

CS5544: Assignment 3
Programming part

Group: Abhijit Tripathy , Swati Lodha

Evaluation

We have the LLVM interpreter to collect dynamic instruction counts for each benchmark.

5.1 Dominators Pass

Definition of a Dominator: In a given CFG, a node d dominates a node n (denoted as $d \text{ dom } n$), if every path from entry to node n , has to go through node d . Intuitively, we can also say that $n \text{ dom } n$.

Dominators can be found using an iterative data flow algorithm, as described below:

1. Meet Function: Intersection; The set of dominators at the entry of a basic block, is the intersection of the set of dominators at the exit of each of its predecessor blocks.
2. Transfer function: Each basic block only adds itself to the set of dominators it receives. Therefore, $OUT[B] = \{b\} \cup IN[B]$
3. The Pass direction is **FORWARD**.
4. Domain for this algorithm is the Power Set of all Basic Blocks.
5. Boundary condition is: $OUT[ENTRY] = \{ENTRY\}$
6. Since meet operator is Intersection, the initial conditions would be: $OUT[B] = U$, where U is the set of all basic blocks in the function.

We use the Dataflow framework library from our previous assignment to implement the Dominators pass. We have also created a dominator map (map<string, set<string>>), which is a map of BasicBlock names and the names of their dominator blocks. The Dominator pass can be used by other passes to utilize the dominator map object. Besides the dominator map, we also print the immediate dominator of each basic block in a loop. A node n has an immediate dominator m , if, $\forall d$, where d is a dominator of n and $d \neq n$, then $d \text{ dom } m$.

The Dominator Pass Algorithm can be summarized as:

OUT[ENTRY] = {ENTRY}

IN[ENTRY] = \emptyset

For all BB other than ENTRY, OUT[BB] = U

While OUT[] changes:

For all BB:

$$IN[BB] = \bigcap_{p \in predecessors(BB)} OUT[p]$$

$$OUT[BB] = \{B\} \cup IN[BB]$$

We have written two microbenchmarks to test Dominators Pass.

Benchmark 1:

```
#include "stdio.h"
int test(int x) {
    int p = 0;
    for (int i = 0; i < 100; i++) {
        p = x * 2;
    }
    return p;
}

int main() {
    int res = test(4);
    printf("The result is %d\n", res);
    return 0;
}
```

This benchmark refers to a simple single loop code, to test whether the Dominator pass is working correctly.

Benchmark 2:

```
#include "stdio.h"

int test(int x) {
    int p = 0;
    for (int i = 0; i < 100; i++){
        if (i < 50){
            for (int j = 0; j < 100; j++){
                p = x * 2;
            }
        }
    }
    int q = 0;
    for (int k = 0; k < 100; k++){
        q = x * 2;
    }
    return p * q;
}

int main(){
    int res = test(4);
    printf("The result is %d\n", res);
    return 0;
}
```

}

The second benchmark contains more complicated condition and loop structures, with nested conditional loops, and multiple outer loops.

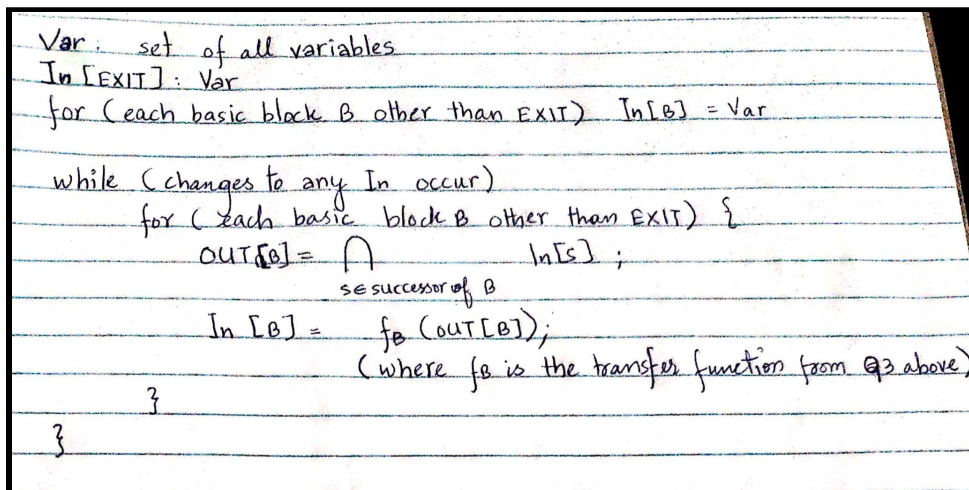
5.2 Dead Code Elimination Pass

Dead Code Elimination Pass is implemented using the Faint analysis.

For faint analysis, our domain is the set of variables and is done in backward direction. We use the following Transfer function for the pass: $f(x) = (x - \text{Kill}(x)) \cup \text{Gen}(x)$

The meet operator is Intersection and for the output, we take an intersection of the input of all successors. In[EXIT], Out[Entry] and Out[Entry] is the set of all variables. The In set of all basic blocks is also set to all variables. Since we want to visit all the successors of a node before the node itself, we do a post-order traversal.

Here's the pseudo-code for the faint analysis:



```
Var: set of all variables
In[EXIT]: Var
for (each basic block B other than EXIT) In[B] = Var

while (changes to any In occur)
  for (each basic block B other than EXIT) {
    OUT[B] =  $\bigcap_{s \in \text{successor of } B} \text{In}[s]$ ;
    In[B] =  $f_B(\text{OUT}[B])$ ;
    (where  $f_B$  is the transfer function from Q3 above)
  }
}
```

We first set up the domain with all the live instructions for implementing the dead code pass. We have an info map that we initialize post domain setup. In this step, we initialize the GenSet and KillSet for each basic block. We do a post-order traversal, as faint analysis is done in the backward direction.

1. For all the operands in each instruction, we handle the phi nodes separately. If it's a phi node, we get each incoming value and set the KillSet bit vector index if it's already present in the domain.
2. If the instruction is present in the domain and the kill set isn't set for the same, we set the genset
3. For all operands in each instruction, if the value of operand is present in the domain, we set the killSet for that instruction

Finally, we add the updated structure to the infoMap. We keep doing this for all the instructions.

We initialize the boundary and init conditions to be the set of all variables. We then create a new DeadCodeEliminationAnalysis variable with required conditions and run the dataflow pass.

Now, we eliminate the dead code instructions and preserve the live ones. We have the faint value for all the basic blocks from the data flow pass output. We have a temporary vector to store the instructions to delete. We go over all the instructions in the code, we skip the live instructions and see if the faint value is set. If yes, then we add those instructions to the temporary vector.

At the end of the loop, we go over all the instructions in the temporary vector. If there's no further use, then we replace the value with "undef" and finally erase them.

Microbenchmarks

We have added 2 microbenchmarks for dead code elimination. Measured the performance gain using dynamic instruction count:

Benchmark #1

```
int a = 0;
int b = 0;
for(int i=0; i<10; i++)
{
    a = x * 2;
    b = a + 5; //dead code
}
return a;
```

In this example, the instruction count for unoptimized code is 77 and the instruction count for optimized code is 67. The reason is because the statement "b = a + 5" inside the loop is dead code (Value of b not used) and the loop is run 10 times. So, in the optimized code, the statement is eliminated and hence we see a difference of 10 instruction counts.

Benchmark #2

```
int a = 0;
int b = 0;
b = a + 5; //dead code
b = b + 1; //dead code
return -1;
```

In this example, the instruction count for unoptimized code is 6 and the instruction count for optimized code is 4. The reason is because the statements “ $b = a + 5$ ” and “ $b = b + 1$ ” are dead code (Value of b not used) as we return the constant at the end of the variable. These two statements are removed from the optimized code and hence we see a difference of 2 instruction counts.

5.3 Loop Invariant Code Motion Pass

A computation is termed loop-invariant if its value does not change as long as the control stays within the loop. Such invariant instructions are then moved to the loop preheader so that it is computed only once.

We use the following criteria to evaluate if a loop computation is invariant:

An instruction inside a loop is invariant if each of its operands is:

1. Constant: A constant value does not change across loop iterations.
2. Defined outside the loop: If all definitions of the operand reaching the instruction are coming from outside the loop, then it won't change across iterations.
3. Defined inside the loop but the definition itself is invariant: If the operand is defined inside the loop, but the definition itself is invariant, then the operand's value won't change across loop iterations.

We modify these rules for the SSA form.

SSA states that a variable is only assigned once inside a program body, and a variable's definition will always dominate its use.

Therefore, we make the following changes to the criteria:

First, we store all loop instructions in a list, say, `loopInstructions`. We also traverse the loop in post order fashion and keep track of loop invariant instructions seen so far in a variable `loopInv`.

Criteria 1 will stay the same for the SSA form.

Criteria 2 - If `loopInstructions` do not contain the operand, it implies that the operand is defined in an instruction outside the loop, and is thus invariant.

Criteria 3 - If `loopInstructions` contains the operand, and so does `loopInv`, then the current instruction is also invariant.

Once we have identified the loop invariant instructions, we need to evaluate potential candidates for code motion. All loop invariant instructions cannot be moved to the pre-header, and we use the following criteria to decide code motion candidates:

A loop invariant instruction is a candidate for code motion if

1. It dominates the loop exit. If an instruction does not dominate loop exit, then hoisting it in the pre-header, would cause a control path that does not enter the loop at all to have an incorrect computation in its path. This could cause issues, especially with risky instructions like division, where the loop header might have a condition to check for *divide-by-zero* cases.

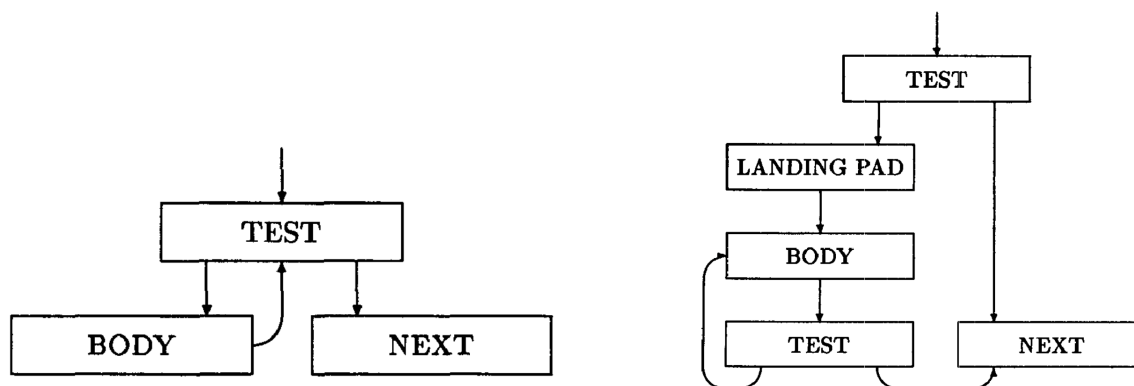
2. It dominates all its uses. If an instruction does not dominate its use, then a control flow along the preheader would assign a computation to the variable before its use and can cause incorrect computations.
3. It does not have multiple definitions inside the loop. If an instruction has multiple definitions inside the loop, then hoisting it to the pre-header will cause incorrect computation.

These rules will be modified for the SSA form, however, we should first discuss another LLVM pass that will simplify the LICM algorithm.

5.3.1. Landing Pad Transformation Pass:

One issue with our current discussion of LICM is if the loop is not executed even once, then also the loop invariant instruction will be executed in the pre-header, which might lead to unpredictable program behavior. One way to fix this issue is to insert a block between the loop's pre-header and header and add a test inside the pre-header block so that control only flows through the new block and the rest of the loop if the loop executes at least once, otherwise it branches off to loop exit. This new block is termed a landing block, and the associated loop transformation is called Landing Pad Transformation Pass.

A typical loop and a Landing pad transformed loop structure are described below:



In LLVM loop terms, the TEST block is also termed as the loop header. Moreover, the body is split into two parts:

1. Loop Body, without the back-edge.
2. Loop Latch, with a back-edge to Loop Header/TEST.

In the figure, we are doing the following operations on the loop:

1. We rotate the loop to form a do-while loop structure, therefore, the test conditions are now moved to the loop latch.
2. We insert a landing pad block between the loop entry and the preheader. We also move the test conditions from the loop header to the pre-header block.

Since the Loop exit now has two incoming edges, we need to join the edges using Phi instructions. This is followed by, replacing the usage of previous instructions outside the loop, with the new Phi variables.

Assuming the following terminology:

PreHeader: Loop preheader

Header: Loop Entry Block with the test conditions for the loop.

Latch: Loop block that has a back-edge to the Loop Header block.

The algorithm can be described in the following steps:

1. Insert a new block between PreHeader and Header, call it LandingPad. This is where the loop invariant computations will be hoisted.
2. Clone non-Phi instructions (test conditions) from the Header to the Latch.
3. Move non-Phi instructions from the Header to the PreHeader.
4. Unify the incoming edges at Loop Exit.
5. Replace usage of Header instructions with the Phi instructions created in Step 4.

The advantage of Landing Pad Transformation is: loop instructions that do not dominate all exits can also be moved to the landing pad, as this block will be executed only if the loop runs at least once.

5.3.2. Impact of Landing Pad and SSA form on Code Motion Algorithm.

The following changes happen to each criterion for Code Motion described in 5.3:

1. Criteria 1 is no longer required, as Landing Pad Transformation ensures that instructions moved to the landing pad are only executed if the loop executes at least once, therefore they don't need to dominate loop exits.
2. Criteria 2 is also relaxed according to SSA rules: all definitions must dominate their uses.
3. Criteria 3 is also relaxed since SSA states that any variable is only assigned once in the entire program.

With this information, we can continue with the loop invariant code motion algorithm, and hoist all invariant computations to the loop landing pad. Therefore, our LICM implementation can only run on bit code that has already been transformed by the Landing Pad Transformation pass.

We use the following three benchmarks to evaluate our implementation.

Benchmark 1:

```
#include "stdio.h"

int test(int x)
{
    int p = 0;
    for (int i = 0; i < 100; i++)
    {
        p = x * 2; // Loop-invariant
    }
}
```

```

        return p;
    }

int main()
{
    int res = test(4);
    printf("The result is %d\n", res);
    return 0;
}

```

This benchmark has a single loop structure with one invariant instruction. We run this benchmark to evaluate the correctness of our algorithm. In our evaluation, the optimized code ran almost 100 less dynamic instructions, compared to the unoptimized code. The improvement is also equal to the number of iterations of the loop, which shows that our implementation saves redundant loop computations (optimized = 514, unoptimized = 607).

Benchmark 2:

```

#include "stdio.h"

int test(int x)
{
    int p = 0;
    for (int i = 0; i < 100; i++)
    {
        for (int j = 0; j < 100; j++)
        {
            p = x * 2; // Loop-invariant
        }
    }
    return p;
}

int main()
{
    int res = test(4);
    printf("The result is %d\n", res);
    return 0;
}

```

This benchmark involves an outer loop and an inner loop, and the loop invariant instruction is placed inside the inner loop. We run this benchmark to evaluate if deeply nested invariant computations are able to bubble out to the landing pad. The dynamic instruction count for the unoptimized code is 60807, while for the optimized code, it is 51923. We can see that improvement is in the order of the number of loop iterations. We have attached a CFG of the optimized and unoptimized bitcodes for this benchmark in the writeup.

Benchmark 3:

```

#include "stdio.h"

int test(int x)
{

```



```

int p = 0;
int q = 0;
for (int i = 0; i < 100; i++)
{
    for (int j = 0; j < 100; j++)
    {
        p = x * 2; // Loop-invariant
        q = p + 2; // Loop-invariant
    }
}
return p;
}

int main()
{
    int res = test(4);
    printf("The result is %d\n", res);
    return 0;
}

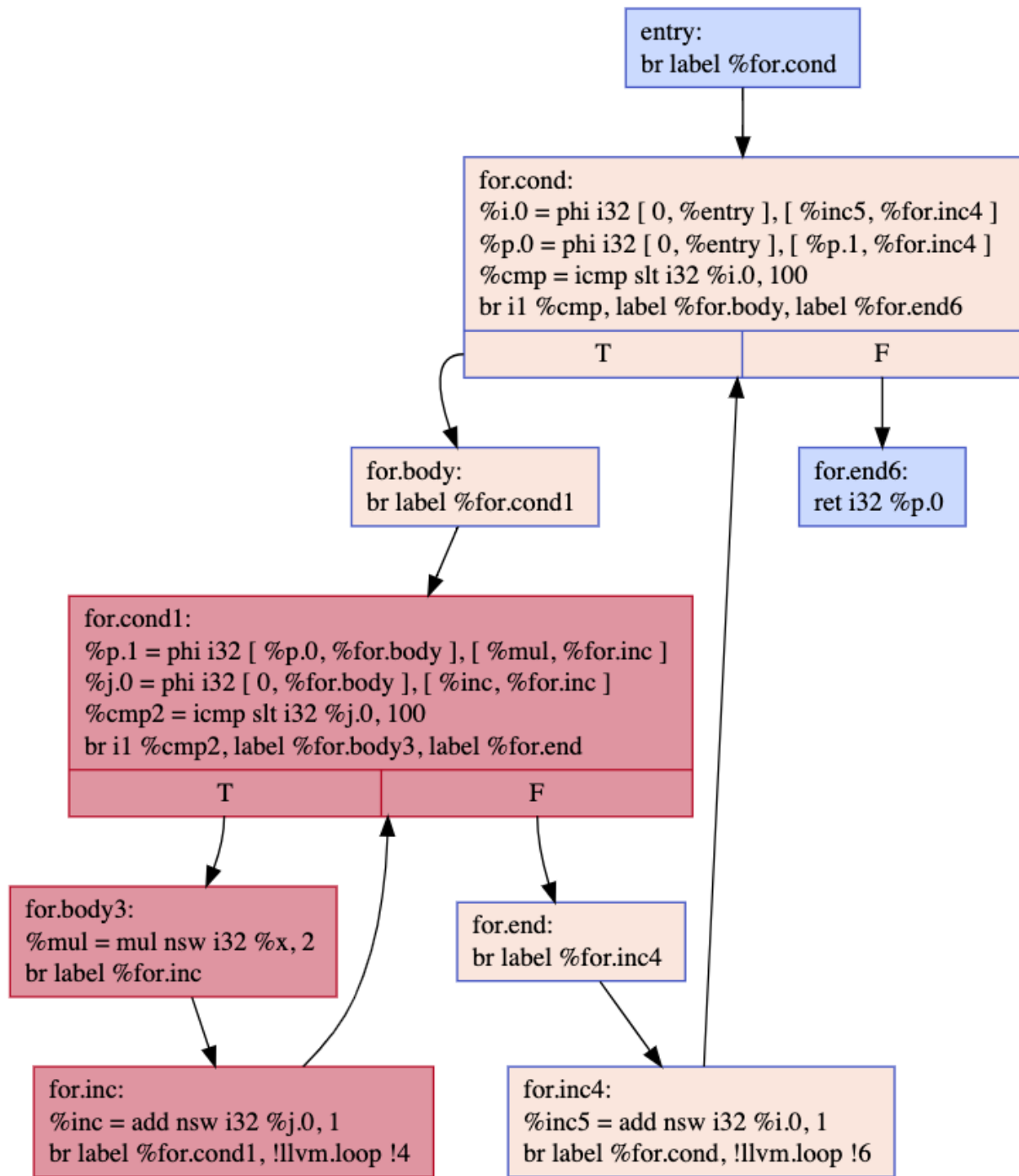
```

This benchmark is similar to benchmark 2, but has an additional invariant instruction, that depends on the first invariant instruction. The dynamic instruction count for the unoptimized code is 70807, while for the optimized code it is 51924. This also shows the improvement is equivalent to the number of loop iterations.

The evaluation result for all the benchmarks for Dead Code Elimination and Loop Invariant Code Motion is given below :

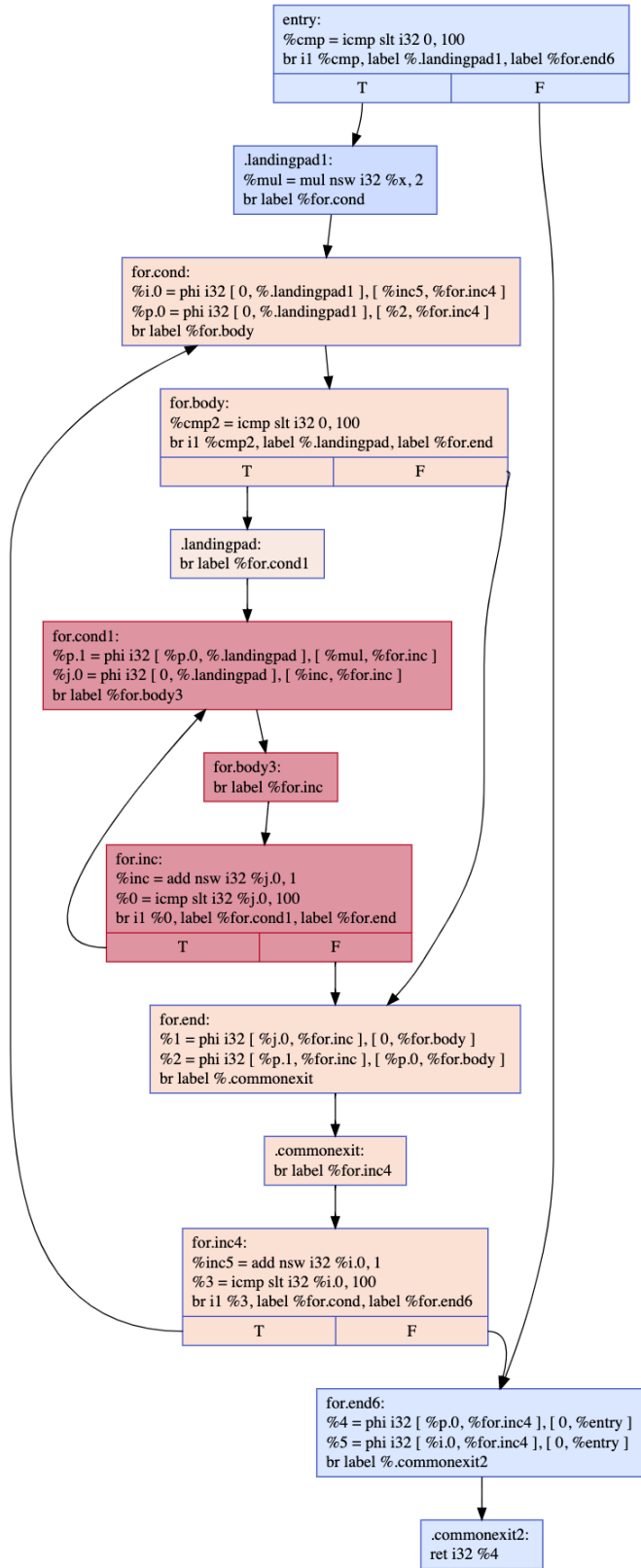
Benchmark Results

Pass	Test file	Dynamic Instruction Count	
		Unoptimized	Optimized
LICM	benchmark1.c	607	514
	benchmark2.c	60807	51923
	benchmark3.c	70807	51924
DCE	test_dce1.c	77	67
	test_dce2.c	6	4



CFG for 'test' function

CFG for unoptimized Benchmark 2 of LICM.



CFG for optimized Benchmark 2 for LICM