Installing LLVM

[1] Download and Extract

Download “LLVM_Package.zip” and unzip the file into “~/llvm” folder.

You should see “clang-3.4.src.tar.gz” and “llvm-3.4.src.tar.gz” files.

Execute the following commands to extract and move the llvm and clang source files.

```
:~/llvm$ tar xvzf llvm-3.4.src.tar.gz
:~/llvm$ tar xvzf clang-3.4.src.tar.gz
:~/llvm$ mv llvm-3.4 src
:~/llvm$ mv clang-3.4 src/tools/clang
```

[2] Compile

To compile LLVM and clang, you need to have C++ compilers.

In Ubuntu (or Mint), you can install those compiler tools by installing “**build-essential**” package. For instance:

```
 $ sudo apt-get install build-essential
```

* you may need to install **python** too if it is not installed: **$ sudo apt-get install python**)
*** install_required_package.sh** file will do the above two commands.

[3] Build

Run the following commands to build.

```
:~/llvm$ mkdir build
:~/llvm$ cd build
:~/llvm/build$ ../src/configure
:~/llvm/build$ make
```

Building LLVM source code would take some time (e.g., ~30 mins).

If the compilation is successful, you will see the following line in the end.

```
llvm[0]: ***** Completed Release+Asserts Build
```

After the build, do “**sudo make install**” to **install the binaries**.
(**Caution**: If you are already using LLVM and clang, the following command may mess up with your installed version.)

```
 $ sudo make install
```

Add Your Own Project

1. You should have “MyLLVMPass” folder.

2. Run “install_project.sh” command to copy files and change Makefile.

```
~/llvm$ ./install_project.sh
```

3. Now the source code for MyLLVMPass is copied under “./src/lib/Transforms/MyLLVMPass/”. Edit “MyLLVMPass.cpp” to add the following code to print “Hello” message.

```
    virtual bool runOnModule(Module &M) {

        bool changed = false;
        errs() << "Hello, I am in the MyLLVMPass \n";
        return changed;
    }
```

4. To compile it, run “compile_myllvmpass.sh”

```
~/llvm$ ./compile_myllvmpass.sh
```

5. The compiled library is in “~/llvm/build/Release+Assert/lib” folder.

```
~/llvm/build/Release+Asserts/lib$ ls -al *.so
-rwxrwxr-x 1 osboxes osboxes    16056 Oct  4 12:57 BugpointPasses.so
-rwxrwxr-x 1 osboxes osboxes 24600840 Oct  4 13:11 libclang.so
-rwxrwxr-x 1 osboxes osboxes 33850360 Oct  4 12:56 libLTO.so
-rwxrwxr-x 1 osboxes osboxes    15752 Oct  4 16:25 LLVMHello.so
-rwxrwxr-x 1 osboxes osboxes    14808 Oct  4 17:12 MyLLVMPass.so
```

6. To run “MyLLVMPass” on a bitcode file, use “run_myllvmpass.sh”

```
~/llvm$ ./run_myllvmpass.sh [input bitcode file] [output bitcode
file]
```

Useful LLVM commands

1. **[COMPILER]** Compile C/C++ program into a bitcode (.bc) file (e.g., compiling YOURFILE.cpp into YOURFILE.bc).

```
$ clang -emit-llvm -o YOURFILE.bc -c YOURFILE.cpp
```

2. **[DISASSEMBLER]** A bitcode file is binary. To get textual representation, you use “llvm-dis”. The below command will generate YOURFILE.ll which is a textual representation of the LLVM bitcode file.

```
$ llvm-dis YOURFILE.bc
```

3. **[INTERPRETER]** Executing LLVM bitcode without creating a machine-dependent binary may be useful in some cases. You can use “lli YOURFILE.ll [args]” to do it.

```
$ lli YOURFILE.bc [ARG1] [ARG2] [...]
```

4. [COMPILER] Compile a bitcode file into a binary.

```
$ clang -o [BINARY FILENAME] [BITCODE FILENAME]
```

5. [GETTING C++ CODE THAT GENERATES LLVM IR] llc is a tool that can generate a C++ program that generates LLVM IR statements of a given bitcode program.

```
$ llc -march=cpp [BITCODE FILENAME] -o [C++ FILENAME]
```

This is particularly useful when you do not know how to create LLVM IR to instrument (e.g., global variable declaration, function calls, etc.). Below is an example program. You can see how LLVM APIs are used to construct IRs in C++ programs. You will borrow some of those in your LLVM pass. Note that this functionality is not supported in the latest LLVM (obsolete).

```
using namespace llvm;

Module* makeLLVMModule();

int main(int argc, char**argv) {
  Module* Mod = makeLLVMModule();
  verifyModule(*Mod, PrintMessageAction);
  PassManager PM;
  PM.add(createPrintModulePass(&outs()));
  PM.run(*Mod);
  return 0;
}


Module* makeLLVMModule() {
 // Module Construction
 Module* mod = new Module("hello.bc", getGlobalContext());
 mod->setDataLayout("e-p:64:64:64-i1:8:8-i8:8:8-i16:16:16-i32:32:32-
i64:64:64-f32:32:32-f64:64:64-v64:64:64-v128:128:128-a0:0:64-s0:64:64-
f80:128:128-n8:16:32:64-S128");
 mod->setTargetTriple("x86_64-unknown-linux-gnu");

 // Type Definitions
 ArrayType* ArrayTy_0 = ArrayType::get(IntegerType::get(mod->getContext(),
8), 14);

 PointerType* PointerTy_1 = PointerType::get(ArrayTy_0, 0);

 std::vector<Type*>FuncTy_2_args;
 FunctionType* FuncTy_2 = FunctionType::get(
  /*Result=*/IntegerType::get(mod->getContext(), 32),
  /*Params=*/FuncTy_2_args,
  /*isVarArg=*/false);
 PointerType* PointerTy_3 = PointerType::get(IntegerType::get(mod-
>getContext(), 32), 0);

 PointerType* PointerTy_4 = PointerType::get(IntegerType::get(mod-
>getContext(), 8), 0);

 std::vector<Type*>FuncTy_6_args;
 FuncTy_6_args.push_back(PointerTy_4);
 FunctionType* FuncTy_6 = FunctionType::get(
  /*Result=*/IntegerType::get(mod->getContext(), 32),
  /*Params=*/FuncTy_6_args,
  /*isVarArg=*/true);

 PointerType* PointerTy_5 = PointerType::get(FuncTy_6, 0);


 // Function Declarations

 Function* func_main = mod->getFunction("main");
 if (!func_main) {
```

6. [MORE BUILD TOOLS] WLLVM: Whole Program LLVM. While our samples are small, hence it is easy to compile them with clang to get llvm bitcode files. However, getting bitcode files of larger programs using complex build tools (e.g., makefile) is not easy. One possible way is to edit all the makefiles to replace gcc/g++ with clang with a right flag (e.g., -emit-llvm), those require significant engineering effort. Moreover, build tools even use pipes to pass results into other tools which make the task even more challenging.
   WLLVM is a wrapper program that makes it easy. With wllvm, all you need to do is changing the compiler. All the flags and options for gcc/g++ will be handled by wllvm (or wllvm++).

Building LLVM documentation

We are using old version of LLVM (version 3.4), thus you may find some differences for APIs and LLVM IR specification that you find from publicly available documentations and the LLVM environment that you are working on. Fortunately, the LLVM source includes a full set of documentation which is located under “src/docs” in Sphinx documentation format. Therefore, it needs Sphinx document generator to build a HTML format document repository, since Makefile calls “sphinx_build” command to build documentation. You can install the package using “brew install sphinx_build” for Mac or “sudo apt install python_sphinx” for Ubuntu systems.

```
$
$ cd docs/
$ make -f Makefile.sphinx # need "sphinx_build” command
$ cd _build/html
```

Using IDE for LLVM

LLVM allows you to edit LLVM source using IDEs with better user interface (e.g., XCode for Mac or VisualStudio for Windows).

From build directory "~/llvm/build", you run "cmake -G Xcode …/llvm" command to create XCode project file. Similarly, you can configure Ninja or Visual Studio for your LLVM source. Look for Getting started page (https://llvm.org/docs/GettingStarted.html) for available options.