How LLVM Optimizes a Function

An optimizing, ahead-of-time compiler is usually structured as:

1. A frontend that converts source code into an intermediate representation (IR).
2. A target-independent optimization pipeline: a sequence of passes that successively rewrite the IR to eliminate inefficiencies and forms that cannot be readily translated into machine code. Sometimes called the "middle end."
3. A target-dependent backend that generates assembly code or machine code.

In some compilers the IR format remains fixed throughout the optimization pipeline, in others the format or semantics change. In LLVM the format and semantics are fixed, and consequently it should be possible to run any sequences of passes you want without introducing miscompilations or crashing the compiler.

The sequence of passes in the optimization pipeline is engineered by compiler developers; its goal is to do a pretty good job in a reasonable amount of time. It gets tweaked from time to time, and of course there's a different set of passes that gets run at each optimization level. A longish-standing topic in compiler research is to use machine learning or something to come up with a better optimization pipeline either in general or else for a specific application domain that is not well-served by the default pipelines.

Some principles for pass design are minimality and orthogonality: each pass should do one thing well, and there should not be much overlap in functionality. In practice, compromises are sometimes made. For example when two passes tend to repeatedly generate work for each other, they may be integrated into a single, larger pass. Also, some IR-level functionality such as constant folding is so ubiquitously useful that it might not make sense as a pass; LLVM, for example, implicitly folds away constant operations as instructions are created.

In this post we'll look at how some of LLVM's optimization passes work. I'll assume you've read this piece about how Clang compiles a function or alternatively that you more or less understand how LLVM IR works. It's particularly useful to understand the SSA (static single assignment) form: Wikipedia will get you started and this book contains more information than you're likely to want to know. Also see the LLVM Language Reference and the list of optimization passes.

We're looking at how Clang/LLVM 6.0.1 optimizes this C++:

```
bool is_sorted(int *a, int n) {
  for (int i = 0; i < n - 1; i++)
    if (a[i] > a[i + 1])
      return false;
  return true;
}
```

Keep in mind that the optimization pipeline is a busy place and we're going to miss out on a lot of fun stuff such as:

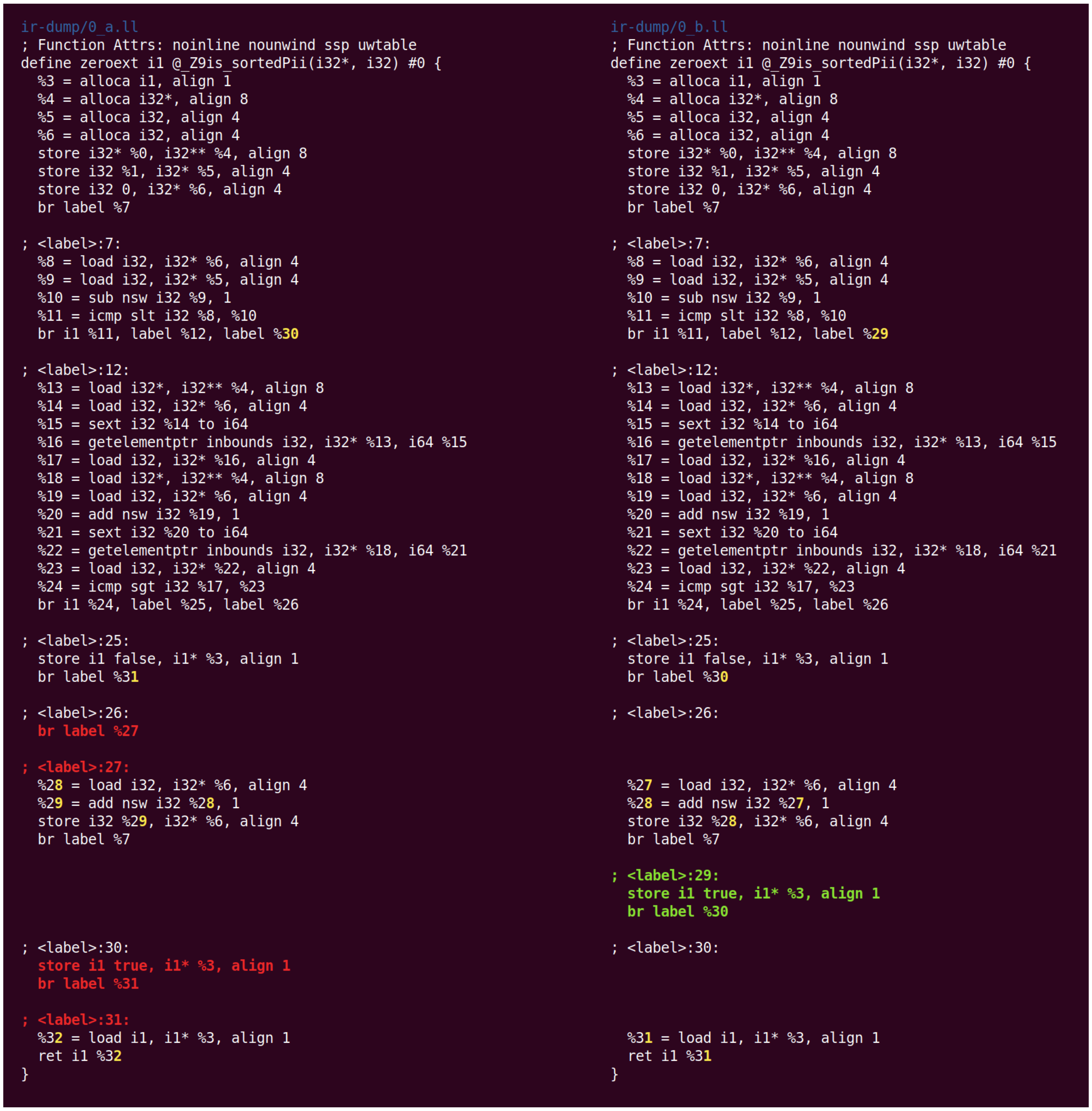
- inlining, an easy but super-important optimization, which can't happen here since we're looking at just one function

- basically everything that is specific to C++ vs C
- autovectorization, which is defeated by the early loop exit

In the text below I'm going to skip over every pass that doesn't make any changes to the code. Also, we're not even looking at the backend and there's a lot going on there as well. Even so, this is going to be a bit of a slog! (Sorry to use images below, but it seemed like the best way to avoid formatting difficulties, I hate fighting with WP themes. Click on the image to get a larger version. I used icdiff.)

Here's the IR file emitted by Clang (I manually removed the "optnone" attribute that Clang put in) and this is the command line that we can use to see the effect of each optimization pass:
`opt -O2 -print-before-all -print-after-all is_sorted2.ll`

The first pass is "simplify the CFG" (control flow graph). Because Clang does not optimize, IR that it emits often contains easy opportunities for cleanup:



```
ir-dump/0_a.ll
; Function Attrs: noinline nounwind ssp uwtable
define zeroext i1 @_Z9is_sortedPii(i32*, i32) #0 {
  %3 = alloca i1, align 1
  %4 = alloca i32*, align 8
  %5 = alloca i32, align 4
  %6 = alloca i32, align 4
  store i32* %0, i32** %4, align 8
  store i32 %1, i32* %5, align 4
  store i32 0, i32* %6, align 4
  br label %7

; <label>:7:
  %8 = load i32, i32* %6, align 4
  %9 = load i32, i32* %5, align 4
  %10 = sub nsw i32 %9, 1
  %11 = icmp slt i32 %8, %10
  br i1 %11, label %12, label %30

; <label>:12:
  %13 = load i32*, i32** %4, align 8
  %14 = load i32, i32* %6, align 4
  %15 = sext i32 %14 to i64
  %16 = getelementptr inbounds i32, i32* %13, i64 %15
  %17 = load i32, i32* %16, align 4
  %18 = load i32*, i32** %4, align 8
  %19 = load i32, i32* %6, align 4
  %20 = add nsw i32 %19, 1
  %21 = sext i32 %20 to i64
  %22 = getelementptr inbounds i32, i32* %18, i64 %21
  %23 = load i32, i32* %22, align 4
  %24 = icmp sgt i32 %17, %23
  br i1 %24, label %25, label %26

; <label>:25:
  store i1 false, i1* %3, align 1
  br label %31

; <label>:26:
  br label %27

; <label>:27:
  %28 = load i32, i32* %6, align 4
  %29 = add nsw i32 %28, 1
  store i32 %29, i32* %6, align 4
  br label %7




; <label>:30:
  store i1 true, i1* %3, align 1
  br label %31

; <label>:31:
  %32 = load i1, i1* %3, align 1
  ret i1 %32
}
```

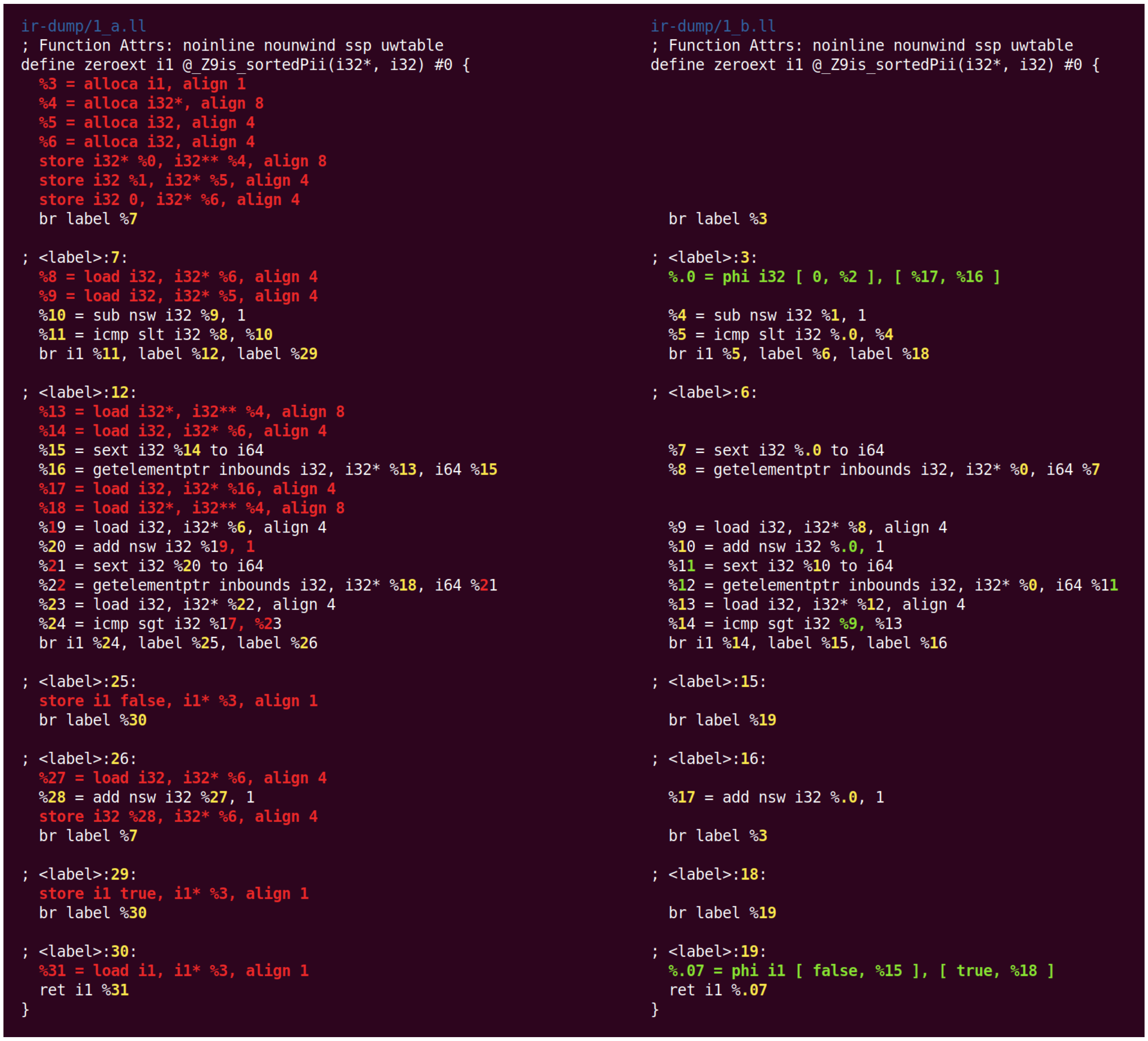
```
ir-dump/0_b.ll
; Function Attrs: noinline nounwind ssp uwtable
define zeroext i1 @_Z9is_sortedPii(i32*, i32) #0 {
  %3 = alloca i1, align 1
  %4 = alloca i32*, align 8
  %5 = alloca i32, align 4
  %6 = alloca i32, align 4
  store i32* %0, i32** %4, align 8
  store i32 %1, i32* %5, align 4
  store i32 0, i32* %6, align 4
  br label %7

; <label>:7:
  %8 = load i32, i32* %6, align 4
  %9 = load i32, i32* %5, align 4
  %10 = sub nsw i32 %9, 1
  %11 = icmp slt i32 %8, %10
  br i1 %11, label %12, label %29

; <label>:12:
  %13 = load i32*, i32** %4, align 8
  %14 = load i32, i32* %6, align 4
  %15 = sext i32 %14 to i64
  %16 = getelementptr inbounds i32, i32* %13, i64 %15
  %17 = load i32, i32* %16, align 4
  %18 = load i32*, i32** %4, align 8
  %19 = load i32, i32* %6, align 4
  %20 = add nsw i32 %19, 1
  %21 = sext i32 %20 to i64
  %22 = getelementptr inbounds i32, i32* %18, i64 %21
  %23 = load i32, i32* %22, align 4
  %24 = icmp sgt i32 %17, %23
  br i1 %24, label %25, label %26

; <label>:25:
  store i1 false, i1* %3, align 1
  br label %30

; <label>:26:



  %27 = load i32, i32* %6, align 4
  %28 = add nsw i32 %27, 1
  store i32 %28, i32* %6, align 4
  br label %7

; <label>:29:
  store i1 true, i1* %3, align 1
  br label %30

; <label>:30:




  %31 = load i1, i1* %3, align 1
  ret i1 %31
}
```

Here, basic block 26 simply jumps to block 27. This kind of block can be eliminated, with jumps to it being forwarded to the destination block. The diff is a bit more confusing that it would have to be due to the implicit block renumbering performed by LLVM. The full set of transformations performed by SimplifyCFG is listed in a comment at the top of the pass:
This file implements dead code elimination and basic block merging, along with a collection of other peephole control flow optimizations. For example:

- Removes basic blocks with no predecessors.
- Merges a basic block into its predecessor if there is only one and the predecessor only has one successor.
- Eliminates PHI nodes for basic blocks with a single predecessor.
- Eliminates a basic block that only contains an unconditional branch.
- Changes invoke instructions to nounwind functions to be calls.
- Change things like "if (x) if (y)" into "if (x&y)".

Most opportunities for CFG cleanup are a result of other LLVM passes. For example, dead code elimination and loop invariant code motion can easily end up creating empty basic blocks.

The next pass to run, SROA (scalar replacement of aggregates), is one of our heavy hitters. The name ends up being a bit misleading since SROA is only one of its functions. This pass examines each alloca instruction (function-scoped memory allocation) and attempts to promote it into SSA registers. A single alloca will turn into multiple registers when it is statically assigned multiple times and also when the alloca is a class or struct that can be split into its components (this splitting is the "scalar replacement" referred to in the name of the pass). A simple version of SROA would give up on stack variables whose addresses are taken, but LLVM's version interacts with alias analysis and ends up being fairly smart (though the smarts are not needed in our example).

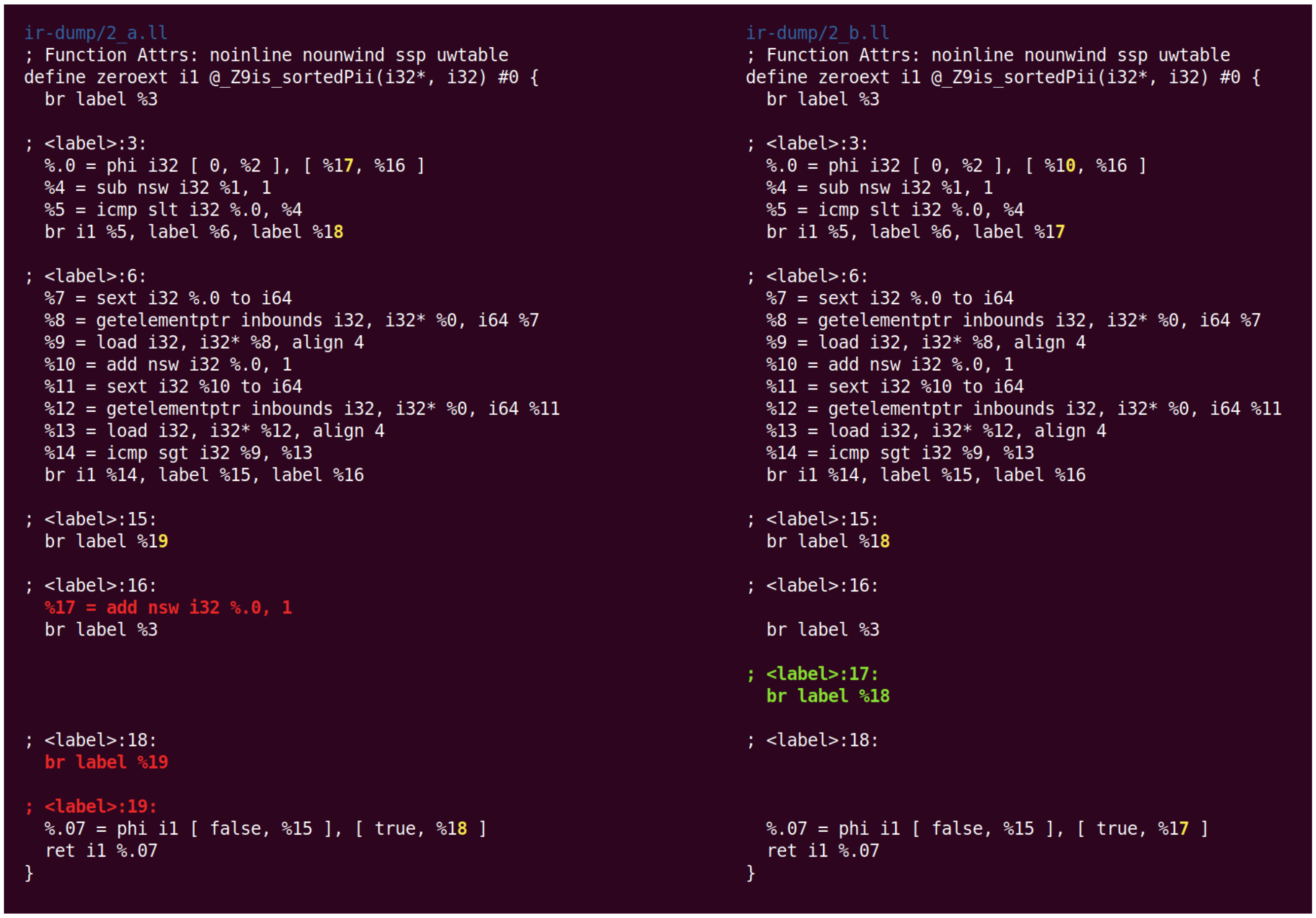


```
ir-dump/1_a.ll
; Function Attrs: noinline nounwind ssp uwtable
define zeroext i1 @_Z9is_sortedPii(i32*, i32) #0 {
  %3 = alloca i1, align 1
  %4 = alloca i32*, align 8
  %5 = alloca i32, align 4
  %6 = alloca i32, align 4
  store i32* %0, i32** %4, align 8
  store i32 %1, i32* %5, align 4
  store i32 0, i32* %6, align 4
  br label %7

; <label>:7:
  %8 = load i32, i32* %6, align 4
  %9 = load i32, i32* %5, align 4
  %10 = sub nsw i32 %9, 1
  %11 = icmp slt i32 %8, %10
  br i1 %11, label %12, label %29

; <label>:12:
  %13 = load i32*, i32** %4, align 8
  %14 = load i32, i32* %6, align 4
  %15 = sext i32 %14 to i64
  %16 = getelementptr inbounds i32, i32* %13, i64 %15
  %17 = load i32, i32* %16, align 4
  %18 = load i32*, i32** %4, align 8
  %19 = load i32, i32* %6, align 4
  %20 = add nsw i32 %19, 1
  %21 = sext i32 %20 to i64
  %22 = getelementptr inbounds i32, i32* %18, i64 %21
  %23 = load i32, i32* %22, align 4
  %24 = icmp sgt i32 %17, %23
  br i1 %24, label %25, label %26

; <label>:25:
  store i1 false, i1* %3, align 1
  br label %30

; <label>:26:
  %27 = load i32, i32* %6, align 4
  %28 = add nsw i32 %27, 1
  store i32 %28, i32* %6, align 4
  br label %7

; <label>:29:
  store i1 true, i1* %3, align 1
  br label %30

; <label>:30:
  %31 = load i1, i1* %3, align 1
  ret i1 %31
}
```

```
ir-dump/1_b.ll
; Function Attrs: noinline nounwind ssp uwtable
define zeroext i1 @_Z9is_sortedPii(i32*, i32) #0 {







  br label %3

; <label>:3:
  %.0 = phi i32 [ 0, %2 ], [ %17, %16 ]

  %4 = sub nsw i32 %1, 1
  %5 = icmp slt i32 %.0, %4
  br i1 %5, label %6, label %18

; <label>:6:


  %7 = sext i32 %.0 to i64
  %8 = getelementptr inbounds i32, i32* %0, i64 %7


  %9 = load i32, i32* %8, align 4
  %10 = add nsw i32 %.0, 1
  %11 = sext i32 %10 to i64
  %12 = getelementptr inbounds i32, i32* %0, i64 %11
  %13 = load i32, i32* %12, align 4
  %14 = icmp sgt i32 %9, %13
  br i1 %14, label %15, label %16

; <label>:15:

  br label %19

; <label>:16:

  %17 = add nsw i32 %.0, 1

  br label %3

; <label>:18:

  br label %19

; <label>:19:
  %.07 = phi i1 [ false, %15 ], [ true, %18 ]
  ret i1 %.07
}
```

After SROA, all alloca instructions (and their corresponding loads and stores) are gone, and the code ends up being much cleaner and more amenable to subsequent optimization (of course, SROA cannot eliminate all allocas in general — this only works when the pointer analysis can completely eliminate aliasing ambiguity). As part of this process, SROA had to insert some phi instructions. Phis are at the core of the SSA representation, and the lack of phis in the code emitted by Clang tells us that Clang emits a trivial kind of SSA where communication between basic blocks is through memory, instead of being through SSA registers.

Next is "early common subexpression elimination" (CSE). CSE tries to eliminate the kind of redundant subcomputations that appear both in code written by people and in partially-optimized code. "Early CSE" is a fast, simple kind of CSE that looks for trivially redundant computations.



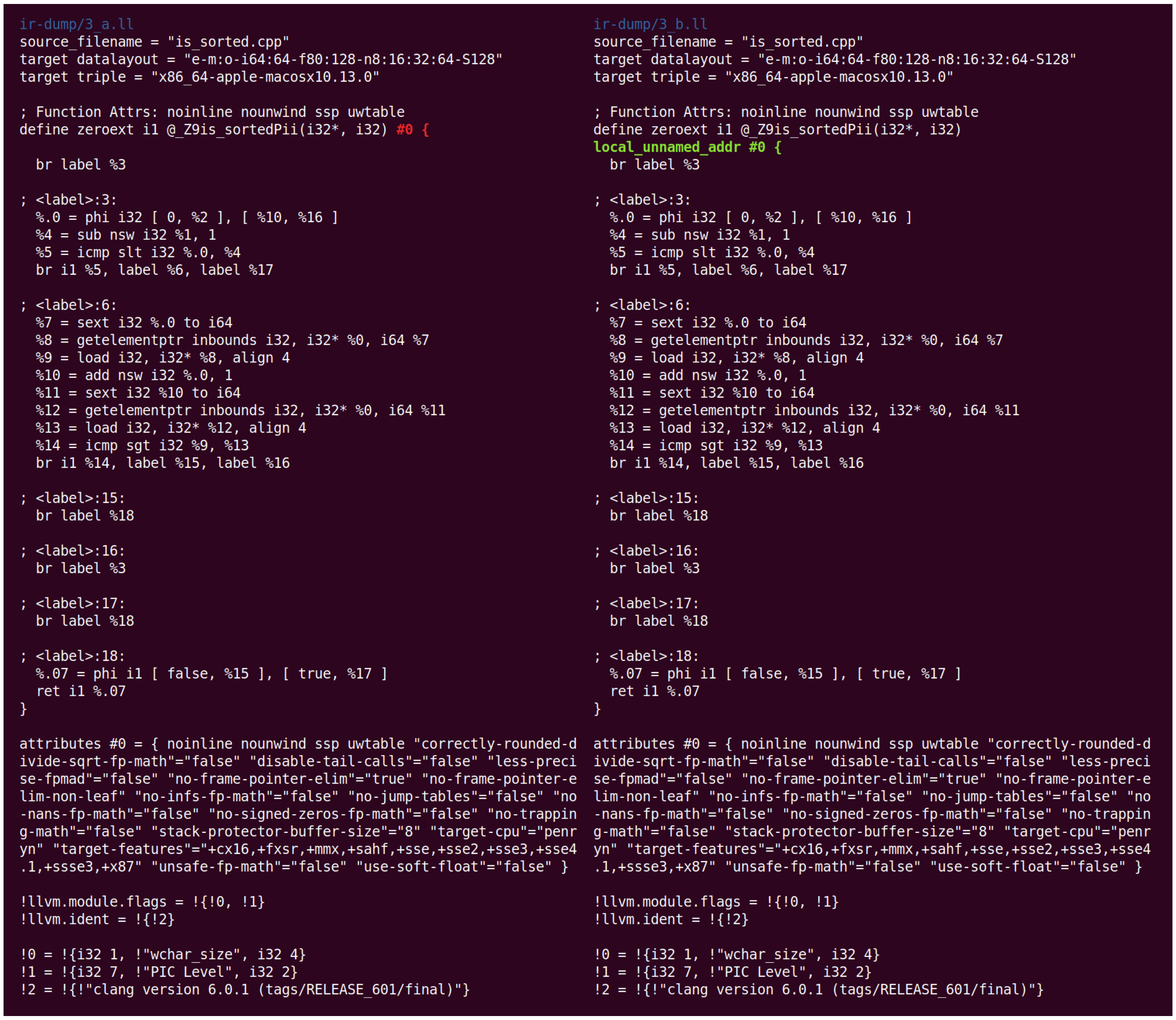
```
ir-dump/2_a.ll
; Function Attrs: noinline nounwind ssp uwtable
define zeroext i1 @_Z9is_sortedPii(i32*, i32) #0 {
  br label %3

; <label>:3:
  %.0 = phi i32 [ 0, %2 ], [ %17, %16 ]
  %4 = sub nsw i32 %1, 1
  %5 = icmp slt i32 %.0, %4
  br i1 %5, label %6, label %18

; <label>:6:
  %7 = sext i32 %.0 to i64
  %8 = getelementptr inbounds i32, i32* %0, i64 %7
  %9 = load i32, i32* %8, align 4
  %10 = add nsw i32 %.0, 1
  %11 = sext i32 %10 to i64
  %12 = getelementptr inbounds i32, i32* %0, i64 %11
  %13 = load i32, i32* %12, align 4
  %14 = icmp sgt i32 %9, %13
  br i1 %14, label %15, label %16

; <label>:15:
  br label %19

; <label>:16:
  %17 = add nsw i32 %.0, 1
  br label %3



; <label>:18:
  br label %19

; <label>:19:
  %.07 = phi i1 [ false, %15 ], [ true, %18 ]
  ret i1 %.07
}
```

```
ir-dump/2_b.ll
; Function Attrs: noinline nounwind ssp uwtable
define zeroext i1 @_Z9is_sortedPii(i32*, i32) #0 {
  br label %3

; <label>:3:
  %.0 = phi i32 [ 0, %2 ], [ %10, %16 ]
  %4 = sub nsw i32 %1, 1
  %5 = icmp slt i32 %.0, %4
  br i1 %5, label %6, label %17

; <label>:6:
  %7 = sext i32 %.0 to i64
  %8 = getelementptr inbounds i32, i32* %0, i64 %7
  %9 = load i32, i32* %8, align 4
  %10 = add nsw i32 %.0, 1
  %11 = sext i32 %10 to i64
  %12 = getelementptr inbounds i32, i32* %0, i64 %11
  %13 = load i32, i32* %12, align 4
  %14 = icmp sgt i32 %9, %13
  br i1 %14, label %15, label %16

; <label>:15:
  br label %18

; <label>:16:

  br label %3

; <label>:17:
  br label %18

; <label>:18:



  %.07 = phi i1 [ false, %15 ], [ true, %17 ]
  ret i1 %.07
}
```

Here both %10 and %17 do the same thing, so uses of one of the values can be rewritten as uses of the other, and then the redundant instruction eliminated. This gives a glimpse of the advantages of SSA: since each register is assigned only once, there's no such thing as multiple versions of a register. Thus, redundant computations can be detected using syntactic equivalence, with no reliance on a deeper program analysis (the same is not true for memory locations, which live outside of the SSA universe).

Next, a few passes that have no effect run, and then "global variable optimizer" which self-describes as:
This pass transforms simple global variables that never have their address taken. If obviously true, it marks read/write globals as constant, deletes variables only stored to, etc.

It makes this change:



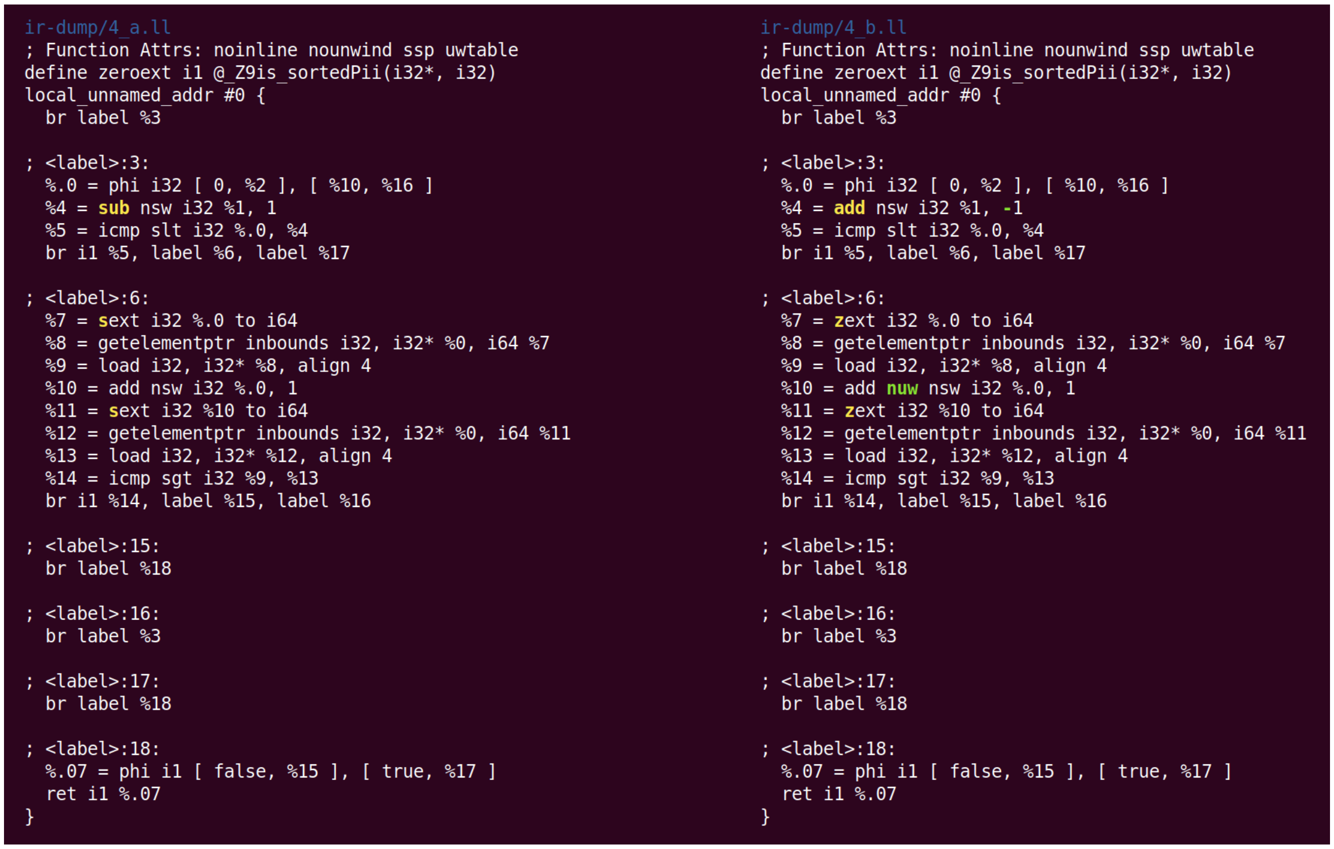
```
ir-dump/3_a.ll
source_filename = "is_sorted.cpp"
target datalayout = "e-m:o-i64:64-f80:128-n8:16:32:64-S128"
target triple = "x86_64-apple-macosx10.13.0"

; Function Attrs: noinline nounwind ssp uwtable
define zeroext i1 @_Z9is_sortedPii(i32*, i32) #0 {

  br label %3

; <label>:3:
  %.0 = phi i32 [ 0, %2 ], [ %10, %16 ]
  %4 = sub nsw i32 %1, 1
  %5 = icmp slt i32 %.0, %4
  br i1 %5, label %6, label %17

; <label>:6:
  %7 = sext i32 %.0 to i64
  %8 = getelementptr inbounds i32, i32* %0, i64 %7
  %9 = load i32, i32* %8, align 4
  %10 = add nsw i32 %.0, 1
  %11 = sext i32 %10 to i64
  %12 = getelementptr inbounds i32, i32* %0, i64 %11
  %13 = load i32, i32* %12, align 4
  %14 = icmp sgt i32 %9, %13
  br i1 %14, label %15, label %16

; <label>:15:
  br label %18

; <label>:16:
  br label %3

; <label>:17:
  br label %18

; <label>:18:
  %.07 = phi i1 [ false, %15 ], [ true, %17 ]
  ret i1 %.07
}

attributes #0 = { noinline nounwind ssp uwtable "correctly-rounded-d
ivide-sqrt-fp-math"="false" "disable-tail-calls"="false" "less-preci
se-fpmad"="false" "no-frame-pointer-elim"="true" "no-frame-pointer-e
lim-non-leaf" "no-infs-fp-math"="false" "no-jump-tables"="false" "no
-nans-fp-math"="false" "no-signed-zeros-fp-math"="false" "no-trappin
g-math"="false" "stack-protector-buffer-size"="8" "target-cpu"="penr
yn" "target-features"="+cx16,+fxsr,+mmx,+sahf,+sse,+sse2,+sse3,+sse4
.1,+ssse3,+x87" "unsafe-fp-math"="false" "use-soft-float"="false" }

!llvm.module.flags = !{!0, !1}
!llvm.ident = !{!2}

!0 = !{i32 1, !"wchar_size", i32 4}
!1 = !{i32 7, !"PIC Level", i32 2}
!2 = !{!"clang version 6.0.1 (tags/RELEASE_601/final)"}
```

```
ir-dump/3_b.ll
source_filename = "is_sorted.cpp"
target datalayout = "e-m:o-i64:64-f80:128-n8:16:32:64-S128"
target triple = "x86_64-apple-macosx10.13.0"

; Function Attrs: noinline nounwind ssp uwtable
define zeroext i1 @_Z9is_sortedPii(i32*, i32)
local_unnamed_addr #0 {
  br label %3

; <label>:3:
  %.0 = phi i32 [ 0, %2 ], [ %10, %16 ]
  %4 = sub nsw i32 %1, 1
  %5 = icmp slt i32 %.0, %4
  br i1 %5, label %6, label %17

; <label>:6:
  %7 = sext i32 %.0 to i64
  %8 = getelementptr inbounds i32, i32* %0, i64 %7
  %9 = load i32, i32* %8, align 4
  %10 = add nsw i32 %.0, 1
  %11 = sext i32 %10 to i64
  %12 = getelementptr inbounds i32, i32* %0, i64 %11
  %13 = load i32, i32* %12, align 4
  %14 = icmp sgt i32 %9, %13
  br i1 %14, label %15, label %16

; <label>:15:
  br label %18

; <label>:16:
  br label %3

; <label>:17:
  br label %18

; <label>:18:
  %.07 = phi i1 [ false, %15 ], [ true, %17 ]
  ret i1 %.07
}

attributes #0 = { noinline nounwind ssp uwtable "correctly-rounded-d
ivide-sqrt-fp-math"="false" "disable-tail-calls"="false" "less-preci
se-fpmad"="false" "no-frame-pointer-elim"="true" "no-frame-pointer-e
lim-non-leaf" "no-infs-fp-math"="false" "no-jump-tables"="false" "no
-nans-fp-math"="false" "no-signed-zeros-fp-math"="false" "no-trappin
g-math"="false" "stack-protector-buffer-size"="8" "target-cpu"="penr
yn" "target-features"="+cx16,+fxsr,+mmx,+sahf,+sse,+sse2,+sse3,+sse4
.1,+ssse3,+x87" "unsafe-fp-math"="false" "use-soft-float"="false" }

!llvm.module.flags = !{!0, !1}
!llvm.ident = !{!2}

!0 = !{i32 1, !"wchar_size", i32 4}
!1 = !{i32 7, !"PIC Level", i32 2}
!2 = !{!"clang version 6.0.1 (tags/RELEASE_601/final)"}
```

The thing that has been added is a function attribute: metadata used by one part of the compiler to store a fact that might be useful to a different part of the compiler. You can read about the rationale for this attribute here.

Unlike other optimizations we've looked at, the global variable optimizer is *interprocedural*, it looks at an entire LLVM module. A module is more or less equivalent to a compilation unit in C or C++. In contrast, *intraprocedural* optimizations look at only one function at a time.

The next pass is the "instruction combiner": InstCombine. It is a large, diverse collection of "peephole optimizations" that (typically) rewrite a handful of instructions connected by data flow into a more efficient form. InstCombine won't change the control flow of a function. In this example it doesn't have a lot to do:

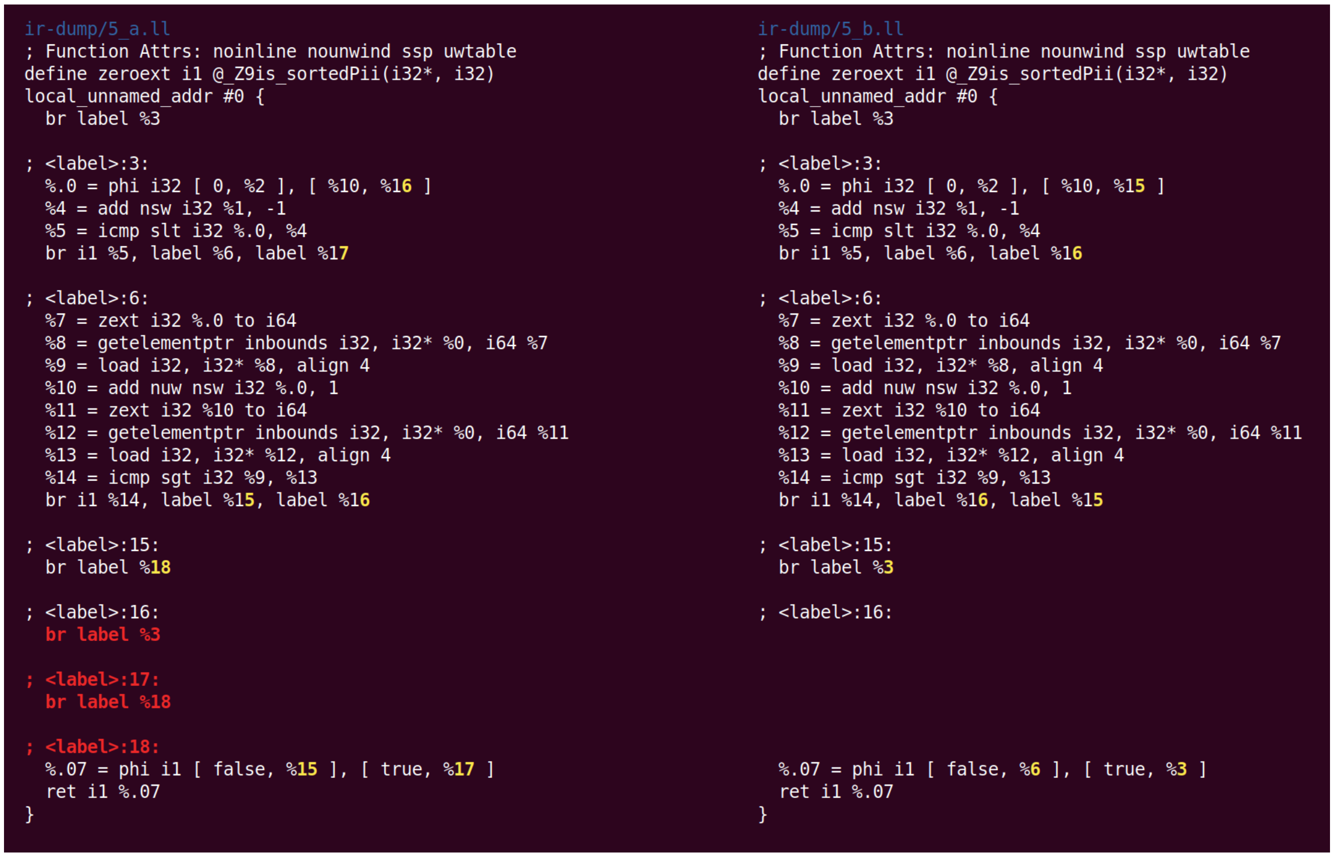


```
ir-dump/4_a.ll
; Function Attrs: noinline nounwind ssp uwtable
define zeroext i1 @_Z9is_sortedPii(i32*, i32)
local_unnamed_addr #0 {
  br label %3

; <label>:3:
  %.0 = phi i32 [ 0, %2 ], [ %10, %16 ]
  %4 = sub nsw i32 %1, 1
  %5 = icmp slt i32 %.0, %4
  br i1 %5, label %6, label %17

; <label>:6:
  %7 = sext i32 %.0 to i64
  %8 = getelementptr inbounds i32, i32* %0, i64 %7
  %9 = load i32, i32* %8, align 4
  %10 = add nsw i32 %.0, 1
  %11 = sext i32 %10 to i64
  %12 = getelementptr inbounds i32, i32* %0, i64 %11
  %13 = load i32, i32* %12, align 4
  %14 = icmp sgt i32 %9, %13
  br i1 %14, label %15, label %16

; <label>:15:
  br label %18

; <label>:16:
  br label %3

; <label>:17:
  br label %18

; <label>:18:
  %.07 = phi i1 [ false, %15 ], [ true, %17 ]
  ret i1 %.07
}
```

```
ir-dump/4_b.ll
; Function Attrs: noinline nounwind ssp uwtable
define zeroext i1 @_Z9is_sortedPii(i32*, i32)
local_unnamed_addr #0 {
  br label %3

; <label>:3:
  %.0 = phi i32 [ 0, %2 ], [ %10, %16 ]
  %4 = add nsw i32 %1, -1
  %5 = icmp slt i32 %.0, %4
  br i1 %5, label %6, label %17

; <label>:6:
  %7 = zext i32 %.0 to i64
  %8 = getelementptr inbounds i32, i32* %0, i64 %7
  %9 = load i32, i32* %8, align 4
  %10 = add nuw nsw i32 %.0, 1
  %11 = zext i32 %10 to i64
  %12 = getelementptr inbounds i32, i32* %0, i64 %11
  %13 = load i32, i32* %12, align 4
  %14 = icmp sgt i32 %9, %13
  br i1 %14, label %15, label %16

; <label>:15:
  br label %18

; <label>:16:
  br label %3

; <label>:17:
  br label %18

; <label>:18:
  %.07 = phi i1 [ false, %15 ], [ true, %17 ]
  ret i1 %.07
}
```

Here instead of subtracting 1 from %1 to compute %4, we've decided to add -1. This is a canonicalization rather than an optimization. When there are multiple ways of expressing a computation, LLVM attempts to canonicalize to a (often arbitrarily chosen) form, which is then what LLVM passes and backends can expect to see. The second change made by InstCombine is canonicalizing two sign-extend operations (sext instructions) that compute %7 and %11 to zero-extends (zext). This is a safe transformation when the compiler can prove that the operand of the sext is non-negative. That is the case here because the loop induction variable starts at zero and stops before it reaches n (if n is negative, the loop never executes at all). The final change is adding the "nuw" (no unsigned wrap) flag to the instruction that produces %10. We can see that this is safe by observing that (1) the induction variable is always increasing and (2) that if a variable starts at zero and increases, it would become undefined by passing the sign wraparound boundary next to INT_MAX before it reaches the unsigned wraparound boundary next to UINT_MAX. This flag could be used to justify subsequent optimizations.

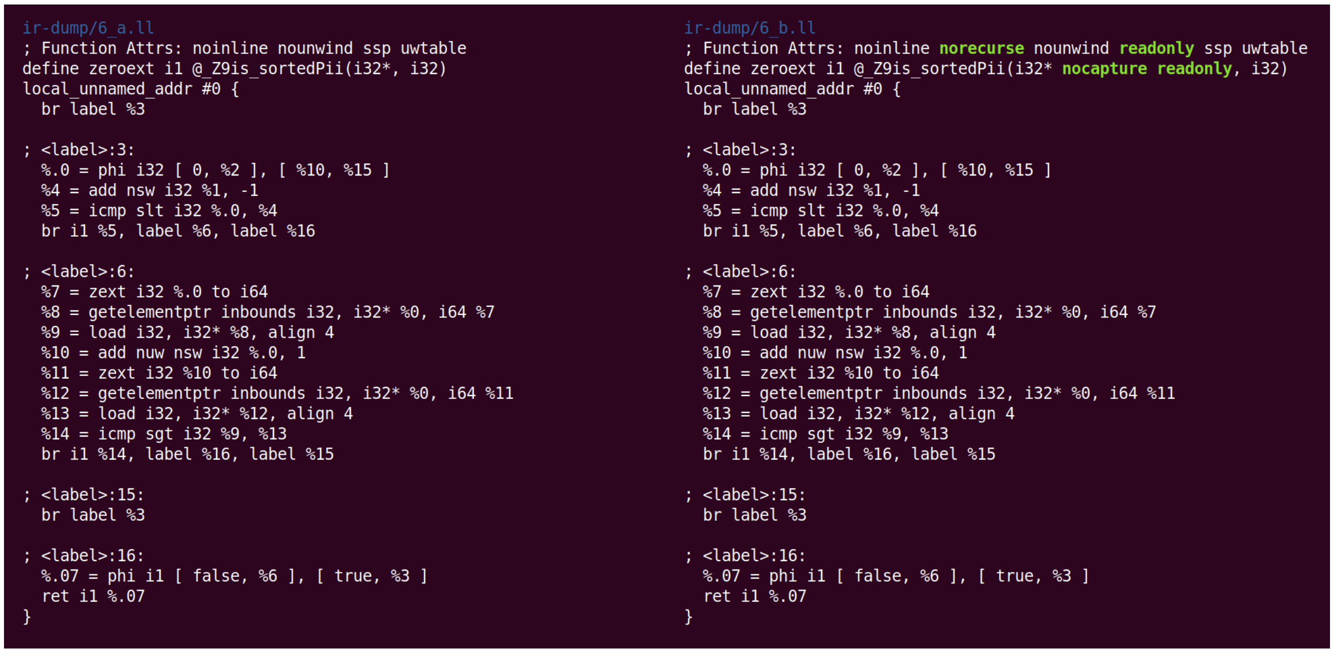
Next, SimplifyCFG runs for a second time, removing two empty basic blocks:



```
ir-dump/5_a.ll
; Function Attrs: noinline nounwind ssp uwtable
define zeroext i1 @_Z9is_sortedPii(i32*, i32)
local_unnamed_addr #0 {
  br label %3

; <label>:3:
  %.0 = phi i32 [ 0, %2 ], [ %10, %16 ]
  %4 = add nsw i32 %1, -1
  %5 = icmp slt i32 %.0, %4
  br i1 %5, label %6, label %17

; <label>:6:
  %7 = zext i32 %.0 to i64
  %8 = getelementptr inbounds i32, i32* %0, i64 %7
  %9 = load i32, i32* %8, align 4
  %10 = add nuw nsw i32 %.0, 1
  %11 = zext i32 %10 to i64
  %12 = getelementptr inbounds i32, i32* %0, i64 %11
  %13 = load i32, i32* %12, align 4
  %14 = icmp sgt i32 %9, %13
  br i1 %14, label %15, label %16

; <label>:15:
  br label %18

; <label>:16:
  br label %3

; <label>:17:
  br label %18

; <label>:18:
  %.07 = phi i1 [ false, %15 ], [ true, %17 ]
  ret i1 %.07
}
```

```
ir-dump/5_b.ll
; Function Attrs: noinline nounwind ssp uwtable
define zeroext i1 @_Z9is_sortedPii(i32*, i32)
local_unnamed_addr #0 {
  br label %3

; <label>:3:
  %.0 = phi i32 [ 0, %2 ], [ %10, %15 ]
  %4 = add nsw i32 %1, -1
  %5 = icmp slt i32 %.0, %4
  br i1 %5, label %6, label %16

; <label>:6:
  %7 = zext i32 %.0 to i64
  %8 = getelementptr inbounds i32, i32* %0, i64 %7
  %9 = load i32, i32* %8, align 4
  %10 = add nuw nsw i32 %.0, 1
  %11 = zext i32 %10 to i64
  %12 = getelementptr inbounds i32, i32* %0, i64 %11
  %13 = load i32, i32* %12, align 4
  %14 = icmp sgt i32 %9, %13
  br i1 %14, label %16, label %15

; <label>:15:
  br label %3

; <label>:16:






  %.07 = phi i1 [ false, %6 ], [ true, %3 ]
  ret i1 %.07
}
```

"Deduce function attributes" annotates the function further:

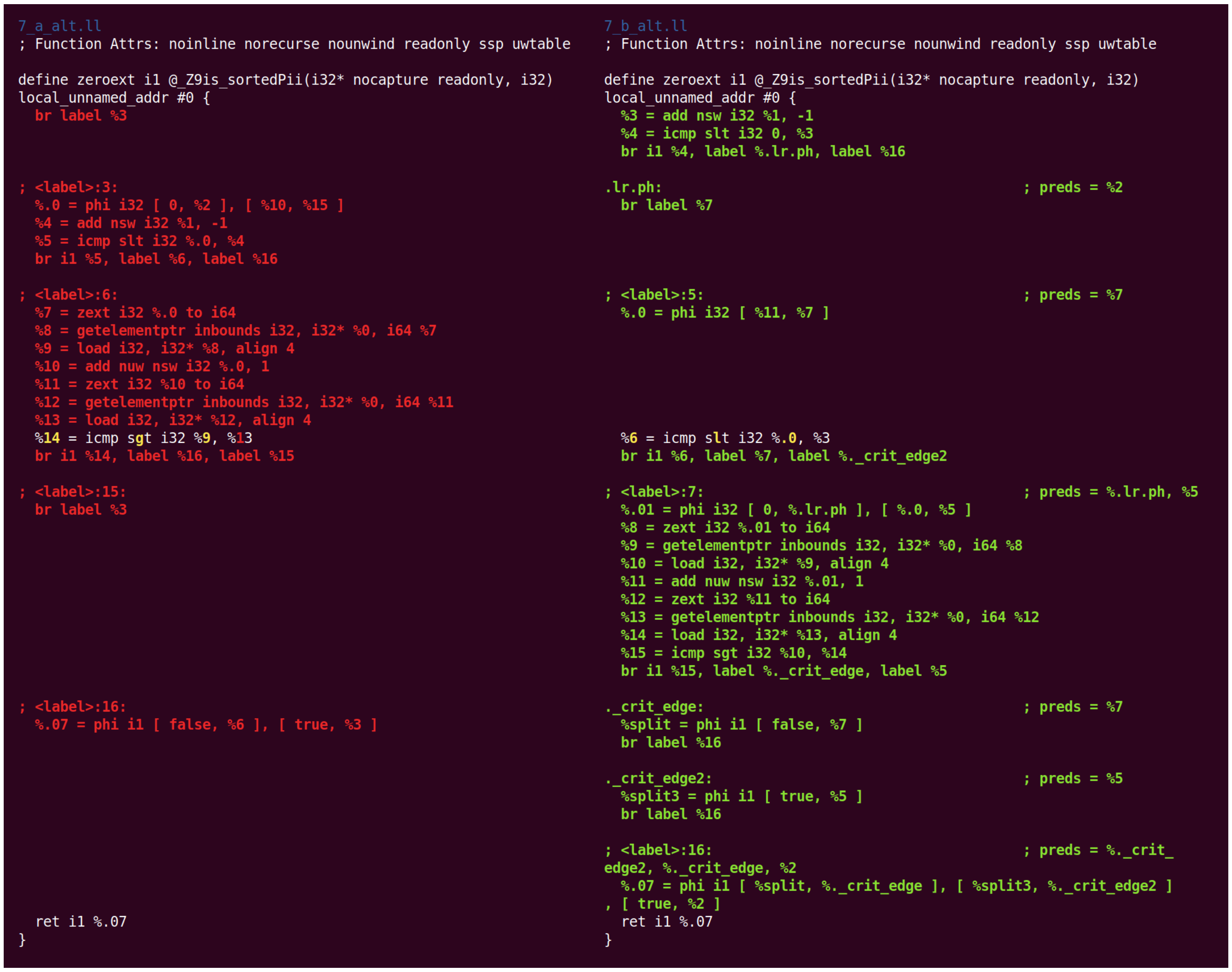


```
ir-dump/6_a.ll
; Function Attrs: noinline nounwind ssp uwtable
define zeroext i1 @_Z9is_sortedPii(i32*, i32)
local_unnamed_addr #0 {
  br label %3

; <label>:3:
  %.0 = phi i32 [ 0, %2 ], [ %10, %15 ]
  %4 = add nsw i32 %1, -1
  %5 = icmp slt i32 %.0, %4
  br i1 %5, label %6, label %16

; <label>:6:
  %7 = zext i32 %.0 to i64
  %8 = getelementptr inbounds i32, i32* %0, i64 %7
  %9 = load i32, i32* %8, align 4
  %10 = add nuw nsw i32 %.0, 1
  %11 = zext i32 %10 to i64
  %12 = getelementptr inbounds i32, i32* %0, i64 %11
  %13 = load i32, i32* %12, align 4
  %14 = icmp sgt i32 %9, %13
  br i1 %14, label %16, label %15

; <label>:15:
  br label %3

; <label>:16:
  %.07 = phi i1 [ false, %6 ], [ true, %3 ]
  ret i1 %.07
}
```

```
ir-dump/6_b.ll
; Function Attrs: noinline norecurse nounwind readonly ssp uwtable
define zeroext i1 @_Z9is_sortedPii(i32* nocapture readonly, i32)
local_unnamed_addr #0 {
  br label %3

; <label>:3:
  %.0 = phi i32 [ 0, %2 ], [ %10, %15 ]
  %4 = add nsw i32 %1, -1
  %5 = icmp slt i32 %.0, %4
  br i1 %5, label %6, label %16

; <label>:6:
  %7 = zext i32 %.0 to i64
  %8 = getelementptr inbounds i32, i32* %0, i64 %7
  %9 = load i32, i32* %8, align 4
  %10 = add nuw nsw i32 %.0, 1
  %11 = zext i32 %10 to i64
  %12 = getelementptr inbounds i32, i32* %0, i64 %11
  %13 = load i32, i32* %12, align 4
  %14 = icmp sgt i32 %9, %13
  br i1 %14, label %16, label %15

; <label>:15:
  br label %3

; <label>:16:
  %.07 = phi i1 [ false, %6 ], [ true, %3 ]
  ret i1 %.07
}
```

"Norecurse" means that the function is not involved in any recursive loop and a "readonly" function does not mutate global state. A "nocapture" parameter is not saved anywhere after its function returns and a "readonly" parameter refers to storage not modified by the function. Also see the full set of function attributes and the full set of parameter attributes.

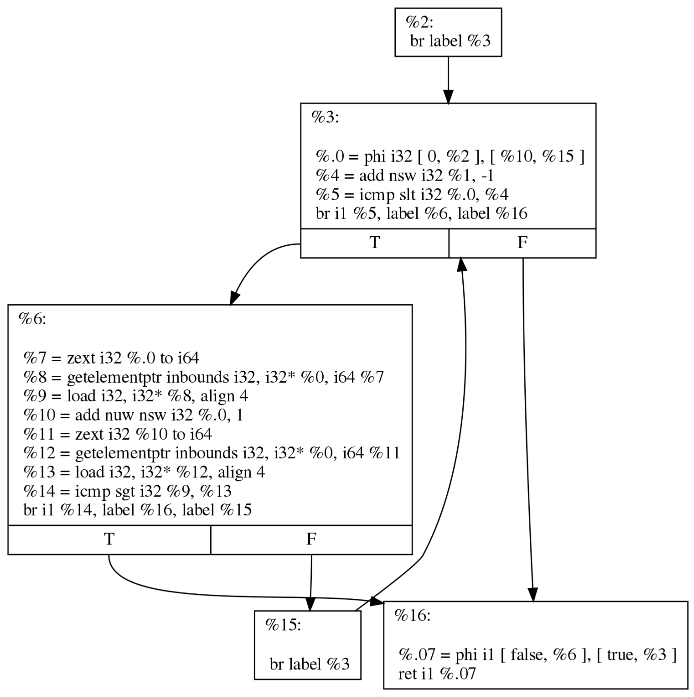
Next, "rotate loops" moves code around in an attempt to improve conditions for subsequent optimizations:



```
7_a_alt.ll
; Function Attrs: noinline norecurse nounwind readonly ssp uwtable

define zeroext i1 @_Z9is_sortedPii(i32* nocapture readonly, i32)
local_unnamed_addr #0 {
  br label %3


; <label>:3:
  %.0 = phi i32 [ 0, %2 ], [ %10, %15 ]
  %4 = add nsw i32 %1, -1
  %5 = icmp slt i32 %.0, %4
  br i1 %5, label %6, label %16

; <label>:6:
  %7 = zext i32 %.0 to i64
  %8 = getelementptr inbounds i32, i32* %0, i64 %7
  %9 = load i32, i32* %8, align 4
  %10 = add nuw nsw i32 %.0, 1
  %11 = zext i32 %10 to i64
  %12 = getelementptr inbounds i32, i32* %0, i64 %11
  %13 = load i32, i32* %12, align 4
  %14 = icmp sgt i32 %9, %13
  br i1 %14, label %16, label %15

; <label>:15:
  br label %3

; <label>:16:
  %.07 = phi i1 [ false, %6 ], [ true, %3 ]

  ret i1 %.07
}
```

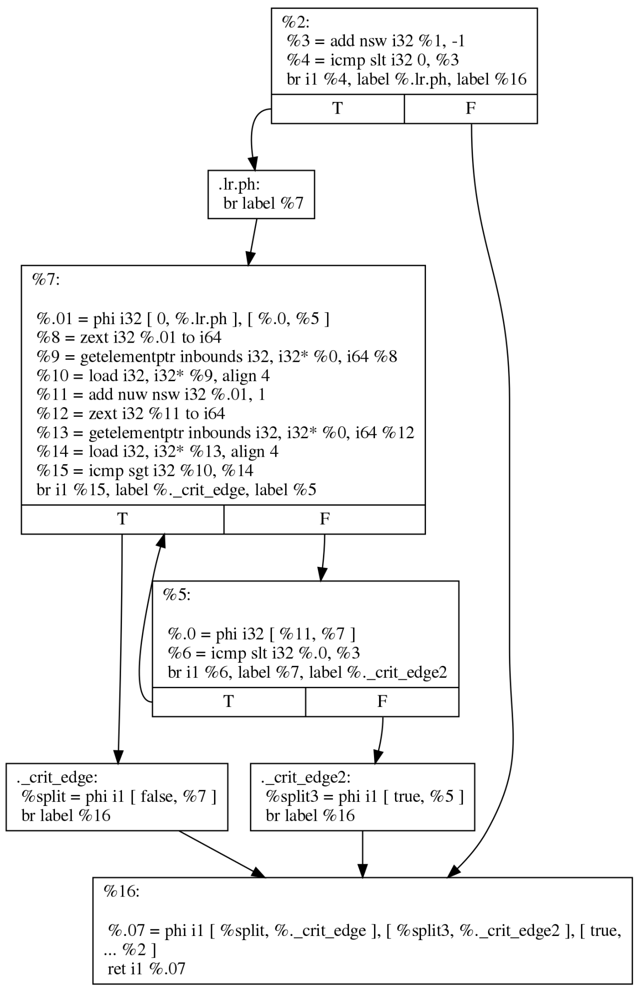
```
7_b_alt.ll
; Function Attrs: noinline norecurse nounwind readonly ssp uwtable

define zeroext i1 @_Z9is_sortedPii(i32* nocapture readonly, i32)
local_unnamed_addr #0 {
  %3 = add nsw i32 %1, -1
  %4 = icmp slt i32 0, %3
  br i1 %4, label %.lr.ph, label %16

.lr.ph:                                           ; preds = %2
  br label %7

; <label>:5:                                      ; preds = %7
  %.0 = phi i32 [ %11, %7 ]

  %6 = icmp slt i32 %.0, %3
  br i1 %6, label %7, label %._crit_edge2

; <label>:7:                                      ; preds = %.lr.ph, %5
  %.01 = phi i32 [ 0, %.lr.ph ], [ %.0, %5 ]
  %8 = zext i32 %.01 to i64
  %9 = getelementptr inbounds i32, i32* %0, i64 %8
  %10 = load i32, i32* %9, align 4
  %11 = add nuw nsw i32 %.01, 1
  %12 = zext i32 %11 to i64
  %13 = getelementptr inbounds i32, i32* %0, i64 %12
  %14 = load i32, i32* %13, align 4
  %15 = icmp sgt i32 %10, %14
  br i1 %15, label %._crit_edge, label %5

._crit_edge:                                      ; preds = %7
  %split = phi i1 [ false, %7 ]
  br label %16

._crit_edge2:                                     ; preds = %5
  %split3 = phi i1 [ true, %5 ]
  br label %16

; <label>:16:                                     ; preds = %._crit_
edge2, %._crit_edge, %2
  %.07 = phi i1 [ %split, %._crit_edge ], [ %split3, %._crit_edge2 ]
, [ true, %2 ]
  ret i1 %.07
}
```

Although this diff looks a bit alarming, there's not all that much happening. We can see what happened more readily by asking LLVM to draw the control flow graph before and after loop rotation. Here are the before (left) and after (right) views:



```
%2:
 br label %3

%3:

 %.0 = phi i32 [ 0, %2 ], [ %10, %15 ]
 %4 = add nsw i32 %1, -1
 %5 = icmp slt i32 %.0, %4
 br i1 %5, label %6, label %16

%6:

 %7 = zext i32 %.0 to i64
 %8 = getelementptr inbounds i32, i32* %0, i64 %7
 %9 = load i32, i32* %8, align 4
 %10 = add nuw nsw i32 %.0, 1
 %11 = zext i32 %10 to i64
 %12 = getelementptr inbounds i32, i32* %0, i64 %11
 %13 = load i32, i32* %12, align 4
 %14 = icmp sgt i32 %9, %13
 br i1 %14, label %16, label %15

%15:

 br label %3

%16:

 %.07 = phi i1 [ false, %6 ], [ true, %3 ]
 ret i1 %.07
```

CFG for '_Z9is_sortedPii' function



```
%2:
 %3 = add nsw i32 %1, -1
 %4 = icmp slt i32 0, %3
 br i1 %4, label %.lr.ph, label %16

.lr.ph:
 br label %7

%7:

 %.01 = phi i32 [ 0, %.lr.ph ], [ %.0, %5 ]
 %8 = zext i32 %.01 to i64
 %9 = getelementptr inbounds i32, i32* %0, i64 %8
 %10 = load i32, i32* %9, align 4
 %11 = add nuw nsw i32 %.01, 1
 %12 = zext i32 %11 to i64
 %13 = getelementptr inbounds i32, i32* %0, i64 %12
 %14 = load i32, i32* %13, align 4
 %15 = icmp sgt i32 %10, %14
 br i1 %15, label %._crit_edge, label %5

%5:

 %.0 = phi i32 [ %11, %7 ]
 %6 = icmp slt i32 %.0, %3
 br i1 %6, label %7, label %._crit_edge2

._crit_edge:
 %split = phi i1 [ false, %7 ]
 br label %16

._crit_edge2:
 %split3 = phi i1 [ true, %5 ]
 br label %16

%16:

 %.07 = phi i1 [ %split, %._crit_edge ], [ %split3, %._crit_edge2 ], [ true,
... %2 ]
 ret i1 %.07
```

CFG for '_Z9is_sortedPii' function

The original code still matches the loop structure emitted by Clang:

```
  initializer
  goto COND
COND:
  if (condition)
    goto BODY
  else
    goto EXIT
BODY:
  body
  modifier
  goto COND
EXIT:
```

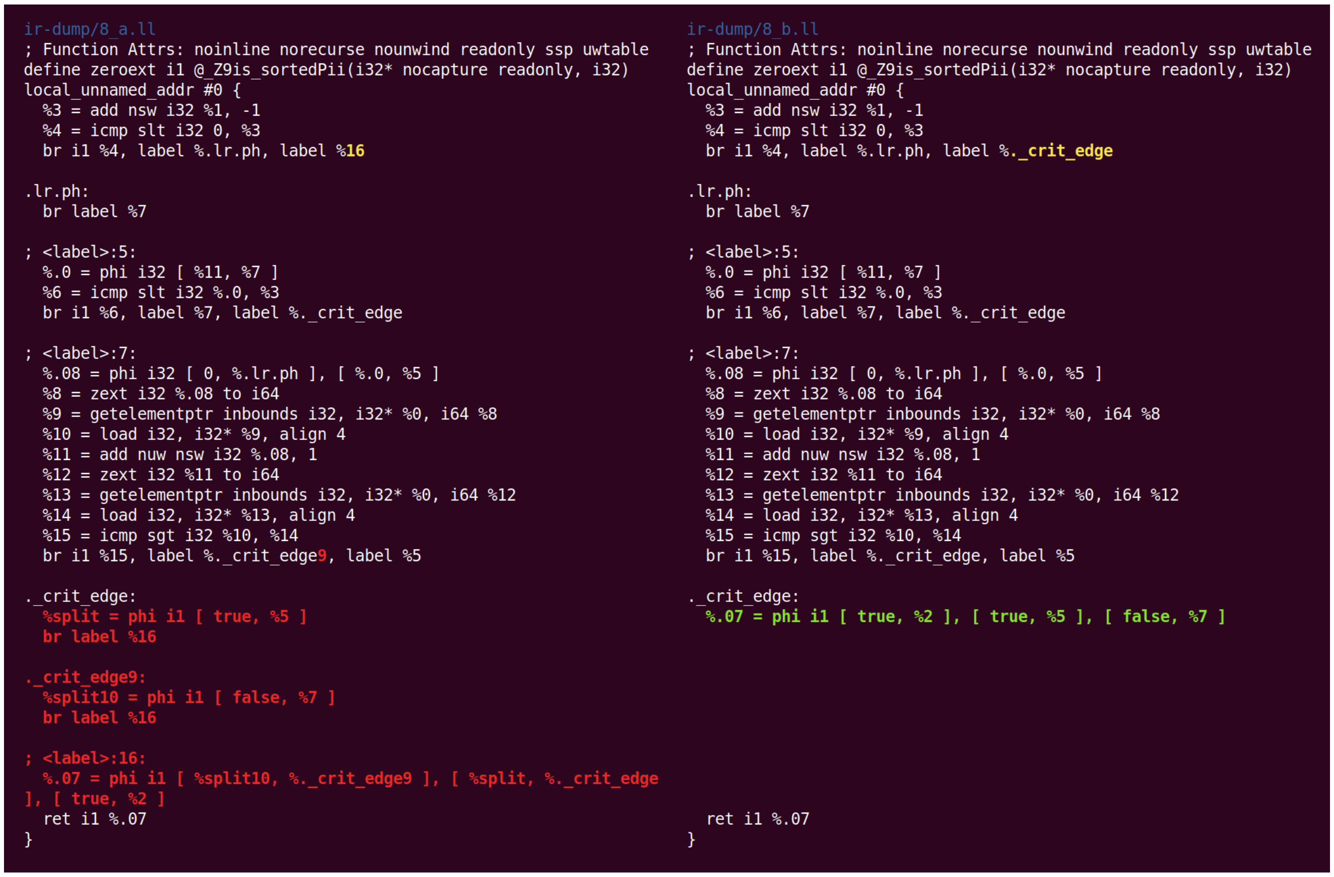
Whereas the rotated loop looks like this:

```
  initializer
  if (condition)
    goto BODY
  else
    goto EXIT
BODY:
  body
  modifier
  if (condition)
    goto BODY
  else
    goto EXIT
EXIT:
```

(Corrected as suggested by Johannes Doerfert below– thanks!)

The point of loop rotation is to remove one branch and to enable subsequent optimizations. I did not find a better description of this transformation on the web (the paper that appears to have introduced the term doesn’t seem all that relevant).

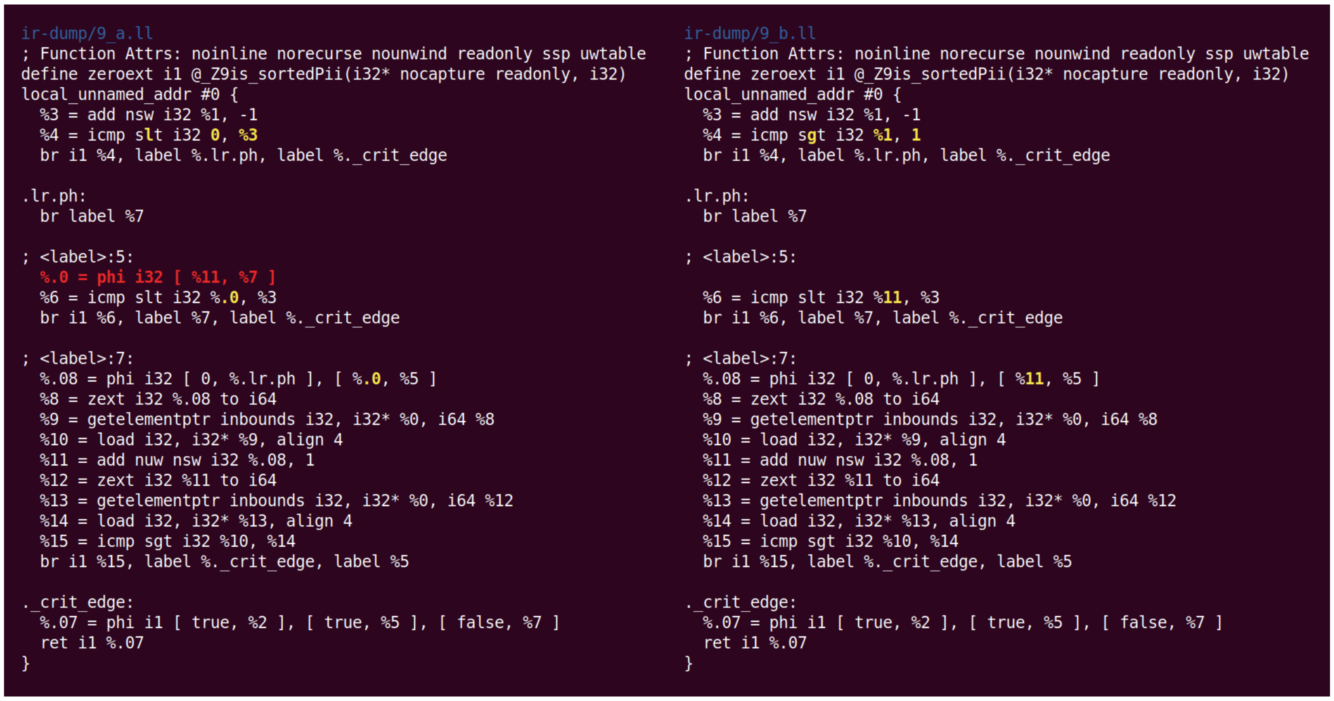
CFG simplification folds away the two basic blocks that contain only degenerate (single-input) phi nodes:



```
ir-dump/8_a.ll
; Function Attrs: noinline norecurse nounwind readonly ssp uwtable
define zeroext i1 @_Z9is_sortedPii(i32* nocapture readonly, i32)
local_unnamed_addr #0 {
  %3 = add nsw i32 %1, -1
  %4 = icmp slt i32 0, %3
  br i1 %4, label %.lr.ph, label %16

.lr.ph:
  br label %7

; <label>:5:
  %.0 = phi i32 [ %11, %7 ]
  %6 = icmp slt i32 %.0, %3
  br i1 %6, label %7, label %._crit_edge

; <label>:7:
  %.08 = phi i32 [ 0, %.lr.ph ], [ %.0, %5 ]
  %8 = zext i32 %.08 to i64
  %9 = getelementptr inbounds i32, i32* %0, i64 %8
  %10 = load i32, i32* %9, align 4
  %11 = add nuw nsw i32 %.08, 1
  %12 = zext i32 %11 to i64
  %13 = getelementptr inbounds i32, i32* %0, i64 %12
  %14 = load i32, i32* %13, align 4
  %15 = icmp sgt i32 %10, %14
  br i1 %15, label %._crit_edge9, label %5

._crit_edge:
  %split = phi i1 [ true, %5 ]
  br label %16

._crit_edge9:
  %split10 = phi i1 [ false, %7 ]
  br label %16

; <label>:16:
  %.07 = phi i1 [ %split10, %._crit_edge9 ], [ %split, %._crit_edge
], [ true, %2 ]
  ret i1 %.07
}
```

```
ir-dump/8_b.ll
; Function Attrs: noinline norecurse nounwind readonly ssp uwtable
define zeroext i1 @_Z9is_sortedPii(i32* nocapture readonly, i32)
local_unnamed_addr #0 {
  %3 = add nsw i32 %1, -1
  %4 = icmp slt i32 0, %3
  br i1 %4, label %.lr.ph, label %._crit_edge

.lr.ph:
  br label %7

; <label>:5:
  %.0 = phi i32 [ %11, %7 ]
  %6 = icmp slt i32 %.0, %3
  br i1 %6, label %7, label %._crit_edge

; <label>:7:
  %.08 = phi i32 [ 0, %.lr.ph ], [ %.0, %5 ]
  %8 = zext i32 %.08 to i64
  %9 = getelementptr inbounds i32, i32* %0, i64 %8
  %10 = load i32, i32* %9, align 4
  %11 = add nuw nsw i32 %.08, 1
  %12 = zext i32 %11 to i64
  %13 = getelementptr inbounds i32, i32* %0, i64 %12
  %14 = load i32, i32* %13, align 4
  %15 = icmp sgt i32 %10, %14
  br i1 %15, label %._crit_edge, label %5

._crit_edge:
  %.07 = phi i1 [ true, %2 ], [ true, %5 ], [ false, %7 ]
  ret i1 %.07
}
```

The instruction combiner rewrites “%4 = 0 s< (%1 - 1)" as "%4 = %1 s> 1″, this kind of thing is useful because it reduces the length of a dependency chain and may also create dead instructions (see the patch making this happen). This pass also eliminates another trivial phi node that was added during loop rotation:



```
ir-dump/9_a.ll
; Function Attrs: noinline norecurse nounwind readonly ssp uwtable
define zeroext i1 @_Z9is_sortedPii(i32* nocapture readonly, i32)
local_unnamed_addr #0 {
  %3 = add nsw i32 %1, -1
  %4 = icmp slt i32 0, %3
  br i1 %4, label %.lr.ph, label %._crit_edge

.lr.ph:
  br label %7

; <label>:5:
  %.0 = phi i32 [ %11, %7 ]
  %6 = icmp slt i32 %.0, %3
  br i1 %6, label %7, label %._crit_edge

; <label>:7:
  %.08 = phi i32 [ 0, %.lr.ph ], [ %.0, %5 ]
  %8 = zext i32 %.08 to i64
  %9 = getelementptr inbounds i32, i32* %0, i64 %8
  %10 = load i32, i32* %9, align 4
  %11 = add nuw nsw i32 %.08, 1
  %12 = zext i32 %11 to i64
  %13 = getelementptr inbounds i32, i32* %0, i64 %12
  %14 = load i32, i32* %13, align 4
  %15 = icmp sgt i32 %10, %14
  br i1 %15, label %._crit_edge, label %5

._crit_edge:
  %.07 = phi i1 [ true, %2 ], [ true, %5 ], [ false, %7 ]
  ret i1 %.07
}
```

```
ir-dump/9_b.ll
; Function Attrs: noinline norecurse nounwind readonly ssp uwtable
define zeroext i1 @_Z9is_sortedPii(i32* nocapture readonly, i32)
local_unnamed_addr #0 {
  %3 = add nsw i32 %1, -1
  %4 = icmp sgt i32 %1, 1
  br i1 %4, label %.lr.ph, label %._crit_edge

.lr.ph:
  br label %7

; <label>:5:

  %6 = icmp slt i32 %11, %3
  br i1 %6, label %7, label %._crit_edge

; <label>:7:
  %.08 = phi i32 [ 0, %.lr.ph ], [ %11, %5 ]
  %8 = zext i32 %.08 to i64
  %9 = getelementptr inbounds i32, i32* %0, i64 %8
  %10 = load i32, i32* %9, align 4
  %11 = add nuw nsw i32 %.08, 1
  %12 = zext i32 %11 to i64
  %13 = getelementptr inbounds i32, i32* %0, i64 %12
  %14 = load i32, i32* %13, align 4
  %15 = icmp sgt i32 %10, %14
  br i1 %15, label %._crit_edge, label %5

._crit_edge:
  %.07 = phi i1 [ true, %2 ], [ true, %5 ], [ false, %7 ]
  ret i1 %.07
}
```

Next, “canonicalize natural loops” runs, it self-describes as:
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.

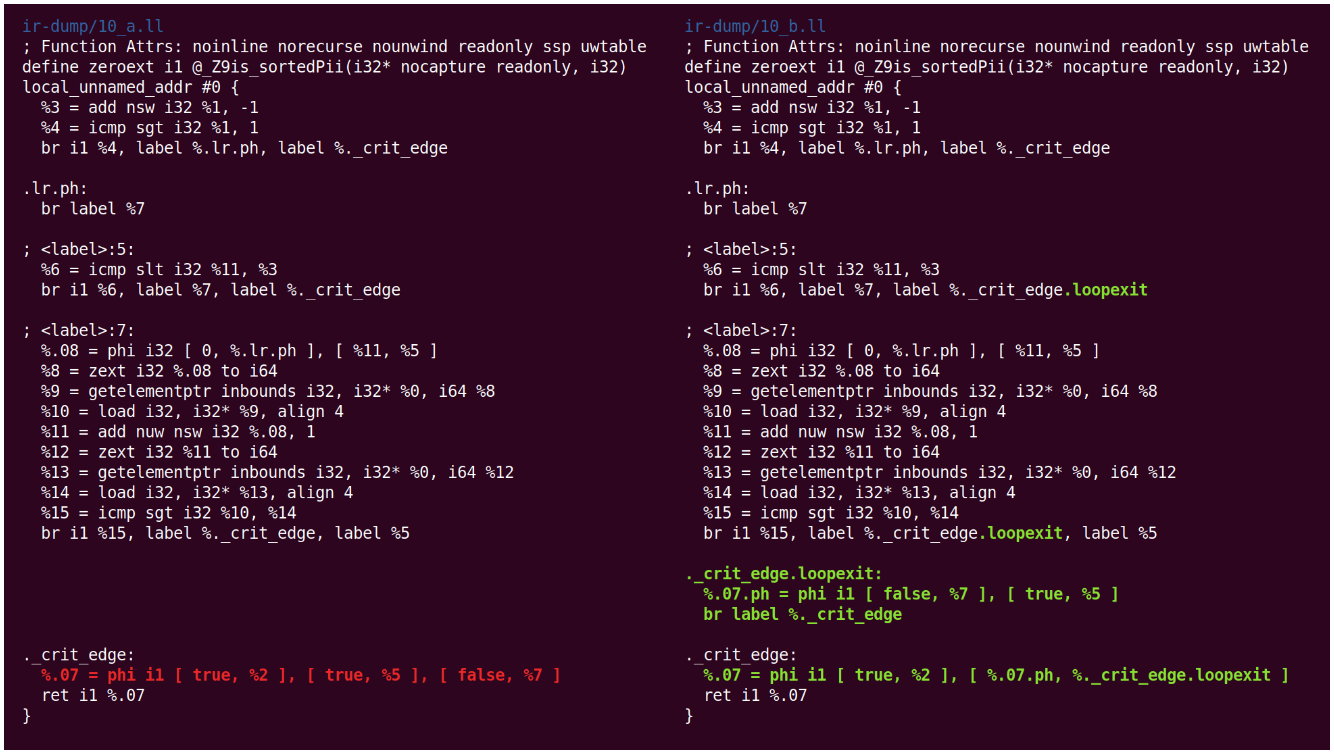
This pass also guarantees that loops will have exactly one backedge.

Indirectbr instructions introduce several complications. If the loop contains or is entered by an indirectbr instruction, it may not be possible to transform the loop and make these guarantees. Client code should check that these conditions are true before relying on them.

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.

Here we can see loop exit-block insertion happen:



```
ir-dump/10_a.ll
; Function Attrs: noinline norecurse nounwind readonly ssp uwtable
define zeroext i1 @_Z9is_sortedPii(i32* nocapture readonly, i32)
local_unnamed_addr #0 {
  %3 = add nsw i32 %1, -1
  %4 = icmp sgt i32 %1, 1
  br i1 %4, label %.lr.ph, label %._crit_edge

.lr.ph:
  br label %7

; <label>:5:
  %6 = icmp slt i32 %11, %3
  br i1 %6, label %7, label %._crit_edge

; <label>:7:
  %.08 = phi i32 [ 0, %.lr.ph ], [ %11, %5 ]
  %8 = zext i32 %.08 to i64
  %9 = getelementptr inbounds i32, i32* %0, i64 %8
  %10 = load i32, i32* %9, align 4
  %11 = add nuw nsw i32 %.08, 1
  %12 = zext i32 %11 to i64
  %13 = getelementptr inbounds i32, i32* %0, i64 %12
  %14 = load i32, i32* %13, align 4
  %15 = icmp sgt i32 %10, %14
  br i1 %15, label %._crit_edge, label %5

._crit_edge:
  %.07 = phi i1 [ true, %2 ], [ true, %5 ], [ false, %7 ]
  ret i1 %.07
}
```

```
ir-dump/10_b.ll
; Function Attrs: noinline norecurse nounwind readonly ssp uwtable
define zeroext i1 @_Z9is_sortedPii(i32* nocapture readonly, i32)
local_unnamed_addr #0 {
  %3 = add nsw i32 %1, -1
  %4 = icmp sgt i32 %1, 1
  br i1 %4, label %.lr.ph, label %._crit_edge

.lr.ph:
  br label %7

; <label>:5:
  %6 = icmp slt i32 %11, %3
  br i1 %6, label %7, label %._crit_edge.loopexit

; <label>:7:
  %.08 = phi i32 [ 0, %.lr.ph ], [ %11, %5 ]
  %8 = zext i32 %.08 to i64
  %9 = getelementptr inbounds i32, i32* %0, i64 %8
  %10 = load i32, i32* %9, align 4
  %11 = add nuw nsw i32 %.08, 1
  %12 = zext i32 %11 to i64
  %13 = getelementptr inbounds i32, i32* %0, i64 %12
  %14 = load i32, i32* %13, align 4
  %15 = icmp sgt i32 %10, %14
  br i1 %15, label %._crit_edge.loopexit, label %5

._crit_edge.loopexit:
  %.07.ph = phi i1 [ false, %7 ], [ true, %5 ]
  br label %._crit_edge

._crit_edge:
  %.07 = phi i1 [ true, %2 ], [ %.07.ph, %._crit_edge.loopexit ]
  ret i1 %.07
}
```

Next up is "induction variable simplification":
This transformation analyzes and transforms the induction variables (and computations derived from them) into simpler forms suitable for subsequent analysis and transformation.

If the trip count of a loop is computable, this pass also makes the following changes:

1. The exit condition for the loop is canonicalized to compare the induction value against the exit value. This turns loops like: 'for (i = 7; i*i < 1000; ++i)' into 'for (i = 0; i != 25; ++i)'
2. Any use outside of the loop of an expression derived from the indvar is changed to compute the derived value outside of the loop, eliminating the dependence on the exit value of the induction variable. If the only purpose of the loop is to compute the exit value of some derived expression, this transformation will make the loop dead.

The effect of this pass here is simply to rewrite the 32-bit induction variable as 64 bits:

```
ir-dump/11_a.ll
; Preheader:
.lr.ph:

  br label %7

; Loop:
; <label>:7:
  %.08 = phi i32 [ 0, %.lr.ph ], [ %11, %5 ]
  %8 = zext i32 %.08 to i64
  %9 = getelementptr inbounds i32, i32* %0, i64 %8
  %10 = load i32, i32* %9, align 4
  %11 = add nuw nsw i32 %.08, 1
  %12 = zext i32 %11 to i64
  %13 = getelementptr inbounds i32, i32* %0, i64 %12
  %14 = load i32, i32* %13, align 4
  %15 = icmp sgt i32 %10, %14
  br i1 %15, label %._crit_edge.loopexit, label %5

; <label>:5:
  %6 = icmp slt i32 %11, %3
  br i1 %6, label %7, label %._crit_edge.loopexit

; Exit blocks
._crit_edge.loopexit:
  %.07.ph = phi i1 [ false, %7 ], [ true, %5 ]
  br label %._crit_edge

._crit_edge.loopexit:
  %.07.ph = phi i1 [ false, %7 ], [ true, %5 ]
  br label %._crit_edge
```

```
ir-dump/11_b.ll
; Preheader:
.lr.ph:
  %5 = sext i32 %3 to i64
  br label %8

; Loop:
; <label>:8:
  %indvars.iv = phi i64 [ 0, %.lr.ph ], [ %indvars.iv.next, %6 ]

  %9 = getelementptr inbounds i32, i32* %0, i64 %indvars.iv
  %10 = load i32, i32* %9, align 4
  %indvars.iv.next = add nuw nsw i64 %indvars.iv, 1

  %11 = getelementptr inbounds i32, i32* %0, i64 %indvars.iv.next
  %12 = load i32, i32* %11, align 4
  %13 = icmp sgt i32 %10, %12
  br i1 %13, label %._crit_edge.loopexit, label %6

; <label>:6:
  %7 = icmp slt i64 %indvars.iv.next, %5
  br i1 %7, label %8, label %._crit_edge.loopexit

; Exit blocks
._crit_edge.loopexit:
  %.07.ph = phi i1 [ false, %8 ], [ true, %6 ]
  br label %._crit_edge

._crit_edge.loopexit:
  %.07.ph = phi i1 [ false, %8 ], [ true, %6 ]
  br label %._crit_edge
```

I don’t know why the zext — previously canonicalized from a sext — is turned back into a sext.

Now “global value numbering” performs a very clever optimization. Showing this one off is basically the entire reason that I decided to write this blog post. See if you can figure it out just by reading the diff:

```
ir-dump/12_a.ll
; Function Attrs: noinline norecurse nounwind readonly ssp uwtable
define zeroext i1 @_Z9is_sortedPii(i32* nocapture readonly, i32)
local_unnamed_addr #0 {
  %3 = add nsw i32 %1, -1
  %4 = icmp sgt i32 %1, 1
  br i1 %4, label %.lr.ph, label %._crit_edge

.lr.ph:
  %5 = sext i32 %3 to i64

  br label %8

; <label>:6:
  %7 = icmp slt i64 %indvars.iv.next, %5
  br i1 %7, label %8, label %._crit_edge.loopexit

; <label>:8:

  %indvars.iv = phi i64 [ 0, %.lr.ph ], [ %indvars.iv.next, %6 ]
  %9 = getelementptr inbounds i32, i32* %0, i64 %indvars.iv
  %10 = load i32, i32* %9, align 4
  %indvars.iv.next = add nuw nsw i64 %indvars.iv, 1
  %11 = getelementptr inbounds i32, i32* %0, i64 %indvars.iv.next
  %12 = load i32, i32* %11, align 4
  %13 = icmp sgt i32 %10, %12
  br i1 %13, label %._crit_edge.loopexit, label %6

._crit_edge.loopexit:
  %.07.ph = phi i1 [ false, %8 ], [ true, %6 ]
  br label %._crit_edge

._crit_edge:
  %.07 = phi i1 [ true, %2 ], [ %.07.ph, %._crit_edge.loopexit ]
  ret i1 %.07
}
```

```
ir-dump/12_b.ll
; Function Attrs: noinline norecurse nounwind readonly ssp uwtable
define zeroext i1 @_Z9is_sortedPii(i32* nocapture readonly, i32)
local_unnamed_addr #0 {
  %3 = add nsw i32 %1, -1
  %4 = icmp sgt i32 %1, 1
  br i1 %4, label %.lr.ph, label %._crit_edge

.lr.ph:
  %5 = sext i32 %3 to i64
  %.pre = load i32, i32* %0, align 4
  br label %8

; <label>:6:
  %7 = icmp slt i64 %indvars.iv.next, %5
  br i1 %7, label %8, label %._crit_edge.loopexit

; <label>:8:
  %9 = phi i32 [ %.pre, %.lr.ph ], [ %12, %6 ]
  %indvars.iv = phi i64 [ 0, %.lr.ph ], [ %indvars.iv.next, %6 ]
  %10 = getelementptr inbounds i32, i32* %0, i64 %indvars.iv

  %indvars.iv.next = add nuw nsw i64 %indvars.iv, 1
  %11 = getelementptr inbounds i32, i32* %0, i64 %indvars.iv.next
  %12 = load i32, i32* %11, align 4
  %13 = icmp sgt i32 %9, %12
  br i1 %13, label %._crit_edge.loopexit, label %6

._crit_edge.loopexit:
  %.07.ph = phi i1 [ false, %8 ], [ true, %6 ]
  br label %._crit_edge

._crit_edge:
  %.07 = phi i1 [ true, %2 ], [ %.07.ph, %._crit_edge.loopexit ]
  ret i1 %.07
}
```

Got it? Ok, the two loads in the loop on the left correspond to a[i] and a[i + 1]. Here GVN has figured out that loading a[i] is unnecessary because a[i + 1] from one loop iteration can be forwarded to the next iteration as a[i]. This one simple trick halves the number of loads issued by this function. Both LLVM and GCC only got this transformation recently.

You might be asking yourself if this trick still works if we compare a[i] against a[i + 2]. It turns out that LLVM is unwilling or unable to do this, but GCC is willing to throw up to four registers at this problem.

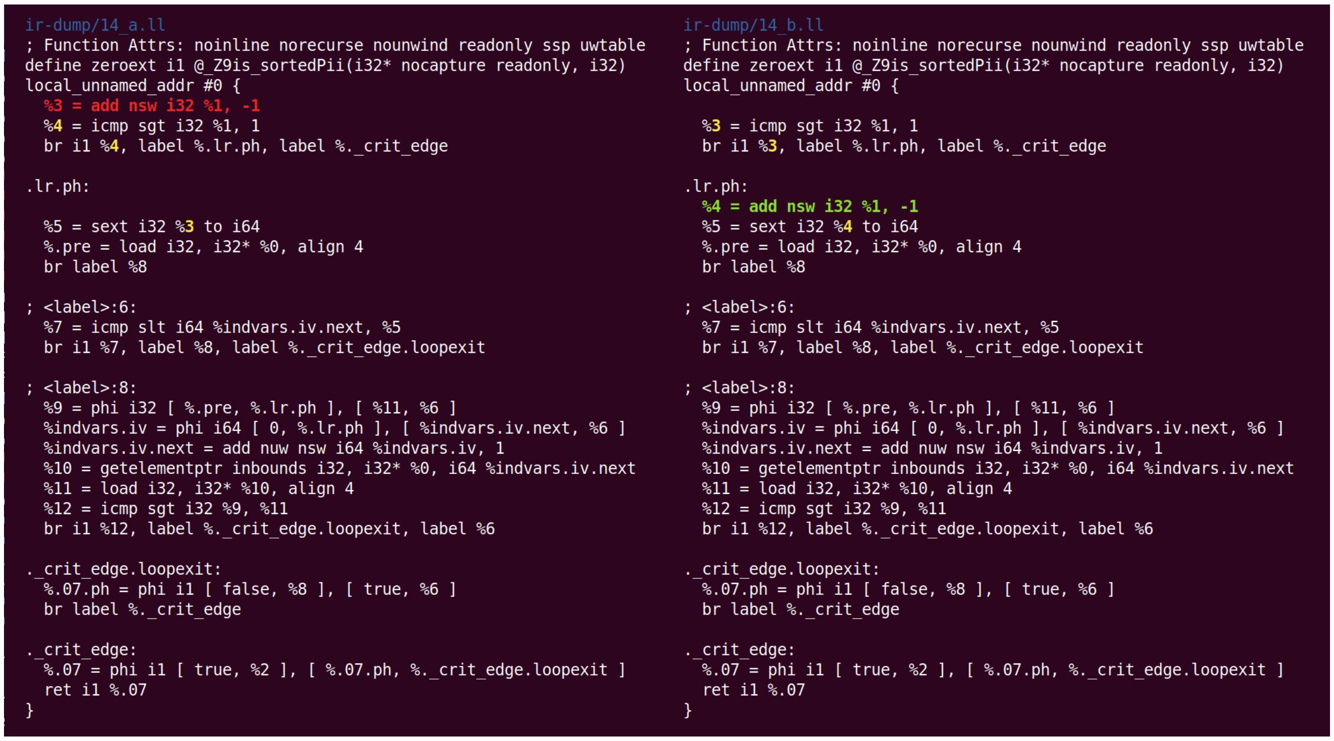
Now "bit-tracking dead code elimination" runs:
This file implements the Bit-Tracking Dead Code Elimination pass. Some instructions (shifts, some ands, ors, etc.) kill some of their input bits. We track these dead bits and remove instructions that compute only these dead bits.

But it turns out the extra cleverness isn't needed since the only dead code is a GEP, and it is trivially dead (GVN removed the load that previously used the address that it computes):

```
ir-dump/13_a.ll
; Function Attrs: noinline norecurse nounwind readonly ssp uwtable
define zeroext i1 @_Z9is_sortedPii(i32* nocapture readonly, i32)
local_unnamed_addr #0 {
  %3 = add nsw i32 %1, -1
  %4 = icmp sgt i32 %1, 1
  br i1 %4, label %.lr.ph, label %._crit_edge

.lr.ph:
  %5 = sext i32 %3 to i64
  %.pre = load i32, i32* %0, align 4
  br label %8

; <label>:6:
  %7 = icmp slt i64 %indvars.iv.next, %5
  br i1 %7, label %8, label %._crit_edge.loopexit

; <label>:8:
  %9 = phi i32 [ %.pre, %.lr.ph ], [ %12, %6 ]
  %indvars.iv = phi i64 [ 0, %.lr.ph ], [ %indvars.iv.next, %6 ]
  %10 = getelementptr inbounds i32, i32* %0, i64 %indvars.iv
  %indvars.iv.next = add nuw nsw i64 %indvars.iv, 1
  %11 = getelementptr inbounds i32, i32* %0, i64 %indvars.iv.next
  %12 = load i32, i32* %11, align 4
  %13 = icmp sgt i32 %9, %12
  br i1 %13, label %._crit_edge.loopexit, label %6

._crit_edge.loopexit:
  %.07.ph = phi i1 [ false, %8 ], [ true, %6 ]
  br label %._crit_edge

._crit_edge:
  %.07 = phi i1 [ true, %2 ], [ %.07.ph, %._crit_edge.loopexit ]
  ret i1 %.07
}
```

```
ir-dump/13_b.ll
; Function Attrs: noinline norecurse nounwind readonly ssp uwtable
define zeroext i1 @_Z9is_sortedPii(i32* nocapture readonly, i32)
local_unnamed_addr #0 {
  %3 = add nsw i32 %1, -1
  %4 = icmp sgt i32 %1, 1
  br i1 %4, label %.lr.ph, label %._crit_edge

.lr.ph:
  %5 = sext i32 %3 to i64
  %.pre = load i32, i32* %0, align 4
  br label %8

; <label>:6:
  %7 = icmp slt i64 %indvars.iv.next, %5
  br i1 %7, label %8, label %._crit_edge.loopexit

; <label>:8:
  %9 = phi i32 [ %.pre, %.lr.ph ], [ %11, %6 ]
  %indvars.iv = phi i64 [ 0, %.lr.ph ], [ %indvars.iv.next, %6 ]

  %indvars.iv.next = add nuw nsw i64 %indvars.iv, 1
  %10 = getelementptr inbounds i32, i32* %0, i64 %indvars.iv.next
  %11 = load i32, i32* %10, align 4
  %12 = icmp sgt i32 %9, %11
  br i1 %12, label %._crit_edge.loopexit, label %6

._crit_edge.loopexit:
  %.07.ph = phi i1 [ false, %8 ], [ true, %6 ]
  br label %._crit_edge

._crit_edge:
  %.07 = phi i1 [ true, %2 ], [ %.07.ph, %._crit_edge.loopexit ]
  ret i1 %.07
}
```

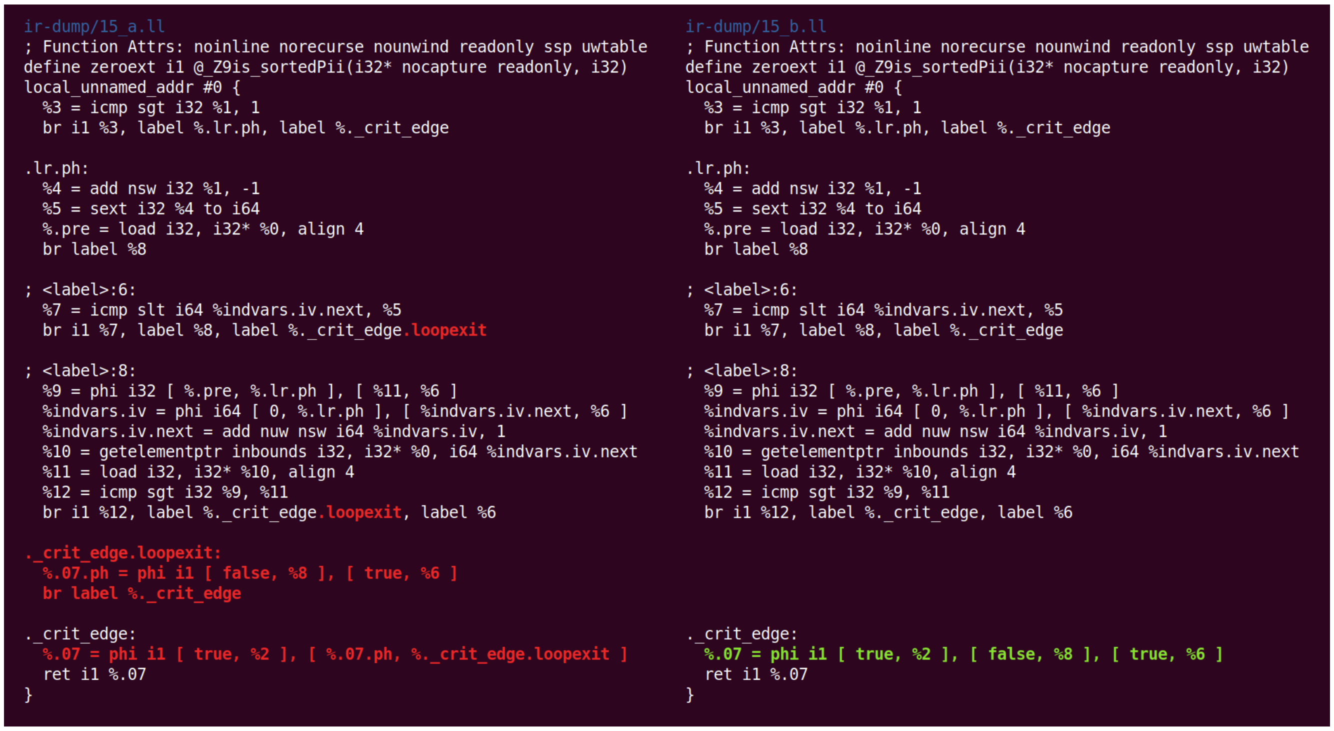
Now the instruction combiner sinks an add into a different basic block. The rationale behind putting this transformation into InstCombine is not clear to me; perhaps there just wasn't a more obvious place to do this one:



```
ir-dump/14_a.ll
; Function Attrs: noinline norecurse nounwind readonly ssp uwtable
define zeroext i1 @_Z9is_sortedPii(i32* nocapture readonly, i32)
local_unnamed_addr #0 {
  %3 = add nsw i32 %1, -1
  %4 = icmp sgt i32 %1, 1
  br i1 %4, label %.lr.ph, label %._crit_edge

.lr.ph:

  %5 = sext i32 %3 to i64
  %.pre = load i32, i32* %0, align 4
  br label %8

; <label>:6:
  %7 = icmp slt i64 %indvars.iv.next, %5
  br i1 %7, label %8, label %._crit_edge.loopexit

; <label>:8:
  %9 = phi i32 [ %.pre, %.lr.ph ], [ %11, %6 ]
  %indvars.iv = phi i64 [ 0, %.lr.ph ], [ %indvars.iv.next, %6 ]
  %indvars.iv.next = add nuw nsw i64 %indvars.iv, 1
  %10 = getelementptr inbounds i32, i32* %0, i64 %indvars.iv.next
  %11 = load i32, i32* %10, align 4
  %12 = icmp sgt i32 %9, %11
  br i1 %12, label %._crit_edge.loopexit, label %6

._crit_edge.loopexit:
  %.07.ph = phi i1 [ false, %8 ], [ true, %6 ]
  br label %._crit_edge

._crit_edge:
  %.07 = phi i1 [ true, %2 ], [ %.07.ph, %._crit_edge.loopexit ]
  ret i1 %.07
}
```

```
ir-dump/14_b.ll
; Function Attrs: noinline norecurse nounwind readonly ssp uwtable
define zeroext i1 @_Z9is_sortedPii(i32* nocapture readonly, i32)
local_unnamed_addr #0 {

  %3 = icmp sgt i32 %1, 1
  br i1 %3, label %.lr.ph, label %._crit_edge

.lr.ph:
  %4 = add nsw i32 %1, -1
  %5 = sext i32 %4 to i64
  %.pre = load i32, i32* %0, align 4
  br label %8

; <label>:6:
  %7 = icmp slt i64 %indvars.iv.next, %5
  br i1 %7, label %8, label %._crit_edge.loopexit

; <label>:8:
  %9 = phi i32 [ %.pre, %.lr.ph ], [ %11, %6 ]
  %indvars.iv = phi i64 [ 0, %.lr.ph ], [ %indvars.iv.next, %6 ]
  %indvars.iv.next = add nuw nsw i64 %indvars.iv, 1
  %10 = getelementptr inbounds i32, i32* %0, i64 %indvars.iv.next
  %11 = load i32, i32* %10, align 4
  %12 = icmp sgt i32 %9, %11
  br i1 %12, label %._crit_edge.loopexit, label %6

._crit_edge.loopexit:
  %.07.ph = phi i1 [ false, %8 ], [ true, %6 ]
  br label %._crit_edge

._crit_edge:
  %.07 = phi i1 [ true, %2 ], [ %.07.ph, %._crit_edge.loopexit ]
  ret i1 %.07
}
```

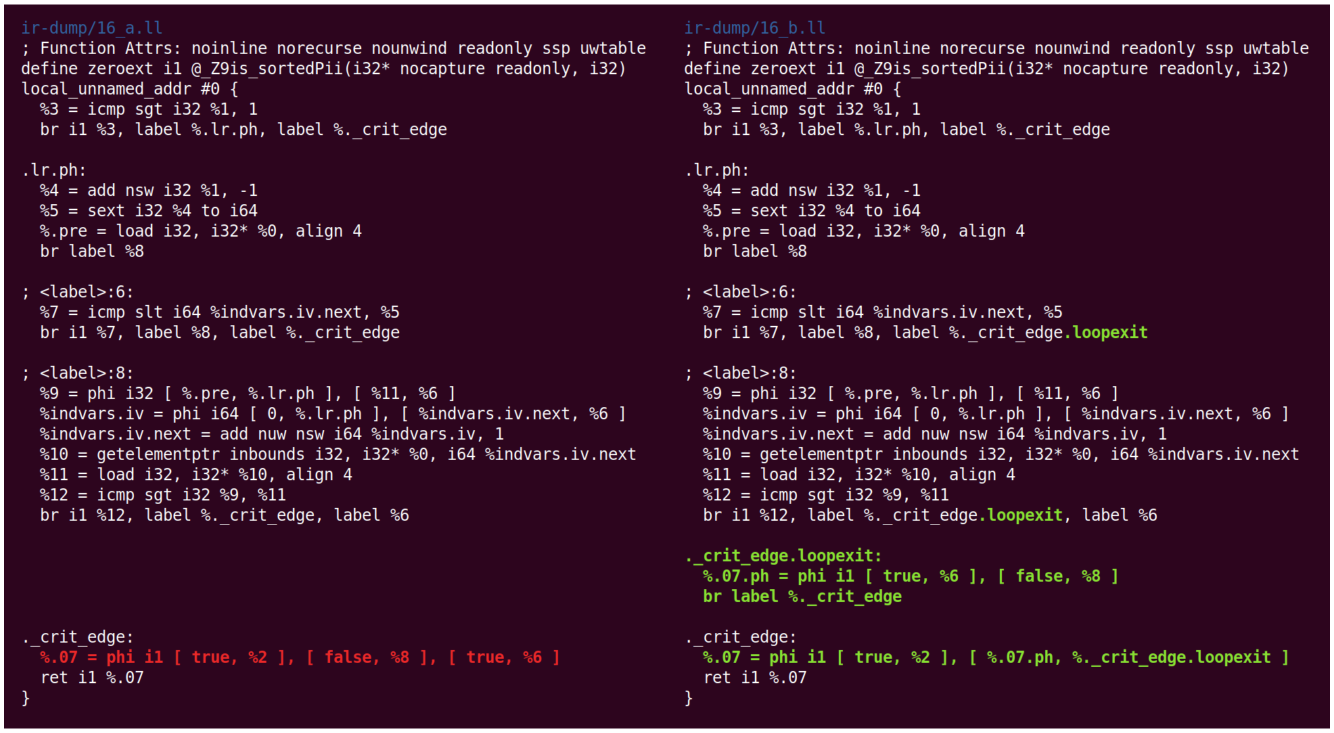
Now things get a tiny bit weird, “jump threading” undoes what “canonicalize natural loops” did earlier:



```
ir-dump/15_a.ll
; Function Attrs: noinline norecurse nounwind readonly ssp uwtable
define zeroext i1 @_Z9is_sortedPii(i32* nocapture readonly, i32)
local_unnamed_addr #0 {
  %3 = icmp sgt i32 %1, 1
  br i1 %3, label %.lr.ph, label %._crit_edge

.lr.ph:
  %4 = add nsw i32 %1, -1
  %5 = sext i32 %4 to i64
  %.pre = load i32, i32* %0, align 4
  br label %8

; <label>:6:
  %7 = icmp slt i64 %indvars.iv.next, %5
  br i1 %7, label %8, label %._crit_edge.loopexit

; <label>:8:
  %9 = phi i32 [ %.pre, %.lr.ph ], [ %11, %6 ]
  %indvars.iv = phi i64 [ 0, %.lr.ph ], [ %indvars.iv.next, %6 ]
  %indvars.iv.next = add nuw nsw i64 %indvars.iv, 1
  %10 = getelementptr inbounds i32, i32* %0, i64 %indvars.iv.next
  %11 = load i32, i32* %10, align 4
  %12 = icmp sgt i32 %9, %11
  br i1 %12, label %._crit_edge.loopexit, label %6

._crit_edge.loopexit:
  %.07.ph = phi i1 [ false, %8 ], [ true, %6 ]
  br label %._crit_edge

._crit_edge:
  %.07 = phi i1 [ true, %2 ], [ %.07.ph, %._crit_edge.loopexit ]
  ret i1 %.07
}
```

```
ir-dump/15_b.ll
; Function Attrs: noinline norecurse nounwind readonly ssp uwtable
define zeroext i1 @_Z9is_sortedPii(i32* nocapture readonly, i32)
local_unnamed_addr #0 {
  %3 = icmp sgt i32 %1, 1
  br i1 %3, label %.lr.ph, label %._crit_edge

.lr.ph:
  %4 = add nsw i32 %1, -1
  %5 = sext i32 %4 to i64
  %.pre = load i32, i32* %0, align 4
  br label %8

; <label>:6:
  %7 = icmp slt i64 %indvars.iv.next, %5
  br i1 %7, label %8, label %._crit_edge

; <label>:8:
  %9 = phi i32 [ %.pre, %.lr.ph ], [ %11, %6 ]
  %indvars.iv = phi i64 [ 0, %.lr.ph ], [ %indvars.iv.next, %6 ]
  %indvars.iv.next = add nuw nsw i64 %indvars.iv, 1
  %10 = getelementptr inbounds i32, i32* %0, i64 %indvars.iv.next
  %11 = load i32, i32* %10, align 4
  %12 = icmp sgt i32 %9, %11
  br i1 %12, label %._crit_edge, label %6



._crit_edge:
  %.07 = phi i1 [ true, %2 ], [ false, %8 ], [ true, %6 ]
  ret i1 %.07
}
```

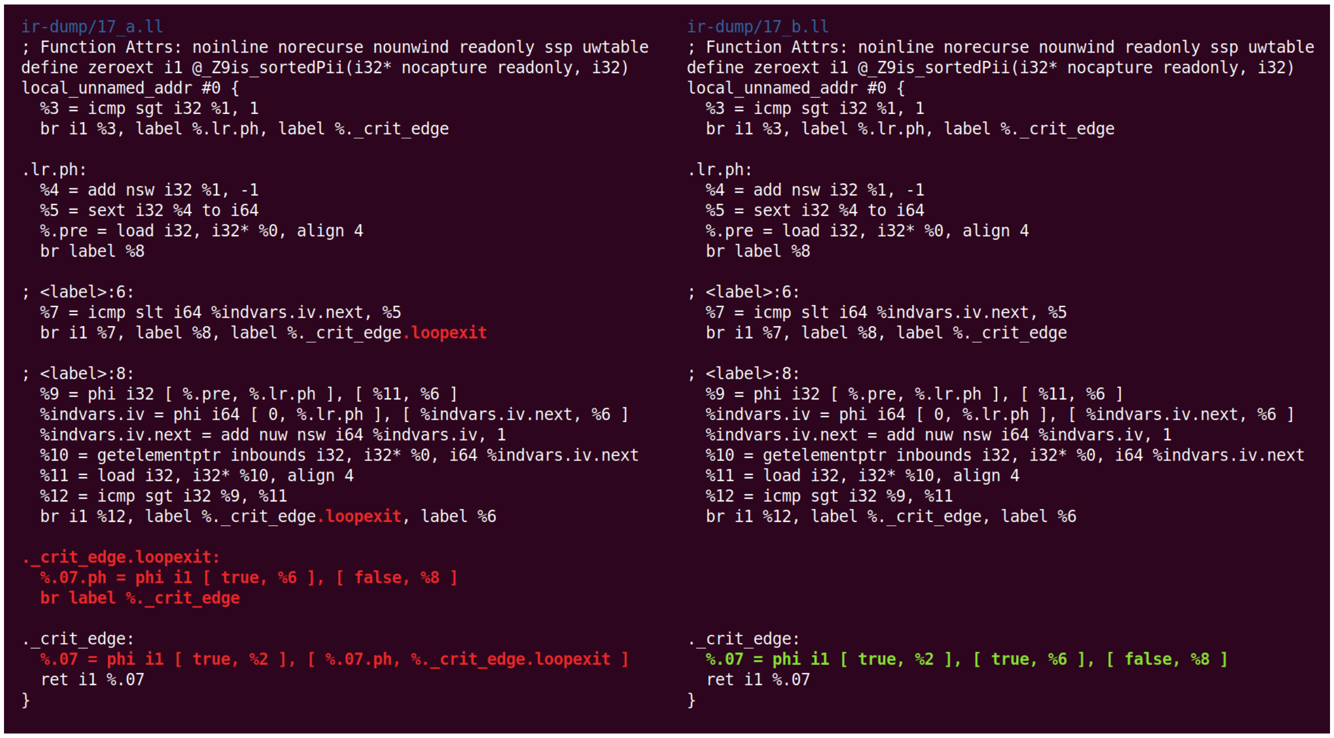
Then we canonicalize it back:



```
ir-dump/16_a.ll
; Function Attrs: noinline norecurse nounwind readonly ssp uwtable
define zeroext i1 @_Z9is_sortedPii(i32* nocapture readonly, i32)
local_unnamed_addr #0 {
  %3 = icmp sgt i32 %1, 1
  br i1 %3, label %.lr.ph, label %._crit_edge

.lr.ph:
  %4 = add nsw i32 %1, -1
  %5 = sext i32 %4 to i64
  %.pre = load i32, i32* %0, align 4
  br label %8

; <label>:6:
  %7 = icmp slt i64 %indvars.iv.next, %5
  br i1 %7, label %8, label %._crit_edge

; <label>:8:
  %9 = phi i32 [ %.pre, %.lr.ph ], [ %11, %6 ]
  %indvars.iv = phi i64 [ 0, %.lr.ph ], [ %indvars.iv.next, %6 ]
  %indvars.iv.next = add nuw nsw i64 %indvars.iv, 1
  %10 = getelementptr inbounds i32, i32* %0, i64 %indvars.iv.next
  %11 = load i32, i32* %10, align 4
  %12 = icmp sgt i32 %9, %11
  br i1 %12, label %._crit_edge, label %6

._crit_edge:
  %.07 = phi i1 [ true, %2 ], [ false, %8 ], [ true, %6 ]
  ret i1 %.07
}
```

```
ir-dump/16_b.ll
; Function Attrs: noinline norecurse nounwind readonly ssp uwtable
define zeroext i1 @_Z9is_sortedPii(i32* nocapture readonly, i32)
local_unnamed_addr #0 {
  %3 = icmp sgt i32 %1, 1
  br i1 %3, label %.lr.ph, label %._crit_edge

.lr.ph:
  %4 = add nsw i32 %1, -1
  %5 = sext i32 %4 to i64
  %.pre = load i32, i32* %0, align 4
  br label %8

; <label>:6:
  %7 = icmp slt i64 %indvars.iv.next, %5
  br i1 %7, label %8, label %._crit_edge.loopexit

; <label>:8:
  %9 = phi i32 [ %.pre, %.lr.ph ], [ %11, %6 ]
  %indvars.iv = phi i64 [ 0, %.lr.ph ], [ %indvars.iv.next, %6 ]
  %indvars.iv.next = add nuw nsw i64 %indvars.iv, 1
  %10 = getelementptr inbounds i32, i32* %0, i64 %indvars.iv.next
  %11 = load i32, i32* %10, align 4
  %12 = icmp sgt i32 %9, %11
  br i1 %12, label %._crit_edge.loopexit, label %6

._crit_edge.loopexit:
  %.07.ph = phi i1 [ true, %6 ], [ false, %8 ]
  br label %._crit_edge

._crit_edge:
  %.07 = phi i1 [ true, %2 ], [ %.07.ph, %._crit_edge.loopexit ]
  ret i1 %.07
}
```

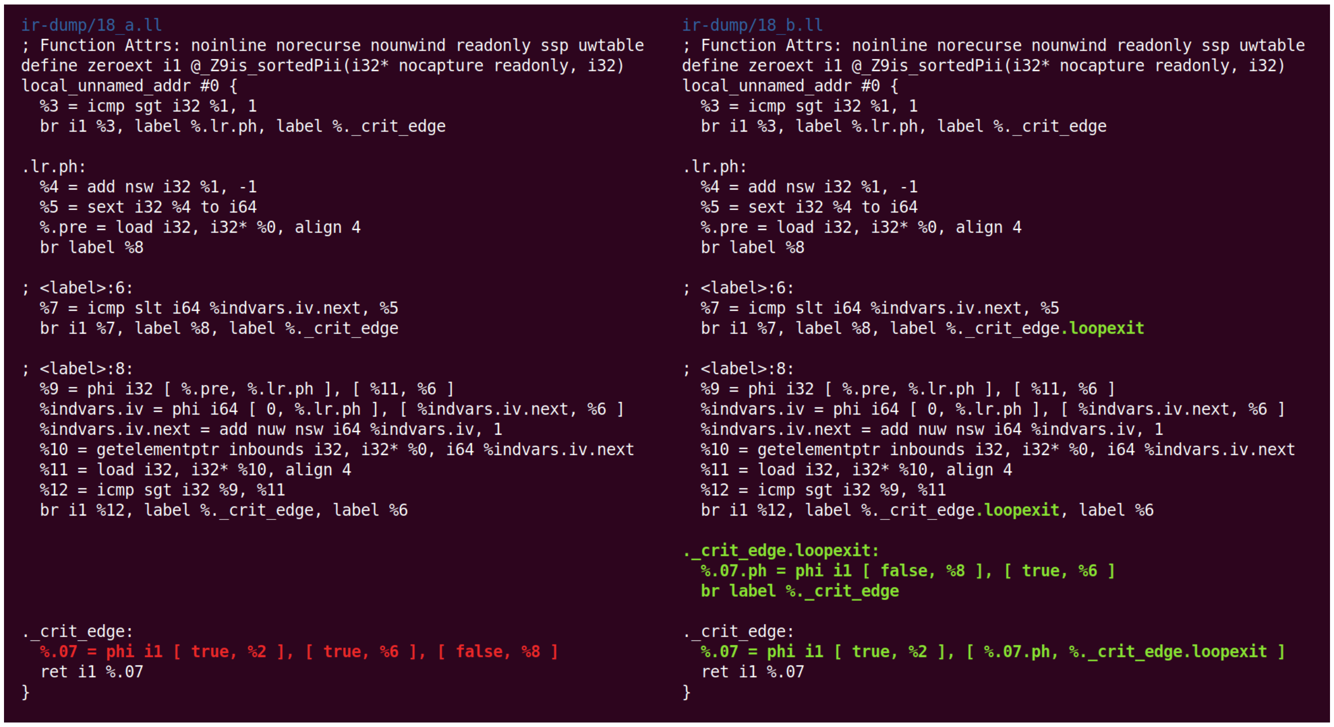
And CFG simplification flips it the other way:



```
ir-dump/17_a.ll
; Function Attrs: noinline norecurse nounwind readonly ssp uwtable
define zeroext i1 @_Z9is_sortedPii(i32* nocapture readonly, i32)
local_unnamed_addr #0 {
  %3 = icmp sgt i32 %1, 1
  br i1 %3, label %.lr.ph, label %._crit_edge

.lr.ph:
  %4 = add nsw i32 %1, -1
  %5 = sext i32 %4 to i64
  %.pre = load i32, i32* %0, align 4
  br label %8

; <label>:6:
  %7 = icmp slt i64 %indvars.iv.next, %5
  br i1 %7, label %8, label %._crit_edge.loopexit

; <label>:8:
  %9 = phi i32 [ %.pre, %.lr.ph ], [ %11, %6 ]
  %indvars.iv = phi i64 [ 0, %.lr.ph ], [ %indvars.iv.next, %6 ]
  %indvars.iv.next = add nuw nsw i64 %indvars.iv, 1
  %10 = getelementptr inbounds i32, i32* %0, i64 %indvars.iv.next
  %11 = load i32, i32* %10, align 4
  %12 = icmp sgt i32 %9, %11
  br i1 %12, label %._crit_edge.loopexit, label %6

._crit_edge.loopexit:
  %.07.ph = phi i1 [ true, %6 ], [ false, %8 ]
  br label %._crit_edge

._crit_edge:
  %.07 = phi i1 [ true, %2 ], [ %.07.ph, %._crit_edge.loopexit ]
  ret i1 %.07
}
```

```
ir-dump/17_b.ll
; Function Attrs: noinline norecurse nounwind readonly ssp uwtable
define zeroext i1 @_Z9is_sortedPii(i32* nocapture readonly, i32)
local_unnamed_addr #0 {
  %3 = icmp sgt i32 %1, 1
  br i1 %3, label %.lr.ph, label %._crit_edge

.lr.ph:
  %4 = add nsw i32 %1, -1
  %5 = sext i32 %4 to i64
  %.pre = load i32, i32* %0, align 4
  br label %8

; <label>:6:
  %7 = icmp slt i64 %indvars.iv.next, %5
  br i1 %7, label %8, label %._crit_edge

; <label>:8:
  %9 = phi i32 [ %.pre, %.lr.ph ], [ %11, %6 ]
  %indvars.iv = phi i64 [ 0, %.lr.ph ], [ %indvars.iv.next, %6 ]
  %indvars.iv.next = add nuw nsw i64 %indvars.iv, 1
  %10 = getelementptr inbounds i32, i32* %0, i64 %indvars.iv.next
  %11 = load i32, i32* %10, align 4
  %12 = icmp sgt i32 %9, %11
  br i1 %12, label %._crit_edge, label %6

._crit_edge:
  %.07 = phi i1 [ true, %2 ], [ true, %6 ], [ false, %8 ]
  ret i1 %.07
}
```

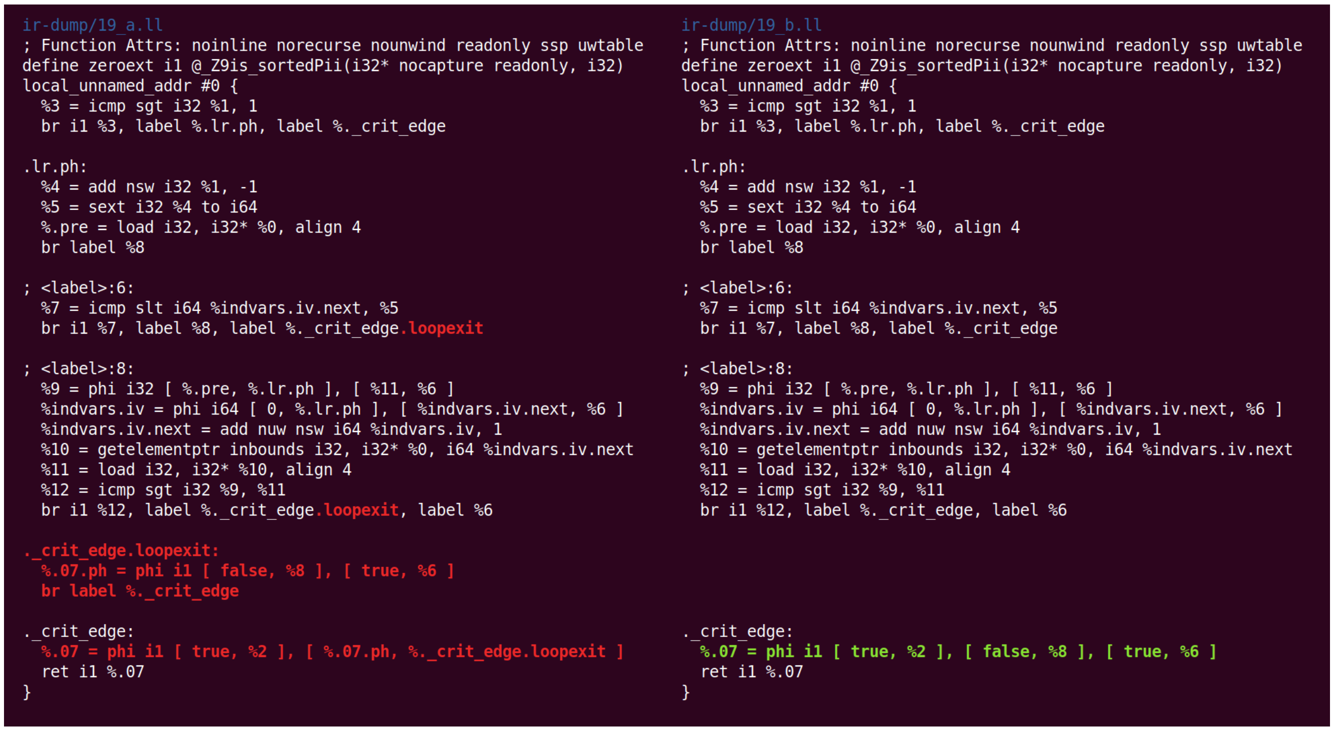
And back:



```
ir-dump/18_a.ll
; Function Attrs: noinline norecurse nounwind readonly ssp uwtable
define zeroext i1 @_Z9is_sortedPii(i32* nocapture readonly, i32)
local_unnamed_addr #0 {
  %3 = icmp sgt i32 %1, 1
  br i1 %3, label %.lr.ph, label %._crit_edge

.lr.ph:
  %4 = add nsw i32 %1, -1
  %5 = sext i32 %4 to i64
  %.pre = load i32, i32* %0, align 4
  br label %8

; <label>:6:
  %7 = icmp slt i64 %indvars.iv.next, %5
  br i1 %7, label %8, label %._crit_edge

; <label>:8:
  %9 = phi i32 [ %.pre, %.lr.ph ], [ %11, %6 ]
  %indvars.iv = phi i64 [ 0, %.lr.ph ], [ %indvars.iv.next, %6 ]
  %indvars.iv.next = add nuw nsw i64 %indvars.iv, 1
  %10 = getelementptr inbounds i32, i32* %0, i64 %indvars.iv.next
  %11 = load i32, i32* %10, align 4
  %12 = icmp sgt i32 %9, %11
  br i1 %12, label %._crit_edge, label %6

._crit_edge:
  %.07 = phi i1 [ true, %2 ], [ true, %6 ], [ false, %8 ]
  ret i1 %.07
}
```

```
ir-dump/18_b.ll
; Function Attrs: noinline norecurse nounwind readonly ssp uwtable
define zeroext i1 @_Z9is_sortedPii(i32* nocapture readonly, i32)
local_unnamed_addr #0 {
  %3 = icmp sgt i32 %1, 1
  br i1 %3, label %.lr.ph, label %._crit_edge

.lr.ph:
  %4 = add nsw i32 %1, -1
  %5 = sext i32 %4 to i64
  %.pre = load i32, i32* %0, align 4
  br label %8

; <label>:6:
  %7 = icmp slt i64 %indvars.iv.next, %5
  br i1 %7, label %8, label %._crit_edge.loopexit

; <label>:8:
  %9 = phi i32 [ %.pre, %.lr.ph ], [ %11, %6 ]
  %indvars.iv = phi i64 [ 0, %.lr.ph ], [ %indvars.iv.next, %6 ]
  %indvars.iv.next = add nuw nsw i64 %indvars.iv, 1
  %10 = getelementptr inbounds i32, i32* %0, i64 %indvars.iv.next
  %11 = load i32, i32* %10, align 4
  %12 = icmp sgt i32 %9, %11
  br i1 %12, label %._crit_edge.loopexit, label %6

._crit_edge.loopexit:
  %.07.ph = phi i1 [ false, %8 ], [ true, %6 ]
  br label %._crit_edge

._crit_edge:
  %.07 = phi i1 [ true, %2 ], [ %.07.ph, %._crit_edge.loopexit ]
  ret i1 %.07
}
```

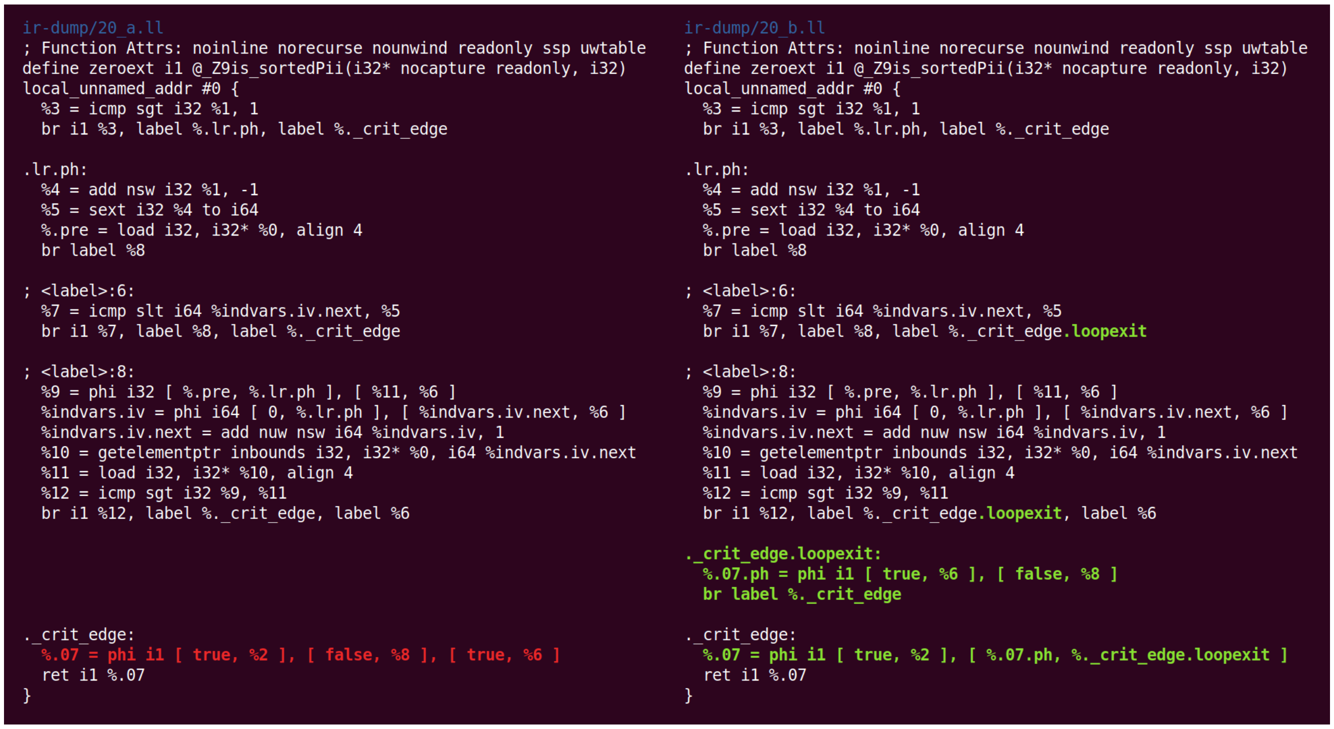
And forth:



```
ir-dump/19_a.ll
; Function Attrs: noinline norecurse nounwind readonly ssp uwtable
define zeroext i1 @_Z9is_sortedPii(i32* nocapture readonly, i32)
local_unnamed_addr #0 {
  %3 = icmp sgt i32 %1, 1
  br i1 %3, label %.lr.ph, label %._crit_edge

.lr.ph:
  %4 = add nsw i32 %1, -1
  %5 = sext i32 %4 to i64
  %.pre = load i32, i32* %0, align 4
  br label %8

; <label>:6:
  %7 = icmp slt i64 %indvars.iv.next, %5
  br i1 %7, label %8, label %._crit_edge.loopexit

; <label>:8:
  %9 = phi i32 [ %.pre, %.lr.ph ], [ %11, %6 ]
  %indvars.iv = phi i64 [ 0, %.lr.ph ], [ %indvars.iv.next, %6 ]
  %indvars.iv.next = add nuw nsw i64 %indvars.iv, 1
  %10 = getelementptr inbounds i32, i32* %0, i64 %indvars.iv.next
  %11 = load i32, i32* %10, align 4
  %12 = icmp sgt i32 %9, %11
  br i1 %12, label %._crit_edge.loopexit, label %6

._crit_edge.loopexit:
  %.07.ph = phi i1 [ false, %8 ], [ true, %6 ]
  br label %._crit_edge

._crit_edge:
  %.07 = phi i1 [ true, %2 ], [ %.07.ph, %._crit_edge.loopexit ]
  ret i1 %.07
}
```

```
ir-dump/19_b.ll
; Function Attrs: noinline norecurse nounwind readonly ssp uwtable
define zeroext i1 @_Z9is_sortedPii(i32* nocapture readonly, i32)
local_unnamed_addr #0 {
  %3 = icmp sgt i32 %1, 1
  br i1 %3, label %.lr.ph, label %._crit_edge

.lr.ph:
  %4 = add nsw i32 %1, -1
  %5 = sext i32 %4 to i64
  %.pre = load i32, i32* %0, align 4
  br label %8

; <label>:6:
  %7 = icmp slt i64 %indvars.iv.next, %5
  br i1 %7, label %8, label %._crit_edge

; <label>:8:
  %9 = phi i32 [ %.pre, %.lr.ph ], [ %11, %6 ]
  %indvars.iv = phi i64 [ 0, %.lr.ph ], [ %indvars.iv.next, %6 ]
  %indvars.iv.next = add nuw nsw i64 %indvars.iv, 1
  %10 = getelementptr inbounds i32, i32* %0, i64 %indvars.iv.next
  %11 = load i32, i32* %10, align 4
  %12 = icmp sgt i32 %9, %11
  br i1 %12, label %._crit_edge, label %6

._crit_edge:
  %.07 = phi i1 [ true, %2 ], [ false, %8 ], [ true, %6 ]
  ret i1 %.07
}
```

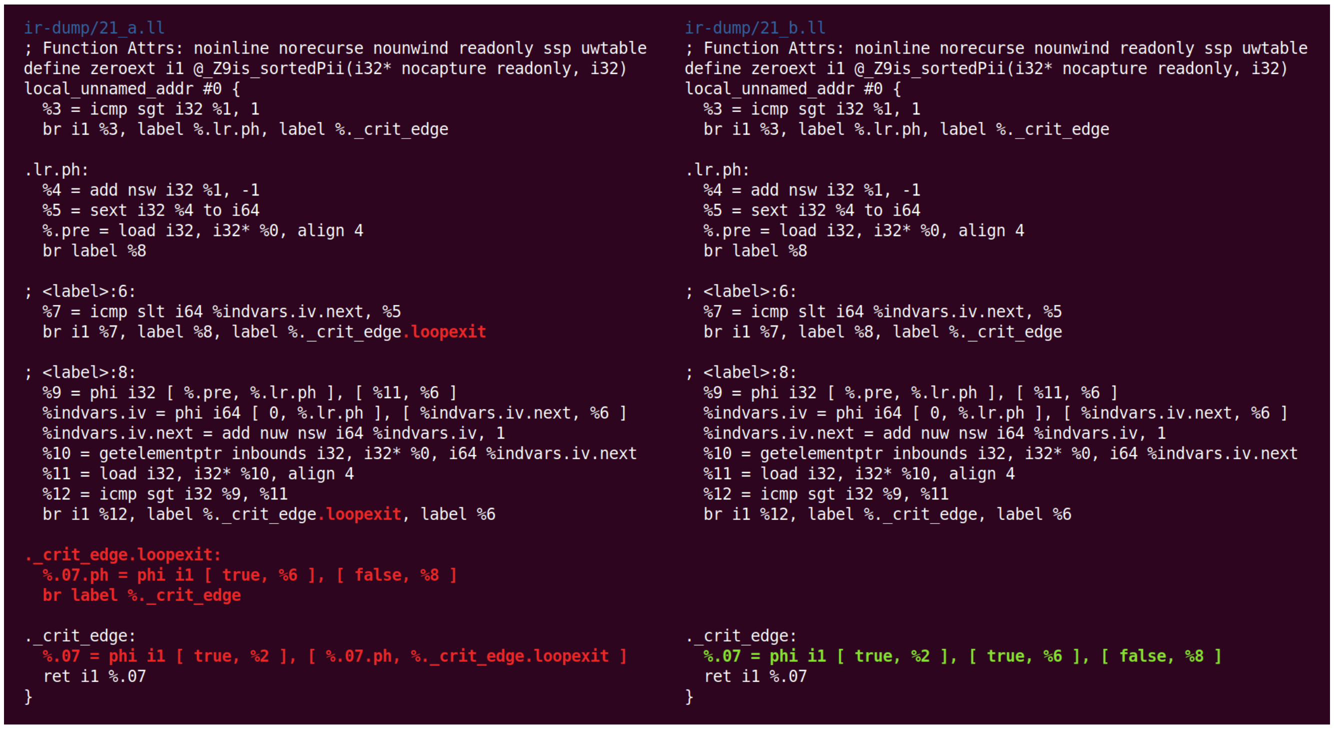
And back:



```
ir-dump/20_a.ll
; Function Attrs: noinline norecurse nounwind readonly ssp uwtable
define zeroext i1 @_Z9is_sortedPii(i32* nocapture readonly, i32)
local_unnamed_addr #0 {
  %3 = icmp sgt i32 %1, 1
  br i1 %3, label %.lr.ph, label %._crit_edge

.lr.ph:
  %4 = add nsw i32 %1, -1
  %5 = sext i32 %4 to i64
  %.pre = load i32, i32* %0, align 4
  br label %8

; <label>:6:
  %7 = icmp slt i64 %indvars.iv.next, %5
  br i1 %7, label %8, label %._crit_edge

; <label>:8:
  %9 = phi i32 [ %.pre, %.lr.ph ], [ %11, %6 ]
  %indvars.iv = phi i64 [ 0, %.lr.ph ], [ %indvars.iv.next, %6 ]
  %indvars.iv.next = add nuw nsw i64 %indvars.iv, 1
  %10 = getelementptr inbounds i32, i32* %0, i64 %indvars.iv.next
  %11 = load i32, i32* %10, align 4
  %12 = icmp sgt i32 %9, %11
  br i1 %12, label %._crit_edge, label %6

._crit_edge:
  %.07 = phi i1 [ true, %2 ], [ false, %8 ], [ true, %6 ]
  ret i1 %.07
}
```

```
ir-dump/20_b.ll
; Function Attrs: noinline norecurse nounwind readonly ssp uwtable
define zeroext i1 @_Z9is_sortedPii(i32* nocapture readonly, i32)
local_unnamed_addr #0 {
  %3 = icmp sgt i32 %1, 1
  br i1 %3, label %.lr.ph, label %._crit_edge

.lr.ph:
  %4 = add nsw i32 %1, -1
  %5 = sext i32 %4 to i64
  %.pre = load i32, i32* %0, align 4
  br label %8

; <label>:6:
  %7 = icmp slt i64 %indvars.iv.next, %5
  br i1 %7, label %8, label %._crit_edge.loopexit

; <label>:8:
  %9 = phi i32 [ %.pre, %.lr.ph ], [ %11, %6 ]
  %indvars.iv = phi i64 [ 0, %.lr.ph ], [ %indvars.iv.next, %6 ]
  %indvars.iv.next = add nuw nsw i64 %indvars.iv, 1
  %10 = getelementptr inbounds i32, i32* %0, i64 %indvars.iv.next
  %11 = load i32, i32* %10, align 4
  %12 = icmp sgt i32 %9, %11
  br i1 %12, label %._crit_edge.loopexit, label %6

._crit_edge.loopexit:
  %.07.ph = phi i1 [ true, %6 ], [ false, %8 ]
  br label %._crit_edge

._crit_edge:
  %.07 = phi i1 [ true, %2 ], [ %.07.ph, %._crit_edge.loopexit ]
  ret i1 %.07
}
```

And forth:



```
ir-dump/21_a.ll
; Function Attrs: noinline norecurse nounwind readonly ssp uwtable
define zeroext i1 @_Z9is_sortedPii(i32* nocapture readonly, i32)
local_unnamed_addr #0 {
  %3 = icmp sgt i32 %1, 1
  br i1 %3, label %.lr.ph, label %._crit_edge

.lr.ph:
  %4 = add nsw i32 %1, -1
  %5 = sext i32 %4 to i64
  %.pre = load i32, i32* %0, align 4
  br label %8

; <label>:6:
  %7 = icmp slt i64 %indvars.iv.next, %5
  br i1 %7, label %8, label %._crit_edge.loopexit

; <label>:8:
  %9 = phi i32 [ %.pre, %.lr.ph ], [ %11, %6 ]
  %indvars.iv = phi i64 [ 0, %.lr.ph ], [ %indvars.iv.next, %6 ]
  %indvars.iv.next = add nuw nsw i64 %indvars.iv, 1
  %10 = getelementptr inbounds i32, i32* %0, i64 %indvars.iv.next
  %11 = load i32, i32* %10, align 4
  %12 = icmp sgt i32 %9, %11
  br i1 %12, label %._crit_edge.loopexit, label %6

._crit_edge.loopexit:
  %.07.ph = phi i1 [ true, %6 ], [ false, %8 ]
  br label %._crit_edge

._crit_edge:
  %.07 = phi i1 [ true, %2 ], [ %.07.ph, %._crit_edge.loopexit ]
  ret i1 %.07
}
```

```
ir-dump/21_b.ll
; Function Attrs: noinline norecurse nounwind readonly ssp uwtable
define zeroext i1 @_Z9is_sortedPii(i32* nocapture readonly, i32)
local_unnamed_addr #0 {
  %3 = icmp sgt i32 %1, 1
  br i1 %3, label %.lr.ph, label %._crit_edge

.lr.ph:
  %4 = add nsw i32 %1, -1
  %5 = sext i32 %4 to i64
  %.pre = load i32, i32* %0, align 4
  br label %8

; <label>:6:
  %7 = icmp slt i64 %indvars.iv.next, %5
  br i1 %7, label %8, label %._crit_edge

; <label>:8:
  %9 = phi i32 [ %.pre, %.lr.ph ], [ %11, %6 ]
  %indvars.iv = phi i64 [ 0, %.lr.ph ], [ %indvars.iv.next, %6 ]
  %indvars.iv.next = add nuw nsw i64 %indvars.iv, 1
  %10 = getelementptr inbounds i32, i32* %0, i64 %indvars.iv.next
  %11 = load i32, i32* %10, align 4
  %12 = icmp sgt i32 %9, %11
  br i1 %12, label %._crit_edge, label %6

._crit_edge:
  %.07 = phi i1 [ true, %2 ], [ true, %6 ], [ false, %8 ]
  ret i1 %.07
}
```

Whew, we're finally done with the middle end! The code at the right is what gets passed to (in this case) the x86-64 backend.

You might be wondering if the oscillating behavior towards the end of the pass pipeline is the result of a compiler bug, but keep in mind that this function is really, really simple and there were a whole bunch of passes mixed in with the flipping and flopping, but I didn't mention them because they didn't make any changes to the code. As far as the second half of the middle end optimization pipeline goes, we're basically looking at a degenerate execution here.

Acknowledgments: Some students in my Advanced Compilers class this fall provided feedback on a draft of this post (I also used this material as an assignment). I ran across the function used here in this excellent set of lecture notes about loop optimizations.