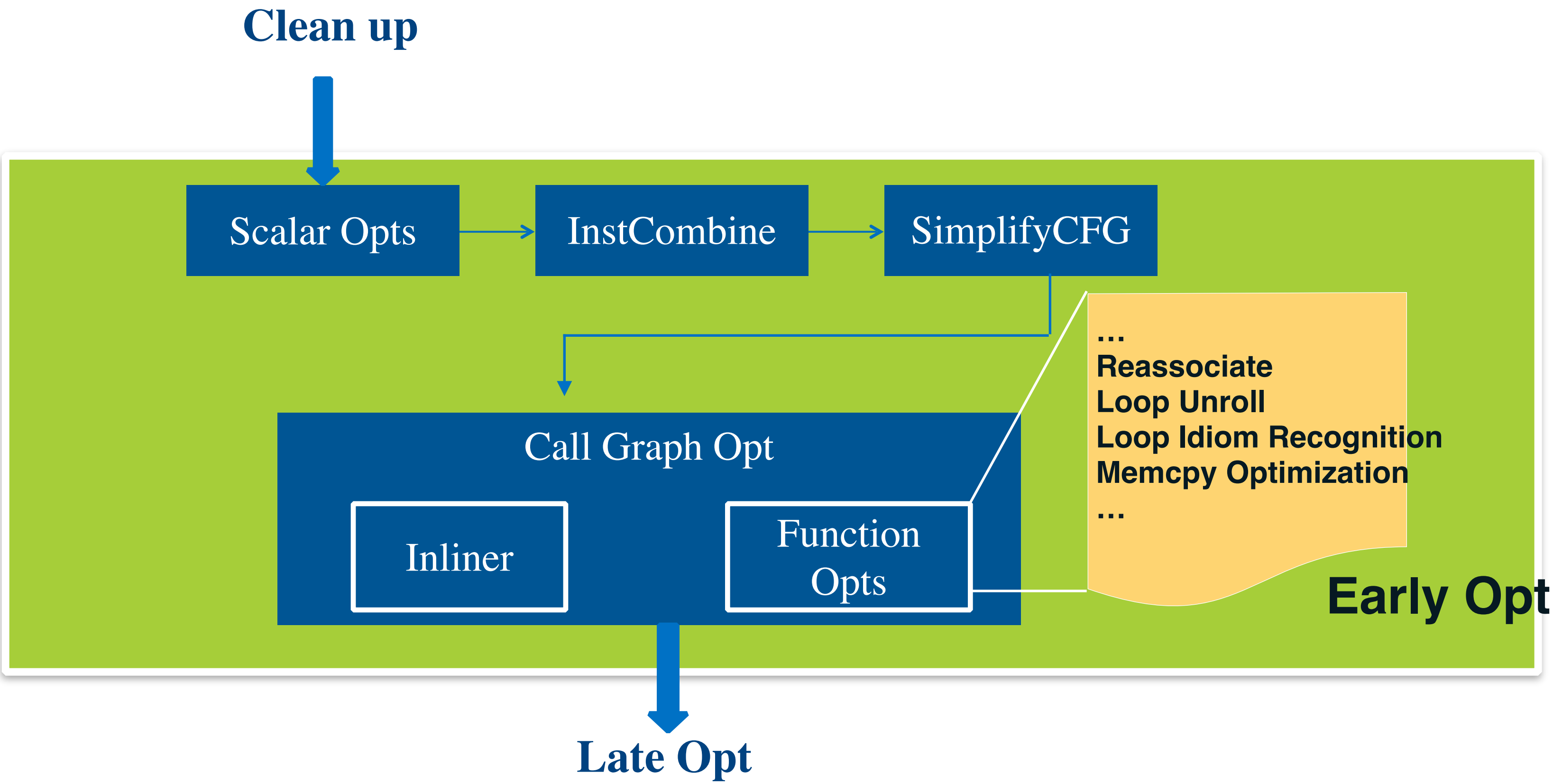
Mid End Optimizer – Early Opt



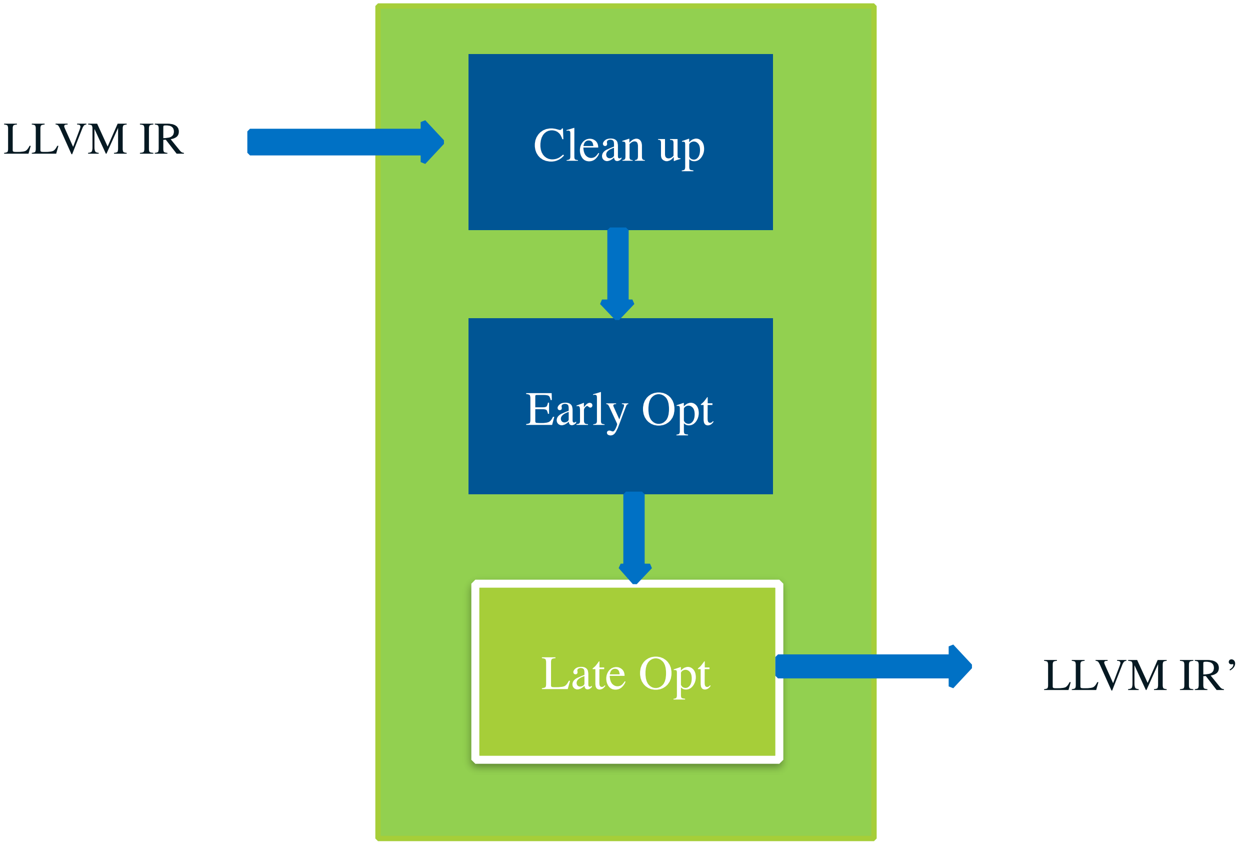
Mid End Optimizer – Early Opt



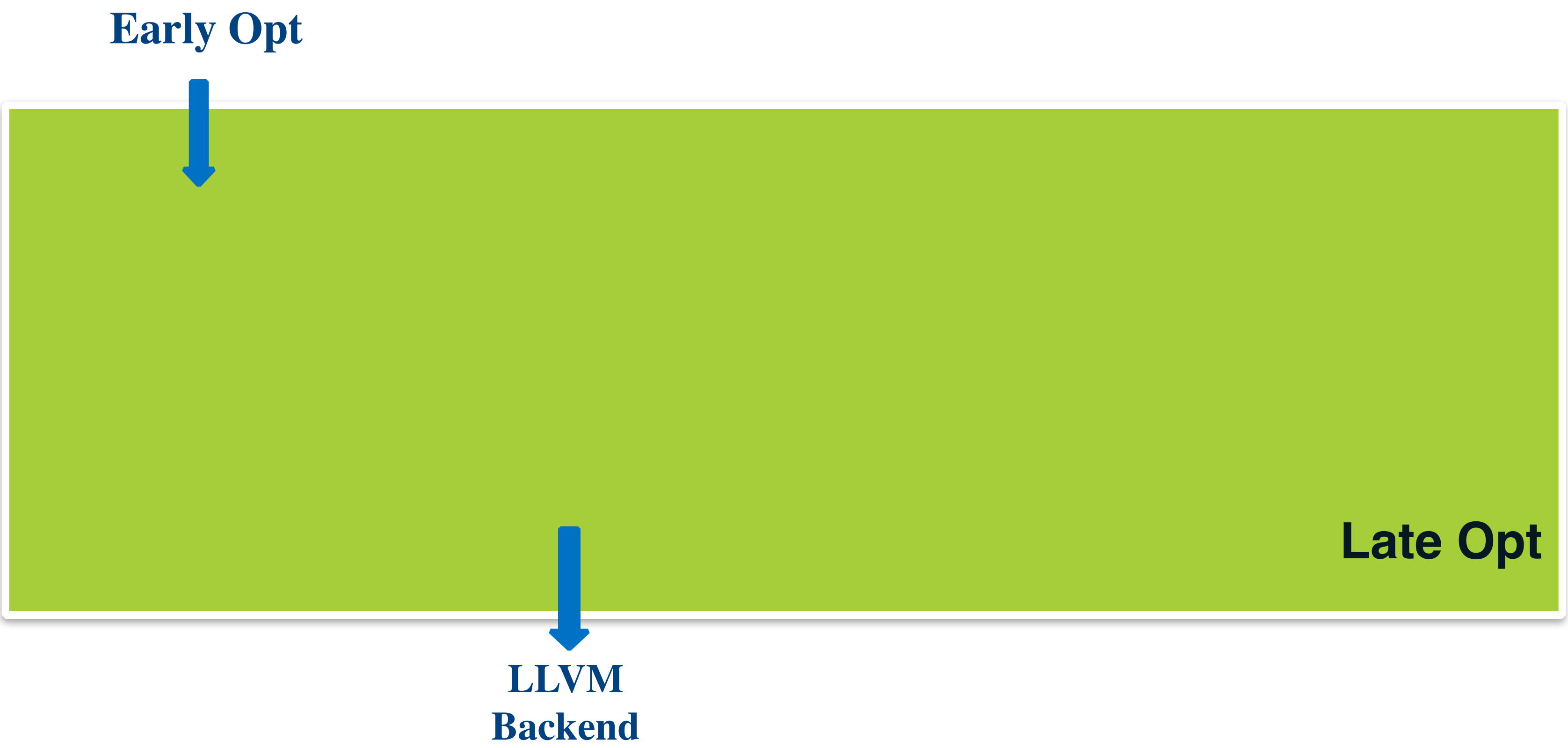
Mid End Optimizer – Early Opt



Mid End Optimizer – Late Opt



Mid End Optimizer – Late Opt

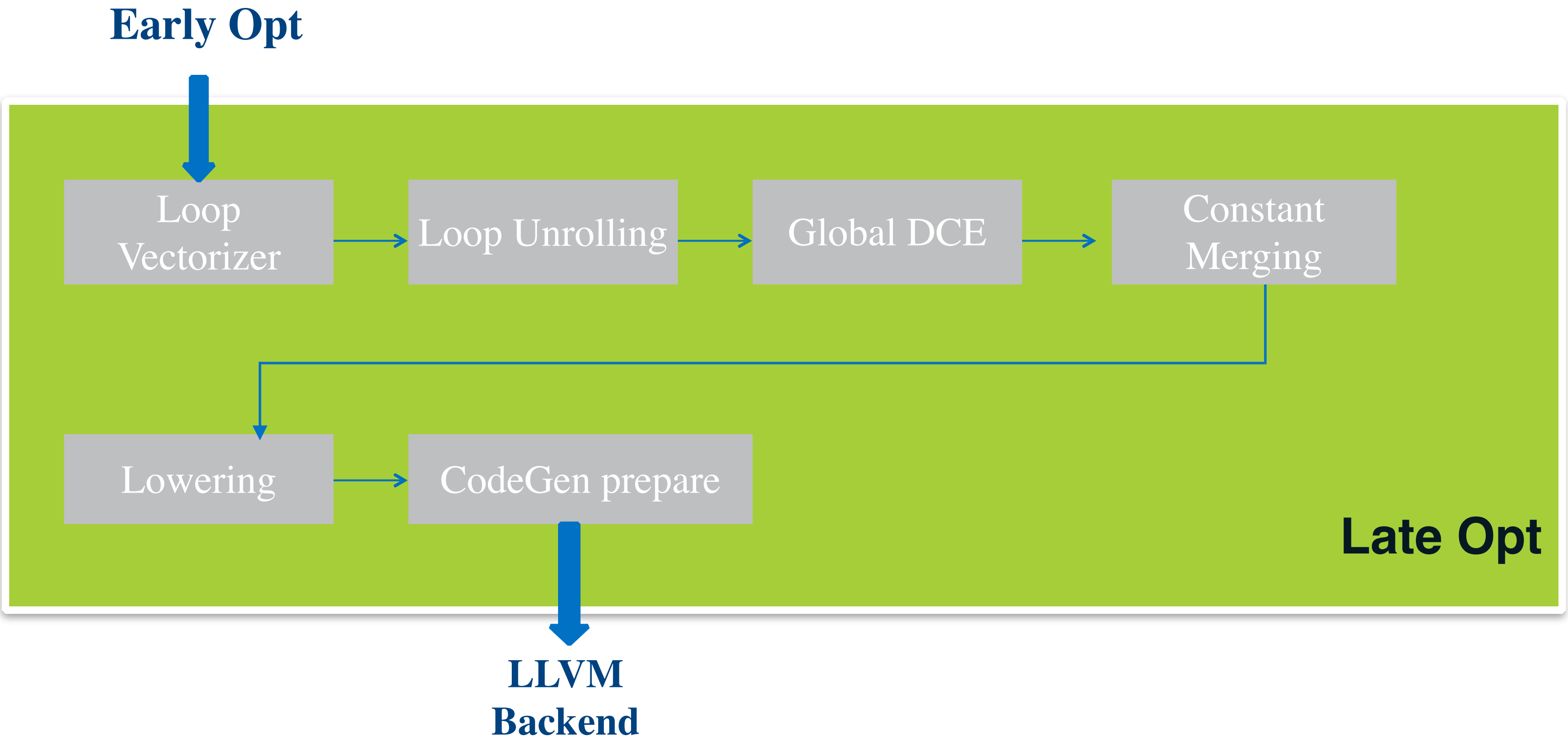


Early Opt

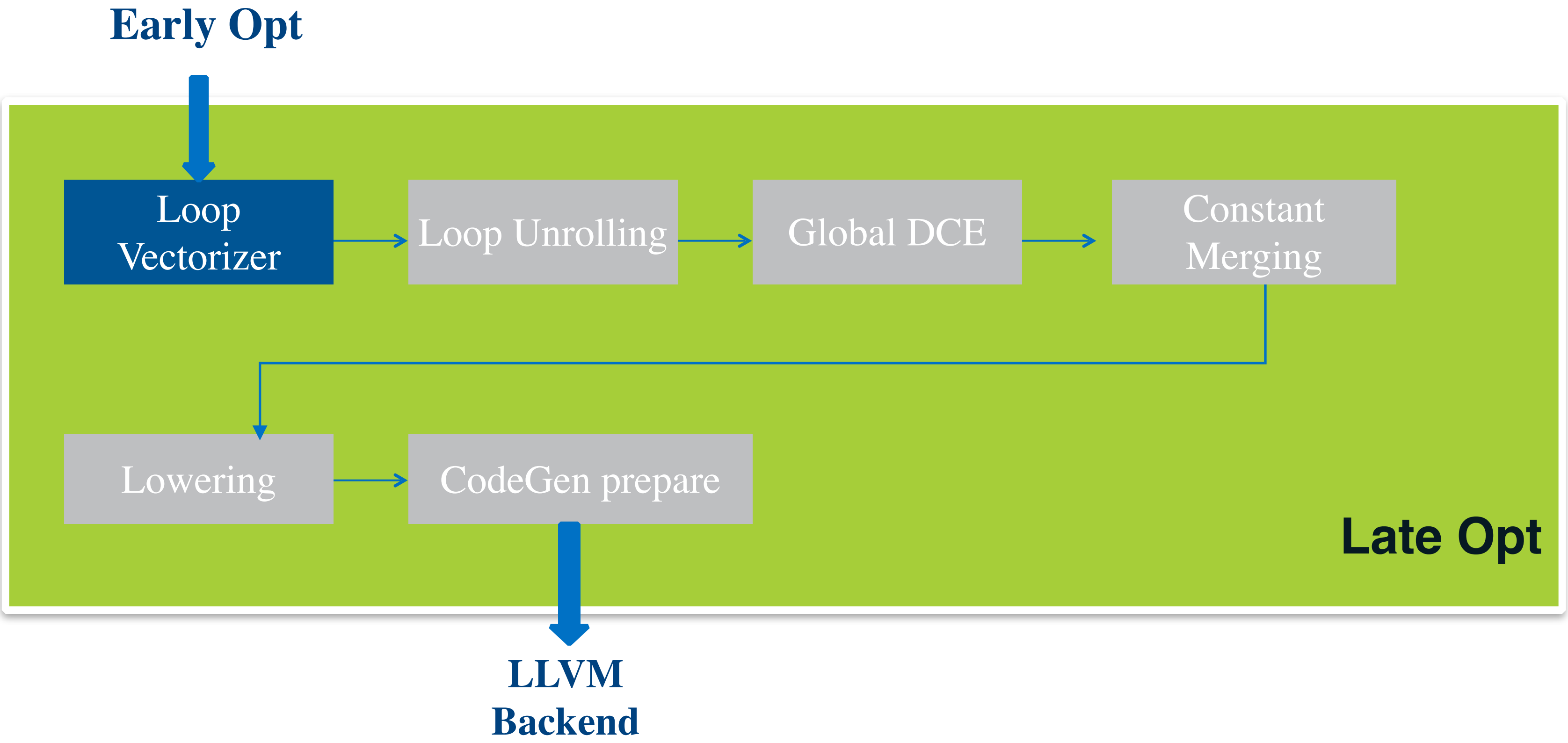
Late Opt

LLVM
Backend

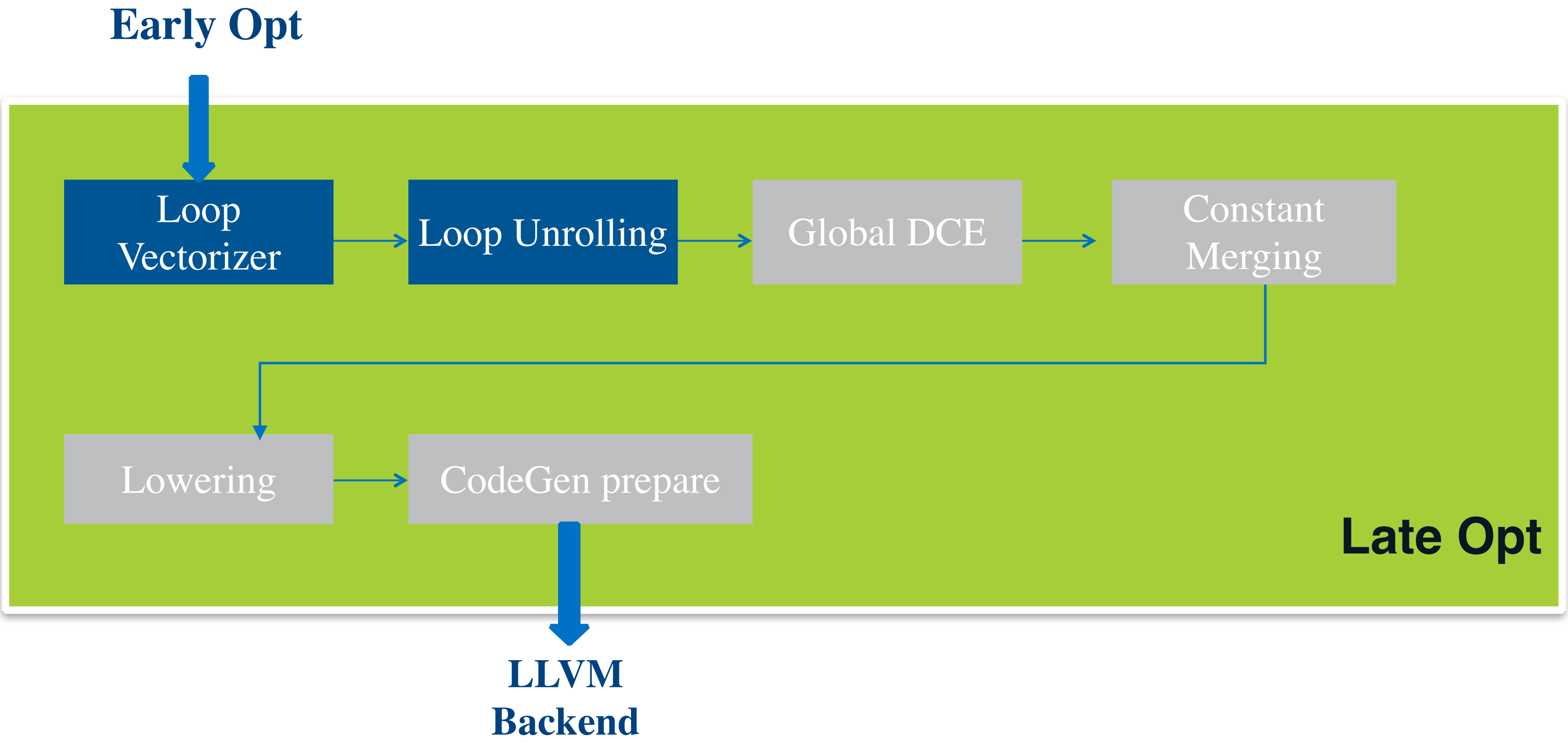
Mid End Optimizer – Late Opt



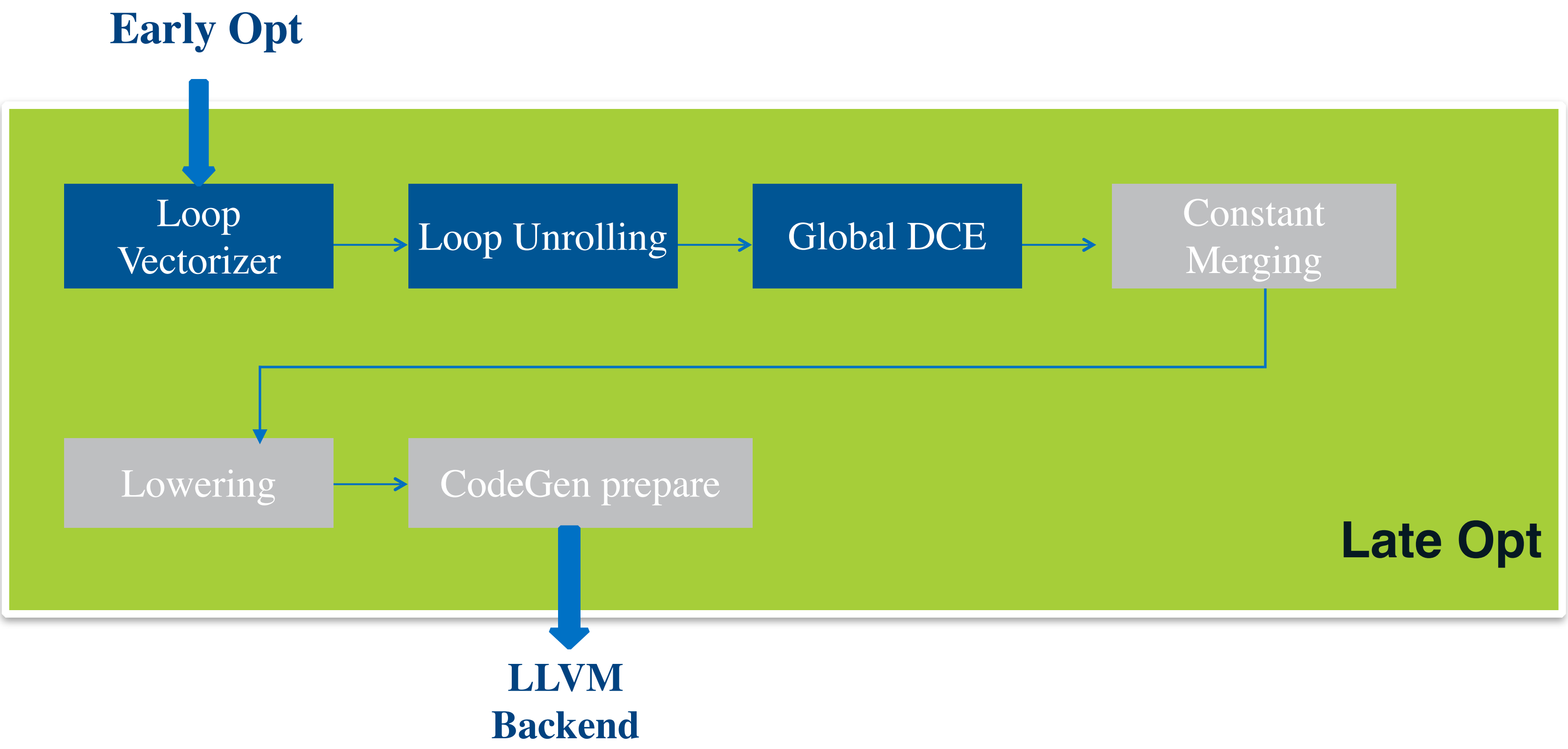
Mid End Optimizer – Late Opt



Mid End Optimizer – Late Opt



Mid End Optimizer – Late Opt



Early Opt

Loop Vectorizer

Loop Unrolling

Global DCE

Constant Merging

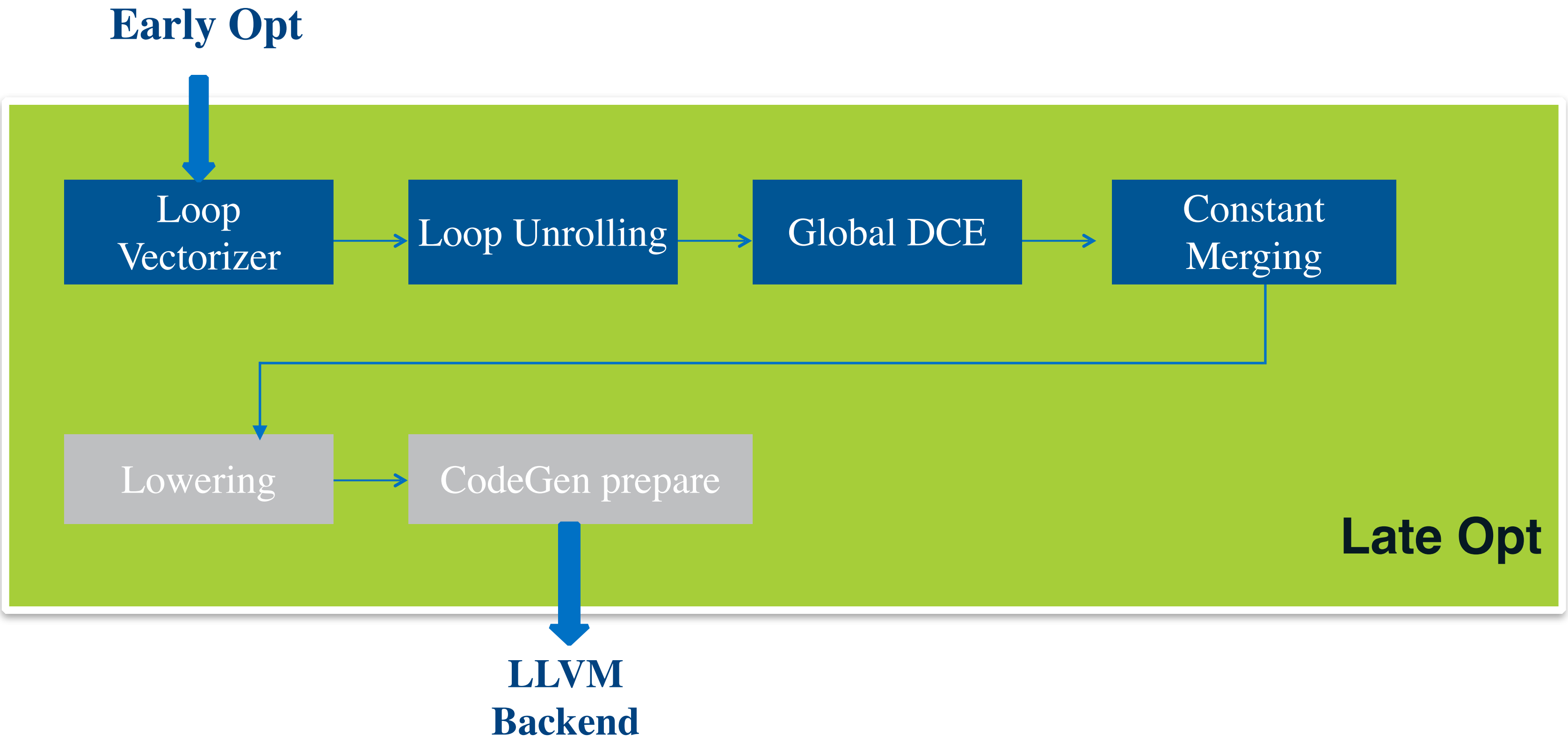
Lowering

CodeGen prepare

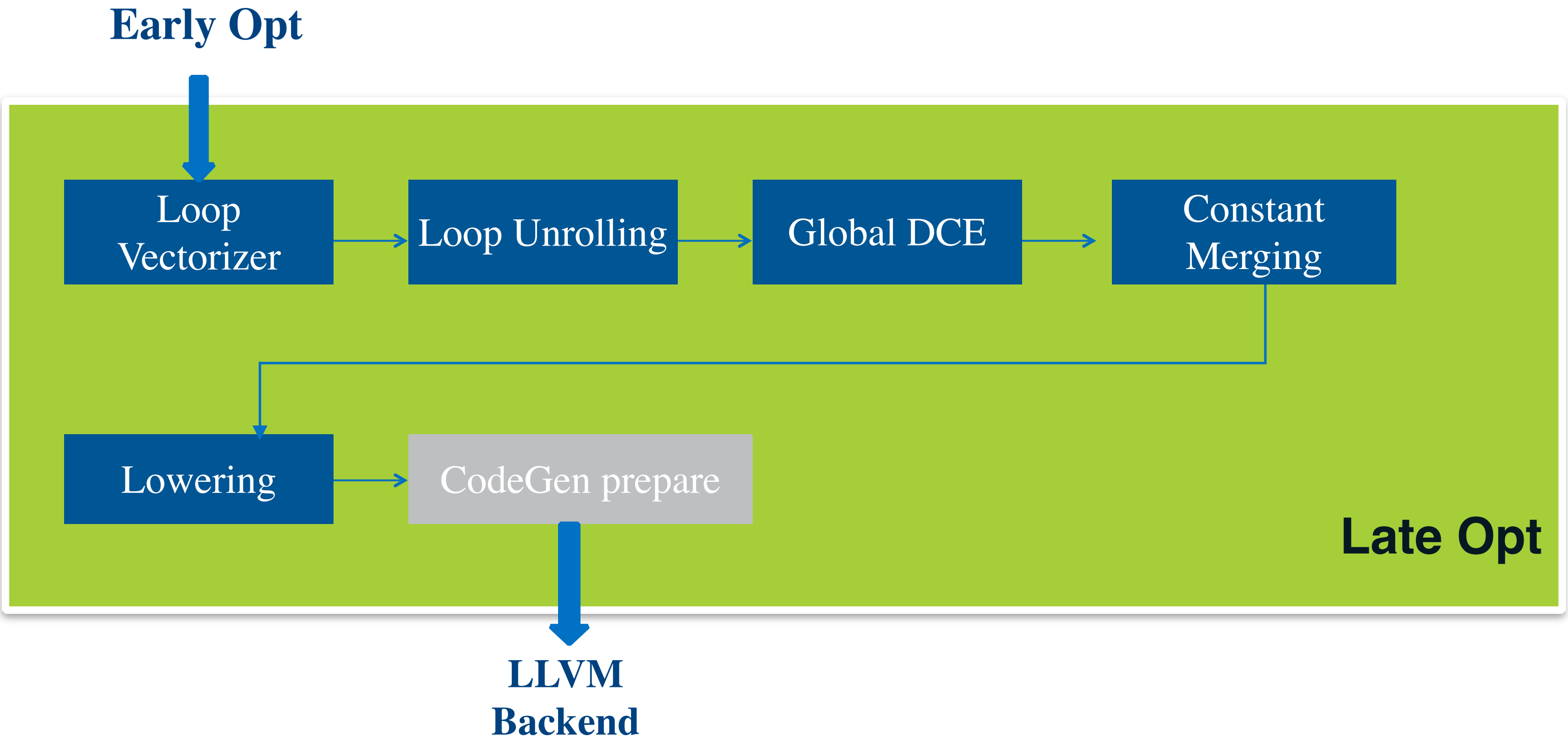
Late Opt

LLVM Backend

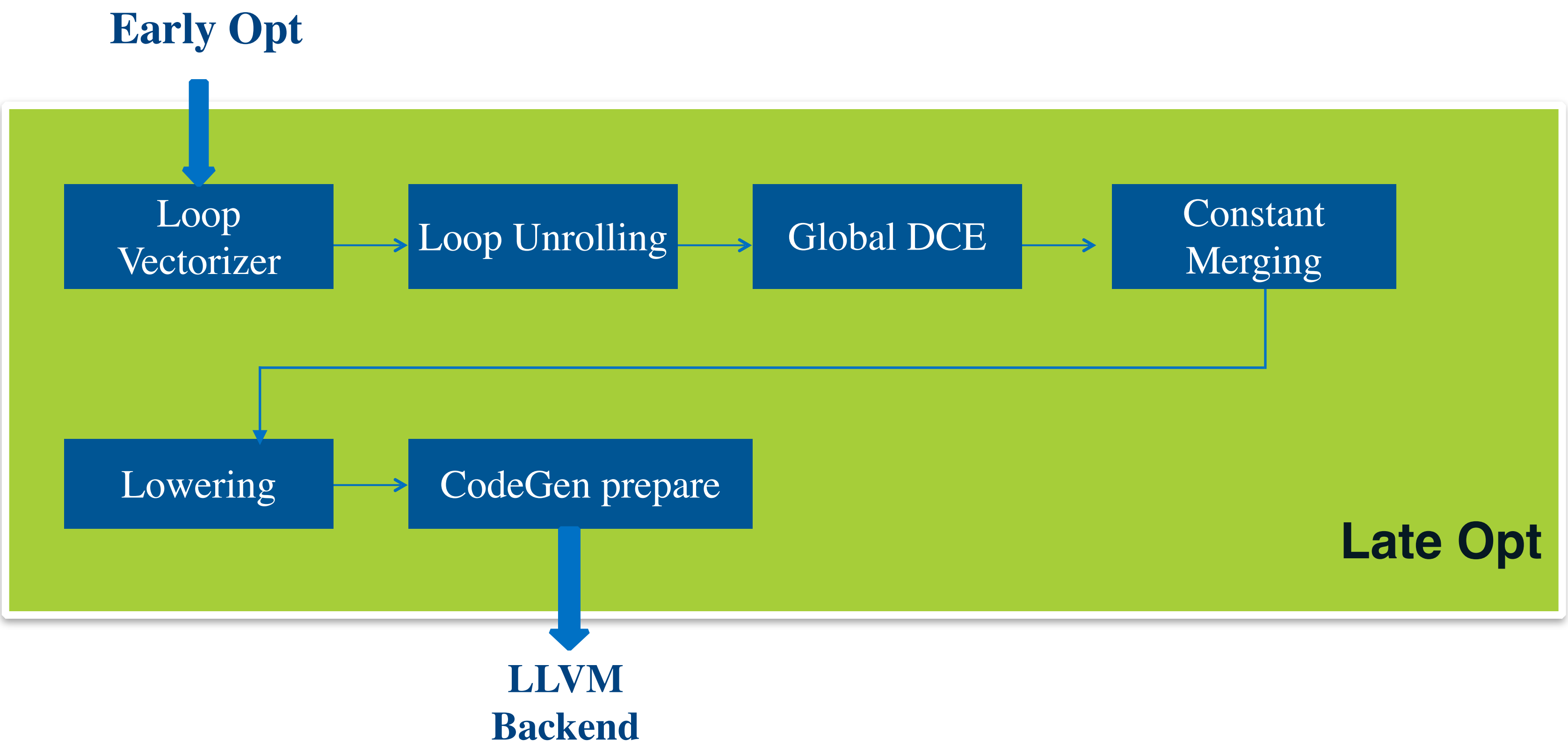
Mid End Optimizer – Late Opt



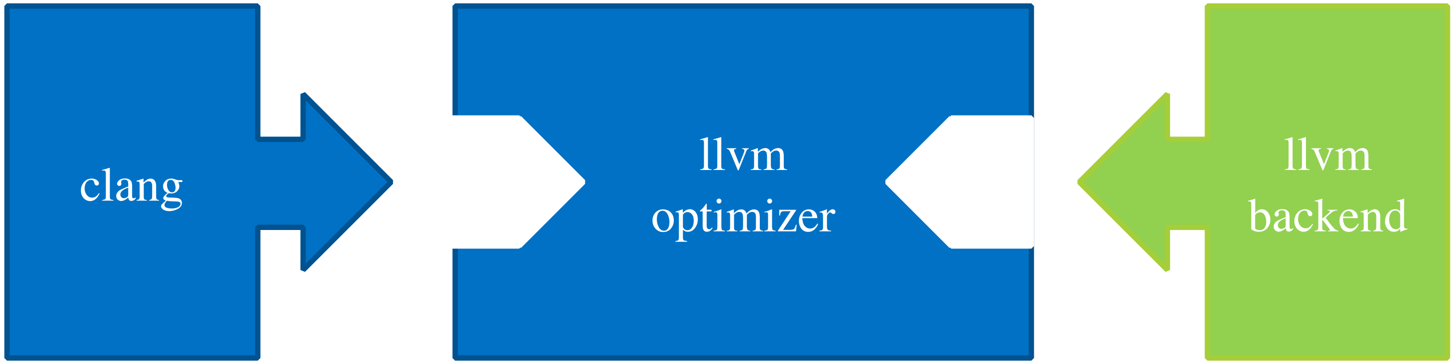
Mid End Optimizer – Late Opt



Mid End Optimizer – Late Opt



LLVM Pipeline



clang

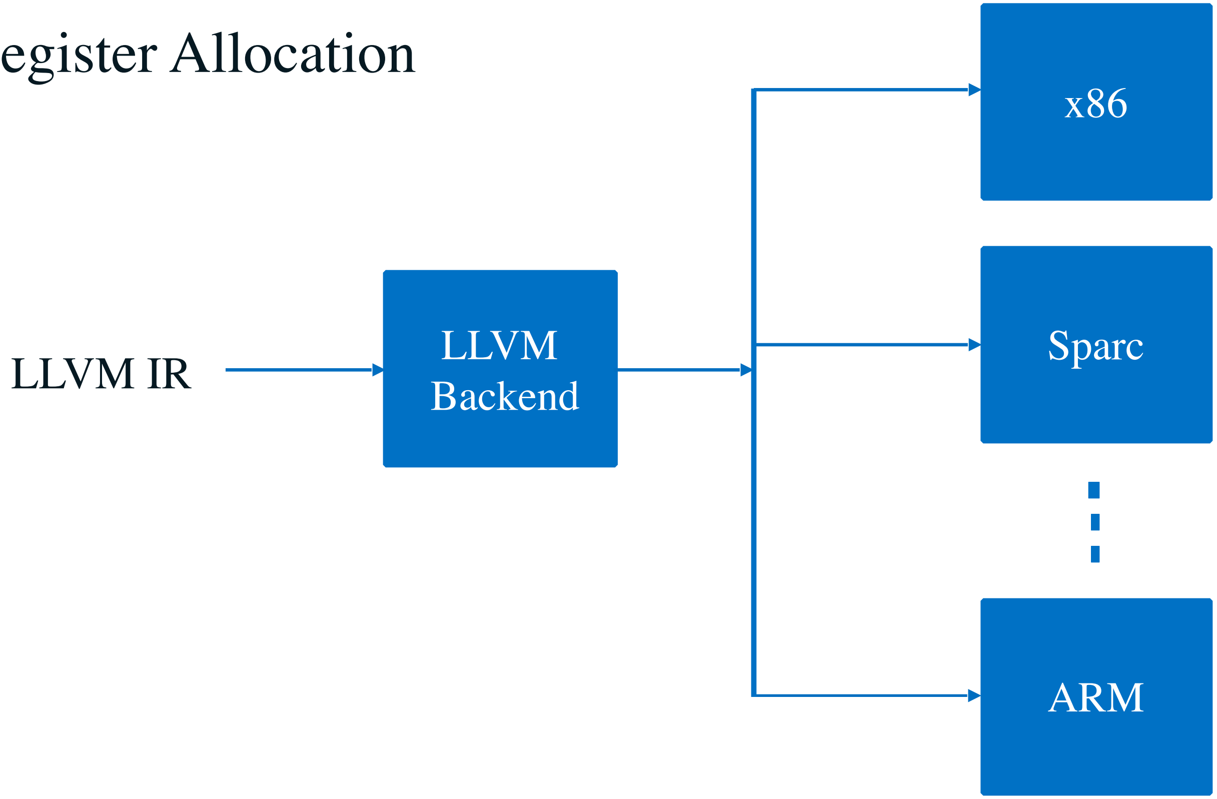
llvm optimizer

llvm backend

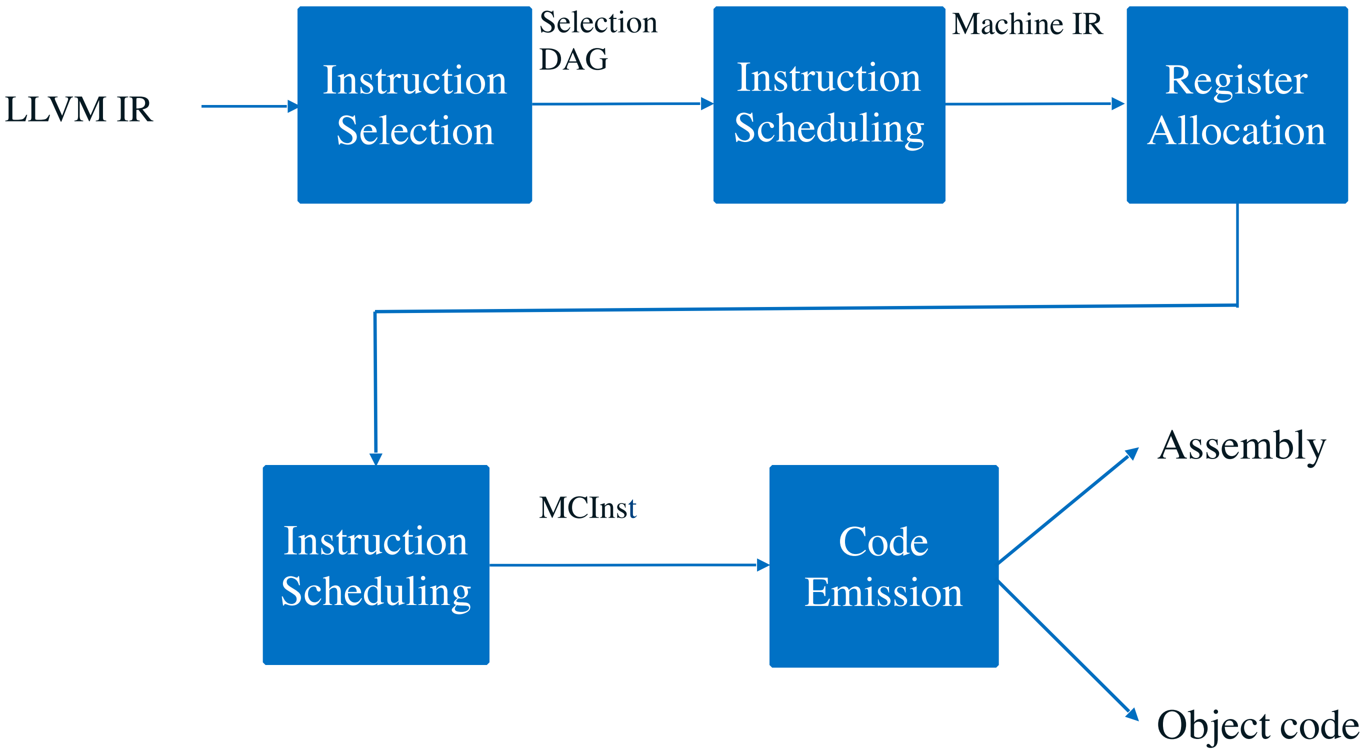
LLVM Backend

LLVM Backend

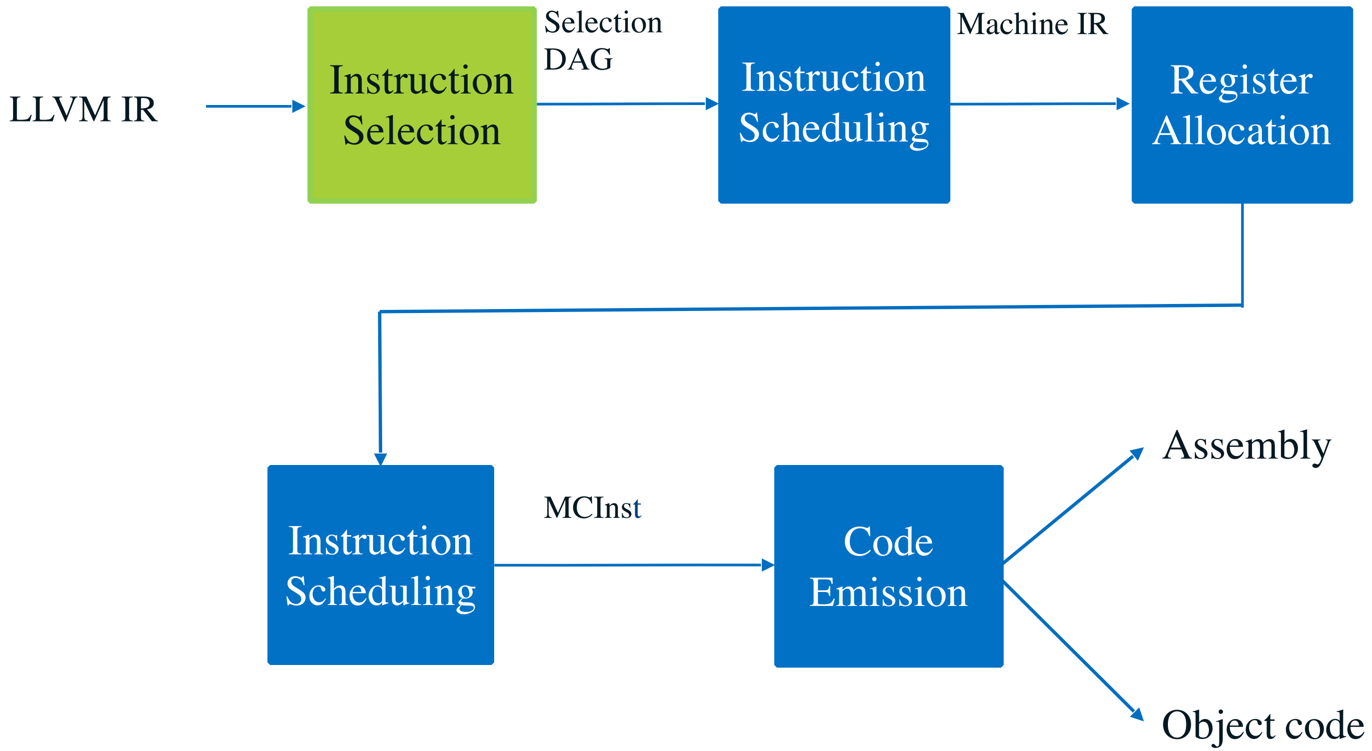
- Code generation for specific architectures via
  - Instruction Selection
  - Instruction Scheduling
  - Register Allocation



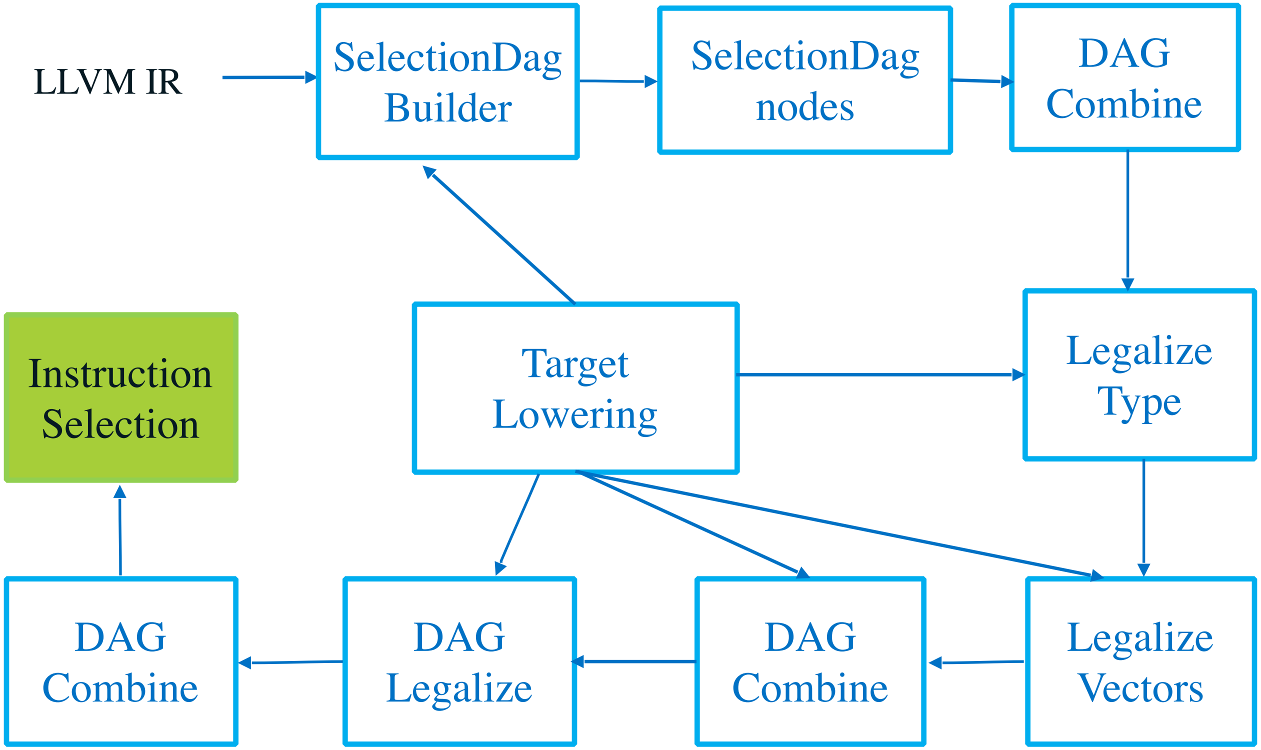
LLVM Backend Pipeline



LLVM Backend Pipeline



LLVM Backend - Instruction Selection Pipeline



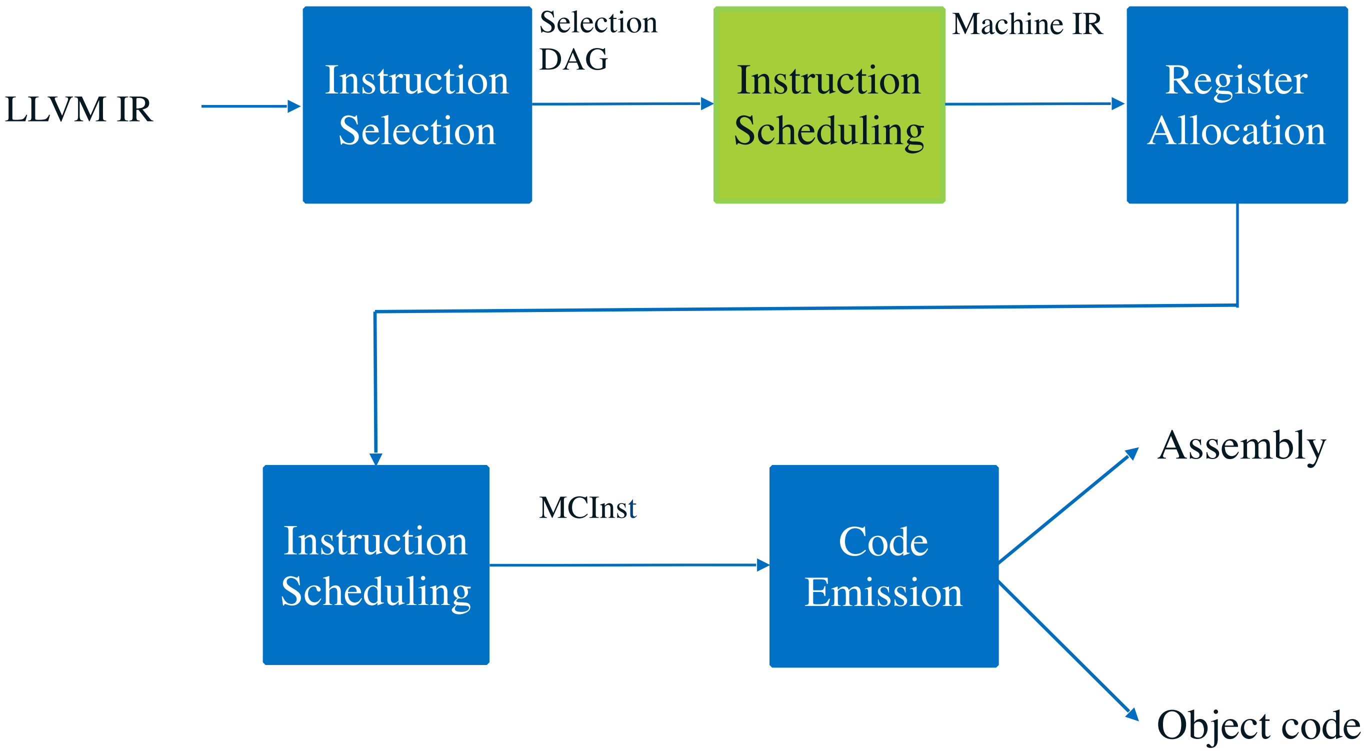
SelectionDAG Instruction Select Phase Example

```
%t1 = add i32 %W, %X
%t2 = mul i32 %t1, %Y
%t3 = add i32 %t2, %Z
```

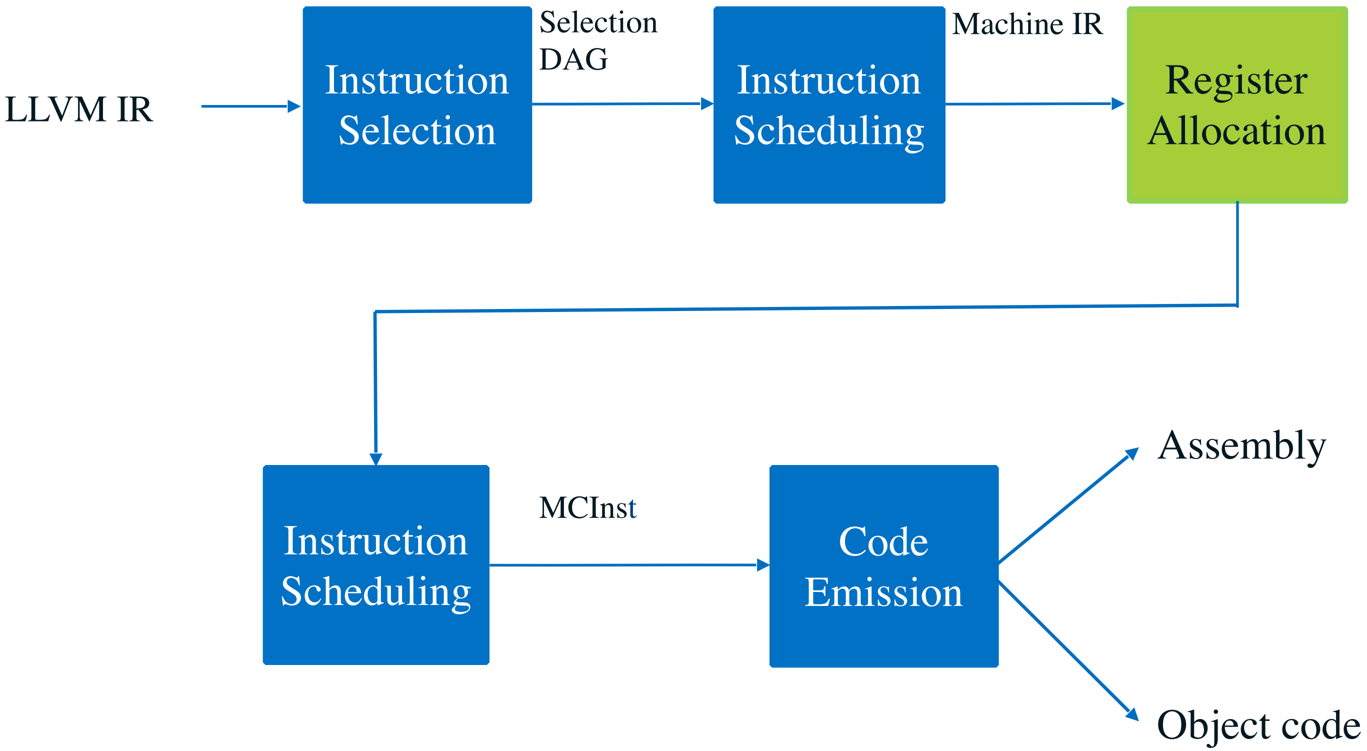
```
(add:i32 (mul:i32 (add:i32 W, X), Y), Z)
```

```
(MADDrr (ADDrr W, X), Y, Z)
```

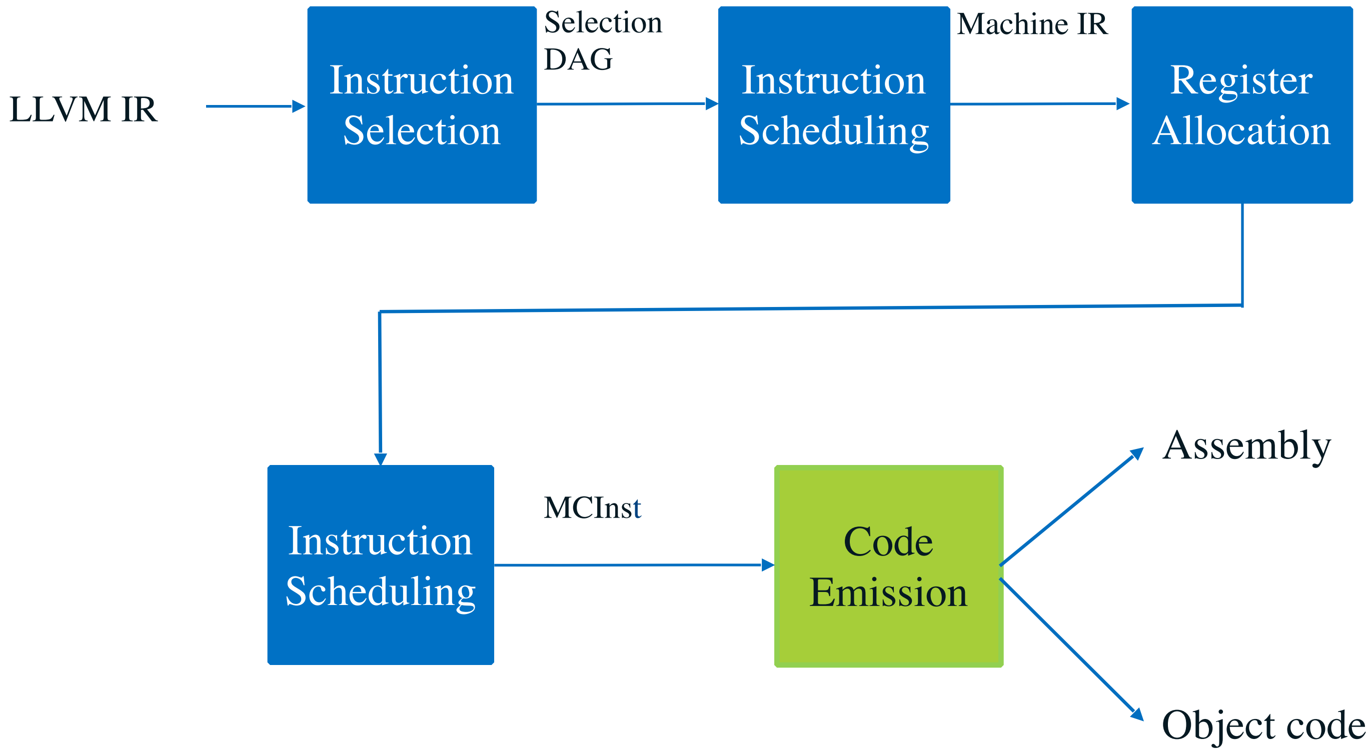
LLVM Backend Pipeline



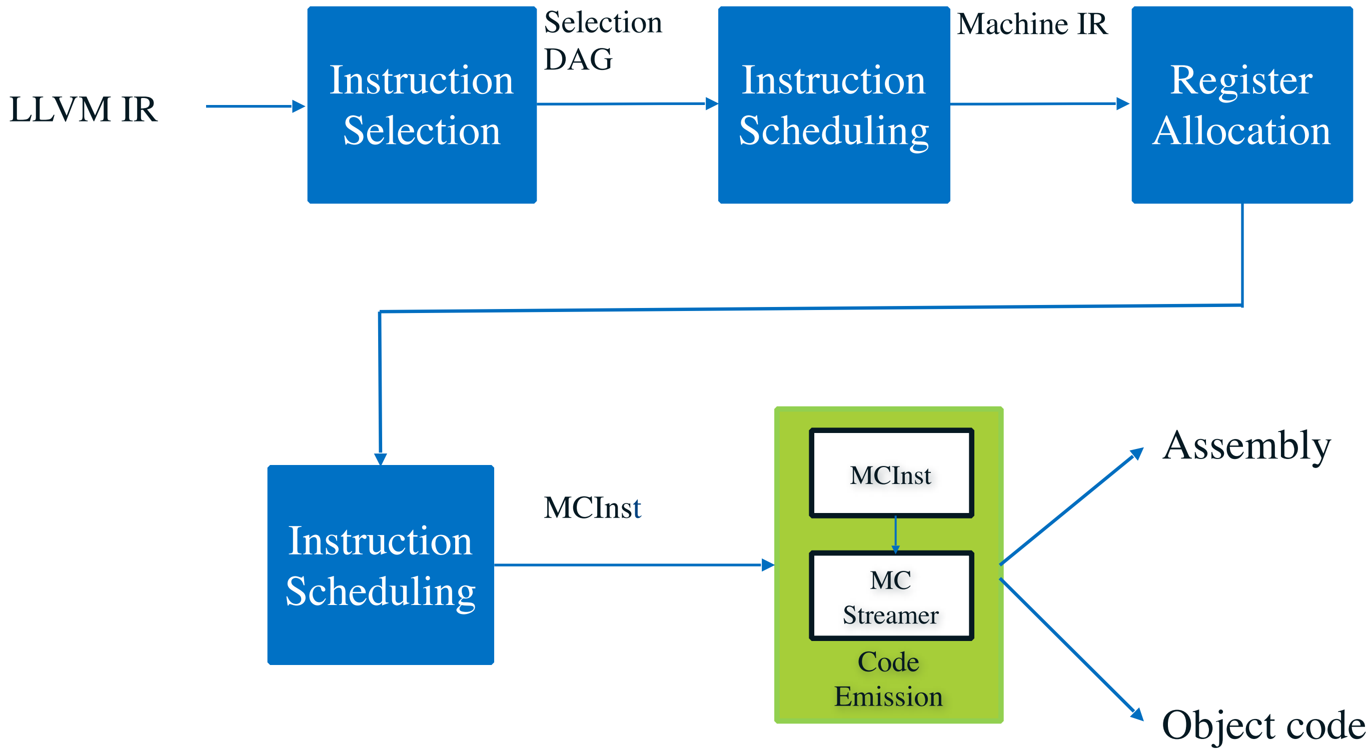
LLVM Backend Pipeline



LLVM Backend Pipeline



LLVM Backend Pipeline



Hands-on

Hands-On : Exercise 0

Build & install clang and llvm

- Objective
  - Learn how to download clang & llvm source from git

  https://github.com/jayakrish/llvm

  https://github.com/jayakrish/clang

  or

  https://github.com/dekasthiti/llvm

  https://github.com/dekasthiti/clang

  - Learn how to configure and build clang & llvm
  - https://github.com/jayakrish/Workshop-Materials
  - https://github.com/dekasthiti/Workshop-Materials

Hands-On : Exercise 1

Hello World with clang

- Objective
    - Difference between clang, gcc, if any
    - Show clang error diagnostics
    - To show how to compile with clang.
    - How to use the different LLVM tools

CLANG vs GCC

CLANG	GCC
ASTs and design are intended to be easily understandable	GCC has a very old codebase which presents a steep learning curve to new developers.
Clang is designed as an API from its inception, allowing it to be reused by source analysis tools, refactoring, IDEs (etc) as well as for code generation.	GCC is built as a monolithic static compiler, which makes it extremely difficult to use as an API and integrate into other tools.
Clang has none of these problems.	Various GCC design decisions make it very difficult to reuse: its build system is difficult to modify, you can't link multiple targets into one binary, you can't link multiple front-ends into one binary, it uses a custom garbage collector, uses global variables extensively, is not reentrant or multi-threadable, etc.
Clang does not implicitly simplify code as it parses it like GCC does. Doing so causes many problems for source analysis tools	If you write "x-x" in your source code, the GCC AST will contain "0", with no mention of 'x'. This is extremely bad for a refactoring tool that wants to rename 'x'

CLANG vs GCC

CLANG	GCC
Clang can serialize its AST out to disk and read it back into another program, which is useful for whole program analysis.	*GCC does not have this.*
Clang is much faster in compile time and uses far less memory than GCC.	
Clang aims to provide extremely clear and concise diagnostics (error and warning messages), and includes support for expressive diagnostics.	*GCC's warnings are sometimes acceptable, but are often confusing and it does not support expressive diagnostics.*
Clang uses a BSD license, which allows it to be embedded in software that is not GPL-licensed.	*GCC is licensed under the GPL license*

LLVM IR Explained

MODULE – top level structure

global variables

functions

Basic Blocks – containers for instructions

library references

symbol-table

target characteristics

target datalayout: <datatype><size>:<abi>:<pref>

target triple:

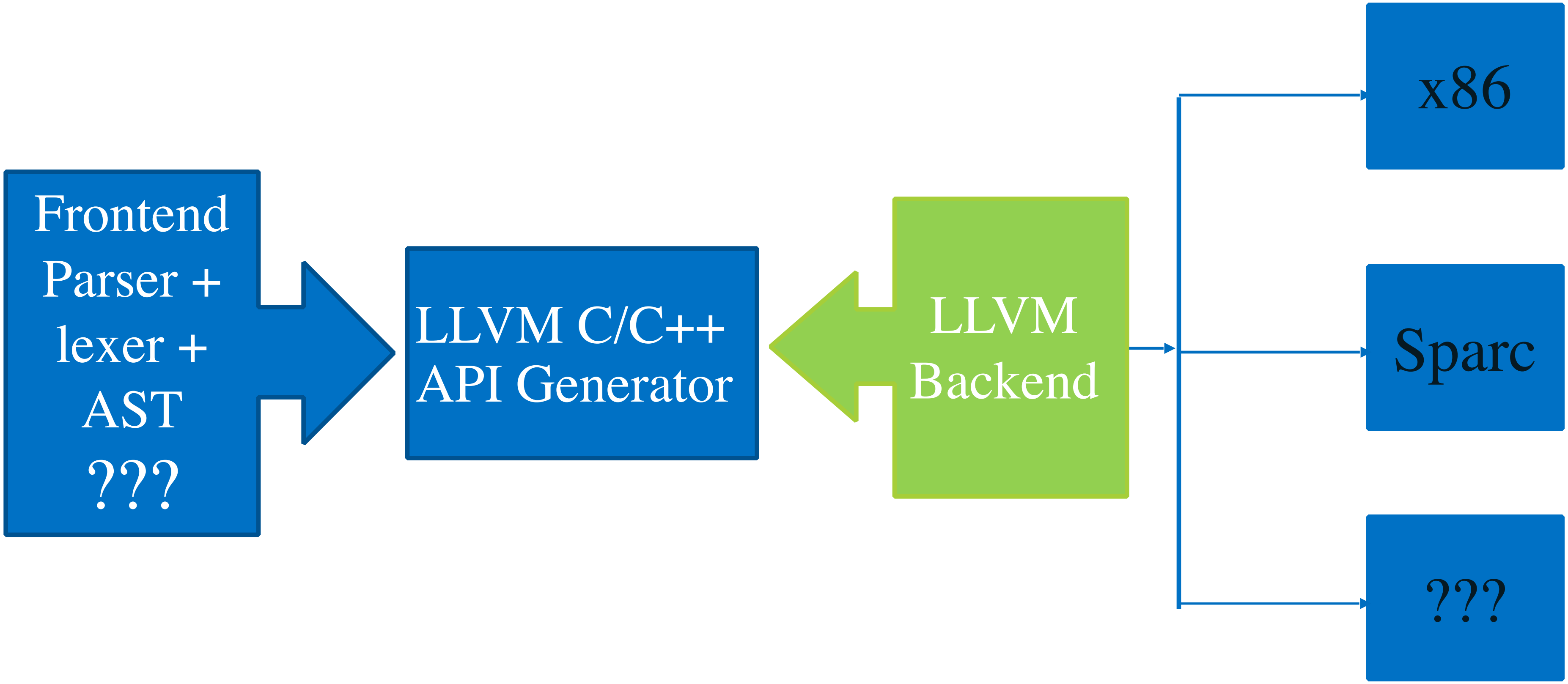
<Architecture> - <sub> - <os> - <vendoe> - <abi>

Hands-On: Exercise 2

Generate LLVM IR using LLVM API

- Objective
  - Learn how to generate LLVM IR with LLVM API
  - Introduce LLVM concepts of Module, IRBuilder, Verifier, ExecutionEngine, PassManager
  - Explain how the LLVM API generator can be used to build a custom compiler.

Build Your Custom Compiler

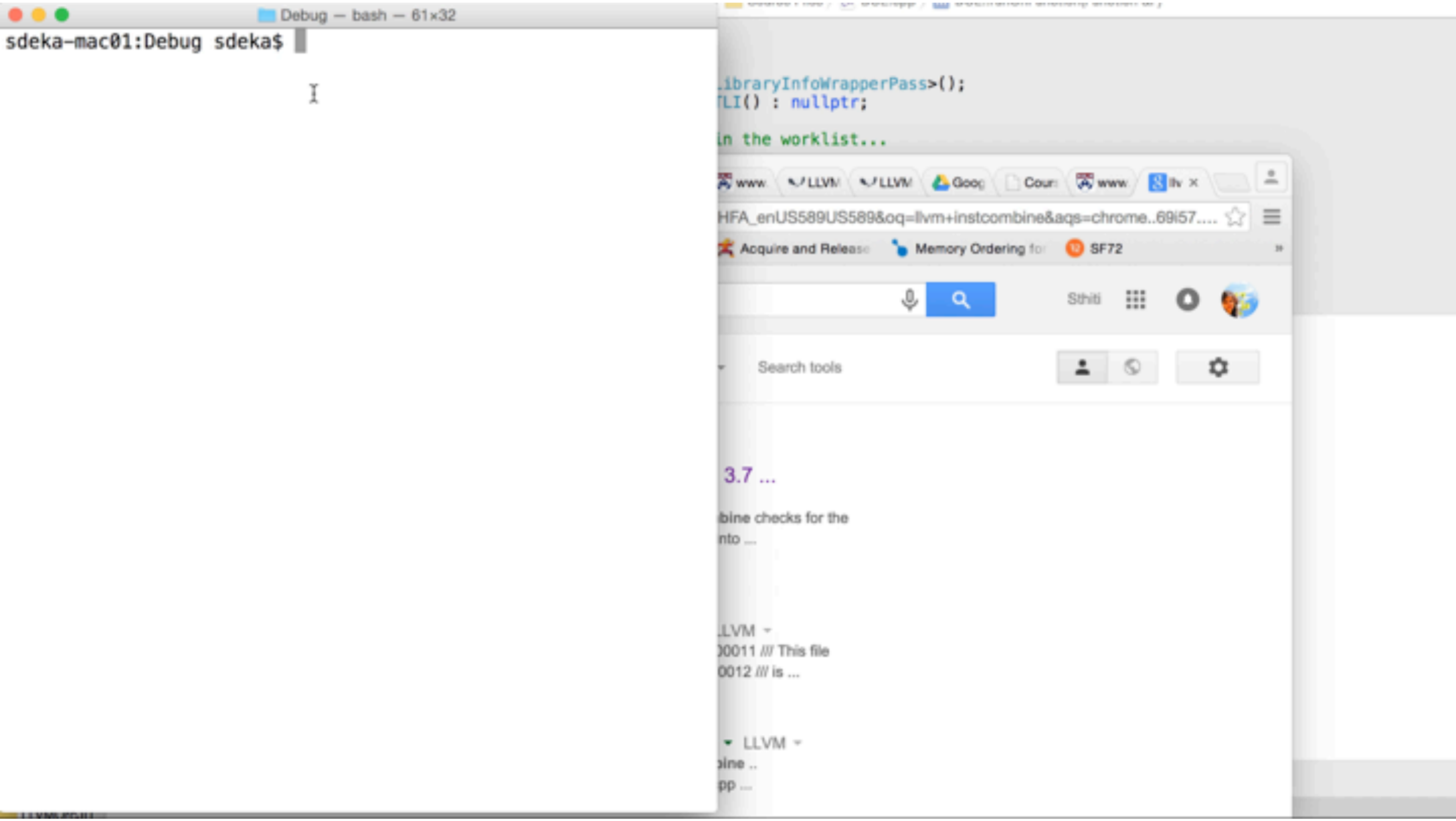


Hands-On: Exercise 3

Print and view state and flow in a program

- Objective
  - Learn how to use LLVM tool opt, to view control flow graph
  - Learn how to use LLVM tool opt, to view call graph

Hands-On: Exercise 3



```
sdeka-mac01:Debug sdeka$
```

```
ibraryInfoWrapperPass>();
TLI() : nullptr;

in the worklist...
```

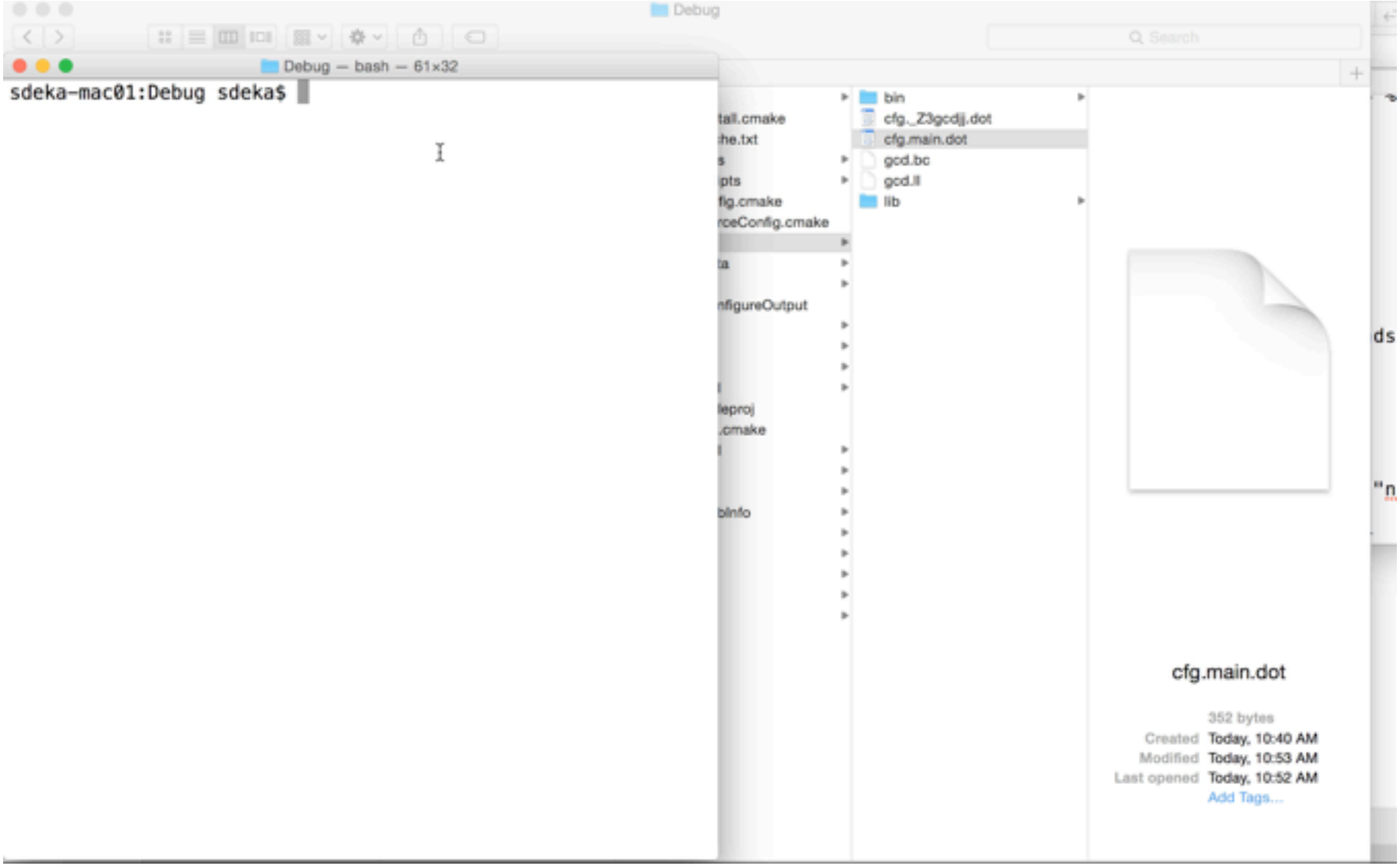
3.7 ...

bine checks for the
nto ...

LLVM
0011 /// This file
0012 /// is ...

LLVM
bine ..
pp ...

Hands-On: Exercise 3



```
sdeka-mac01:Debug sdeka$ 
```

cfg.main.dot

352 bytes

Created Today, 10:40 AM

Modified Today, 10:53 AM

Last opened Today, 10:52 AM

Add Tags...

Hands-On: Exercise 4

Write a pass using LLVM API

- Objective
  - Learn how to write a pass
  - Learn about LLVM pass types
  - Learn about LLVM data structures such as iterators, dense maps
  - Learn about the tools, commands and headers used to do this exercise

Hands-On: Exercise 4

Pass in LLVM

- Pass Hierarchy

Hands-On: Exercise 4

Pass in LLVM

- Pass Hierarchy
  - Module Pass

Hands-On: Exercise 4

Pass in LLVM

- Pass Hierarchy
  - Module Pass
  - Function Pass

Hands-On: Exercise 4

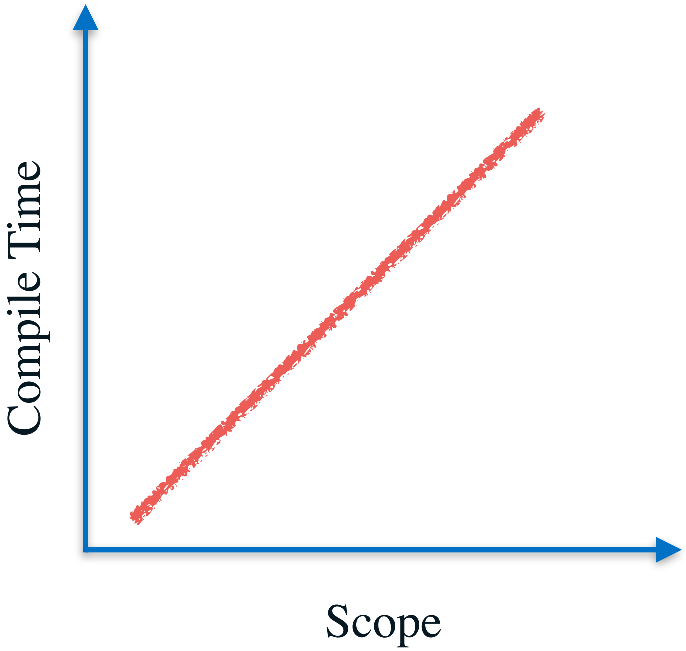
Pass in LLVM

- Pass Hierarchy
  - Module Pass
  - Function Pass
  - Basic Block Pass

Hands-On: Exercise 4

Pass in LLVM

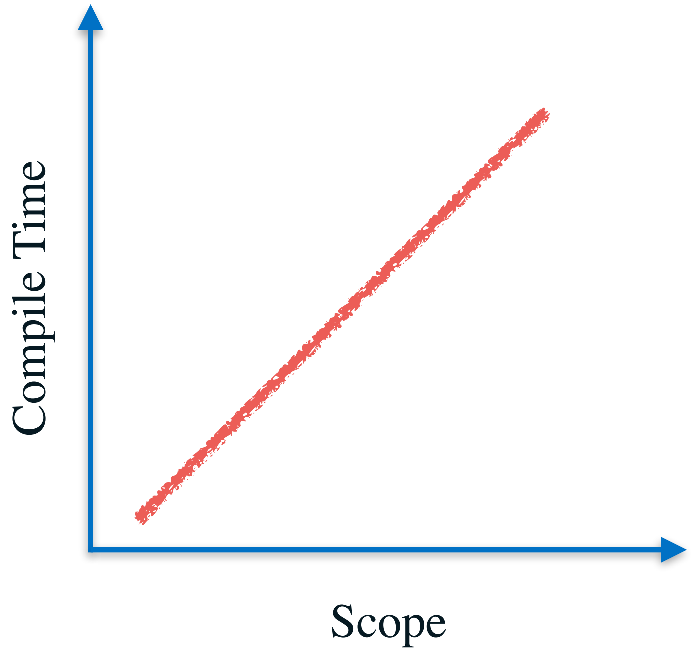
- Pass Hierarchy
  - Module Pass
  - Function Pass
  - Basic Block Pass



Hands-On: Exercise 4

Pass in LLVM

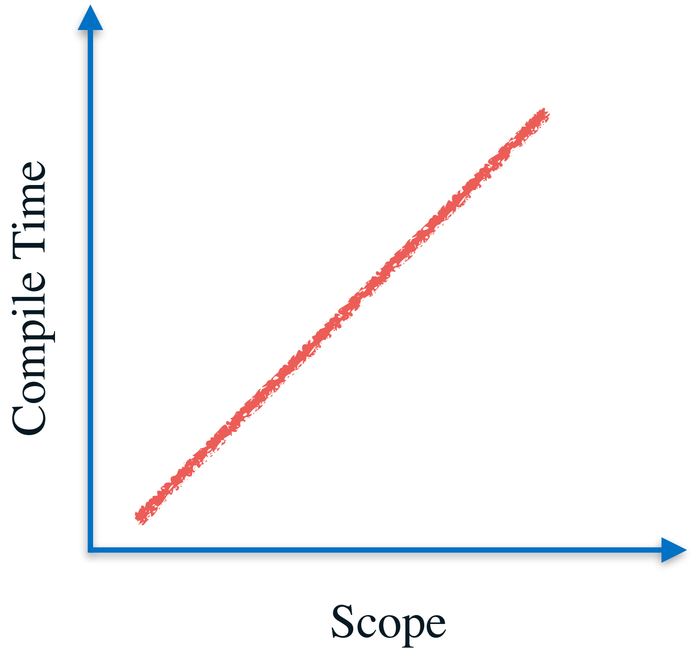
- Pass Examples



Hands-On: Exercise 4

Pass in LLVM

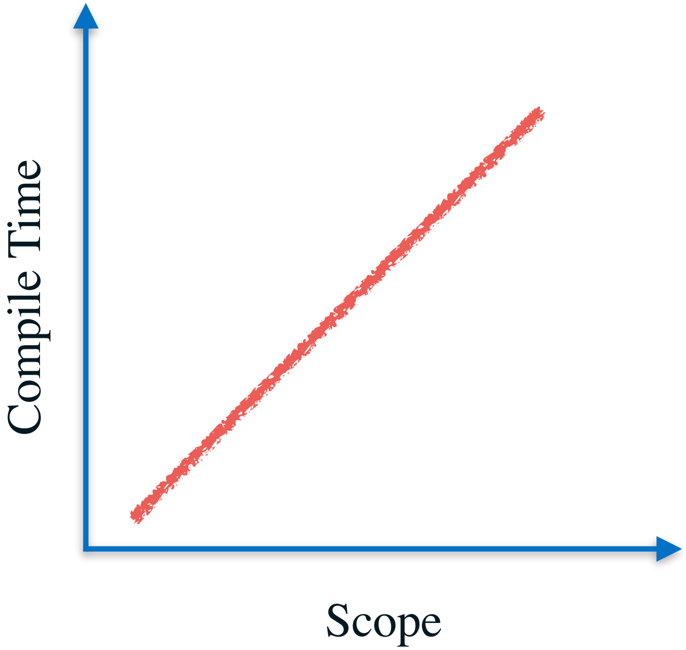
- Pass Examples
  - Module Pass
    - Eg: Call Graph



Hands-On: Exercise 4

Pass in LLVM

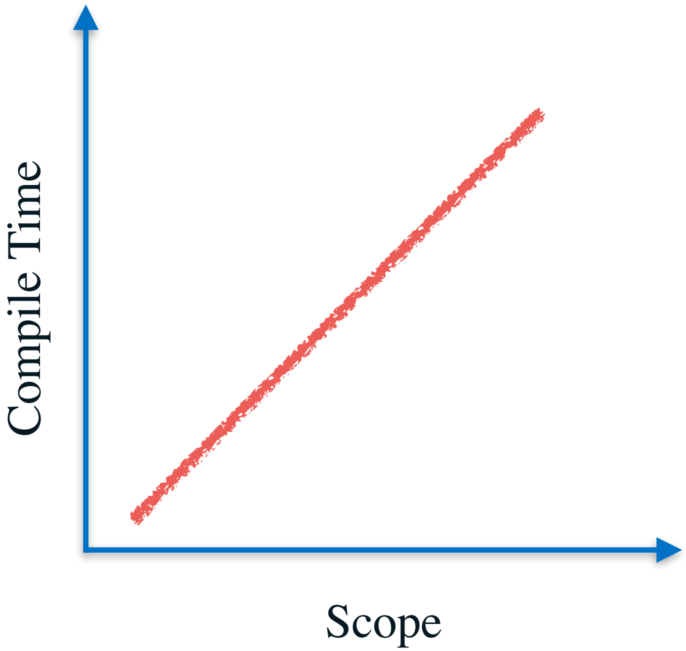
- Pass Examples
  - Module Pass
    - Eg: Call Graph
  - Function Pass
    - Eg: Constant Propagation



Hands-On: Exercise 4

Pass in LLVM

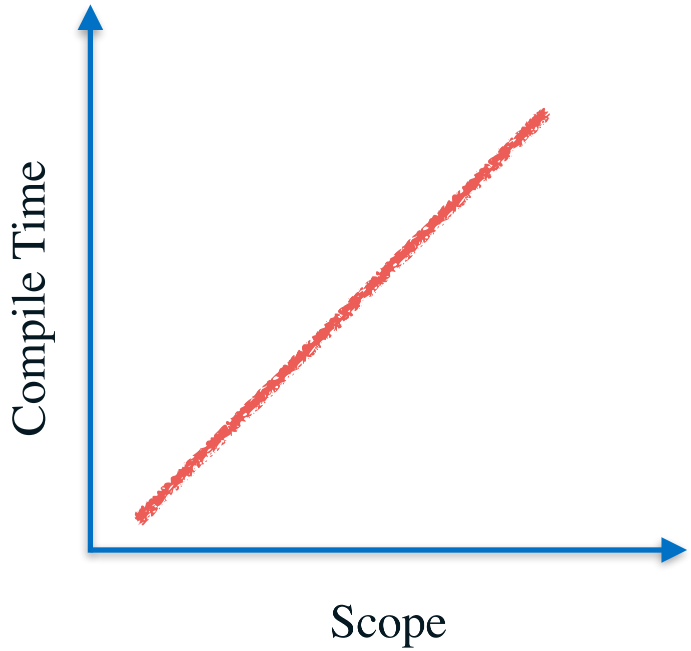
- Pass Examples
  - Module Pass
    - Eg: Call Graph
  - Function Pass
    - Eg: Constant Propagation
  - Basic Block Pass
    - Eg: Dead Instruction Elimination



Hands-On: Exercise 4

Pass in LLVM

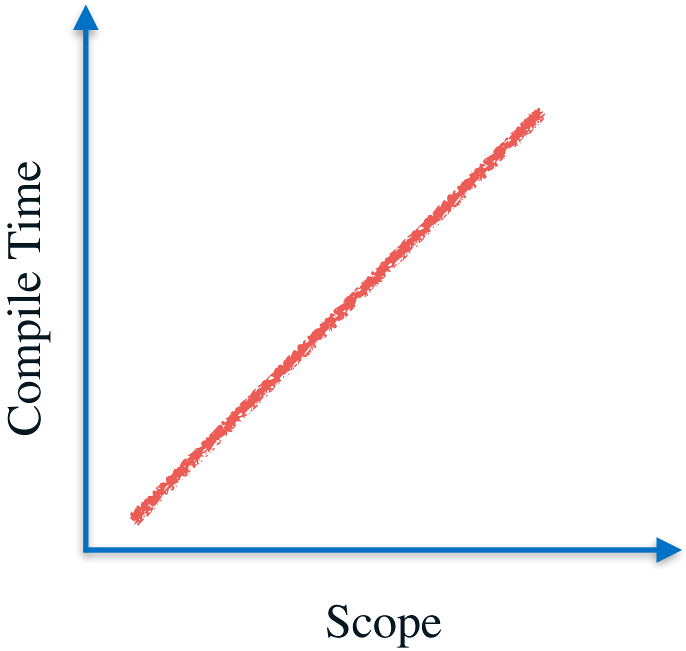
- Pass Interfaces



Hands-On: Exercise 4

Pass in LLVM

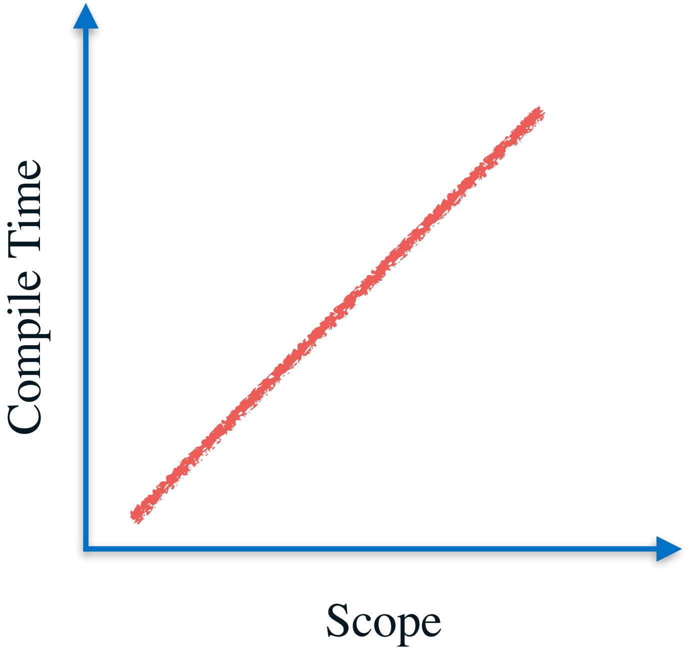
- Pass Interfaces
  - Module Pass::
    - runOnModule(Module &M)



Hands-On: Exercise 4

Pass in LLVM

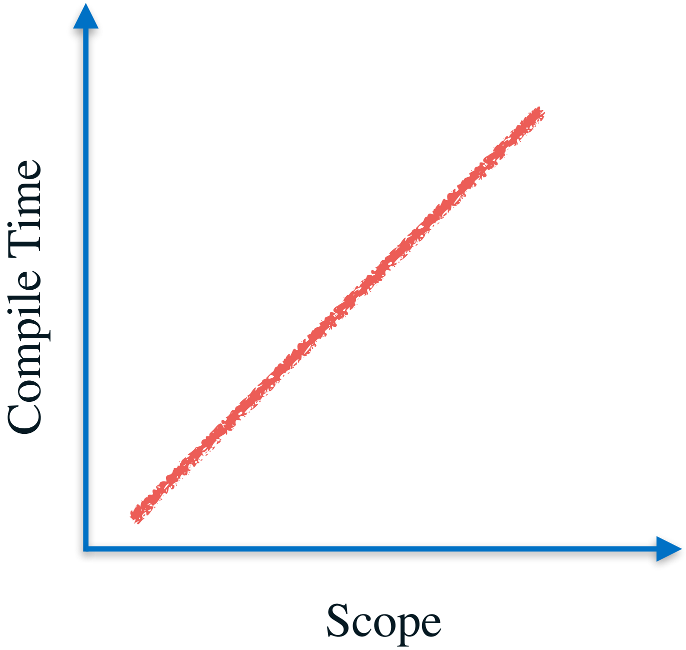
- Pass Interfaces
  - Module Pass::
    - runOnModule(Module &M)
  - Function Pass ::
    - runOnFunction(Function &F)



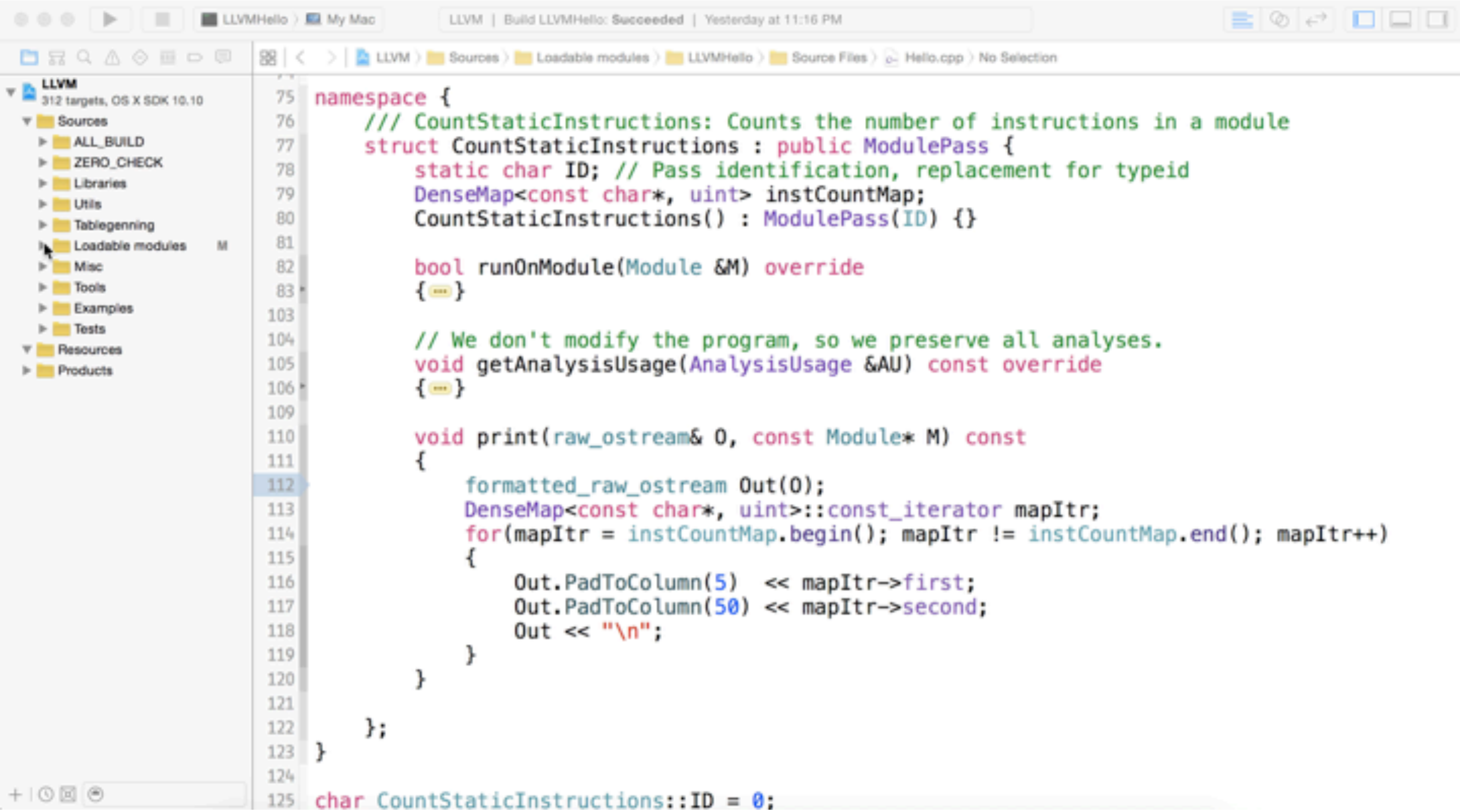
Hands-On: Exercise 4

Pass in LLVM

- Pass Interfaces
  - Module Pass::
    - runOnModule(Module &M)
  - Function Pass ::
    - runOnFunction(Function &F)
  - Basic Block Pass ::
    - runOnBasicBlock(BasicBlock &B)

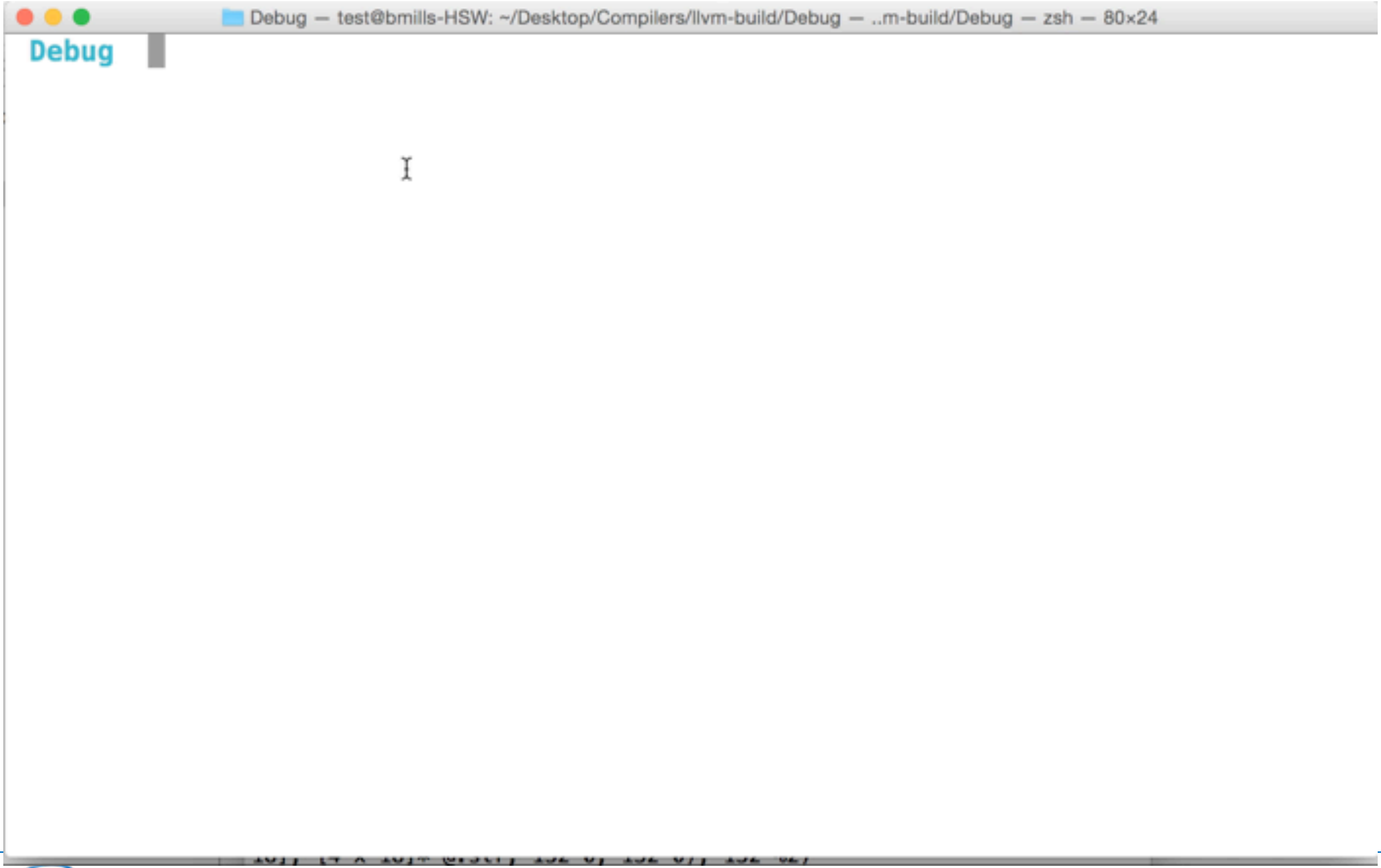


Hands-On: Exercise 4 - Hello Pass


LLVMHello 〉 My Mac
LLVM | Build LLVMHello: Succeeded | Yesterday at 11:16 PM
LLVM 〉 Sources 〉 Loadable modules 〉 LLVMHello 〉 Source Files 〉 Hello.cpp 〉 No Selection
LLVM
312 targets, OS X SDK 10.10
Sources
ALL_BUILD
ZERO_CHECK
Libraries
Utils
Tablegenning
Loadable modules
M
Misc
Tools
Examples
Tests
Resources
Products

```cpp
namespace {
    /// CountStaticInstructions: Counts the number of instructions in a module
    struct CountStaticInstructions : public ModulePass {
        static char ID; // Pass identification, replacement for typeid
        DenseMap<const char*, uint> instCountMap;
        CountStaticInstructions() : ModulePass(ID) {}

        bool runOnModule(Module &M) override
        {⋯}

        // We don't modify the program, so we preserve all analyses.
        void getAnalysisUsage(AnalysisUsage &AU) const override
        {⋯}

        void print(raw_ostream& O, const Module* M) const
        {
            formatted_raw_ostream Out(O);
            DenseMap<const char*, uint>::const_iterator mapItr;
            for(mapItr = instCountMap.begin(); mapItr != instCountMap.end(); mapItr++)
            {
                Out.PadToColumn(5)  << mapItr->first;
                Out.PadToColumn(50) << mapItr->second;
                Out << "\n";
            }
        }

    };
}

char CountStaticInstructions::ID = 0;
```

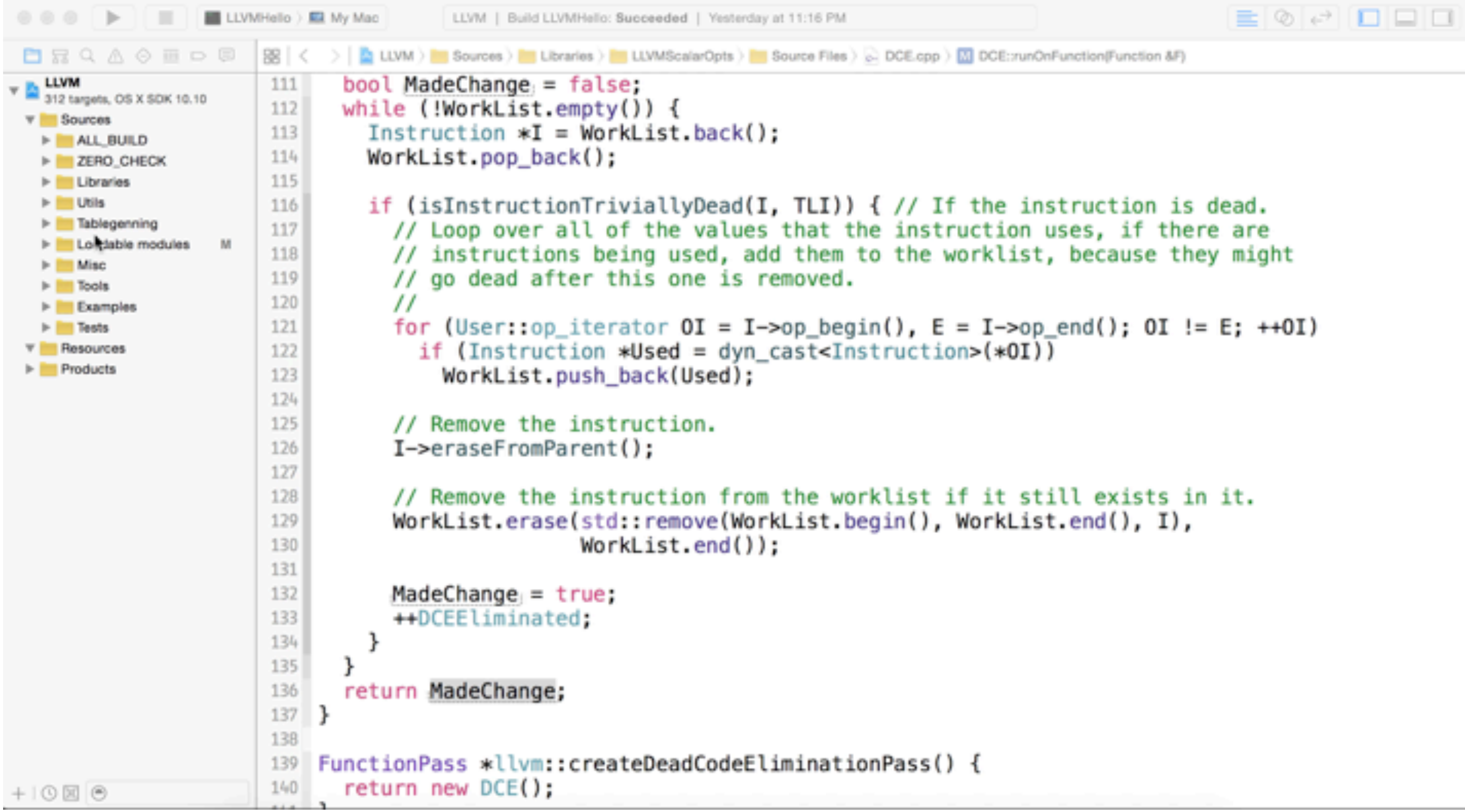
Hands-On: Exercise 4 - Hello Pass



Debug — test@bmills-HSW: ~/Desktop/Compilers/llvm-build/Debug — ..m-build/Debug — zsh — 80×24

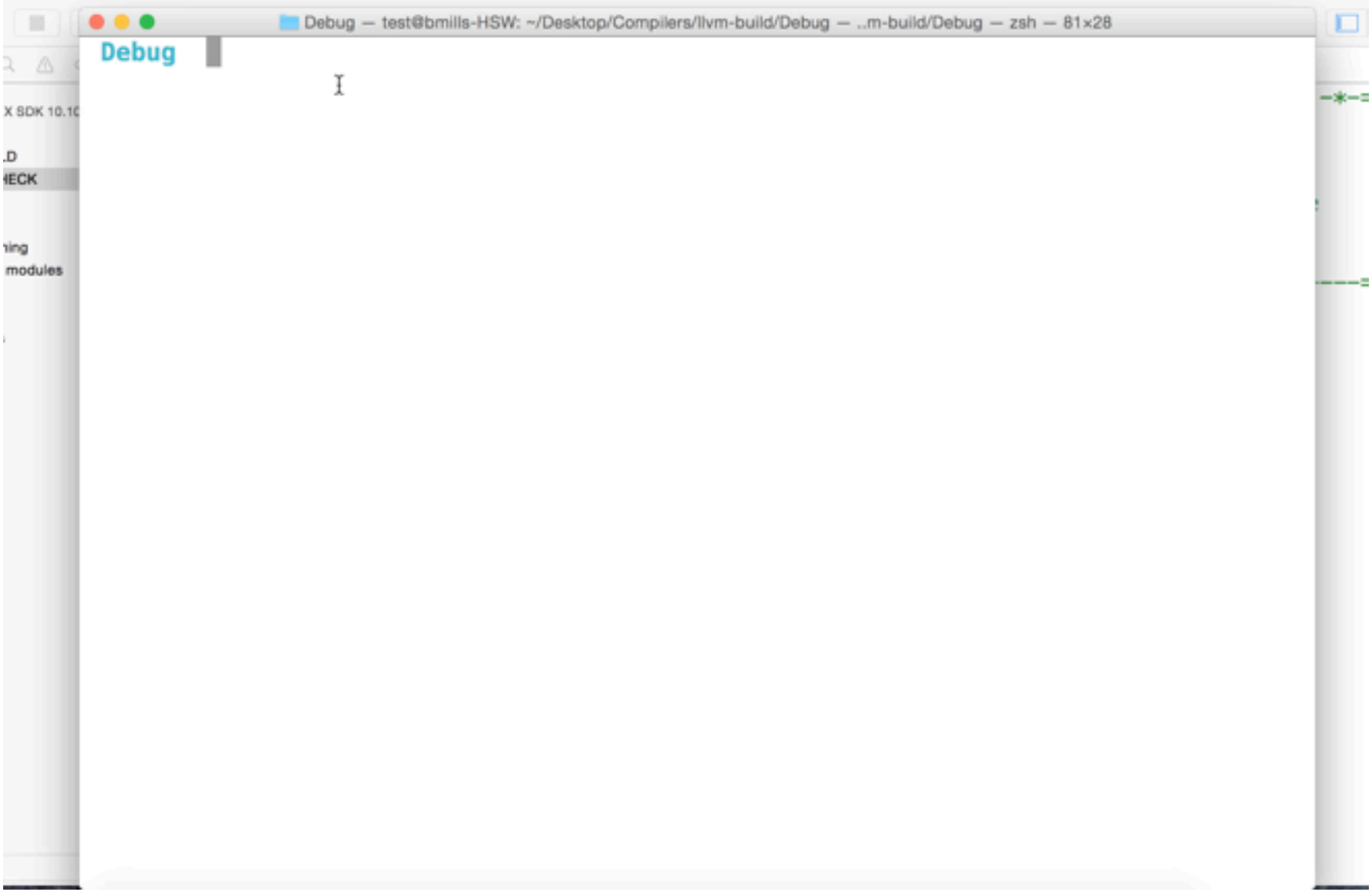
```
Debug
```

Hands-On: Exercise 4 - Count Instructions



```
  bool MadeChange = false;
  while (!WorkList.empty()) {
    Instruction *I = WorkList.back();
    WorkList.pop_back();

    if (isInstructionTriviallyDead(I, TLI)) { // If the instruction is dead.
      // Loop over all of the values that the instruction uses, if there are
      // instructions being used, add them to the worklist, because they might
      // go dead after this one is removed.
      //
      for (User::op_iterator OI = I->op_begin(), E = I->op_end(); OI != E; ++OI)
        if (Instruction *Used = dyn_cast<Instruction>(*OI))
          WorkList.push_back(Used);

      // Remove the instruction.
      I->eraseFromParent();

      // Remove the instruction from the worklist if it still exists in it.
      WorkList.erase(std::remove(WorkList.begin(), WorkList.end(), I),
                     WorkList.end());

      MadeChange = true;
      ++DCEEliminated;
    }
  }
  return MadeChange;
}

FunctionPass *llvm::createDeadCodeEliminationPass() {
  return new DCE();
}
```

Hands-On: Exercise 4 - Count Instructions



Debug — test@bmills-HSW: ~/Desktop/Compilers/llvm-build/Debug — ..m-build/Debug — zsh — 81×28

```
Debug
```

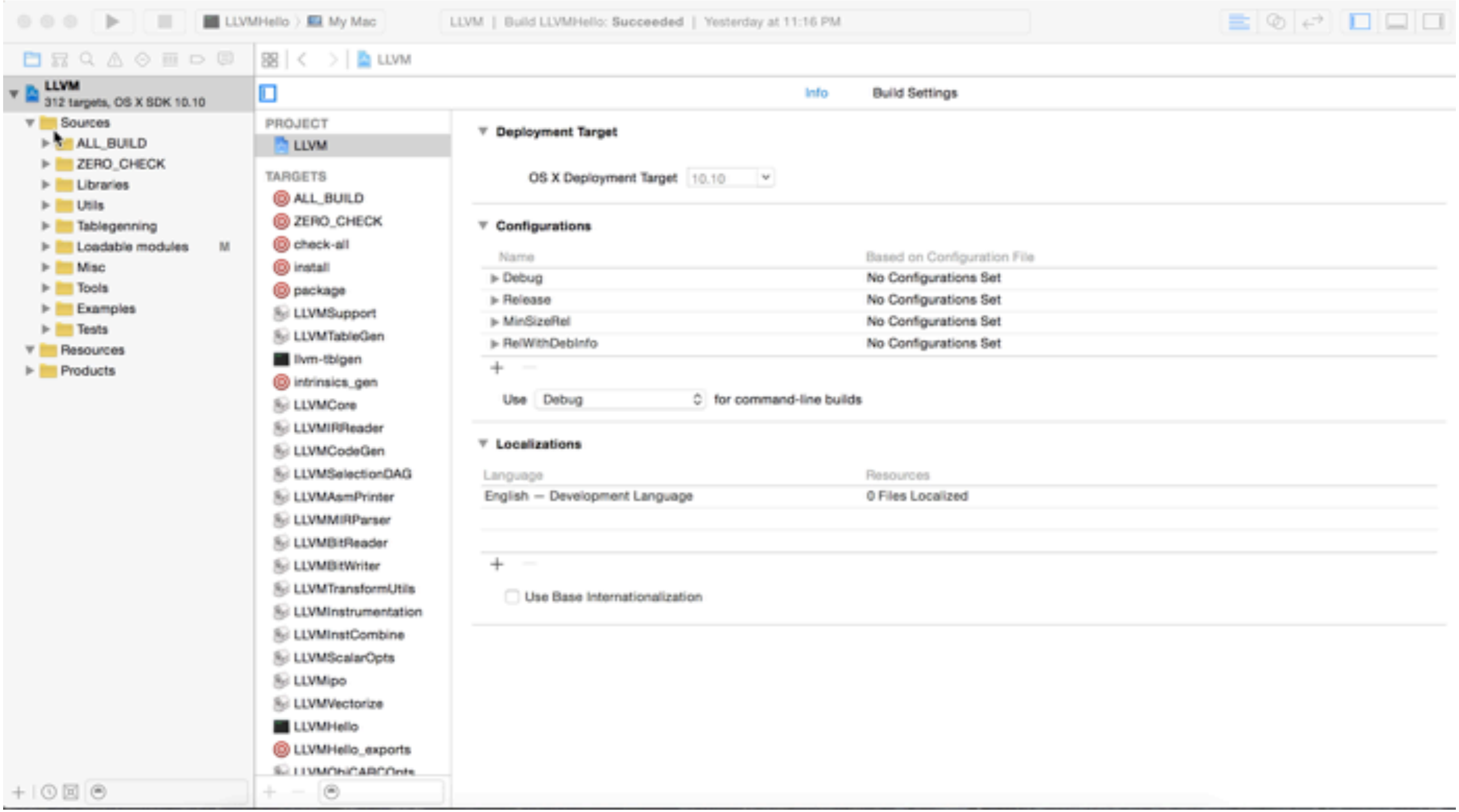
X SDK 10.1(
.D
HECK
ning
modules

Hands-On: Exercise 5

Walk through an LLVM transformation pass

- Objective
  - Learn how to write an LLVM transformation pass
  - Learn how an LLVM transformation pass modifies the LLVM IR
  - Learn how to verify correctness of a transformation pass

Hands-On: Exercise 5 - Dead Code Elimination



Deployment Target

OS X Deployment Target: 10.10

Configurations

Name	Based on Configuration File
▶ Debug	No Configurations Set
▶ Release	No Configurations Set
▶ MinSizeRel	No Configurations Set
▶ RelWithDebInfo	No Configurations Set

Use Debug for command-line builds

Localizations

Language	Resources
English — Development Language	0 Files Localized

□ Use Base Internationalization

Hands-On: Exercise 5 - Dead Code Elimination

Debug — test@bmills-HSW: ~/Desktop/Compilers/llvm-build/Debug — ..m-build/Debug — zsh — 67×21

```
Debug
```

Summing Up

- LLVM Compiler
  - Design overview
  - IR, API
  - Optimizations, Analysis Passes
- Tools
- Examples

References

http://llvm.org/docs/index.html#

http://llvm.org/docs/GettingStarted.html

http://clang.llvm.org/get_started.html

http://llvm.org/docs/LangRef.html

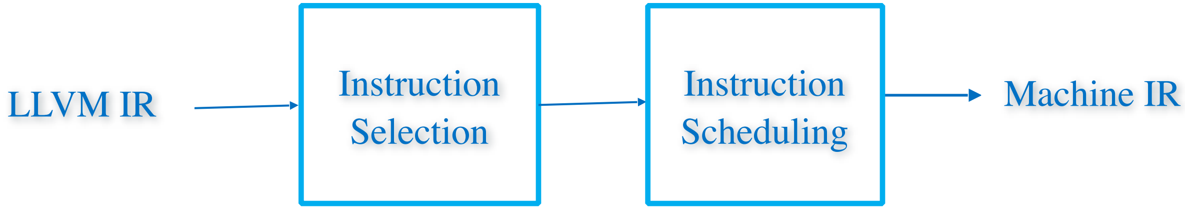
http://cgo.org/cgo2015/conference/workshops-and-tutorials/#llvm

https://soco.intel.com/groups/inside-llvm

BYOC!

Backup

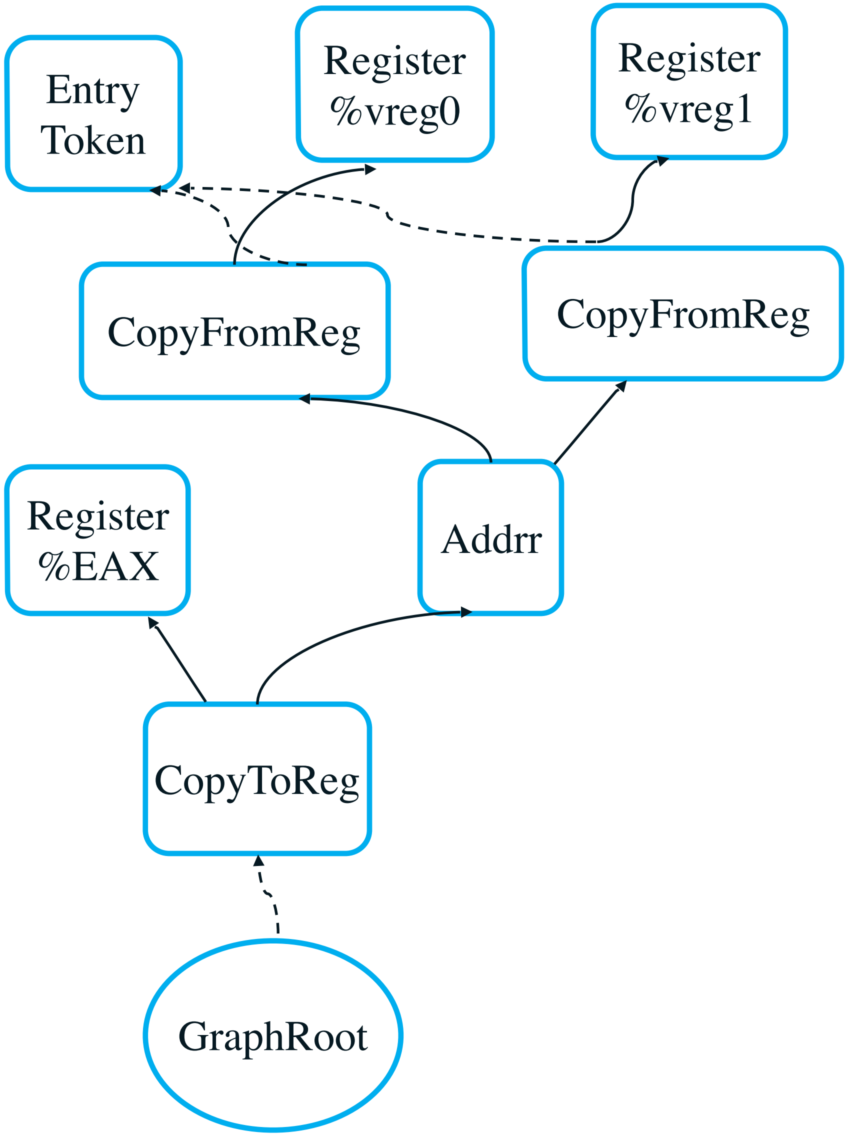
LLVM Backend - Instruction Scheduling



$llc -print-machineinstrsmachine dumps instructions after all registered passes

$llc -print-machineinstrs=<pass-name> dumps instructions after a specific pass

SelectionDAG Example



Example Machine IR

```
BB#0: derived from LLVM BB %entry

    Live Ins: %I0 %I1

  %vreg1<def> = COPY %I1; IntRegs:%vreg1

  %vreg0<def> = COPY %I0; IntRegs:%vreg0

  %vreg2<def> = ADDrr %vreg1, %vreg0; IntRegs:
%vreg2,%vreg1,%vreg0

  %I0<def> = COPY %vreg2; IntRegs:%vreg2

  RETL 8, %I0<imp-use>
```

SelectionDAG Instruction Select Phase Example

```
def MADDrr: Instx86<(outs GRRegs:$dst),
            (ins GRRegs: $src1, GRRegs:$src2, GRRegs: $src3),
           “madd $dst, $src1, $src2, $src3",
           [(set i32:$dst, (add(mul(i32:$src1, i32:$src2), i32:$src3))];


def ADDrr: Instx86<(outs GRRegs:$dst),
            (ins GRRegs: $src1, GRRegs:$src2),
           "add $dst, $src1, $src2",
           [(set i32:$dst, (add i32:$src1, i32:$src2))];
```

MCInst Example

```
addl -4(%rbp), %esi
 ## <MCInst #89 ADD32rm
 ##  <MCOperand Reg:29>
 ##  <MCOperand Reg:29>
 ##  <MCOperand Reg:36>
 ##  <MCOperand Imm:1>
 ##  <MCOperand Reg:0>
 ##  <MCOperand Imm:-4>
 ##  <MCOperand Reg:0>>
```

Assembly Example

```
.cfi_def_cfa_register %rbp
  movl    %edi, -4(%rbp)
  movl    %esi, -8(%rbp)
  addl    -4(%rbp), %esi
  movl    %esi, %eax
  popq    %rbp
  ret
  .cfi_endproc
```

TableGen for LLVM backends

- LLVM’s domain specific language
- Helps LLVM understand the target architecture
- Minimizes repetition and errors
- Declare machine specific aspects in single location but has multiple uses

Example:<Target>InstrInfo.td interpreted by AsmWriter backend or SelectionDAGISel backend

```
def ADDrr: Instx86<(outs GRRegs:$dst),
                  (ins GRRegs: $src1, GRRegs:$src2),
         "add $dst, $src1, $src2",
         [(set i32:$dst, (add i32:$src1, i32:$src2))];
```

Backend Code Structure

Main libraries are found in the lib directory and its subfolders

- CodeGen: generic code gen algorithms
- MC: Low level functionality for assemblers, disassemblers and object file types (eg: ELF COFF)
- TableGen: tool to generate C++ code based on high level descriptors found in .td file
- Target: Target specific implementation each within a different subfolder under Target eg: Target/x86

Generating DAG Graphs

llc –march=x86 –[OPTION] sum.ll

Where Option is:

-view-dag-combine1-dags displays the DAG after being built, before the first optimization pass.

-view-legalize-dags displays the DAG before Legalization.

-view-dag-combine2-dags displays the DAG before the second optimization pass.

-view-isel-dags displays the DAG before the Select phase.

-view-sched-dags displays the DAG before Scheduling.

LLVM Backend Classes

SelectionDAGISel – main base class for instruction selection

SelectionDAGBuilder::visit - SelectionDAGISel goes through IR and calls the visit() dispathcer on them

TargetLowering – an important interface to convey target specific information to target-independent algorithms

SelectionDAGISel::Select – custom instruction selection code is inserted here

InstrEmitter::EmitMachineNode – Scheduler uses this class to emit instructions into a machinebasicblock

TableGen Files

Target/<target>/<target>InstInfo.td – to implement instructions

Target/<target>/<target>RegisterInfo.td – to implement registers/ register classes

Target/<target>/<target>CallingConv.td – to implement calling conventions

<target>ISelLowering.cpp – Target specific hooks to lower instructions to target specific code. eg: LowerFormalArgs(), LowerRet()

<target>InstructionPrinter.cpp – printOperand() is given to the stream to print the instruction