# Compiler construction

Lecture 3: LLVM language and tools

Magnus Myreen

Spring 2019

Chalmers University of Technology — Gothenburg University

# Introduction to LLVM

## Register machines

**Fast but scarce**
Registers are places for data inside the CPU.

+ up to 10 times faster access than to main memory
- expensive; typically just 32 of them in a 32-bit CPU

Typically, arithmetic operations, conditional jumps, etc. operate on values stored in registers.

Most modern assembly languages use registers, which correspond closely to the machine registers.

**Low Level Virtual Machine (LLVM)**

- LLVM is a virtual machine
- LLVM has an unbounded number of registers
- A later step does register allocation, mapping virtual registers to real machine registers

## The LLVM project

**The LLVM Infrastructure**

- A collection of (C++) software libraries and tools to help in building compilers, debuggers, program analysers, etc.
- Tools available on Studat Linux machines
- Can also be downloaded to your own computer, see llvm.org

**History**

- Started as academic project at University of Illinois in 2002
- Now a large open source project with many contributors and a growing user base

**Related projects**

**Clang** C/C++ front end; aims to replace GCC
**CLI** MicroSoft Common Language Interface
**GHC** has a LLVM backend

## ACM Software System Award

LLVM was the 2012 winner of the ACM Software System Award.

Previous winners include:

- VMware
- Make
- Java
- Spin
- Coq
- Apache
- WWW
- TCP/IP
- Postscript
- TEX
- Unix
- ...

## The LLVM language

**Characteristic features**

- Three adress-code with two source registers and one destination register:
  `%t2 = add i32 %t0, %t1`
- One source can be a value:
  `%t5 = add i32 %t3, 42`
- Instructions are typed:
  ```
  %t8 = fadd double %t6, %t7
  store i32 %t5 , i32* %r
  ```
- New register for each result, i.e., Static Single Assignment form

## Hello world in LLVM

```
@hw = internal constant [13 x i8] c"hello world\0A\00"
declare i32 @puts(i8*)

define i32 @main () {
entry: %t1 = bitcast [13 x i8]* @hw to i8*
       %t2 = call i32 @puts(i8* %t1)
       ret i32 %t2
}
```

**Comments**

- The string `@hw` is a global constant (global names start with an `@`-sign); note escape sequences!
- The library function `@puts` is declared, we provide its type signature
- `@hw` is cast to type of argument to `@puts`, better (type-safe) solution later

## An illegal LLVM program

```
declare void @printInt(i32 %n)
define i32 @main() {
entry: %t1 = call i32 @sum(i32 100)
       call void  @printInt(i32 %t1)
       ret i32 0
}
define i32 @sum (i32 %n) {
entry: %sum = i32 0
       %i = i32 0
       br label %lab1
lab1:  %i = add i32 %i, 1
       %sum = add i32 %sum, %i
       %t = icmp eq i32 %i, %n
       br i1 %t, label %end, label %lab1
end:   ret i32 %sum
}
```

## An illegal LLVM program

```
declare void @printInt(i32 %n)
define i32 @main() {
entry: %t1 = call i32 @sum(i32 100)
       call void  @printInt(i32 %t1)
       ret i32 0
}
define i32 @sum (i32 %n) {
entry: %sum = i32 0
       %i = i32 0
       br label %lab1
lab1:  %i = add i32 %i, 1
       %sum = add i32 %sum, %i
       %t = icmp eq i32 %i, %n
       br i1 %t, label %end, label %lab1
end:   ret i32 %sum
}
```

**Reasons**

- Not SSA form: Two assignments to `%i` and `%sum`
- There is no `%reg = val` instruction

## Corrected program

```
define i32 @sum (i32 %n) {
entry: %sum = alloca i32
       store i32 0, i32* %sum
       %i = alloca i32
       store i32 0, i32* %i
       br label %lab1
lab1:  %t1 = load i32, i32* %i
       %t2 = add i32 %t1, 1
       %t3 = load i32, i32* %sum
       %t4 = add i32 %t2, %t3
       store i32 %t2, i32* %i
       store i32 %t4, i32* %sum
       %t5 = icmp eq i32 %t2, %n
       br i1 %t5, label %end,
                  label %lab1
end:   ret i32 %t4
}
```

## Corrected program

```
define i32 @sum (i32 %n) {
entry: %sum = alloca i32
       store i32 0, i32* %sum
       %i = alloca i32
       store i32 0, i32* %i
       br label %lab1
lab1:  %t1 = load i32, i32* %i
       %t2 = add i32 %t1, 1
       %t3 = load i32, i32* %sum
       %t4 = add i32 %t2, %t3
       store i32 %t2, i32* %i
       store i32 %t4, i32* %sum
       %t5 = icmp eq i32 %t2, %n
       br i1 %t5, label %end,
                  label %lab1
end:   ret i32 %t4
}
```

**Comments**

- `%i` and `%sum` are now pointers to memory locations
- Only one assignment to any register

**Problem**

This program has a lot more memory traffic!

What can LLVM's optimizer do about that?

## Optimizing `@sum`

```
> opt -mem2reg sum.ll > sumreg.bc
> llvm-dis sumreg.bc
> cat sumreg.ll
```

```
define i32 @sum(i32 %n) {
entry:
  br label %lab1
lab1:
  %i.0 = phi i32 [ 0, %entry ], [ %t2, %lab1 ]
  %sum.0 = phi i32 [ 0, %entry ], [ %t4, %lab1 ]
  %t2 = add i32 %i.0, 1
  %t4 = add i32 %t2, %sum.0
  %t5 = icmp eq i32 %t2, %n
  br i1 %t5, label %end, label %lab1
end:
  ret i32 %t4
}
```

## Φ 'functions'

### Single Static Assignment (SSA) form

- Only one assignment in the program text to each variable
- But dynamically, this assignment can be executed many times
- Many stores to a memory location are allowed
- Also, Φ (`phi`) instructions can be used, in the beginning of a basic block
  - Value is one of the arguments, depending on from which block control came to this block
  - Register allocation tries to keep these variables in same real register

**Why SSA form?**
Many code optimizations can be done more efficiently (later).

## Optimizing even further

**Many optimization passes**
The LLVM optimizer `opt` implements many code analysis and improvement methods.

To get a default selection, give command line argument:

-O3 (previously known as `-std-compile-opts`)

**Result after `opt -O3` (1/2)**

```
declare void @printInt(i32)

define i32 @main() {
entry:
  tail call void @printInt(i32 5050)
  ret i32 0
}
```

## Optimizing `@sum` further

**Result after `opt -O3` (2/2)**

```
define i32 @sum(i32 %n) nounwind readnone {
entry:
  %0 = shl i32 %n, 1
  %1 = add i32 %n, -1
  %2 = zext i32 %1 to i33
  %3 = add i32 %n, -2
  %4 = zext i32 %3 to i33
  %5 = mul i33 %2, %4
  %6 = lshr i33 %5, 1
  %7 = trunc i33 %6 to i32
  %8 = add i32 %0, %7
  %9 = add i32 %8, -1
  ret i32 %9
}
```

## Analysis of optimized code for `@sum`

### Observations

- Previous loop with execution time $O(n)$ has been optimized to code without loop, running in constant time
- Recall $1 + 2 + \ldots + n = \frac{n(n+1)}{2}$, check that optimized code computes this
- Why extensions/truncations to and from 33 bits?
- What happens when $n$ is negative?

### Optimization

- `opt -O3` includes many optimization passes
- Use `-time-passes` for an overview
- We will discuss some of these algorithms later

## `@printInt` and other IO functions

**Part of `runtime.ll`**

```
@dnl = internal constant [4 x i8] c"%d\0A\00"

declare i32 @printf(i8*, ...)

define void @printInt(i32 %x) {
entry: %t0 = getelementptr [4 x i8], [4 x i8]* @dnl, i32 0
                                                    , i32 0
       call i32 (i8*, ...)* @printf(i8* %t0, i32 %x)
       ret void
}
```

We provide this file on the course web site; you just have to make sure that it is available for linking.

## Linking and running the program

**Linking is done by `llvm-link`**

```
> llvm-link sumopt.bc runtime.bc -o a.out.bc
> llc --filetype=obj a.out.bc
> gcc a.out.o
> ./a.out
5050
```

When creating an executable file:

- Link the bitcode files with `llvm-link`.
- Compile the bitcode file to a native object file using `llc`
- Use a C compiler to link with `libc` and produce an executable

## What is in `a.out.bc`

**Disassemble it[1]!**

```
> cat a.out.bc | llvm-dis -
```

```
; ModuleID = 'a.out.bc'

@dnl = internal constant [4 x i8] c"%d\0A\00"

define i32 @main() {
entry:
  %t0 = getelementptr [4 x i8]* @dnl, i32 0, i32 0
  call i32 (i8*, ...)* @printf(i8* %t0, i32 5050)
  ret i32 0
}

declare i32 @printf(i8*, ...)
```

[1]Result slightly edited

## LLVM language and tools

## Types in LLVM

**An incomplete list**
Below $t$ and $t_i$ are types and $n$ an integer literal.

- $n$ bit integers: `in`
- `float` and `double`
- Labels: `label`
- The void type: `void`
- Functions: $t\ (t_1, t_2, \ldots, t_n)$
- Pointer types: $t*$
- Structures: $\{t_1, t_2, \ldots, t_n\}$
- Arrays: $[n \times t]$

## Named types and type equality

**Named types**
We can give names to types, for example:

```
%length = type i32
%list = type %Node*
%Node = type { i32, %Node* }

%tree = type %Node2*
%Node2 = type { %tree, i32, %tree }

%matrix = type [ 100 x [ 100 x double ] ]
```

**Type equality**
LLVM uses structural equality for types.

When disassembling bitcode files that contain several structurally equal types with different names, this may give confusing results.

## Identifiers

**Local identifiers**
Registers and named types have local names and start with a %-sign.

**Global identifiers**
Functions and global variables have global names and start with an @-sign.

JAVALETTE does not have global variables, but you will need to define global names for string literals, as in

```
@hw = internal constant [13 x i8] c"hello world\0A\00"
```

After this definition, `@hw` has type `[13 x i8]*`.

## Constants

**Literals**

- Integer and floating-point literals are as expected
- `true` and `false` are literals of type `i1`
- `null` is a literal of any pointer type

**Aggregates**
Constant expressions of structure and array types can be formed; not needed by JAVALETTE.

## Function definitions

**Function definition form**

```
define t @name(t₁ x₁, t₂ x₂, ..., tₙ xₙ) {
l₁:  block₁
l₂:  block₂
  ...
lₘ:  blockₘ
}
```

where `@name` is a global name (the name of the function), the $x_i$ are local names (the parameters) and the $block_i$ are labeled basic blocks.

**Basic blocks**
A basic block is a label ($l_i$) followed by a colon and a sequence of LLVM instructions, each on a separate line. The last instruction must be a terminator instruction.

## Function declarations

**Type-checking**
The LLVM assembler does type-checking. Hence it must know the types of all external functions, i.e., functions used but not defined in the compiled unit.

**Simple function declaration**
The basic form is: `declare t @name(t₁, t₂, ..., tₙ)`

For JAVALETTE, this is necessary for IO functions. The compiler would typically insert in each file:

```
declare void   @printInt(i32)
declare void   @printDouble(double)
declare void   @printString(i8*)
declare i32    @readInt()
declare double @readDouble()
```

## LLVM tools

- `llvm-as` An assembler that translates llvm code to bitcode (`prog.ll` to `prog.bc`)
- `llvm-dis` A disassembler that translates in the opposite direction
- `lli` An interpreter/JIT compiler that executes a bitcode file containing a `@main` function
- `llvm-link` A linker that links together several bitcode files
- `llc` A compiler that translates a bitcode to native assembler or object files
- `opt` An optimizer that optimizes bitcode; many options to decide on which optimizations to run; use `-O3` to get a default selection
- `clang` Drop-in replacement for GCC

## Use of LLVM in your compiler

**Default mode**
Your code generator produces an assembler file (`.ll`). Then your main program uses system calls to first assemble this with `llvm-as`, optimize with `opt` and then link together with `runtime.bc`.

**Other modes**
More advanced and we do not recommend these for this project.

- C++ programmers can use the LLVM libraries to build in-memory representation and then output bitcode file
- Haskell programmers can access C++ libraries via Hackage package `LLVM`

If you want to use non-standard libraries that you haven't written yourselves, make sure to get Magnus' approval first.

## LLVM instructions

**Basic collection**
Basic JAVALETTE will only need the following instructions:

- Terminator instructions: `ret` and `br`
- Arithmetic operations:
  - For integers `add`, `sub`, `mul`, `sdiv` and `srem`
  - For doubles `fadd`, `fsub`, `fmul` and `fdiv`
- Memory access: `alloca`, `load`, `getelementptr` and `store`
- Other: `icmp`, `fcmp` and `call`

Some of the extensions will need more instructions.

**Next time**
Code generation for LLVM.