```
#include <stdio.h>

#define abs(a) ( ((a) < 0) ? -(a) : (a) )
#define max( a, b ) ( ((a) > (b)) ? (a) : (b) )
#define min( a, b ) ( ((a) < (b)) ? (a) : (b) )

int inData[] = {52, 84};

// square-root approximation:
int main() {

    // sqrt(52^2 + 84^2) = 98.79
    // This should be approximated as 100
    int a = inData[0];
    int b = inData[1];

    int x = max(abs(a), abs(b));
    int y = min(abs(a), abs(b));
    int sqrt = max(x, x-(x>>3)+(y>>1));

    printf("Result: %d\n", sqrt);
    if (sqrt == 100) {
        printf("RESULT: PASS\n");
    } else {
        printf("RESULT: FAIL\n");
    }
    return sqrt;
                                                    }
```

The passes occurred every time we ran the command are the same. The number of each kind of passes is the same. When I wrote a blank main function, exactly the same passes have been run. Thus, I guess these are the basic passes that will be gone through in every function. However, more passes will be automatically run according to the results from analysis passes, for example, loop transform passes will be added if there is a loop structure.

I tried -O1 -O2 as well, they preform less passes than the -O3, but I didn't find any relevant information about -OX flags. Compared to -O3, -O1 didn't do analysis and transformation for global variables.

```
# produces pre-link time optimization binary bitcode: sra.prelto.bc
clang sra.c -emit-llvm -c -fno-builtin -m32 -I ../lib/include/ -I/usr/include/i386-linux-gnu -O3 -mllvm
-inline-threshold=-100 -o sra.prelto.1.bc
```

Reduce Verilog code from 4214 to 863. And the modelsim result is still correct for sra.c .

The **opt** command is the modular LLVM optimizer and analyzer. It takes LLVM source files as input, runs the specified optimizations or analyses on it, and then outputs the optimized file or the analysis results. The function of **opt** depends on whether the **-analyze** option is given.

===--===

... Pass execution timing report ...

===---===

Total Execution Time: 0.0040 seconds (0.0505 wall clock)

--System Time--	--User+System--	---Wall Time---	--- Name ---
0.0000 (0.0%)	0.0000 (0.0%)	0.0237 (46.9%)	Early CSE
0.0000 (0.0%)	0.0000 (0.0%)	0.0111 (22.0%)	Reassociate expressions
0.0000 (0.0%)	0.0000 (0.0%)	0.0087 (17.3%)	Simplify the CFG
0.0000 (0.0%)	0.0000 (0.0%)	0.0043 (8.6%)	Combine redundant instructions
0.0000 (0.0%)	0.0000 (0.0%)	0.0003 (0.6%)	Scalar Replacement of Aggregates (DT)
0.0000 (0.0%)	0.0000 (0.0%)	0.0003 (0.5%)	Simplify well-known library calls
0.0000 (0.0%)	0.0000 (0.0%)	0.0002 (0.5%)	Global Value Numbering
0.0000 (0.0%)	0.0000 (0.0%)	0.0002 (0.4%)	Jump Threading
0.0000 (0.0%)	0.0000 (0.0%)	0.0002 (0.4%)	Bitcode Writer
0.0000 (0.0%)	0.0000 (0.0%)	0.0001 (0.3%)	Combine redundant instructions
0.0000 (0.0%)	0.0000 (0.0%)	0.0001 (0.2%)	Interprocedural Sparse Conditional Constant Propagation
0.0000 (0.0%)	0.0000 (0.0%)	0.0001 (0.2%)	Combine redundant instructions
0.0000 (0.0%)	0.0000 (0.0%)	0.0001 (0.2%)	Natural Loop Information
0.0000 (0.0%)	0.0000 (0.0%)	0.0001 (0.2%)	Combine redundant instructions
0.0000 (0.0%)	0.0000 (0.0%)	0.0001 (0.2%)	Combine redundant instructions
0.0000 (0.0%)	0.0000 (0.0%)	0.0001 (0.1%)	Tail Call Elimination
0.0000 (0.0%)	0.0000 (0.0%)	0.0001 (0.1%)	Function Integration/Inlining
0.0000 (0.0%)	0.0000 (0.0%)	0.0001 (0.1%)	Global Variable Optimizer
0.0000 (0.0%)	0.0000 (0.0%)	0.0001 (0.1%)	Dead Argument Elimination
0.0000 (0.0%)	0.0000 (0.0%)	0.0000 (0.1%)	Dominator Tree Construction
0.0000 (0.0%)	0.0000 (0.0%)	0.0000 (0.1%)	Simplify the CFG
0.0000 (0.0%)	0.0000 (0.0%)	0.0000 (0.1%)	Deduce function attributes
0.0000 (0.0%)	0.0000 (0.0%)	0.0000 (0.1%)	Basic CallGraph Construction
0.0000 (0.0%)	0.0000 (0.0%)	0.0000 (0.1%)	Module Verifier
0.0000 (0.0%)	0.0000 (0.0%)	0.0000 (0.1%)	Early CSE
0.0000 (0.0%)	0.0000 (0.0%)	0.0000 (0.1%)	Dead Store Elimination
0.0000 (0.0%)	0.0000 (0.0%)	0.0000 (0.0%)	Scalar Evolution Analysis
0.0000 (0.0%)	0.0000 (0.0%)	0.0000 (0.0%)	Remove unused exception handling info
0.0000 (0.0%)	0.0000 (0.0%)	0.0000 (0.0%)	Jump Threading
0.0000 (0.0%)	0.0000 (0.0%)	0.0000 (0.0%)	Merge Duplicate Global Constants
0.0000 (0.0%)	0.0000 (0.0%)	0.0000 (0.0%)	Lazy Value Information Analysis
0.0000 (0.0%)	0.0000 (0.0%)	0.0000 (0.0%)	Lower 'expect' Intrinsics
0.0000 (0.0%)	0.0000 (0.0%)	0.0000 (0.0%)	MemCpy Optimization
0.0000 (0.0%)	0.0000 (0.0%)	0.0000 (0.0%)	Sparse Conditional Constant Propagation
0.0000 (0.0%)	0.0000 (0.0%)	0.0000 (0.0%)	Memory Dependence Analysis
0.0000 (0.0%)	0.0000 (0.0%)	0.0000 (0.0%)	Dominator Tree Construction
0.0000 (0.0%)	0.0000 (0.0%)	0.0000 (0.0%)	Dead Global Elimination
0.0000 (0.0%)	0.0000 (0.0%)	0.0000 (0.0%)	Dominator Tree Construction
0.0000 (0.0%)	0.0000 (0.0%)	0.0000 (0.0%)	Dominator Tree Construction
0.0000 (0.0%)	0.0000 (0.0%)	0.0000 (0.0%)	Dominator Tree Construction
0.0040 (100.0%)	0.0040 (100.0%)	0.0000 (0.0%)	Simplify the CFG
0.0000 (0.0%)	0.0000 (0.0%)	0.0000 (0.0%)	Value Propagation
0.0000 (0.0%)	0.0000 (0.0%)	0.0000 (0.0%)	Value Propagation

```
  0.0000 (  0.0%)   0.0000 (  0.0%)   0.0000 (  0.0%)  No Alias Analysis (always returns 'may' alias)
  0.0000 (  0.0%)   0.0000 (  0.0%)   0.0000 (  0.0%)  Basic Alias Analysis (stateless AA impl)
  0.0000 (  0.0%)   0.0000 (  0.0%)   0.0000 (  0.0%)  Aggressive Dead Code Elimination
  0.0000 (  0.0%)   0.0000 (  0.0%)   0.0000 (  0.0%)  Lazy Value Information Analysis
  0.0000 (  0.0%)   0.0000 (  0.0%)   0.0000 (  0.0%)  Simplify the CFG
  0.0000 (  0.0%)   0.0000 (  0.0%)   0.0000 (  0.0%)  Simplify the CFG
  0.0000 (  0.0%)   0.0000 (  0.0%)   0.0000 (  0.0%)  Memory Dependence Analysis
  0.0000 (  0.0%)   0.0000 (  0.0%)   0.0000 (  0.0%)  Promote 'by reference' arguments to scalars
  0.0000 (  0.0%)   0.0000 (  0.0%)   0.0000 (  0.0%)  Memory Dependence Analysis
  0.0000 (  0.0%)   0.0000 (  0.0%)   0.0000 (  0.0%)  Strip Unused Function Prototypes
  0.0000 (  0.0%)   0.0000 (  0.0%)   0.0000 (  0.0%)  Scalar Replacement of Aggregates (SSAUp)
  0.0000 (  0.0%)   0.0000 (  0.0%)   0.0000 (  0.0%)  Preliminary module verification
  0.0000 (  0.0%)   0.0000 (  0.0%)   0.0000 (  0.0%)  Target Library Information
  0.0000 (  0.0%)   0.0000 (  0.0%)   0.0000 (  0.0%)  No Alias Analysis (always returns 'may' alias)
  0.0040 (100.0%)   0.0040 (100.0%)   0.0505 (100.0%)  Total

Total 57 Passes listed,  37 different Passes.
../../llvm/Release+Asserts/bin/opt -O3 -time-passes sra.bc > /dev/null
```

LLVM Introduction to Passes: http://llvm.org/docs/Passes.html#simplify-libcalls
Optimizations are implemented as Passes that traverse some portion of a program to either collect information or transform the program. **Analysis passes** compute information that other passes can use or for debugging or program visualization purposes. **Transform passes** can use (or invalidate) the analysis passes. **Transform passes** all mutate the program in some way. **Utility passes** provides some utility but don't otherwise fit categorization. For example passes to extract functions to bitcode or write a module to bitcode are neither analysis nor transform passes.

-early-cse Early CSE (Common Subexpression Elimination**):**
CSE matches lexically identical expressions and replaces these expressions with a temporary register that stores the result of the expression.

In the following code:

```
a = b * c + g;
d = b * c * d;
```

it may be worth transforming the code to:

```
tmp = b * c;
a = tmp + g;
d = tmp * d;
```

```
if the time cost of storing and retrieving "tmp" outweighs the cost of calculating
"b * c" an extra time.
```

-reassociate: Reassociate expressions

This pass reassociates commutative expressions in an order that is designed to promote better constant propagation, GCSE, LICM, PRE, etc.

For example: $4 + (x + 5) \Rightarrow x + (4 + 5)$

In the implementation of this algorithm, constants are assigned rank = 0, function arguments are rank = 1, and other values are assigned ranks corresponding to the reverse post order traversal of current function (starting at 2), which effectively gives values in deep loops higher rank than values not in loops.

-simplifycfg: Simplify the CFG (*Context-free grammar)?(Control Flow Graphs)*

Control Flow Graphs(CFG) The control flow graph is a directed graph wherein the nodes are either program statements or basic blocks and an edge A → B exists if B can execute immediately after A. Special nodes .S and E denote the start and exit points respectively. By convention, all CFGs include an edge S → E, although this edge is not always shown. It is generally simpler to let each node in the flowgraph be a single instruction, rather than a basic block. Using basic blocks has speed and memory advantages, but it isn't worthwhile in a research compiler. We'll assume each node is a single instruction.

Control Dependence Graph (CDG) Paper by Lam and Wall demonstrated the importance of control-dependence in parallelization. Ultimately, we need to define the control-dependence graph (CDG),Intuitive definition of Control-dependence: Statement A is control-dependent on a branch B if the decision at B can affect the number of times A executes. In a block-structured programming language, each statement is control-dependent on its enclosing conditionals.

Performs dead code elimination and basic block merging. Specifically:

1. Removes basic blocks with no predecessors.
2. Merges a basic block into its predecessor if there is only one and the predecessor only has one successor.
3. Eliminates PHI nodes for basic blocks with a single predecessor.
4. Eliminates a basic block that only contains an unconditional branch.

Dead Code is code that is either never executed or, if it is executed, its result is never used by the program. Therefore dead code can be eliminated from the program. For example:

```
int foo(void)
{
  int a = 24;
  int b = 25; /* Assignment to dead variable */
  int c;
  c = a << 2;
  return c;
  b = 24; /* Unreachable code */
  return 0;
}
```

Block Merging: Some compilers benefit if the program graph can be transformed into a smaller number of larger basic blocks. For example:

```
int a;

int b;

void f (int x, int y)
{
  goto L1;                    /* basic block 1 */

L2:                           /* basic block 2 */
```

```
  b = x + y;
  goto L3;

L1:                             /* basic block 3 */
  a = x + y;
  goto L2;

L3:                             /* basic block 4 */
  return;
}
```

The code fragment below shows the function after the basic blocks have been rearranged and combined into one large basic block.

```
int a;

int b;

void f (int x, int y)
{
  a = x + y;                    /* basic block 1 */
  b = x + y;
  return;
}
```

Below is the same code fragment after CSE (common subexpression elimination) Elimination.

int a;

```
int b;

void f (int x, int y)
{
  register int __t;             /* compiler generated temp */

  __t = x + y;                  /* assign CSE to temp */
  a = __t;
  b = __t;
  return;
}
```

-instcombine: Combine redundant instructions

This pass algebraically combine instructions to fewer,simple instructions. x+x=2*x=shift left logically 1 bit

Combine instructions to form fewer, simple instructions. This pass does not modify the CFG This pass is where algebraic simplification happens.

This pass combines things like:

```
%Y = add i32 %X, 1
%Z = add i32 %Y, 1
```

into:

```
%Z = add i32 %X, 2
```

This is a simple worklist driven algorithm.

This pass guarantees that the following canonicalizations are performed on the program:

- If a binary operator has a constant operand, it is moved to the right- hand side.
- Bitwise operators with constant operands are always grouped so that shifts are performed first, then `or`s, then `and`s, then `xor`s.
- Compare instructions are converted from <, >, ≤, or ≥ to = or ≠ if possible.
- All `cmp` instructions on boolean values are replaced with logical operations.
- `add X, X` is represented as `mul X, 2` ⇒ `shl X, 1`
- Multiplies with a constant power-of-two argument are transformed into shifts.
- … etc.

opt

-scalarrepl: Scalar Replacement of Aggregates (DT) ??

The well-known scalar replacement of aggregates transformation. This transform breaks up `alloca` instructions of aggregate type (structure or array) into individual `alloca` instructions for each member if possible. Then, if possible, it transforms the individual `alloca` instructions into nice clean scalar SSA form(Static single assignment form:is a property of an intermediate representation (IR), which says that each variable is assigned exactly once).

This combines a simple scalar replacement of aggregates algorithm with the `mem2reg` algorithm because often interact, especially for C++ programs. As such, iterating between `scalarrepl`, then `mem2reg` until we run out of things to promote works well.
-mem2reg: Promote Memory to Register
This file promotes memory references to be register references. It promotes `alloca` instructions which only have `load`s and `store`s as uses. An `alloca` is transformed by using dominator frontiers to place `phi` nodes, then traversing the function in depth-first order to rewrite `load`s and `store`s as appropriate. This is just the standard SSA construction algorithm to construct "pruned" SSA form.

The dominance frontier: a node **d** *dominates* a node **n** if every path from the *start node* to **n** must go through **d**.The dominance frontier of a node x is the set of edges y → z where x dom y but x not strictly dom z.

phi nodes: nodes after a branch out from a nature loop.

-simplify-libcalls: Simplify well-known library calls

Applies a variety of small optimizations for calls to specific well-known function calls (e.g. runtime library functions). For example, a call `exit(3)` that occurs within the `main()` function can be transformed into simply `return 3`.

-gvn: Global Value Numbering(GVN)

This pass performs global value numbering to eliminate fully and partially redundant instructions. It also performs redundant load elimination.
Global value numbering (GVN) is a compiler optimization based on the SSA intermediate

representation. It sometimes helps eliminate redundant code that common subexpression elimination (CSE) does not.

```
w := 3
x := 3
y := x + 4
z := w + 4
```

a good GVN routine would assign the same value number to `w` and `x`, and the same value number to `y` and `z`.Using this information, the previous code fragment may be safely transformed into:

```
w := 3
x := w
y := w + 4
z := y
```

The reason that GVN is sometimes more powerful than CSE comes from the fact that CSE matches lexically identical expressions whereas the GVN tries to determine an underlying equivalence. For instance, in the code:

```
a := c × d
e := c
f := e × d
```

CSE would *not* eliminate the recomputation assigned to `f`, but even a poor GVN algorithm should discover and eliminate this redundancy.

-jump-threading: Jump Threading

Jump threading tries to find distinct threads of control flow running through a basic block. This pass looks at blocks that have multiple predecessors and multiple successors. If one or more of the predecessors of the block can be proven to always cause a jump to one of the successors, we forward the edge from the predecessor to the successor by duplicating the contents of this block.

An example of when this can occur is code like this:

```
if () { ...
  X = 4;
}
if (X < 3) {
```

In this case, the unconditional branch at the end of the first if can be revectored to the false side of the second if. .Jump threading. This pass detects a condition jump that branches to an identical or inverse test. Such jumps can be `` `threaded' `` through the second conditional test. The source code for this pass is in `` `jump.c' ``. This optimization is only performed if `` `-fthread-jumps' `` is enabled.

Bitcode Writer (not found)

-ipsccp: Interprocedural Sparse Conditional Constant Propagation

An interprocedural variant of **-sccp: Sparse Conditional Constant Propagation**

Sparse conditional constant propagation and merging, which can be summarized as:

1. Assumes values are constant unless proven otherwise
2. Assumes BasicBlocks are dead unless proven otherwise
3. Proves values to be constant, and replaces them with constants
4. Proves conditional branches to be unconditional

Note that this pass has a habit of making definitions be dead. It is a good idea to to run a DCE(Dead Code Elimination) pass sometime after running this pass.

-loops: Natural Loop Information

This analysis is used to identify natural loops and determine the loop depth of various nodes of the CFG. Note that the loops identified may actually be several natural loops that share the same header node... not just a single natural loop.

-tailcallelim: Tail Call Elimination

It transforms self recursion to a non-recursive, iterative loop.
A function call is said to be tail recursive if there is nothing to do after the function returns except return its value.

This file transforms calls of the current function (self recursion) followed by a return instruction with a branch to the entry of the function, creating a loop. This pass also implements the following extensions to the basic algorithm:

- Trivial instructions between the call and return do not prevent the transformation from taking place, though currently the analysis cannot support moving any really useful instructions (only dead ones).
- This pass transforms functions that are prevented from being tail recursive by an associative expression to use an accumulator variable, thus compiling the typical naive factorial or `fib` implementation into efficient code. (recursive->while)
- TRE is performed if the function returns void, if the return returns the result returned by the call, or if the function returns a run-time constant on all exits from the function. It is possible, though unlikely, that the return returns something else (like constant 0), and can still be TRE'd. It can be TRE'd if *all other* return instructions in the function return the exact same value. **TRE** is an open-source library for texts search, which works like regular expression engine with ability of fuzzy string searching. ??
- If it can prove that callees do not access theier caller stack frame, they are marked as eligible for tail call elimination (by the code generator).

-inline: Function Integration/Inlining

It replaces a function call site with the body of callees.

Bottom-up inlining of functions into callees.

-globalopt: Global Variable Optimizer

This pass transforms simple global variables that never have their address taken. If obviously true, it marks read/write globals as constant, deletes variables only stored to, etc.

-deadargelim: Dead Argument Elimination

This pass deletes dead arguments from internal functions. Dead argument elimination removes arguments which are directly dead, as well as arguments only passed into function calls as dead arguments of other functions. This pass also deletes dead arguments in a similar way.

This pass is often useful as a cleanup pass to run after **aggressive interprocedural passes**, which add possibly-dead arguments.

Interprocedural optimization (IPO) is a collection of compiler techniques used in computer programming to improve performance in programs containing many frequently used functions of small or medium length. IPO differs from other compiler optimization because it analyzes the entire program; other optimizations look at only a single function, or even a single block of code.

IPO seeks to reduce or eliminate duplicate calculations, inefficient use of memory, and to simplify iterative sequences such as loops. If there is a call to another routine that occurs within a loop, IPO analysis may determine that it is best to inline that. Additionally, IPO may re-order the routines for better memory layout and locality.

IPO may also include typical compiler optimizations on a whole-program level, for example dead code elimination, which removes code that is never executed. To accomplish this, the compiler tests for branches that are never taken and removes the code in that branch. IPO also tries to ensure better use of constants. Modern compilers offer IPO as an option at compile-time. The actual IPO process may occur at any step between the human-readable source code and producing a finished executable binary program.

-domtree: Dominator Tree Construction

This pass is a simple dominator construction algorithm for finding forward dominators.

-functionattrs: Deduce function attributes

A simple interprocedural pass which walks the call-graph, looking for functions which do not access or only read non-local memory, and marking them readnone/readonly. In addition, it marks function arguments (of pointer type) 'nocapture' if a call to the function does not create any copies of the pointer value that outlive the call. This more or less means that the pointer is only dereferenced, and not returned from the function or stored in a global. This pass is implemented as a bottom-up traversal of the call-graph.

Function attributes are set to communicate additional information about a function. Function attributes are considered to be part of the function, not of the function type, so functions with different parameter attributes can have the same function type.

Function attributes are simple keywords that follow the type specified. If multiple attributes are needed, they are space separated. For example:

```
define void @f() noinline { ... }??
define void @f() alwaysinline { ... }
define void @f() alwaysinline optsize { ... }
define void @f() optsize { ... }
```

-basiccg: Basic CallGraph Construction

Yet to be written. (0.0)!

-verify: Module Verifier
-preverify: Preliminary module verification

Ensures that the module is in the form required by the Module Verifier pass.

Running the verifier runs this pass automatically, so there should be no need to use it directly.

Verifies an LLVM IR code. This is useful to run after an optimization which is undergoing testing. Note that `llvm-as` verifies its input before emitting bitcode, and also that malformed bitcode is likely to make LLVM crash. All language front-ends are therefore encouraged to verify their output before performing optimizing transformations.

- Both of a binary operator's parameters are of the same type.
- Verify that the indices of mem access instructions match other operands.
- Verify that arithmetic and other things are only performed on first-class types. Verify that shifts and logicals only happen on integrals f.e.
- All of the constants in a switch statement are of the correct type.
- The code is in valid SSA form.
- It is illegal to put a label into any other type (like a structure) or to return one.
- Only phi nodes can be self referential: `%x = add i32 %x, %x` is invalid.
- PHI nodes must have an entry for each predecessor, with no extras.
- PHI nodes must be the first thing in a basic block, all grouped together.
- PHI nodes must have at least one entry.
- All basic blocks should only end with terminator insts, not contain them.
- The entry node to a function must not have predecessors.
- All Instructions must be embedded into a basic block.
- Functions cannot take a void-typed parameter.
- Verify that a function's argument list agrees with its declared type.
- It is illegal to specify a name for a void value.
- It is illegal to have a internal global value with no initializer.
- It is illegal to have a ret instruction that returns a value that does not agree with the function return value type.
- Function call argument types match the function prototype.
- All other things that are tested by asserts spread about the code.

Note that this does not provide full security verification (like Java), but instead just tries to ensure that code is well-formed.

-dse: Dead Store Elimination

A trivial dead store elimination that only considers basic-block local redundant stores.

-scalar-evolution: Scalar Evolution Analysis

The `ScalarEvolution` analysis can be used to analyze and catagorize scalar expressions in loops. It specializes in recognizing general induction variables, representing them with the abstract and opaque `SCEV` class. Given this analysis, trip counts of loops and other important properties can be

obtained.

This analysis is primarily useful for induction variable substitution and strength reduction.

-prune-eh: Remove unused exception handling info

This file implements a simple interprocedural pass which walks the **call-graph**, turning `invoke` instructions into `call` instructions if and only if the callee cannot throw an exception. It implements this as a bottom-up traversal of the call-graph.

-constmerge: Merge Duplicate Global Constants

Merges duplicate global constants together into a single constant that is shared. This is useful because some passes (ie TraceValues) insert a lot of string constants into the program, regardless of whether or not an existing string is available.

-lazy-value-info: Lazy Value Information Analysis

Interface for lazy computation of value constraint information.

Lower 'expect' Intrinsics' (Not Found)

-loweratomic - Lower atomic intrinsics to non-atomic form
-lowerinvoke - Lower invoke and unwind, for unwindless code generators
-lowersetjmp - Lower Set Jump
-lowerswitch - Lower SwitchInst's to branches ??

-memcpyopt: MemCpy Optimization

This pass performs various transformations related to eliminating memcpy calls, or transforming sets of stores into memset's.

-sccp: Sparse Conditional Constant Propagation

Sparse conditional constant propagation and merging, which can be summarized as:

1. Assumes values are constant unless proven otherwise
2. Assumes BasicBlocks are dead unless proven otherwise
3. Proves values to be constant, and replaces them with constants
4. Proves conditional branches to be unconditional

Note that this pass has a habit of making definitions be dead. It is a good idea to to run a DCE pass sometime after running this pass.

-memdep: Memory Dependence Analysis

An analysis that determines, for a given memory operation, what preceding memory operations it depends on. It builds on alias analysis information, and tries to provide a lazy, caching interface to a common kind of alias information query.

-globaldce: Dead Global Elimination

This transform is designed to eliminate unreachable internal globals from the program. It uses an

aggressive algorithm, searching out globals that are known to be alive. After it finds all of the globals which are needed, it deletes whatever is left over. This allows it to delete recursive chunks of the program which are unreachable.

-correlated-propagation Value Propagation(Not Found)

-no-aa: No Alias Analysis (always returns 'may' alias)

This is the default implementation of the Alias Analysis interface. It always returns "I don't know" for alias queries. NoAA is unlike other alias analysis implementations, in that it does not chain to a previous analysis. As such it doesn't follow many of the rules that other alias analyses must.

-basicaa: Basic Alias Analysis (stateless AA impl)

A basic alias analysis pass that implements identities (two different globals cannot alias, etc), but does no stateful analysis.v

Alias analysis is a technique in compiler theory, used to determine if a storage location may be accessed in more than one way. Two pointers are said to be aliased if they point to the same location.

-adce: Aggressive Dead Code Elimination

ADCE aggressively tries to eliminate code. This pass is similar to DCE but it assumes that values are dead until proven otherwise. This is similar to SCCP, except applied to the liveness of values.

-argpromotion: Promote 'by reference' arguments to scalars

This pass promotes "by reference" arguments to be "by value" arguments. In practice, this means looking for internal functions that have pointer arguments. If it can prove, through the use of alias analysis, that an argument is *only* loaded, then it can pass the value into the function instead of the address of the value. This can cause recursive simplification of code and lead to the elimination of allocas (especially in C++ template code like the STL).

This pass also handles aggregate arguments that are passed into a function, scalarizing them if the elements of the aggregate are only loaded. Note that it refuses to scalarize aggregates which would require passing in more than three operands to the function, because passing thousands of operands for a large array or structure is unprofitable!

Note that this transformation could also be done for arguments that are only stored to (returning the value instead), but does not currently. This case would be best handled when and if LLVM starts supporting multiple return values from functions.

-strip-dead-prototypes: Strip Unused Function Prototypes

This pass loops over all of the functions in the input module, looking for dead declarations and removes them. Dead declarations are declarations of functions for which no implementation is available (i.e., declarations for unused library functions).

-scalarrepl-ssa Scalar Replacement of Aggregates (SSAUp) (Not Found)

-targetlibinfo Target Library Information (Not Found)

MAKEFILE with Passes:

clang sra.c -emit-llvm -c -fno-builtin -m32 -I ../lib/include/ -I/usr/include/i386-linux-gnu -O3 -mllvm -inline-threshold=-100 -o sra.prelto.1.bc
linking may produce llvm mem-family intrinsics
../../llvm/Release+Asserts/bin/llvm-ld -disable-inlining -disable-opt sra.prelto.1.bc -b=sra.prelto.linked.bc
performs intrinsic lowering so that the linker may be optimized
../../llvm/Release+Asserts/bin/opt -load=../../cloog/install/lib/libisl.so -load=../../cloog/install/lib/libcloog-isl.so -load=../../llvm/tools/polly/Release+Asserts/lib/LLVMPolly.so -load=../../llvm/Release+Asserts/lib/LLVMLegUp.so -legup-config=../legup.tcl -legup-prelto < sra.prelto.linked.bc > sra.prelto.bc
produces sra.bc binary bitcode and a.out shell script: lli sra.bc
../../llvm/Release+Asserts/bin/llvm-ld -disable-inlining -disable-opt sra.prelto.bc ../lib/llvm/liblegup.a -b=sra.bc
produces textual bitcodes: sra.prelto.1.ll sra.prelto.ll sra.ll
../../llvm/Release+Asserts/bin/llvm-dis sra.prelto.1.bc
../../llvm/Release+Asserts/bin/llvm-dis sra.prelto.linked.bc
../../llvm/Release+Asserts/bin/llvm-dis sra.prelto.bc
../../llvm/Release+Asserts/bin/llvm-dis sra.bc
produces verilog: sra.v
../../llvm/Release+Asserts/bin/llc -legup-config=../../hwtest/CycloneII.tcl -legup-config=../legup.tcl -march=v sra.bc -o sra.v

Output for -O0:

scheduling.legup.rpt:
113 states, 116 registers used

Critical Warning (332148): Timing requirements not met ?
modelsim:
run 7000000000000000ns
Result: 100
RESULT: PASS
At t= 145000 clk=1 finish=1 return_val= 100
** Note: $finish : sra.v(4210)
Time: 145 ns Iteration: 3 Instance: /main_tb

Output for -O3:

scheduling.legup.rpt:
10 states, 24 registers used

Critical Warning (332148): Timing requirements not met ?
modelsim:
run 7000000000000000ns
Result: 100
RESULT: PASS

```
# At t=            21000 clk=1 finish=1 return_val=       100
# ** Note: $finish    : sra.v(859)
#    Time: 21 ns  Iteration: 3  Instance: /main_tb


llvm -dis

opt -help
  Optimizations available:
  -aa-eval                    - Exhaustive Alias Analysis Precision Evaluator
    -adce                     - Aggressive Dead Code Elimination
    -always-inline              - Inliner for always_inline functions
    -argpromotion               - Promote 'by reference' arguments to scalars
    -basicaa                  - Basic Alias Analysis (stateless AA impl)
    -basiccg                  - Basic CallGraph Construction
    -block-placement            - Profile Guided Basic Block Placement
    -break-crit-edges           - Break critical edges in CFG
    -codegenprepare             - Optimize for code generation
    -constmerge               - Merge Duplicate Global Constants
    -constprop                - Simple constant propagation
    -correlated-propagation       - Value Propagation
    -count-aa                 - Count Alias Analysis Query Responses
    -dce                      - Dead Code Elimination
    -deadargelim                - Dead Argument Elimination
    -deadarghaX0r               - Dead Argument Hacking (BUGPOINT USE ONLY; DO NOT
USE)
    -deadtypeelim               - Dead Type Elimination
    -debug-aa                 - AA use debugger
    -die                      - Dead Instruction Elimination
    -domfrontier                - Dominance Frontier Construction
    -domtree                  - Dominator Tree Construction
    -dot-callgraph              - Print Call Graph to 'dot' file
    -dot-cfg                  - Print CFG of function to 'dot' file
    -dot-cfg-only               - Print CFG of function to 'dot' file (with no function bodies)
    -dot-dom                  - Print dominance tree of function to 'dot' file
    -dot-dom-only               - Print dominance tree of function to 'dot' file (with no function
bodies)
    -dot-postdom                - Print postdominance tree of function to 'dot' file
    -dot-postdom-only             - Print postdominance tree of function to 'dot' file (with no
function bodies)
    -dot-regions                - Print regions of function to 'dot' file
    -dot-regions-only             - Print regions of function to 'dot' file (with no function bodies)
    -dse                      - Dead Store Elimination
    -early-cse                - Early CSE
    -extract-blocks             - Extract Basic Blocks From Module (for bugpoint use)
    -functionattrs              - Deduce function attributes
    -globaldce                - Dead Global Elimination
    -globalopt                - Global Variable Optimizer
    -globalsmodref-aa             - Simple mod/ref analysis for globals
```

-gvn - Global Value Numbering
-indvars - Canonicalize Induction Variables
-inline - Function Integration/Inlining
-insert-edge-profiling - Insert instrumentation for edge profiling
-insert-optimal-edge-profiling - Insert optimal instrumentation for edge profiling
-insert-path-profiling - Insert instrumentation for Ball-Larus path profiling
-instcombine - Combine redundant instructions
-instcount - Counts the various types of Instructions
-instnamer - Assign names to anonymous instructions
-instsimplify - Remove redundant instructions
-internalize - Internalize Global Symbols
-intervals - Interval Partition Construction
-ipconstprop - Interprocedural constant propagation
-ipsccp - Interprocedural Sparse Conditional Constant Propagation
-iv-users - Induction Variable Users
-jump-threading - Jump Threading
-lazy-value-info - Lazy Value Information Analysis
-lcssa - Loop-Closed SSA Form Pass
-lda - Loop Dependence Analysis
-libcall-aa - LibCall Alias Analysis
-licm - Loop Invariant Code Motion
-lint - Statically lint-checks LLVM IR
-loop-deletion - Delete dead loops
-loop-extract - Extract loops into new functions
-loop-extract-single - Extract at most one loop into a new function
-loop-idiom - Recognize loop idioms
-loop-instsimplify - Simplify instructions in loops
-loop-reduce - Loop Strength Reduction
-loop-rotate - Rotate Loops
-loop-simplify - Canonicalize natural loops
-loop-unroll - Unroll loops
-loop-unswitch - Unswitch loops
-loops - Natural Loop Information
-loweratomic - Lower atomic intrinsics to non-atomic form
-lowerinvoke - Lower invoke and unwind, for unwindless code generators
-lowersetjmp - Lower Set Jump
-lowerswitch - Lower SwitchInst's to branches
-mem2reg - Promote Memory to Register
-memcpyopt - MemCpy Optimization
-memdep - Memory Dependence Analysis
-mergefunc - Merge Functions
-mergereturn - Unify function exit nodes
-module-debuginfo - Decodes module-level debug info
-no-aa - No Alias Analysis (always returns 'may' alias)
-no-path-profile - No Path Profile Information
-no-profile - No Profile Information
-partial-inliner - Partial Inliner
-path-profile-loader - Load path profile information from file
-path-profile-verifier - Compare the path profile derived edge profile against the edge

profile.
-postdomfrontier - Post-Dominance Frontier Construction
-postdomtree - Post-Dominator Tree Construction
-preverify - Preliminary module verification
-print-alias-sets - Alias Set Printer
-print-callgraph - Print a call graph
-print-callgraph-sccs - Print SCCs of the Call Graph
-print-cfg-sccs - Print SCCs of each function CFG
-print-dbginfo - Print debug info in human readable form
-print-dom-info - Dominator Info Printer
-print-externalfnconstants - Print external fn callsites passed constants
-print-function - Print function to stderr
-print-memdeps - Print MemDeps of function
-print-module - Print module to stderr
-print-used-types - Find Used Types
-profile-estimator - Estimate profiling information
-profile-loader - Load profile information from llvmprof.out
-profile-verifier - Verify profiling information
-prune-eh - Remove unused exception handling info
-reassociate - Reassociate expressions
-reg2mem - Demote all values to stack slots
-regions - Detect single entry single exit regions
-scalar-evolution - Scalar Evolution Analysis
-scalarrepl - Scalar Replacement of Aggregates (DT)
-scalarrepl-ssa - Scalar Replacement of Aggregates (SSAUp)
-sccp - Sparse Conditional Constant Propagation
-scev-aa - ScalarEvolution-based Alias Analysis
-simplify-libcalls - Simplify well-known library calls
-simplifycfg - Simplify the CFG
-sink - Code sinking
-sretpromotion - Promote sret arguments to multiple ret values
-strip - Strip all symbols from a module
-strip-dead-debug-info - Strip debug info for unused symbols
-strip-dead-prototypes - Strip Unused Function Prototypes
-strip-debug-declare - Strip all llvm.dbg.declare intrinsics
-strip-nondebug - Strip all symbols, except dbg symbols, from a module
-tailcallelim - Tail Call Elimination
-tailduplicate - Tail Duplication
-targetdata - Target Data Layout
-targetlibinfo - Target Library Information
-tbaa - Type-Based Alias Analysis
-verify - Module Verifier
-view-cfg - View CFG of function
-view-cfg-only - View CFG of function (with no function bodies)
-view-dom - View dominance tree of function
-view-dom-only - View dominance tree of function (with no function bodies)
-view-postdom - View postdominance tree of function
-view-postdom-only - View postdominance tree of function (with no function bodies)
-view-regions - View regions of function

```
  -view-regions-only                - View regions of function (with no function bodies)
-time-passes                        - Time each pass, printing elapsed time for each on exit
-verify-dom-info                    - Verify dominator info (time consuming)
-verify-each                        - Verify after each transform
-verify-loop-info                   - Verify loop info (time consuming)
-verify-region-info                 - Verify region info (time consuming)
-version                           - Display the version of this program
```