# LLVM Cheat Sheet

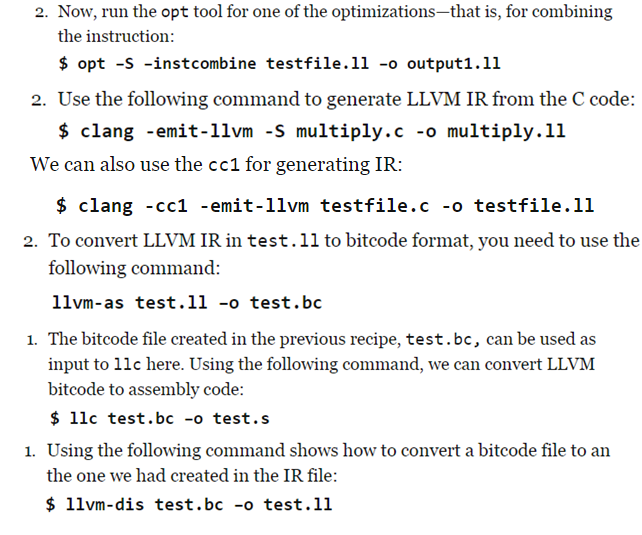
Identifiers

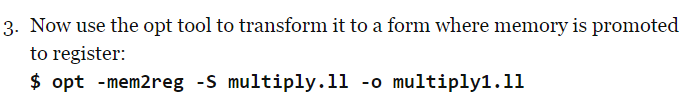
Globals start with @

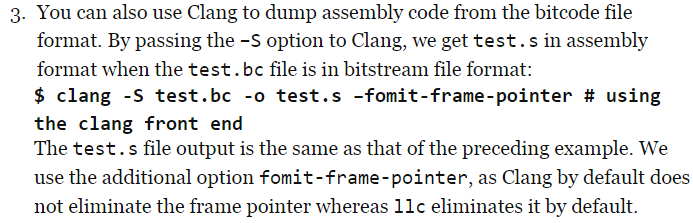
‘=’ does not mean assignment in the bytecode

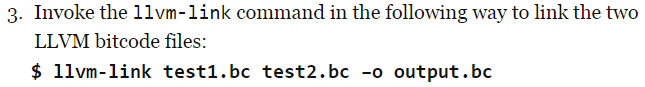
Install dtrace with ‘sudo apt-get install systemtap-sdt-dev’

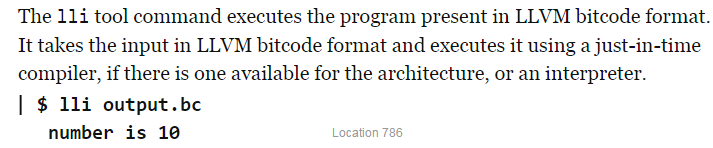
# LLVM Tools

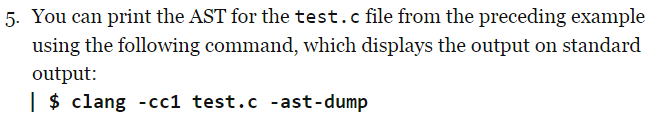




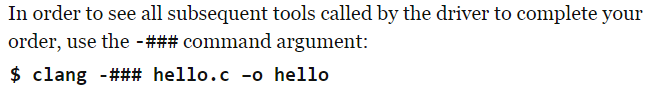






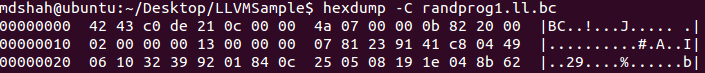


See which passes are being run at each optimization level when running opt with 

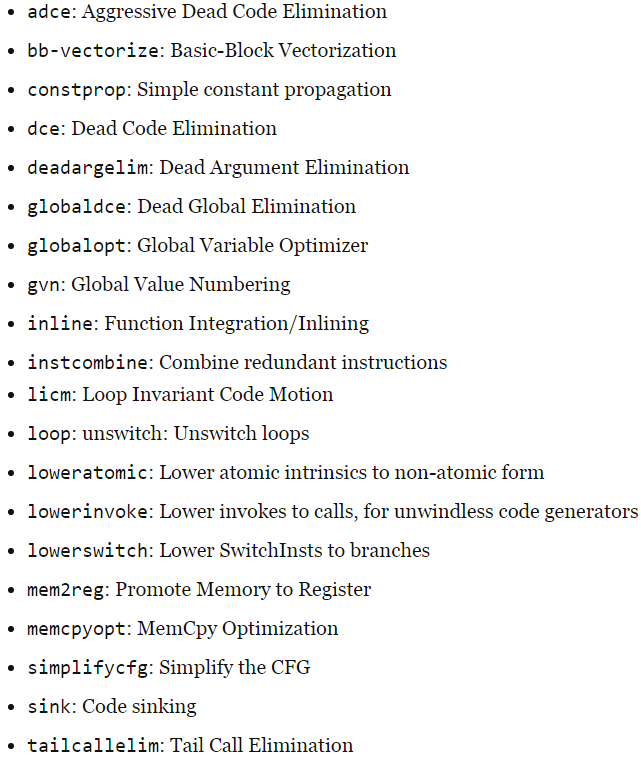


## Hexdump

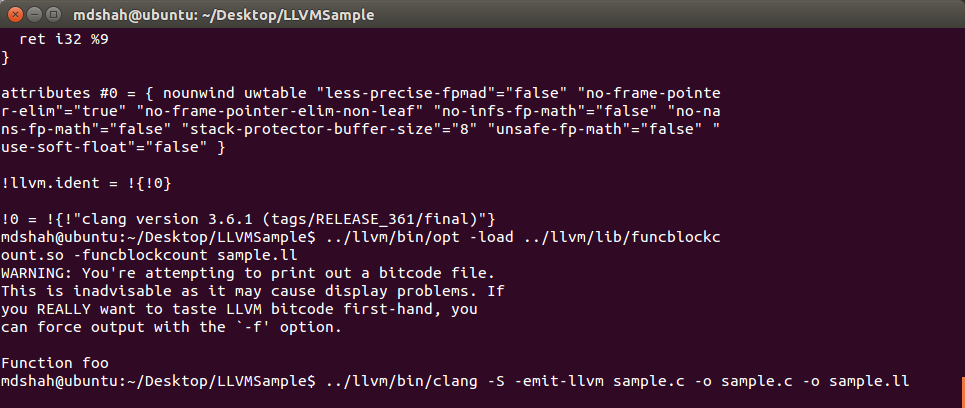
Output Canonical form



# List of LLVM Optimizations with opt



# Loading a custom optimization pass



## Running my custom function



* -oad specifies which shared library to load “funcblockcount.so”
* funcblockcount is the pass name that is registered   
  (e.g. , this pass would be called with ‘must-aa’)
* sample.ll is the bitcode to run.

# Setting up CSmith

Homepage: <http://embed.cs.utah.edu/csmith/>

1. Make sure git is install on your system.
2. Create a new directory somewhere
3. Run ‘git clone <https://github.com/csmith-project/csmith.git>’
4. cd into the root folder where you made a new directory (from step 2)
5. Then run ‘./configure’
   1. I got an error initially about not having a few Unix programs (note I was running on a virtual machine, so this is expected):
      1. ‘sudo apt-get install m4’
      2. ‘sudo apt-get install autoconf’
      3. Run ‘./configure’ once more.
6. Then run ‘make’
7. From the CSmith folder, you should then be able to find an executable in the /src folder.

# Help for using CSmith

## Command line options:

--help or -h: print this information.  
 -hh: describe extra options probably useful only for Csmith developers.  
 --version or -v: print the version of Csmith.  
 --seed <seed> or -s <seed>: use <seed> instead of a random seed generated by Csmith.  
 --output <filename> or -o <filename>: specify the output file name.  
 --argc | --no-argc: genereate main function with/without argv and argc being passed (enabled by default).  
 --arrays | --no-arrays: enable | disable arrays (enabled by default).  
 --bitfields | --no-bitfields: enable | disable full-bitfields structs (disabled by default).  
 --checksum | --no-checksum: enable | disable checksum calculation (enabled by default).  
 --comma-operators | --no-comma-operators: enable | disable comma operators (enabled by default).  
 --compound-assignment | --no-compound-assignment: enable | disable compound assignments (enabled by default).  
 --concise: generated programs with minimal comments (disabled by default).  
 --consts | --no-consts: enable | disable const qualifier (enabled by default).  
 --divs | --no-divs: enable | disable divisions (enabled by default).  
 --embedded-assigns | --no-embedded-assigns: enable | disable embedded assignments as sub-expressions (enabled by default).  
 --pre-incr-operator | --no-pre-incr-operator: enable | disable pre ++ operator (enabled by default).  
 --pre-decr-operator | --no-pre-decr-operator: enable | disable pre -- operator (enabled by default).  
 --post-incr-operator | --no-post-incr-operator: enable | disable post ++ operator (enabled by default).  
 --post-decr-operator | --no-post-decr-operator: enable | disable post -- operator (enabled by default).  
 --unary-plus-operator | --no-unary-plus-operator: enable | disable + operator (enabled by default).  
 --jumps | --no-jumps: enable | disable jumps (enabled by default).  
 --longlong| --no-longlong: enable | disable long long (enabled by default).  
 --int8 | --no-int8: enable | disable int8\_t (enabled by default).  
 --uint8 | --no-uint8: enable | disable uint8\_t (enabled by default).  
 --float | --no-float: enable | disable float (disabled by default).  
 --main | --nomain: enable | disable to generate main function (enabled by default).  
 --math64 | --no-math64: enable | disable 64-bit math ops (enabled by default).  
 --inline-function | --no-inline-function: enable | disable inline attributes on generated functions.  
 --inline-function-prob <num>: set the probability of each function being marked as inline (default is 50).  
 --max-array-dim <num>: limit array dimensions to <num>. (default 3)  
 --max-array-len-per-dim <num>: limit array length per dimension to <num> (default 10).  
 --max-block-depth <num>: limit depth of nested blocks to <num> (default 5).  
 --max-block-size <size>: limit the number of non-return statements in a block to <size> (default 4).  
 --max-expr-complexity <num>: limit expression complexities to <num> (default 10).  
 --max-funcs <num>: limit the number of functions (besides main) to <num> (default 10).  
 --max-pointer-depth <depth>: limit the indirect depth of pointers to <depth> (default 2).

--max-struct-fields <num>: limit the number of struct fields to <num> (default 10).   
 --max-union-fields <num>: limit the number of union fields to <num> (default 5).   
 --muls | --no-muls: enable | disable multiplications (enabled by default).  
 --safe-math | --no-safe-math: Emit safe math wrapper functions (enabled by default).  
 --packed-struct | --no-packed-struct: enable | disable packed structs by adding #pragma pack(1) before struct definition (disabled by default).  
 --paranoid | --no-paranoid: enable | disable pointer-related assertions (disabled by default).  
 --pointers | --no-pointers: enable | disable pointers (enabled by default).  
 --quiet: generate programs with less comments (disabled by default).  
 --structs | --no-structs: enable | disable to generate structs (enable by default).  
 --unions | --no-unions: enable | disable to generate unions (enable by default).  
 --volatiles | --no-volatiles: enable | disable volatiles (enabled by default).  
 --volatile-pointers | --no-volatile-pointers: enable | disable volatile pointers (enabled by default).  
 --const-pointers | --no-const-pointers: enable | disable const pointers (enabled by default).  
 --builtins | --no-builtins: enable | disable to generate builtin functions (disabled by default).  
 --enable-builtin-kinds k1,k2 | --disable-builtin-kinds k1,k2: enable | disable certain kinds of builtin functions.  
 --builtin-function-prob <num>: set the probability of choosing a builtin function (default is 20).  
 --lang-cpp : generate C++ code (C by default).

### 'Advanced' command line options that are probably only useful for Csmith's

### original developers:

--max-split-files <num>: evenly split a generated program into <num> different files(default 0).  
 --split-files-dir <dir>: generate split-files into <dir> (default ./output).  
 --dfs-exhaustive: enable depth first exhaustive random generation (disabled by default).  
 --expand-struct: enable the expansion of struct in the exhaustive mode. Only works in the exhaustive mode and cannot used with --no-structs.  
 --compact-output: print generated programs in compact way. Only works in the exhaustive mode.  
 --max-nested-struct-level <num>: limit maximum nested level of structs to <num>(default 0). Only works in the exhaustive mode.  
 --struct-output <file>: dump structs declarations to <file>. Only works in the exhaustive mode.  
 --prefix-name: prefix names of global functions and variables with increasing numbers. Only works in the exhaustive mode.  
 --sequence-name-prefix: prefix names of global functions and variables with sequence numbers.Only works in the exhaustive mode.  
 --compatible-check: disallow trivial code such as i = i in random programs. Only works in the exhaustive mode.  
 --msp: enable certain msp related features   
 --ccomp: generate compcert-compatible code  
 --splat: enable splat extension  
 --klee: enable klee extension  
 --crest: enable crest extension  
 --coverage-test: enable coverage-test extension  
 --coverage-test-size <num>: specify size (default 500) of the array generated to test coverage. Can only be used with --coverage-test.  
 --func1\_max\_params <num>: specify the number of symbolic variables passed to func\_1 (default 3). Only used when --splat | --crest | --klee | --coverage-test is enabled.  
 --fixed-struct-fields: fix the size of struct fields to max-struct-fields (default 10).  
 --return-structs | --no-return-structs: enable | disable return structs from a function (enabled by default).  
 --arg-structs | --no-arg-structs: enable | disable structs being used as args (enabled by default).  
 --return-unions | --no-return-unions: enable | disable return unions from a function (enabled by default).  
 --arg-unions | --no-arg-unions: enable | disable unions being used as args (enabled by default).

--take-union-field-addr | --take-no-union-field-addr: allow | disallow addresses of union fields to be taken (allowed by default).  
 --vol-struct-union-fields | --no-vol-struct-union-fields: enable | disable volatile struct/union fields (enabled by default)  
 --delta-monitor [simple]: specify the type of delta monitor. Only [simple] type is supported now.  
 --delta-input [file]: specify the file for delta input.  
 --delta-output [file]: specify the file for delta output (default to <delta-input>).  
 --go-delta [simple]: run delta reduction on <delta-input>.  
 --no-delta-reduction: output the same program as <delta-input>. Only works with --go-delta option.  
 --dump-default-probabilities <file>: dump the default probability settings into <file>  
 --dump-random-probabilities <file>: dump the randomized probabilities into <file>  
 --probability-configuration <file>: use probability configuration <file>  
 --random-random: enable random probabilities.  
 --enable-access-once: enable testing access once macro.  
 --strict-volatile-rule: make sure only one volatile access between any pair of sequence points.   
 --addr-taken-of-locals: enable addr-taken of local vars. [default]  
 --no-addr-taken-of-locals: disable addr-taken of local vars.   
 --fresh-array-ctrl-var-names: create fresh names [i1,i2,i3...] for array control vars rather than use unique names such as i, j, k.  
 --math-notmp: make csmith generate code for safe\_math\_macros\_notmp.  
 --strict-const-arrays: restrict array elements to constants.  
 --partial-expand <assignment[,for[,block[,if-else[,invoke[,return]]]]]: partial-expand controls which what kind of statements should be generated first. For example, it could make Csmith start to generate if-else without go over assignment or for.  
 --dangling-global-pointers | --no-dangling-global-pointers: enable | disable to reset all dangling global pointers to null at the end of main. (enabled by default)  
 --check-global: print the values of all integer global variables.  
 --monitor-funcs <name1,name2...>: dump the checksums after each statement in the monitored functions.  
 --step-hash-by-stmt: dump the checksum after each statement. It is applied to all functions unless --monitor-funcs is specified.  
 --stop-by-stmt <num>: try to stop generating statements after the statement with id <num>.  
 --const-as-condition: enable const to be conditions of if-statements.   
 --match-exact-qualifiers: match exact const/volatile qualifiers for LHS and RHS of assignments.  
 --reduce <file>: reduce random program under the direction of the configuration file.  
 --return-dead-pointer | --no-return-dead-pointer: allow | disallow functions from returning dangling pointers (disallowed by default).  
 --identify-wrappers: assign ids to used safe math wrappers.  
 --safe-math-wrappers <id1,id2...>: specifiy ids of wrapper functions that are necessary to avoid undefined behaviors, use 0 to specify none.  
 --mark-mutable-const: mark constants that can be mutated with parentheses (disabled by default).  
 --force-non-uniform-arrays | --no-force-non-uniform-arrays: force integer arrays to be initialized with multiple values (enabled by default).  
 --null-ptr-deref-prob <N>: allow null pointers to be dereferenced with probability N% (0 by default).  
 --dangling-ptr-deref-prob <N>: allow dangling pointers to be dereferenced with probability N% (0 by default).

--union-read-type-sensitive | --no-union-read-type-sensitive: allow | disallow reading an union field when there is no risk of reading padding bits (enabled by default).  
 --max-struct-nested-level: controls the max depth of nested structs (default is 3).  
 --no-hash-value-printf: do not emit printf on the index of an array  
 --no-signed-char-index: do not allow a var of type char to be used as array index

# PGO Notes

The following is a list of the profile-guided optimizations:

* **Inlining** – For example, if there exists a function A that frequently calls function B, and function B is relatively small, then profile-guided optimizations will inline function B in function A.
* **Virtual Call Speculation**– If a virtual call, or other call through a function pointer, frequently targets a certain function, a profile-guided optimization can insert a conditionally-executed direct call to the frequently-targeted function, and the direct call can be inlined.
* **Register Allocation** – Optimizing with profile data results in better register allocation.
* **Basic Block Optimization** – Basic block optimization allows commonly executed basic blocks that temporally execute within a given frame to be placed in the same set of pages (locality). This minimizes the number of pages used, thus minimizing memory overhead.
* **Size/Speed Optimization**– Functions where the program spends a lot of time can be optimized for speed.
* **Function Layout**– Based on the call graph and profiled caller/callee behavior, functions that tend to be along the same execution path are placed in the same section.
* **Conditional Branch Optimization** – With the value probes, profile-guided optimizations can find if a given value in a switch statement is used more often than other values. This value can then be pulled out of the switch statement. The same can be done with if/else instructions where the optimizer can order the if/else so that either the if or else block is placed first depending on which block is more frequently true.
* **Dead Code Separation**– Code that is not called during profiling is moved to a special section that is appended to the end of the set of sections. This effectively keeps this section out of the often-used pages.
* **EH Code Separation** – The EH code, being exceptionally executed, can often be moved to a separate section when profile-guided optimizations can determine that the exceptions occur only on exceptional conditions.
* **Memory Intrinsics** – The expansion of intrinsics can be decided better if it can be determined if an intrinsic is called frequently. An intrinsic can also be optimized based on the block size of moves or copies.

# Setting up Linux Perf tools

Run ‘sudo apt-get install linux-tools-common’

Upon running ‘perf’ in the console, you will likely get instructions for then installing the specific version of ‘perf’ for your kernel.

‘sudo apt-get install linux-tools-3.16.0-43-generic’ should then install the correct perf.

## Using Precise Event-Based Sampling (PEBS) (on Intel chips)

<https://software.intel.com/en-us/forums/topic/277553>

# [Adding Metadata to Instructions in LLVM IR](http://stackoverflow.com/questions/13425794/adding-metadata-to-instructions-in-llvm-ir)

<http://stackoverflow.com/questions/13425794/adding-metadata-to-instructions-in-llvm-ir>

1. Is metadata the right mechanism to use?

If your "other tool" is not a pass in itself, then yes, I think metadata is the best approach - keeps everything in the IR, easy to identify by eye, simple to manually add for testing, and - perhaps most importantly - does not collide with anything else, as long as you don't reuse existing metadata kinds.

However, if your "other tool" is a pass by itself, there's an alternative: you can make one pass dependent on the other, and than use information from the earlier directly in the later pass. The advantage is that you don't have to modify the IR.

2. How to use a custom metadata node?

Use the char\* variant of setMetadata, like so:

LLVMContext& C = Inst->getContext();

MDNode\* N = MDNode::get(C, MDString::get(C, "my md string content"));

Inst->setMetadata("my.md.name", N);

And if it's the first time the string is used in a setMetadata, it will automatically register my.md.name as a new kind in the module (it's actually consistent in the entire context,

I believe). You can later on retrieve the string by using:

cast<MDString>(Inst->getMetadata("my.md.name")->getOperand(0))->getString();

If you want to invoke getMetadata or setMetadata repeatedly from the same scope, though, you can also use Module::getMDKindID to just get the actual kind used, and use the variations of these methods that use the kind value.

Finally, be aware that you can limit the metadata node scope to be inside a function - use the MDNode::get(..., ..., true) variant for that - though I never used it myself.

|  |
| --- |
|  |

# Example of adding a function in-memory

<http://stackoverflow.com/questions/14035836/why-its-so-trouble-when-i-want-to-insert-some-instructions-in-basicblock>

Creating a module

<http://www.opensource.apple.com/source/lldb/lldb-69/llvm/examples/ModuleMaker/ModuleMaker.cpp?txt>

Creating a function

<http://www.opensource.apple.com/source/lldb/lldb-69/llvm/examples/Fibonacci/fibonacci.cpp?txt>

# LLVM Code Coverage Tools

<http://llvm.org/docs/CoverageMappingFormat.html>

### <https://sites.google.com/site/arnamoyswebsite/Welcome/updates-news/loopiteratorisabitdifferentinllvm>

### Loop iterator is a bit different in LLVM

|  |
| --- |
| 1.**Iterating over the functions in a Module**  *for(Module::iterator func = M.begin(), y = M.end(); func!= y; ++func)  //here func gives you a function*  *2.***Iterating over the basic blocks in a function**  *for(Function::iterator block = func.begin(), y = func.end(); block!= y; ++block)  //here block gives you a basic block*  But if you try to write the same code while iterating over the basic blocks in a loop (as I did), you are not lucky.  You have to use a special iterator called block\_iterator here.  So for a loop, the iterator should be written like the following -  *for(Loop::block\_iterator block = loop.block\_begin(), y = loop.block\_end(); block!= y; ++block)  //here block gives you a basic block*inside a loop |

## Getting all load and store instructions in LLVM

<http://stackoverflow.com/questions/7526550/instrumenting-c-c-codes-using-llvm>

The load and store instruction will only give accesses that are made to the heap using pointers. In order to get all memory accesses you also have to look at the values which can represent a memory location on the stack. Whether the value is written to the stack or stored in a register is determined during the register allocation phase which occurs in an optimization pass of the backend. Meaning that it's platform dependent and shouldn't be relied on.

Now unless you provide more information about what kind of memory accesses you're looking for, in what context and how you intend to instrument them, I can't help you much more then this.

## Inserting Instructions

<http://www.widecodes.com/0yxVPjUXjP/instrumenting-cc-code-using-llvm.html>

class ThePass : public llvm::BasicBlockPass {

public:

ThePass() : BasicBlockPass() {}

virtual bool runOnBasicBlock(llvm::BasicBlock &bb);

};

bool ThePass::runOnBasicBlock(BasicBlock &bb) {

bool retval = true;

for (BasicBlock::iterator bbit = bb.begin(), bbie = bb.end(); bbit != bbie;

++bbit) { // Make loop work given updates

Instruction \*i = bbit;

CallInst \* beforeCall = // INSERT THIS

beforeCall->insertBefore(i);

if (!i->isTerminator()) {

CallInst \* afterCall = // INSERT THIS

afterCall->insertAfter(i);

}

}

return retval;

}

<http://stackoverflow.com/questions/7806689/instrumenting-c-c-code-using-llvm>

## Using Clang for source-source translalation (AST Rewriter)

<http://adamrehn.com/articles/ast-instrumentation-examples-by-language>

## PGO Data Profile information

### BranchProbabilityInfo.h

<http://llvm.org/docs/doxygen/html/BranchProbabilityInfo_8h_source.html>

### BlockFrequencyInfo.h

<http://llvm.org/docs/doxygen/html/classllvm_1_1BlockFrequencyInfo.html>

Print out the block frequencies:   
<http://llvm.org/releases/2.1/docs/CommandGuide/html/llvm-prof.html>

[LLVMdev] Question about BlockFrequencyInfo   
<http://lists.cs.uiuc.edu/pipermail/llvmdev/2014-September/076544.html>

# [LLVMdev] Instructions that access memory @ the IR level

If you're writing a pass, you probably want to use

Instruction::mayReadFromMemory() and Instruction::mayWriteToMemory().

If you need something more specific, you can also take a look at their

implementations (in lib/VMCore/Instruction.cpp).

<http://lists.cs.uiuc.edu/pipermail/llvmdev/2011-July/041245.html>

## Treemap Visualization

Treemap in Python

<http://wiki.scipy.org/Cookbook/Matplotlib/TreeMap>

Sneiderman Algorithm: <http://hcil.cs.umd.edu/trs/91-03/91-03.html>

## Academic Websites with Homework using LLVM

<http://cseweb.ucsd.edu/classes/sp14/cse231-a/proj1.html#part1>

<http://www.cs.ucr.edu/~benavidz/cs201_spring15/project.html>

<http://pages.cs.wisc.edu/~fischer/cs701.f14/proj2.html>

<http://cpss2012.cse.psu.edu/slides/tan_instructions.pdf>

<http://www.cs.sfu.ca/~wsumner/teaching/886/15/project1.html>

LLVM Pin

<http://eces.colorado.edu/~blomsted/llvmpin/llvmpin.html>

## Analyzing function Control Flow Graphs (CFGs) with LLVM

<http://eli.thegreenplace.net/2013/09/16/analyzing-function-cfgs-with-llvm>

## LLVM Debugging information format

Basically, the debug information allows you to compile a program with “-O0 -g” and get full debug information, allowing you to arbitrarily modify the program as it executes from a debugger. Compiling a program with “-O3 -g” gives you full debug information that is always available and accurate for reading (e.g., you get accurate stack traces despite tail call elimination and inlining), but you might lose the ability to modify the program and call functions where were optimized out of the program, or inlined away completely.

<http://llvm.org/docs/SourceLevelDebugging.html>

<http://llvm.org/doxygen/DebugInfo_8h.html>

LLVM IR Embedded Metadata

<http://nondot.org/~sabre/LLVMNotes/EmbeddedMetadata.txt>

# [LLVMdev] Mapping bitcode to source code (Feb 10, 2010)

<http://lists.cs.uiuc.edu/pipermail/llvmdev/2010-February/029282.html>

# LLVM Coding Standards

Things such as not using the <iostream> library

<http://llvm.org/releases/2.0/docs/CodingStandards.html>

List of people who answer LLVM Questions:

<http://stackoverflow.com/users/164925/anton-korobeynikov>