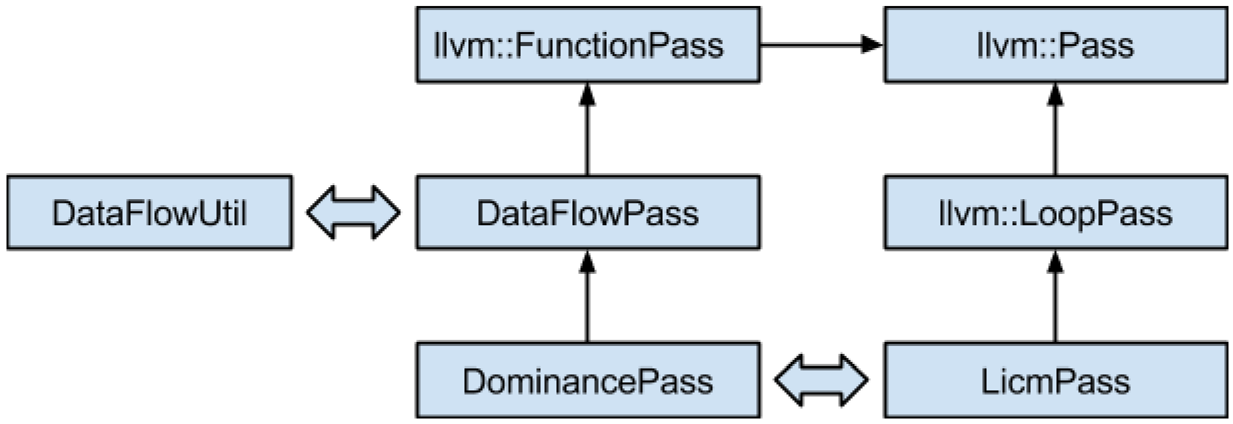
Writeup
15-745 Homework 3
Aditya Bhandaru (akbhanda), Zhe Qian (zheq)

LICM: Loop Invariant Code Motion

The architecture diagram below shows the relationship between modules in the program.



Our `LicmPass` is not dependent on the `DominancePass` running first. Instead we instantiate a `DominancePass` object and call the public `runOnBlocks()` function to retrieve the data. We arrange things like this for a couple reasons:

- The `DominancePass` class is more or less stateless between invocations of `runOnFunction()` and `runOnBlocks()`
- We build and run `DominancePass` as a separate pass.
- The above points make each pass easier to test separately for correctness.

The LicmPass instantiates a `DominancePass` object, and invokes the dataflow functionality using `runOnBlocks` while passing in a vector of `BasicBlock`s belonging to the current `Loop` object. The `DominancePass` then returns a full dominator tree ready for traversal.

LICM Algorithm

Once we have the dominator tree, we do a pre-order traversal. With this approach, we know we are seeing the definitions for every operand before they are used. We compute the reaching definitions using this approach. For each node in the dominator tree, we inspect the block and traverse its instructions in forward order. We then inspect each instruction to determine invariance (i.e. does it meet the criteria described in the handout and in class).

A breakdown of the checks is given below.

- `isSafeToSpeculativelyExecute()` - Determine if executing the given instruction is safe even though it may not be used. This can save us from throwing exceptions where we shouldn't after code motion.

- `!I->mayReadFromMemory()` - We cannot reason about changes in memory during execution. This is pertinent in multithreading environments, or even interrupts.
- `!isa<LandingPadInst>(I)` - Cannot move the instruction if it is the target of a jump or something.
- The operands of this instruction are invariant. That is, they were either already marked as invariant, they are not instructions, or they are outside the loop.

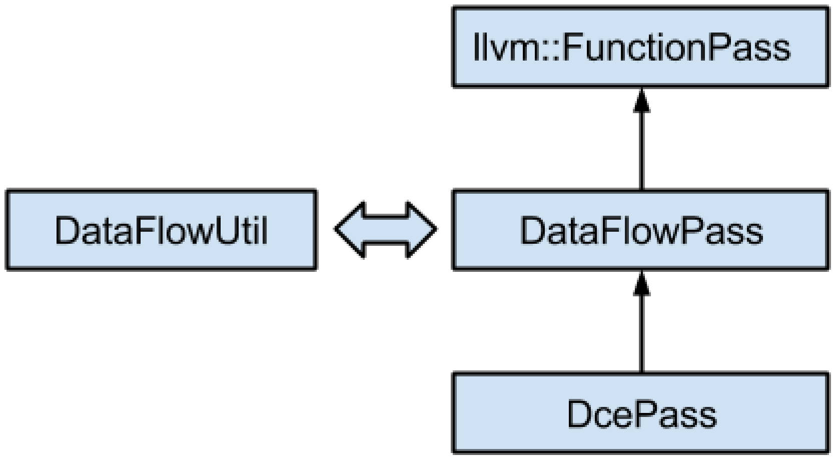
Finally we accumulate a set of instructions that are marked as invariant for each loop, started with the deepest nested loops. We move all these instructions to the preheader. As we process loops of shallower depth, they can then analyze the newly exposed invariant instructions in the preheaders of subloops to allow for efficient bubbling.

LICM: Results
You can find the source listing and descriptions for benchmarks in the appendix. The results are shown below.

Benchmark	**Instructions exe. before LICM**	**Instructions exe. after LICM**
licm-bench1.c	4294	4024
licm-bench2.c	29882	23484
licm-bench3.c	623	513

DCE: Dead Code Elimination

The architecture diagram for the dead code elimination pass is below.



Like our previous assignment (Homework 2), we have the DcePass derive from our DataFlowPass abstract class. The DcePass class is more like a wrapper for liveness dataflow analysis. Once it does the liveness analysis on a BasicBlock granularity, it has logic to determine the liveness for particular instructions in that block.

DCE: Implementation

Then, for the given function, we iterate through all the instruction in order. For each instruction, we see if it meets **any** of the following criteria.

- `isa<TerminatorInst>(I) ||`
- `isa<DbgInfoIntrinsic>(I) ||`
- `isa<LandingPadInst>(I) ||`
- `I->mayHaveSideEffects()`
- The instruction is live at the end of the basic block, or something subsequently kills it within that basic block.

For instructions that are not live, we can safely erase them from the parent block.

DCE: Benchmarks

You can find the source listing and description for benchmarks in the appendix. The results are shown below.

Benchmark	**Instructions exe. before LICM**	**Instructions exe. after LICM**
dce-bench1.c	31	6
dce-bench2.c	194	99
dce-bench3.c	16030	6008

Appendix

LICM: Benchmark 1

Proof of concept nested loops with bubbling. The invariant instructions are labeled in the code as well as how far they should bubble.

licm-bench1.c

```
int loop (int a, int b, int c) {
  int i;
  int ret = 0;
  for (i = a; i < b; i++) {
    int x = c + 4;   // should bubble above i loop
    int j, k;
    for (j = i; j < b; j++) {
      int y = x + i; // should bubble above j loop
      int z = x + a; // should bubble above i loop
      ret += (i * j) + y;
    }
    for (k = i; k < c; k++) {
      ret += 1;
    }
  }
  return ret;
}

int main(int argc, char const *argv[]) {
  int a = loop(4, 20, 40);
  return a;
}
```

licm-bench1.ll (before)

```
; ModuleID = 'licm-bench1.reg.bc'
target datalayout =
"e-p:32:32:32-i1:8:8-i8:8:8-i16:16:16-i32:32:32-i64:32:64-f32:32:32-f64:32:64-v64:64:64-v
128:128:128-a0:0:64-f80:32:32-n8:16:32-S128"
target triple = "i386-pc-linux-gnu"

; Function Attrs: nounwind
define i32 @loop(i32 %a, i32 %b, i32 %c) #0 {
entry:
  br label %for.cond

for.cond:                                         ; preds = %for.inc15, %entry
  %i.0 = phi i32 [ %a, %entry ], [ %inc16, %for.inc15 ]
  %ret.0 = phi i32 [ 0, %entry ], [ %ret.2, %for.inc15 ]
  %cmp = icmp slt i32 %i.0, %b
  br i1 %cmp, label %for.body, label %for.end17

for.body:                                         ; preds = %for.cond
  %add = add nsw i32 %c, 4
  br label %for.cond1

for.cond1:                                        ; preds = %for.inc, %for.body
  %ret.1 = phi i32 [ %ret.0, %for.body ], [ %add7, %for.inc ]
  %j.0 = phi i32 [ %i.0, %for.body ], [ %inc, %for.inc ]
  %cmp2 = icmp slt i32 %j.0, %b
  br i1 %cmp2, label %for.body3, label %for.end

for.body3:                                        ; preds = %for.cond1
  %add4 = add nsw i32 %add, %i.0
  %add5 = add nsw i32 %add, %a
  %mul = mul nsw i32 %i.0, %j.0
  %add6 = add nsw i32 %mul, %add4
  %add7 = add nsw i32 %ret.1, %add6
  br label %for.inc

for.inc:                                          ; preds = %for.body3
  %inc = add nsw i32 %j.0, 1
  br label %for.cond1

for.end:                                          ; preds = %for.cond1
  br label %for.cond8

for.cond8:                                        ; preds = %for.inc12, %for.end
  %ret.2 = phi i32 [ %ret.1, %for.end ], [ %add11, %for.inc12 ]
  %k.0 = phi i32 [ %i.0, %for.end ], [ %inc13, %for.inc12 ]
  %cmp9 = icmp slt i32 %k.0, %c
  br i1 %cmp9, label %for.body10, label %for.end14

for.body10:                                       ; preds = %for.cond8
  %add11 = add nsw i32 %ret.2, 1
  br label %for.inc12

for.inc12:                                        ; preds = %for.body10
  %inc13 = add nsw i32 %k.0, 1
  br label %for.cond8

for.end14:                                        ; preds = %for.cond8
  br label %for.inc15

for.inc15:                                        ; preds = %for.end14
  %inc16 = add nsw i32 %i.0, 1
  br label %for.cond

for.end17:                                        ; preds = %for.cond
  ret i32 %ret.0
}

; Function Attrs: nounwind
define i32 @main(i32 %argc, i8** %argv) #0 {
entry:
  %call = call i32 @loop(i32 4, i32 20, i32 40)
  ret i32 %call
}

attributes #0 = { nounwind "less-precise-fpmad"="false" "no-frame-pointer-elim"="true"
"no-frame-pointer-elim-non-leaf" "no-infs-fp-math"="false" "no-nans-fp-math"="false"
"stack-protector-buffer-size"="8" "unsafe-fp-math"="false" "use-soft-float"="false" }

!llvm.ident = !{!0}
```

```
!0 = metadata !{metadata !"clang version 3.4 (tags/RELEASE_34/final)"}
```

licm-bench1.ll (after)

```
; ModuleID = 'out.bc'
target datalayout =
"e-p:32:32:32-i1:8:8-i8:8:8-i16:16:16-i32:32:32-i64:32:64-f32:32:32-f64:32:64-v64:64:64-v
128:128:128-a0:0:64-f80:32:32-n8:16:32-S128"
target triple = "i386-pc-linux-gnu"

; Function Attrs: nounwind
define i32 @loop(i32 %a, i32 %b, i32 %c) #0 {
entry:
  %add = add nsw i32 %c, 4
  %add5 = add nsw i32 %add, %a
  br label %for.cond

for.cond:                                         ; preds = %for.inc15, %entry
  %i.0 = phi i32 [ %a, %entry ], [ %inc16, %for.inc15 ]
  %ret.0 = phi i32 [ 0, %entry ], [ %ret.2, %for.inc15 ]
  %cmp = icmp slt i32 %i.0, %b
  br i1 %cmp, label %for.body, label %for.end17

for.body:                                         ; preds = %for.cond
  %add4 = add nsw i32 %add, %i.0
  br label %for.cond1

for.cond1:                                        ; preds = %for.inc, %for.body
  %ret.1 = phi i32 [ %ret.0, %for.body ], [ %add7, %for.inc ]
  %j.0 = phi i32 [ %i.0, %for.body ], [ %inc, %for.inc ]
  %cmp2 = icmp slt i32 %j.0, %b
  br i1 %cmp2, label %for.body3, label %for.end

for.body3:                                        ; preds = %for.cond1
  %mul = mul nsw i32 %i.0, %j.0
  %add6 = add nsw i32 %mul, %add4
  %add7 = add nsw i32 %ret.1, %add6
  br label %for.inc

for.inc:                                          ; preds = %for.body3
  %inc = add nsw i32 %j.0, 1
  br label %for.cond1

for.end:                                          ; preds = %for.cond1
  br label %for.cond8

for.cond8:                                        ; preds = %for.inc12, %for.end
  %ret.2 = phi i32 [ %ret.1, %for.end ], [ %add11, %for.inc12 ]
  %k.0 = phi i32 [ %i.0, %for.end ], [ %inc13, %for.inc12 ]
  %cmp9 = icmp slt i32 %k.0, %c
  br i1 %cmp9, label %for.body10, label %for.end14

for.body10:                                       ; preds = %for.cond8
  %add11 = add nsw i32 %ret.2, 1
  br label %for.inc12

for.inc12:                                        ; preds = %for.body10
```

```
  %inc13 = add nsw i32 %k.0, 1
  br label %for.cond8

for.end14:                                        ; preds = %for.cond8
  br label %for.inc15

for.inc15:                                        ; preds = %for.end14
  %inc16 = add nsw i32 %i.0, 1
  br label %for.cond

for.end17:                                        ; preds = %for.cond
  ret i32 %ret.0
}

; Function Attrs: nounwind
define i32 @main(i32 %argc, i8** %argv) #0 {
entry:
  %call = call i32 @loop(i32 4, i32 20, i32 40)
  ret i32 %call
}

attributes #0 = { nounwind "less-precise-fpmad"="false" "no-frame-pointer-elim"="true"
"no-frame-pointer-elim-non-leaf" "no-infs-fp-math"="false" "no-nans-fp-math"="false"
"stack-protector-buffer-size"="8" "unsafe-fp-math"="false" "use-soft-float"="false" }

!llvm.ident = !{!0}

!0 = metadata !{metadata !"clang version 3.4 (tags/RELEASE_34/final)"}
```

LICM: Benchmark 2

Like before we use for loops with clear induction variables. Here we illustrate a triple for loop. This should test how well we bubble instructions. It is also much longer than the previous test.

licm-bench2.c

```
int loop (int a, int b, int c) {
  int i;
  int ret = 0;
  for (i = 0; i < a; i++) {
    int x = c + 4;   // should bubble above i loop
    int j, k;
    for (j = 0; j < b; j++) {
      int y = x + i; // should bubble above j loop
      int z = x + a; // should bubble above i loop
      ret += (i * j) + y;
      for (k = 0; k < c; k++) {
        int w = (i + j) * c; // should bubble above k
        int r = z + k;
        ret *= r;
      }
    }
  }
  return ret;
}
```

```
int main(int argc, char const *argv[]) {
  int a = loop(4, 20, 40);
  return a;
}
```

licm-bench2.ll (before)

```
; ModuleID = 'licm-bench2.reg.bc'
target datalayout =
"e-p:32:32:32-i1:8:8-i8:8:8-i16:16:16-i32:32:32-i64:32:64-f32:32:32-f64:32:64-v64:64:64-v
128:128:128-a0:0:64-f80:32:32-n8:16:32-S128"
target triple = "i386-pc-linux-gnu"

; Function Attrs: nounwind
define i32 @loop(i32 %a, i32 %b, i32 %c) #0 {
entry:
  br label %for.cond

for.cond:                                         ; preds = %for.inc18, %entry
  %i.0 = phi i32 [ 0, %entry ], [ %inc19, %for.inc18 ]
  %ret.0 = phi i32 [ 0, %entry ], [ %ret.1, %for.inc18 ]
  %cmp = icmp slt i32 %i.0, %a
  br i1 %cmp, label %for.body, label %for.end20

for.body:                                         ; preds = %for.cond
  %add = add nsw i32 %c, 4
  br label %for.cond1

for.cond1:                                        ; preds = %for.inc15, %for.body
  %ret.1 = phi i32 [ %ret.0, %for.body ], [ %ret.2, %for.inc15 ]
  %j.0 = phi i32 [ 0, %for.body ], [ %inc16, %for.inc15 ]
  %cmp2 = icmp slt i32 %j.0, %b
  br i1 %cmp2, label %for.body3, label %for.end17

for.body3:                                        ; preds = %for.cond1
  %add4 = add nsw i32 %add, %i.0
  %add5 = add nsw i32 %add, %a
  %mul = mul nsw i32 %i.0, %j.0
  %add6 = add nsw i32 %mul, %add4
  %add7 = add nsw i32 %ret.1, %add6
  br label %for.cond8

for.cond8:                                        ; preds = %for.inc, %for.body3
  %ret.2 = phi i32 [ %add7, %for.body3 ], [ %mul14, %for.inc ]
  %k.0 = phi i32 [ 0, %for.body3 ], [ %inc, %for.inc ]
  %cmp9 = icmp slt i32 %k.0, %c
  br i1 %cmp9, label %for.body10, label %for.end

for.body10:                                       ; preds = %for.cond8
  %add11 = add nsw i32 %i.0, %j.0
  %mul12 = mul nsw i32 %add11, %c
  %add13 = add nsw i32 %add5, %k.0
  %mul14 = mul nsw i32 %ret.2, %add13
  br label %for.inc

for.inc:                                          ; preds = %for.body10
  %inc = add nsw i32 %k.0, 1
  br label %for.cond8

for.end:                                          ; preds = %for.cond8
  br label %for.inc15

for.inc15:                                        ; preds = %for.end
  %inc16 = add nsw i32 %j.0, 1
  br label %for.cond1

for.end17:                                        ; preds = %for.cond1
  br label %for.inc18

for.inc18:                                        ; preds = %for.end17
  %inc19 = add nsw i32 %i.0, 1
  br label %for.cond

for.end20:                                        ; preds = %for.cond
  ret i32 %ret.0
}

; Function Attrs: nounwind
define i32 @main(i32 %argc, i8** %argv) #0 {
entry:
  %call = call i32 @loop(i32 4, i32 20, i32 40)
  ret i32 %call
}

attributes #0 = { nounwind "less-precise-fpmad"="false" "no-frame-pointer-elim"="true"
"no-frame-pointer-elim-non-leaf" "no-infs-fp-math"="false" "no-nans-fp-math"="false"
"stack-protector-buffer-size"="8" "unsafe-fp-math"="false" "use-soft-float"="false" }

!llvm.ident = !{!0}

!0 = metadata !{metadata !"clang version 3.4 (tags/RELEASE_34/final)"}
```

licm-bench2.ll (after)

```
; ModuleID = 'out.bc'
target datalayout =
"e-p:32:32:32-i1:8:8-i8:8:8-i16:16:16-i32:32:32-i64:32:64-f32:32:32-f64:32:64-v64:64:64-v
128:128:128-a0:0:64-f80:32:32-n8:16:32-S128"
target triple = "i386-pc-linux-gnu"

; Function Attrs: nounwind
define i32 @loop(i32 %a, i32 %b, i32 %c) #0 {
entry:
  %add = add nsw i32 %c, 4
  %add5 = add nsw i32 %add, %a
  br label %for.cond

for.cond:                                         ; preds = %for.inc18, %entry
  %i.0 = phi i32 [ 0, %entry ], [ %inc19, %for.inc18 ]
  %ret.0 = phi i32 [ 0, %entry ], [ %ret.1, %for.inc18 ]
  %cmp = icmp slt i32 %i.0, %a
  br i1 %cmp, label %for.body, label %for.end20
for.body:                                         ; preds = %for.cond
  %add4 = add nsw i32 %add, %i.0
  br label %for.cond1

for.cond1:                                        ; preds = %for.inc15, %for.body
  %ret.1 = phi i32 [ %ret.0, %for.body ], [ %ret.2, %for.inc15 ]
  %j.0 = phi i32 [ 0, %for.body ], [ %inc16, %for.inc15 ]
  %cmp2 = icmp slt i32 %j.0, %b
  br i1 %cmp2, label %for.body3, label %for.end17

for.body3:                                        ; preds = %for.cond1
  %mul = mul nsw i32 %i.0, %j.0
  %add6 = add nsw i32 %mul, %add4
  %add7 = add nsw i32 %ret.1, %add6
  %add11 = add nsw i32 %i.0, %j.0
  %mul12 = mul nsw i32 %add11, %c
  br label %for.cond8

for.cond8:                                        ; preds = %for.inc, %for.body3
  %ret.2 = phi i32 [ %add7, %for.body3 ], [ %mul14, %for.inc ]
  %k.0 = phi i32 [ 0, %for.body3 ], [ %inc, %for.inc ]
  %cmp9 = icmp slt i32 %k.0, %c
  br i1 %cmp9, label %for.body10, label %for.end

for.body10:                                       ; preds = %for.cond8
  %add13 = add nsw i32 %add5, %k.0
  %mul14 = mul nsw i32 %ret.2, %add13
  br label %for.inc

for.inc:                                          ; preds = %for.body10
  %inc = add nsw i32 %k.0, 1
  br label %for.cond8

for.end:                                          ; preds = %for.cond8
  br label %for.inc15

for.inc15:                                        ; preds = %for.end
  %inc16 = add nsw i32 %j.0, 1
  br label %for.cond1

for.end17:                                        ; preds = %for.cond1
  br label %for.inc18

for.inc18:                                        ; preds = %for.end17
  %inc19 = add nsw i32 %i.0, 1
  br label %for.cond

for.end20:                                        ; preds = %for.cond
  ret i32 %ret.0
}

; Function Attrs: nounwind
define i32 @main(i32 %argc, i8** %argv) #0 {
entry:
  %call = call i32 @loop(i32 4, i32 20, i32 40)
  ret i32 %call
}
```

```
attributes #0 = { nounwind "less-precise-fpmad"="false" "no-frame-pointer-elim"="true"
"no-frame-pointer-elim-non-leaf" "no-infs-fp-math"="false" "no-nans-fp-math"="false"
"stack-protector-buffer-size"="8" "unsafe-fp-math"="false" "use-soft-float"="false" }
!llvm.ident = !{!0}
!0 = metadata !{metadata !"clang version 3.4 (tags/RELEASE_34/final)"}
```

LICM: Benchmark 3

This benchmark differs from the others in that it uses a while loop and contains a branch (and therefore a PHI instruction) within the loop. The 2 instructions inside the branches are invariant, but the branch taken is dependent on the induction variable. We should still be able to hoist the instructions within the branches.

licm-bench3.c

```
int loop (int a, int b, int c)
{
  int ret = 0;
  while (a < (b + c)) {
    int x;
    if (a % 2) {
      x = 5 + b; // should get hoisted
    } else {
      x = 4 + c; // should get hoisted
    }
    ret += x;
    a++;
  }
  return ret;
}

int main(int argc, char const *argv[]) {
  int a = loop(4, 20, 40);
  return a;
}
```

licm-bench3.ll (before)

```
; ModuleID = 'licm-bench3.reg.bc'
target datalayout =
"e-p:32:32:32-i1:8:8-i8:8:8-i16:16:16-i32:32:32-i64:32:64-f32:32:32-f64:32:64-v64:64:64-v
128:128:128-a0:0:64-f80:32:32-n8:16:32-S128"
target triple = "i386-pc-linux-gnu"

; Function Attrs: nounwind
define i32 @loop(i32 %a, i32 %b, i32 %c) #0 {
entry:
  br label %while.cond

while.cond:                                       ; preds = %if.end, %entry
  %ret.0 = phi i32 [ 0, %entry ], [ %add3, %if.end ]
  %a.addr.0 = phi i32 [ %a, %entry ], [ %inc, %if.end ]
  %add = add nsw i32 %b, %c
  %cmp = icmp slt i32 %a.addr.0, %add
  br i1 %cmp, label %while.body, label %while.end
while.body:                                       ; preds = %while.cond
  %rem = srem i32 %a.addr.0, 2
  %tobool = icmp ne i32 %rem, 0
  br i1 %tobool, label %if.then, label %if.else

if.then:                                          ; preds = %while.body
  %add1 = add nsw i32 5, %b
  br label %if.end

if.else:                                          ; preds = %while.body
  %add2 = add nsw i32 4, %c
  br label %if.end

if.end:                                           ; preds = %if.else, %if.then
  %x.0 = phi i32 [ %add1, %if.then ], [ %add2, %if.else ]
  %add3 = add nsw i32 %ret.0, %x.0
  %inc = add nsw i32 %a.addr.0, 1
  br label %while.cond

while.end:                                        ; preds = %while.cond
  ret i32 %ret.0
}

; Function Attrs: nounwind
define i32 @main(i32 %argc, i8** %argv) #0 {
entry:
  %call = call i32 @loop(i32 4, i32 20, i32 40)
  ret i32 %call
}

attributes #0 = { nounwind "less-precise-fpmad"="false" "no-frame-pointer-elim"="true"
"no-frame-pointer-elim-non-leaf" "no-infs-fp-math"="false" "no-nans-fp-math"="false"
"stack-protector-buffer-size"="8" "unsafe-fp-math"="false" "use-soft-float"="false" }

!llvm.ident = !{!0}

!0 = metadata !{metadata !"clang version 3.4 (tags/RELEASE_34/final)"}
```

licm-bench3.ll (after)

```
; ModuleID = 'out.bc'
target datalayout =
"e-p:32:32:32-i1:8:8-i8:8:8-i16:16:16-i32:32:32-i64:32:64-f32:32:32-f64:32:64-v64:64:64-v
128:128:128-a0:0:64-f80:32:32-n8:16:32-S128"
target triple = "i386-pc-linux-gnu"

; Function Attrs: nounwind
define i32 @loop(i32 %a, i32 %b, i32 %c) #0 {
entry:
  %add = add nsw i32 %b, %c
  %add1 = add nsw i32 5, %b
  %add2 = add nsw i32 4, %c
  br label %while.cond

while.cond:                                       ; preds = %if.end, %entry
  %ret.0 = phi i32 [ 0, %entry ], [ %add3, %if.end ]
  %a.addr.0 = phi i32 [ %a, %entry ], [ %inc, %if.end ]
  %cmp = icmp slt i32 %a.addr.0, %add
  br i1 %cmp, label %while.body, label %while.end

while.body:                                       ; preds = %while.cond
  %rem = srem i32 %a.addr.0, 2
  %tobool = icmp ne i32 %rem, 0
  br i1 %tobool, label %if.then, label %if.else

if.then:                                          ; preds = %while.body
  br label %if.end

if.else:                                          ; preds = %while.body
  br label %if.end

if.end:                                           ; preds = %if.else, %if.then
  %x.0 = phi i32 [ %add1, %if.then ], [ %add2, %if.else ]
  %add3 = add nsw i32 %ret.0, %x.0
  %inc = add nsw i32 %a.addr.0, 1
  br label %while.cond

while.end:                                        ; preds = %while.cond
  ret i32 %ret.0
}

; Function Attrs: nounwind
define i32 @main(i32 %argc, i8** %argv) #0 {
entry:
  %call = call i32 @loop(i32 4, i32 20, i32 40)
  ret i32 %call
}

attributes #0 = { nounwind "less-precise-fpmad"="false" "no-frame-pointer-elim"="true"
"no-frame-pointer-elim-non-leaf" "no-infs-fp-math"="false" "no-nans-fp-math"="false"
"stack-protector-buffer-size"="8" "unsafe-fp-math"="false" "use-soft-float"="false" }

!llvm.ident = !{!0}

!0 = metadata !{metadata !"clang version 3.4 (tags/RELEASE_34/final)"}
```

DCE: Benchmark 1

This benchmark has dead instruction about x in it. That is the x is not used anywhere.

dce-bench1.c

```
int deadvar(int a, int b) {
       int c = 2 + b;
       int z = 2 * c;
  int x = z - 2;
       return c + a;
}

int main(int argc, char const *argv[]) {
  int a = deadvar(3, 4);
  return a;
}
```

dce-bench1.ll (before)

```
; ModuleID = 'dce-bench1.reg.bc'
target datalayout =
"e-p:32:32:32-i1:8:8-i8:8:8-i16:16:16-i32:32:32-i64:32:64-f32:32:32-f64:32:64-v64:64:64-v
128:128:128-a0:0:64-f80:32:32-n8:16:32-S128"
target triple = "i386-pc-linux-gnu"

; Function Attrs: nounwind
define i32 @deadvar(i32 %a, i32 %b) #0 {
entry:
  %add = add nsw i32 2, %b
  %mul = mul nsw i32 2, %add
  %sub = sub nsw i32 %mul, 2
  %add1 = add nsw i32 %add, %a
  ret i32 %add1
}

; Function Attrs: nounwind
define i32 @main(i32 %argc, i8** %argv) #0 {
entry:
  %call = call i32 @deadvar(i32 3, i32 4)
  ret i32 %call
}

attributes #0 = { nounwind "less-precise-fpmad"="false" "no-frame-pointer-elim"="true"
"no-frame-pointer-elim-non-leaf" "no-infs-fp-math"="false" "no-nans-fp-math"="false"
"stack-protector-buffer-size"="8" "unsafe-fp-math"="false" "use-soft-float"="false" }
!llvm.ident = !{!0}
!0 = metadata !{metadata !"clang version 3.4 (tags/RELEASE_34/final)"}
```

dce-bench1.ll (after)

```
; ModuleID = 'result1.o'
target datalayout =
"e-p:32:32:32-i1:8:8-i8:8:8-i16:16:16-i32:32:32-i64:32:64-f32:32:32-f64:32:64-v64:64:64-v
128:128:128-a0:0:64-f80:32:32-n8:16:32-S128"
target triple = "i386-pc-linux-gnu"

; Function Attrs: nounwind
define i32 @deadvar(i32 %a, i32 %b) #0 {
entry:
  %add = add nsw i32 2, %b
  %mul = mul nsw i32 2, %add
  %add1 = add nsw i32 %add, %a
  ret i32 %add1
}

; Function Attrs: nounwind
define i32 @main(i32 %argc, i8** %argv) #0 {
entry:
  %call = call i32 @deadvar(i32 3, i32 4)
  ret i32 %call
}

attributes #0 = { nounwind "less-precise-fpmad"="false" "no-frame-pointer-elim"="true"
"no-frame-pointer-elim-non-leaf" "no-infs-fp-math"="false" "no-nans-fp-math"="false"
"stack-protector-buffer-size"="8" "unsafe-fp-math"="false" "use-soft-float"="false" }

!llvm.ident = !{!0}

!0 = metadata !{metadata !"clang version 3.4 (tags/RELEASE_34/final)"}
```

DCE: Benchmark 2

This benchmark consists of a loop with a dead branch inside. That is, the branch is not dependent on the induction variable.

dce-bench2.c

```
int dcebench2(int a, int b) {
       int c = 2 + b;
       int z = 2 * c;
       int x = z - 2;
       for(int i = 0; i < 10; i++) {
              if (b < 0)
                     c = c + 1;
              else
                     c = c + a;
       }
       return c + a;
}

int main(int argc, char const *argv[]) {
       dcebench2(2, 3);
       return 0;
}
```

dce-bench2.ll (before)

```
; ModuleID = 'dce-bench2.reg.bc'
target datalayout =
"e-p:32:32:32-i1:8:8-i8:8:8-i16:16:16-i32:32:32-i64:32:64-f32:32:32-f64:32:64-v64:64:64-v
128:128:128-a0:0:64-f80:32:32-n8:16:32-S128"
target triple = "i386-pc-linux-gnu"

; Function Attrs: nounwind
define i32 @dcebench2(i32 %a, i32 %b) #0 {
entry:
  %add = add nsw i32 2, %b
  %mul = mul nsw i32 2, %add
  %sub = sub nsw i32 %mul, 2
  br label %for.cond

for.cond:                                         ; preds = %for.inc, %entry
  %c.0 = phi i32 [ %add, %entry ], [ %c.1, %for.inc ]
  %i.0 = phi i32 [ 0, %entry ], [ %inc, %for.inc ]
  %cmp = icmp slt i32 %i.0, 10
  br i1 %cmp, label %for.body, label %for.end

for.body:                                         ; preds = %for.cond
  %cmp1 = icmp slt i32 %b, 0
  br i1 %cmp1, label %if.then, label %if.else

if.then:                                          ; preds = %for.body
  %add2 = add nsw i32 %c.0, 1
  br label %if.end

if.else:                                          ; preds = %for.body
  %add3 = add nsw i32 %c.0, %a
  br label %if.end

if.end:                                           ; preds = %if.else, %if.then
  %c.1 = phi i32 [ %add2, %if.then ], [ %add3, %if.else ]
  br label %for.inc

for.inc:                                          ; preds = %if.end
  %inc = add nsw i32 %i.0, 1
  br label %for.cond

for.end:                                          ; preds = %for.cond
  %add4 = add nsw i32 %c.0, %a
  ret i32 %add4
}

; Function Attrs: nounwind
define i32 @main(i32 %argc, i8** %argv) #0 {
entry:
  %call = call i32 @dcebench2(i32 2, i32 3)
  ret i32 0
}

attributes #0 = { nounwind "less-precise-fpmad"="false" "no-frame-pointer-elim"="true"
"no-frame-pointer-elim-non-leaf" "no-infs-fp-math"="false" "no-nans-fp-math"="false"
"stack-protector-buffer-size"="8" "unsafe-fp-math"="false" "use-soft-float"="false" }

!llvm.ident = !{!0}

!0 = metadata !{metadata !"clang version 3.4 (tags/RELEASE_34/final)"}
```

dce-bench2.ll (after)

```
; ModuleID = 'result2.o'
target datalayout =
"e-p:32:32:32-i1:8:8-i8:8:8-i16:16:16-i32:32:32-i64:32:64-f32:32:32-f64:32:64-v64:64:64-v
128:128:128-a0:0:64-f80:32:32-n8:16:32-S128"
target triple = "i386-pc-linux-gnu"

; Function Attrs: nounwind
define i32 @dcebench2(i32 %a, i32 %b) #0 {
entry:
  %add = add nsw i32 2, %b
  %mul = mul nsw i32 2, %add
  br label %for.cond

for.cond:                                         ; preds = %for.inc, %entry
  %c.0 = phi i32 [ %add, %entry ], [ %c.1, %for.inc ]
  %i.0 = phi i32 [ 0, %entry ], [ %inc, %for.inc ]
  %cmp = icmp slt i32 %i.0, 10
  br i1 %cmp, label %for.body, label %for.end

for.body:                                         ; preds = %for.cond
  %cmp1 = icmp slt i32 %b, 0
  br i1 %cmp1, label %if.then, label %if.else

if.then:                                          ; preds = %for.body
  %add2 = add nsw i32 %c.0, 1
  br label %if.end

if.else:                                          ; preds = %for.body
  %add3 = add nsw i32 %c.0, %a
  br label %if.end

if.end:                                           ; preds = %if.else, %if.then
  %c.1 = phi i32 [ %add2, %if.then ], [ %add3, %if.else ]
  br label %for.inc

for.inc:                                          ; preds = %if.end
  %inc = add nsw i32 %i.0, 1
  br label %for.cond

for.end:                                          ; preds = %for.cond
  %add4 = add nsw i32 %c.0, %a
  ret i32 %add4
}

; Function Attrs: nounwind
define i32 @main(i32 %argc, i8** %argv) #0 {
entry:
  %call = call i32 @dcebench2(i32 2, i32 3)
  ret i32 0
}

attributes #0 = { nounwind "less-precise-fpmad"="false" "no-frame-pointer-elim"="true"
"no-frame-pointer-elim-non-leaf" "no-infs-fp-math"="false" "no-nans-fp-math"="false"
"stack-protector-buffer-size"="8" "unsafe-fp-math"="false" "use-soft-float"="false" }

!llvm.ident = !{!0}

!0 = metadata !{metadata !"clang version 3.4 (tags/RELEASE_34/final)"}
```

DCE: Benchmark 3

This benchmark consists of a simple loop with 2 dead assignments within it.

dce-bench3.c

```
int dcebench3(int a, int b) {
        int c = 2 + b;
        int w = 2 * c;

        for(int i = 0; i < 1000; i++) {
                w = b + c;
                c = c + i;
```

```
       }

       return c + a;
}

int main(int argc, char const *argv[]) {
       dcebench3(2, 3);
       return 0;
}
```

dce-bench3.ll (before)

```
; ModuleID = 'dce-bench3.reg.bc'
target datalayout =
"e-p:32:32:32-i1:8:8-i8:8:8-i16:16:16-i32:32:32-i64:32:64-f32:32:32-f64:32:64-v64:64:64-v
128:128:128-a0:0:64-f80:32:32-n8:16:32-S128"
target triple = "i386-pc-linux-gnu"

; Function Attrs: nounwind
define i32 @dcebench3(i32 %a, i32 %b) #0 {
entry:
  %add = add nsw i32 2, %b
  %mul = mul nsw i32 2, %add
  br label %for.cond

for.cond:                                         ; preds = %for.inc, %entry
  %c.0 = phi i32 [ %add, %entry ], [ %add2, %for.inc ]
  %i.0 = phi i32 [ 0, %entry ], [ %inc, %for.inc ]
  %cmp = icmp slt i32 %i.0, 1000
  br i1 %cmp, label %for.body, label %for.end

for.body:                                         ; preds = %for.cond
  %add1 = add nsw i32 %b, %c.0
  %add2 = add nsw i32 %c.0, %i.0
  br label %for.inc

for.inc:                                          ; preds = %for.body
  %inc = add nsw i32 %i.0, 1
  br label %for.cond

for.end:                                          ; preds = %for.cond
  %add3 = add nsw i32 %c.0, %a
  ret i32 %add3
}

; Function Attrs: nounwind
define i32 @main(i32 %argc, i8** %argv) #0 {
entry:
  %call = call i32 @dcebench3(i32 2, i32 3)
  ret i32 0
}

attributes #0 = { nounwind "less-precise-fpmad"="false" "no-frame-pointer-elim"="true"
"no-frame-pointer-elim-non-leaf" "no-infs-fp-math"="false" "no-nans-fp-math"="false"
"stack-protector-buffer-size"="8" "unsafe-fp-math"="false" "use-soft-float"="false" }

!llvm.ident = !{!0}
```

```
!0 = metadata !{metadata !"clang version 3.4 (tags/RELEASE_34/final)"}
```

dce-bench3.ll (after)

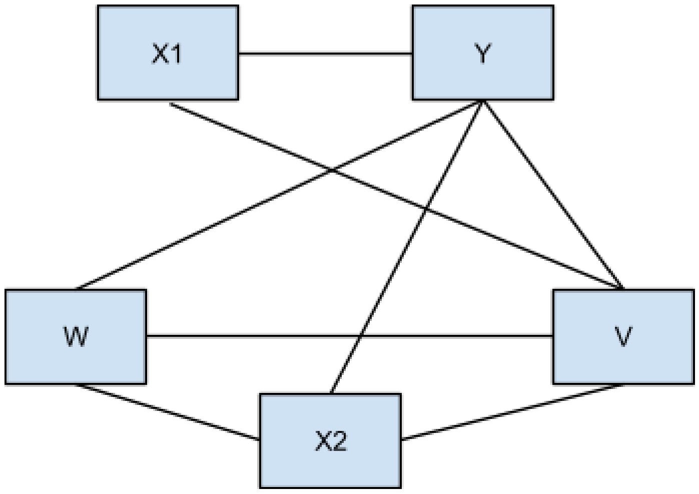
```
; ModuleID = 'result3.o'
target datalayout =
"e-p:32:32:32-i1:8:8-i8:8:8-i16:16:16-i32:32:32-i64:32:64-f32:32:32-f64:32:64-v64:64:64-v
128:128:128-a0:0:64-f80:32:32-n8:16:32-S128"
target triple = "i386-pc-linux-gnu"

; Function Attrs: nounwind
define i32 @dcebench3(i32 %a, i32 %b) #0 {
entry:
  %add = add nsw i32 2, %b
  br label %for.cond

for.cond:                                         ; preds = %for.inc, %entry
  %c.0 = phi i32 [ %add, %entry ], [ %add2, %for.inc ]
  %i.0 = phi i32 [ 0, %entry ], [ %inc, %for.inc ]
  %cmp = icmp slt i32 %i.0, 1000
  br i1 %cmp, label %for.body, label %for.end

for.body:                                         ; preds = %for.cond
  %add2 = add nsw i32 %c.0, %i.0
  br label %for.inc

for.inc:                                          ; preds = %for.body
  %inc = add nsw i32 %i.0, 1
  br label %for.cond

for.end:                                          ; preds = %for.cond
  %add3 = add nsw i32 %c.0, %a
  ret i32 %add3
}

; Function Attrs: nounwind
define i32 @main(i32 %argc, i8** %argv) #0 {
entry:
  %call = call i32 @dcebench3(i32 2, i32 3)
  ret i32 0
}

attributes #0 = { nounwind "less-precise-fpmad"="false" "no-frame-pointer-elim"="true"
"no-frame-pointer-elim-non-leaf" "no-infs-fp-math"="false" "no-nans-fp-math"="false"
"stack-protector-buffer-size"="8" "unsafe-fp-math"="false" "use-soft-float"="false" }

!llvm.ident = !{!0}

!0 = metadata !{metadata !"clang version 3.4 (tags/RELEASE_34/final)"}
```

Theory Questions
Aditya Bhandaru (akbhanda), Zhe Qian (zheq)

3.1 Register Allocation
3.1.1 Build the Interference Graph

3.1.1.1 Find out Nodes(Live Ranges)
According to liveness and reaching definition analysis, X, Y, V and W (Z has no live range since it only has def), X’s live range can be split.

3.1.1.2 Find out edges
Variables that are live simultaneously.

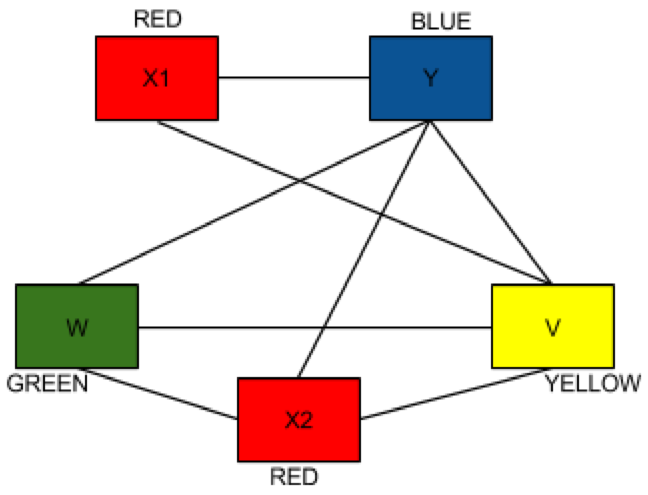


3.1.2 Coloring

3.1.2.1 Pick node with degree < n = 4 and push it to stack iteratively

Push order: X1, Y, W, X2, V

3.1.2.2 Pop node from stack and color them

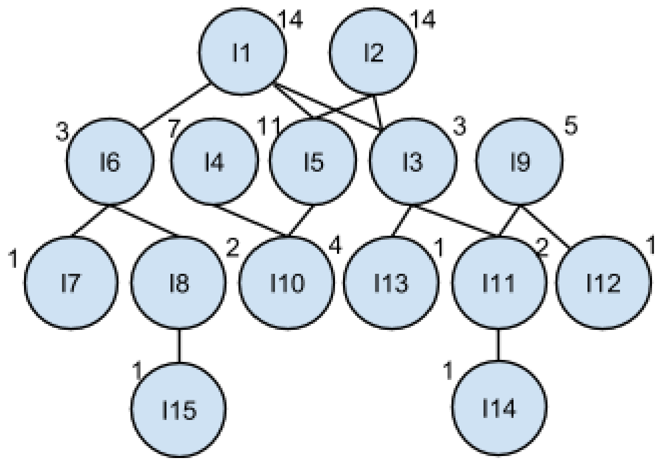


3.2 Instruction Scheduling

3.2.1 Forward List Scheduling

3.2.1.1 Data Precedence Graph(DPG)

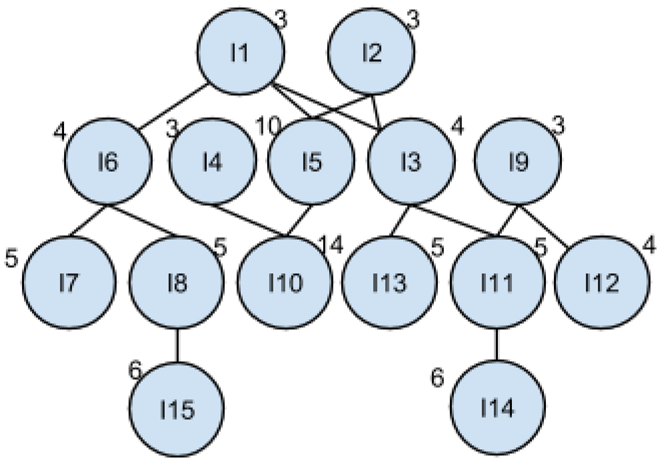
Graph is shown below.



3.2.1.2 Scheduling result

Cycle	AU1	AU2	L/S	Ready List
0			I1	I1, I2, I4, I9
1			I2	I2, I4, I9
2			I4	I4, I9
3	I6		I9	I9, I6
4	I3	I5		I3, I5, I8, I7
5	I8		I7	I8, I7, I13
6	I11		I13	I13, I11, I12, I15
7			I12	I12, I15, I14
8			i15	I15, I14
9			I14	I14
10	---	---	---	---
11	I10			I10

3.2.2 Reverse List Scheduling
3.2.1.1 Data Precedence Graph(DPG)



Since the longest(the most time-consuming) path I1 -> I5 -> I10 can no longer be scheduled better in the Forward List Scheduling, so the performance won’t get better.

Code Listing

Aditya Bhandaru (akbhanda), Zhe Qian (zheq)

dataflow.h

```
// 15-745 S14 Assignment 2: dataflow.h
// Group: akbhanda, zheq
////////////////////////////////////////////////////////////////////////////////

#ifndef __CLASSICAL_DATAFLOW_DATAFLOW_H__
#define __CLASSICAL_DATAFLOW_DATAFLOW_H__

#include <iostream>
#include <queue>
#include <set>
#include <vector>

#include "llvm/IR/Instructions.h"
#include "llvm/ADT/BitVector.h"
#include "llvm/ADT/DenseMap.h"
#include "llvm/ADT/DepthFirstIterator.h"
#include "llvm/ADT/SmallSet.h"
#include "llvm/ADT/ValueMap.h"
#include "llvm/Support/CFG.h"
#include "llvm/Support/raw_ostream.h"
#include "llvm/Pass.h"

#include "util.h"


namespace llvm {


enum Meet {
  INTERSECTION,
  UNION
};

enum Direction {
  FORWARDS,
  BACKWARDS
};


class DataFlowPass : public FunctionPass {
 public:
  DataFlowPass(char ID, Meet meet, Direction direction);
  void initStates(const BlockList& blocks, BlockStates& states);
  void initStates(const BlockVector& blocks, BlockStates& states);
  void traverseForwards(const BasicBlock* start, BlockStates& states);
  void traverseBackwards(const BasicBlock* start, BlockStates& states);
  void meetFn(const Assignments& in, Assignments& out);
  void display(const BlockList& blocks, BlockStates& states);
  BlockStates runOnBlocks(const BlockList& blocks);
  BlockStates runOnBlocks(const BlockVector& blocks);

  // data flow API
  virtual Assignments top(const BasicBlock& block) = 0;
  virtual Assignments init(const BasicBlock& block) = 0;
  virtual Assignments generate(const BasicBlock& block) = 0;
  virtual Assignments kill(const BasicBlock& block) = 0;
  virtual void transferFn(const Assignments& generate, const Assignments& kill,
      const Assignments& input, Assignments& output) = 0;

  // pass API
  virtual bool runOnFunction(Function& F);
  virtual void getAnalysisUsage(AnalysisUsage& AU) const;

 protected:
  const Meet _meet;
  const Direction _direction;

 private:
  void initState(const BasicBlock& block, BlockState& state);
};


}

#endif
```

dataflow.cpp

```
// 15-745 S14 Assignment 2: dataflow.cpp
// Group: akbhanda, zheq
////////////////////////////////////////////////////////////////////////////////

#include "dataflow.h"

using std::cout;
using std::endl;

namespace llvm {

//
// Subclasses must call this constructor during construction.
// This constructor calls the FunctionPass constructor (it extends that class).
// Use initializer listed to assign constants on instantiation.
//
DataFlowPass::DataFlowPass(char ID, Meet meet, Direction direction) :
    FunctionPass(ID),
    _meet(meet),
    _direction(direction) { };


//
// Compute the generate and kill sets for each basic block in the given
// function. The generate and kill functions are overriden by the subclass.
//
void DataFlowPass::initStates(const BlockList& blocks, BlockStates& states) {
  for (BlockList::const_iterator FI = blocks.begin(), FE = blocks.end();
      FI != FE; ++FI) {
    const BasicBlock& block = *FI;
    BlockState state;
    initState(block, state);
    states.insert(BlockStatePair(&block, state));
  }
}

void DataFlowPass::initStates(const BlockVector& blocks, BlockStates& states) {
  for (BlockVector::const_iterator FI = blocks.begin(), FE = blocks.end();
      FI != FE; ++FI) {
    const BasicBlock& block = **FI;
    BlockState state;
    initState(block, state);
    states.insert(BlockStatePair(&block, state));
  }
}

void DataFlowPass::initState(const BasicBlock& block, BlockState& state) {
  // pre-compute generate and kill sets
  state.generates = generate(block);
  state.kills = kill(block);
  // init the input or output based on direction
  if (_direction == FORWARDS) {
    state.out = init(block);
  } else {
    state.in = init(block);
  }
}


//
// Do a forwards traversal on the Control Flow Graph to perform the given
// analysis. The BlockState map is updated during the traversal.
//
void DataFlowPass::traverseForwards(const BasicBlock* start, BlockStates& states) {
  std::queue<const BasicBlock*> worklist;
  std::set<const BasicBlock*> visited;

  // Set up initial conditions.
  BlockState& start_state = states[start];
  start_state.out = top(*start);
  worklist.push(start);

  // Process queue until it is empty.
  while (!worklist.empty()) {
    // inspect 1st element
    const BasicBlock* current = worklist.front();
    worklist.pop();

    // determine the meet of all successors
    BlockState& state = states[current];
    Assignments meet = state.out;
    for (const_pred_iterator I = pred_begin(current), IE = pred_end(current);
        I != IE; ++I) {
      BlockState& pred_state = states[*I];
      meetFn(pred_state.out, meet);
    }

    // see if we need to inspect this node
    if (visited.count(current) && DataFlowUtil::setEquals(state.in, meet)) {
      continue;
    }

    // perform transfer function
    state.in = meet;
    visited.insert(current);
    transferFn(state.generates, state.kills, state.in, state.out);

    // Add all predecessors to the worklist.
    for (succ_const_iterator I = succ_begin(current), IE = succ_end(current);
        I != IE; ++I) {
      worklist.push(*I);
    }
  }
}


//
// Do a backwards traversal on the Control Flow Graph to perform the given
// analysis. The BlockState map is updated during the traversal.
//
void DataFlowPass::traverseBackwards(const BasicBlock* start, BlockStates& states) {
  std::queue<const BasicBlock*> worklist;
  std::set<const BasicBlock*> visited;

  // Set up initial conditions.
  BlockState& start_state = states[start];
  start_state.in = top(*start);
  worklist.push(start);

  // Process queue until it is empty.
  while (!worklist.empty()) {
    // inspect 1st element
    const BasicBlock* current = worklist.front();
    worklist.pop();

    // determine the meet of all successors
    BlockState& state = states[current];
    Assignments meet = state.in;
    for (succ_const_iterator I = succ_begin(current), IE = succ_end(current);
        I != IE; ++I) {
      BlockState& succ_state = states[*I];
      meetFn(succ_state.in, meet);
    }

    // See if we need to inspect this node.
    if (visited.count(current) && DataFlowUtil::setEquals(state.in, meet)) {
      continue;
    }

    // Perform transfer function.
    state.out = meet;
    visited.insert(current);
    transferFn(state.generates, state.kills, state.out, state.in);

    // Add all predecessors to the worklist.
    for (const_pred_iterator I = pred_begin(current), IE = pred_end(current);
        I != IE; ++I) {
      worklist.push(*I);
    }
  }
}


//
// Performs the correct meet operation on two input sets.
// The output is stored in the second set. The first set is unmodified.
//
void DataFlowPass::meetFn(const Assignments& in, Assignments& out) {
  if (_meet == UNION) {
    DataFlowUtil::setUnion(out, in);
  } else if (_meet == INTERSECTION) {
    DataFlowUtil::setIntersect(out, in);
  }
}


//
// Called after the pass is complete.
// Show the results of the pass for each program point b/t blocks.
//
void DataFlowPass::display(const BlockList& blocks, BlockStates& states) {
  for (BlockList::const_iterator I = blocks.begin(), IE = blocks.end();
      I != IE; ++I) {
    const BasicBlock* block = &(*I);
    BlockState& state = states[block];
    if (I == blocks.begin()) {
      DataFlowUtil::print(state.in);
      cout << endl;
    }
    block->dump();
    DataFlowUtil::print(state.out);
    cout << endl;
  }
  cout << endl;
}


//
// Generic helper functions for arbitrary sets of blocks
//
BlockStates DataFlowPass::runOnBlocks(const BlockList& blocks) {
  BlockStates states;
  // First pass: precompute generate and kill sets.
  initStates(blocks, states);
  // iterate for a forwards pass
  if (_direction == FORWARDS) {
    const BasicBlock* start = &(blocks.front());
    traverseForwards(start, states);
  }
  // iterate for a backwards pass
  else if (_direction == BACKWARDS) {
    const BasicBlock* start = &(blocks.back());
    traverseBackwards(start, states);
  }
  // return copy of states
  return states;
}

BlockStates DataFlowPass::runOnBlocks(const BlockVector& blocks) {
  BlockStates states;
  // First pass: precompute generate and kill sets.
  initStates(blocks, states);
  // iterate for a forwards pass
  if (_direction == FORWARDS) {
    const BasicBlock* start = blocks.front();
    traverseForwards(start, states);
  }
  // iterate for a backwards pass
  else if (_direction == BACKWARDS) {
    const BasicBlock* start = blocks.back();
    traverseBackwards(start, states);
  }
  return states;
}


//
// Called by the FunctionPass API in LLVM.
//
bool DataFlowPass::runOnFunction(Function& fn) {
  cout << "Function: " << fn.getName().data() << endl << endl;
  BlockList& blocks = fn.getBasicBlockList();
  BlockStates states = runOnBlocks(blocks);

  // Does not modify the incoming Function.
  display(blocks, states);
  return false;
}


void DataFlowPass::getAnalysisUsage(AnalysisUsage& AU) const {
  AU.setPreservesCFG();
}

}
```

dominance.h

```
// 15-745 S14 Assignment 3: dominance.h
// Group: akbhanda, zheq
////////////////////////////////////////////////////////////////////////////////

#ifndef __DOMINANCE_PASS_H__
#define __DOMINANCE_PASS_H__

#include <set>
#include <queue>
#include <vector>
```

```
#include "llvm/ADT/DenseMap.h"
#include "llvm/IR/Function.h"
#include "llvm/Pass.h"

#include "dataflow.h"
#include "util.h"

namespace llvm {

class DominancePass : public DataFlowPass {
 public:
  // datatypes
  class Node;
  class Node {
   public:
    Node(const Node* _parent, BasicBlock* _data)
        : data(_data), parent(_parent) { };
    ~Node() {
      for (int i = 0; i < children.size(); i++) {
        delete children[i];
      }
    };
    const Node* parent;
    BasicBlock* data;
    std::vector<Node *> children;
  };

  // methods
  static char ID;
  DominancePass();
  Assignments top(const BasicBlock& block);
  Assignments init(const BasicBlock& block);
  Assignments generate(const BasicBlock& block);
  Assignments kill(const BasicBlock& block);
  void transferFn(const Assignments& generate, const Assignments& kill,
    const Assignments& input, Assignments& output);

  // deriving tree
  DominancePass::Node getDominatorTree(const BlockVector& blocks,
    BlockStates& states, BasicBlock* preheader);
  BasicBlock* getIdom(BlockStates& states, BasicBlock* node);
};


}

#endif
```

dominance.cpp

```
// 15-745 S14 Assignment 3: dominance.cpp
// Group: akbhanda, zheq
////////////////////////////////////////////////////////////////////////////////

#include "dominance.h"
```

```
using std::cerr;
using std::cout;
using std::endl;

namespace llvm {


DominancePass::DominancePass() : DataFlowPass(ID, INTERSECTION, FORWARDS) { };


//
// Override the function for generating the top set for the pass.
//
Assignments DominancePass::top(const BasicBlock& block) {
       return Assignments();
}


//
// Override the init function for BlockStates
//
Assignments DominancePass::init(const BasicBlock& block) {
  const Function& fn = *(&block)->getParent();
  Assignments all;
  for (Function::const_iterator I = fn.begin(), IE = fn.end(); I != IE; ++I) {
    const BasicBlock* block = I;
    Assignment assign(block);
    all.insert(assign);
  }
  return all;
}


//
// Override generate function of DataFlowPass to use uses().
//
Assignments DominancePass::generate(const BasicBlock& block) {
  Assignments genSet;
  genSet.insert(Assignment(&block));
  return genSet;
}


//
// Override generate function of DataFlowPass to use defines().
// Notation and naming may change later.
//
Assignments DominancePass::kill(const BasicBlock& block) {
  return Assignments();
}


//
// Subclasses override the transfer function.
// More transparent way to provide function pointers.
//
```

```
void DominancePass::transferFn(const Assignments& generate,
    const Assignments& kill, const Assignments& input, Assignments& output) {
  output = input;
  DataFlowUtil::setUnion(output, generate);
}


//
// This function derives the dominator tree for a given set of blocks and
// the computed block states resulting from this pass.
//
DominancePass::Node DominancePass::getDominatorTree(const BlockVector& blocks,
    BlockStates& states, BasicBlock* preheader) {
  // initial state
  DenseMap<BasicBlock*, Node*> visited;
  std::vector<BasicBlock*> stack;
  Node* root = new Node(NULL, preheader);
  visited[preheader] = root;

  // this loop runs until we have visited all nodes
  for (int i = 0; i < blocks.size(); i++) {
    BasicBlock* current = blocks[i];
    stack.clear();

    // create depth stack
    while (current && !visited.count(current)) {
      stack.push_back(current);
      BasicBlock* idom = getIdom(states, current);
      // edge case when we do not have an idom
      if (idom) current = idom;
      else current = preheader;
    }

    // pop off stack while creating nodes
    while (!stack.empty()) {
      BasicBlock* block = stack.back();
      stack.pop_back();
      // parent is the last node we visited
      Node* parent = visited[current];
      Node* node = new Node(parent, block);
      parent->children.push_back(node);
      // set up next iteration
      visited[block] = node;
      current = block;
    }
  }

  // Return a copy of the root object.
  // When the copy goes out of scope, the Node destructor will run and
  // free up all the memory we allocated.
  return *root;
}


//
// For a given node, find the immediate dominator in the tree using a reverse
// breadth first search. If the current node is not the initial node, and the
// current node is in the initial node's strict dominators list, return the
// current node.
//
BasicBlock* DominancePass::getIdom(BlockStates& states, BasicBlock* node) {
  std::queue<BasicBlock*> worklist;
  std::set<BasicBlock*> visited;
  // Set initial conditions.
  Assignments stops = states[node].out;
  worklist.push(node);
  while (!worklist.empty()) {
    BasicBlock* current = worklist.front();
    worklist.pop();
    // Skip if visited.
    if (visited.count(current)) {
      continue;
    }
    // Did we find the idom?
    if (node != current && stops.count(Assignment(current)) > 0) {
      return current;
    }
    // Mark visited and add all predecessors to the worklist.
    visited.insert(current);
    for (pred_iterator I = pred_begin(current), IE = pred_end(current);
        I != IE; ++I) {
      worklist.push(*I);
    }
  }
  // Return null if there is no idom node (possible for 1st block).
  return NULL;
}


//
// Do the following to meet the FunctionPass API
//
char DominancePass::ID = 0;
RegisterPass<DominancePass> X("dominance", "15745 DominancePass");


}
```

loop-invariant-code-motion.h

```
// 15-745 S14 Assignment 3: loop-invariant-code-motion.h
// Group: akbhanda, zheq
////////////////////////////////////////////////////////////////////////////////

#ifndef _LOOP_INVARIANT_CODE_MOTION_H_
#define _LOOP_INVARIANT_CODE_MOTION_H_

#include <iostream>
#include <set>
#include <vector>

#include "llvm/Analysis/LoopInfo.h"
#include "llvm/Analysis/LoopPass.h"
```

```
#include "llvm/Analysis/ValueTracking.h"
#include "llvm/IR/Constants.h"
#include "llvm/IR/LegacyPassManagers.h"
#include "llvm/Pass.h"

#include "dominance.h"
#include "util.h"

namespace llvm {

typedef std::set<Instruction*> Invariants;

class LicmPass : public LoopPass {
 public:
  static char ID;
  LicmPass();
  void showDominators(const BlockVector& blocks, BlockStates& states,
      BasicBlock* preheader);

  // LoopPass API
  virtual bool runOnLoop(Loop *loop, LPPassManager &LPM);
  virtual void getAnalysisUsage(AnalysisUsage& AU) const;

 private:
  bool isInvariant(const Loop* loop, Invariants& invariants, Instruction* instr);
  bool isInvariant(const Loop* loop, Invariants& invariants, Value* operand);
  bool isTopLevel(const Loop* loop, const BasicBlock* block);
  void inspectBlock(const Loop* loop, BasicBlock* block, Invariants& invariants);
  void findInvariants(
    const Loop* loop,
    const BlockVector& blocks,
    BlockStates& states,
    BasicBlock* preheader,
    Invariants& invariants);

  // pass instances
  DominancePass dominance;

  // data from LLVM passes
  LoopInfo* info;
};

}

#endif
```

loop-invariant-code-motion.cpp

```
// 15-745 S14 Assignment 3: loop-invariant-code-motion.cpp
// Group: akbhanda, zheq
////////////////////////////////////////////////////////////////////////////////

#include "loop-invariant-code-motion.h"

using std::cout;
using std::cerr;
using std::endl;
using std::set;
using std::vector;


namespace llvm {

//
// Call the constructor for the super class.
//
LicmPass::LicmPass() : LoopPass(ID) {
  // do nothing for now
}


//
// Helper function for printing out dominator information.
//
void LicmPass::showDominators(const BlockVector& blocks,
    BlockStates& states, BasicBlock* preheader) {
  for (BlockVector::const_iterator I = blocks.begin(), IE = blocks.end();
      I != IE; ++I) {
    BasicBlock* block = *I;
    BasicBlock* idom = dominance.getIdom(states, block);
    cerr << block->getName().data() << " idom ";
    if (idom) {
      cerr << idom->getName().data();
    } else {
      cerr << preheader->getName().data();
    }
    cerr << endl;
  }
  cerr << endl;
}


//
// Helper function to determine if the given block is in a sub-loop or not.
//
bool LicmPass::isTopLevel(const Loop* loop, const BasicBlock* block) {
  return info->getLoopFor(block) == loop;
}


//
// Check if instruction is invariant.
//
bool LicmPass::isInvariant(const Loop* loop,
   Invariants& invariants, Instruction* instr) {

    // Must also satisfy these conditions to ensure safety of movement.
    if (!isSafeToSpeculativelyExecute(instr) ||
        instr->mayReadFromMemory() ||
        isa<LandingPadInst>(instr)) {
      return false;
    }
    // See if all operands are invariant.
    for (User::op_iterator OI = instr->op_begin(), OE = instr->op_end();
        OI != OE; ++OI) {
      Value *operand = *OI;
      if (!isInvariant(loop, invariants, operand)) {
        return false;
      }
    }

    // Satisfies all conditions.
    return true;
}


//
// iterate through operands and check to see if they are invariant.
// An invariant is any non-instruction or must meet 1 of the following.
// - constant (TODO: try and constant fold here)
// - all reaching definitions for operand are outside the loop
// - both have exactly 1 reaching definition and it is invariant
//
bool LicmPass::isInvariant(const Loop* loop,
    Invariants& invariants, Value* operand) {
  // invariance check for instruction operands
  if (Instruction* instr = dyn_cast<Instruction>(operand)) {
    return invariants.count(instr) || !loop->contains(instr);
  }
  // All non-instructions are invariant.
  return true;
}


//
// Find loop invariant instructions in a given block.
// We can do this block by block because we are traversing the dominator
// tree in pre-order, which accumulating invariants in our set.
//
void LicmPass::inspectBlock(const Loop* loop, BasicBlock* block,
    Invariants& invariants) {
  // No need to iterate over subloops because they will already have their
  // instructions hoisted. Skip if we are in the subloop.
  bool top_level = isTopLevel(loop, block);
  if (!top_level) {
    return;
  }

  // Iterate through all the intructions.
  for (BasicBlock::iterator J = block->begin(), JE = block->end();
      J != JE; ++J) {
    Instruction& instr = *J;
    bool invariant = isInvariant(loop, invariants, &instr);
    if (invariant) {
      invariants.insert(&instr);
    }
  }
}
```

```
//
// For a given loop, find the loop invariant instructions and populate the
// set of invariant instructions found.
//
void LicmPass::findInvariants(
    const Loop* loop,
    const BlockVector& blocks,
    BlockStates& states,
    BasicBlock* preheader,
    Invariants& invariants) {
  // Compute the dominance tree.
  DominancePass::Node dom_tree = dominance.getDominatorTree(
      blocks, states, preheader);
  vector<DominancePass::Node*> stack;
  stack.push_back(&dom_tree);
  // Do a pre-order traversal over the Dominance Tree.
  while (!stack.empty()) {
    DominancePass::Node* node = stack.back();
    stack.pop_back();
    if (node->parent) {
      inspectBlock(loop, node->data, invariants);
    }
    for (int i = 0; i < node->children.size(); i++) {
      stack.push_back(node->children[i]);
    }
  }
}


//
// Override the runOnLoop function provided by LoopPass.
// Return true because we intend on modifying the control flow graph.
//
bool LicmPass::runOnLoop(Loop *loop, LPPassManager &LPM) {
  unsigned int depth = loop->getLoopDepth();
  info = &getAnalysis<LoopInfo>();
  cout << "Loop<depth = " << depth << ">" << endl;

  // check if there is a preheader
  BasicBlock* preheader = loop->getLoopPreheader();
  if (!preheader) {
    return false;
  }

  // Run analysis passes on blocks.
  const BlockVector& blocks = loop->getBlocks();
  BlockStates states = dominance.runOnBlocks(blocks);
  showDominators(blocks, states, preheader);

  // Cache all the invariants we have found so far.
  Invariants invariants;
  findInvariants(loop, blocks, states, preheader, invariants);

  // Finally, hoist the invariants into the preheader.
  // We want to delete this instruction and move it to the end of the
  // preheader (before the branch). We can do with with moveBefore.
  for (Invariants::iterator I = invariants.begin(), IE = invariants.end();
      I != IE; ++I) {
    Instruction* end = &(preheader->back());
    Instruction* instr = *I;
    instr->moveBefore(end);
  }

  // assume we modified the loop
  return true;
};

//
// Set so that we can modify the control flow graph.
// TODO: Is this correct?
//
void LicmPass::getAnalysisUsage(AnalysisUsage& AU) const {
  AU.setPreservesCFG();
  AU.addRequired<LoopInfo>();
}

//
// Do the following to meet the LoopPass API
//
char LicmPass::ID = 0;
RegisterPass<LicmPass> Y("cd-licm", "15745 Liveness");

}
```

dead-code-elimination.h

```
// 15-745 S14 Assignment 3: dead-code-elimination.h
// Group: akbhanda, zheq
////////////////////////////////////////////////////////////////////////////////

#ifndef _DEAD_CODE_ELIMINATION_H_
#define _DEAD_CODE_ELIMINATION_H_

#include <iostream>
// #include <set>
// #include <vector>

#include "llvm/IR/Function.h"
#include "llvm/Analysis/ValueTracking.h"
#include "llvm/IR/IntrinsicInst.h"
#include "llvm/IR/Instructions.h"
#include "llvm/IR/Constants.h"
#include "llvm/Pass.h"
#include "llvm/Transforms/Utils/Local.h"

#include "dataflow.h"
#include "util.h"

namespace llvm {

class DcePass : public DataFlowPass {
 public:
  static char ID;
  DcePass();
  Assignments top(const BasicBlock& block);
  Assignments init(const BasicBlock& block);
  Assignments generate(const BasicBlock& block);
  Assignments kill(const BasicBlock& block);
  void transferFn(const Assignments& generate, const Assignments& kill,
    const Assignments& input, Assignments& output);
  bool runOnFunction(Function& fn);

 private:
  bool isDead(Instruction* instr);
};

}

#endif
```

dead-code-elimination.cpp

```
// 15-745 S14 Assignment 3: dead-code-elimination.cpp
// Group: akbhanda, zheq
////////////////////////////////////////////////////////////////////////////////

#include "dead-code-elimination.h"


using std::cerr;
using std::cout;
using std::endl;
using std::set;


namespace llvm {


DcePass::DcePass() :
    DataFlowPass(ID, UNION, BACKWARDS) { };


//
// Override the function for generating the top set for the pass.
//
Assignments DcePass::top(const BasicBlock& block) {
    return Assignments();
}


//
// Override the init function for BlockStates
//
Assignments DcePass::init(const BasicBlock& block) {
  return Assignments();
}
```

```
//
// Override generate function of DataFlowPass to use defines().
//
Assignments DcePass::generate(const BasicBlock& block) {
  return DataFlowUtil::uses(block);
}


//
// Override generate function of DataFlowPass to use kills().
// Notation and naming may change later.
//
Assignments DcePass::kill(const BasicBlock& block) {
  return DataFlowUtil::defines(block);
}


//
// Subclasses override the transfer function.
// More transparent way to provide function pointers.
//
void DcePass::transferFn(const Assignments& generate,
    const Assignments& kill, const Assignments& input, Assignments& output) {
  output = input;
  DataFlowUtil::setSubtract(output, kill);
  meetFn(generate, output);
}


//
// String of checks for a dead assignment.
//
bool DcePass::isDead(Instruction* instr) {
  if (!instr->use_empty())
    return false;
  if (isa<TerminatorInst>(instr))
    return false;
  if (isa<DbgInfoIntrinsic>(instr))
    return false;
  if (isa<LandingPadInst>(instr))
    return false;
  if (instr->mayHaveSideEffects())
    return false;
  return true;
}


//
// Methods for smaller granularities.
//
bool DcePass::runOnFunction(Function& fn) {
  cerr << "Function: " << fn.getName().data() << endl;
  BlockList& blocks = fn.getBasicBlockList();
  BlockStates states = runOnBlocks(blocks);
  set<Instruction*> dead;
  bool changed = false;
  cerr << "- Instructions to remove:" << endl;
  for (BlockList::iterator I = blocks.begin(), IE = blocks.end();
      I != IE; ++I) {
    BasicBlock* block = &(*I);
    dead.clear();
    // Iterate through instructions and inspect.
    for(BasicBlock::iterator J = block->begin(), JE = block->end();
        J != JE; ++J) {
      Instruction* instr = &(*J);
      if (isDead(instr)) {
        dead.insert(instr);
      }
    }
    // Remove all marked instructions.
    for(set<Instruction*>::iterator I = dead.begin(), IE = dead.end();
        I != IE; ++I) {
      Instruction* instr = *I;
      instr->dump();
      instr->eraseFromParent();
      changed = true;
    }
  }

  // Return whether or not we removed any instructions.
  cerr << endl;
  return changed;
}

//
// Do the following to meet the FunctionPass API
//
char DcePass::ID = 0;
RegisterPass<DcePass> W("cd-dce", "15745 DcePass");


}
```

util.h

```
// 15-745 S14 Assignment 2: util.h
// Group: akbhanda, zheq
////////////////////////////////////////////////////////////////////////////////

#ifndef __UTIL_H__
#define __UTIL_H__

#include <iostream>
#include <set>
#include <utility>
#include <vector>

#include "llvm/ADT/DenseMap.h"
#include "llvm/ADT/SmallSet.h"
#include "llvm/IR/Function.h"
#include "llvm/IR/Instructions.h"
```

```
namespace llvm {

//
// Useful types
//

class Assignment {
 public:
  Assignment(const Value *ptr) : pointer(ptr) { };
  const Value* pointer;
  inline bool operator==(const Assignment& rhs) const {
    return pointer == rhs.pointer;
  }
  inline bool operator<(const Assignment& rhs) const {
    return pointer < rhs.pointer;
  }
};

typedef std::set<Assignment> Assignments;

class BlockState {
 public:
  Assignments in;
  Assignments out;
  Assignments generates;
  Assignments kills;
};

typedef DenseMap<const BasicBlock*, BlockState> BlockStates;
typedef std::pair<const BasicBlock*, BlockState> BlockStatePair;
typedef Function::BasicBlockListType BlockList;
typedef std::vector<BasicBlock*> BlockVector;

//
// Static library functions (for reuse)
//

class DataFlowUtil {
 public:
  static Assignments uses(const BasicBlock& block);
  static Assignments defines(const BasicBlock& block);
  static Assignments kills(const BasicBlock& block);
  static Assignments all(const Function& fn);
  static void setSubtract(Assignments& dest, const Assignments& src);
  static void setUnion(Assignments& dest, const Assignments& src);
  static void setIntersect(Assignments& dest, const Assignments& src);
  static bool setEquals(const Assignments& a, const Assignments& b);
  static void print(const Assignments& assignments);
};

}

#endif
```

util.cpp

```
// 15-745 S14 Assignment 2: util.cpp
// Group: akbhanda, zheq
//
// Overview: This file contains useful typedefs and utility functions for the
//   the DataFlowPass class and its subclasses. The DataFlowUtil class is
//   completely stateless.
////////////////////////////////////////////////////////////////////////////////

#include "util.h"

using std::cerr;
using std::cout;
using std::endl;

namespace llvm {

//
// For a given BasicBlock, compute which variables are used.
//
Assignments DataFlowUtil::uses(const BasicBlock& block) {
  Assignments useSet;
  for (BasicBlock::const_reverse_iterator it = block.rbegin(); it != block.rend(); ++it)
{
    const Instruction& instr = *it;
    const User* user = &instr;
    // iterate through all operands
    User::const_op_iterator OI, OE;
    for(OI = user->op_begin(), OE = user->op_end(); OI != OE; ++OI) {
      Value* val = *OI;
      // check if the operand is used
      if ((isa<Instruction>(val) || isa<Argument>(val)) && (!isa<PHINode>(val))) {
        useSet.insert(Assignment(val));
      }
    }
    useSet.erase(Assignment(&instr));
  }
  return useSet;
}


//
// For a given BasicBlock, compute which variables are defined.
//
Assignments DataFlowUtil::defines(const BasicBlock& block) {
  Assignments defSet;
  for (BasicBlock::const_iterator it = block.begin(); it != block.end(); ++it) {
    // there's no result area for an instr, every instruction is actually a definition
    const Instruction& instr = *it;
    defSet.insert(Assignment(&instr));
  }
  return defSet;
}


Assignments DataFlowUtil::kills(const BasicBlock& block) {
  Assignments killSet;
  const Function& function = *block.getParent();
  for (BasicBlock::const_iterator it = block.begin(); it != block.end(); ++it) {
    const Instruction& inst = *it;

    for(Function::const_iterator itrF = function.begin(); itrF != function.end(); ++itrF)
{
      const BasicBlock& bb = *itrF;
      if (&bb == &block) {
        for(BasicBlock::const_iterator itrB = bb.begin(); itrB != bb.end(); ++itrB) {
          const Instruction& instr = *itrB;
          if (&inst == &instr) {
            killSet.insert(Assignment(&instr));
          }
        }
      }
    }
  }
  return killSet;
}


Assignments DataFlowUtil::all(const Function& fn) {
  Assignments all;

  // Function arguments are values too.
  for (Function::const_arg_iterator I = fn.arg_begin(), IE = fn.arg_end();
      I != IE; ++I) {
    const Argument* arg = &(*I);
    Assignment assign(arg);
    all.insert(assign);
  }

  // Populate with all instructions.
  // TODO: What do we do with loads and stores? Treat the same for now.
  for (Function::const_iterator I = fn.begin(), IE = fn.end(); I != IE; ++I) {
    const BasicBlock& block = *I;
    for (BasicBlock::const_iterator J = block.begin(), JE = block.end();
        J != JE; ++J) {
      const Instruction* instr = &(*J);
      Assignment assign(instr);
      all.insert(assign);
    }
  }
  return all;
}


//
// The following functions perform basic set operations in O(n*log(m)) time,
// where m and n are the sizes of the sets. For our purposes, this is fast
// enough.
//
// The result of these operations is stored back into the 1st argument.
//
void DataFlowUtil::setSubtract(Assignments& dest, const Assignments& src) {
  for (Assignments::const_iterator i = src.begin(); i != src.end(); ++i) {
    const Assignment& sub = *i;
    dest.erase(sub);
  }
}

void DataFlowUtil::setUnion(Assignments& dest, const Assignments& src) {
  for (Assignments::const_iterator i = src.begin(); i != src.end(); ++i) {
    const Assignment& add = *i;
    dest.insert(add);
  }
}

void DataFlowUtil::setIntersect(Assignments& dest, const Assignments& src) {
  Assignments result;
  for (Assignments::const_iterator i = src.begin(); i != src.end(); ++i) {
    const Assignment& test = *i;
    if (src.count(test) > 0 && dest.count(test) > 0) {
      result.insert(test);
    }
  }
  // rewrite the destination
  dest = result;
}

//
// Determine if 2 sets contain the same elements.
//
bool DataFlowUtil::setEquals(const Assignments& a, const Assignments& b) {
  // make sure sets are the same length
  if (a.size() != b.size()) {
    return false;
  }

  // ensure they contain the same elements
  for (Assignments::const_iterator i = a.begin(); i != a.end(); ++i) {
    const Assignment& test = *i;
    if (b.count(test) < 1) {
      return false;
    }
  }
  return true;
}

//
// Helper/debug function to quickly print out a set of assignments.
// Later if we change how we implement assignment sets, we will have to update
// this function.
//
void DataFlowUtil::print(const Assignments& assignments) {
  cerr << "{ ";
  for (Assignments::const_iterator I = assignments.begin(),
      IE = assignments.end(); I != IE; ++I) {
    cerr << (*I).pointer->getName().data() << " ";
  }
  cerr << "}";
}

}
```

Makefile

```
all: dominance.so loop-invariant-code-motion.so dead-code-elimination.so

CXXFLAGS = -rdynamic $(shell llvm-config --cxxflags) -g -O0

dataflow.o: dataflow.cpp dataflow.h
dominance.o: dominance.cpp dominance.h
util.o: util.cpp util.h

%.so: %.o dataflow.o util.o dominance.o
	$(CXX) -dylib -flat_namespace -shared $^ -o $@

clean:
	rm -f *.o *~ *.so

.PHONY: clean all
```