

# LLVM & LLVM Bitcode Introduction



# What is LLVM ? (1/2)

- LLVM (Low Level Virtual Machine) is a compiler infrastructure
  - Written by C++ & STL
- History
  - The LLVM project started in 2000 at the University of Illinois
  - BSD-style license (Berkeley Software Distribution License)
  - LLVM: A compilation framework for lifelong program analysis & transformation (a published paper by Chris Lattner, Vikram Adve)(CGO 04)
  - 2005, Apple hired Chris Lattner



Chris Lattner



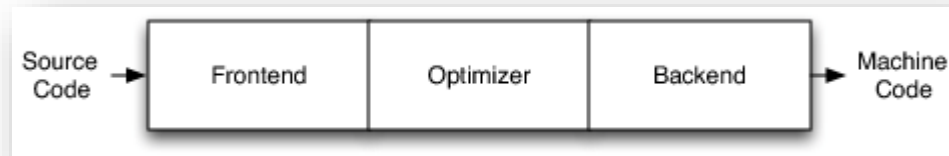
# What is LLVM? (2/2)

- Targets of LLVM
  - Lifelong optimization
  - Integration
    - AOT (ahead-of-time) compiler, JIT (just-in-time) compiler, interpreter
- Compare with GCC
  - More advanced architecture
  - Better optimizations
  - Faster compilation
  - GCC currently support more targets than LLVM

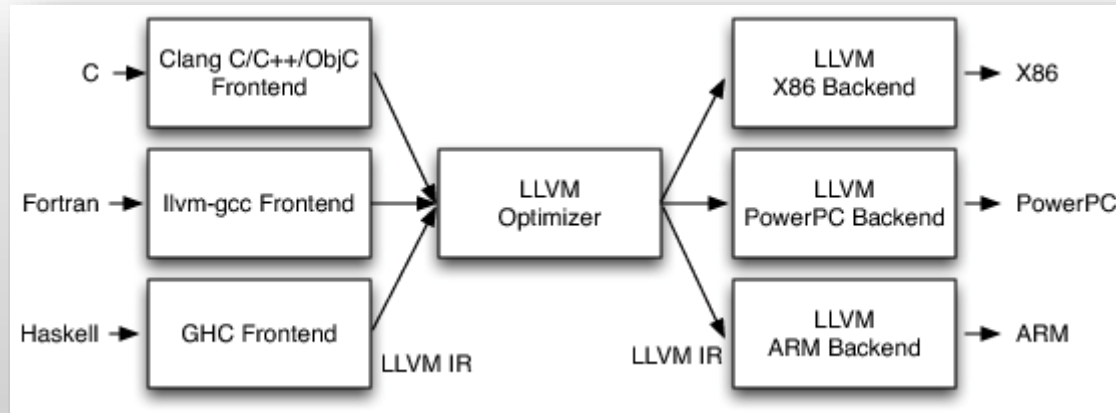


# LLVM retargetability design

## Traditional Three-Phase Compiler design

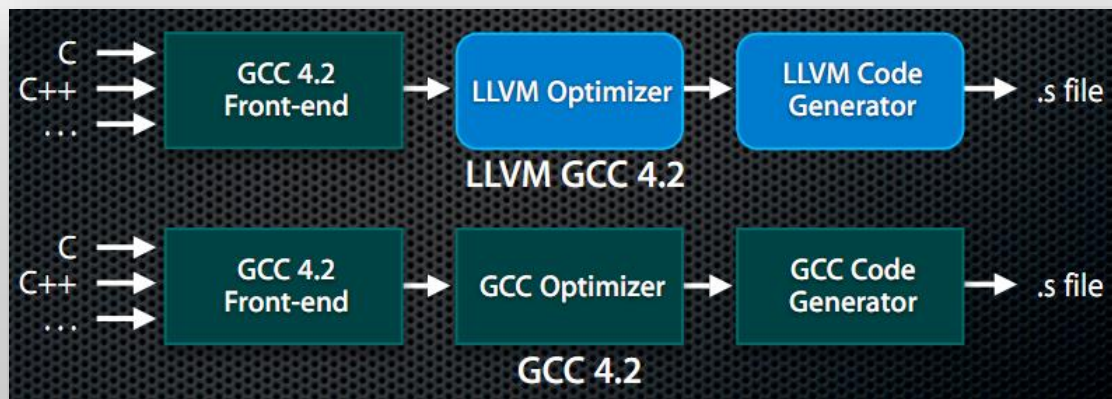


## LLVM retargetability

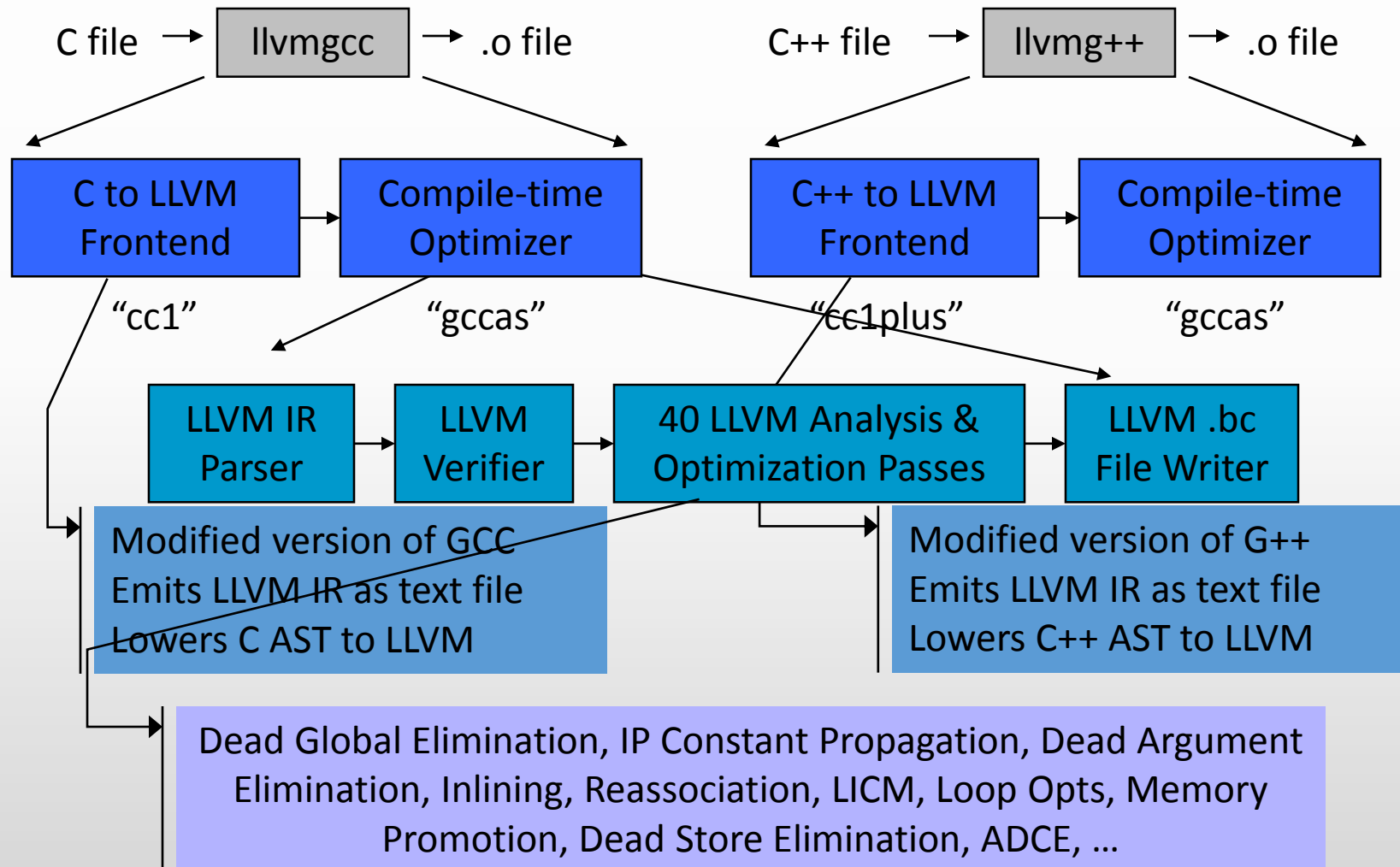


# LLVM tools

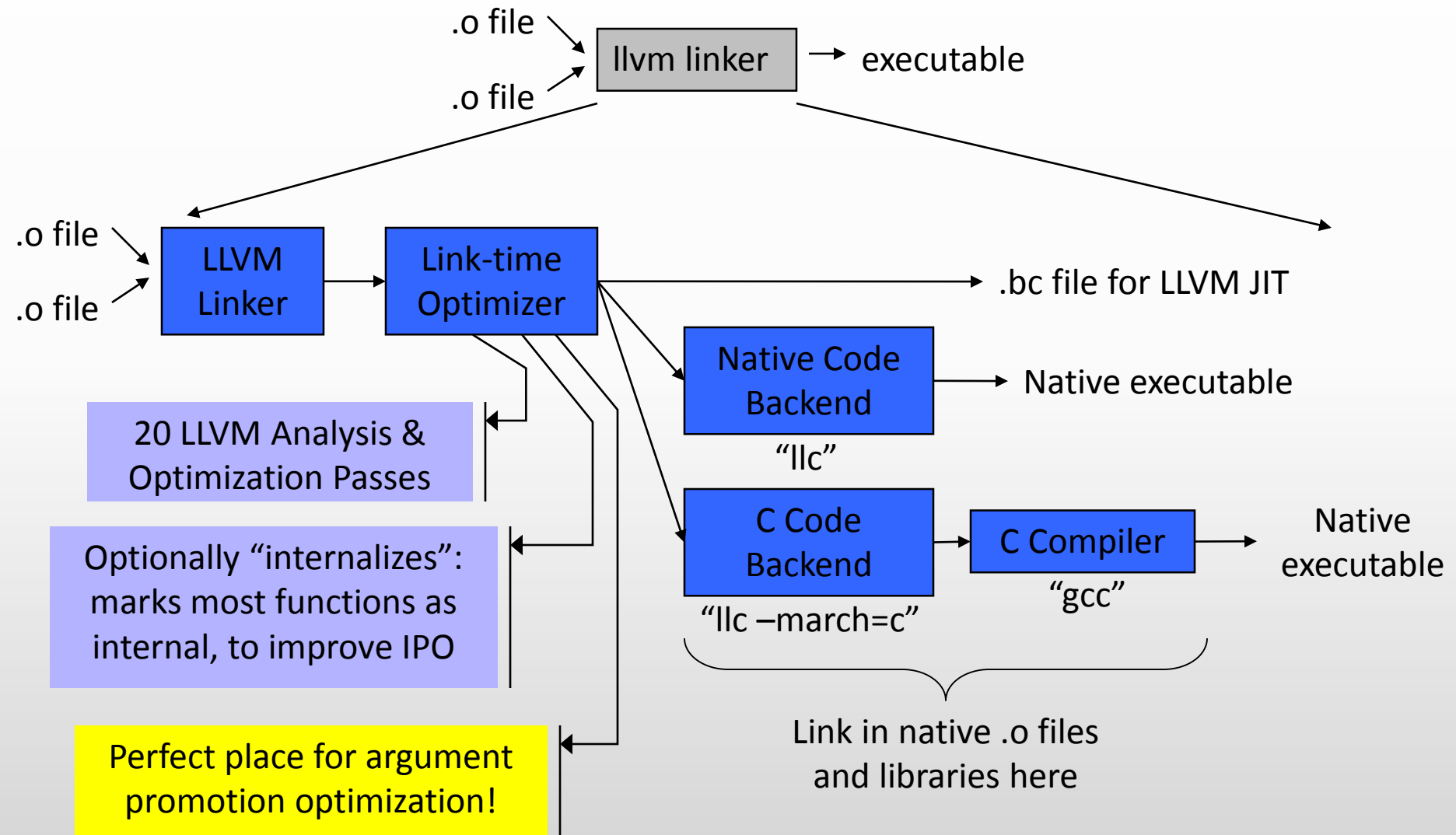
- **llvm-as**: assemble a human-readable .ll file into bitcode
- **llvm-dis**: disassemble a bitcode file into a human-readable .ll file
- **opt**: run a series of LLVM-to-LLVM optimizations on a bitcode file
- **llc**: generate native machine code for a bitcode file
- **lli**: directly run a program compiled to bitcode using a JIT compiler or interpreter
- **llvm-link**: link several bitcode files into one
- **clang**: C, C++, Object C front-end for LLVM
- **llvm-gcc**: GCC-based C front-end for LLVM
- **llvm-g++**: GCC-based C++ front-end for LLVM



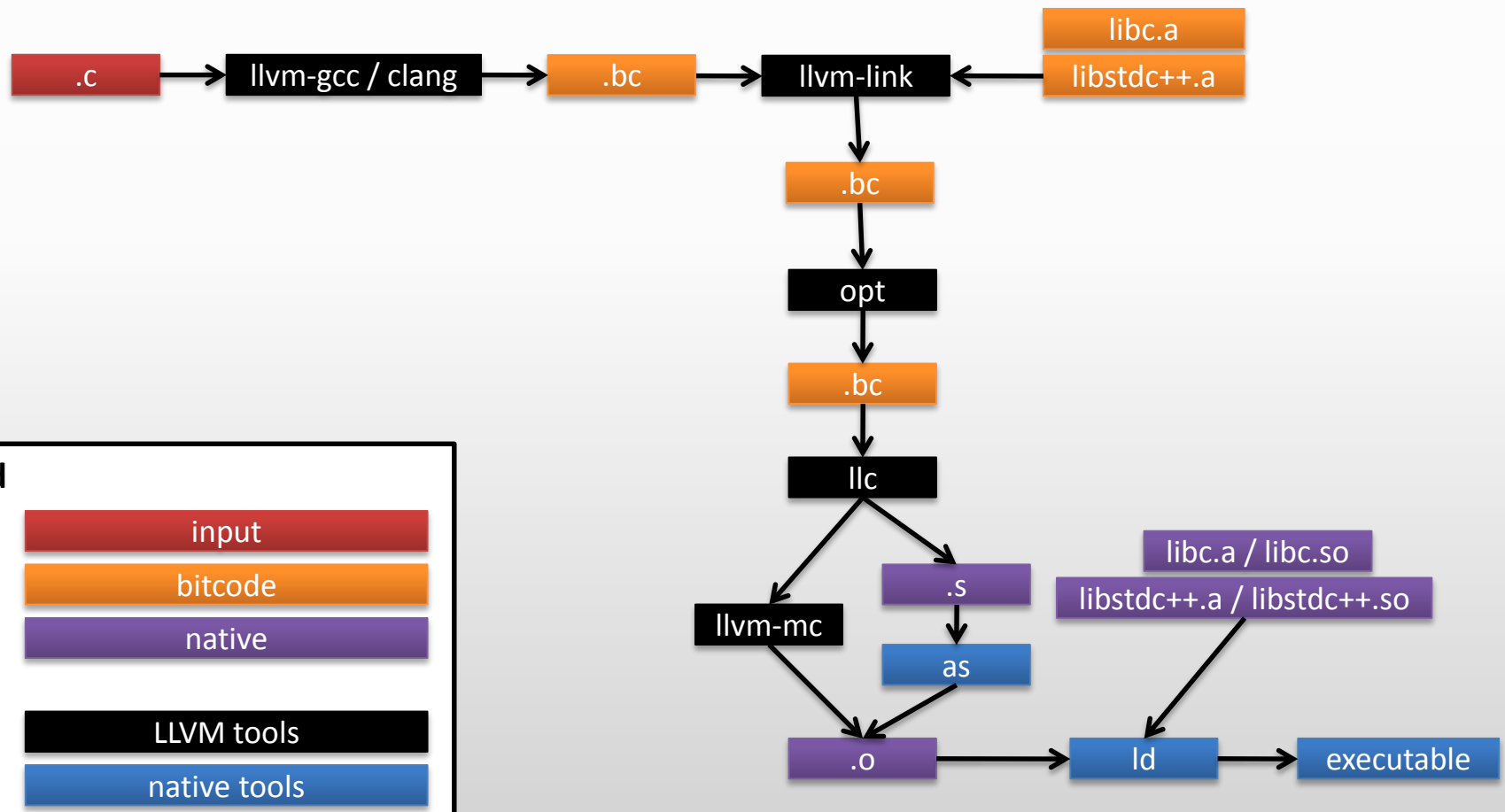
# Looking into events at compile-time



# Looking into events at link-time

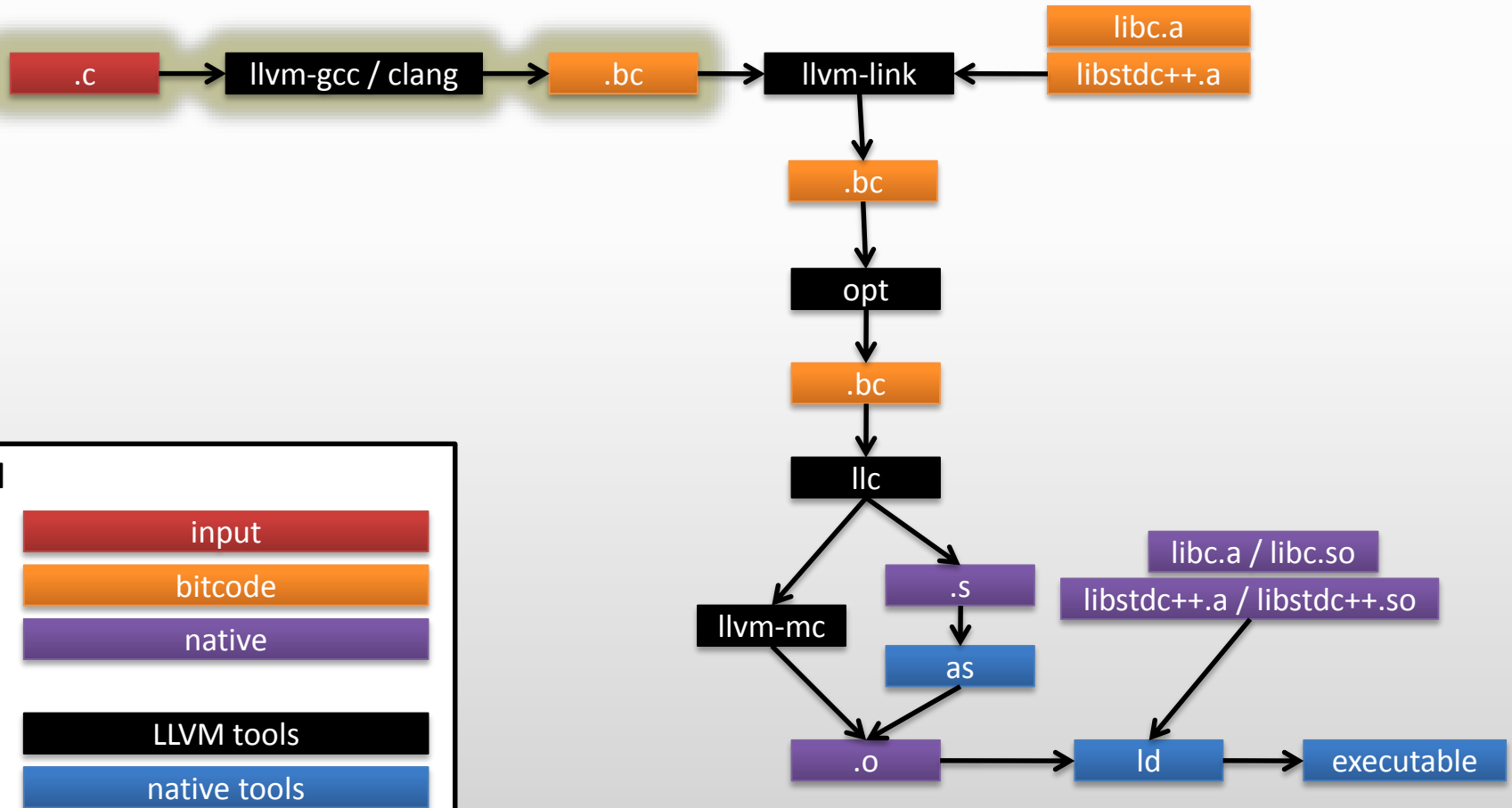


# LLVM framework





# Source code to LLVM IR




# LLVM IR

- LLVM code representation
  - In memory compiler IR (Intermediate Representation)
  - Human readable assembly language – LLVM IR (\*.ll)
  - On-disk **bitcode** representation (\*.bc)
- LLVM IR is SSA form (Static single assignment form)
  - Each variable is assigned exactly once
  - Use-def chains are explicit and each contains a single element

# Global variable & Array representation

```
long long x[3][3] = {{ 1, 2, 3 }, { 4, 5, 6 }, { 7, 8, 9 }};  
long long y[3][3] = {{ 9, 8, 7 }, { 6, 5, 4 }, { 3, 2, 1 }};  
long long z[3][3];
```


```
int main()  
{  
    int sum = 0;  
  
    for(int i = 0; i < 3; ++i) {  
        for(int j = 0; j < 3; ++j) {  
            z[i][j] = 0;  
  
            for(int k = 0; k < 3; ++k) {  
                z[i][j] += x[i][k] * y[k][j];  
            }  
        }  
    }  
}
```



```
@x = global [3 x [3 x i64]]  
      [[3 x i64] [i64 1, i64 2, i64 3],  
      [3 x i64] [i64 4, i64 5, i64 6],  
      [3 x i64] [i64 7, i64 8, i64 9]  
      ], align 8  
@y = global [3 x [3 x i64]]  
      [[3 x i64] [i64 9, i64 8, i64 7],  
      [3 x i64] [i64 6, i64 5, i64 4],  
      [3 x i64] [i64 3, i64 2, i64 1]  
      ], align 8  
@z = common global [3 x [3 x i64]] zeroinitializer, align 8
```

# Function entry & Local variables

```
long long x[3][3] = {{ 1, 2, 3 }, { 4, 5, 6 }, { 7, 8, 9 }};  
long long y[3][3] = {{ 9, 8, 7 }, { 6, 5, 4 }, { 3, 2, 1 }};  
long long z[3][3];  
  
int main()  
{  
    int sum = 0;  
  
    for(int i = 0; i < 3; ++i) {  
        for(int j = 0; j < 3; ++j) {  
            z[i][j] = 0;  
  
            for(int k = 0; k < 3; ++k) {  
                z[i][j] += x[i][k] * y[k][j];  
            }  
        }  
    }  
}
```



```
define i32 @main() nounwind {  
entry:  
    %retval = alloca i32                                ; <i32*> [#uses=2]  
    %k = alloca i32                                       ; <i32*> [#uses=6]  
    %j = alloca i32                                       ; <i32*> [#uses=8]  
    %i = alloca i32                                       ; <i32*> [#uses=8]  
    %sum = alloca i32                                     ; <i32*> [#uses=4]  
    %"alloca point" = bitcast i32 0 to i32              ; <i32> [#uses=0]  
}
```

# Inner-most loop

```
long long x[3][3] = {{ 1, 2, 3 }, { 4, 5, 6 }, { 7, 8, 9 }};  
long long y[3][3] = {{ 9, 8, 7 }, { 6, 5, 4 }, { 3, 2, 1 }};  
long long z[3][3];
```

```
int main()  
{  
    int sum = 0;  
  
    for(int i = 0; i < 3; ++i) {  
        for(int j = 0; j < 3; ++j) {  
            z[i][j] = 0;  
  
            for(int k = 0; k < 3; ++k) {  
                z[i][j] += x[i][k] * y[k][j];  
            }  
        }  
    }  
}
```

```
bb2:                                ; preds = %bb3  
%5 = load i32* %i, align 4        ; <i32> [#uses=1]  
%6 = load i32* %j, align 4        ; <i32> [#uses=1]  
%7 = load i32* %i, align 4        ; <i32> [#uses=1]  
%8 = load i32* %j, align 4        ; <i32> [#uses=1]  
%9 = getelementptr inbounds [3 x [3 x i64]]* @z, i32 0, i32 %7 ; <[3 x i64]*> [#uses=1]  
%10 = getelementptr inbounds [3 x i64]* %9, i32 0, i32 %8 ; <i64*> [#uses=1]  
%11 = load i64* %10, align 8       ; <i64> [#uses=1]  
%12 = load i32* %i, align 4        ; <i32> [#uses=1]  
%13 = load i32* %k, align 4        ; <i32> [#uses=1]  
%14 = getelementptr inbounds [3 x [3 x i64]]* @x, i32 0, i32 %12 ; <[3 x i64]*> [#uses=1]  
%15 = getelementptr inbounds [3 x i64]* %14, i32 0, i32 %13 ; <i64*> [#uses=1]  
%16 = load i64* %15, align 8       ; <i64> [#uses=1]  
%17 = load i32* %k, align 4        ; <i32> [#uses=1]  
%18 = load i32* %j, align 4        ; <i32> [#uses=1]  
%19 = getelementptr inbounds [3 x [3 x i64]]* @y, i32 0, i32 %17 ; <[3 x i64]*> [#uses=1]  
%20 = getelementptr inbounds [3 x i64]* %19, i32 0, i32 %18 ; <i64*> [#uses=1]  
%21 = load i64* %20, align 8       ; <i64> [#uses=1]  
%22 = mul i64 %16, %21             ; <i64> [#uses=1]  
%23 = add nsw i64 %11, %22         ; <i64> [#uses=1]  
%24 = getelementptr inbounds [3 x [3 x i64]]* @z, i32 0, i32 %5 ; <[3 x i64]*> [#uses=1]  
%25 = getelementptr inbounds [3 x i64]* %24, i32 0, i32 %6 ; <i64*> [#uses=1]  
store i64 %23, i64* %25, align 8  
%26 = load i32* %k, align 4        ; <i32> [#uses=1]  
%27 = add nsw i32 %26, 1           ; <i32> [#uses=1]  
store i32 %27, i32* %k, align 4  
br label %bb3
```

# LLVM command

- Generate the \*.bc
  - `$ clang -c -emit-llvm a.c -o a.bc`
  - `$ llvm-dis a.bc -o a.ll`
- Generate the \*.ll (human-readable)
  - `$ clang -S -emit-llvm a.c -o a.ll`
- Using interpreter to run bitcode
  - `$ lli test.bc`

# How to build the LLVM

- <http://llvm.org/docs/GettingStarted.html#getting-started>
- Download llvm 3.2, clang., Compiler-RT from <http://llvm.org/releases/download.html#3.2>

```
$ tar xzf llvm-3.2.src.tar.gz
```

```
$ cd llvm-3.2.src/tool
```

```
$ tar xzf clang-3.2.src.tar.gz
```

```
$ mv clang-3.2.src.tar.gz clang
```

```
$ cd llvm-3.2.src/projects
```

```
$ tar xzf compiler-rt-3.2.src.tar.gz
```

```
$ mv compiler-rt-3.2.src compiler-rt
```

```
$ cd where-you-want-to-build-llvm
```

```
$ ../llvm/configure
```

```
$ make
```

# LLVM conclusion

- Integration
  - Ex: clang for static compiler and for JIT compiler
- Low level IR
  - SSA-based
  - Language-independent
  - Machine-independent
  - Allow libraries and portions written by different language
- More and more languages and targets support



# Reference

- LLVM official website
  - <http://llvm.org/>
  - <http://llvm.org/docs/GettingStarted.html>
- LLVM IR
  - <http://llvm.org/docs/LangRef.html>