Introduction to LLVM

Yajin Zhou (http://yajin.org)

Zhejiang University

Credits: DCD888 (Fernando M Q Pereira)

The Mission of the Compiler Writer

The goal of a compiler writer is to bridge the gap between programming languages and the hardware; hence, making programmers more productive

A compiler writer builds ***bridges*** *between* ***people*** *and* ***machines****, and this task is each day more challenging.*

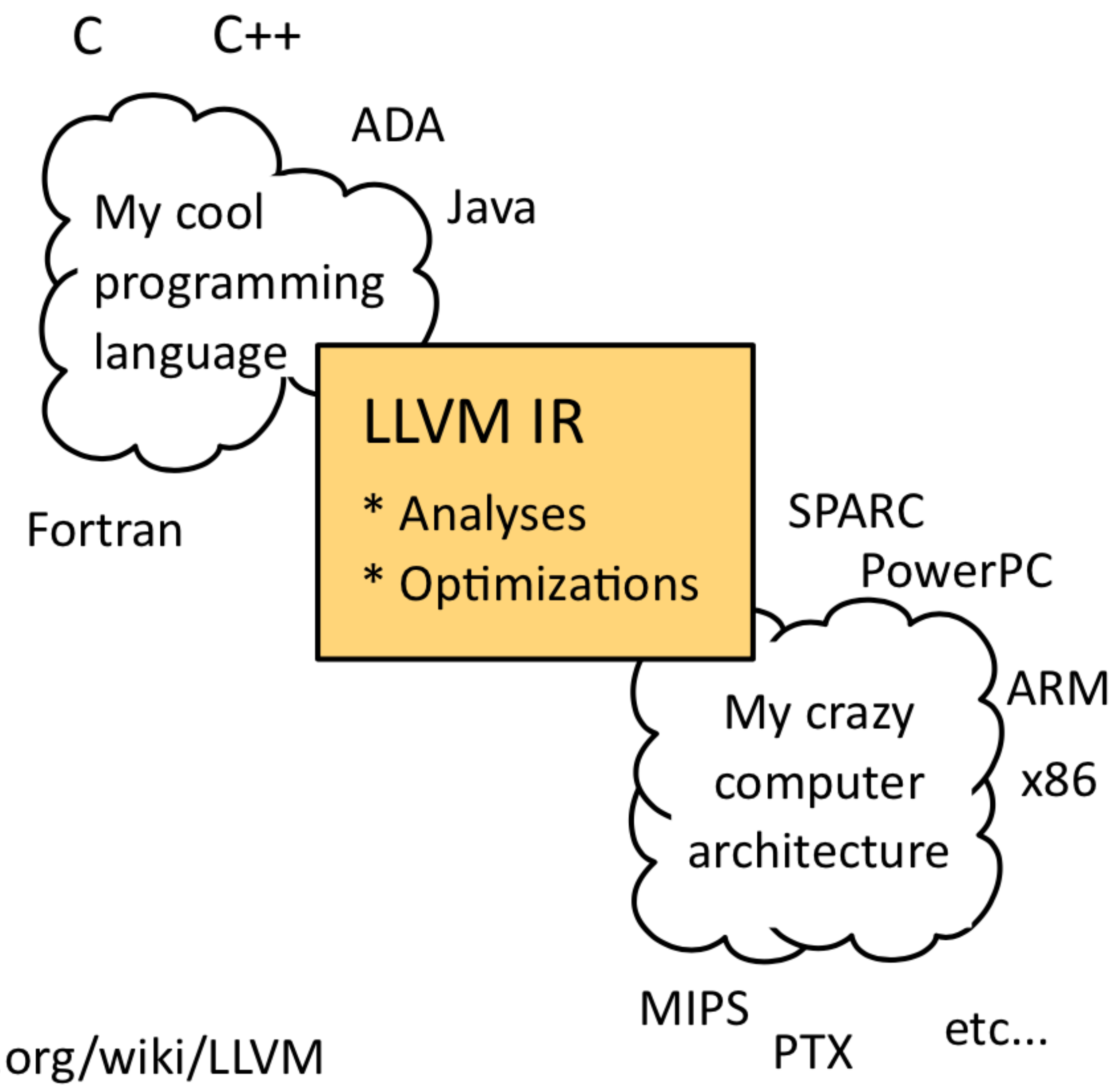
Software engineers want abstractions that let them stay closer to the specification of the problems that they need to solve.

Hardware engineers want efficiency. To obtain every little nanosecond of speed, they build machines each time more (beautifully) complex.

What is LLVM?

LLVM is a compiler infrastructure designed as a set of reusable libraries with well-defined interfaces♤.

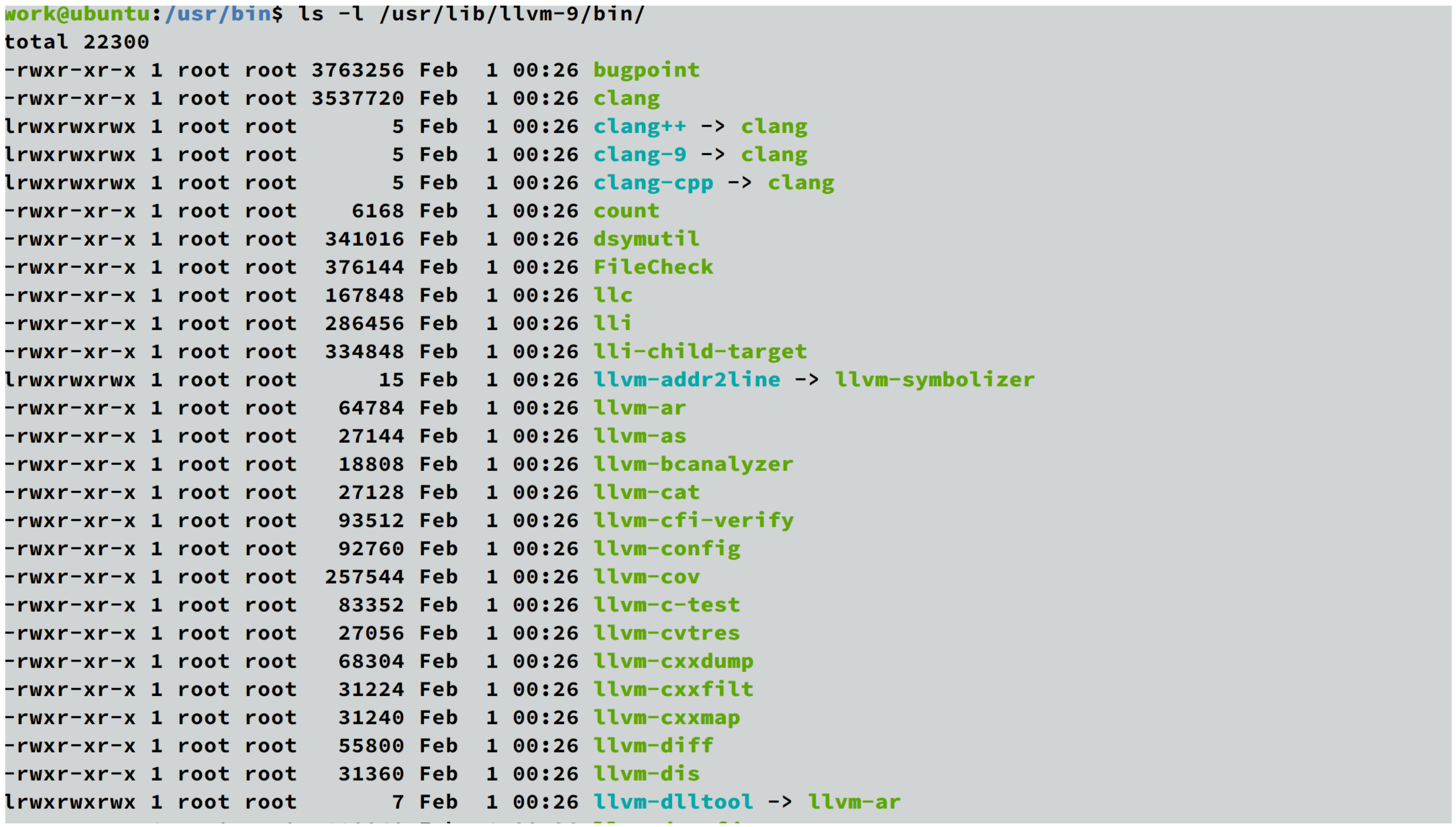
- Implemented in C++
- Several front-ends
- Several back-ends
- First release: 2003
- Open source
- `http://llvm.org/`



♤: Taken from wikipedia at http://en.wikipedia.org/wiki/LLVM

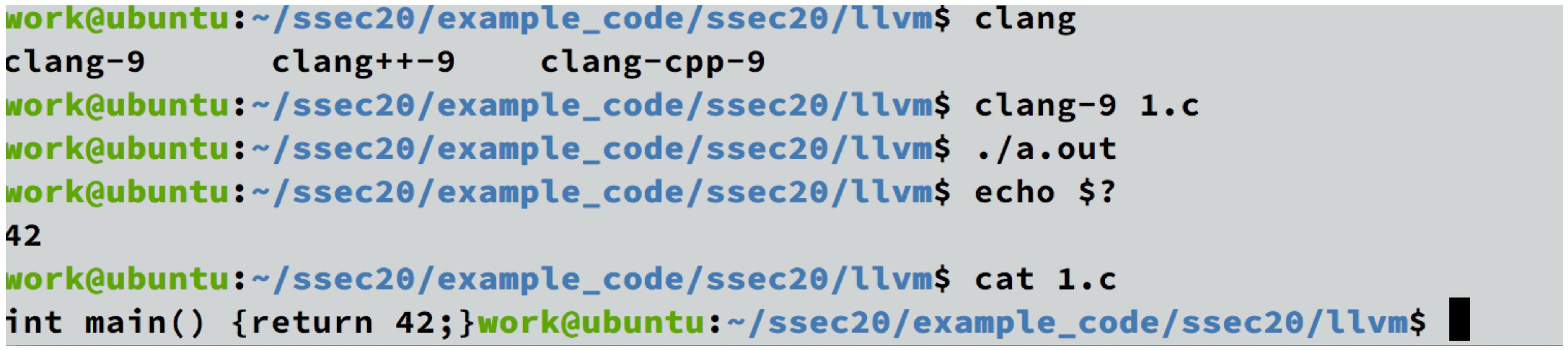
LLVM is a Compilation Infrastructure

- It is a framework that comes with lots of tools to compile and optimize code.



```
work@ubuntu:/usr/bin$ ls -l /usr/lib/llvm-9/bin/
total 22300
-rwxr-xr-x 1 root root 3763256 Feb  1 00:26 bugpoint
-rwxr-xr-x 1 root root 3537720 Feb  1 00:26 clang
lrwxrwxrwx 1 root root       5 Feb  1 00:26 clang++ -> clang
lrwxrwxrwx 1 root root       5 Feb  1 00:26 clang-9 -> clang
lrwxrwxrwx 1 root root       5 Feb  1 00:26 clang-cpp -> clang
-rwxr-xr-x 1 root root    6168 Feb  1 00:26 count
-rwxr-xr-x 1 root root  341016 Feb  1 00:26 dsymutil
-rwxr-xr-x 1 root root  376144 Feb  1 00:26 FileCheck
-rwxr-xr-x 1 root root  167848 Feb  1 00:26 llc
-rwxr-xr-x 1 root root  286456 Feb  1 00:26 lli
-rwxr-xr-x 1 root root  334848 Feb  1 00:26 lli-child-target
lrwxrwxrwx 1 root root      15 Feb  1 00:26 llvm-addr2line -> llvm-symbolizer
-rwxr-xr-x 1 root root   64784 Feb  1 00:26 llvm-ar
-rwxr-xr-x 1 root root   27144 Feb  1 00:26 llvm-as
-rwxr-xr-x 1 root root   18808 Feb  1 00:26 llvm-bcanalyzer
-rwxr-xr-x 1 root root   27128 Feb  1 00:26 llvm-cat
-rwxr-xr-x 1 root root   93512 Feb  1 00:26 llvm-cfi-verify
-rwxr-xr-x 1 root root   92760 Feb  1 00:26 llvm-config
-rwxr-xr-x 1 root root  257544 Feb  1 00:26 llvm-cov
-rwxr-xr-x 1 root root   83352 Feb  1 00:26 llvm-c-test
-rwxr-xr-x 1 root root   27056 Feb  1 00:26 llvm-cvtres
-rwxr-xr-x 1 root root   68304 Feb  1 00:26 llvm-cxxdump
-rwxr-xr-x 1 root root   31224 Feb  1 00:26 llvm-cxxfilt
-rwxr-xr-x 1 root root   31240 Feb  1 00:26 llvm-cxxmap
-rwxr-xr-x 1 root root   55800 Feb  1 00:26 llvm-diff
-rwxr-xr-x 1 root root   31360 Feb  1 00:26 llvm-dis
lrwxrwxrwx 1 root root       7 Feb  1 00:26 llvm-dlltool -> llvm-ar
```

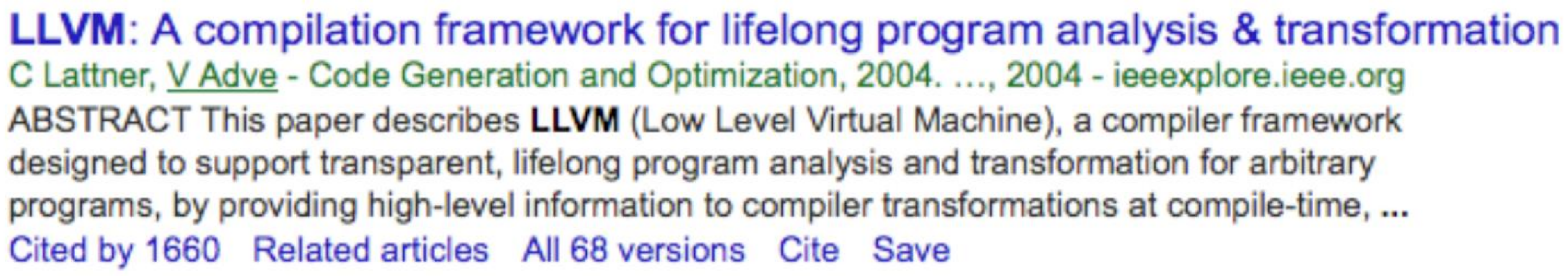
LLVM is a Compilation Infrastructure



```
work@ubuntu:~/ssec20/example_code/ssec20/llvm$ clang
clang-9      clang++-9    clang-cpp-9
work@ubuntu:~/ssec20/example_code/ssec20/llvm$ clang-9 1.c
work@ubuntu:~/ssec20/example_code/ssec20/llvm$ ./a.out
work@ubuntu:~/ssec20/example_code/ssec20/llvm$ echo $?
42
work@ubuntu:~/ssec20/example_code/ssec20/llvm$ cat 1.c
int main() {return 42;}work@ubuntu:~/ssec20/example_code/ssec20/llvm$ 
```

Why to Learn LLVM?

- Intensively used in the academia♧:

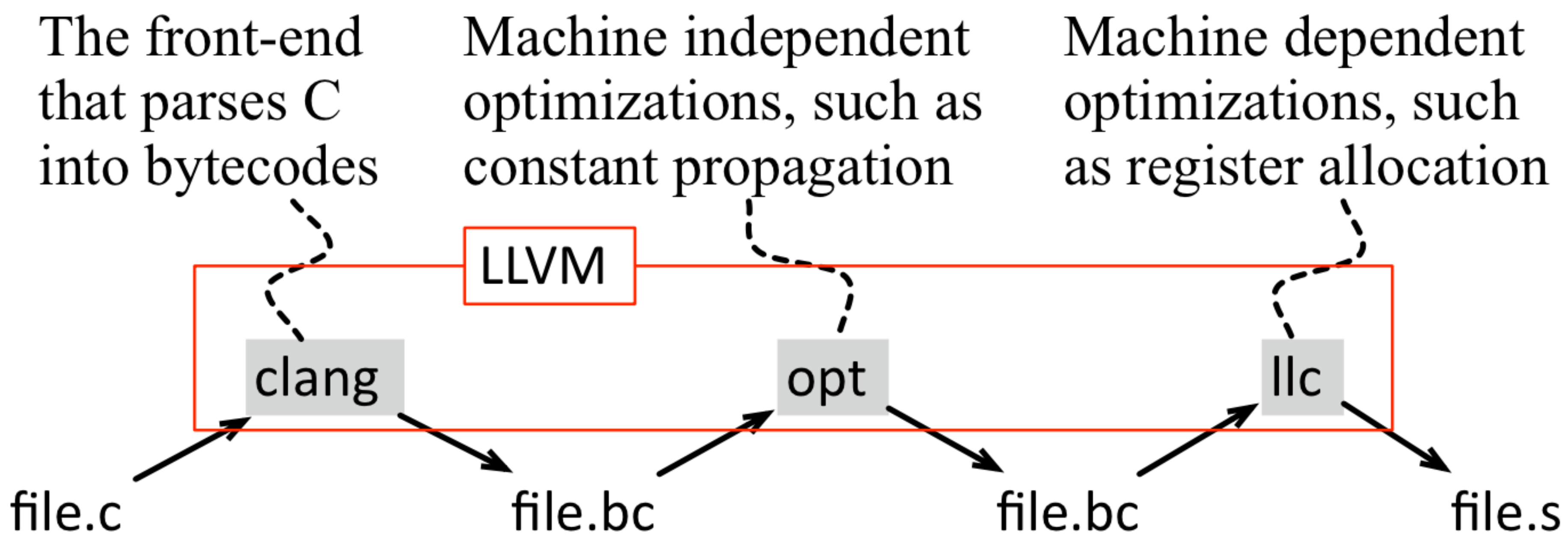


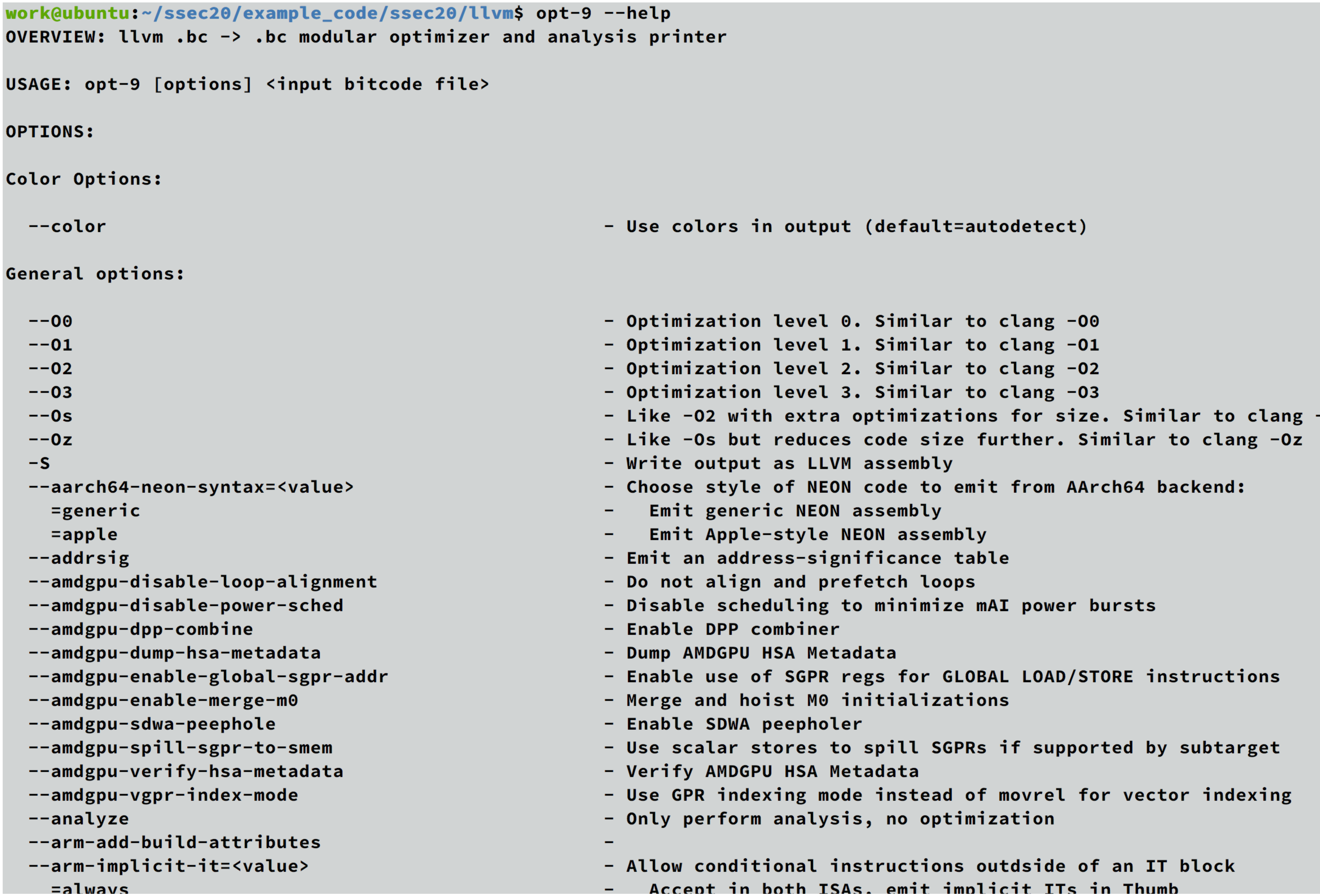
LLVM: A compilation framework for lifelong program analysis & transformation
C Lattner, V Adve - Code Generation and Optimization, 2004. ..., 2004 - ieeexplore.ieee.org
ABSTRACT This paper describes **LLVM** (Low Level Virtual Machine), a compiler framework designed to support transparent, lifelong program analysis and transformation for arbitrary programs, by providing high-level information to compiler transformations at compile-time, ...
Cited by 1660 Related articles All 68 versions Cite Save

- Used by many companies
  - LLVM is maintained by Apple.
  - ARM, NVIDIA, Mozilla, Cray, etc.
- Clean and modular interfaces.
- Important awards:
  - Most cited CGO paper; ACM Software System Award 2012

Optimizations in Practice

- The **opt** tool, available in the LLVM toolbox, performs machine independent optimizations.
- There are many optimizations available through opt.
  - To have an idea, type `opt --help.`


The front-end that parses C into bytecodes
Machine independent optimizations, such as constant propagation
Machine dependent optimizations, such as register allocation
LLVM
clang
opt
llc
file.c
file.bc
file.bc
file.s



```
work@ubuntu:~/ssec20/example_code/ssec20/llvm$ opt-9 --help
OVERVIEW: llvm .bc -> .bc modular optimizer and analysis printer

USAGE: opt-9 [options] <input bitcode file>

OPTIONS:

Color Options:

  --color                                           - Use colors in output (default=autodetect)

General options:

  --O0                                              - Optimization level 0. Similar to clang -O0
  --O1                                              - Optimization level 1. Similar to clang -O1
  --O2                                              - Optimization level 2. Similar to clang -O2
  --O3                                              - Optimization level 3. Similar to clang -O3
  --Os                                              - Like -O2 with extra optimizations for size. Similar to clang -
  --Oz                                              - Like -Os but reduces code size further. Similar to clang -Oz
  -S                                                - Write output as LLVM assembly
  --aarch64-neon-syntax=<value>                     - Choose style of NEON code to emit from AArch64 backend:
    =generic                                        -   Emit generic NEON assembly
    =apple                                          -   Emit Apple-style NEON assembly
  --addrsig                                         - Emit an address-significance table
  --amdgpu-disable-loop-alignment                   - Do not align and prefetch loops
  --amdgpu-disable-power-sched                      - Disable scheduling to minimize mAI power bursts
  --amdgpu-dpp-combine                              - Enable DPP combiner
  --amdgpu-dump-hsa-metadata                        - Dump AMDGPU HSA Metadata
  --amdgpu-enable-global-sgpr-addr                  - Enable use of SGPR regs for GLOBAL LOAD/STORE instructions
  --amdgpu-enable-merge-m0                          - Merge and hoist M0 initializations
  --amdgpu-sdwa-peephole                            - Enable SDWA peepholer
  --amdgpu-spill-sgpr-to-smem                       - Use scalar stores to spill SGPRs if supported by subtarget
  --amdgpu-verify-hsa-metadata                      - Verify AMDGPU HSA Metadata
  --amdgpu-vgpr-index-mode                          - Use GPR indexing mode instead of movrel for vector indexing
  --analyze                                         - Only perform analysis, no optimization
  --arm-add-build-attributes                        -
  --arm-implicit-it=<value>                         - Allow conditional instructions outdside of an IT block
    =always                                         -   Accept in both ISAs, emit implicit ITs in Thumb
```

Levels of Optimizations

- Like gcc, clang supports different levels of optimizations, e.g., -O0 (default), -O1, -O2 and -O3.
- To find out which optimization each level uses, you can try:

llvm-as is the LLVM assembler. It reads a file containing human-readable LLVM assembly language, translates it to LLVM bytecode, and writes the result into a file or to standard output.

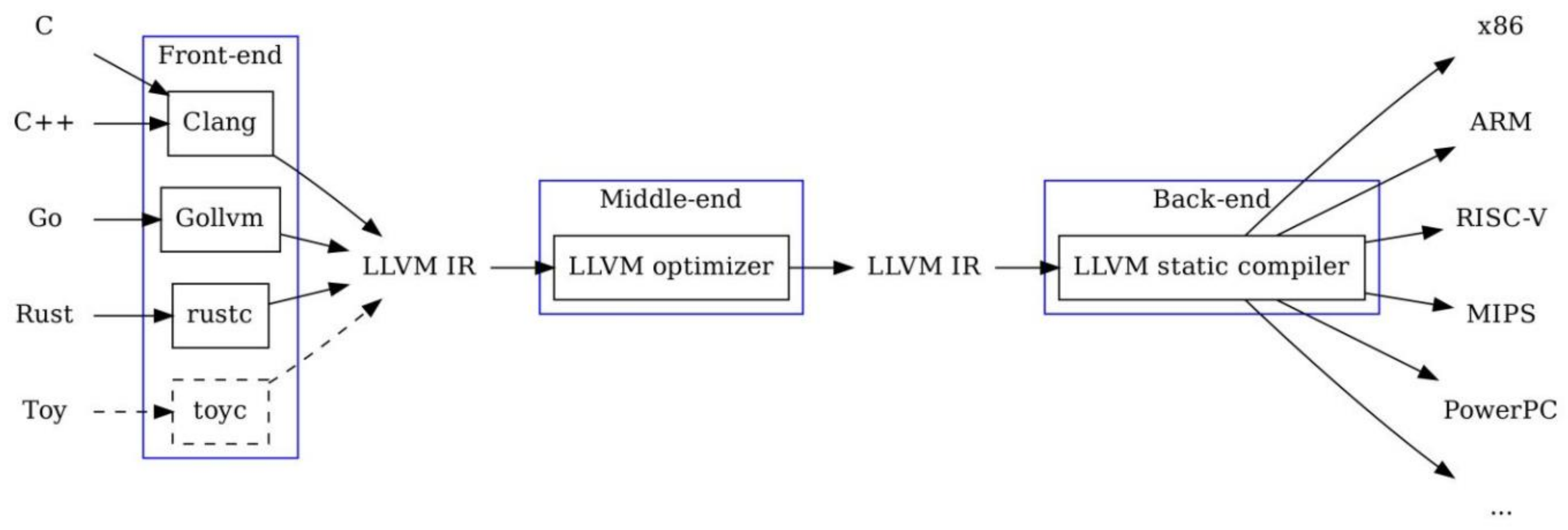
```
$> llvm-as < /dev/null | opt -O3 -disable-output -debug-pass=Arguments
```

In my system (LLVM/Darwin), -O1 gives me:

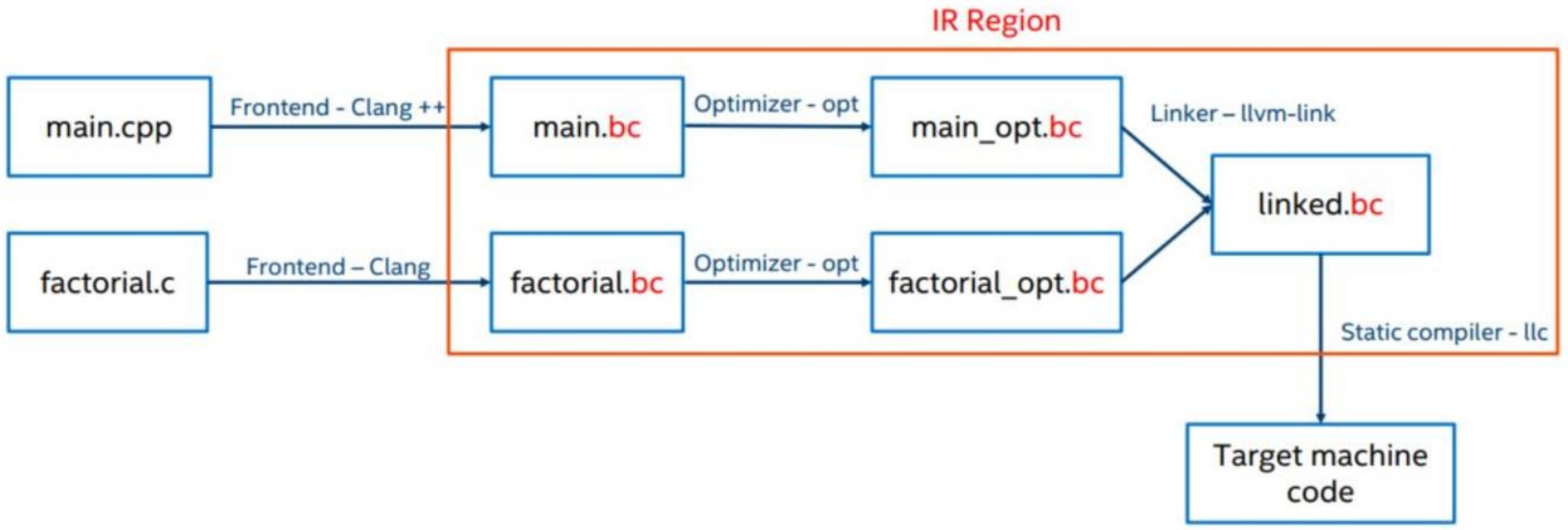
-targetlibinfo -no-aa -tbaa -basicaa -notti -globalopt -ipsccp -deadargelim -instcombine -simplifycfg -basiccg -prune-eh -inline-cost -always-inline -functionattrs -sroa -domtree -early-cse -lazy-value-info -jump-threading -correlated-propagation -simplifycfg - instcombine -tailcallelim -simplifycfg -reassociate -domtree -loops -loop-simplify -lcssa -loop-rotate -licm -lcssa -loop-unswitch -instcombine -scalar-evolution -loop-simplify - lcssa -indvars -loop-idiom -loop-deletion -loop-unroll -memdep -memcpyopt -sccp - instcombine -lazy-value-info -jump-threading -correlated-propagation -domtree - memdep -dse -adce -simplifycfg -instcombine -strip-dead-prototypes -preverify - domtree -verify

LLVM IR

- Low-level language, which is similar to RISC instruction
- However, it can express high-level semantics



LLVM IR


IR Region
main.cpp
Frontend - Clang ++
main.bc
Optimizer - opt
main_opt.bc
Linker – llvm-link
linked.bc
factorial.c
Frontend – Clang
factorial.bc
Optimizer - opt
factorial_opt.bc
Static compiler - llc
Target machine code

LLVM IR

- LLVM represents programs, internally, via its own instruction set.
  - The LLVM optimizations manipulate these bytecodes.
  - We can program directly on them.
  - We can also interpret them.

```
int callee(const int* X) {
  return *X + 1;
}

int main() {
  int T = 4;
  return callee(&T);
}
```

```
$> clang –c –emit-llvm f.c –o f.bc

$> opt –mem2reg f.bc –o f.bc

$> llvm-dis f.bc

$> cat f.ll
```

```
; Function Attrs: nounwind ssp
define i32 @callee(i32* %X) #0 {
entry:
  %0 = load i32* %X, align 4
  %add = add nsw i32 %0, 1
  ret i32 %add
}
```

LLVM IR

- Bytecode is a form of instruction set designed for efficient execution by a software interpreter.
  - They are portable!
  - Example: Java bytecodes.
- The tool **`lli`** directly executes programs in LLVM bitcode format.
  - `lli` may compile these bytecodes just-in-time, if a JIT is available.

```
$> echo "int main() {printf(\"Oi\n\");}" > t.c

$> clang -c -emit-llvm t.c -o t.bc

$> lli t.bc
```

We Can Program Directly on the IR

This is C

```
int callee(const int* X) {
  return *X + 1;
}

int main() {
  int T = 4;
  return callee(&T);
}
```

```
$> clang -c -emit-llvm ex0.c -o
ex0.bc

$> opt -mem2reg -instnamer
ex0.bc -o ex0.bc

$> llvm-dis < ex0.bc
```

This is LLVM

```
; Function Attrs: nounwind ssp
define i32 @callee(i32* %X) #0 {
entry:
  %tmp = load i32* %X, align 4
  %add = add nsw i32 %tmp, 1
  ret i32 %add
}

; Function Attrs: nounwind ssp
define i32 @main() #0 {
entry:
  %T = alloca i32, align 4
  store i32 4, i32* %T, align 4
  %call = call i32 @callee(i32* %T)
  ret i32 %call
}
```

Which opts could we apply on this code?

♧: although this is not something to the faint of heart.

We Can Program Directly on the IR

<u>This is the original bytecode</u>

```
; Function Attrs: nounwind ssp
define i32 @callee(i32* %X) #0 {
entry:
  %tmp = load i32* %X, align 4
  %add = add nsw i32 %tmp, 1
  ret i32 %add
}

; Function Attrs: nounwind ssp
define i32 @main() #0 {
entry:
  %T = alloca i32, align 4
  store i32 4, i32* %T, align 4
  %call = call i32 @callee(i32* %T)
  ret i32 %call
}
```

<u>This is the optimized bytecode</u>

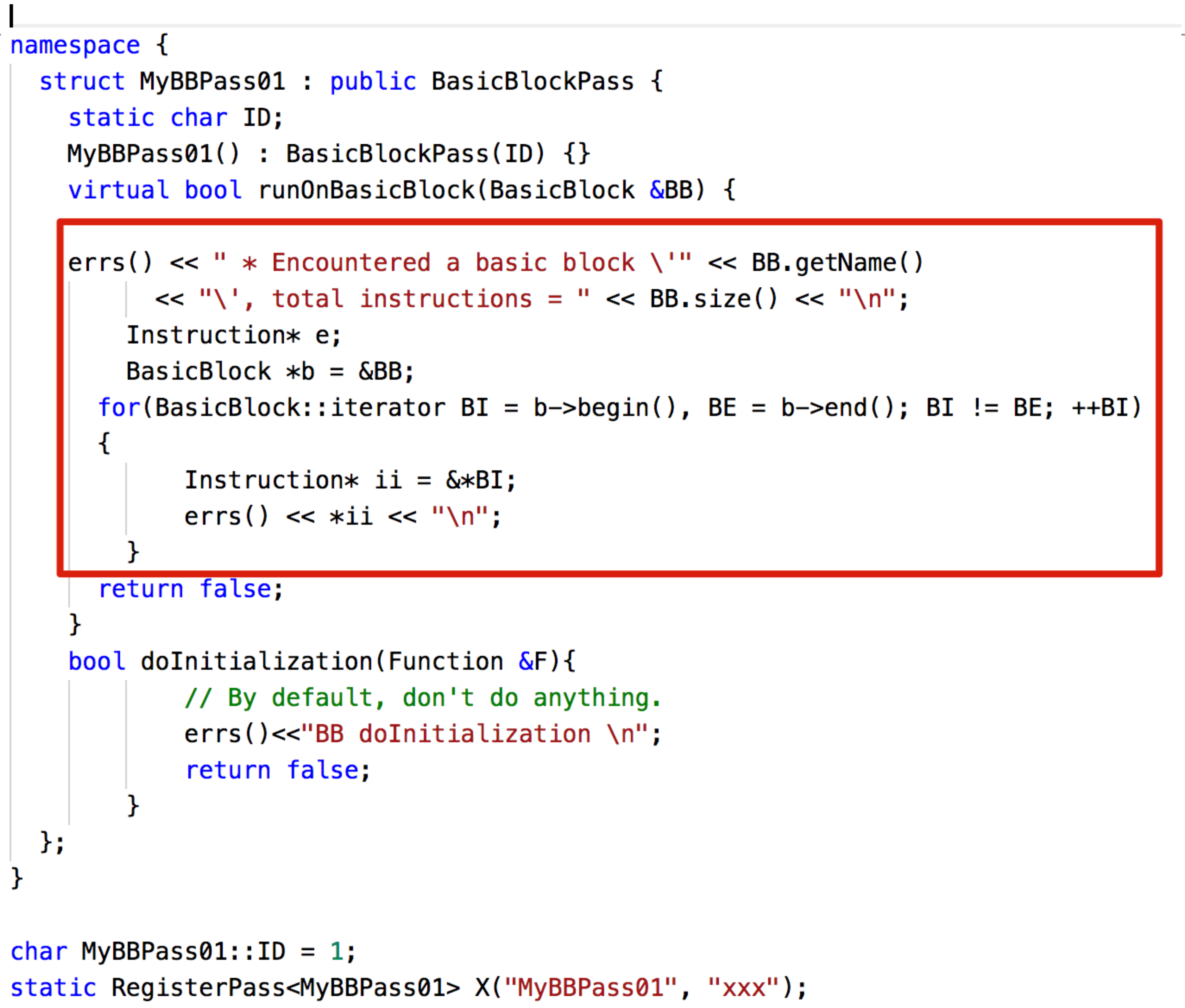
```
; Function Attrs: nounwind ssp
define i32 @callee(i32 %X) #0 {
entry:
  %add = add nsw i32 %X, 1
  ret i32 %add
}

; Function Attrs: nounwind ssp
define i32 @main() #0 {
entry:
  %call = call i32 @callee(i32 4)
  ret i32 %call
}
```

Can you point all the differences between the files?

We can compile and execute the bytecode file:

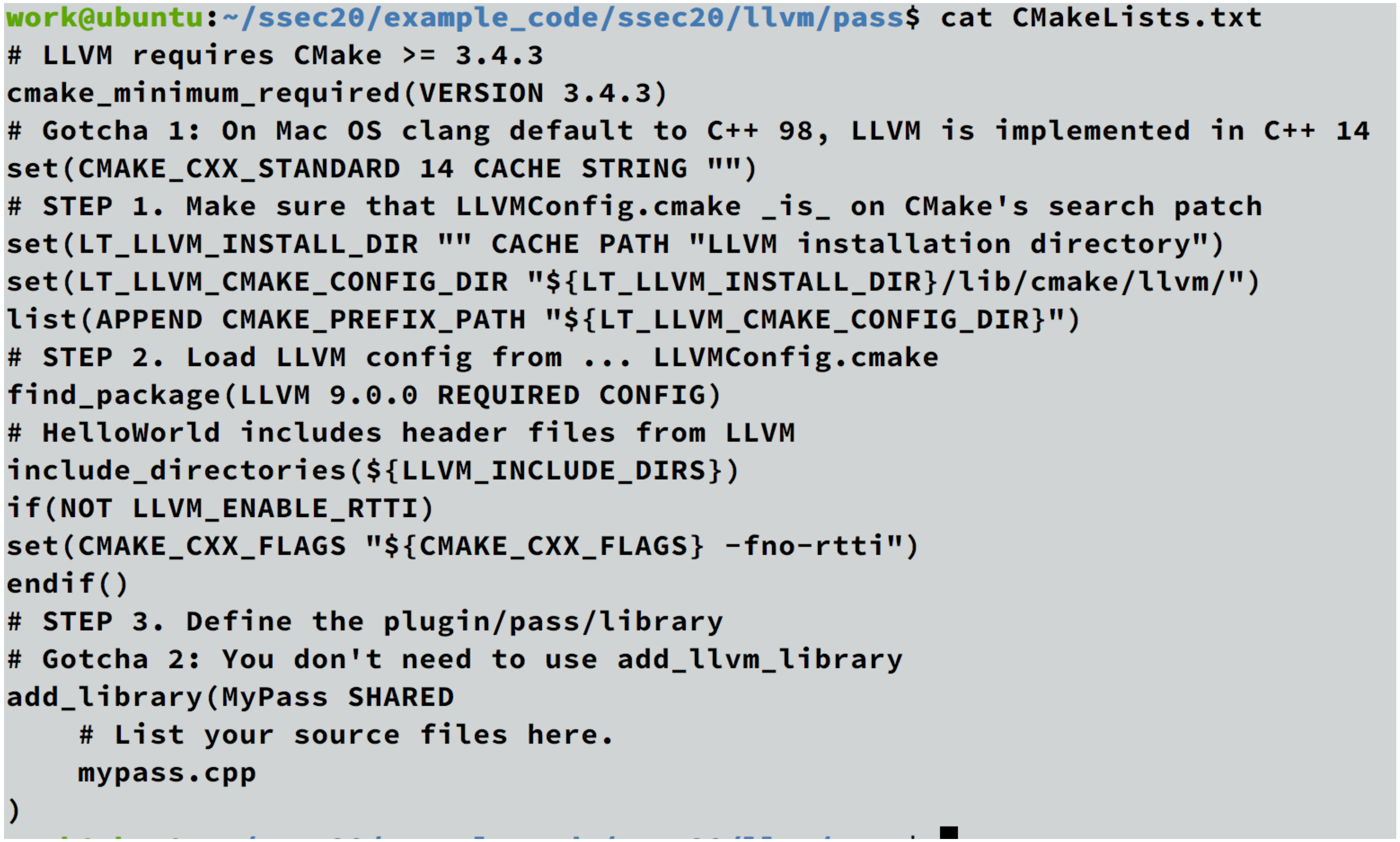
```
$> clang ex0.hack.ll
$> ./a.out
$> echo $?
$> 5
```

Writing an LLVM Pass



```
namespace {
 struct MyBBPass01 : public BasicBlockPass {
  static char ID;
  MyBBPass01() : BasicBlockPass(ID) {}
  virtual bool runOnBasicBlock(BasicBlock &BB) {

  errs() << " * Encountered a basic block \'" << BB.getName()
     << "\', total instructions = " << BB.size() << "\n";
    Instruction* e;
    BasicBlock *b = &BB;
   for(BasicBlock::iterator BI = b->begin(), BE = b->end(); BI != BE; ++BI)
   {
      Instruction* ii = &*BI;
      errs() << *ii << "\n";
    }
   return false;
  }
  bool doInitialization(Function &F){
      // By default, don't do anything.
      errs()<<"BB doInitialization \n";
      return false;
    }
 };
}

char MyBBPass01::ID = 1;
static RegisterPass<MyBBPass01> X("MyBBPass01", "xxx");
```

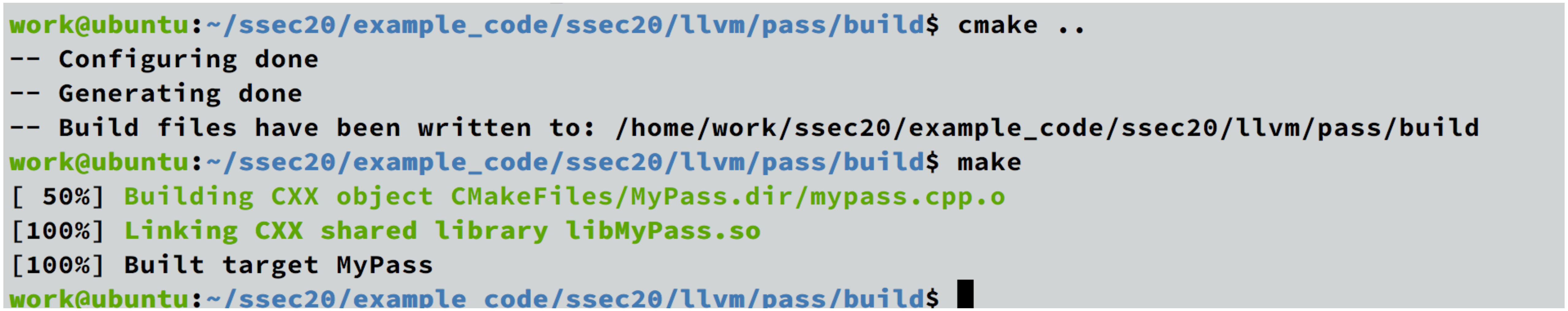
Writing an LLVM Pass



```
work@ubuntu:~/ssec20/example_code/ssec20/llvm/pass$ cat CMakeLists.txt
# LLVM requires CMake >= 3.4.3
cmake_minimum_required(VERSION 3.4.3)
# Gotcha 1: On Mac OS clang default to C++ 98, LLVM is implemented in C++ 14
set(CMAKE_CXX_STANDARD 14 CACHE STRING "")
# STEP 1. Make sure that LLVMConfig.cmake _is_ on CMake's search patch
set(LT_LLVM_INSTALL_DIR "" CACHE PATH "LLVM installation directory")
set(LT_LLVM_CMAKE_CONFIG_DIR "${LT_LLVM_INSTALL_DIR}/lib/cmake/llvm/")
list(APPEND CMAKE_PREFIX_PATH "${LT_LLVM_CMAKE_CONFIG_DIR}")
# STEP 2. Load LLVM config from ... LLVMConfig.cmake
find_package(LLVM 9.0.0 REQUIRED CONFIG)
# HelloWorld includes header files from LLVM
include_directories(${LLVM_INCLUDE_DIRS})
if(NOT LLVM_ENABLE_RTTI)
set(CMAKE_CXX_FLAGS "${CMAKE_CXX_FLAGS} -fno-rtti")
endif()
# STEP 3. Define the plugin/pass/library
# Gotcha 2: You don't need to use add_llvm_library
add_library(MyPass SHARED
    # List your source files here.
    mypass.cpp
)
```

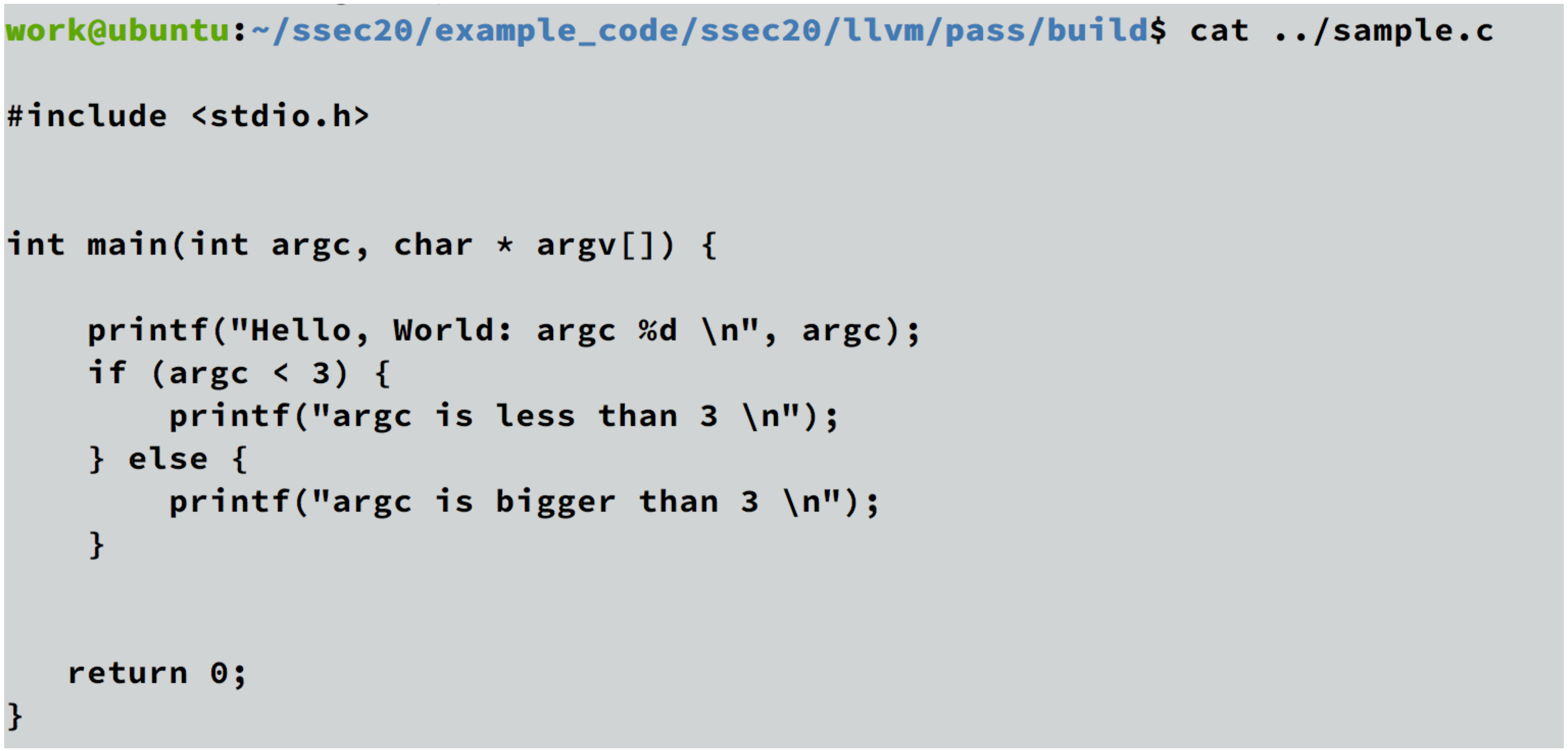
Build the pass

- mkdir build
- cd build



```
work@ubuntu:~/ssec20/example_code/ssec20/llvm/pass/build$ cmake ..
-- Configuring done
-- Generating done
-- Build files have been written to: /home/work/ssec20/example_code/ssec20/llvm/pass/build
work@ubuntu:~/ssec20/example_code/ssec20/llvm/pass/build$ make
[ 50%] Building CXX object CMakeFiles/MyPass.dir/mypass.cpp.o
[100%] Linking CXX shared library libMyPass.so
[100%] Built target MyPass
work@ubuntu:~/ssec20/example_code/ssec20/llvm/pass/build$
```

Run the pass

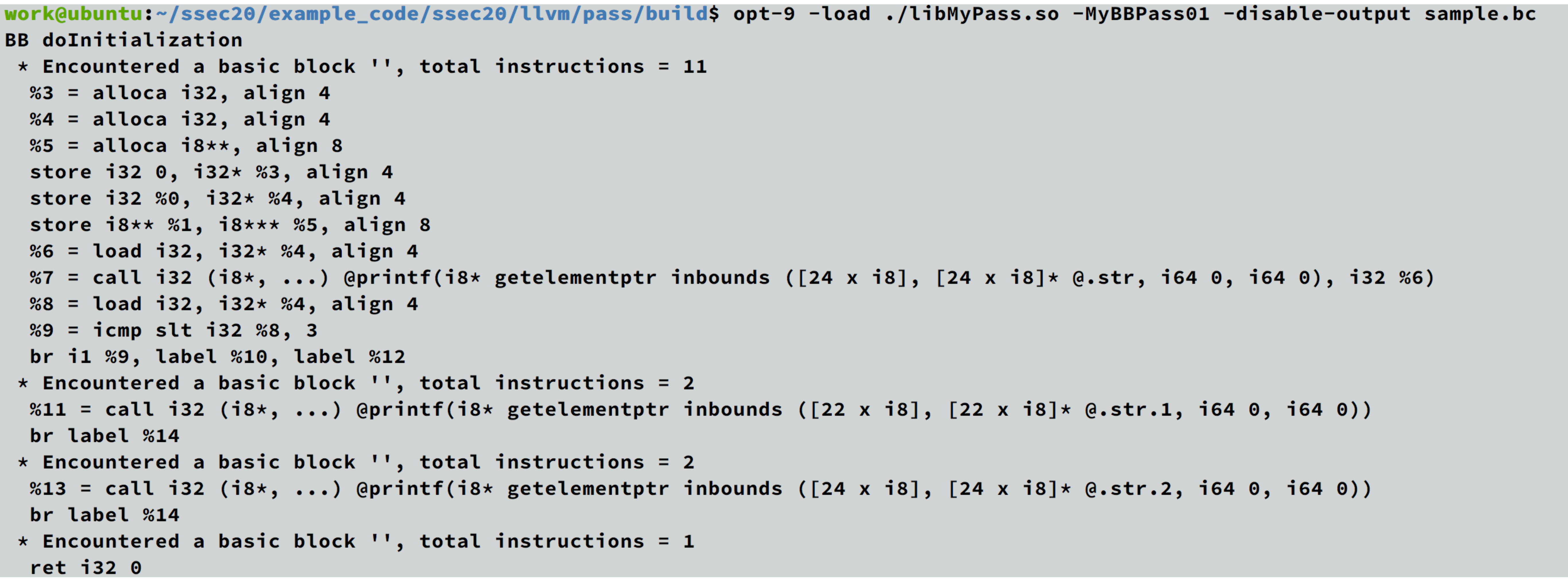


```
work@ubuntu:~/ssec20/example_code/ssec20/llvm/pass/build$ cat ../sample.c

#include <stdio.h>


int main(int argc, char * argv[]) {

    printf("Hello, World: argc %d \n", argc);
    if (argc < 3) {
        printf("argc is less than 3 \n");
    } else {
        printf("argc is bigger than 3 \n");
    }


   return 0;
}
```

- clang-9 -c -emit-llvm ../sample.c -o sample.bc
- llvm-dis-9 sample.bc

Run the pass

```
; Function Attrs: noinline nounwind optnone uwtable
define dso_local i32 @main(i32, i8**) #0 {
  %3 = alloca i32, align 4
  %4 = alloca i32, align 4
  %5 = alloca i8**, align 8
  store i32 0, i32* %3, align 4
  store i32 %0, i32* %4, align 4
  store i8** %1, i8*** %5, align 8
  %6 = load i32, i32* %4, align 4
  %7 = call i32 (i8*, ...) @printf(i8* getelementptr inbounds ([24 x i8], [24 x i8]* @.str, i64 0, i64 0), i32 %6)
  %8 = load i32, i32* %4, align 4
  %9 = icmp slt i32 %8, 3
  br i1 %9, label %10, label %12

10:                                               ; preds = %2
  %11 = call i32 (i8*, ...) @printf(i8* getelementptr inbounds ([22 x i8], [22 x i8]* @.str.1, i64 0, i64 0))
  br label %14

12:                                               ; preds = %2
  %13 = call i32 (i8*, ...) @printf(i8* getelementptr inbounds ([24 x i8], [24 x i8]* @.str.2, i64 0, i64 0))
  br label %14

14:                                               ; preds = %12, %10
  ret i32 0
}
```



```
work@ubuntu:~/ssec20/example_code/ssec20/llvm/pass/build$ opt-9 -load ./libMyPass.so -MyBBPass01 -disable-output sample.bc
BB doInitialization
 * Encountered a basic block '', total instructions = 11
  %3 = alloca i32, align 4
  %4 = alloca i32, align 4
  %5 = alloca i8**, align 8
  store i32 0, i32* %3, align 4
  store i32 %0, i32* %4, align 4
  store i8** %1, i8*** %5, align 8
  %6 = load i32, i32* %4, align 4
  %7 = call i32 (i8*, ...) @printf(i8* getelementptr inbounds ([24 x i8], [24 x i8]* @.str, i64 0, i64 0), i32 %6)
  %8 = load i32, i32* %4, align 4
  %9 = icmp slt i32 %8, 3
  br i1 %9, label %10, label %12
 * Encountered a basic block '', total instructions = 2
  %11 = call i32 (i8*, ...) @printf(i8* getelementptr inbounds ([22 x i8], [22 x i8]* @.str.1, i64 0, i64 0))
  br label %14
 * Encountered a basic block '', total instructions = 2
  %13 = call i32 (i8*, ...) @printf(i8* getelementptr inbounds ([24 x i8], [24 x i8]* @.str.2, i64 0, i64 0))
  br label %14
 * Encountered a basic block '', total instructions = 1
  ret i32 0
```