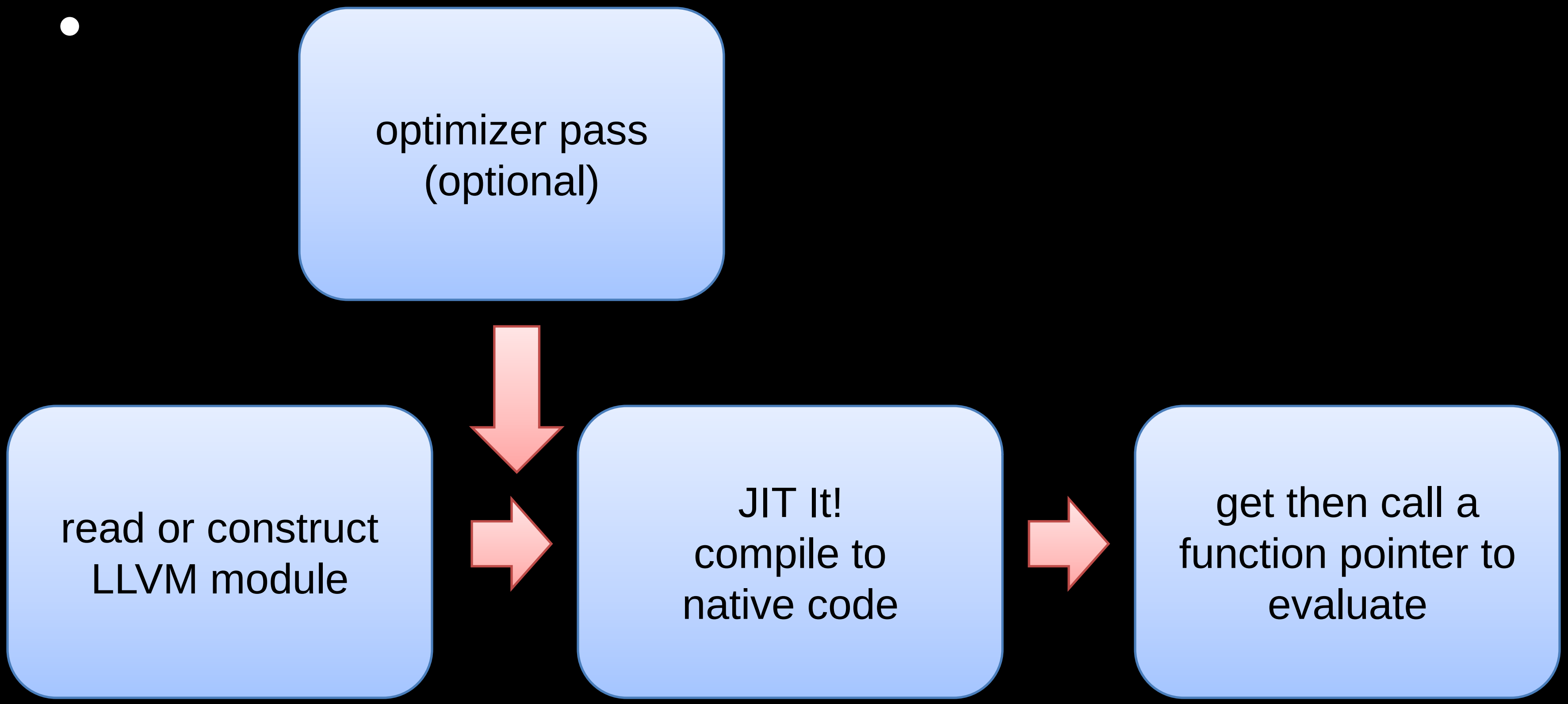
How To Use JIT
&
Make JIT Support

Agenda

- How to use JIT
  - Construct Module
  - Read Module
  - JIT & Execute
- Make JIT support

How to use JIT

-



What is LLVM module

- LLVM programs are composed of "Module"s
- a translation unit of the input programs
- consists of functions, global variables, and symbol table entries

"hello world" module

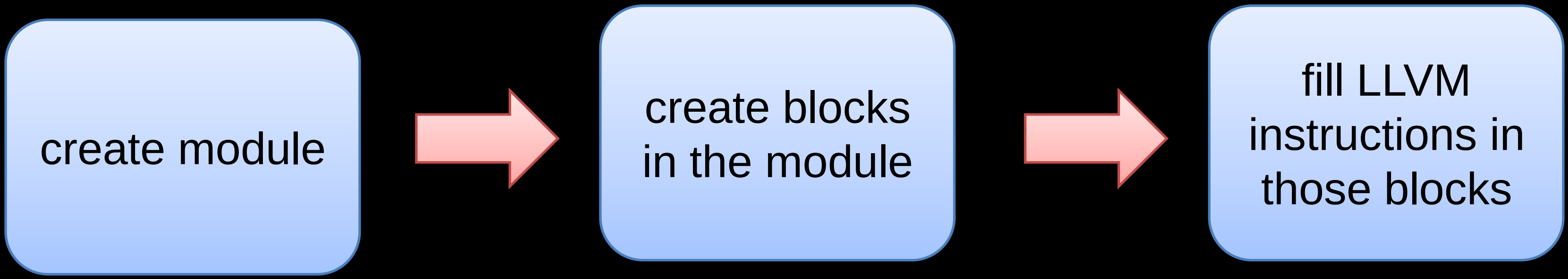
```
; Declare the string constant as a global constant...
@.LC0 = internal constant [13 x i8] c"hello world\0A\00" ;

; External declaration of the puts function
declare i32 @puts(i8 *) ; i32(i8 *)*

; Definition of main function
define i32 @main() {
    ; Convert [13x i8 ]* to i8 *...
    %cast210 = getelementptr [13 x i8 ]* @.LC0, i64 0, i64 0

    ; Call puts function to write out the string to stdout...
    call i32 @puts(i8 * %cast210) ; i32 ret i32 0
}
```

Construct LLVM module



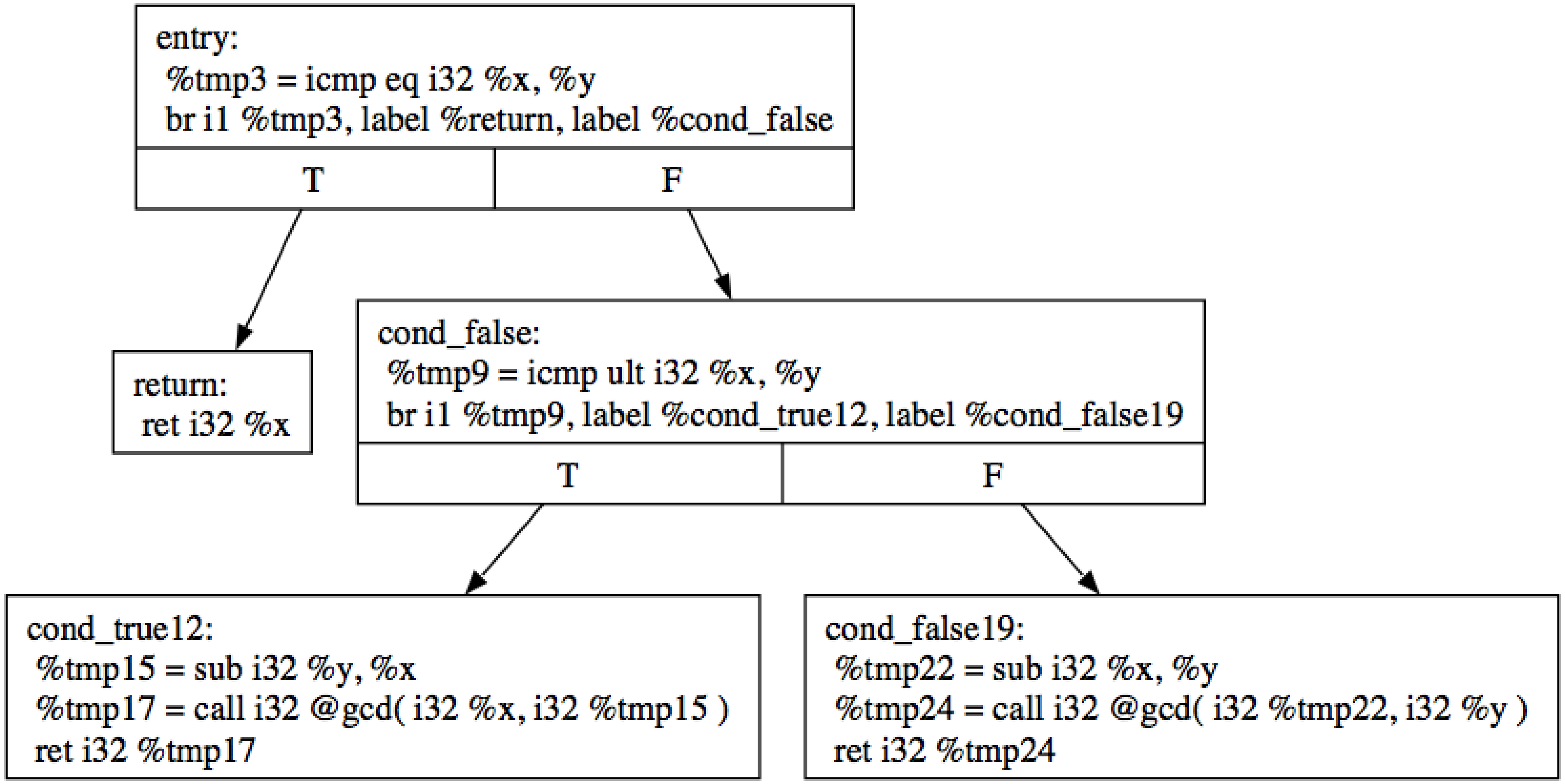
Greatest Common Divisor (GCD) algorithm

```
unsigned gcd( unsigned x, unsigned y ) {

    if ( x == y ) {
        return x;
    } else if ( x < y ) {
        return gcd( x, y - x );
    } else {
        return gcd( x - y, y);
    }

}
```

CFG for ‘gcd’ function



CFG for 'gcd' function

LLVM IR

```
; ModuleID = 'tut2'

define i32 @gcd(i32 %x, i32 %y) {
entry:
        %tmp = icmp eq i32 %x, %y               ; <i1> [#uses=1]
        br i1 %tmp, label %return, label %cond_false

return:         ; preds = %entry
        ret i32 %x

cond_false:             ; preds = %entry
        %tmp2 = icmp ult i32 %x, %y             ; <i1> [#uses=1]
        br i1 %tmp2, label %cond_true, label %cond_false1

cond_true:              ; preds = %cond_false
        %tmp3 = sub i32 %y, %x          ; <i32> [#uses=1]
        %tmp4 = call i32 @gcd(i32 %x, i32 %tmp3)                ; <i32> [#uses=1]
        ret i32 %tmp4

cond_false1:            ; preds = %cond_false
        %tmp5 = sub i32 %x, %y          ; <i32> [#uses=1]
        %tmp6 = call i32 @gcd(i32 %tmp5, i32 %y)                ; <i32> [#uses=1]
        ret i32 %tmp6
}
```

#include the appropriate LLVM header files

```
#include <llvm/Module.h>

#include <llvm/Function.h>

#include <llvm/PassManager.h>

#include <llvm/Analysis/Verifier.h>

#include <llvm/Assembly/PrintModulePass.h>

#include <llvm/Support/IRBuilder.h>

using namespace llvm;
```

main()

```
Module* makeLLVMModule();  // do the real work of creating the module

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

    Module* Mod = makeLLVMModule();  // creating a module

    verifyModule(*Mod, PrintMessageAction);  // verifying it

    PassManager PM;
    PM.add(new PrintModulePass(&llvm::cout));
    PM.run(*Mod);  // running the PrintModulePass on it

    delete Mod;
    return 0;
}
```

verifyModule()

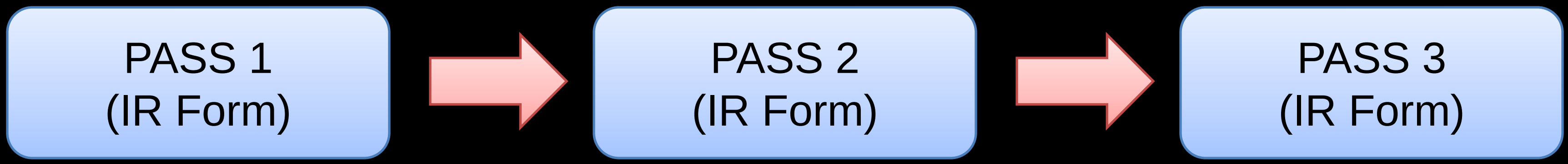
- print an error message if your LLVM module is malformed

```
%x = add i32 1, %x
```

- the definition of %x does not dominate all of its uses, NOT a legal SSA form (every variable is assigned exactly once)

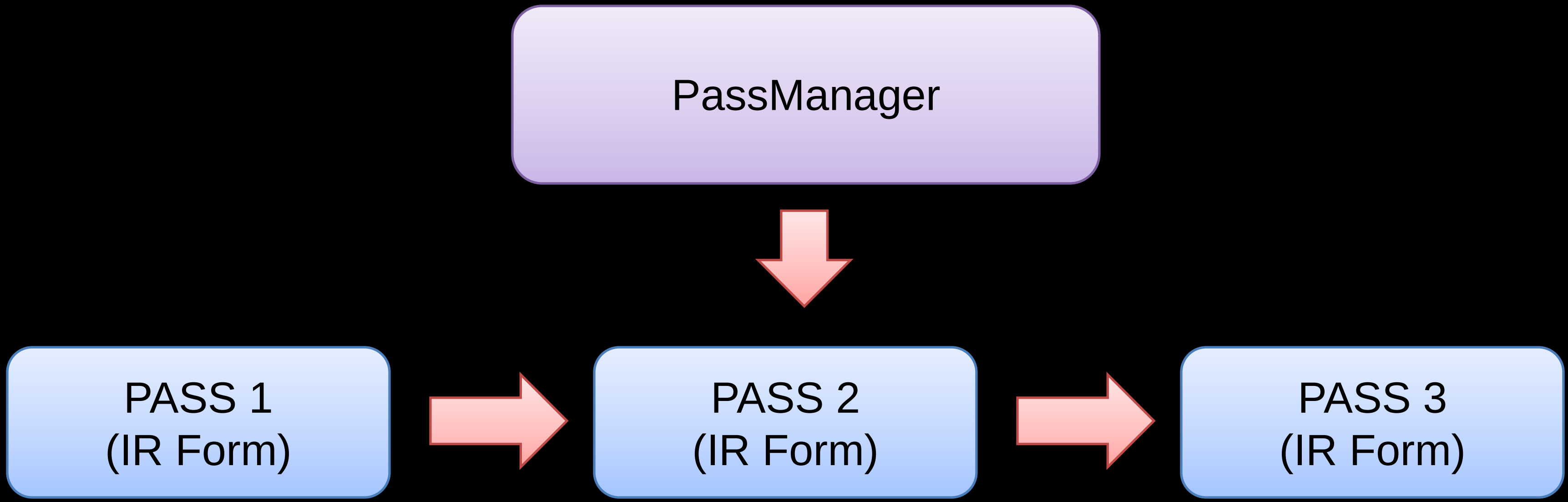
What is a pass?

- perform the transformations and optimizations that make up the compiler
- build the analysis results that are used by these transformations



What PassManager does

- takes a list of passes, ensures their prerequisites are set up correctly, and then schedules passes to run efficiently



PrintModulePass

- a pass that prints out the entire module when it is executed

```
PM.add(new PrintModulePass(&llvm::cout));
```

- add this pass and schedule it

makeLLVMModule()

```
Module* makeLLVMModule() {
    Module* mod = new Module("tut2");

    // getOrInsertFunction() doesn't actually return a Function*
    // it will return a cast of the existing function if the function
    // already existed with a different prototype
    Constant* c = mod->getOrInsertFunction("gcd",
    /*ret type*/                          IntegerType::get(32),
    /*args*/                              IntegerType::get(32),
                                          IntegerType::get(32),
    /*varargs terminated with null*/      NULL);

    Function* gcd = cast<Function>(c);
```

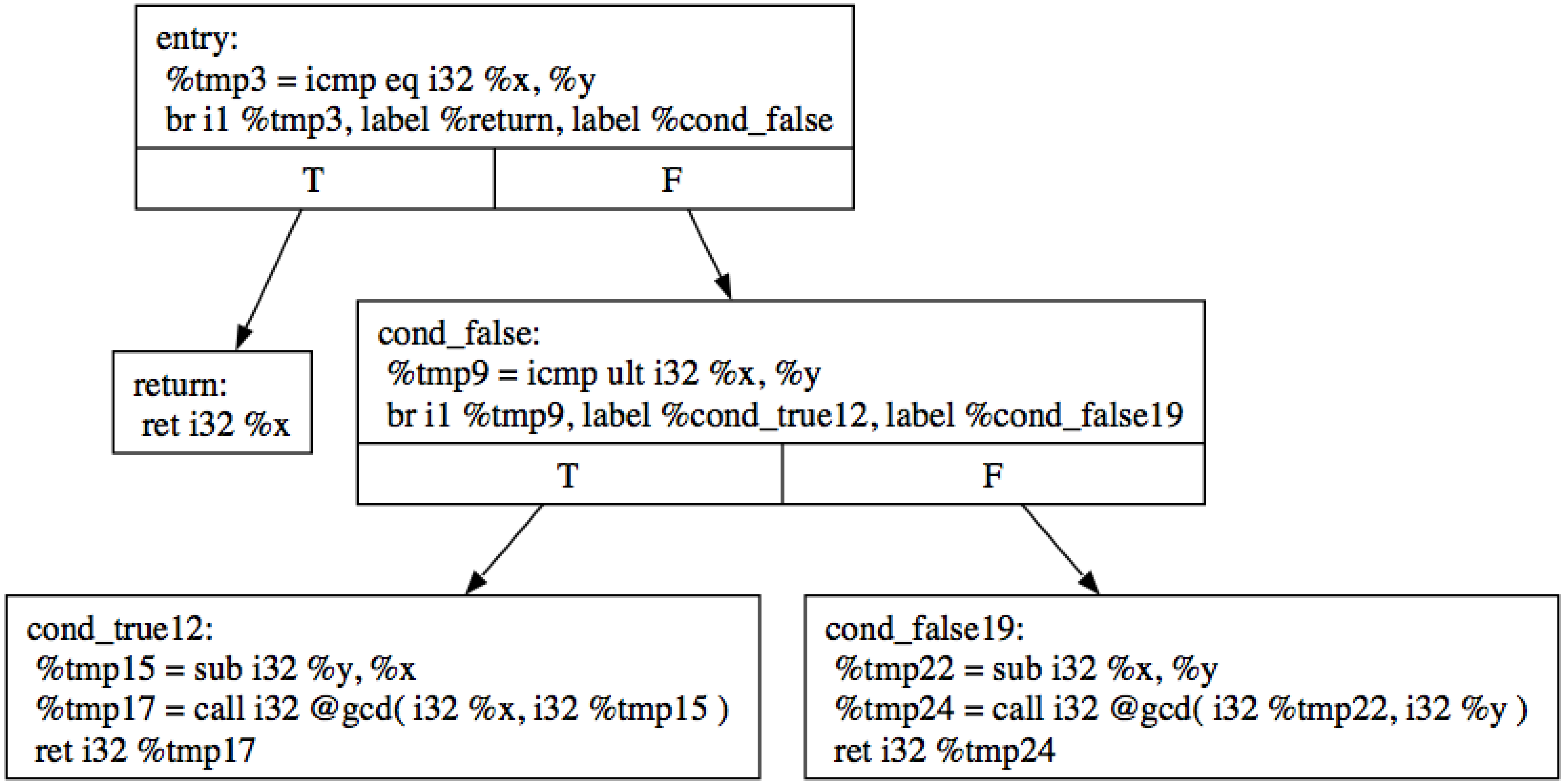
makeLLVMModule()

```
// make looking at our output somewhat more
// pleasant
Function::arg_iterator args = gcd->arg_begin();
Value* x = args++;
x->setName("x");
Value* y = args++;
y->setName("y");
```

makeLLVMModule()

```
// a block is basically a set of instructions that can be branched to
// and is executed linearly until the block is terminated by one of a
// small number of control flow instructions, such as br or ret
// we're making use of LLVM's automatic name uniquing
BasicBlock* entry = BasicBlock::Create("entry", gcd);
BasicBlock* ret = BasicBlock::Create("return", gcd);
BasicBlock* cond_false = BasicBlock::Create("cond_false", gcd);
BasicBlock* cond_true = BasicBlock::Create("cond_true", gcd);
BasicBlock* cond_false_2 = BasicBlock::Create("cond_false", gcd);
```

CFG for ‘gcd’ function



CFG for 'gcd' function

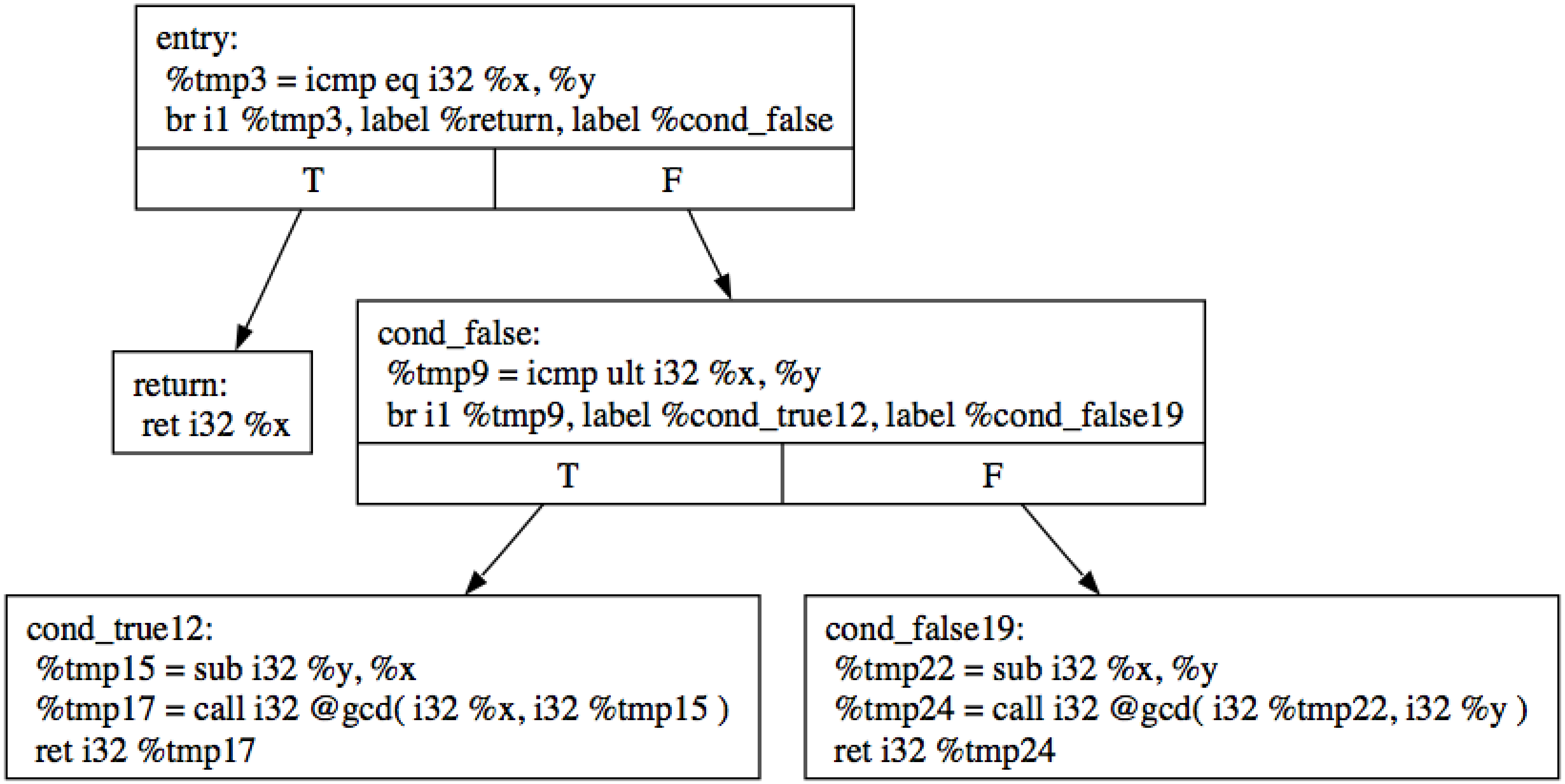
makeLLVMModule()

```
IRBuilder<> builder(entry); // working on "entry" block
```

```
entry:
        %tmp = icmp eq i32 %x, %y                    ; <i1> [#uses=1]
        br i1 %tmp, label %return, label %cond_false
```

```
Value* xEqualsY = builder.CreateICmpEQ(x, y, "tmp");
builder.CreateCondBr(xEqualsY, ret, cond_false);
```

CFG for ‘gcd’ function



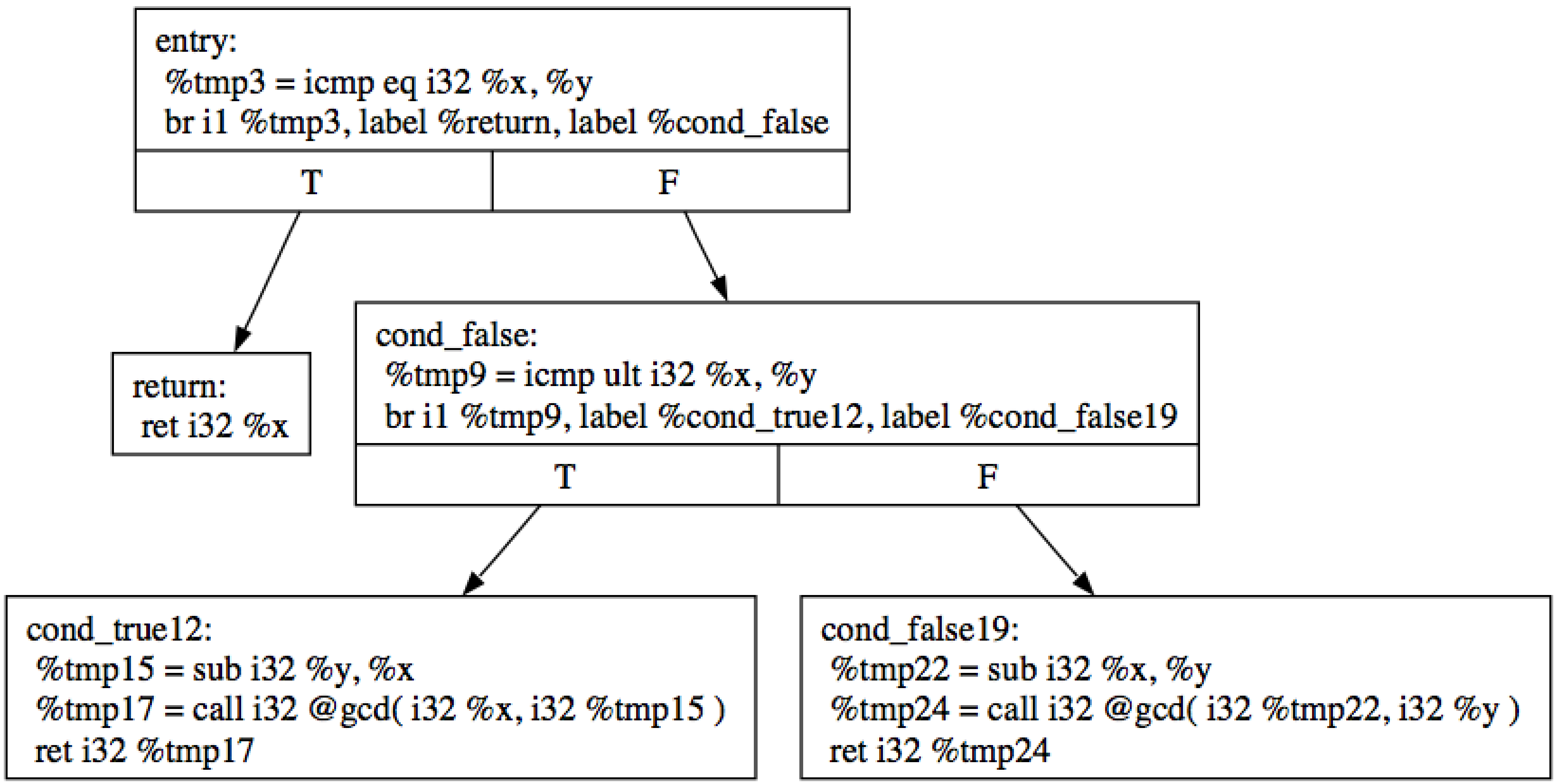
CFG for 'gcd' function

makeLLVMModule()

```
return:                ; preds = %entry
        ret i32 %x
```

```
// use SetInsertPoint to retarget our current block
builder.SetInsertPoint(ret);
builder.CreateRet(x);
```

CFG for ‘gcd’ function



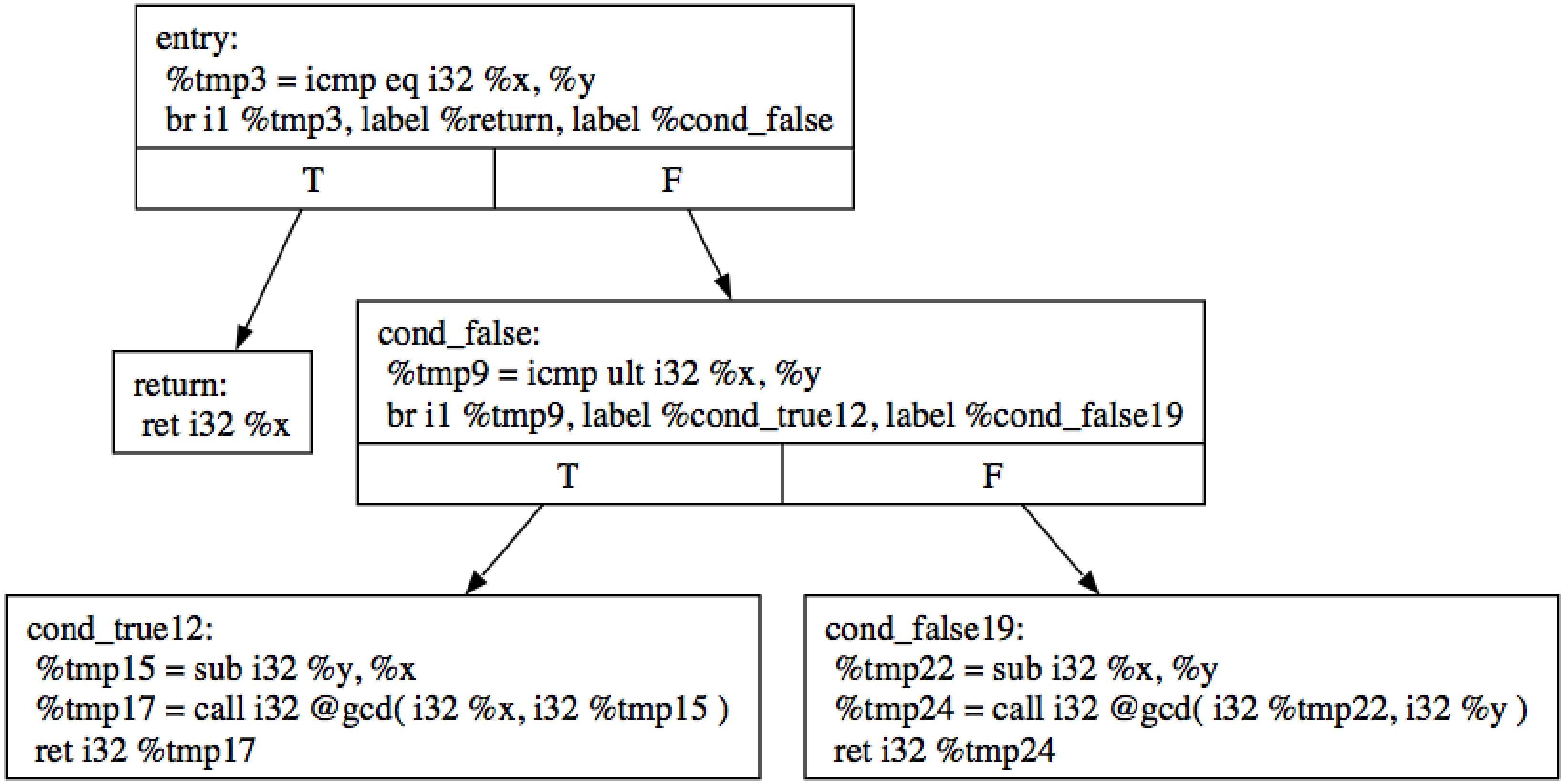
CFG for 'gcd' function

makeLLVMModule()

```
cond_false:                    ; preds = %entry
        %tmp2 = icmp ult i32 %x, %y                        ; <i1> [#uses=1]
        br i1 %tmp2, label %cond_true, label %cond_false1
```

```
// In LLVM, integer types do not carry sign
// Whether a signed or unsigned interpretation is desired is
// specified in the instruction
builder.SetInsertPoint(cond_false);
Value* xLessThanY = builder.CreateICmpULT(x, y, "tmp");
builder.CreateCondBr(xLessThanY, cond_true, cond_false_2);
```

CFG for ‘gcd’ function



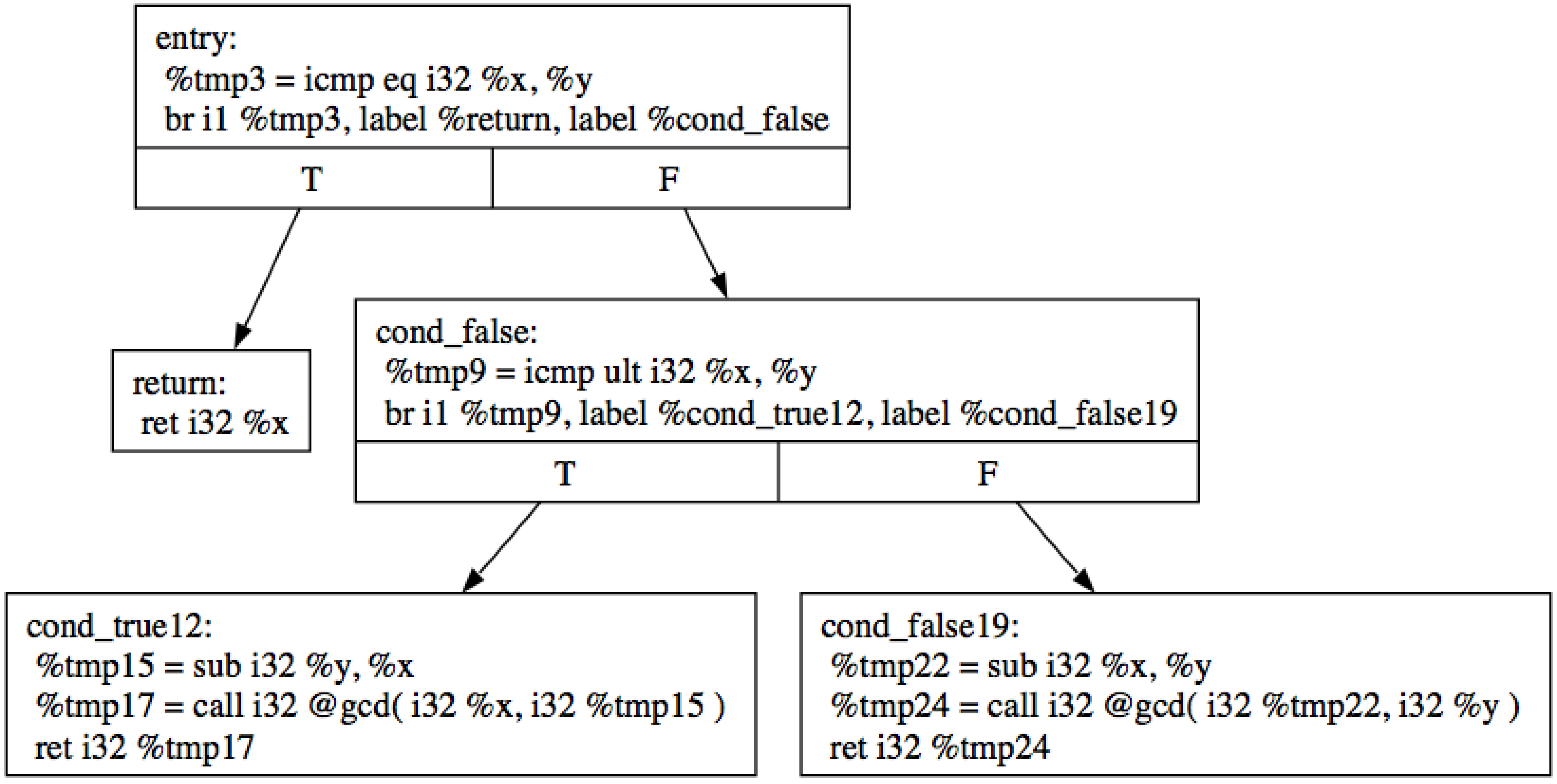
CFG for 'gcd' function

makeLLVMModule()

```
cond_true:                    ; preds = %cond_false
        %tmp3 = sub i32 %y, %x            ; <i32> [#uses=1]
        %tmp4 = call i32 @gcd(i32 %x, i32 %tmp3); <i32> [#uses=1]
        ret i32 %tmp4
```

```
// To create a call instruction, we have to create a vector (or any
// other container with InputInterators) to hold the arguments.
// We then pass in the beginning and ending iterators for this vector
builder.SetInsertPoint(cond_true);
Value* yMinusX = builder.CreateSub(y, x, "tmp");
std::vector<Value*> args1;
args1.push_back(x);
args1.push_back(yMinusX);
Value* recur_1 = builder.CreateCall(gcd, args1.begin(), args1.end(), "tmp");
builder.CreateRet(recur_1);
```

CFG for ‘gcd’ function



CFG for 'gcd' function

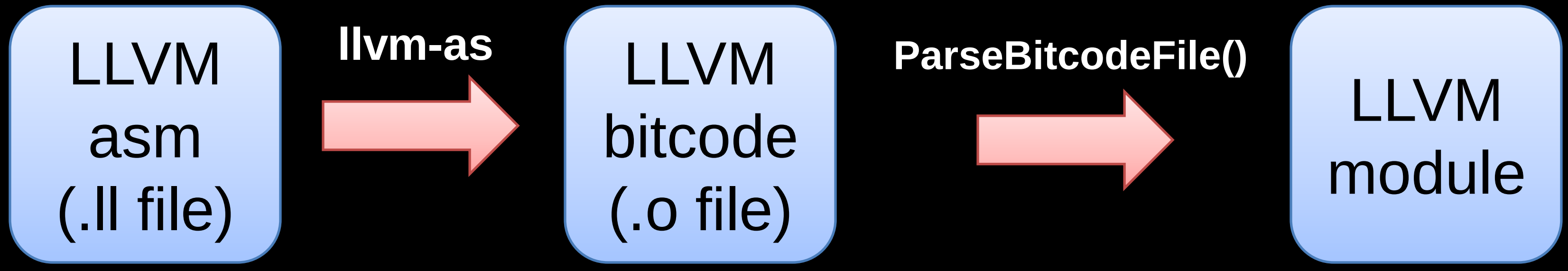
makeLLVMModule()

```
cond_false1:                    ; preds = %cond_false
        %tmp5 = sub i32 %x, %y                  ; <i32> [#uses=1]
        %tmp6 = call i32 @gcd(i32 %tmp5, i32 %y); <i32> [#uses=1]
        ret i32 %tmp6
```

```
  Value* xMinusY = builder.CreateSub(x, y, "tmp");
  std::vector<Value*> args2;
  args2.push_back(xMinusY);
  args2.push_back(y);
  Value* recur_2 = builder.CreateCall(gcd, args2.begin(), args2.end(), "tmp");
  builder.CreateRet(recur_2);

  return mod;
}
```

Read LLVM Module



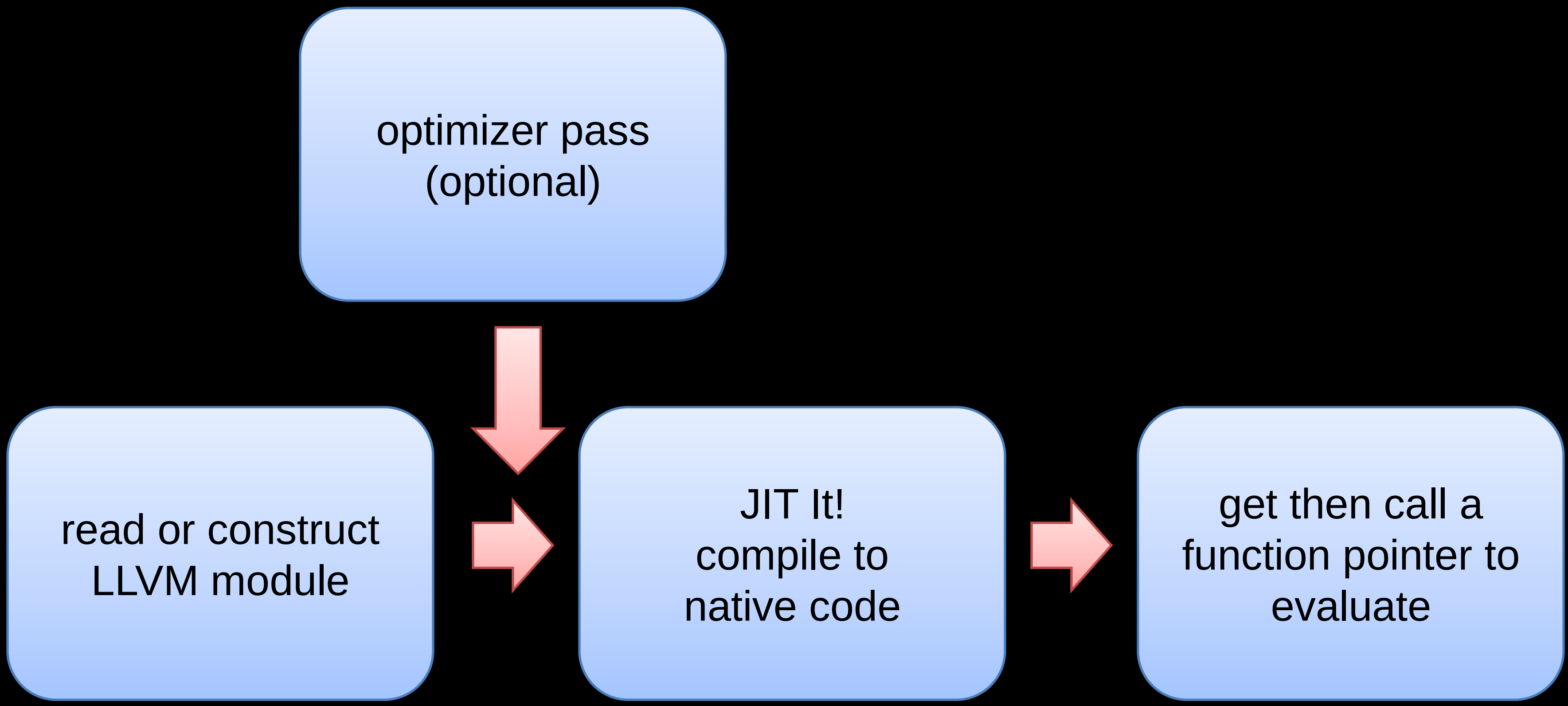
#include the appropriate LLVM header files

```
#include <llvm/Support/MemoryBuffer.h>
#include <llvm/Bitcode/ReaderWriter.h>
```

makeLLVMModule()

```
// create module from bitcode file
Module* makeLLVMModule() {
    std::string error;
    Module* Mod = 0;
    // load in the bitcode file containing the functions
    if( MemoryBuffer* buffer =
                    MemoryBuffer::getFile( "ops.o", &error ) ) {
        // read the specified bitcode file, returning the module
        Mod = ParseBitcodeFile( buffer, &error );
        delete buffer;
    }
    return Mod;
}
```

JIT & Execute


optimizer pass
(optional)
read or construct
LLVM module
JIT It!
compile to
native code
get then call a
function pointer to
evaluate

ExecutionEngine

- a key module to use JIT facility with LLVM
- can compile a function in the module into the native code
- provide a interface to call JIT-ted function

#include the appropriate LLVM header files

```
#include <llvm/ExecutionEngine/JIT.h>

#include <llvm/ModuleProvider.h>

#include <llvm/ExecutionEngine/GenericValue.h>
```

llvm/ExecutionEngine/JIT.h

- force the JIT to link in on certain operating systems (Windows)
- llvm/ExecutionEngine/ExecutionEngine.h do the real work
  - define the abstract interface that implements execution support for llvm

llvm/ModuleProvider.h

- an abstract interface for loading a module from some place
- allow incremental or random access loading of functions from the file

llvm/ExecutionEngine/GenericValue.h

- represent an LLVM value of arbitrary type

main()

```
// Now we create the JIT
// Create a ExecutinEngine for module Mod
ExistingModuleProvider* MP = new ExistingModuleProvider(Mod);
ExecutionEngine* EE = ExecutionEngine::create(MP, false);

// Get a (LLVM) function to be JIT-ted from module M
Function *GCD = cast<Function>( Mod->getOrInsertFunction("gcd",
            Type::Int32Ty,
            Type::Int32Ty,
            Type::Int32Ty,
            NULL) );
```

main()

```
// Call the function with arguments
std::vector<GenericValue> Args(2);
Args[0].IntVal = APInt(32, 10);
Args[1].IntVal = APInt(32, 5);

// Compile a function and execute it
// Provides function argument through GenericValue array
GenericValue gv = EE->runFunction(GCD, Args);
```

compile & run

```
$ c++ -g tut2.cpp `llvm-config --cxxflags --ldflags --libs engine` -o tut2

$ ./tut2
```

JIT Support

- JIT code generator emits machine code and auxiliary structures as binary output that can be written directly to memory
- currently Alpha, ARM, PowerPC, and X86 have JIT support

XXXCodeEmitter.cpp

- contain a machine function pass that transforms target-machine instructions (assembly) into relocatable machine code (binary)

```
// relocate function in XXXJITInfo.cpp will fill in the actual address
// of SYMBOL
jump      SYMBOL
       ...(omit)
SYMBOL
```

XXXJITInfo.cpp

- implement the JIT interfaces for target-specific code-generation activities
- minimum implementation
  - getLazyResolverFunction
  - emitFunctionStub
  - relocate
  - callback

getLazyResolverFunction

- initialize the JIT, gives the target a function that is used for compilation
- trivial implementation - make the parameter as the global JITCompilerFunction and returns the callback function that will be used a function wrapper

```
// ARMJITInfo.cpp

TargetJITInfo::LazyResolverFn
ARMJITInfo::getLazyResolverFunction(JITCompilerFn F) {
  JITCompilerFunction = F;
  return ARMCompilationCallback;
}
```

http://llvm.org/doxygen/ARMJITInfo_8cpp-source.html

callback

```
// ARMJITInfo.cpp
void ARMCompilationCallback(void);  // the callback as a wrapper
  asm(
      //store register before calling JIT
      "stmdb sp!, {r0, r1, r2, r3, lr}\n“

      "mov r0, lr\n"
      "sub sp, sp, #4\n"
      // call the C portion of the callback
      "bl " ASMPREFIX "ARMCompilationCallbackC\n“
      "add sp, sp, #4\n"
      // restore register after calling JIT
      "ldr r0, [sp,#20]\n"
      "ldr r1, [sp,#16]\n"
      "str r1, [sp,#20]\n"
      "str r0, [sp,#16]\n"
      "ldmia sp!, {r0, r1, r2, r3, lr, pc}\n"
```

http://llvm.org/doxygen/ARMJITInfo_8cpp-source.html

emitFunctionStub

```
// ARMJITInfo.cpp
// returns a native function with a specified address for a callback function
void *ARMJITInfo::emitFunctionStub(const Function* F, void *Fn,
                                                     MachineCodeEmitter &MCE) {
  // If this is just a call to an external function, emit a branch instead of a
  // call.  The code is the same except for one bit of the last instruction.
  if (Fn != (void*)(intptr_t)ARMCompilationCallback) {
    // branch to the corresponding function addr
    // the stub is 8-byte size and 4-aligned
  } else {
    // branch and link to the compilation callback
    // the stub is 16-byte size and 4-aligned
}
                                                   ... (omit)
void ARMCompilationCallback(void);
```

http://llvm.org/doxygen/ARMJITInfo_8cpp-source.html

relocate

```
// ARMJITInfo.cpp

// changes the addresses of referenced globals, based on relocation types
/// relocate - Before the JIT can run a block of code that has been emitted,
/// it must rewrite the code to contain the actual addresses of any
/// referenced global symbols.
void ARMJITInfo::relocate(void *Function, MachineRelocation *MR,
                          unsigned NumRelocs, unsigned char* GOTBase) {
    for (unsigned i = 0; i != NumRelocs; ++i, ++MR) {
                               ...(omits)
    switch ((ARM::RelocationType)MR->getRelocationType()) {
        case ARM::reloc_arm_relative: {
        }
        case ARM::reloc_arm_branch: {
        }
    }
```

http://llvm.org/doxygen/ARMJITInfo_8cpp-source.html

XXXTargetMachine

- add getJITInfo method that return a TargetJITInfo object

```
// AlphaTargetMachine.h

virtual AlphaJITInfo* getJITInfo() {
    return &JITInfo;
}
```

http://llvm.org/doxygen/ARMJITInfo_8cpp-source.html

References

- LLVM Tutorial
- LLVM Language Reference Manual
- LLVM - 2.0 and beyond!
- Writing an LLVM Pass
- Writing an LLVM Compiler Backend
- Adding LLVM JIT – Syoyo Fujita
- http://llvm.org/devmtg/2008-08-23/llvm_first_steps.pdf
- http://blogger.godfat.org/2008/10/llvm-rubinius.html
- $(LLVM_SRC_ROOT)/examples/HowToUseJIT
- $(LLVM_SRC_ROOT)/lib/Target