中山大學 SUN YAT-SEN UNIVERSITY
计算机学院（软件学院）
SCHOOL OF COMPUTER SCIENCE AND ENGINEERING

Compilation Principle
编 译 原 理

第18讲：中间代码(1)

张献伟

xianweiz.github.io

DCS290, 5/9/2024

Review Questions

- Examples of semantic analysis?

  Define-use, scoping, typing, …

- Differences of SDD and SDT?

  SDD: syntax-directed definition, high-level abstraction
  SDT: syntax-directed translation scheme

- S-SDD and L-SDD, which is more general?

  L-SDD, allow to have both synthesized and inherited attributes.

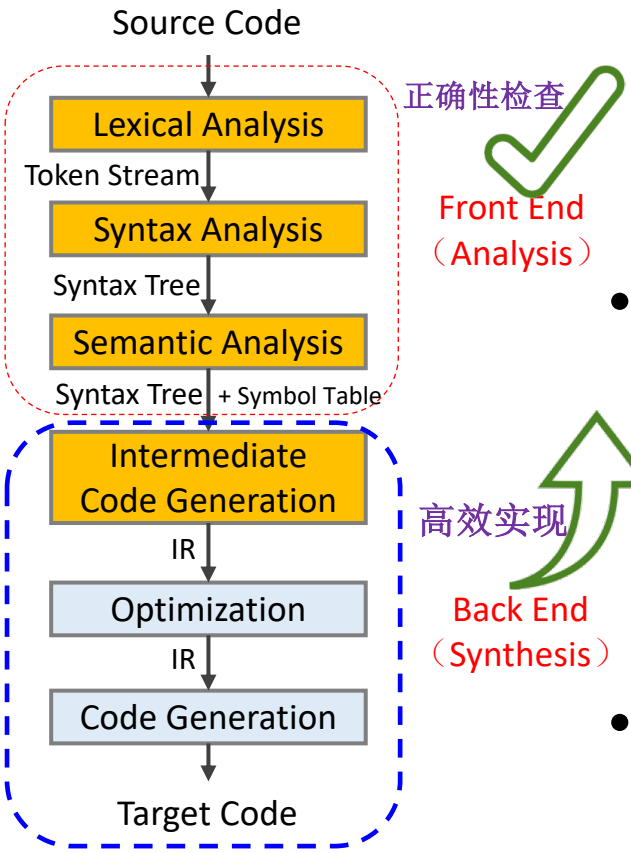
- What marker is used for?

  To calculate inherited attributes (or to transform L-SDD into S-SDD).

- Is symbol table still useful after semantic analysis?

  Yes. Still used in later code generation (and even debugging).

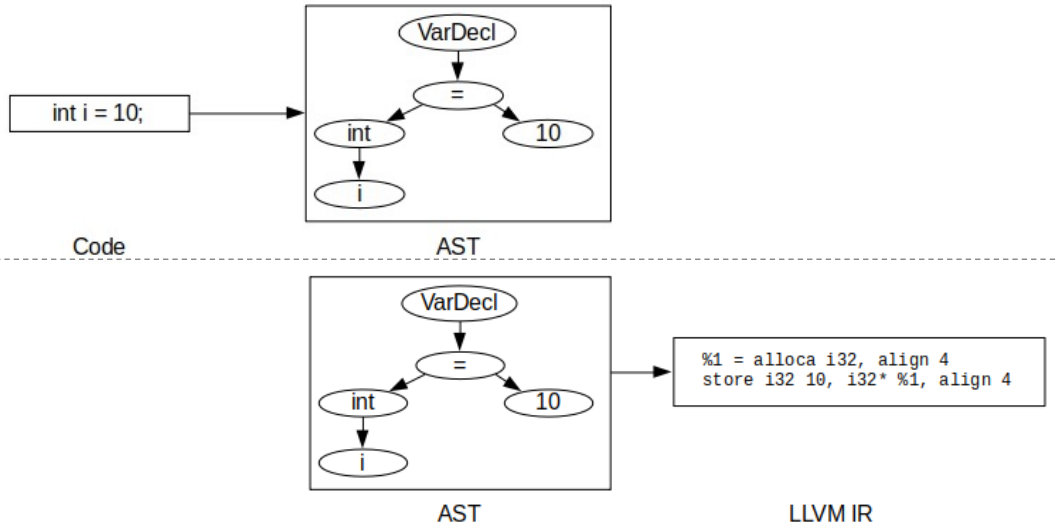
Compilation Phases[编译阶段]



- **Lexical**: source code → tokens
  - RE, NFA, DFA, …
  - Is the program lexically well-formed?
    - E.g., x#y = 1
- **Syntax**: tokens → AST or parse tree
  - CFG, LL(1), LALR(1), …
  - Is the input program syntactically well-formed?
    - E.g., x=1 y=2, for(i = 1)
- **Semantic**: AST → AST + symbol table
  - SDD, SDT, typing, scoping, …
  - Does the input program has a well-defined meaning?
    - E.g., int x; y = x(1)

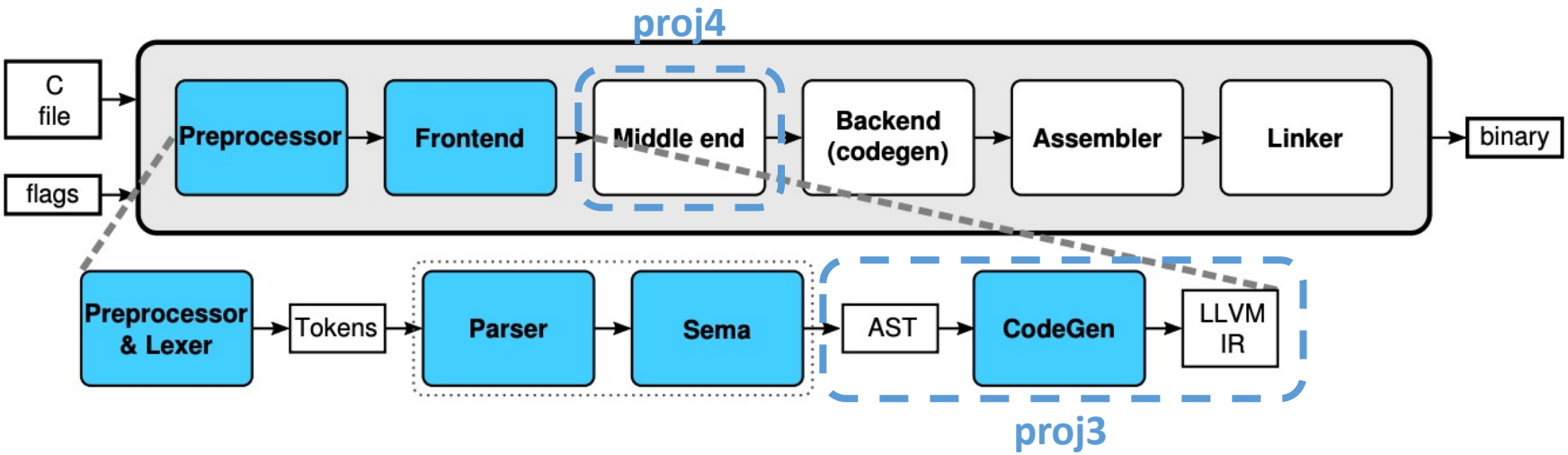
Generating Code: AST to IR[IR生成]

- By now, we have
  - An AST, annotated with scope and type information
- Next, to generate **intermediate representation** (IR)
  - Traverse the AST after the parsing[另外遍历]
    - Writing a *codeGen()* method for the appropriate kinds of AST nodes
  - Syntax-directed translation[语法制导]
    - Generate code while parsing



http://freecompilercamp.org/clang-basics/

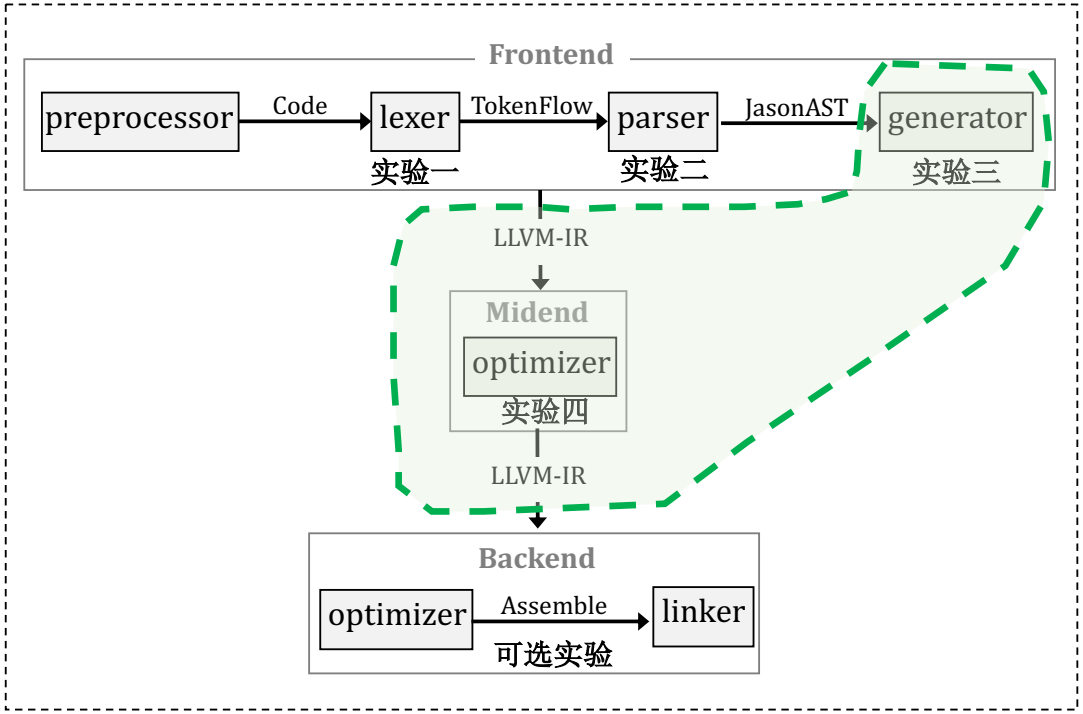
CodeGen[中间代码生成]

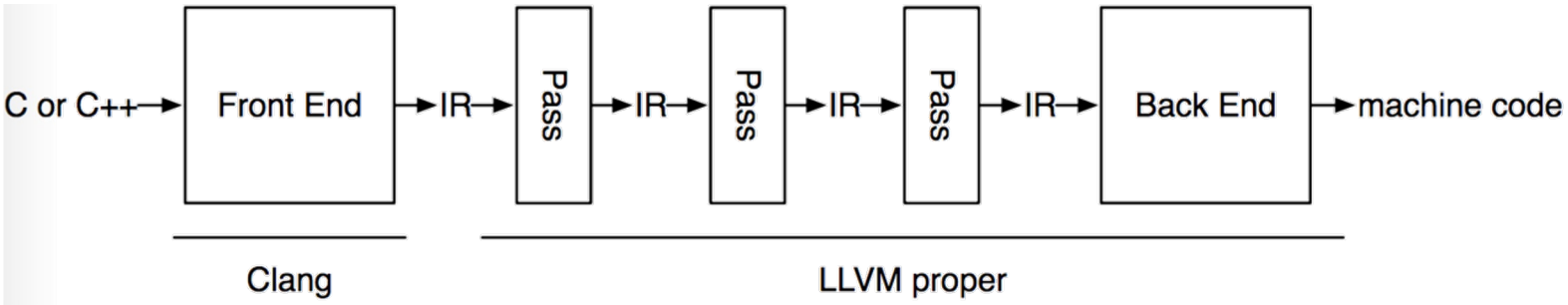


- Not to be confused with LLVM CodeGen! (which generates machine code)
- Uses AST visitors, IRBuilder, and TargetInfo
  - AST visitors
    - *RecursiveASTVisitor* for visiting the full AST
    - *StmtVisitor* for visiting Stmt and Expr
    - *TypeVisitor* for Type hierarchy

https://llvm.org/devmtg/2019-10/slides/ClangTutorial-Stulova-vanHaastregt.pdf, https://clang.llvm.org/docs/RAVFrontendAction.html

CodeGen (cont.)





Proj3: IR Generator

task/3/main.cpp

```
44      // 从 ASG 发射到 LLVM IR
45      llvm::LLVMContext ctx;
46      EmitIR emitIR(mgr, ctx);
47      auto& mod = emitIR(asg);
48      mgr.gc();
49
50      // 先把 LLVM IR 写出到文件里，再检查合不合法
51      mod.print(outFile, nullptr, false, true);
```

task/3/EmitIR.cpp

```
18    llvm::Module&
19    EmitIR::operator()(asg::TranslationUnit* tu)
20    {
21      for (auto&& i : tu->decls)
22        self(i);
23      return mMod;
24    }
```

AST → IR: Example

$clang -Xclang -ast-dump -fsyntax-only 000_main.sysu.c

```
TranslationUnitDecl 0x1d2654a8 <<invalid sloc>> <invalid sloc>
                ... cutting out internal declarations of clang ...
`-FunctionDecl 0x2cf71448 <../tester/functional/000_main.sysu.c:1:1, line:3:1> line:1:5 main 'int ()'
  `-CompoundStmt 0x2cf71560 <col:11, line:3:1>
    `-ReturnStmt 0x2cf71550 <line:2:5, col:12>
      `-IntegerLiteral 0x2cf71530 <col:12> 'int' 3
```

$clang -emit-llvm -S 000_main.sysu.c

```
; ModuleID = '../tester/functional/000_main.sysu.c'
source_filename = "../tester/functional/000_main.sysu.c"
target datalayout = "e-m:e-i8:8:32-i16:16:32-i64:64-i128:128-n32:64-S128"
target triple = "aarch64-unknown-linux-gnu"

; Function Attrs: noinline nounwind optnone
define dso_local i32 @main() #0 {
  %1 = alloca i32, align 4
  store i32 0, i32* %1, align 4
  ret i32 3
}

attributes #0 = { noinline nounwind optnone "correctly-rounded-divide-sqrt-fp-math"
="false" "disable-tail-calls"="false" "frame-pointer"="non-leaf" "less-precise-fpma
d"="false" "min-legal-vector-width"="0" "no-infs-fp-math"="false" "no-jump-tables"=
"false" "no-nans-fp-math"="false" "no-signed-zeros-fp-math"="false" "no-trapping-ma
th"="true" "stack-protector-buffer-size"="8" "target-cpu"="generic" "target-feature
s"="+neon" "unsafe-fp-math"="false" "use-soft-float"="false" }

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

!0 = !{i32 1, !"wchar_size", i32 4}
!1 = !{!"Debian clang version 11.0.1-2"}
```

AST → IR: HelloWorld

Source

```
1  int main(){
2      return 3;
3  }
```

AST

```
TranslationUnitDecl 0xa8e6558 <<invalid sloc>> <invalid sloc>
        ... cutting out internal declarations of clang ...
-FunctionDecl 0xa942a10 <generator/000_main.sysu.c:1:1, line:3:1> line:1:5 main 'int ()'
  `-CompoundStmt 0xa942b28 <col:11, line:3:1>
    `-ReturnStmt 0xa942b18 <line:2:5, col:12>
      `-IntegerLiteral 0xa942af8 <col:12> 'int' 3
```

IR

```
1  define dso_local i32 @main() {
2    %1 = alloca i32, align 4
3    store i32 0, ptr %1, align 4
4    ret i32 3
5  }
```

```
1  define dso_local i32 @main() {
2    ret i32 3
3  }
```

AST → IR: Local Variable

Source

```
1  int main(){
2      int a = 3;
3      return a;
4  }
```

```
1  int main(){
2      return 3;
3  }
```

AST

```
TranslationUnitDecl 0xb7ae558 <<invalid sloc>> <invalid sloc>
        ... cutting out internal declarations of clang ...
`-FunctionDecl 0xb80abf8 <line:6:1, line:9:1> line:6:5 main 'int ()'
  `-CompoundStmt 0xb80ad98 <col:11, line:9:1>
    |-DeclStmt 0xb80ad38 <line:7:5, col:14>
    | `-VarDecl 0xb80acb0 <col:5, col:13> col:9 used a 'int' cinit
    |   `-IntegerLiteral 0xb80ad18 <col:13> 'int' 3
    `-ReturnStmt 0xb80ad88 <line:8:5, col:12>
      `-ImplicitCastExpr 0xb80ad70 <col:12> 'int' <LValueToRValue>
        `-DeclRefExpr 0xb80ad50 <col:12> 'int' lvalue Var 0xb80acb0 'a' 'int'
```

IR

```
1  define dso_local i32 @main() {
2    %1 = alloca i32, align 4
3    store i32 3, ptr %1, align 4
4    %2 = load i32, ptr %1, align 4
5    ret i32 %2
6  }
```

临时寄存器/变量：分配栈空间，地址存入%1，大小同i32类型，4B对齐

写内存：将'3'写入%1对应的内存中，4B对齐

读内存：将%1对应的内存中的数据读取到%2中

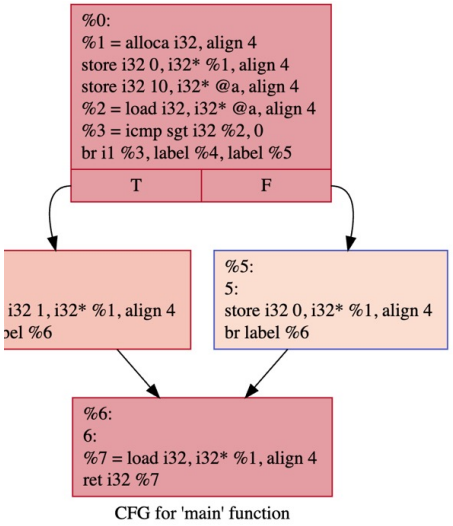
函数返回

AST → IR: Basic Blocks

- $clang -emit-llvm -S 027_if2.sysu.c

```c
int a;
int main(){
    a = 10;
    if(a > 0){
        return 1;
    }
    else{
        return 0;
    }
}
```

```
define dso_local i32 @main() {
  %1 = alloca i32, align 4
  store i32 0, ptr %1, align 4
  store i32 10, ptr @a, align 4
  %2 = load i32, ptr @a, align 4
  %3 = icmp sgt i32 %2, 0
  br i1 %3, label %4, label %5

4:                                                ; preds = %0
  store i32 1, ptr %1, align 4
  br label %6

5:                                                ; preds = %0
  store i32 0, ptr %1, align 4
  br label %6

6:                                                ; preds = %5, %4
  %7 = load i32, ptr %1, align 4
  ret i32 %7
}
```



CFG for 'main' function

http://viz-js.com/

Example: 027_if2.sysu.c

- The program
  - Global variable (int a;)
  - Variable assignment (a = 10;)
  - Binary operation (a > 0)
  - Branch (if-else)

```c
int main(){
    return 3;
}
```

```c
int main(){
    int a = 3;
    return a;
}
```



```c
int a;
int main(){
    a = 10;
    if(a > 0){
        return 1;
    }
    else{
        return 0;
    }
}
```

src → AST: Example

ir_test.c

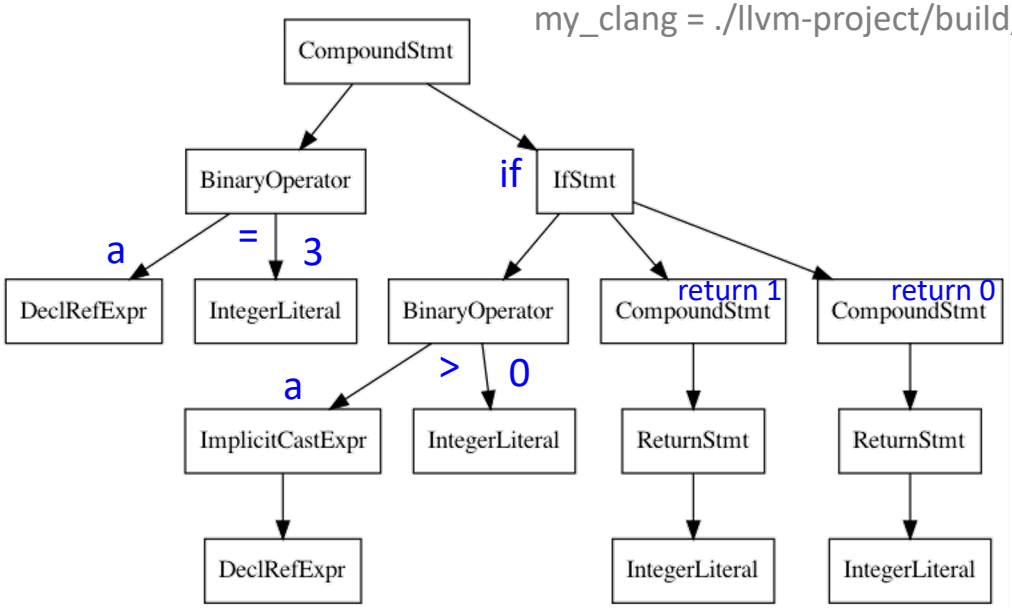
```
1 int a;
2
3 int main() {
4   a = 3;
5
6   if (a > 0) {
7     return 1;
8   } else {
9     return 0;
10  }
11 }
```

$<*my_clang*> -cc1 -ast-view ir_test.c

$dot -Tpng -o ir_test.png ir_test.dot

my_clang = ./llvm-project/build/bin/clang

AST



AST → IR: Example

$<*my_clang*> -Xclang -ast-dump -fsyntax-only ir_test.c

```
TranslationUnitDecl 0x1d2654a8 <<invalid sloc>> <invalid sloc>
... cutting out internal declarations of clang ...
|-VarDecl 0x13800f670 <ir_test.c:1:1, col:5> col:5 used a 'int'
`-FunctionDecl 0x13800f778 <line:3:1, line:11:1> line:3:5 main 'int ()'
  `-CompoundStmt 0x13800f9a8 <col:12, line:11:1>
```

$<*my_clang*> -emit-llvm -S ir_test.c

```
; ModuleID = 'ir_test.c'
source_filename = "ir_test.c"
target datalayout = "e-m:o-i64:64-i128:128-n32:64-S128"
target triple = "arm64-apple-macosx11.0.0"

@a = global i32 0, align 4

; Function Attrs: noinline nounwind optnone ssp uwtable(sync)
define i32 @main() #0 {
entry:
  %retval = alloca i32, align 4
  store i32 0, ptr %retval, align 4
  store i32 3, ptr @a, align 4
  %0 = load i32, ptr @a, align 4
  %cmp = icmp sgt i32 %0, 0
  br i1 %cmp, label %if.then, label %if.else

if.then:                                          ; preds = %entry
  store i32 1, ptr %retval, align 4
  br label %return

if.else:                                          ; preds = %entry
  store i32 0, ptr %retval, align 4
  br label %return

return:                                           ; preds = %if.else, %if.then
  %1 = load i32, ptr %retval, align 4
  ret i32 %1
}

attributes #0 = { noinline nounwind optnone ssp uwtable(sync) "frame-pointer"="non-leaf" "no-trapping-math"="true" "s
otprod,+fp-armv8,+fp16fml,+fullfp16,+lse,+neon,+ras,+rcpc,+rdm,+sha2,+sha3,+v8.1a,+v8.2a,+v8.3a,+v8.4a,+v8.5a,+v8a,+z

!llvm.module.flags = !{!0, !1, !2, !3}
!llvm.ident = !{!4}

!0 = !{i32 1, !"wchar_size", i32 4}
!1 = !{i32 8, !"PIC Level", i32 2}
!2 = !{i32 7, !"uwtable", i32 1}
!3 = !{i32 7, !"frame-pointer", i32 1}
!4 = !{!"clang version 17.0.0 (https://github.com/llvm/llvm-project.git f759275c1c8ed91f19a6b8db228115c7f75d460b)"}
```

AST → IR: Example (cont.)

```
; ModuleID = 'ir_test.c'
source_filename = "ir_test.c"
target datalayout = "e-m:o-i64:64-i128:128-n32:64-S128"
target triple = "arm64-apple-macosx11.0.0"

@a = global i32 0, align 4

; Function Attrs: noinline nounwind optnone ssp uwtable(sync)
define i32 @main() #0 {
entry:
  %retval = alloca i32, align 4
  store i32 0, ptr %retval, align 4
  store i32 3, ptr @a, align 4
  %0 = load i32, ptr @a, align 4
  %cmp = icmp sgt i32 %0, 0
  br i1 %cmp, label %if.then, label %if.else

if.then:
  store i32 1, ptr %retval, align 4
  br label %return

if.else:
  store i32 0, ptr %retval, align 4
  br label %return

return:
  %1 = load i32, ptr %retval, align 4
  ret i32 %1
}

attributes #0 = { noinline nounwind optnone ssp uwtable(sync) "frame-pointer"="non-leaf" "no-trapping-math"="true" "s
otprod,+fp-armv8,+fp16fml,+fullfp16,+lse,+neon,+ras,+rcpc,+rdm,+sha2,+sha3,+v8.1a,+v8.2a,+v8.3a,+v8.4a,+v8.5a,+v8a,+z

!llvm.module.flags = !{!0, !1, !2, !3}
!llvm.ident = !{!4}

!0 = !{i32 1, !"wchar_size", i32 4}
!1 = !{i32 8, !"PIC Level", i32 2}
!2 = !{i32 7, !"uwtable", i32 1}
!3 = !{i32 7, !"frame-pointer", i32 1}
!4 = !{!"clang version 17.0.0 (https://github.com/llvm/llvm-project.git f759275c1c8ed91f19a6b8db228115c7f75d460b)"}
```

注释

源文件名

目标平台：数据布局[1], {endian, mangling, data align, native reg size, stack align}

目标平台：arch-vendor-os

全局变量定义：@<变量名> = <可见域> <类型> 初值, 4B对齐

函数定义：define <返回类型> @<函数名>（参数）#属性[2]

```
// allocate memory for a 32b value
// store 0 in the memory slot pointed by the register
// store 3 in the memory slot of 'a'
// do comparison
// branch

 ; preds = %entry     // if (a > 0) {
// store 1 into 'retval'
// jump to return

 ; preds = %entry     // } else {
// store 0 into 'retval'
// jump to return

 ; preds = %if.else, %if.then
// load 'retval'
// return
```

```
1 int a;
2
3 int main() {
4   a = 3;
5
6   if (a > 0) {
7     return 1;
8   } else {
9     return 0;
10  }
11 }
```

函数属性

模块级别元数据信息[3]

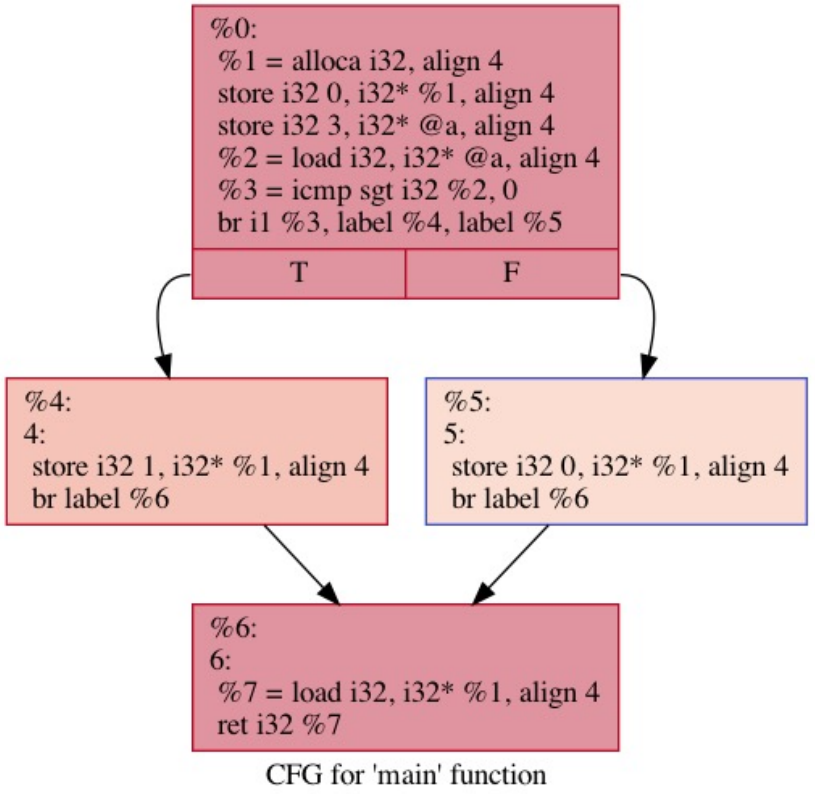
Clang版本信息

[1] https://llvm.org/docs/LangRef.html#data-layout
[2] https://llvm.org/docs/LangRef.html#function-attributes
[3] LLVM之IR 篇（1）：零基础快速入门 LLVM IR

AST → IR: Example (cont.)

$clang -emit-llvm -S ir_test.c

```
; ModuleID = 'ir_test.c'
source_filename = "ir_test.c"
target datalayout = "e-m:e-i8:8:32-i16:16:32-i64:64-i128:128-n32:64-S128
target triple = "aarch64-unknown-linux-gnu"

@a = dso_local global i32 0, align 4

; Function Attrs: noinline nounwind optnone
define dso_local i32 @main() #0 {
  %1 = alloca i32, align 4
  store i32 0, i32* %1, align 4
  store i32 3, i32* @a, align 4
  %2 = load i32, i32* @a, align 4
  %3 = icmp sgt i32 %2, 0
  br i1 %3, label %4, label %5

4:                                                ; preds = %0
  store i32 1, i32* %1, align 4
  br label %6

5:                                                ; preds = %0
  store i32 0, i32* %1, align 4
  br label %6

6:                                                ; preds = %5, %4
  %7 = load i32, i32* %1, align 4
  ret i32 %7
}

attributes #0 = { noinline nounwind optnone "correctly-rounded-divide-sqrt
" "no-infs-fp-math"="false" "no-jump-tables"="false" "no-nans-fp-math"="fa
features"="+neon" "unsafe-fp-math"="false" "use-soft-float"="false" }

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

!0 = !{i32 1, !"wchar_size", i32 4}
!1 = !{!"Debian clang version 11.0.1-2"}
```



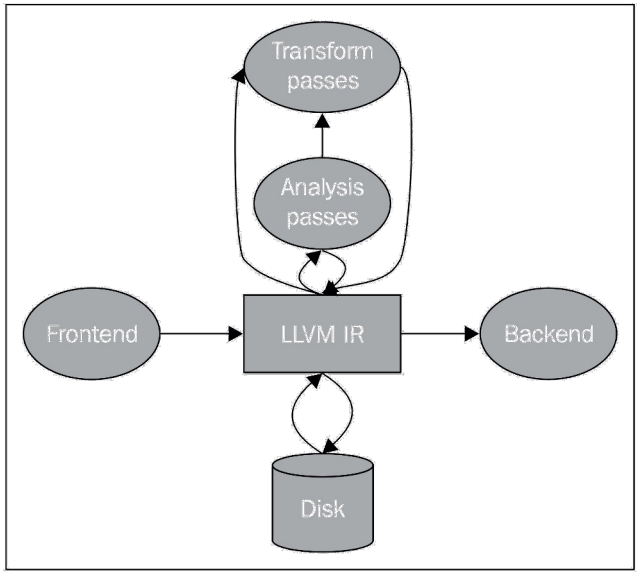
CFG for 'main' function

$dot -Tpng -o ir_test.png .ir_test.dot

$opt -dot-cfg ir_test.ll

IR Forms

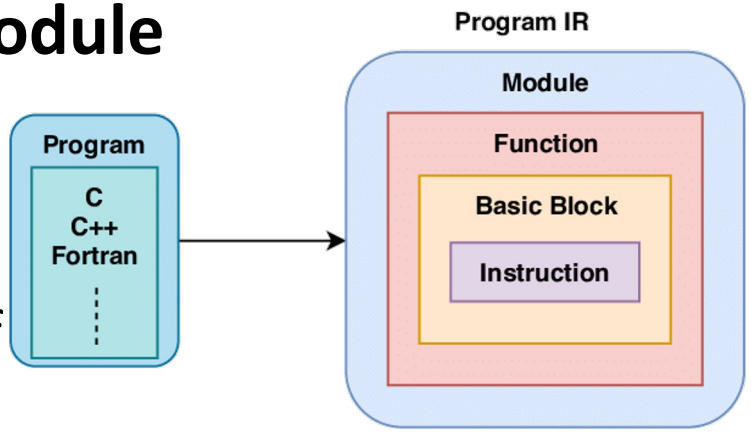
- Three different forms (these three forms are equivalent)
  - In-memory compiler IR [在内存中的编译中间语言]
  - On-disk bitcode file [.bc, 在硬盘上存储的二进制中间语言]
  - Human readable plain text file [.ll, 人类可读的代码语言]
- Translate to bitcode file[2]: $llvm-as *.ll [-o *.bc]
  - Reverse: $llvm-dis *.bc -o *.ll
  - Further compile the bitcode[3]:
    - $llc -march=x86 *.bc -o out.x86
- Execute the IR file[1]: $lli *.ll
  - Result: $echo $?

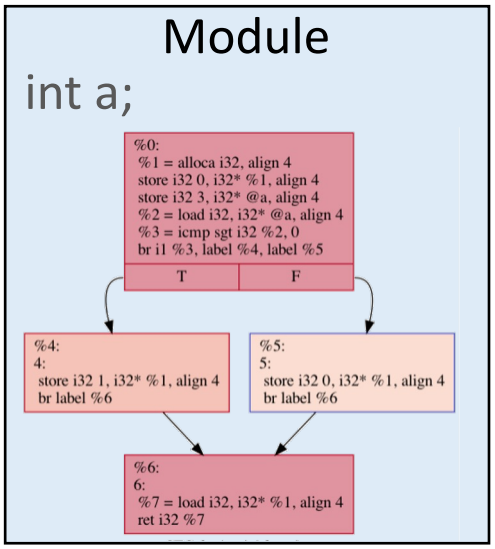


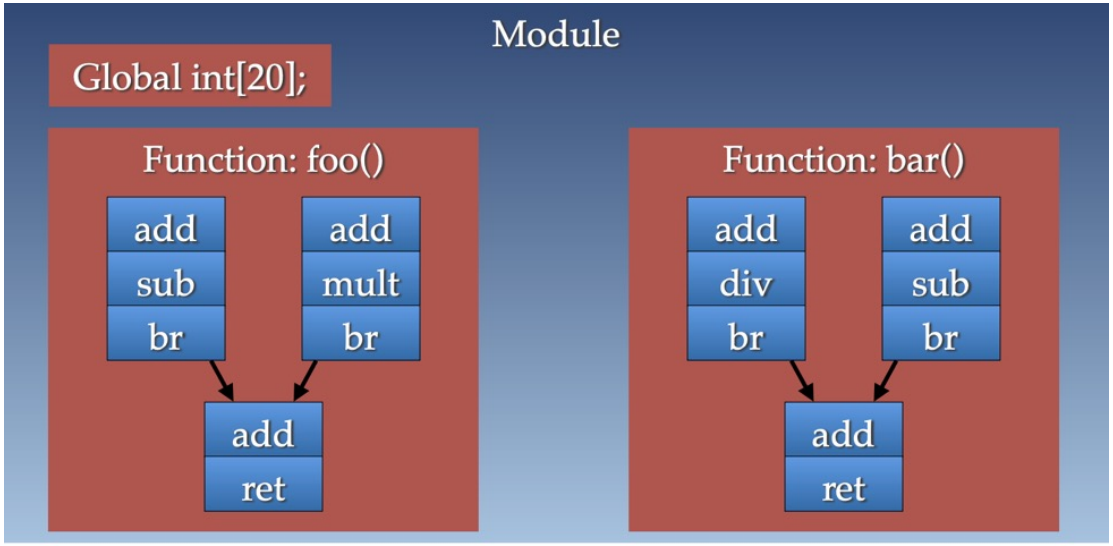
[1] https://www.llvm.org/docs/CommandGuide/lli.html
[2] https://www.llvm.org/docs/CommandGuide/llvm-as.html
[3] https://www.llvm.org/docs/CommandGuide/llc.html

IR Overview

- Each assembly/bitcode file is a **Module**
- Each Module is comprised of
  - Global variables
  - A set of **Function**s which consists of
    - A set of **Basic Blocks**
      - Which is further comprised of a set of **Instructions**







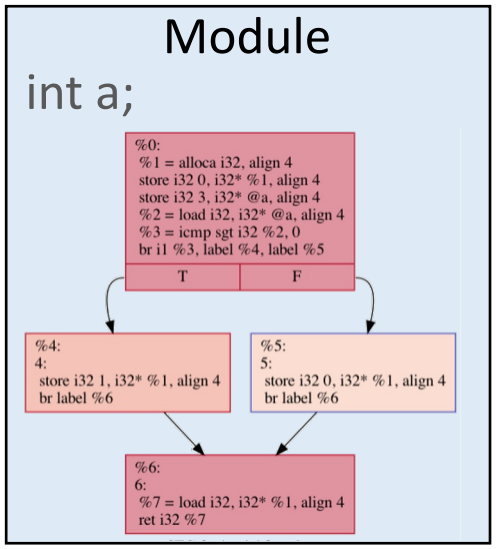
https://cs.rochester.edu/u/ejohns48/secdev19/secdev20-llvm-tutorial-version4_copy.pdf

IR Overview (cont.)

- LLVM IR resembles **three-address code** (TAC)
  - Two source operands, one separate destination operand
  - **Static Single Assignment** (SSA) form, making life easier for optimization writers[静态单赋值]
    - SSA means we define variables before use and assign to variables only once
- LLVM IR is machine independent[机器无关]
  - An unlimited set of virtual registers (labelled %0, %1, %2, …)
    - It's the backend's job to map from virtual to physical registers
  - Rather than allocating specific sizes of datatypes, we retain types
    - Again, the backend will take this type info and map it to platform's datatype

LLVM Steps

- Context: llvm:LLVMContext TheContext
- Module: llvm::Module TheModule("-", TheContext);
  - Function: auto function = llvm::Function::Create(…, TheModule)
    - auto **block** = llvm::BasicBlock::Create(**TheContext**, "entry", **function**)
      - llvm::IRBuilder<> builder(block);
        - builder.CreateRet(tmp);




```
1 int a;
2
3 int main() {
4   a = 3;
5
6   if (a > 0) {
7     return 1;
8   } else {
9     return 0;
10  }
11 }
```

```
#include <llvm/IR/IRBuilder.h>
#include <llvm/IR/LLVMContext.h>
#include <llvm/IR/Module.h>
#include <llvm/IR/Type.h>
#include <llvm/IR/Verifier.h>
#include <llvm/Support/JSON.h>
#include <llvm/Support/MemoryBuffer.h>
#include <llvm/Support/raw_ostream.h>
```

Three-Address Code[三地址码]

- High-level assembly where each operation has **at most three** operands. Generic form is X = Y op Z[最多3个操作数]
  - where X, Y, Z can be variables, constants, or compiler-generated temporaries holding intermediate values
- Characteristics[特性]
  - Assembly code for an 'abstract machine'
  - Long expressions are converted to multiple instructions
  - Control flow statements are converted to jumps[控制流->跳转]
  - Machine independent
    - Operations are generic (not tailored to any specific machine)
    - Function calls represented as generic call nodes
    - Uses symbolic names rather than register names (actual locations of symbols are yet to be determined)
- Design goal: for easier machine-independent optimization

Example

- For example, x * y + x * y is translated to

  t1 = x * y ; t1, t2, t3 are temporary variables

  t2 = x * y

  t3 = t1 + t2

  - Can be generated through a depth-first traversal of AST
  - Internal nodes in AST are translated to temporary variables

- Notice: repetition of x * y[重复]
  - Can be later eliminated through a compiler optimization called common subexpression elimination (CSE)[通用子表达式消除]

    t1 = x * y

    t3 = t1 + t1

  - Using TAC rather than AST makes it:
    - Easier to spot opportunities (just find matching RHS)
    - Easier to manipulate IR (AST is much more cumbersome)

Three-Address Statements

- Assignment statement[二元赋值]

  x = y op z

  where op is an arithmetic or logical operation (binary operation)

- Assignment statement[一元赋值]

  x = op y

  where op is an unary operation such as -, not, shift

- Copy statement[拷贝]

  x = y

- Unconditional jump statement[无条件跳转]

  goto L

  where L is label

Three-Address Statements (cont.)

- Conditional jump statement[条件跳转]

  if (x relop y) goto L

  where relop is a relational operator such as =, ≠, >, <

- Procedural call statement[过程调用]: may have too many addr

  param x_1, ..., param x_n, call F_y, n

  As an example, foo(x_1, x_2, x_3) is translated to

  param x_1

  param x_2

  param x_3

  call foo, 3

- Procedural call return statement[过程调用返回]

  return y

  where y is the return value (if applicable)

Three-Address Statements (cont.)

- Indexed assignment statement[索引]

  x = y[i]

  or

  y[i] = x

  where x is a scalar variable and y is an array variable

- Address and pointer operation statement[地址和指针]

  x = & y ; a pointer x is set to address of y

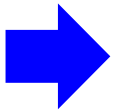
  y = * x ; y is set to the value of location

  ; pointed to by pointer x

  *y = x ; location pointed to by y is assigned x

Example: TAC

```
i = 1
do {
  a[i] = x * 5;
  i ++;
} while (i <= 10);
```



```
   i = 1
L: t1 = x * 5
   t2 = &a
   t3 = sizeof(int)
   t4 = t3 * i
   t5 = t2 + t4
   *t5 = t1
   i = i + 1
   if i <= 10 goto L
```

a[i]

Source program

Three-address code

Example: TAC (cont.)

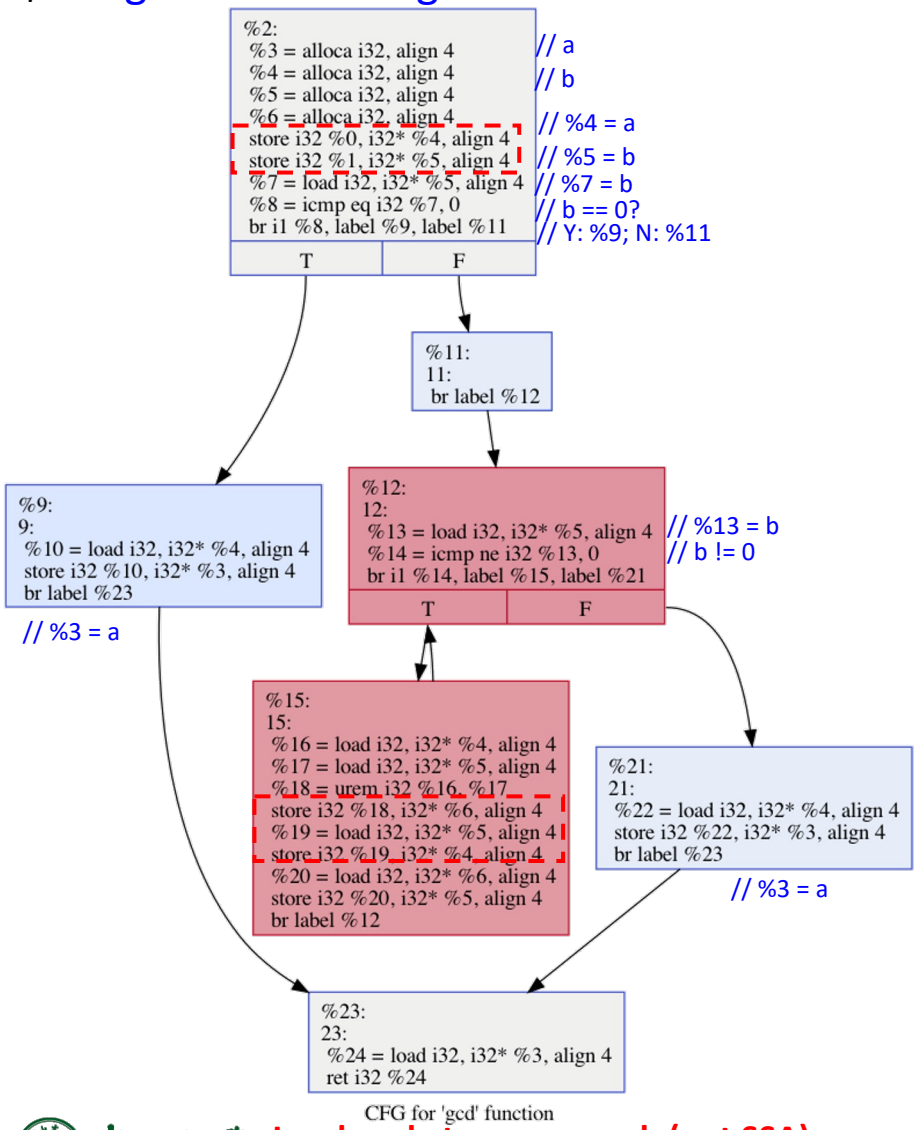
```
i = 1
do {
  a[i] = x * 5;
  i ++;
} while (i <= 10);
```

```
1  int i = 1, x = 2;
2  int a[10];
3
4  int main(){
5
6    do {
7      a[1] = x * 5;
8      i++;
9    } while (i <= 10);
10
11   return 0;
12 }
```

```
@i = dso_local global i32 1, align 4
@x = dso_local global i32 2, align 4
@a = dso_local global [10 x i32] zeroinitializer, align 4

; Function Attrs: noinline nounwind optnone
define dso_local i32 @main() #0 {
  %1 = alloca i32, align 4
  store i32 0, i32* %1, align 4
  br label %2

2:                                                ; preds = %7, %0
  %3 = load i32, i32* @x, align 4      // %3 = x
  %4 = mul nsw i32 %3, 5               // %4 = %3 x 5
  store i32 %4, i32* getelementptr inbounds ([10 x i32], [10 x i32]* @a, i64 0, i64 1), align 4
  %5 = load i32, i32* @i, align 4      // %5 = i                    // addr(@a + 0 + 1*4) = %4
  %6 = add nsw i32 %5, 1               // %6 = %5 + 1
  store i32 %6, i32* @i, align 4       // i = %6
  br label %7

7:                                                ; preds = %2
  %8 = load i32, i32* @i, align 4      // %8 = i
  %9 = icmp sle i32 %8, 10             // %9 = (i <= 10)
  br i1 %9, label %2, label %10        // T: %2, F: %10

10:                                               ; preds = %7
  ret i32 0
}
```

Example: IR and SSA

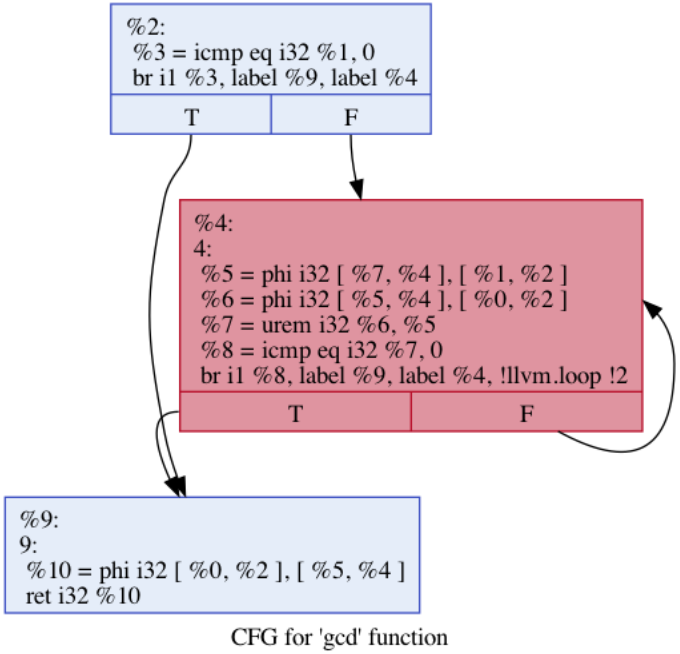
$clang -emit-llvm -S gcd.c



CFG for 'gcd' function

```
1  unsigned gcd(unsigned a, unsigned b) {
2    if (b == 0)
3      return a;
4    while (b != 0) {
5      unsigned t = a % b;
6      a = b;
7      b = t;
8    }
9    return a;
10 }
```

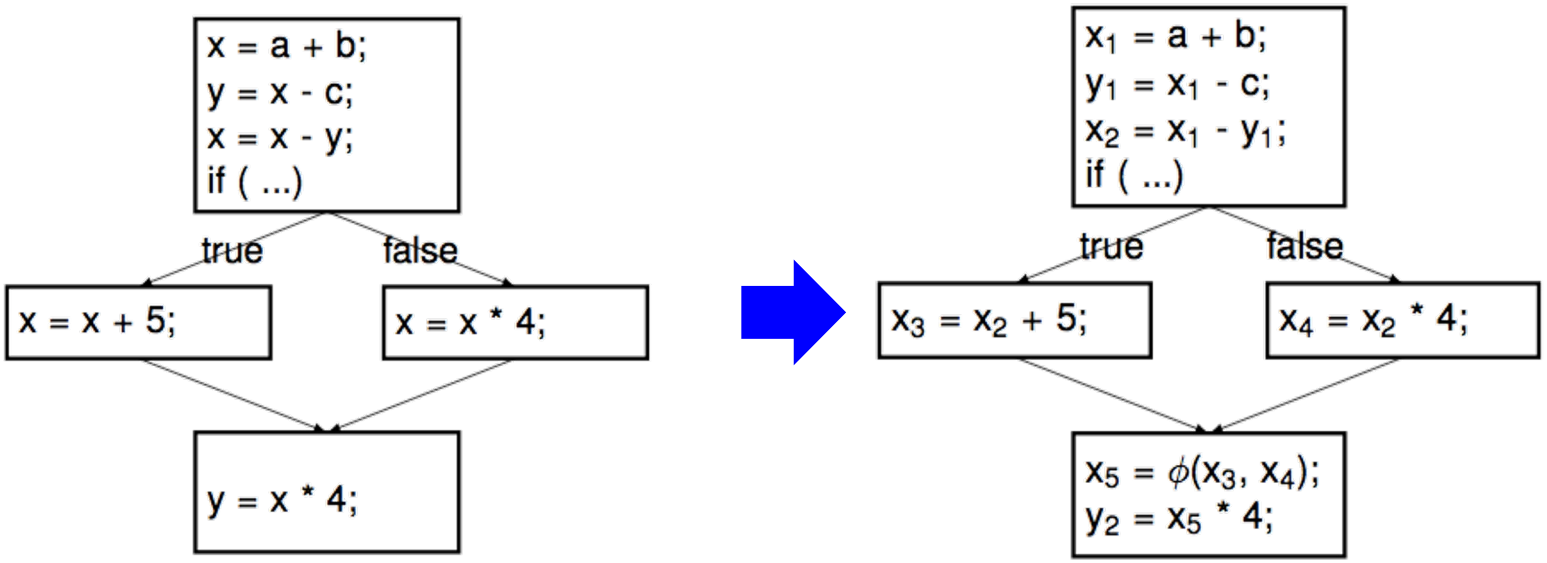
$clang -emit-llvm -S -O1 gcd.c



CFG for 'gcd' function

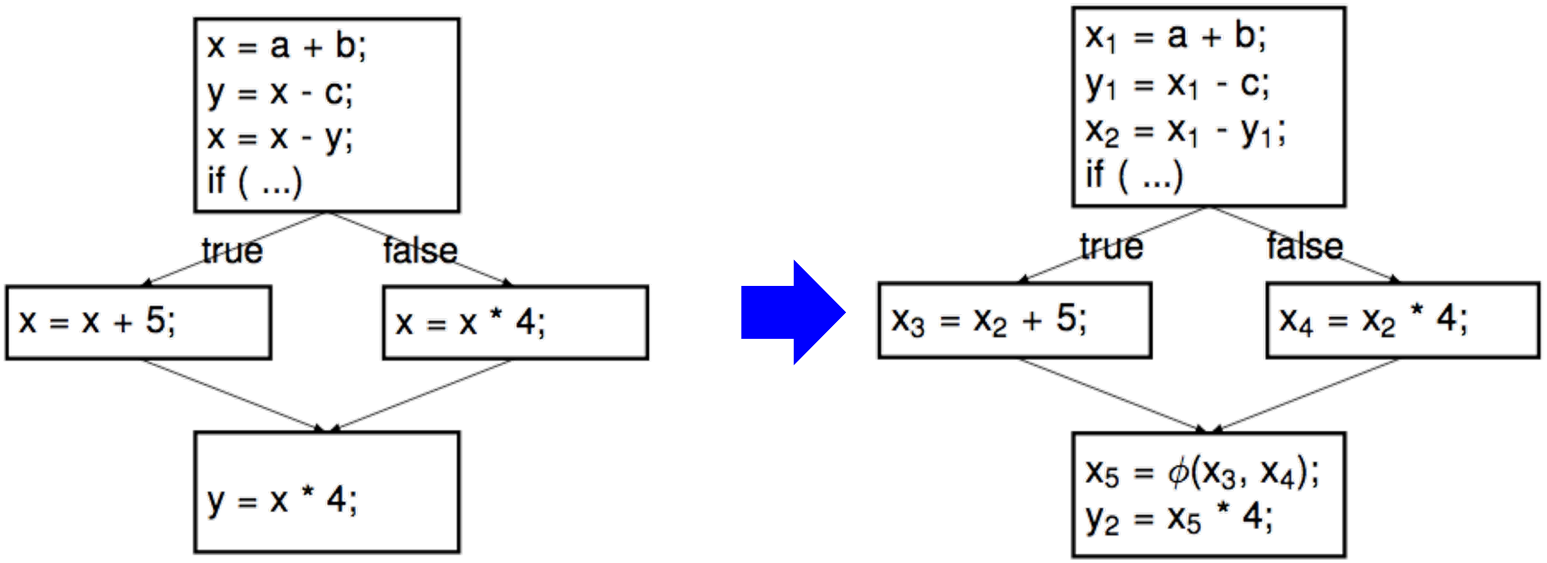
Single Static Assignment[静态单赋值]

- Every variable is assigned to exactly once statically[仅一次]
  - Give variable a different version name on every assignment
    - e.g., x → x_1, x_2, ..., x_5 for each static assignment of x
  - Now value of each variable guaranteed not to change
  - On a control flow merge, ϕ-function combines two versions
    - e.g. $x_5 = \phi(x_3, x_4)$: means x_5 is either x_3 or x_4



Benefits of SSA

- SSA is an IR that facilitates certain code optimizations
  - SSA tells you when an optimization shouldn't happen
  - Suppose compiler performs CSE on previous example:
    - Without SSA, (incorrectly) tempted to eliminate second x * 4
      - x = x * 4; y = x * 4; ➔ x = x * 4; y = x;
      - E.g., x = 5 * 4; y = 20 * 4; → x = 5 * 4; y = 20
    - With SSA, x_2 * 4 and x_5 * 4 are clearly different values



Benefits of SSA (cont.)

- SSA is an IR that facilitates certain code optimizations
  - SSA tells you when an optimization should happen
  - Suppose compiler performs dead code elimination (DCE): (DCE removes code that computes dead values)

```
x = a + b;          x1 = a + b;
x = c - d;    =>    x2 = c - d;
y = x * b;          y1 = x2 * b;
```

  - Without SSA, not very clear whether there are dead values
  - With SSA, x_1 is never used and clearly a dead value
- Why does SSA work so well with compiler optimizations?
  - SSA makes flow of values explicit in the IR[数据流显现]
  - Without SSA, need a separate dataflow graph
  - Will discuss more in **Compiler Optimization** section