

# Documentation

## Step by Step Documentation

Below is the step by step Documentation of the tools used during the course of the project. This documentation is in order of the steps followed by us as the project proceeded. Each documentation displays the a)\* *installation procedure* the tool, b)\* *it's usage* and c)\* *its contribution* in the project.

\*a) Installation Procedure - How to install a particular tool on the Operating System, its required patches, how to set up the environment, how to configure, how to build, how to make etc.

\*b) Usage - How to use a particular tool, how to run, how to get desired results, how to set input, how to obtain output etc.

\*c) Contribution - Why the tool was necessary, its output and how outputs will pave the way for further proceeding of project work.

---

## **1) GCC Compiler (GNU Compiler Collection)**

GCC is a key component of "GNU Toolchain", for developing applications, as well as operating systems. The GNU Toolchain includes:

1. GNU Compiler Collection (GCC): a compiler suit that supports many languages, such as C/C++, Objective-C and Java.
2. GNU Make: an automation tool for compiling and building applications.
3. GNU Binutils: a suit of binary utility tools, including linker and assembler.
4. GNU Debugger (GDB).
5. GNU Autotools: A build system including Autoconf, Autoheader, Automake and Libtool.
6. GNU Bison: a parser generator (similar to lex and yacc).

GCC is portable and run in many operating platforms. GCC (and GNU Toolchain) is currently available on all Unixes. They are also ported to Windows by MinGW and Cygwin. GCC is also a cross-compiler, for producing executables on different platform.

### Installation Procedure

Type the following command on the terminal : `sudo apt-get install gcc`

This command will install the latest version of gcc on the computer.

The Latest Version of Ubuntu i.e. 16.04 has pre-installed gcc compiler. Hence you will get a message like this :

```
nishit@nishit:~$ sudo apt-get install gcc
Reading package lists... Done
Building dependency tree
Reading state information... Done
gcc is already the newest version (4:5.3.1-1ubuntu1).
0 upgraded, 0 newly installed, 0 to remove and 218 not upgraded.
```

All the Environment variables and Path will be added automatically.

### Usage

To compile a C program, just type : `gcc <filename>`

File must be .c file.

We use the -o option to specify the output file name.

The -o flag is used like this:

```
gcc [options] [source files] [object files] -o output file
```

To use different optimization levels while compiling, use the -O flag with a prefix

-----  
O0 - No Optimization

O1 - Level 1 Optimization

O2 - Level 2 Optimization

O3 - Level 3 Optimization

Ofast - Level 4 Optimization  
-----

```
gcc -Olevel [options] [source files] [object files] [-o output file]
```

Suppose the file name is myfile.c Build myfile.c on terminal and run the output file myfile:

```
gcc myfile.c -o myfile
```

```
/myfile
```

To display the time of program execution, use the time command of linux.

```
time ./<output_filename>
```

### Contribution

The initial phase of the project is done on gcc making use of Optimization flags and time to compare the execution time of the *Matrix Multiplication\** program on different levels as shown:

(\*Matrix Multiplication on a size of matrix 400 x 400 which includes numerous arithmetic operations is used as a benchmark for the project)

```
nishit@nishit:~$ gcc -O0 matrix_mult.c
nishit@nishit:~$ time ./a.out
400, 1.171602

real    0m1.195s
user    0m1.192s
sys     0m0.000s
nishit@nishit:~$ gcc -O1 matrix_mult.c
nishit@nishit:~$ time ./a.out
400, 0.078639

real    0m0.101s
user    0m0.096s
sys     0m0.000s
nishit@nishit:~$ gcc -O2 matrix_mult.c
nishit@nishit:~$ time ./a.out
400, 0.100647

real    0m0.129s
user    0m0.124s
sys     0m0.004s
nishit@nishit:~$ gcc -O3 matrix_mult.c
nishit@nishit:~$ time ./a.out
400, 0.060930

real    0m0.093s
user    0m0.084s
sys     0m0.012s
nishit@nishit:~$ gcc -Ofast matrix_mult.c
nishit@nishit:~$ time ./a.out
400, 0.031325

real    0m0.052s
user    0m0.028s
sys     0m0.020s
```

---

## 2) GNU PLOT

Gnuplot is a portable command-line driven graphing utility for Linux, OS/2, MS Windows, OSX, VMS, and many other platforms. The source code is copyrighted but freely distributed (i.e., you don't have to pay for it). It was originally created to allow scientists and students to visualize mathematical functions and data interactively, but has grown to support many non-interactive uses such as web scripting. It is also used as a plotting engine by third-party applications like Octave. Gnuplot has been supported and under active development since 1986.

### Installation

Type the following command on the terminal : `sudo apt-get install gnuplot-x11`

### Usage

This tool is used to plot graphs on the screen.

This tool takes a file containing values of x and y which are comma separated.

Different coordinates are written on different lines.

To draw a plot, simply type gnuplot on the terminal. The gnuplot shell will open.

In the shell just type: `plot "<filename>" with lines-points`

To give label to an axis, type : `set xlabel "labelname"`  
`set ylabel "labelname"`

To set the title on a graph, use: `set title "title"`

To draw multiple curves on a single graph:

```
plot "file1" using 1:2 title "file1", \
    "file2" using 1:2 title "file2", \
    "file3" using 1:2 title "file3", \
    "file4" using 1:2 title "file4", \
    "file5" using 1:2 title "file5", \
    "file6" using 1:2 title "file6"
```

### Contribution

GNU PLOT gives a comparative study on a graph by which the survey of execution time becomes easy and easy to understand giving an overall picture.

---

## **3) Cross-Compiler**

A cross compiler is a compiler capable of creating executable code for a platform other than the one on which the compiler is running. For example, a compiler that runs on a Windows 7 PC but generates code that runs on Android smartphone is a cross compiler.

### Installation

The cross compiler requires the following packages as assistant for successful installation.

`ia32-livs , ia32-libs and lib32z1`

To install just type: `sudo apt-get install <package_name>`

To install the cross compiler, type the following commands:

```
sudo apt-get install libc6-armel-cross libc6-dev-armel-cross
```

```
sudo apt-get install binutils-arm-linux-gnueabi
```

```
sudo apt-get install libncurses5-dev
```

### Usage

To cross compile a C file for the ARM Architecture, use the following command

```
arm-linux-gnueabi-gcc -static -Ox filename.c -o output
```

(x represents optimization level : 0/1/2/3/fast)

This will create a 32-bit LSB Executable file for the ARM Architecture.

You can see the file type using the file command: `file <filename>`

```
nishit@nishit:~$ arm-linux-gnueabi-gcc -O1 -static matrix_mult.c -o mat
nishit@nishit:~$ file mat
mat: ELF 32-bit LSB executable, ARM, EABI5 version 1 (SYSV), statically linked,
for GNU/Linux 3.2.0, BuildID[sha1]=4014ef7d8f81b14bc55f95e1105434eb990873fd, not
stripped
```

### Contribution

We need a cross compiler as to compile a code that would run on ARM Architecture on which we can estimate the power consumed for the survey.

-----

## **4) Gem5**

The gem5 simulator is a modular platform for computer-system architecture research, encompassing system-level architecture as well as processor microarchitecture. ARM: gem5 can model up to 64 (heterogeneous) cores of a Realview ARM platform, and boot unmodified Linux and Android with a combination of in-order and out-of-order CPUs. The ARM implementation supports 32 or 64-bit kernels and applications.

### Installation

```
sudo apt-get update
```

```
sudo apt-get install swig gcc m4 python python-dev
```

```
libgoogle-perftools-dev
```

```
mercurial scons g++ build-essential
```

```
hg clone http://repo.gem5.org/gem5
cd gem5/
scons build/ARM/gem5.opt -j2

build/ARM/gem5.opt configs/example/se.py -c
tests/test-progs/hello/bin/arm/linux/hel lo
```

The above commands will install gem5 simulator on the system and set up the ARM Architecture for Simulator on the PC.

## Usage

To generate the stats file, follow the steps:

```
cd gem5
build/ARM/gem5.opt configs/example/se.py -c
tests/test-progs/hello/bin/arm/linux/<exe>
```

This will generate the stats file and store it in the subdirectory **m5out**.

The stats file looks like this :

```
|----- Begin Simulation Statistics -----|
sim_seconds          0.153623                # Number of seconds simulated
sim_ticks            153622636500             # Number of ticks simulated
final_tick           153622636500             # Number of ticks from beginning of simulation (restored from checkpoints and never reset)
sim_freq             1000000000000            # Frequency of simulated ticks
host_inst_rate       1750812                  # Simulator instruction rate (inst/s)
host_op_rate         1758146                  # Simulator op (including micro ops) rate (op/s)
host_tick_rate       879872751                # Simulator tick rate (ticks/s)
host_mem_usage       653756                   # Number of bytes of host memory used
host_seconds         174.76                   # Real time elapsed on the host
sim_insts            385963754                # Number of instructions simulated
sim_ops              307245259                # Number of ops (including micro ops) simulated
system.voltage_domain.voltage 1               # Voltage in Volts
system.clk_domain.clock 1000                 # Clock period in ticks
system.mem_ctrls.pwrStateResidencyTicks::UNDEFINED 153622636500 # Cumulative time (in ticks) in various power states
system.mem_ctrls.bytes_read::cpu.inst 1223855076 # Number of bytes read from this memory
system.mem_ctrls.bytes_read::cpu.data 587855297 # Number of bytes read from this memory
system.mem_ctrls.bytes_read::total 1810910673 # Number of bytes read from this memory
system.mem_ctrls.bytes_inst_read::cpu.inst 1223855076 # Number of instructions bytes read from this memory
system.mem_ctrls.bytes_inst_read::total 1223855076 # Number of instructions bytes read from this memory
system.mem_ctrls.bytes_written::cpu.data 69449375 # Number of bytes written to this memory
system.mem_ctrls.bytes_written::total 69449375 # Number of bytes written to this memory
system.mem_ctrls.num_reads::cpu.inst 385963760 # Number of read requests responded to by this memory
system.mem_ctrls.num_reads::cpu.data 146523681 # Number of read requests responded to by this memory
system.mem_ctrls.num_reads::total 452487450 # Number of read requests responded to by this memory
system.mem_ctrls.num_writes::cpu.data 17362375 # Number of write requests responded to by this memory
system.mem_ctrls.num_writes::total 17362375 # Number of write requests responded to by this memory
system.mem_ctrls.bw_read::cpu.inst 7966632417 # Total read bandwidth from this memory (bytes/s)
system.mem_ctrls.bw_read::cpu.data 3821413370 # Total read bandwidth from this memory (bytes/s)
system.mem_ctrls.bw_read::total 11788845787 # Total read bandwidth from this memory (bytes/s)
system.mem_ctrls.bw_inst_read::cpu.inst 7966632417 # Instruction read bandwidth from this memory (bytes/s)
system.mem_ctrls.bw_inst_read::total 7966632417 # Instruction read bandwidth from this memory (bytes/s)
system.mem_ctrls.bw_write::cpu.data 452877744 # Write bandwidth from this memory (bytes/s)
system.mem_ctrls.bw_write::total 452877744 # Write bandwidth from this memory (bytes/s)
system.mem_ctrls.bw_total::cpu.inst 7966632417 # Total bandwidth to/from this memory (bytes/s)
system.mem_ctrls.bw_total::cpu.data 4273491114 # Total bandwidth to/from this memory (bytes/s)
system.mem_ctrls.bw_total::total 12240123532 # Total bandwidth to/from this memory (bytes/s)
system.mem_ctrls.readReqs 0                   # Number of read requests accepted
system.mem_ctrls.writeReqs 0                   # Number of write requests accepted
system.mem_ctrls.readBursts 0                  # Number of DRAM read bursts, including those serviced by the write queue
system.mem_ctrls.writeBursts 0                 # Number of DRAM write bursts, including those merged in the write queue
system.mem_ctrls.bytesReadDRAM 0               # Total number of bytes read from DRAM
system.mem_ctrls.bytesReadWrQ 0                # Total number of bytes read from write queue
system.mem_ctrls.bytesWritten 0                # Total number of bytes written to DRAM
```

## Contribution

This tool will simulate the 32-bit ARM Executable file and create a stats file that will be used for calculating power using another tool McPAT.

---

## 5) GEM5toMcPAT

GEM5toMcPAT is a script which converts GEM5 simulation statistics to McPAT compatible inputs.

### Installation/Downloading

Just clone the repository like this

```
git clone https://bitbucket.org/dskhudia/gem5tomcpat.git
```

### Usage

Firstly, copy the stats file generated by Gem5 and copy to the gem5tomcpat folder.

```
cp m5out/stats.txt /home/saurabh/gem5tomcpat/statsfile.txt
```

The aim now is to convert this txt file to an XML file. Use the following commands for it:

```
cd gem5tomcpat/  
./GEM5ToMcPAT.py -o input.xml statsfile.txt config.json  
template-xeon.xml
```

This will create XML file with name input.xml in the gem5tomcpat folder.

### Contribution

Using the suitable input for McPAT, we can now provide input to McPAT which will estimate the power consumed by the program.

---

## 6) McPAT

McPAT (Multicore Power, Area, and Timing) is an integrated power, area, and timing modeling framework for multithreaded, multicore, and manycore architectures. It models power, area, and timing simultaneously and consistently and supports comprehensive early stage design space exploration for multicore and manycore processor configurations. McPAT has a flexible XML interface to facilitate its use with different performance simulators.

### Installation

Download the Source from <https://code.google.com/archive/p/mcpat/downloads>

Extract the file : `tar -xzf mcpat*.tar.gz`

Change the directory to McPAT08release

Run make : `make`

## Usage

Firstly, copy the XML File from gem5tomcpat folder to McPAT08release folder using the following command.

```
cp input.xml /home/<user name>/McPAT08release/
```

Then, run the tool to generate the power results

```
./mcpat -infile input.xml > result.txt
```

By using this command, the tool will take input.xml file as input and generate the result and store it in the result.txt file. This file now contains all the results of power consumed by the program during its course of execution.

-----  
The result.txt file looks like this

```
McPAT (version 0.8 of Aug, 2010) is computing the target processor...

McPAT (version 0.8 of Aug, 2010) results (current print level is 2, please increase print level to see the details in components):
*****
Technology 65 nm
Using Long Channel Devices When Appropriate
Interconnect metal projection= aggressive interconnect technology projection
Core clock Rate(MHz) 3400
*****

Processor:
Area = 398.552 mm^2
Peak Power = 93.2242 W
Total Leakage = 23.7461 W
Peak Dynamic = 69.4781 W
Subthreshold Leakage = 22.4102 W
Gate Leakage = 1.33588 W
Runtime Dynamic = 0.510817 W

Total Cores: 2 cores
Device Type= ITRS high performance device type
Area = 114.276 mm^2
Peak Dynamic = 52.0155 W
Subthreshold Leakage = 11.6018 W
Gate Leakage = 1.17902 W
Runtime Dynamic = 0.510541 W

Total L3s:
Device Type= ITRS high performance device type
Area = 278.843 mm^2
Peak Dynamic = 4.84476 W
Subthreshold Leakage = 10.7416 W
Gate Leakage = 0.144361 W
Runtime Dynamic = 6.76223e-05 W

Total NoCs (Network/Bus):
Device Type= ITRS high performance device type
Area = 5.43221 mm^2
Peak Dynamic = 12.6178 W
Subthreshold Leakage = 0.0668215 W
Gate Leakage = 0.0124914 W
Runtime Dynamic = 0.00020822 W
*****
```

With this, the initial phase i.e Execution Time and Power Survey is completed.

-----

## **7) LLVM**

The LLVM compiler infrastructure project (formerly Low Level Virtual Machine) is a "collection of modular and reusable compiler and toolchain technologies" used to develop compiler front ends and back ends.



LLVM is written in C++ and is designed for compile-time, link-time, run-time, and "idle-time" optimization of programs written in arbitrary programming languages. Originally implemented for C and C++, the language-agnostic design of LLVM has since spawned a wide variety of front ends: languages with compilers that use LLVM include ActionS, Python, R, Ruby, Rust, CUDA, Scala and Swift.

### Installation

---

```
mkdir llvm
cd llvm
wget http://llvm.org/releases/3.7.1/llvm-3.7.1.src.tar.xz
wget http://llvm.org/releases/3.7.1/cfe-3.7.1.src.tar.xz
wget http://llvm.org/releases/3.7.1/compiler-rt-3.7.1.src.tar.xz

tar xf ./llvm-3.7.1.src.tar.xz
tar xf ./cfe-3.7.1.src.tar.xz
tar xf ./compiler-rt-3.7.1.src.tar.xz

mv ./llvm-3.7.1.src ./llvm-3.7.1
mv ./cfe-3.7.1.src ./clang
mv ./clang ./llvm-3.7.1/tools/
mv ./compiler-rt-3.7.1.src ./compiler-rt
mv ./compiler-rt ./llvm-3.7.1/projects/

mkdir ./build
cd ./build
../llvm-3.7.1/configure
../llvm-3.7.1/configure -help

time make -j 3
make check-all

gedit ~/.bashrc
#(add export PATH=$PATH:$HOME/llvm/build/Release+Asserts/bin and
close)
sudo gedit /etc/ld.so.conf.d/llvm.conf
#/home/<my user name>/llvm/build/Release+Asserts/lib

make install
```

---

## Usage

LLVM uses clang as Front-end, so we can use this front-end to avail the facilities of the LLVM Compiler Collection.

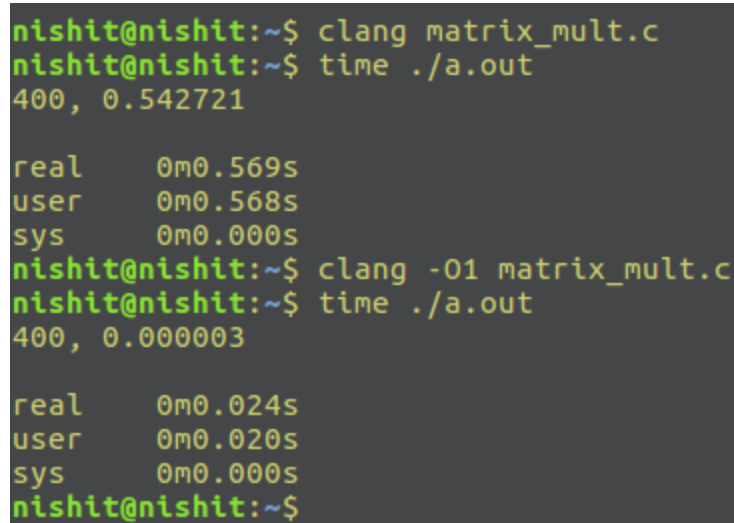
To compile the program, just type

```
clang <filename>
```

The file must be a C/C++ file.

You can use -o like just similar to gcc to direct the output to an output file. Also you can make use of inbuilt optimizations using the -Ox flags as described previously in case of gcc.

(x stands for Optimization Level - 0/1/2/3/fast)



```
nishit@nishit:~$ clang matrix_mult.c
nishit@nishit:~$ time ./a.out
400, 0.542721

real    0m0.569s
user    0m0.568s
sys     0m0.000s
nishit@nishit:~$ clang -O1 matrix_mult.c
nishit@nishit:~$ time ./a.out
400, 0.000003

real    0m0.024s
user    0m0.020s
sys     0m0.000s
nishit@nishit:~$
```

LLVM also allows to create byte code files (.bc format) and can directly execute those bytecode files.

To create a byte code file, type the following command :

```
clang -c -emit-llvm filename.c -o bytecode_file.bc
```

To run the **bytecode\_file.bc** file, type this on the Terminal:

```
lli bytecode_file.bc
```

Example:

```
nishit@nishit:~$ clang -c -emit-llvm matrix_mult.c -o matrix.bc
nishit@nishit:~$ time lli matrix.bc
400, 0.390440

real    0m0.424s
user    0m0.412s
sys     0m0.008s
nishit@nishit:~$
```

### Contribution

LLVM is an open-source software which allows modifications to be made to the intermediate code which is basically the application of Pass which user makes.

These passes can be either Optimization Passes, Analysis Passes, Transformation Passes etc.

LLVM can provide the middle layers of a complete compiler system, taking intermediate representation (IR) code from a compiler and emitting an optimized IR. This new IR can then be converted and linked into machine-dependent assembly language code for a target platform.

LLVM can accept the IR from the GNU Compiler Collection (GCC) toolchain, allowing it to be used with a wide array of extant compilers written for that project.

---

## **8) Writing an LLVM Pass**

The LLVM Pass Framework is an important part of the LLVM system, because LLVM passes are where most of the interesting parts of the compiler exist. Passes perform the transformations and optimizations that make up the compiler, they build the analysis results that are used by these transformations, and they are, above all, a structuring technique for compiler code.

All LLVM passes are subclasses of the Pass class, which implement functionality by overriding virtual methods inherited from Pass. Depending on how your pass works, you should inherit from the ModulePass, CallGraphSCCPass, FunctionPass, or LoopPass, or BasicBlockPass classes, which gives the system more information about what your pass does, and how it can be combined with other passes. One of the main features of the LLVM Pass Framework is that it schedules passes to run in an efficient way based on the constraints that your pass meets (which are indicated by which class they derive from).

---

We start by showing you how to construct a pass, everything from setting up the code, to compiling, loading, and executing it.

## Quick Start - Writing hello world

Here we describe how to write the "hello world" of passes. The "Hello" pass is designed to simply print out the name of non-external functions that exist in the program being compiled. It does not modify the program at all, it just inspects it. The source code and files for this pass are available in the LLVM source tree in the lib/Transforms/Hello directory.

### 1. Setting up the build environment:

First, you need to create a new directory. For this example, we'll assume that you made home/my\_pass. Next, you must set up a build script (Makefile) that will compile the source code for the new pass. To do this, copy the following into Makefile:

### # Makefile for my\_pass

```
-----
cmake_minimum_required(VERSION 3.1)
find_package(LLVM REQUIRED CONFIG)
add_definitions(${LLVM_DEFINITIONS})
include_directories(${LLVM_INCLUDE_DIRS})
link_directories(${LLVM_LIBRARY_DIRS})
add_subdirectory(arithmetic_pass) # Use your pass name here.
-----
```

### # Include the makefile implementation stuff in the arithmetic\_pass subfolder

```
-----
add_library(SkeletonPass MODULE
    # List your source files here.
    mypass.cpp
)

# Use C++11 to compile our pass (i.e., supply -std=c++11).
target_compile_features(SkeletonPass PRIVATE cxx_range_for cxx_auto_type)

# LLVM is (typically) built with no C++ RTTI. We need to match that;
# otherwise, we'll get linker errors about missing RTTI data.
set_target_properties(SkeletonPass PROPERTIES
    COMPILE_FLAGS "-fno-rtti"
)

# Get proper shared-library behavior (where symbols are not necessarily
# resolved when the shared library is linked) on OS X.
if(APPLE)
    set_target_properties(SkeletonPass PROPERTIES
        LINK_FLAGS "-undefined dynamic_lookup"
    )
endif(APPLE)
-----
```

This makefile specifies that all of the .cpp files in the current directory are to be compiled and linked together into a build/arithmetic\_pass/libSkeletonPass.so shared object that can be dynamically loaded by the opt or bugpoint tools via their -load options. If your operating system uses a suffix other than .so (such as windows or Mac OS/X), the appropriate extension will be used.

2. Now that we have the build scripts set up, we just need to write the code for the pass itself.

#### a) Basic code required

Now that we have a way to compile our new pass, we just have to write it. Start out with:

```
#include "llvm/Pass.h"
#include "llvm/Function.h"
```

Which are needed because we are writing a Pass, and we are operating on Function's.

Next we have:

```
using namespace llvm;
```

... which is required because the functions from the include files live in the llvm namespace.

#### b) Next we have:

```
namespace {
```

... which starts out an anonymous namespace. Anonymous namespaces are to C++ what the "static" keyword is to C (at global scope). It makes the things declared inside of the anonymous namespace only visible to the current file. If you're not familiar with them, consult a decent C++ book for more information.

#### c) Next, we declare our pass itself:

```
struct Hello : public FunctionPass {
```

This declares a "Hello" class that is a subclass of FunctionPass. The different builtin pass subclasses are described in detail later, but for now, know that FunctionPass's operate a function at a time.

```
    static char ID;
    Hello() : FunctionPass((intptr_t)&ID) {}
```

This declares pass identifier used by LLVM to identify pass. This allows LLVM to avoid using expensive C++ runtime information.

```
    virtual bool runOnFunction(Function &F) {
        llvm::cerr << "Hello: " << F.getName() << "\n";
```

```

        return false;
    }
}; // end of struct Hello

```

d) We declare a "runOnFunction" method, which overloads an abstract virtual method inherited from FunctionPass. This is where we are supposed to do our thing, so we just print out our message with the name of each function.

```
char Hello::ID = 0;
```

We initialize pass ID here. LLVM uses ID's address to identify pass so initialization value is not important.

```

    RegisterPass<Hello> X("hello", "Hello World Pass");
} // end of anonymous namespace

```

e) Lastly, we register our class Hello, giving it a command line argument "hello", and a name "Hello World Pass".

**As a whole, the .cpp file looks like:**

```

-----
#include "llvm/Pass.h"
#include "llvm/Function.h"
using namespace llvm;
namespace {
    struct Hello : public FunctionPass {
        static char ID;
        Hello() : FunctionPass((intptr_t)&ID) {}
        virtual bool runOnFunction(Function &F) {
            llvm::cerr << "Hello: " << F.getName() << "\n";
            return false;
        }
    };
    char Hello::ID = 0;
    RegisterPass<Hello> X("hello", "Hello World Pass");
}
-----

```

3. Now that it's all together, compile the file with a simple "gmake" command in the local directory and you should get a new "Debug/lib/Hello.so file. Note that everything in this file is contained in an anonymous namespace: this reflects the fact that passes are self contained units that do not need external interfaces (although they can have them) to be useful.

4. Running a pass with opt

Now that you have a brand new shiny shared object file, we can use the `opt` command to run an LLVM program through your pass. Because you registered your pass with the `RegisterPass` template, you will be able to use the `opt` tool to access it, once loaded.

To test it, follow the example at the end of the Getting Started Guide to compile "Hello World" to LLVM. We can now run the bitcode file (`hello.bc`) for the program through our transformation like this (or course, any bitcode file will work):

```
$ opt -load ../../my_pass/build/arithmetic_pass/libSkeletonPass.so
-hello < hello.bc > /dev/null
Hello: __main
Hello: puts
Hello: main
```

The `-load` option specifies that `opt` should load your pass as a shared object, which makes `-hello` a valid command line argument (which is one reason you need to register your pass). Because the hello pass does not modify the program in any interesting way, we just throw away the result of `opt` (sending it to `/dev/null`).

---

Example:

```
nishit@nishit:~$ cd my_pass/
nishit@nishit:~/my_pass$ mkdir build
nishit@nishit:~/my_pass$ cd build
nishit@nishit:~/my_pass/build$ cmake ..
-- The C compiler identification is GNU 5.4.0
-- The CXX compiler identification is GNU 5.4.0
-- Check for working C compiler: /usr/bin/cc
-- Check for working C compiler: /usr/bin/cc -- works
-- Detecting C compiler ABI info
-- Detecting C compiler ABI info - done
-- Detecting C compile features
-- Detecting C compile features - done
-- Check for working CXX compiler: /usr/bin/c++
-- Check for working CXX compiler: /usr/bin/c++ -- works
-- Detecting CXX compiler ABI info
-- Detecting CXX compiler ABI info - done
-- Detecting CXX compile features
-- Detecting CXX compile features - done
-- Configuring done
-- Generating done
-- Build files have been written to: /home/nishit/my_pass/build
nishit@nishit:~/my_pass/build$ make
Scanning dependencies of target SkeletonPass
[ 50%] Building CXX object arithmetic_pass/CMakeFiles/SkeletonPass.dir/nypass.cpp.o
[100%] Linking CXX shared module libSkeletonPass.so
[100%] Built target SkeletonPass
nishit@nishit:~/my_pass/build$ clang matrix_mult.c
clang: error: no such file or directory: 'matrix_mult.c'
clang: error: no input files
nishit@nishit:~/my_pass/build$ cd
nishit@nishit:~$ clang matrix_mult.c
nishit@nishit:~$ time ./a.out
400, 0.539161

real    0m0.563s
user    0m0.556s
sys      0m0.008s
nishit@nishit:~$ clang -c -emit-llvm matrix_mult.c -o matrix_multwithpass.bc
nishit@nishit:~$ opt -load my_pass/build/arithmetic_pass/libSkeletonPass.so -my-local-opts -disable-output matrix_multwithpass.bc
nishit@nishit:~$ time lli matrix_multwithpass.bc
400, 0.454424

real    0m0.478s
user    0m0.472s
sys      0m0.004s
nishit@nishit:~$
```