**ENPM617 FINAL PROJECT REPORT**

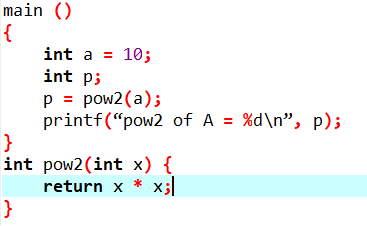
1. **PROJECT OVERVIEW:**

LLVM is the name of the compiler which is open-source software. The primary goal of this project is to implement a **function cloning pass** that can automatically transform indirect call-transfer instructions to direct unconditional control-transfer instructions. This is because indirect call-transfer instructions can slow on architectures that do not have suitable branch prediction hardware.

In this project, we have to clone the function which starts with the letter “p” and the clone function should not have any return value.

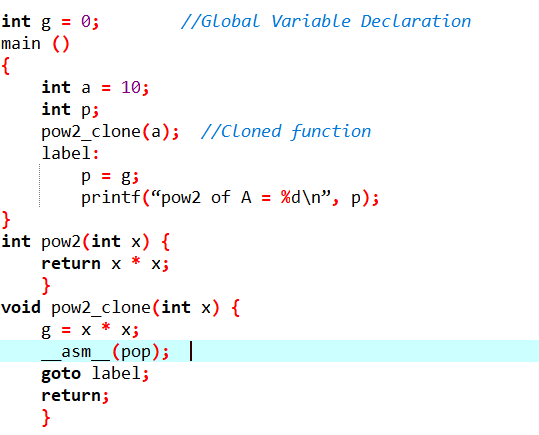
**Sample C program:**

Let us consider the sample C program which is operating on indirect call-transfer instructions.



**fig 1.1**

The above-mentioned C program has to transform to direct control-transfer instructions by cloning the function **pow2()**.



**fig 1.2**

*\_asm\_(pop)* and *goto label* instructions are replaced by using *pop\_direct\_branch()*. This function provides the exact same functionality of pop instruction followed by the goto statement.

1. **IMPLEMENTATION OF FUNCTION CLONING PASS:**

**2.1) Algorithm:**

*STEP 1****:*** Start

*STEP 2*: Declare a global variable to store the return value of the function which is cloned

*STEP 3*: Iterate over every instruction in the module and check for call instructions.

*STEP 4*: Get the called function.

*STEP 5*: Clone the called function if the called function name starts with the letter “p” and not a declaration otherwise *goto* Step

*STEP 6*: Create a new store instruction that transfers the output of the return value of the cloned instruction into the global variable.

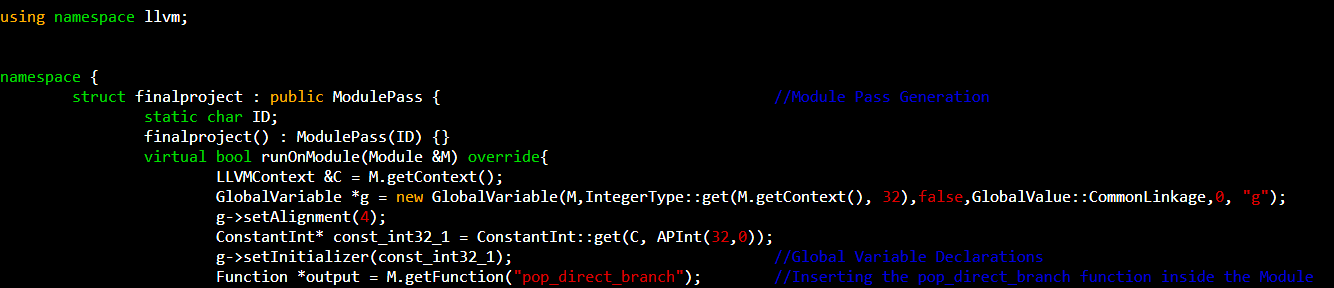
*STEP 7*: Call the pop\_direct\_branch function

*STEP 9*: Create a new Load instruction for loading the return value from the global variable to the appropriate variable.

*STEP 10*: Replace the call site with the newly cloned function.

**2.2) Pass Generation**

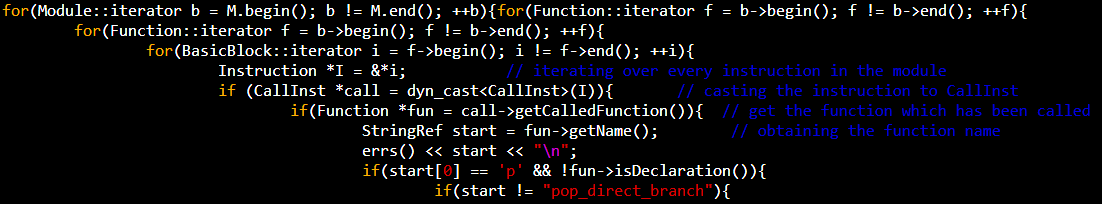
1) Consider the *ModulePass* class where we are considering an entire program as a unit and modifying the module by the transformation using *virtual bool* *runOnModule(Module &M).*



Taking the Module as M new GlobalVariable needs to be inserted into the module. The appropriate constructor and required attributes for creating a new Global variable are extracted by converting the .c file of the global variable declaration to the .cpp file.

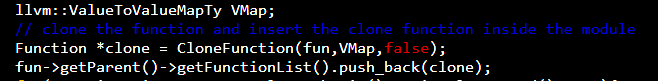
Since the function *pop\_direct\_branch* has to be inserted into the module. So, *M.getFunction()* will get the function from the module.

2) To check every instruction for the call function inside the module, we need to iterate over instructions in the module and cast every instruction to CallInst to check whether it is a call instruction.

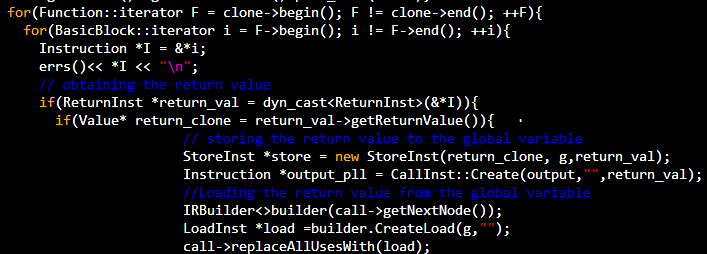


It is required to get the called function and check whether the name of the function starts with the letter “p” or whether the function is a declaration. *getCalledFunction()* obtains the function which has been called and *getName()* gives the name of the function.

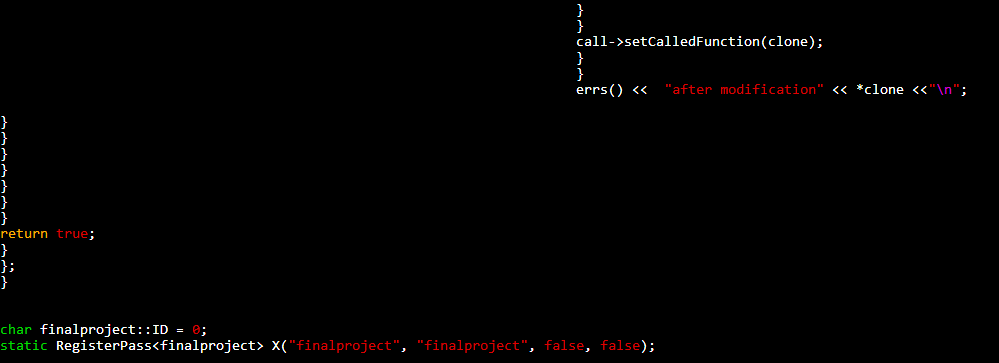
1. Clone the function and add the cloned function into the module. *CloneFunction()* clones the function. *getParent()* obtains the module to which the original function belongs and it is required to get the list of functions inside the module by *getFunctionList()*. *push\_back()* is used to push the cloned function inside the module.



1. After cloning the function, it is required to check for return instruction in the clone function by iterating over instructions in the function and storing the return value in the global variable. This is done by casting every instruction to *ReturnInst* class and obtaining the return value by *getReturnValue().* *StoreInst* class is used to store the return value to the global value. Load the return value from the global variable into the appropriate variable after where the call instruction is cloned in the original function. *IRBuilder* is used for creating the load instruction through *CreateLoad().*



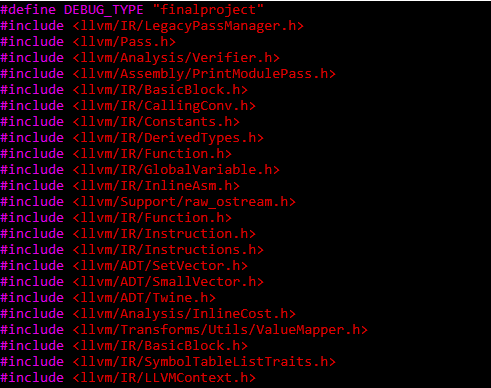
Replace all the uses using the new load instruction using *replaceAllUsesWith()*

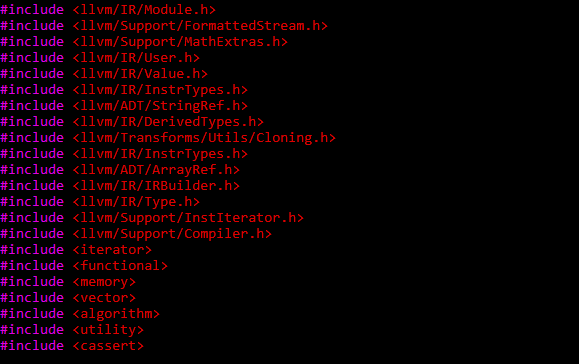
1. Since we have replaced the old function with the new cloned function. It is required to set the call to a new cloned function instead of the old function. *setCalledFunction()* is used to set the call from the old function to the new cloned function. 

As the pass is transformed, we need to *return true.* The pass has to be registered for the implementation of the changes to the user programs.

**3)LIBRARIES USED**

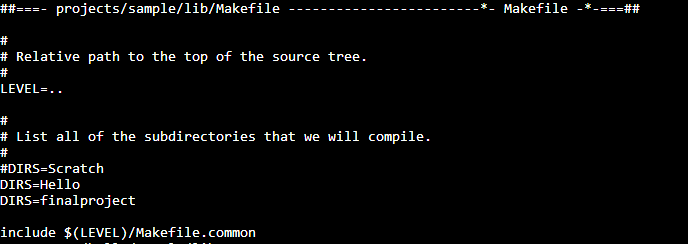
The following libraries are used for extracting the appropriate class and constructors for completing the project.



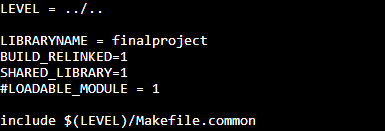


**4**) **MODIFIED MAKE FILE FOR COMPILING THE PASS**

The project sub-directory must be included in the make file for compiling the pass and a new make file has been created for the program generated.



*DIRS= finalproject* is the added sub-directory in the make file for compiling the pass.



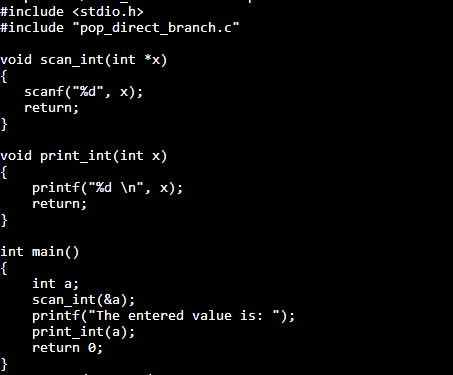
Created make file for the program to get executed with the appropriate requirements.

**4) RESULTS:**

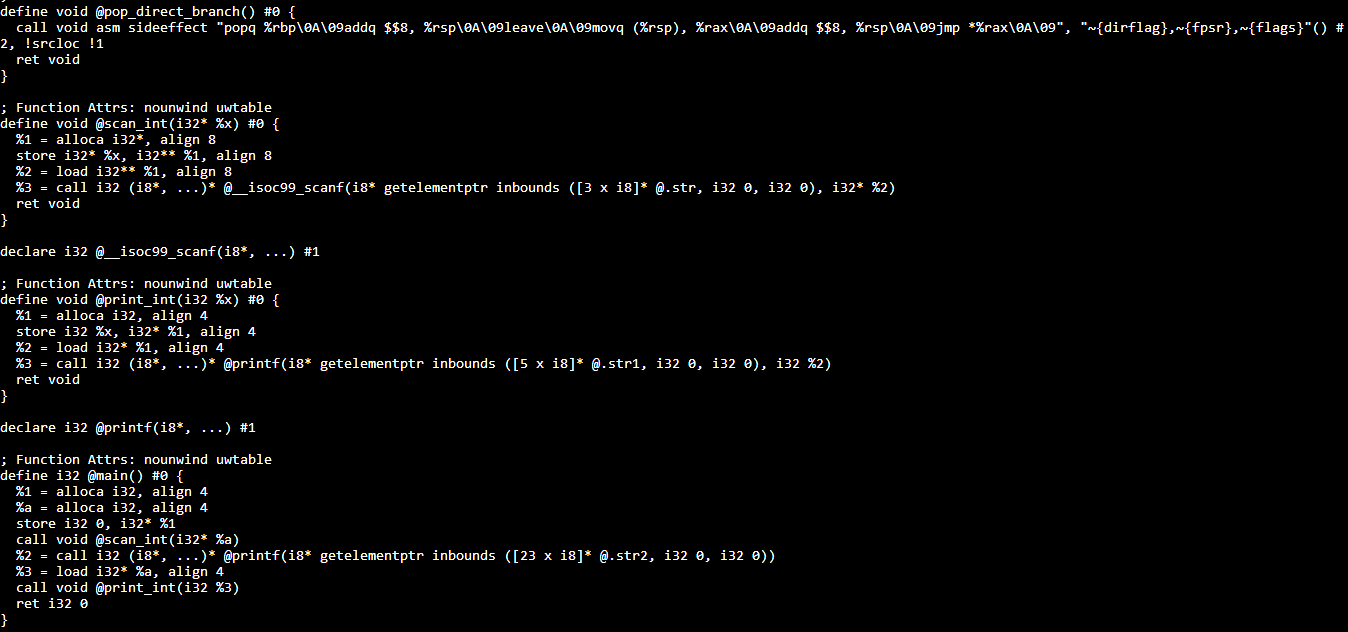
We have loaded the program in the test cases provided on the GLUE machine. We have extracted all the modified .ll files for all the test cases provided.

For demonstrating our results, we are considering two of the test cases provided and the remaining test cases replicate the same approach.

**Example 0:**

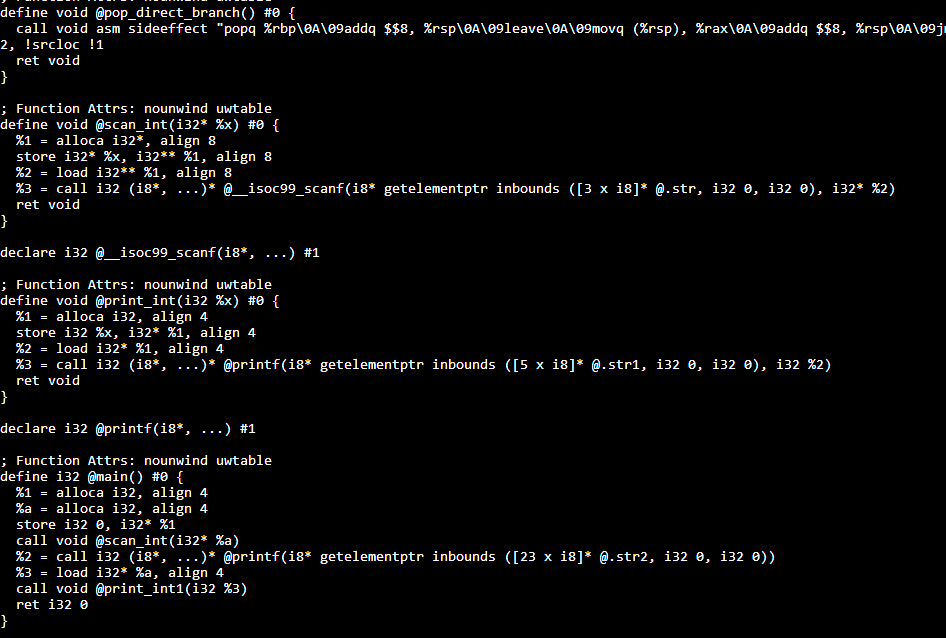
****

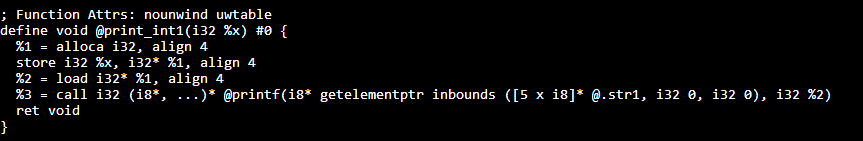
The .c file of example0 is provided above.



.ll file for example0

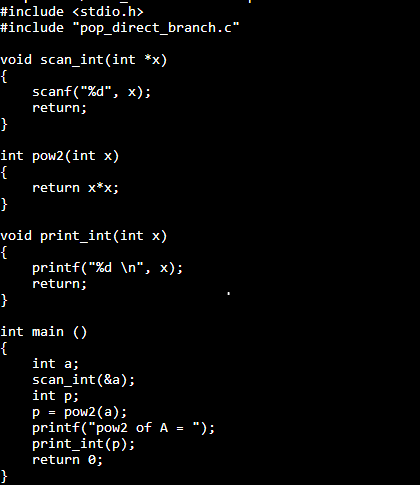
The example0 test case has four functions including pop\_direct\_branch. According to our algorithm. Cloning of the function should be expected only when the function name starts with the letter “p” and it is not a declaration. After cloning the function if the function does not expect any return value there is no necessity for the addition of instructions.



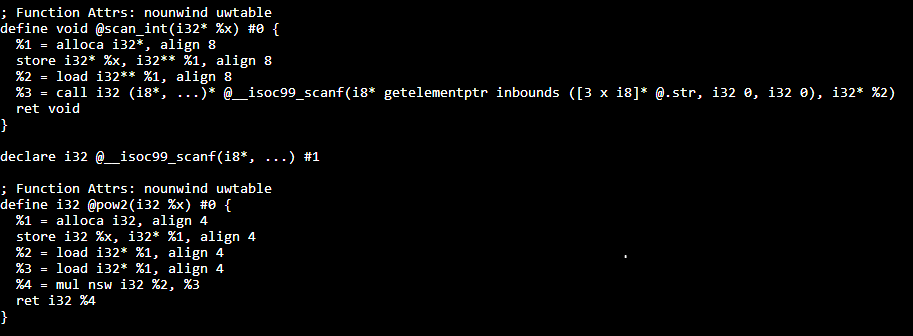


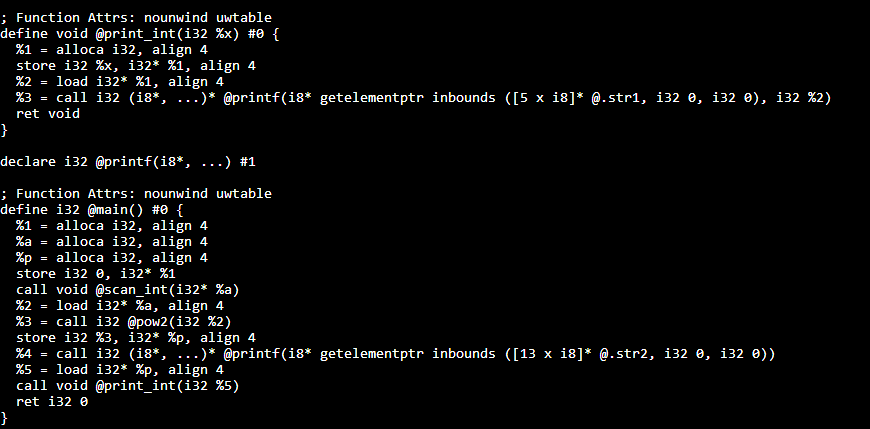
The above code is the modified .ll file where there is no change in the old .ll file to the new .ll file. *print\_int()* is the function before cloning and *print\_int1()* is the function cloned. Since there is no return value for *print\_int()* function no instruction has been added.

**Example1:**



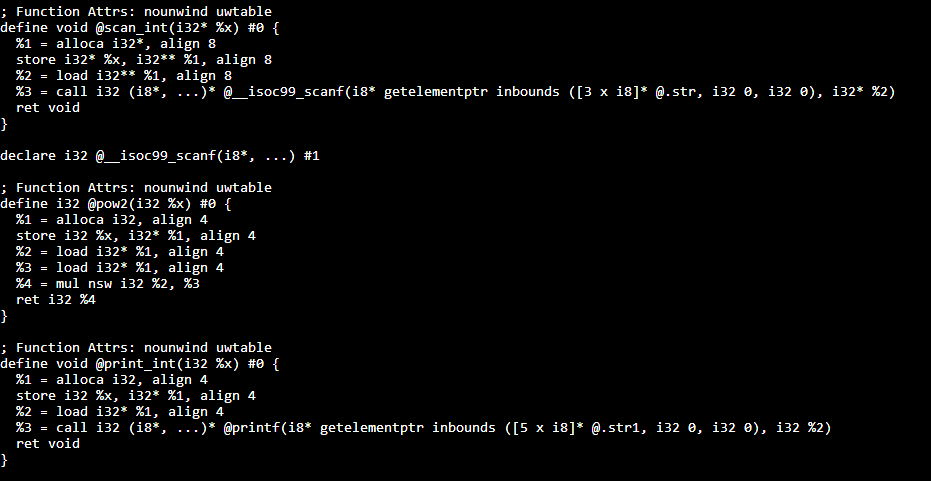
.c file of Example1

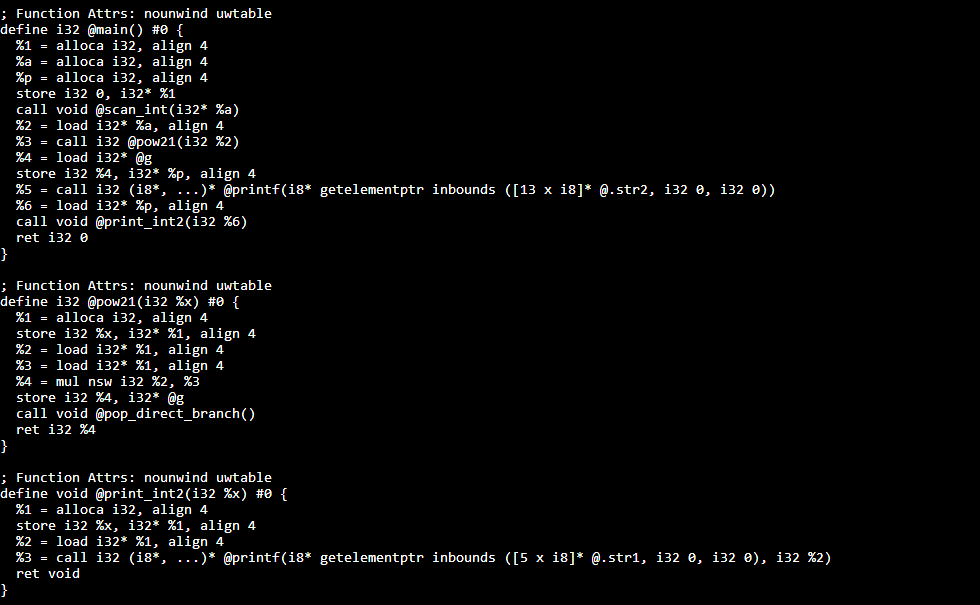




.ll of Example1

The example1 test case has five functions including pop\_direct\_branch . There are two functions starting with the letter “p” i.e..., pow2() and print\_int(). The print\_int() does not return any value. So, there is no addition of any new instructions into the function print\_int(). As pow2() has a return value we need to add the store and load instruction for storing the return value into the global variable and reading the value from the global variable by calling the function pop\_direct\_branch().





Modified .ll file of Example1

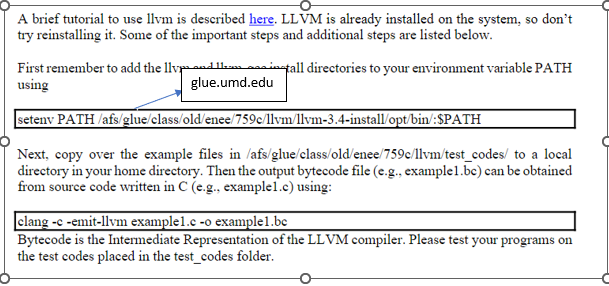
The above modified .ll file shows the cloned functions *pow21()* and *print\_int2()*. Since the *pow21()* function has a return value. The return value is stored inside the global variable (*store i32 %4, i32\* @g)*. After the store instruction, we need to call the function *pop\_direct\_branch() (call void @pop\_direct\_branch()).* We intended to read the value from the global variable so load instruction is included in the main function*(%4 = Load i32\* @g).*

As the *print\_int2()* does not have any return value, no instruction has been inserted.

**5) ISSUES**

Below are the few issues we encountered during the execution of this project in the GLUE machine

1. The *setenv* path in the project document has to be modified



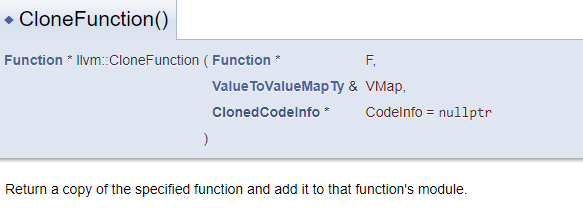
1. Since there is no Legacy pass manager in the installed LLVM in the GLUE machine. The iteration of instructions is performed by converting the iterator into instruction with the following command.



The above iteration can be easily implemented in the latest llvm using **auto.**

1. The related doxygen files and class description on the internet have been modified according to the LLVM16 and many methods, and constructors have been added and changed in LLVM16.

According to the doxygen file of cloneFunction() the description states that generated clone will be added to that function’s module, LLVM3.4 does not add the clone function into the module instead we are required to add the function into the module manually. Since we are assuming obtaining the constructors according to the doxygen files they are providing some misconceptions while implementing the program in the GLUE machine.



Manual addition in LLVM 3.4 is done by following the below approach.

