2023-08-01 · 5263 words · 43 minutes

#dark-arts · #assembly · #toolchains

A Gentle Introduction to LLVM IR

The other day, I saw this tweet. In it, Andrew Gallant argues that reaching for LLVM IR, instead of assembly, is a useful tool for someone working on performance. Unfortunately, learning material on LLVM is usually aimed at compiler engineers, not generalist working programmers.

Now, *I'm* a compiler engineer, so my answer is *of course* you should know your optimizer's IR. But I do think there's a legitimate reason to be able to read it, in the same way that being able to read assembly to understand what your processor is doing is a powerful tool. I wrote an introduction to assembly over a year ago (still have to finish the followups… 💀), which I recommend reading first.

Learning LLVM IR is similar, but it helps you understand what your *compiler* is doing to create highly optimized code. LLVM IR is very popular, and as such well-documented and reasonably well-specified, to the point that we can just treat it as a slightly weird programming language.

In this article, I want to dig into what LLVM IR *is* and how to read it.

What's LLVM IR?

"LLVM" is an umbrella name for a number of software components that can be used to build compilers. If you write performance-critical code, you've probably heard of it.

Its flagship product is Clang, a high-end C/C++/Objective-C compiler. Clang follows the orthodox compiler architecture: a frontend that parses source code into an AST and lowers it into an *intermediate representation*, an "IR"; an optimizer (or "middle-end") that transforms IR into better IR, and a backend that converts IR into machine code for a particular platform.

```
                      optimizer (opt)
                           ___
                          |   v
 .c file  -->  AST  -->  LLVM IR  -->  assembly
           ^         ^             ^
        parser    lowering    backend (llc)

\____________________/  \_____________________/
    Clang Frontend                LLVM
```

LLVM often also refers to just the optimizer and backend parts of Clang; this is can be thought of as a compiler for the "LLVM language" or "LLVM assembly". Clang, and other language frontends like Rust, essentially compile to LLVM IR, which LLVM then compiles to machine code.

LLVM IR is well documented and... *somewhat* stable, which makes it a very good compilation target, since language implementers can re-use the thousands of engineer hours poured into LLVM already. The source of truth for "what is LLVM IR?" is the LangRef.

LLVM IR is also binary format (sometimes called "bitcode"), although we will be working exclusively with its text format (which uses the `.ll` extension).

LLVM-targeting compilers will have debugging flags to make them emit IR instead of their final output. For Clang, this is e.g. `clang++ -S -emit-llvm foo.cc`, while for Rust this is `rustc --emit=llvm-ir foo.rs`. Godbolt will also respect these options and correctly display LLVM IR output.

Back to Basic Blocks

LLVM IR can be quite intimidating to read, since it contains much more ancillary information than an assembly dump. Consider this function:

```
pub fn square(x: i32) -> i32 {
  x * x
}
```

godbolt Rust

If you click on the "Godbolt" widget, it will take you to a Godbolt that lowers it to LLVM IR. Most of that code is just metadata, but it's really intimidating!

Starting from compiler output will have a steep difficulty curve, because we have to face the full complexity of LLVM IR. For Rust, this will likely mean encountering exception-handling, which is how panics are implemented, and function attributes that forward Rust's guarantees (e.g. non-null pointers) to LLVM.

Instead, we'll start by introducing the basic syntax of LLVM IR, and *then* we'll tackle reading compiler output.

A Trivial Function

The meat of LLVM IR is function definitions, introduced with a `define`. There is also `declare`, which has exactly the same purpose as a function without a body in C: it brings an external symbol into scope.

For example, the following function takes no arguments and returns immediately:

```
define void @do_nothing() {
  ret void
}
```

LLVM IR

The return type of the function (`void`) immediately follows the `define` keyword; the name of the function starts with an `@`, which introduces us to the concept of *sigils*: every user-defined symbol starts with a sigil, indicating what kind of symbol it is. `@` is used for global and functions: things you can take the address of (when used as a value, they are always `ptr` -typed).

The body of a function resembles assembly: a list of labels and instructions. Unlike ordinary assembly, however, there are significant restrictions on the structure of these instructions.

In this case, there is only one instruction: a `void` -typed return. Unlike most assembly languages, LLVM IR is strongly typed, and requires explicit type annotations almost everywhere.

Here is another trivial function.

```
define void @do_not_call() {
  unreachable
}
```

LLVM IR

This function will trigger undefined behavior upon being called: the `unreachable` instruction represents a codepath that the compiler can assume is never executed; this is unlike e.g. the unimplemented `ud2` instruction in x86, which is guaranteed to issue a fault.

This is an important distinction between LLVM IR and an assembly language: some operations are explicitly left undefined to leave room for potential optimizations. For example, LLVM can reason that, because `@do_not_call` immediately triggers undefined behavior, all calls to `@do_not_call` are also unreachable (and propagate unreachability from there).

Purely Scalar Code

Let's start with basic functions that only operate on integers. Consider the following function, that squares a 32-bit integer:

```
define i32 @square(i32 %x) {
  %1 = mul i32 %x, %x
  ret i32 %1
}
```

LLVM IR

Now our function takes arguments and has multiple instructions.

The argument is specified as `i32 %x`. Names a `%` sigil are sort of like local variables, but with some restrictions that make them more optimization-friendly; as we'll see later, they're not really "variable" at all. LLVM sometimes calls them *registers*; in a sense, LLVM IR is assembly for an abstract machine with an infinite number of registers. I'll be calling `%`-prefixed names "registers" throughout this article.

`i32` is a primitive integer types. All integer types in LLVM are of the form `iN`, for any `N` (even non-multiples of eight). There are no signed or unsigned types; instead, instructions that care about signedness will specify which semantic they use.

The first instruction is a `mul i32`, which multiples the two `i32` operands together, and returns a value; we assign this to the new register `%1`[1]. The next instruction returns this value.

The other arithmetic operations have the names you expect: `add`, `sub`, `and`, `or`, `xor`, `shl` (shift left). There are two division and remainder instructions, signed (`sdiv` , `srem`) and unsigned (`udiv` , `urem`). There two shift right instructions, again signed (`ashr`) and unsigned (`lshr`).

> Exercise for the reader: why are `/`, `%`, and `>>` the only operations with signed and unsigned versions?

We can also convert from one integer type to another using `trunc`, `zext`, and `sext`, which truncate, zero-extend, and sign-extend, respectively (`sext` and `zext` are another signed/unsigned pair). For example, if we wanted the `square` function to never overflow, we could write

```
define i64 @square(i32 %x) {
  %1 = sext i32 %x to i64
  %2 = mul i64 %1, %1
  ret i64 %2
}
```
LLVM IR

Here, we cast `%x` to `i64` by sign-extension (since we've decided we're squaring signed integers) and then square the result. `trunc` and `zext` both have the same syntax as `sext`.

"I'll Be Back"

Of course, interesting functions have *control flow*. Suppose we want a safe division function: division by zero is UB, so we need to handle it explicitly. Perhaps something like this:

```
fn safe_div(n: u64, d: u64) -> u64 {
  if d == 0 { return u64::MAX; }
  n / d
}
```
Rust

We could try doing this using `select`, LLVM's "ternary" operation.

```
define i64 @safe_div(i64 %n, i64 %d) {
  %1 = icmp eq i64 %d, 0
  %2 = udiv i64 %n, %d
  %3 = select i1 %1, i64 -1, i64 %2
  ret i64 %3
}
```
LLVM IR

However, this has a problem: division by zero is UB[2], and `select` is not short-circuiting: its semantics are closer to that of `cmov` in x86.

To compile this correctly, need to use the `br` instruction, which represents a general branch operation[3]. In C terms, a `br i1 %cond, label %a, label %b` is equivalent to `if (cond) goto a; else goto b;`.

This is how we might write that:

```
define i64 @safe_div(i64 %n, i64 %d) {
  %1 = icmp eq i64 %d, 0
  br i1 %1, label %iszero, label %nonzero

iszero:
  ret i64 -1

nonzero:
  %2 = udiv i64 %n, %d
  ret i64 %2
}
```
LLVM IR

Now our function has labels, which are used by the `br` instruction as jump targets.

In the first block, we do the `d == 0` check, implemented by an `icmp eq` instruction. This returns an `i1` (the type LLVM uses for booleans). We then pass the result into a `br` instruction, which jumps to the first label if it's zero, otherwise to the second if it isn't.

The second block is the early-return; it returns the "sentinel" value; the third block is self-explanatory.

Each of these blocks is a "basic block": a sequence of non-control flow operations, plus an instruction that moves control flow away from the block. These blocks form the control flow graph (CFG) of the function.

There are a few other "block terminator" instructions. The one-argument form of `br` takes a single label, and is a simple unconditional `goto`. There's also `switch`, which is similar to a C `switch`:

```
switch i32 %value, label %default [
  i32 0, label %if_zero
  i32 1, label %if_one,
  ; etc
]
```
LLVM IR

The type of the `switch` must be an integer type. Although you could represent this operation with a chain of `br`s, a separate `switch` instruction makes it easier for LLVM to generate jump tables.

`unreachable`, which we saw before, is a special terminator that does not trigger control flow per se, but which can terminate a block because reaching it is undefined behavior; it is equivalent to e.g. `std::unreachable()` in C++.

> **LLVM Deleted My Code!**
>
> The `unreachable` instruction provides a good example of why LLVM uses a basic block CFG: a naive dead code elimination (DCE) optimization pass can be implemented as follows:
>
> 1. Fill a set with every block that ends in `unreachable`.
> 2. For every block, if its terminator references a block in the unreachable set, delete that label from the terminator. For example, if we have `br i1 %c, label`

> `%a, label %b`, and the unreachable set contains `%a`, we can replace this with a `br label %b`.
> 3. If every outgoing edge from a block is deleted in (2), replace the terminator with `unreachable`.
> 4. Delete all blocks in the unreachable set.
> 5. Repeat from (1) as many times as desired.
>
> Intuitively, `unreachable`s bubble *upwards* in the CFG, dissolving parts of the CFG among them. Other passes can generate `unreachable`s to represent UB: interplay between this and DCE results in the "the compiler *will* delete your code" outcome from UB.
>
> The actual DCE pass is much more complicated, since function calls make it harder to decide if a block is "pure" and thus transparently deletable.

But, what if we want to implement something more complicated, like `a / b + 1`? This expression needs the intermediate result, so we can't use two return statements as before.

Working around this is not so straightforward: if we try to assign the same register in different blocks, the IR verifier will complain. This brings us to the concept of static single assignment.

Phony! Phony!

LLVM IR is a *static single assignment form* (SSA) IR. LLVM was actually started at the turn of the century to create a modern SSA optimizer as an academic project. These days, SSA is extremely fashionable for optimizing imperative code.

SSA form means that every register is assigned by at most one instruction per function. Different executions of the same block in the same function may produce different values for particular registers, but we cannot *mutate* already-assigned registers.

In other words:

1. Every register is guaranteed to be initialized by a single expression.
2. Every register depends only on the values of registers assigned before its definition.

This has many useful properties for writing optimizations: for example, within a basic block, every use of a particular register `%x` always refers to the same value, which makes optimizations like global value numbering and constant-folding much simpler to write, since the state of a register throughout a block doesn't need to be tracked separately.

In SSA, we reinterpret mutation as many *versions* of a single variable. Thus, we might lower `x += y` as

```
%x.1 = add i32 %x.0, %y.0                                    LLVM IR
```

Here, we've used a `var.n` convention to indicate which version of a variable a specific register represents (LLVM does not enforce any naming conventions).

However, when loops enter the mix, it's not clear how to manage versions. The number of registers in a function is static, but the number of loop iterations is dynamic.

Concretely, how do we implement this function?

```
fn pow(x: u32, y: u32) -> u32 {
  let mut r = 1;
  for i in 0..y {
```

```rust
      r *= x;
    }
    r
  }
```
Rust

We could try something like this:

```llvm
define i32 @pow(i32 %x, i32 %y) {
  br label %loop

loop:
  %r = add i32 %r, %x   ; ERROR: Recursive definition.
  %i = add i32 %i, 1    ; ERROR: Recursive definition.
  %done = icmp eq i32 %i, %y
  br i1 %done, label %exit, label %loop

exit:
  ret i32 %r
}
```
LLVM IR

But there's a problem! What are the original definitions of `%r` and `%i`? The IR verifier will complain that these registers depend directly on themselves, which violates SSA form. What's the "right" way to implement this function?

One option is to ask LLVM! We'll implement the function poorly, and let the optimizer clean it up for us.

First, let's write the function using memory operations, like `load`s and `store`s, to implement mutation. We can use the `alloca` instruction to create statically-sized stack slots; these instructions return a `ptr` [^clang-codegen].

> **Clang Makes a Mess, LLVM Cleans It Up**
>
> Incidentally, this is how Clang and Rust both generate LLVM IR: stack variables are turned into `alloca`s and manipulated through loads and stores; temporaries are mostly turned into `%regs`s, but the compiler will sometimes emit extra allocas to avoid thinking too hard about needing to create `phi` instructions.

> This is pretty convenient, because it avoids needing to think very hard about SSA form outside of LLVM, and LLVM can trivially eliminate unnecessary allocas. The code I wrote for the codegen of `@pow` is very similar to what Rust would send to LLVM (although because we used an iterator, there's a lot of extra junk Rust emits that LLVM has to work to eliminate).

```llvm
define i32 @pow(i32 %x, i32 %y) {
  ; Create slots for r and the index, and initialize them.
  ; This is equivalent to something like
  ;   int i = 0, r = 1;
  ; in C.
  %r = alloca i32
  %i = alloca i32
  store i32 1, ptr %r
  store i32 0, ptr %i
  br label %loop_start

loop_start:
  ; Load the index and check if it equals y.
  %i.check = load i32, ptr %i
  %done = icmp eq i32 %i.check, %y
  br i1 %done, label %exit, label %loop

loop:
  ; r *= x
  %r.old = load i32, ptr %r
  %r.new = mul i32 %r.old, %x
  store i32 %r.new, ptr %r

  ; i += 1
  %i.old = load i32, ptr %i
  %i.new = add i32 %i.old, 1
  store i32 %i.new, ptr %i

  br label %loop_start

exit:
  %r.ret = load i32, ptr %r
  ret i32 %r.ret
}
```
LLVM IR

Next, we can pass this into the LLVM optimizer. The command `opt`, which is part of the LLVM distribution, runs specific optimizer passes on the IR. In our case, we want `opt -p mem2reg`, which runs a single "memory to register" pass. We can also just run `opt --O2` or similar to get similar[4] optimizations to the ones `clang -O2` runs.

This is the result.

```llvm
; After running through `opt -p mem2reg`
define i32 @pow(i32 %x, i32 %y) {
start:
  br label %loop_start

loop_start:
  %i.0 = phi i32 [0, %start], [%i.new, %loop]
  %r.0 = phi i32 [1, %start], [%r.new, %loop]
  %done = icmp eq i32 %i.0, %y
  br i1 %done, label %exit, label %loop

loop:
  %r.new = mul i32 %r.0, %x
  %i.new = add i32 %i.0, 1
  br label %loop_start

exit:
  ret i32 %r.0
}
```

LLVM IR

The `alloca`s are gone, but now we're faced with a new instruction: `phi`. "φ node" is jargon from the original SSA paper; the greek letter φ means "phoney". These instructions select a value from a list based on which basic block we jumped to the block from.

For example, `phi i32 [0, %start], [%i.new, %loop]` says "this value should be 0 if we came from the `start` block; otherwise `%i.new` if it came from `%loop`".

Unlike all other instructions, `phi` can refer to values that are not defined in all blocks that dominate the current block. This lets us have a dynamic number of versions of a variable! Here's what that looks like in a dynamic execution context.

> A block `%a` is said to dominate a block `%b` if each of its predecessors is either `%a` or a block dominated by `%a`. In

> other words, every path from the first block to `%b` passes through `%a`. In general instructions can only refer to values defined in previous instructions in the current block or values from blocks that dominate it.

1. `%start` directly jumps into `%loop_start`. The first block cannot be a jump target, since it cannot have `phi` nodes because its predecessors include function's callsite.

2. In `%loop_start`, since we've entered from `%start`, `%i.0` and `%r.0` are selected to be the first versions of the (platonic) `i` and `r` variables, i.e., their initial values; we jump to `%loop`.

3. Then, `%loop` is dominated by `%loop_start` so we can use `%i.0` and `%r.0` there directly; these are the `*=` and `+=` operations. Then we jump back to `%loop_start`.

4. Back in `%loop_start`, the `phi`s now select `%i.new` and `%r.new`, so now `%i.0` and `%r.0` are the *second* versions of `i` and `r`. By induction, the nth execution of `%loop_start` has the nth versions of `i` and `r`.

5. When we finally get sent to `%exit`, we can use `%r.0` (since `%loop_start` dominates `%r.0`), which will be the `%y`th version of `r`; this is our return value.

This is a good place to stop and think about what we've done so far. SSA, domination, and `phi`s can be hard to wrap your head around, and are not absolutely necessary for reading most IR. However, it is absolutely worth trying to understand, because it captures essential facts about how compilers like to reason about code[5].

With `phi` and `br`, we can build arbitrarily complicated control flow within a function[6].

Types and Aggregates

Now that we have basic scalar functions, let's review LLVM's type system.

We've seen `i32` and its friends; these are arbitrary-bit-with integers. `i1` is special because it is used as the boolean type. LLVM optimizations have been known to generate integer types with non-power-of-two sizes.

LLVM also has `float` and `double`, and some exotic float types like `bfloat`; these use their own arithmetic instructions with different options. I'll pass on them in this explainer; see `fadd` and friends in the LangRef for more.

We've also seen `void`, which is only used as a return value, and `ptr`, which is an untyped[7] pointer.

We've also seen the `label` pseudo-type, which represents a block label. It does not appear directly at runtime and has limited uses; the `token` and `metadata` types are similar.

Arrays are spelled `[n x T]`; the number must be an integer and the type must have a definite size. E.g., `[1024 x i8]`. Zero-sized arrays are supported.

Structs are spelled `{T1, T2, ...}`. E.g., `{i64, ptr}` is a Rust slice. Struct fields do not have names and are indexed, instead. The form `<{...}>` is a *packed* struct, which removes inter-field padding. E.g. `#[repr(packed)]` compiles down to this.

Vectors are like arrays but spelled `<n x T>`. These are used to represent types used in SIMD operations. For example, adding two `<4 x i32>` would lower to an AVX2 vector add on x86. I will not touch on SIMD stuff beyond this, although at higher optimization levels LLVM will merge scalar operations into vector operations, so you may come across them.

Type aliases can be created at file scope with the syntax

```llvm
%Slice = type {i64, ptr}
```
LLVM IR

This means that `%T` can be either a type or a register/label inside of a function, depending on syntactic position.

Operations on Aggregates

The `insertvalue` and `extractvalue` can be used with struct or array types to statically access a field. For example,

```llvm
%MyStruct = type {i32, {[5 x i8], i64}}

; In Rust-like syntax, this is `let v = s.1.0[4];`
%v = extractvalue %MyStruct %s, 1, 0, 4
```
LLVM IR

`insertvalue` is the reverse: it produces a copy of the aggregate with a specific field changed. It *does not* mutate in-place, because SSA forbids that.

```llvm
; In Rust-like syntax, this is
;   let s2 = { let mut s = s; s2.1.1 = 42; s };
%s2 = insertvalue %MyStruct %s, i64 42, 1, 1
```
LLVM IR

There are similar operations called `insertelement` and `extractelement` work on vectors, but have slightly different syntax and semantics.

Finally, there's `getelementptr`, the "pointer arithmetic instruction", often abbreviated to GEP. A GEP can be used to calculate an offset pointer into a struct. For example,

```llvm
define ptr @get_inner_in_array(ptr %p, i64 %idx) {
  %q = getelementptr %MyStruct, ptr %p, i64 %idx, i32 1, i32 1
  ret ptr %q
}
```
LLVM IR

This function takes in a pointer, ostensibly pointing to an array of `%MyStruct` s, and an index. This returns a pointer to the `i64` field of the `%idx` th element of `%p` .

A few important differences between GEP and `extractvalue` :

1. It takes an untyped pointer instead of a value of a particular struct/array type.
2. There is an extra parameter that specifies an index; from the perspective of GEP, every pointer is a pointer to an array of unspecified bound. When operating on a pointer that does not (at runtime) point to an array, an index operand of `0` is still required. (Alternatively, you can view a pointer to `T` as being a pointer to a one-element array.)
3. The index parameters need explicit types.

LLVM provides a helpful[8] FAQ on the GEP instruction: https://llvm.org/docs/GetElementPtr.html.

Other Operations

Some other operations are very relevant for reading IR, but don't fit into any specific category. As always, the LangRef provides a full description of what all of these instructions do.

Function Calls

`call` , which calls any `ptr` as a function. For example:

```
; Arguments are passed in parentheses.
%r = call i32 @my_func(i32 %x)
```
LLVM IR

Note that this could have been a `%reg` instead of a `@global` , which indicates a function pointer call.

Sometimes you will see `invoke`, which is used to implement "call a function inside of a C++ `try {}` block". This is rare in Rust, but can occur in some C++ code.

Function calls are often noisy areas of IR, because they will be very heavily annotated.

Synchronization

The `load` and `store` instructions we've already seen can be annotated as `atomic`, which is used to implement e.g. `AtomicU32::load` in Rust; this requires that an atomic ordering be specified, too. E.g.,

```
%v = load atomic i32, ptr %p acquire
```
LLVM IR

The `fence` operation is a general memory fence operation corresponding to e.g. Rust's `std::sync::atomic::fence` function.

`cmpxchg` provides the CAS (compare-and-swap) primitive. It returns a `{T, i1}` containing the old value and whether the CAS succeeded. `cmpxchg weak` implements the spuriously-failing "weak CAS" primitive.

Finally, `atomicrmw` performs a read-modify-write (e.g., `*p = op(*p, val)`) atomically. This is used to implement things like `AtomicU32::fetch_add` and friends.

All of these operations, except for `fence`, can also be marked as `volatile`. In LLVM IR, much like in Rust but unlike in C/C++, individual loads and stores are volatile (i.e., have compiler-invisible side-effects). `volatile` *can* be combined with atomic operations (e.g. `load atomic volatile`), although most languages don't provide access to these (except older C++ versions).

Reinterpret Shenanigans

`bitcast` is what `mem::transmute` and `reinterpret_cast` in Rust and C++, respectively, ultimately compile into. It can convert any non-aggregate type (integers, vectors) to any other type of the same bit width. For example, it can be used to get at the bits of a floating-point value:

```llvm
%bits = bitcast double %fp to i64
```
LLVM IR

It also used to be what was used to cast pointer types (e.g. `i32*` to `i8*`). Pointers are now all untyped (`ptr`) so this use is no longer present.

However, `bitcast` cannot cast between pointer and integer data. For this we must use the `inttoptr` and `ptrtoint`[9] instructions. These have the same syntax, but interact with the sketchy semantics of pointer-to-integer conversion and pointer provenance. This part of LLVM's semantics is a bit of an ongoing trashfire; see Ralf Jung's post for an introduction to this problem.

Intrinsics

There is also a vast collection of LLVM intrinsics, which are specified in the LangRef. For example, if we need a particular built-in memcpy, we can bring it into scope with a `declare`:

```llvm
; ptr %dst, ptr %src, i64 %len, i1 %volatile
declare void @llvm.memcpy.p0.p0.i64(ptr, ptr, i64, i1)
```
LLVM IR

All of the LLVM intrinsics are functions that start with `llvm.`; diving into all of them is far beyond what we can do here.

I'm also leaving out discussion of floating point, SIMD, and exception handling, each of which would require their own articles!

Undefined Behavior

LLVM exists to generate optimized code, and optimizations require that we declare certain machine states "impossible", so that we can detect when we can simplify what the programmer has said. This is "undefined behavior".

For example, we've already encountered `unreachable`, which LLVM assumes cannot be executed. Division by zero and accessing memory out of bounds is also undefined.

Most LLVM UB factors through the concept of "poisoned values". A poison value can be thought of as "taking on every value at once", whichever is convenient for the current optimization pass with no respect to any other passes. This also means that if optimizations *don't* detect a use of poison, it is ok from LLVM's perspective to give you a garbage value. This is most visible at `-O0`, which performs minimal optimization.

Using a poison value as a pointer in a `load`, `store`, or `call` must be UB, because LLVM can choose it to be a null pointer. It also can't be the denominator of a `udiv` or similar, because LLVM can choose it to be zero, which is UB. Passing poison into a `br` or a `switch` is also defined to be UB.

LLVM can perform dataflow analysis to try to determine what operations a poisonous value that was used in a UB way came from, and thus assume those operations cannot produce poison. Because all operations (other than `select` and `phi`) with a poison input produce poison, backwards reasoning allows LLVM to propagate UB forward. This is where so-called "time traveling UB" comes from.

Many operations generate poison. For example, in C, signed overflow is UB, so addition lowers to an `add nsw` (`nsw` stands for no signed wrap). Instead of wrapping on overflow, the instruction produces poison. There is also an unsigned version of the annotation, `nuw`.

Many other operations have "less defined" versions, which are either generated by optimizations, or inserted directly by the compiler that invokes LLVM when the language rules allow it (see C above). More examples include:

- `udiv` and friends have an `exact` annotation, which requires that the division have a zero remainder, else poison.
- `getelementptr` has an `inbounds` annotation, which produces poison if the access is actually out of bounds. This changes it from a pure arithmetic operation to one more closely matching C's pointer arithmetic restrictions. GEP without `inbounds` corresponds to Rust's `<*mut T>::wrapping_offset()` function.
- Floating point operations marked with `nnan` and `ninf` will produce poison instead of a NaN or an infinite value, respectively (or when a NaN or infinity is an argument).

Creating poison is *not* UB; only using it is. This is weaker than the way UB works in most languages; in C, overflow is instantly UB, but in LLVM overflow that is never "witnessed" is simply ignored. This is a simpler operational semantics for reasoning about the validity of optimizations: UB must often be viewed as a side-effect, because the compiler will generate code that puts the program into a broken state. For example, division by zero will cause a fault in many architectures. This means UB-causing operations cannot always be reordered soundly. Replacing "causes UB" with "produces poison" ensures the vast majority of operations are pure and freely reorderable.

Let's go back to our original Rust example!

```
pub fn square(x: i32) -> i32 {
  x * x
}
```

godbolt Rust

This is the output, with metadata redacted and some things moved around for readability.

```
source_filename = "example.b6eb2c7a6b40b4d2-cgu.0"
target datalayout = "e-m:e-p270:32:32-p271:32:32-p272:64:64-i64:64-f80:12
target triple = "x86_64-unknown-linux-gnu"

; example::square
define i32 @_ZN7example6square17hb32bcde4463f37c3E(i32 %x) unnamed_addr #
start:
  %0 = call { i32, i1 } @llvm.smul.with.overflow.i32(i32 %x, i32 %x)
  %_2.0 = extractvalue { i32, i1 } %0, 0
  %_2.1 = extractvalue { i32, i1 } %0, 1
  %1 = call i1 @llvm.expect.i1(i1 %_2.1, i1 false)
  br i1 %1, label %panic, label %bb1

bb1:
  ret i32 %_2.0

panic:
  ; core::panicking::panic
  call void @_ZN4core9panicking5panic17ha338a74a5d65bf6fE(
    ptr align 1 @str.0,
    i64 33,
    ptr align 8 @alloc_1368addac7d22933d93af2809439e507
  )
  unreachable
}

declare { i32, i1 } @llvm.smul.with.overflow.i32(i32, i32) #1
declare i1 @llvm.expect.i1(i1, i1) #2

; core::panicking::panic
declare void @_ZN4core9panicking5panic17ha338a74a5d65bf6fE(ptr align 1, i

@alloc_9be5c135c0f7c91e35e471f025924b11 = private unnamed_addr constant
  <{ [15 x i8] }>
  <{ [15 x i8] c"/app/example.rs" }>, align 1

@alloc_1368addac7d22933d93af2809439e507 = private unnamed_addr constant
  <{ ptr, [16 x i8] }> <{
    ptr @alloc_9be5c135c0f7c91e35e471f025924b11,
    [16 x i8] c"\0F\00\00\00\00\00\00\00\02\00\00\00\03\00\00\00"
  }>, align 8

@str.0 = internal constant [33 x i8] c"attempt to multiply with overflow"
```

```
attributes #0 = { nonlazybind uwtable }
attributes #1 = { nocallback nofree nosync nounwind speculatable willretu
attributes #2 = { nocallback nofree nosync nounwind willreturn memory(non
attributes #3 = { cold noinline noreturn nonlazybind uwtable }
```
LLVM IR

The main function is `@_ZN7example6square17hb32bcde4463f37c3E`, which is the mangled name of `example::square`. Because this code was compiled in debug mode, overflow panics, so we need to generate code for that. The first operation is a `call` to the LLVM intrinsic for "multiply and tell us if it overflowed". This returns the equivalent of a `(i32, bool)`; we extract both value out of it with `extractvalue`. We then pass the bool through `@llvm.expect`, which is used to tell the optimizer to treat the panicking branch as "cold". The success branch goes to a return, which returns the product; otherwise, we go to a function that calls `core::panicking::panic()` to panic the current thread. This function never returns, so we can terminate the block with an `unreachable`.

The rest of the file consists of:

- `declare`s for the llvm intrinsics we used.
- A `declare` for `core::panicking::panic`. Any external function we call needs to be `declare`d. This also gives us a place to hang attributes for the function off of.
- Global constants for a `core::panic::Location` and a panic message.
- Attributes for the functions above.

This is a good place to mention attributes: LLVM has all kinds of attributes that can be placed on functions (and function calls) to record optimization-relevant information. For example, `@llvm.expect.i1` is annotated as `willreturn`, which means this function will eventually return; this means that, for example, any UB that comes after the function is guaranteed to occur after finite time, so LLVM can conclude that the code is unreachable despite the call to `@llvm.expect.i1`. The full set of attributes is vast, but the LangRef documents all of them!

Conclusion

LLVM IR is huge, bigger than any individual ISA, because it is intended to capture *every* interesting operation. It also has a rich annotation language, so passes can record information for future passes to make use of. Its operational semantics attempt to leave enough space for optimizations to occur, while ensuring that multiple sound optimizations in sequence are not unsound (this last part is a work in progress).

Being able to read assembly reveals what will happen, exactly, when code is executed, but reading IR, before and after optimization, shows how the compiler is *thinking* about your code. Using `opt` to run individual optimization passes can also help further this understanding (in fact, "bisecting on passes" is a powerful debugging technique in compiler development).

I got into compilers by reading LLVM IR. Hopefully this article inspires you to learn more, too! ■

---

(1) Registers within a function may have a numeric name. They must be defined in order: you must define `%0` (either as a register or a label), then `%1`, then `%2`, etc. These are often used to represent "temporary results".

If a function does not specify names for its parameters, they will be given the names `%0`, `%1`, etc implicitly, which affect what the first explicit numeric register name you can use is. Similarly, if the function does not start with a label, it will be implicitly be given the next numeric name.

This can result in significant confusion, because if we have `define void @foo(i32, i32) { ... }`, the arguments will be `%0` and `%1`, but if we tried to write `%2 = add i32 %0, %1`, we would get an extremely confusing parser error, because `%2` is already taken as the name of the first block. ↺

---

(2) For some reason, the optimizer can't figure out that the `select` is redundant? Alive2 (an SMT-solver correctness checker for optimizations) seems to agree this is a valid optimization.

So I've filed a bug. :D ↺

---

(3) If you read my assembly article, you'll recall that there are many branch instructions. On RV, we have `beq`, `bne`, `bgt`, and `bge`. Later on in the compilation process, after the optimizer runs, LLVM will perform *instruction selection* (isel) to choose the best machine instruction(s) to implement a particular LLVM instruction (or sequence), which is highly context-dependent: for example, we want to fuse an `icmp eq` followed by a `br` on the result into a `beq`.

Isel is far outside my wheelhouse, and doing it efficiently and profitably is an active area of academic research. ↺

---

(4) Not exactly the same: language frontends like Clang and Rust will perform their own optimizations. For example, I have an open bug for LLVM being unable to convert `&&` into `&` in some cases; this was never noticed, because Clang performs this optimization while lowering from C/C++ to LLVM, but Rust does not do the equivalent optimization. ↺

---

(5) A more intuitive model is used in more modern IRs, like MLIR. In MLIR, you cannot use variables defined in other blocks;

instead, each block takes a set of *arguments*, just like a function call. This is equivalent to `phi` instructions, except that now instead of selecting which value we want in the target, each predecessor specifies what it wants to send to the target.

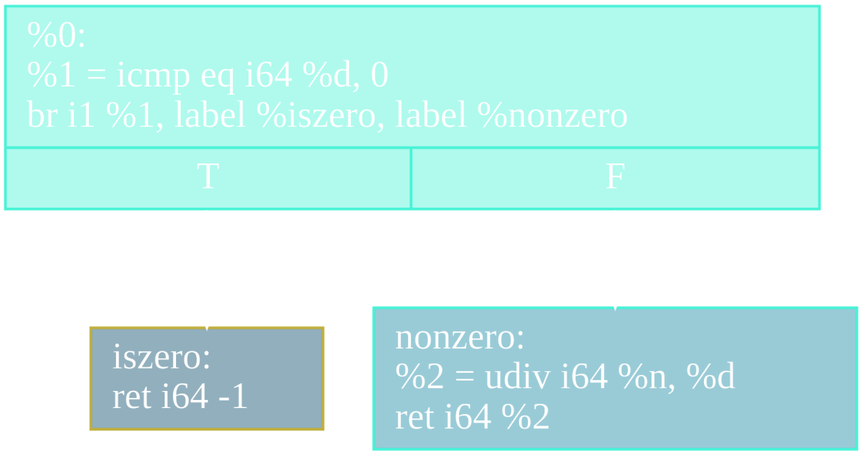
If we instead treat each block as having "arguments", we can rewrite it in the following fantasy syntax where register names are scoped to their block.

```
;; Not actual LLVM IR! ;;

define i32 @pow(i32 %x, i32 %y) {
  br %loop_start(i32 0, i32 1)

loop_start(i32 %i, i32 %r)
  %done = icmp eq i32 %i.0, %y
  br i1 %done, %exit(i32 %r), %loop(i32 %i, i32 %r)

loop(i32 %i, i32 %r)
  %r.new = mul i32 %r, %x
  %i.new = add i32 %i, 1
  br %loop_start(i32 %i, i32 %r)

exit(i32 %r)
  ret i32 %r
}
```
LLVM IR

↺

---

(6) What does the CFG look like? LLVM contains "optimization" passes that print the CFG as a file as a `.dot` file, which can be rendered with the `dot` command. For `@safe_div`, we get something like the following.



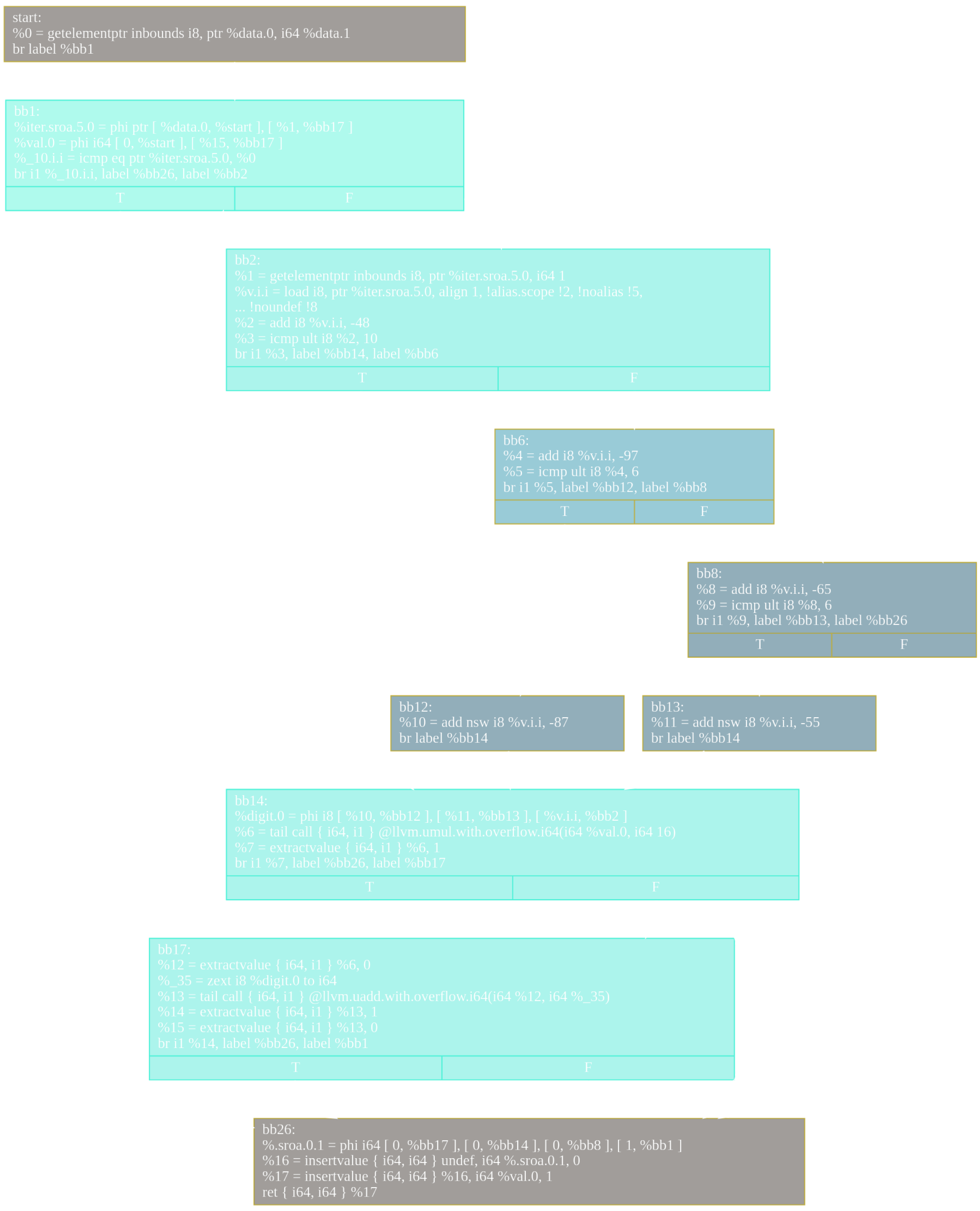
This is useful for understanding complex functions. Consider this Rust hex-parsing function.

```
// hex.rs
#[no_mangle]
fn parse_hex(data: &str) -> Option<u64> {
  let mut val = 0u64;
  for c in data.bytes() {
    let digit = match c {
      b'0'..=b'9' => c,
      b'a'..=b'f' => c - b'a' + 10,
      b'A'..=b'F' => c - b'A' + 10,
      _ => return None,
    };

    val = val.checked_mul(16)?;
    val = val.checked_add(digit as u64)?;
  }
  Some(val)
}
```
Rust

Then, we can generate our CFG with some shell commands.

```
$ rustc -O --crate-type lib --emit llvm-ir hex.rs
$ opt -p dot-cfg -o /dev/null hex.ll
Writing '.parse_hex.dot'...
$ dot -Tsvg .parse_hex.dot -o parse_hex.svg
```
Shell

The result is this mess.



```
start:
%0 = getelementptr inbounds i8, ptr %data.0, i64 %data.1
br label %bb1
```

```
bb1:
%iter.sroa.5.0 = phi ptr [ %data.0, %start ], [ %1, %bb17 ]
%val.0 = phi i64 [ 0, %start ], [ %15, %bb17 ]
%_10.i.i = icmp eq ptr %iter.sroa.5.0, %0
br i1 %_10.i.i, label %bb26, label %bb2
```

```
bb2:
%1 = getelementptr inbounds i8, ptr %iter.sroa.5.0, i64 1
%v.i.i = load i8, ptr %iter.sroa.5.0, align 1, !alias.scope !2, !noalias !5,
... !noundef !8
%2 = add i8 %v.i.i, -48
%3 = icmp ult i8 %2, 10
br i1 %3, label %bb14, label %bb6
```

```
bb6:
%4 = add i8 %v.i.i, -97
%5 = icmp ult i8 %4, 6
br i1 %5, label %bb12, label %bb8
```

```
bb8:
%8 = add i8 %v.i.i, -65
%9 = icmp ult i8 %8, 6
br i1 %9, label %bb13, label %bb26
```

```
bb12:
%10 = add nsw i8 %v.i.i, -87
br label %bb14
```

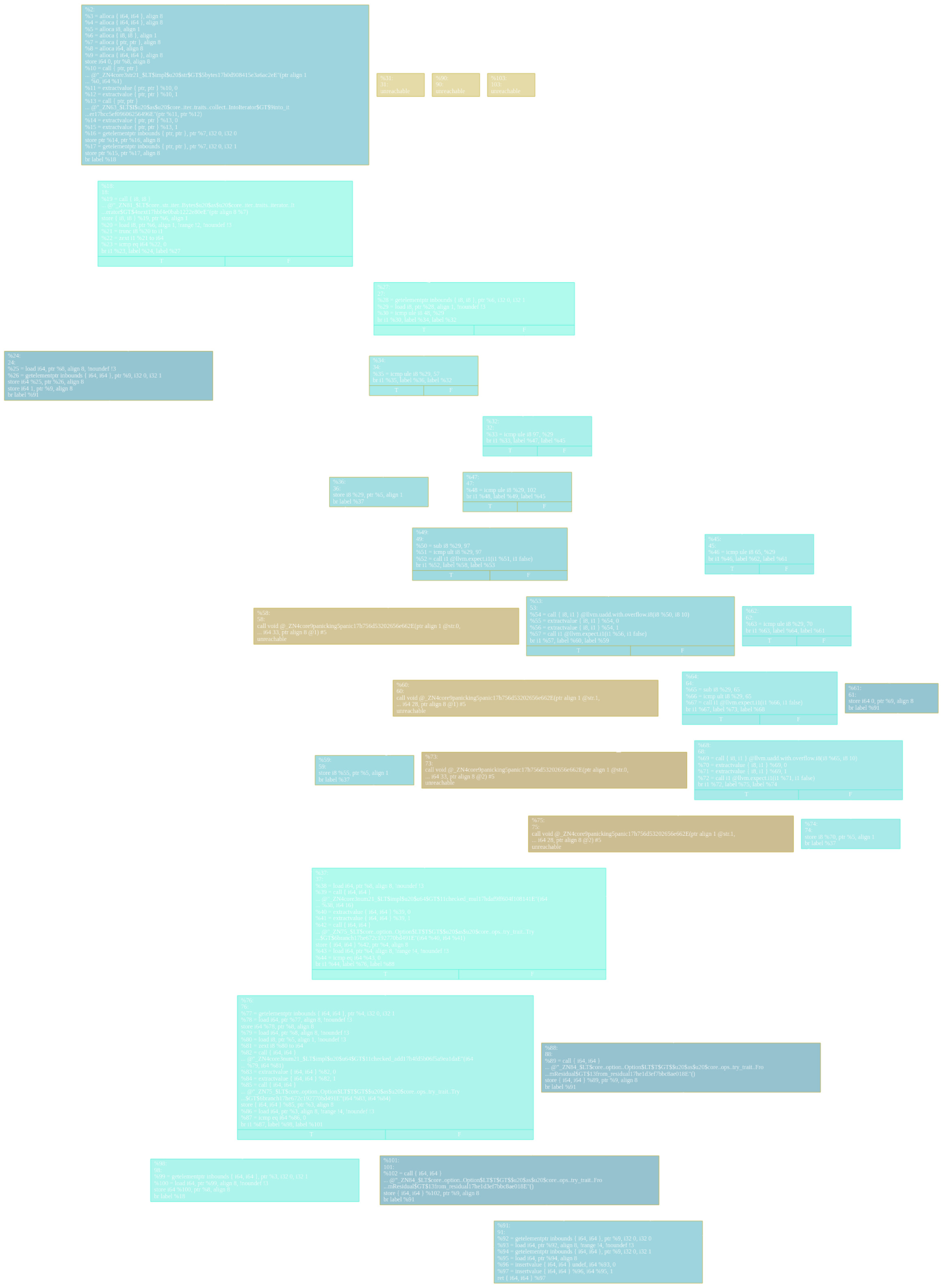
```
bb13:
%11 = add nsw i8 %v.i.i, -55
br label %bb14
```

```
bb14:
%digit.0 = phi i8 [ %10, %bb12 ], [ %11, %bb13 ], [ %v.i.i, %bb2 ]
%6 = tail call { i64, i1 } @llvm.umul.with.overflow.i64(i64 %val.0, i64 16)
%7 = extractvalue { i64, i1 } %6, 1
br i1 %7, label %bb26, label %bb17
```

```
bb17:
%12 = extractvalue { i64, i1 } %6, 0
%_35 = zext i8 %digit.0 to i64
%13 = tail call { i64, i1 } @llvm.uadd.with.overflow.i64(i64 %12, i64 %_35)
%14 = extractvalue { i64, i1 } %13, 1
%15 = extractvalue { i64, i1 } %13, 0
br i1 %14, label %bb26, label %bb1
```

```
bb26:
%.sroa.0.1 = phi i64 [ 0, %bb17 ], [ 0, %bb14 ], [ 0, %bb8 ], [ 1, %bb1 ]
%16 = insertvalue { i64, i64 } undef, i64 %.sroa.0.1, 0
%17 = insertvalue { i64, i64 } %16, i64 %val.0, 1
ret { i64, i64 } %17
```

Without optimizations, we get a bigger mess (most optimization passes are various CFG cleanups).



Exercise: try to trace through what each basic block is doing. You will want to open the SVGs in a separate tab to do that. I recommend following the optimized version, since it is much less noisy.

Comparing optimized vs. unoptimized is a good way to see how much the compiler does to simplify the stuff the language frontend gives it. At -O0? All allocas. At -O2? No allocas! ↺

---

(7) Once upon a time we had typed pointers, like `i32*`. These turned out to generate more problems than they solved, requiring frequent casts in IR in exchange for mediocre type safety. See https://llvm.org/docs/OpaquePointers.html for a more complete history. ↺

---

(8) Sarcasm. ↺

---

(9) I quietly judge LLVM for having instructions named `inttoptr` when `int2ptr` just reads so much nicer. ↺

Related Posts

- 2025-04-29 / ***Protobuf Tip #3: Accepting Mistakes We Can't Fix***
- 2025-04-22 / ***Protobuf Tip #3: Enum Names Need Prefixes***
- 2025-04-21 / ***Cheating the Reaper in Go***