```
// Multi-dimensional arrays
// Expected result: 156

#include <stdio.h>
int array[2][2][3] = { { {1, 2, 3}, {4, 5, 6} } , { {7, 8, 9}, {10, 11, 12} } };

int fct(int *array, int size) {
    int result = 0;
    int i;
    for (i = 0; i < size; i++) {
        result += array[i];
    }
    return result;
}

int main() {
    int result = 0;
    int a, b, c;
    for (a = 0; a < 2; a++) {
        for (b = 0; b < 2; b++) {
            for (c = 0; c < 3; c++) {
                result += array[a][b][c];
            }
        }
    }

    result += fct((int *)array, 12);

    printf("Result: %d\n", result);
    if (result == 156) {
        printf("RESULT: PASS\n");
    } else {
        printf("RESULT: FAIL\n");
    }
    return result;
}


../../llvm/Release+Asserts/bin/opt -O3 -time-passes array.bc > /dev/null
```

```
===-------------------------------------------------------------------------===
                      ... Pass execution timing report ...
===-------------------------------------------------------------------------===
  Total Execution Time: 0.0080 seconds (0.0221 wall clock)

   ---User Time---   --User+System--   ---Wall Time---  --- Name ---
   0.0000 (  0.0%)   0.0000 (  0.0%)   0.0146 ( 66.1%) Induction Variable Users 5
   0.0000 (  0.0%)   0.0000 (  0.0%)   0.0011 (  5.1%)  Induction Variable Simplification 4
   0.0000 (  0.0%)   0.0000 (  0.0%)   0.0008 (  3.4%)  Unroll loops 10
   0.0000 (  0.0%)   0.0000 (  0.0%)   0.0006 (  2.5%)  Function Integration/Inlining
   0.0000 (  0.0%)   0.0000 (  0.0%)   0.0005 (  2.3%)  Simplify the CFG
   0.0040 ( 50.0%)   0.0040 ( 50.0%)   0.0005 (  2.3%)  Global Value Numbering
   0.0000 (  0.0%)   0.0000 (  0.0%)   0.0004 (  1.9%)  Rotate Loops 9
   0.0000 (  0.0%)   0.0000 (  0.0%)   0.0003 (  1.6%)  Combine redundant instructions
   0.0000 (  0.0%)   0.0000 (  0.0%)   0.0003 (  1.4%)  Combine redundant instructions
   0.0000 (  0.0%)   0.0000 (  0.0%)   0.0002 (  1.0%)  Scalar Replacement of Aggregates (DT)
   0.0000 (  0.0%)   0.0000 (  0.0%)   0.0002 (  1.0%)  Combine redundant instructions
   0.0000 (  0.0%)   0.0000 (  0.0%)   0.0002 (  0.9%)  Canonicalize natural loops 2
   0.0000 (  0.0%)   0.0000 (  0.0%)   0.0002 (  0.9%)  Combine redundant instructions
   0.0000 (  0.0%)   0.0000 (  0.0%)   0.0002 (  0.8%)  Bitcode Writer
   0.0000 (  0.0%)   0.0000 (  0.0%)   0.0002 (  0.7%)  Natural Loop Information
   0.0040 ( 50.0%)   0.0040 ( 50.0%)   0.0002 (  0.7%)  Early CSE
   0.0000 (  0.0%)   0.0000 (  0.0%)   0.0001 (  0.6%)  Loop Invariant Code Motion 6
   0.0000 (  0.0%)   0.0000 (  0.0%)   0.0001 (  0.6%)  Value Propagation
   0.0000 (  0.0%)   0.0000 (  0.0%)   0.0001 (  0.6%)  Loop-Closed SSA Form Pass 7
   0.0000 (  0.0%)   0.0000 (  0.0%)   0.0001 (  0.6%)  Interprocedural Sparse Conditional Constant
Propagation
   0.0000 (  0.0%)   0.0000 (  0.0%)   0.0001 (  0.5%)  Combine redundant instructions
   0.0000 (  0.0%)   0.0000 (  0.0%)   0.0001 (  0.4%)  Simplify well-known library calls
   0.0000 (  0.0%)   0.0000 (  0.0%)   0.0001 (  0.3%)  Loop-Closed SSA Form Pass 7
   0.0000 (  0.0%)   0.0000 (  0.0%)   0.0001 (  0.3%)  Reassociate expressions
   0.0000 (  0.0%)   0.0000 (  0.0%)   0.0001 (  0.3%)  Dominator Tree Construction
   0.0000 (  0.0%)   0.0000 (  0.0%)   0.0001 (  0.2%)  Canonicalize natural loops
   0.0000 (  0.0%)   0.0000 (  0.0%)   0.0000 (  0.2%)  Early CSE
   0.0000 (  0.0%)   0.0000 (  0.0%)   0.0000 (  0.2%)  Dominator Tree Construction
   0.0000 (  0.0%)   0.0000 (  0.0%)   0.0000 (  0.2%)  Sparse Conditional Constant Propagation
   0.0000 (  0.0%)   0.0000 (  0.0%)   0.0000 (  0.2%)  Recognize loop idioms 8
   0.0000 (  0.0%)   0.0000 (  0.0%)   0.0000 (  0.2%)  Scalar Evolution Analysis
   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%)  No Alias Analysis (always returns 'may' alias)
   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%)  Jump Threading
   0.0000 (  0.0%)   0.0000 (  0.0%)   0.0000 (  0.1%)  Jump Threading
   0.0000 (  0.0%)   0.0000 (  0.0%)   0.0000 (  0.1%)  Dead Global 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%)  Value Propagation
   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%)  Loop-Closed SSA Form Pass 7
   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%)  Dominator Tree Construction
  0.0000 (  0.0%)   0.0000 (  0.0%)   0.0000 (  0.1%)  Basic Alias Analysis (stateless AA impl) 1
  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%)  Unswitch loops 11
  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%)  Delete dead loops 3
  0.0000 (  0.0%)   0.0000 (  0.0%)   0.0000 (  0.1%)  Lazy Value Information Analysis
  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 (  0.0%)  Tail Call Elimination
  0.0000 (  0.0%)   0.0000 (  0.0%)   0.0000 (  0.0%)  Basic CallGraph Construction
  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%)  Dead Store Elimination
  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%)  Aggressive Dead Code Elimination
  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%)  Global Variable Optimizer
  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%)  Dead Argument Elimination
  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%)  Lower 'expect' Intrinsics
  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%)  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%)  Target Library Information
  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%)  No Alias Analysis (always returns 'may' alias)
  0.0080 (100.0%)   0.0080 (100.0%)   0.0221 (100.0%)  Total
```

70 Passes Listed,48 Different Passes

-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.

-loop-simplify: Canonicalize natural loops

This pass performs several transformations to transform natural loops into a simpler form, which makes subsequent analyses and transformations simpler and more effective.

Loop pre-header insertion guarantees that there is a single, non-critical entry edge from outside of the loop to the loop header. This simplifies a number of analyses and transformations, such as LICM.

Loop exit-block insertion guarantees that all exit blocks from the loop (blocks which are outside of the loop that have predecessors inside of the loop) only have predecessors from inside of the loop (and are thus dominated by the loop header). This simplifies transformations such as store-sinking that are built into LICM.

This pass also guarantees that loops will have exactly one backedge.

Note that the simplifycfg pass will clean up blocks which are split out but end up being unnecessary, so usage of this pass should not pessimize generated code.

This pass obviously modifies the CFG, but updates loop information and dominator information.

-loop-deletion: Delete dead loops

This file implements the Dead Loop Deletion Pass. This pass is responsible for eliminating loops with non-infinite computable trip counts that have no side effects or volatile instructions, and do not contribute to the computation of the function's return value.

Induction Variable Simplification(Not Found)

-iv-users: Induction Variable Users

Bookkeeping for "interesting" users of expressions computed from induction variables.

-licm: Loop Invariant Code Motion

This pass performs loop invariant code motion, attempting to remove as much code from the body of a loop as possible. It does this by either hoisting code into the preheader block, or by sinking code to the exit blocks if it is safe. This pass also promotes must-aliased memory locations in the loop to live in registers, thus hoisting and sinking "invariant" loads and stores.

This pass uses alias analysis for two purposes:

- Moving loop invariant loads and calls out of loops. If we can determine that a load or call inside of a loop never aliases anything stored to, we can hoist it or sink it like any other instruction.
- Scalar Promotion of Memory - If there is a store instruction inside of the loop, we try to move the store to happen AFTER the loop instead of inside of the loop. This can only happen if a few conditions are true:
    - The pointer stored through is loop invariant.
    - There are no stores or loads in the loop which *may* alias the pointer. There are no calls in the loop which mod/ref the pointer.

  If these conditions are true, we can promote the loads and stores in the loop of the pointer to use a temporary alloca'd variable. We then use the mem2reg functionality to construct the appropriate SSA form for the variable.

-lcssa: Loop-Closed SSA Form Pass

This pass transforms loops by placing phi nodes at the end of the loops for all values that are live across the loop boundary. For example, it turns the left into the right code:

```
for (...)              for (...)
  if (c)                 if (c)
    X1 = ...               X1 = ...
  else                   else
    X2 = ...               X2 = ...
  X3 = phi(X1, X2)       X3 = phi(X1, X2)
```

```
... = X3 + 4            X4 = phi(X3)
                    ... = X4 + 4
```

This is still valid LLVM; the extra phi nodes are purely redundant, and will be trivially eliminated by InstCombine. The major benefit of this transformation is that it makes many other loop optimizations, such as LoopUnswitching, simpler.

Recognize loop idioms(Not Found)

-loop-rotate: Rotate Loops

A simple loop rotation transformation.

-loop-unroll: Unroll loops

This pass implements a simple loop unroller. It works best when loops have been canonicalized by the -indvars pass, allowing it to determine the trip counts of loops easily.

-loop-unswitch: Unswitch loops

This pass transforms loops that contain branches on loop-invariant conditions to have multiple loops. For example, it turns the left into the right code:

```
for (...)              if (lic)
  A                      for (...)
  if (lic)                 A; B; C
    B                  else
  C                      for (...)
                         A; C
```

This can increase the size of the code exponentially (doubling it every time a loop is unswitched) so we only unswitch if the resultant code will be smaller than a threshold.

This pass expects LICM to be run before it to hoist invariant conditions out of the loop, to make the unswitching opportunity obvious.

 # produces pre-link time optimization binary bitcode: array.prelto.bc
clang array.c -emit-llvm -c -fno-builtin -m32 -I ../lib/include/ -I/usr/include/i386-linux-gnu -O3 -mllvm -inline-threshold=-100 -o array.prelto.1.bc
 # linking may produce llvm mem-family intrinsics
../../llvm/Release+Asserts/bin/llvm-ld -disable-inlining -disable-opt array.prelto.1.bc -b=array.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 < array.prelto.linked.bc > array.prelto.bc
 # produces array.bc binary bitcode and a.out shell script: lli array.bc
../../llvm/Release+Asserts/bin/llvm-ld -disable-inlining -disable-opt array.prelto.bc ../lib/llvm/liblegup.a -b=array.bc
 # produces textual bitcodes: array.prelto.1.ll array.prelto.ll array.ll
../../llvm/Release+Asserts/bin/llvm-dis array.prelto.1.bc
../../llvm/Release+Asserts/bin/llvm-dis array.prelto.linked.bc
../../llvm/Release+Asserts/bin/llvm-dis array.prelto.bc
../../llvm/Release+Asserts/bin/llvm-dis array.bc
 # produces verilog: array.v
../../llvm/Release+Asserts/bin/llc -legup-config=../../hwtest/CycloneII.tcl -legup-config=../legup.tcl -march=v array.bc -o array.v

Output for -O0:
scheduling.legup.rpt:
50 states, 47 registers
modelsim:
Result: 156
RESULT: PASS
At t= 875000 clk=1 finish=1 return_val= 156
** Note: $finish : array.v(3591)
Time: 875 ns Iteration: 3 Instance: /main_tb

Output for -O3:
scheduling.legup.rpt:
19 states, 32 registers(register's number is not accurate)

modelsim:
Result: 156
RESULT: PASS
At t= 117000 clk=1 finish=1 return_val= 156
** Note: $finish : array.v(1945)
Time: 117 ns Iteration: 3 Instance: /main_tb