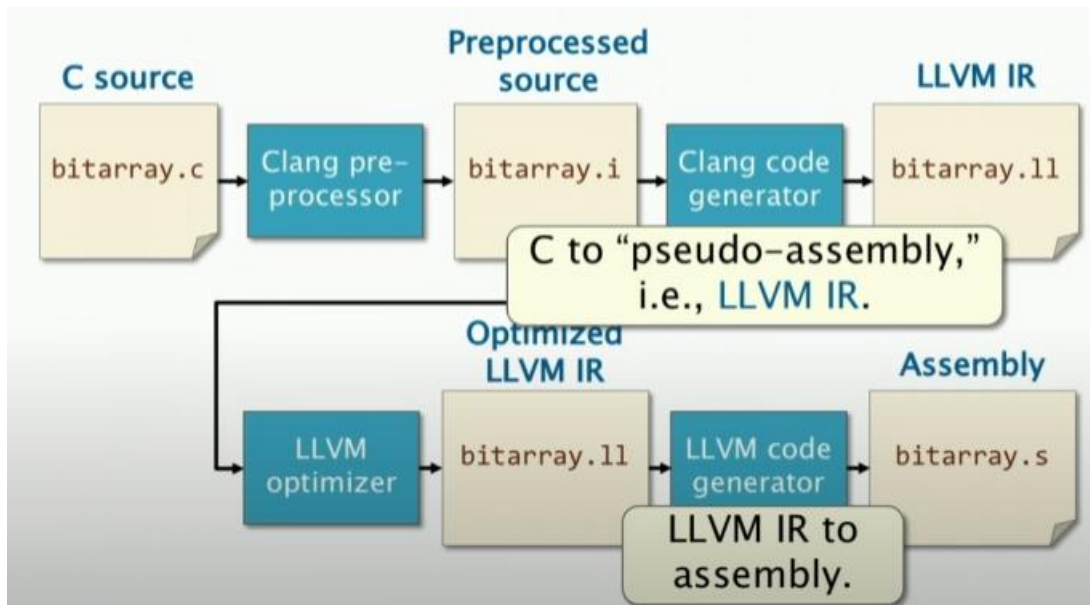


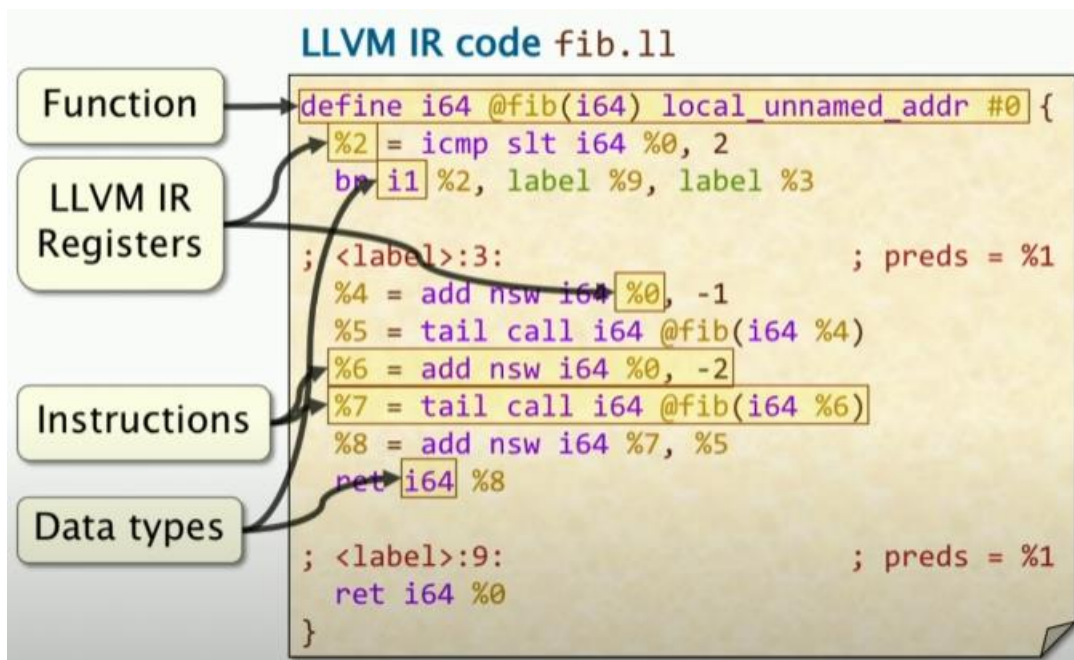
Unit 5 C to Assembly

Assembly reveals what compiler optimized and help us debug if the compiler optimization creates a bug.



The process that translate C source code to Assembly using Clang/LLVM.

An example code of LLVM IR.



LLVM IR uses a simple instruction format `<destination operand> = <opcode> <source operand>`.

The differences between LLVM IR and Assembly

1. LLVM IR has smaller instruction set
2. Infinite LLVM IR registers, similar to variables in C
3. No implicit FLAGS register or condition code
4. No explicit stack pointer or frame pointer
5. C-like type system
6. C-like functions

LLVM IR Registers

LLVM registers are like C variables which supports an infinite number of registers and register names are local to each LLVM IR function.

LLVM IR Instruction

Syntax for instructions that produce a value

`%<name> = <opcode> <operand list>`

Syntax for other instructions

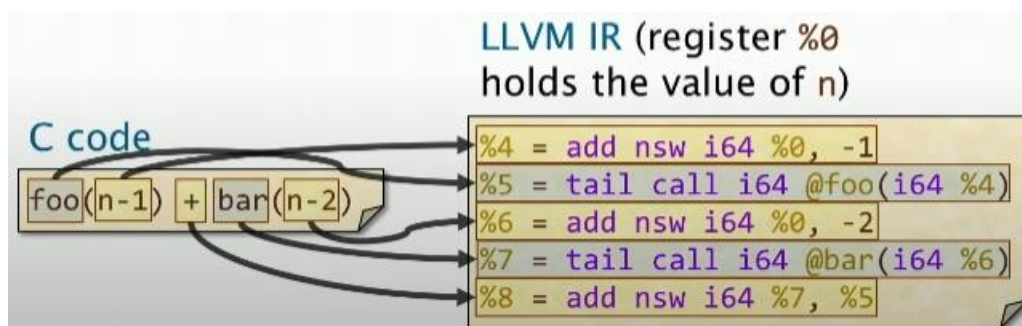
`<opcode> <operand list>`

| Type or operation | | Example(s) |
|----------------------|---------------------------|--|
| Data movement | Stack allocation | <code>alloca</code> |
| | Memory read | <code>load</code> |
| | Memory write | <code>store</code> |
| | Type conversion | <code>bitcast, ptrtoint</code> |
| Arithmetic and logic | Integer arithmetic | <code>add, sub, mul, div, shl, shr</code> |
| | Floating-point arithmetic | <code>fadd, fmul</code> |
| | Binary logic | <code>and, or, xor, not</code> |
| | Boolean logic | <code>icmp</code> |
| | Address calculation | <code>getelementptr</code> |
| Control flow | Unconditional jump | <code>br <location></code> |
| | Conditional jump | <code>br <condition>, <true>, <false></code> |
| | Subroutines | <code>call, ret</code> |
| | Maintaining SSA form | <code>phi</code> |

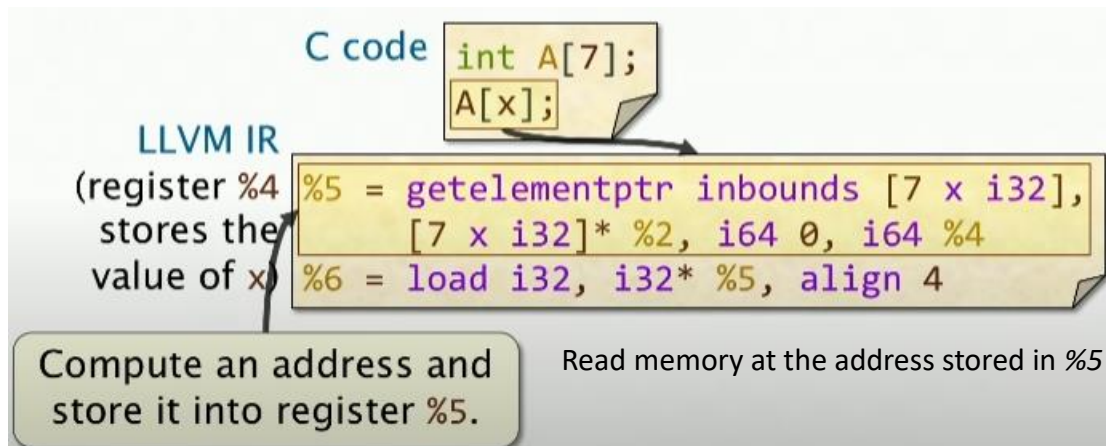
LLVM IR Data Types

- Integers: `i <number>` (Number indicate how many bits)
- Floating-points: `double, float`
- Arrays: `[<number> x <type>]`
- Structs: `{<type>, ...}`
- Vectors: `<<number> x <type>>`
- Pointers: `<type>*`
- Labels: `label`

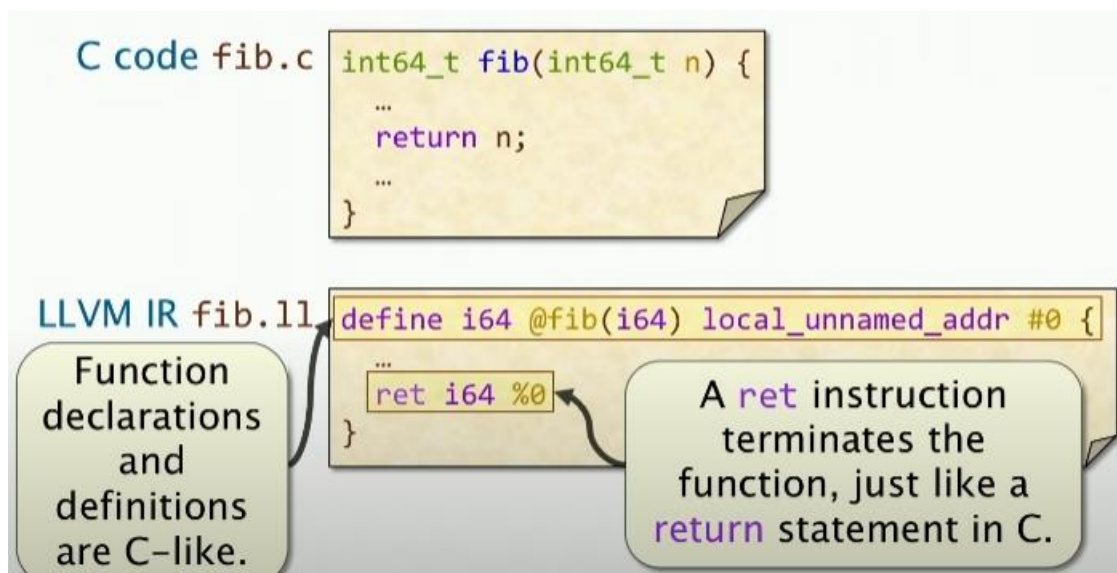
Straight Line C Code in LLVM IR



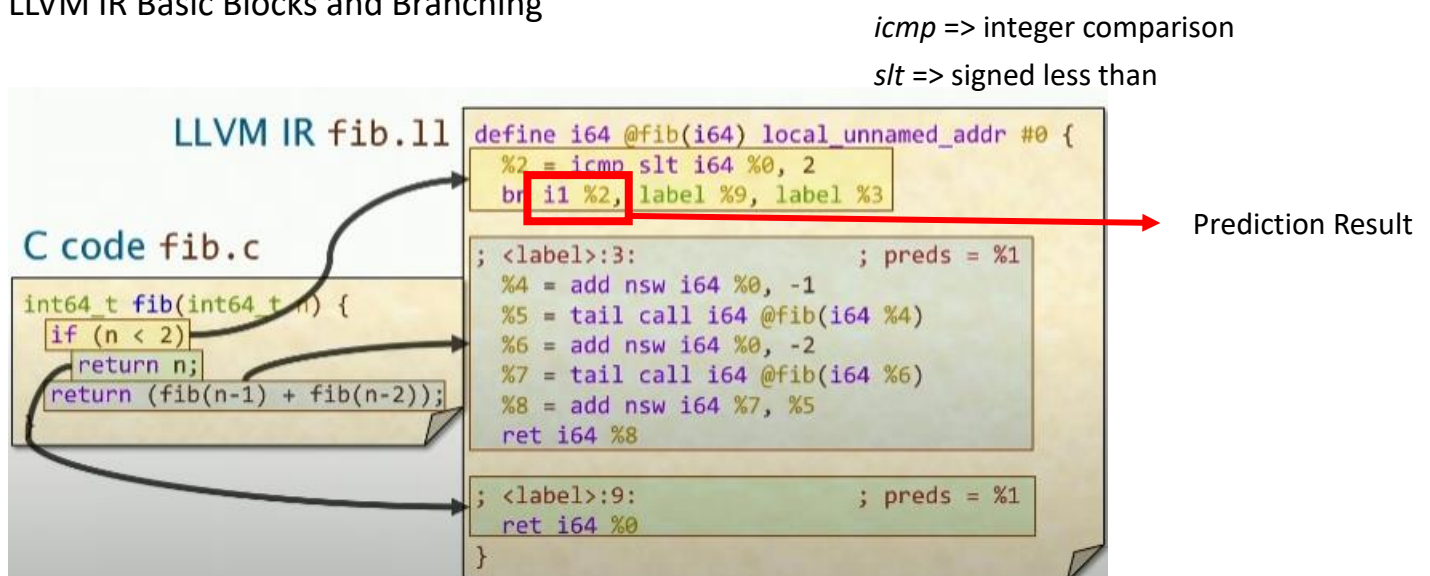
Aggregate Types



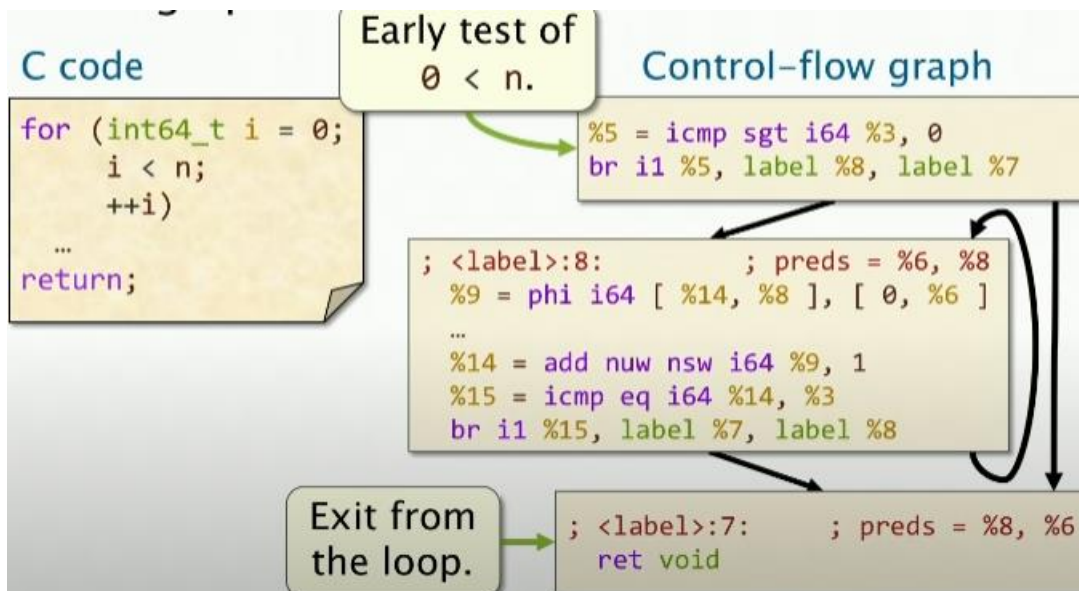
LLVM IR Functions



LLVM IR Basic Blocks and Branching



Basic Blocks: Sequences of instructions where control only enters through the first and exits from the last.



Unit 9 What Compilers Can and Cannot do

Compiler is a software and they can have bugs.

If you know what compiler well do, you don't need to do it yourself. (Make the code clean and simple)

From LLVM IR to Optimized LLVM IR

Optimizing compiler performs a sequence of transformation passes on the IR code.

Each transformation pass analyzes and edits the code.

The order of the transformations is predetermined.

Compiler Reports

-*Rpass* = *<string>*: Produces reports of which optimizations matching *<string>* were successful.

-*Rpass-missed*=*<string>*: Produces reports of which optimizations matching *<string>* were not successful.

-*Rpass-analysis*=*<string>*: Produces reports of the analysis performed by optimizations matching *<string>*

<string> is a regular expression *“.*”* for the whole report.

Compiler Optimization Compare to New Bentley Rules

Optimized compiler can perform some of the New Bentley Rules and some extra optimizations. For data structures, the compiler is good at utilize registers since accessing memory is expensive compare to registers.

~~New Bentley Rules~~

Compiler Optimizations

Data structures

- ~~• Packing and encoding~~
- ~~• Augmentation~~
- ~~• Precomputation~~
- ~~• Compile-time initialization~~
- ~~• Caching~~
- ~~• Lazy evaluation~~
- ~~• Sparsity~~

Loops

- Hoisting
- ~~• Sentinels~~
- Loop unrolling
- Loop fusion*
- Eliminating wasted iterations*

*Restrictions may apply.

Logic

- Constant folding and propagation
- Common-subexpression elimination
- Algebraic identities
- Short-circuiting
- Ordering tests*
- ~~• Creating a fast path~~
- Combining tests*

Functions

- Inlining
- Tail-recursion elimination
- ~~• Coarsening recursion~~

More Compiler Optimizations

Data structures

- Register allocation
- Memory to registers
- Scalar replacement of aggregates
- Alignment

Loops

- Vectorization
- Unswitching
- Idiom replacement
- Loop fission*
- Loop skewing*
- Loop tiling*
- Loop interchange*

*In development in Clang/LLVM.

Logic

- Elimination of redundant instructions
- Strength reduction
- Dead-code elimination
- Idiom replacement
- Branch reordering
- Global value numbering

Functions

- Unswitching
- Argument elimination

Compiler Optimizations may change over time.