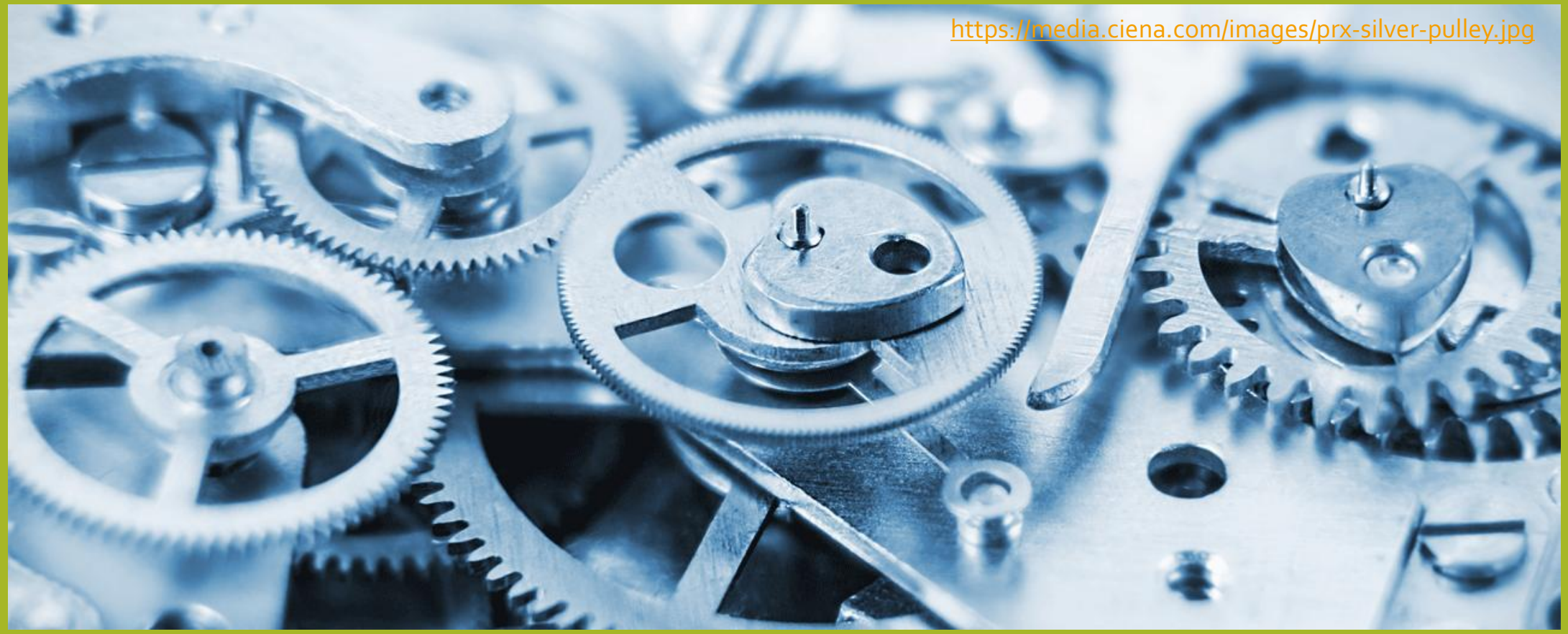

https://media.ciena.com/images/prx-silver-pulley.jpg

# C程序的汇编代码生成

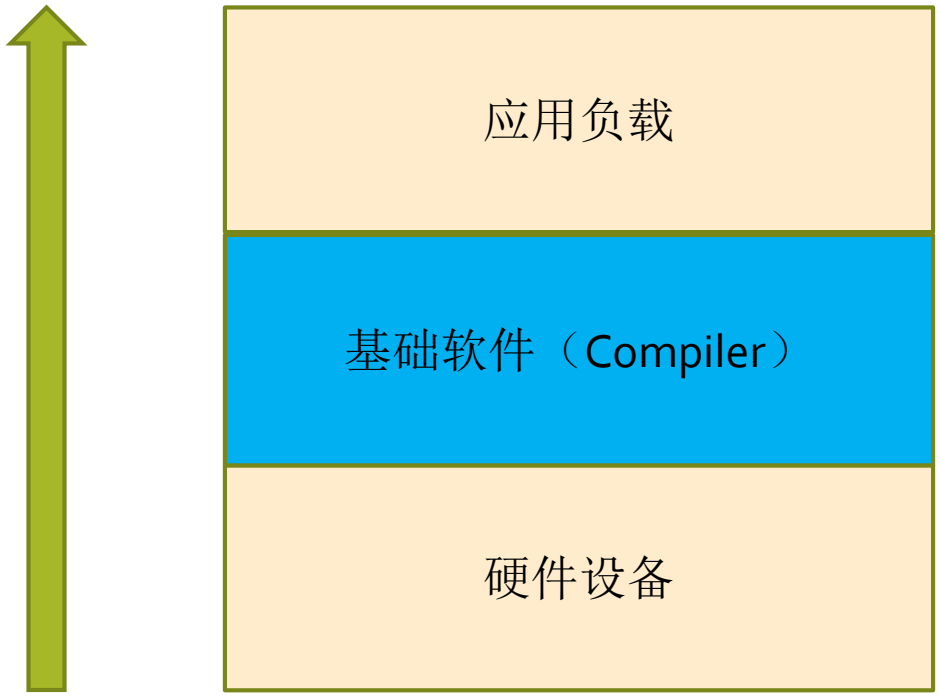
黄 波
bhuang@dase.ecnu.edu.cn

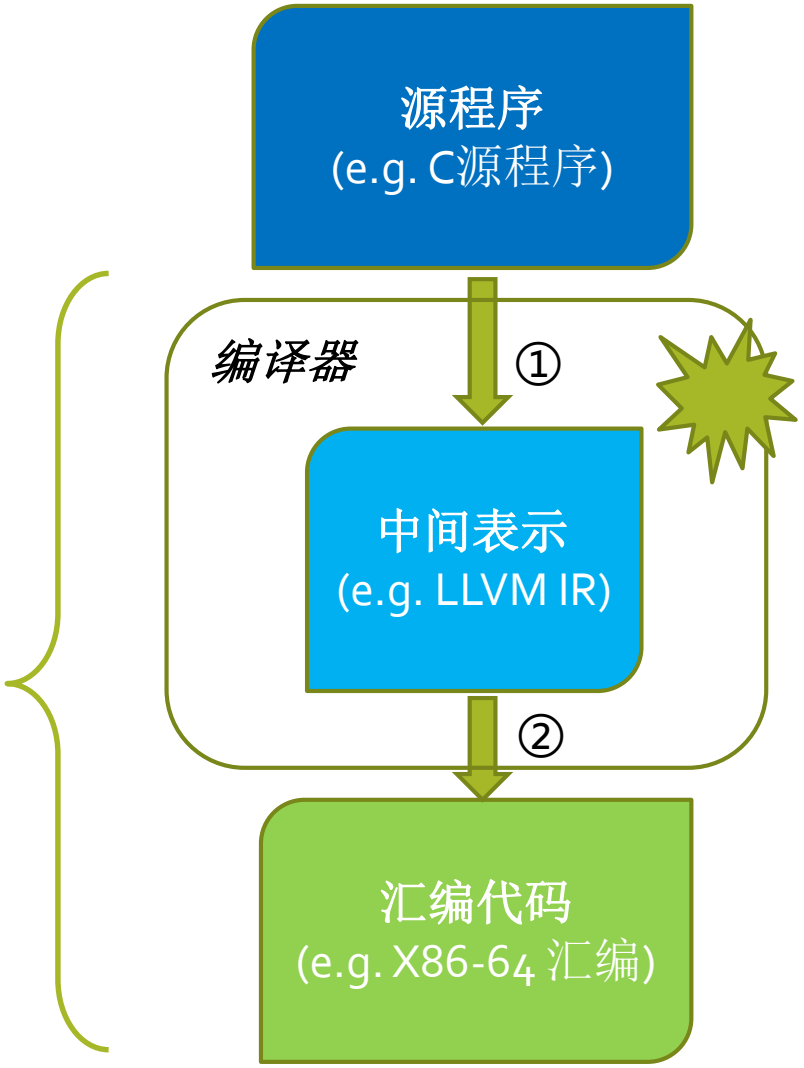
SOLE 系统优化实验室
华东师范大学

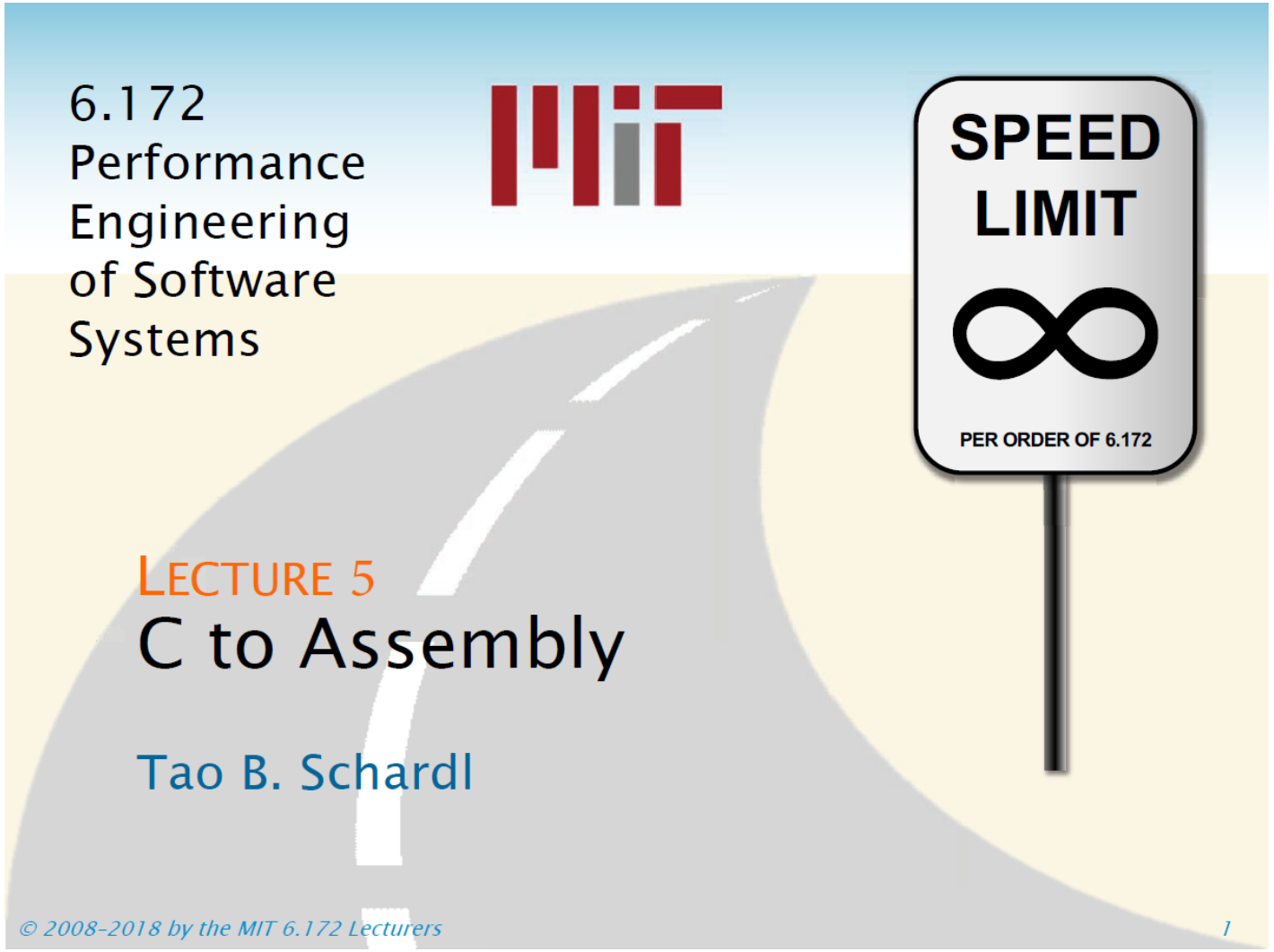
** Recompiled from MIT's OCW*

# 本次课的关注点

**Scale up**
**全栈思维**

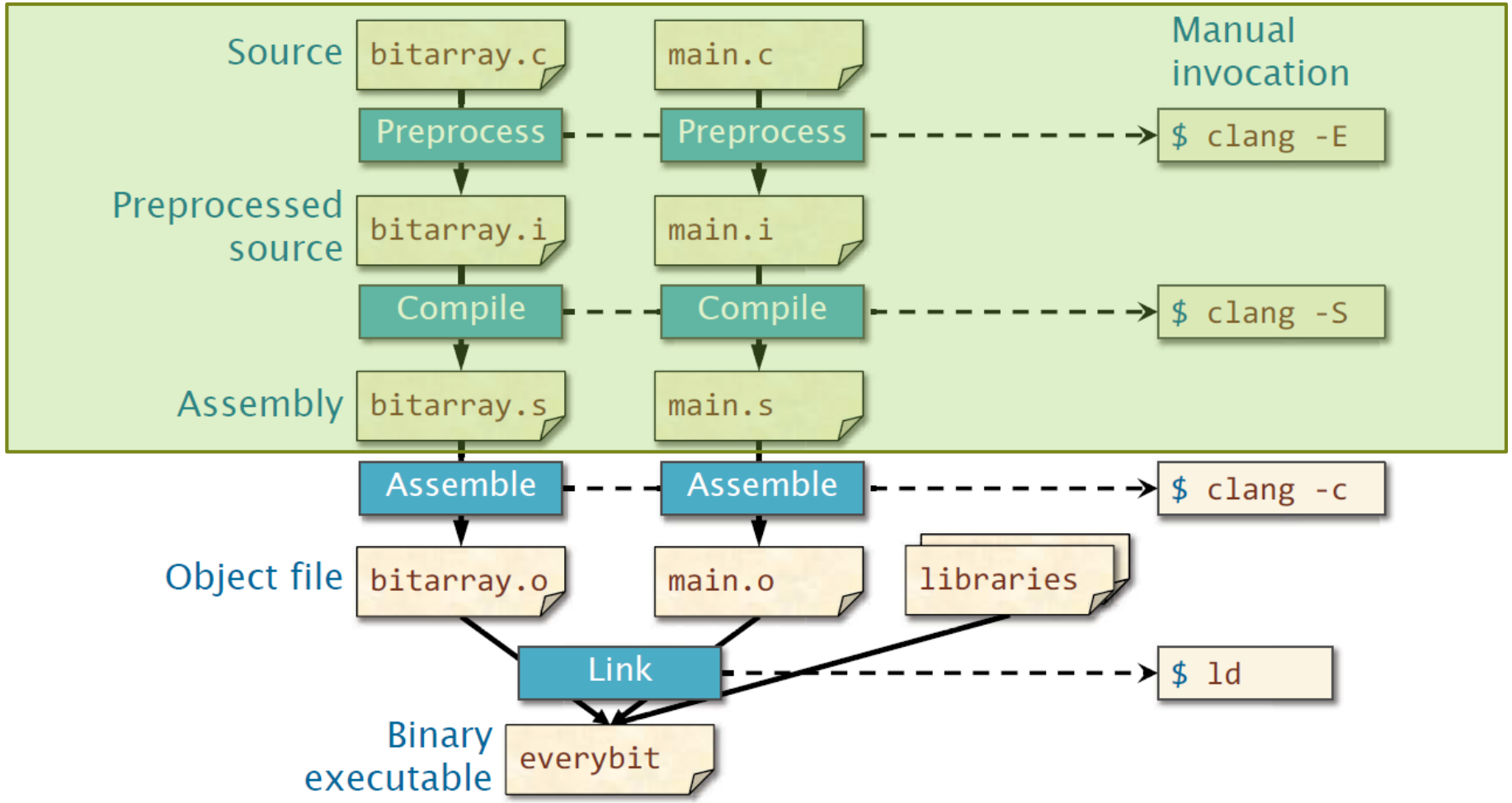




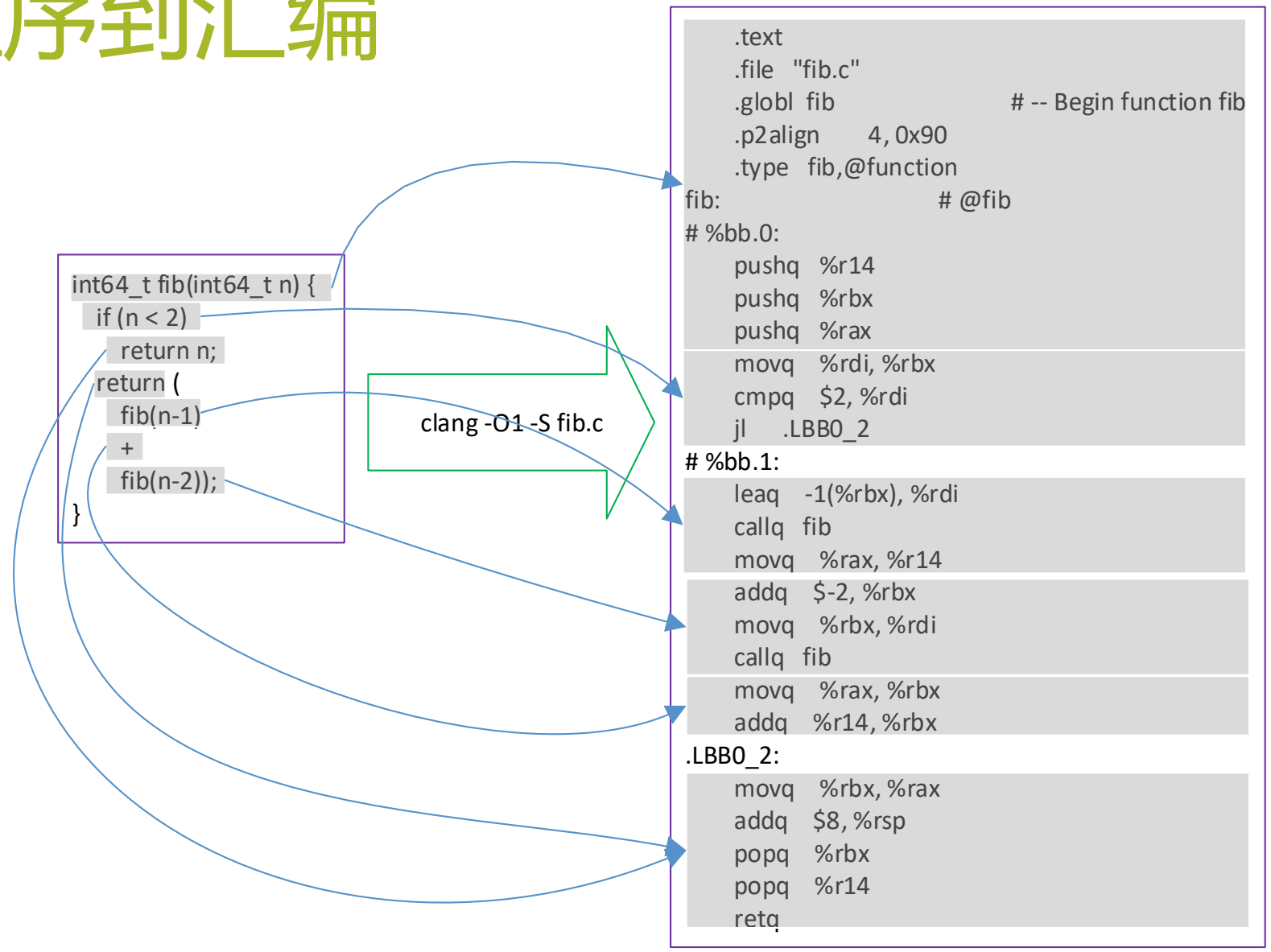

https://ocw.mit.edu/courses/electrical-engineering-and-computer-science/6-172-performance-engineering-of-software-systems-fall-2018/lecture-slides/MIT6_172F18_lec5.pdf

# “编译”过程的4个阶段


Source
bitarray.c
main.c
Manual invocation
Preprocess
Preprocess
$ clang -E
Preprocessed source
bitarray.i
main.i
Compile
Compile
$ clang -S
Assembly
bitarray.s
main.s
Assemble
Assemble
$ clang -c
Object file
bitarray.o
main.o
libraries
Link
$ ld
Binary executable
everybit

# 从的C源程序到汇编

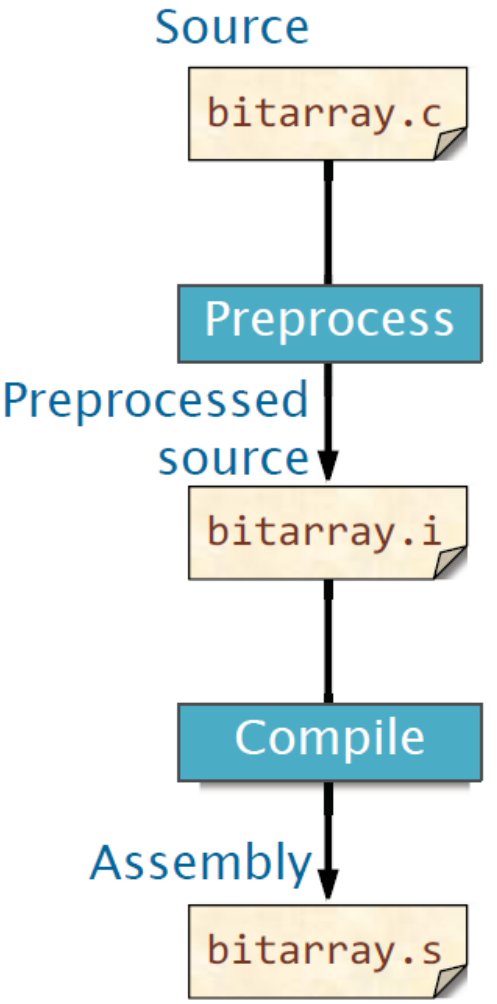
**不同的编译器或者同一编译器的不同编译选项会生成不同的汇编代码！**



# C程序如何被编译成汇编代码的?

编译器在把C程序编译成汇编代码的过程中需要做大量的工作，常见的有：

- 选择合适的寄存器和内存位置来存储程序中的数据
- 在满足数据依赖和正确性的前提下，在寄存器和内存之间进行数据移动
- 选择合适的汇编指令来实现C程序中的语句和运算
- 用相应的跳转指令来实现C程序中的条件分支语句及各类循环控制
- 正确处理调用函数和被调用函数之间的协同
- 尝试各种代码优化方案，让生成的汇编代码运行得更快




**正确性 + 性能**

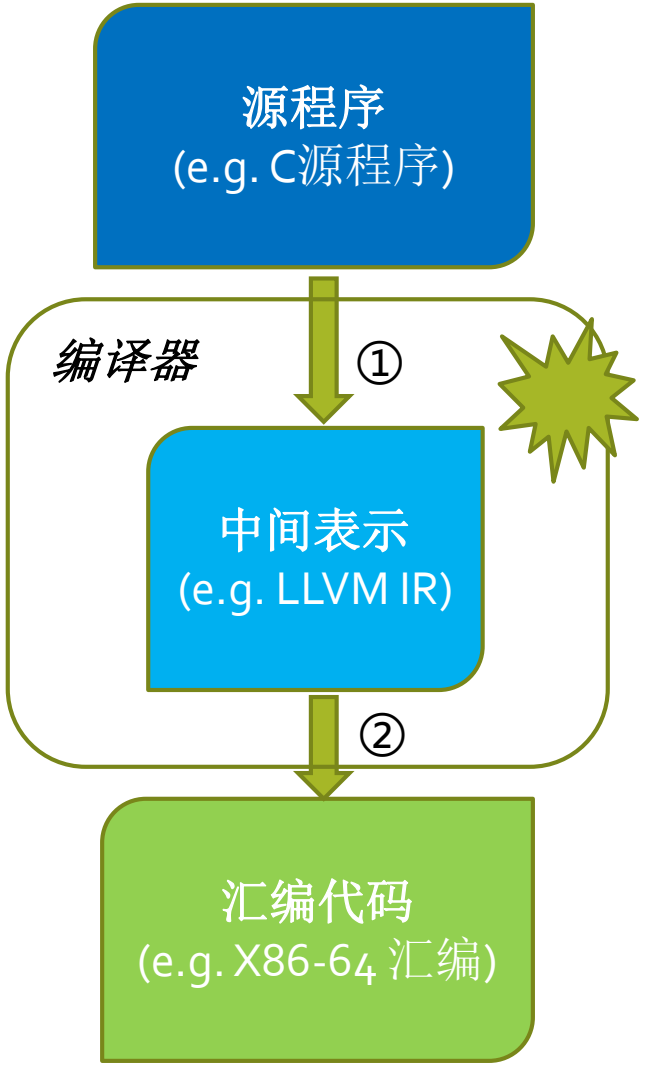
# 本次课程提纲

**C程序转换成LLVM IR （①）**

- 直线代码到LLVM IR的转换
- C函数到LLVM IR的转换
- 条件分支语句到LLVM IR的转换
- 循环语句到LLVM IR的转换
- LLVM IR中的属性简介

**LLVM IR转换成汇编程序（②）**

- 汇编制导指令与程序的内存布局
- 函数调用规范



# 常见的LLVM IR指令

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

Q: 什么是phi指令?

# 直线代码到LLVM IR的转换

C程序中的直线代码指的是不含条件分支语句和循环语句的一串代码. .

```
int f(int a, int b, int c) {
    int Result;
    Result = a+b;
    Result = Result*c + a;
    Result = c*b+Result;
    return Result;
}
```

```
define dso_local i32 @f(i32 %0, i32 %1, i32 %2) local_unnamed_addr #0 {
  %4 = shl i32 %1, 1
  %5 = add i32 %4, %0
  %6 = mul i32 %5, %2
  %7 = add i32 %6, %0
  ret i32 %7
}
```

*clang -O1 -S -emit-llvm*

# 聚合数据类型

如果C程序中的某个变量的类型是聚合数据类型（比如数组和结构），编译器通常把这个变量存储在内存中。

访问方式：编译器会生成计算地址的指令，然后根据计算出来的地址进行相应的内存读/写

在LLVM IR中，getelementptr是专门用于计算内存地址的指令，简称GEP指令：

<result> = getelementptr <ty>, <ty>* <ptrval>, {<ty> <index>}*

- 第一个 <ty> 表示第一个索引所指向的类型
- 第二个 <ty> 表示后面的指针基址 <ptrval> 的类型
- <ty> <index> 表示一组索引的类型和值

**索引的类型和索引指向的基本类型是不一样的，索引的类型一般为 i32 或 i64，而索引指向的基本类型确定的是增加索引值时指针的偏移量*

https://llvm.org/docs/GetElementPtr.html

# getelementptr 示例

```
int A[7];
int f(int x)
{
    return A[x];
}
```
(a)

```
define dso_local i32 @f(i32 %0) local_unnamed_addr #0 {
  %2 = sext i32 %0 to i64
  %3 = getelementptr inbounds [7 x i32], [7 x i32]* @A, i64 0, i64 %2
  %4 = load i32, i32* %3, align 4, !tbaa !2
  ret i32 %4
}
```

&A[0]+4x

```
struct S {
    char c;
    int B[4][10];
};
int f(struct S *A, int x) {
    return A->B[x][3];
}
```
(b)

```
%struct.S = type { i8, [4 x [10 x i32]] }
define dso_local i32 @f(%struct.S* nocapture readonly %0, i32 %1)
local_unnamed_addr #0 {
  %3 = sext i32 %1 to i64
  %4 = getelementptr inbounds %struct.S, %struct.S* %0, i64 0, i32 1, i64 %3, i64 3
  %5 = load i32, i32* %4, align 4, !tbaa !2
  ret i32 %5
}
```

Q: 第二段LLVM IR中的那条getelementptr指令应该如何解读?

# C函数到LLVM IR的转换

## LLVM IR中的函数跟C程序中的函数非常相似

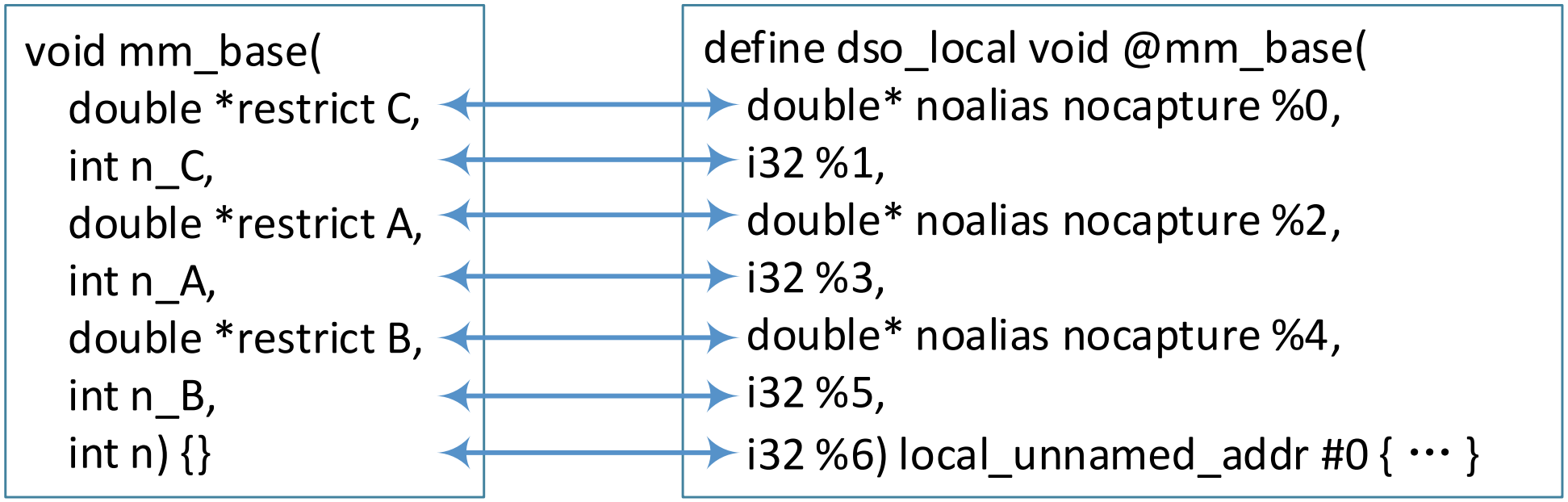
- LLVM IR中的函数声明和定义非常类似于C中的函数声明与定义
- 在函数定义中，LLVM IR的ret指令也跟C程序中的return语句相似，起到终止当前函数运行的效果

```
int f(int a, int b, int c) {
    int Result;
    Result = a+b;
    Result = Result*c + a;
    Result = c*b+Result;
    return Result;
}
```

```
define dso_local i32 @f(i32 %0, i32 %1, i32 %2) local_unnamed_addr #0 {
  %4 = shl i32 %1, 1
  %5 = add i32 %4, %0
  %6 = mul i32 %5, %2
  %7 = add i32 %6, %0
  ret i32 %7
}
```

# LLVM IR函数参数

- LLVM IR函数参数也可以直接映射到C程序中的函数参数，其中函数参数依次命名为%0、%1、%2等




Q: 这里LLVM IR中的noalias与nocapture是什么意思?

# 基本块

与C函数体对应的LLVM IR函数体中，整个函数体被划分成多个基本块（basic block），每个基本块由一系列的LLVM IR指令构成，控制流只能从一个基本块的第一条指令进来，并从一个基本块的最后一条指令离开

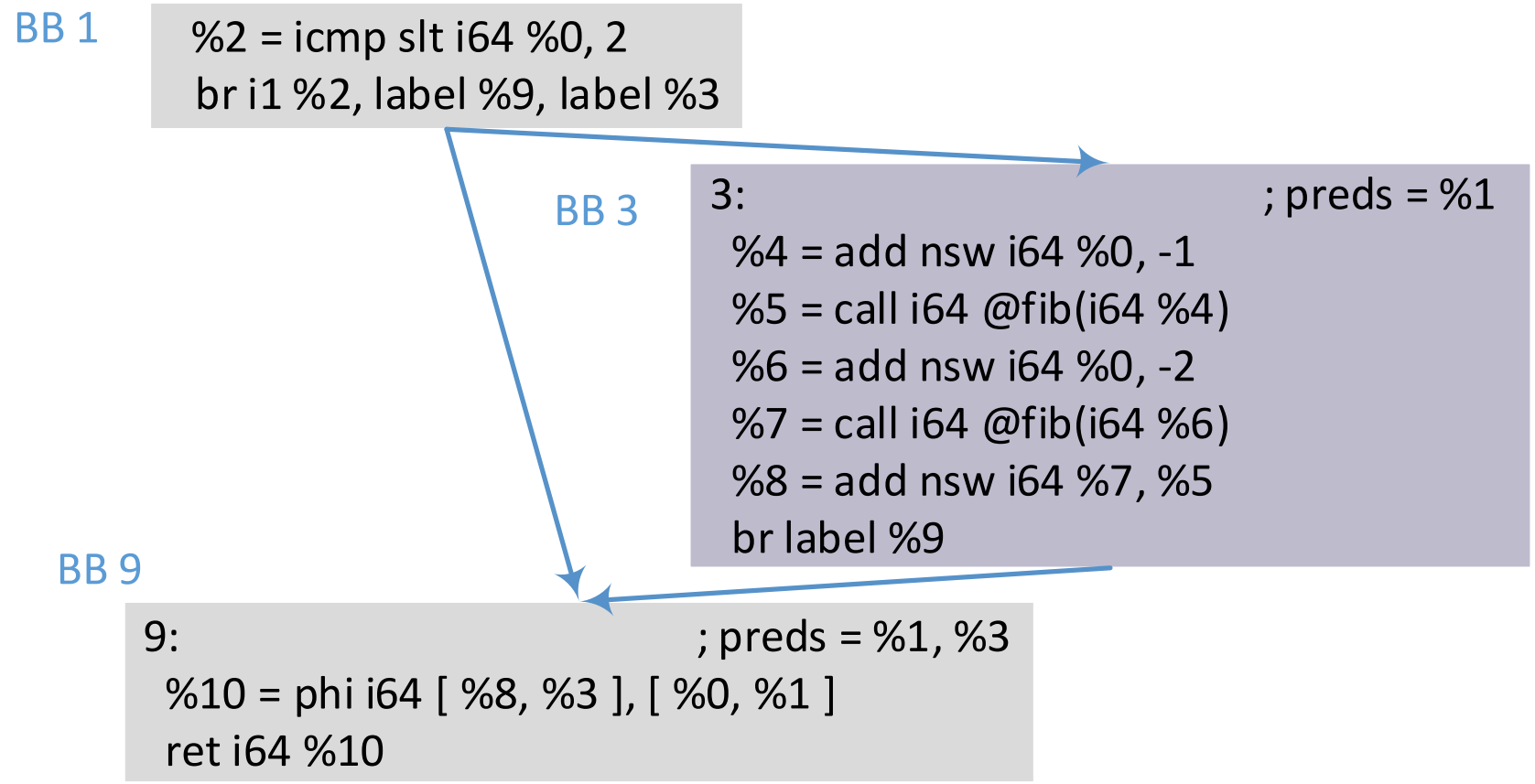
```
int64_t fib(int64_t n) {
  if (n < 2)
        return n;
  return (fib(n-1) + fib(n-2));
}
```

```
; Function Attrs: nounwind readnone uwtable
define dso_local i64 @fib(i64 %0) local_unnamed_addr #0 {
  %2 = icmp slt i64 %0, 2
  br i1 %2, label %9, label %3

3:                                                ; preds = %1
  %4 = add nsw i64 %0, -1
  %5 = call i64 @fib(i64 %4)
  %6 = add nsw i64 %0, -2
  %7 = call i64 @fib(i64 %6)
  %8 = add nsw i64 %7, %5
  br label %9

9:                                                ; preds = %1, %3
  %10 = phi i64 [ %8, %3 ], [ %0, %1 ]
  ret i64 %10
}
```

*clang -O1 -S -emit-llvm fib.c*

# 控制流图（CFG）

函数体内的不同基本块之间可能被每个基本块的最后一条跳转指令连接。如果以基本块作为图的节点，基本块之间的连接作为图的边，就形成了控制流图。



# C中的条件分支语句

C程序中的一个条件判断语句被转换成LLVM IR中的一条条件跳转指令

```
int64_t fib(int64_t n) {
   if (n < 2)
          return n;
   return (fib(n-1) + fib(n-2));
}
```

```
; Function Attrs: nounwind readnone uwtable
define dso_local i64 @fib(i64 %0) local_unnamed_addr #0 {
  %2 = icmp slt i64 %0, 2
  br i1 %2, label %9, label %3

3:                                                ; preds = %1
  ...

  ...

9:                                                ; preds = %1, %3
  ...
}
```

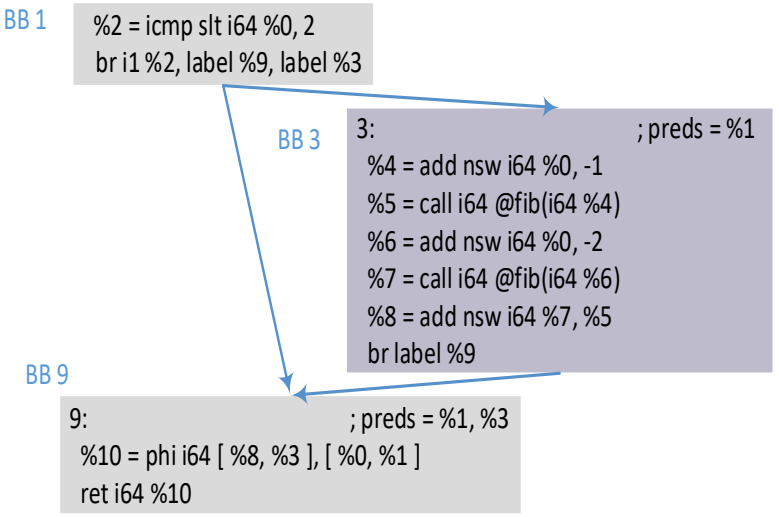
# LLVM IR中的条件跳转指令

LLVM IR中的条件跳转指令有三个参数：1位整数谓词+2个基本块标号

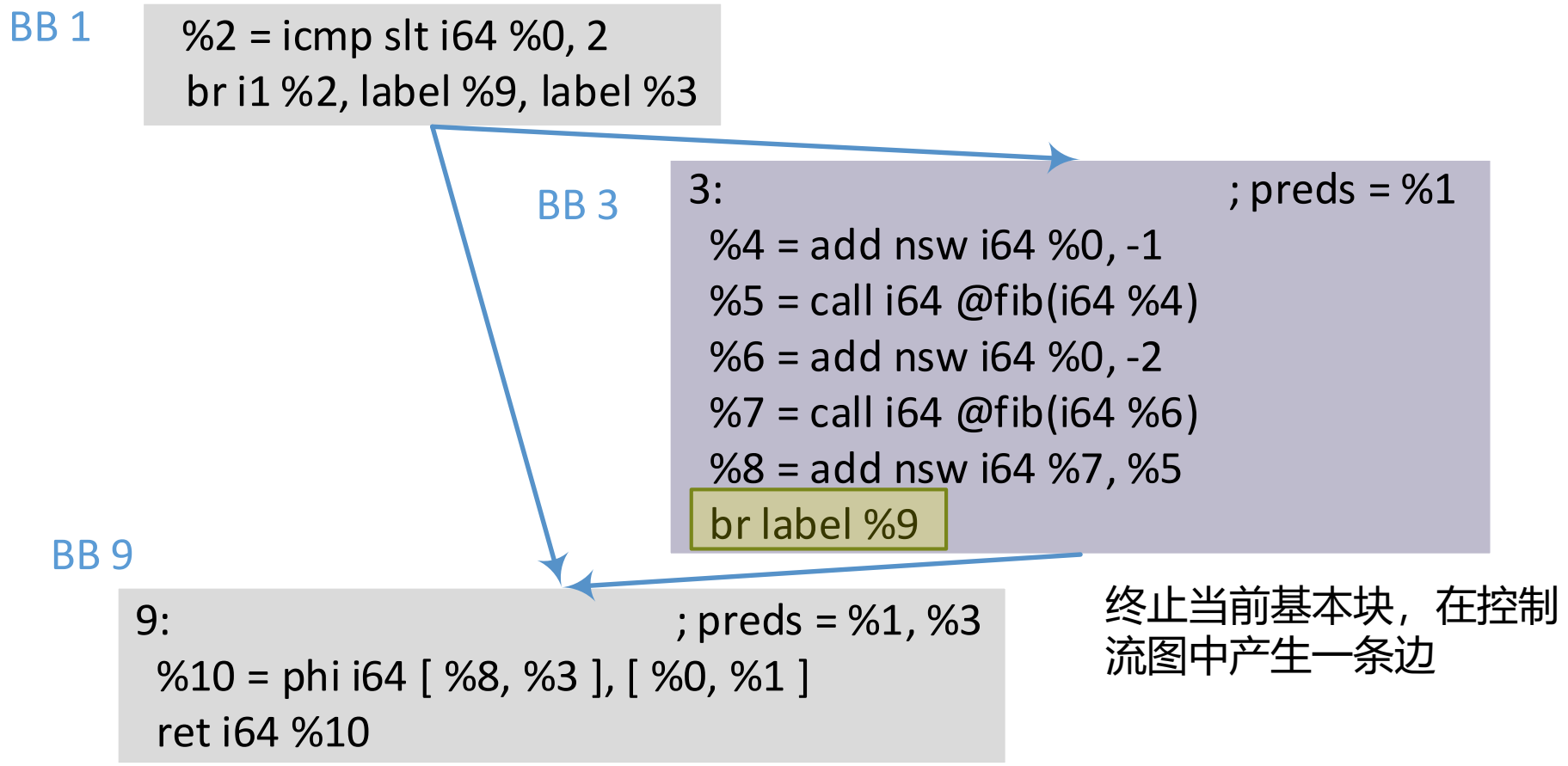
```
; Function Attrs: nounwind readnone uwtable
define dso_local i64 @fib(i64 %0) local_unnamed_addr #0 {
  %2 = icmp slt i64 %0, 2
  br i1 %2, label %9, label %3

3:                                                ; preds = %1
  ...
  ...

9:                                                ; preds = %1, %3
  ...
}
```

%2 == 0

%2 == 1

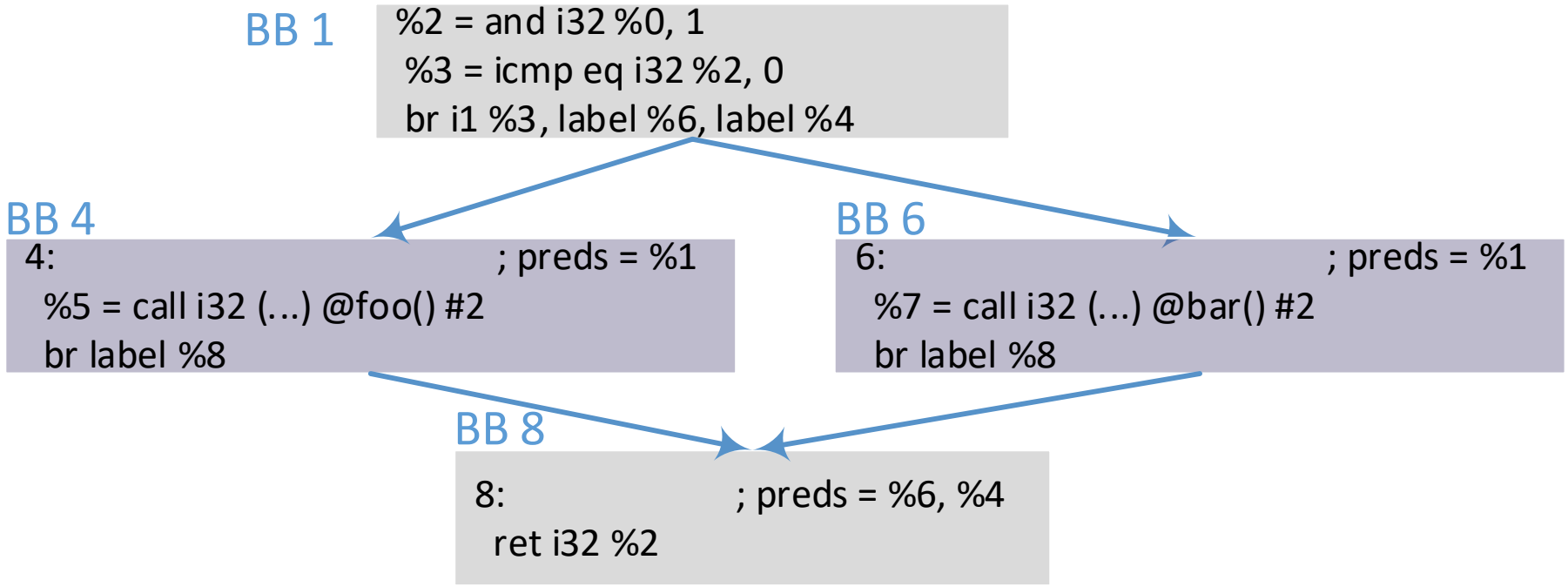


# LLVM IR中的无条件跳转指令



# 标准条件分支语句的控制流图模式

```
int baz(int x){
    if (x & 1) {
        foo();
    } else {
        bar();
    }
    return (x & 1);
}
```



# for循环到LLVM IR的变换

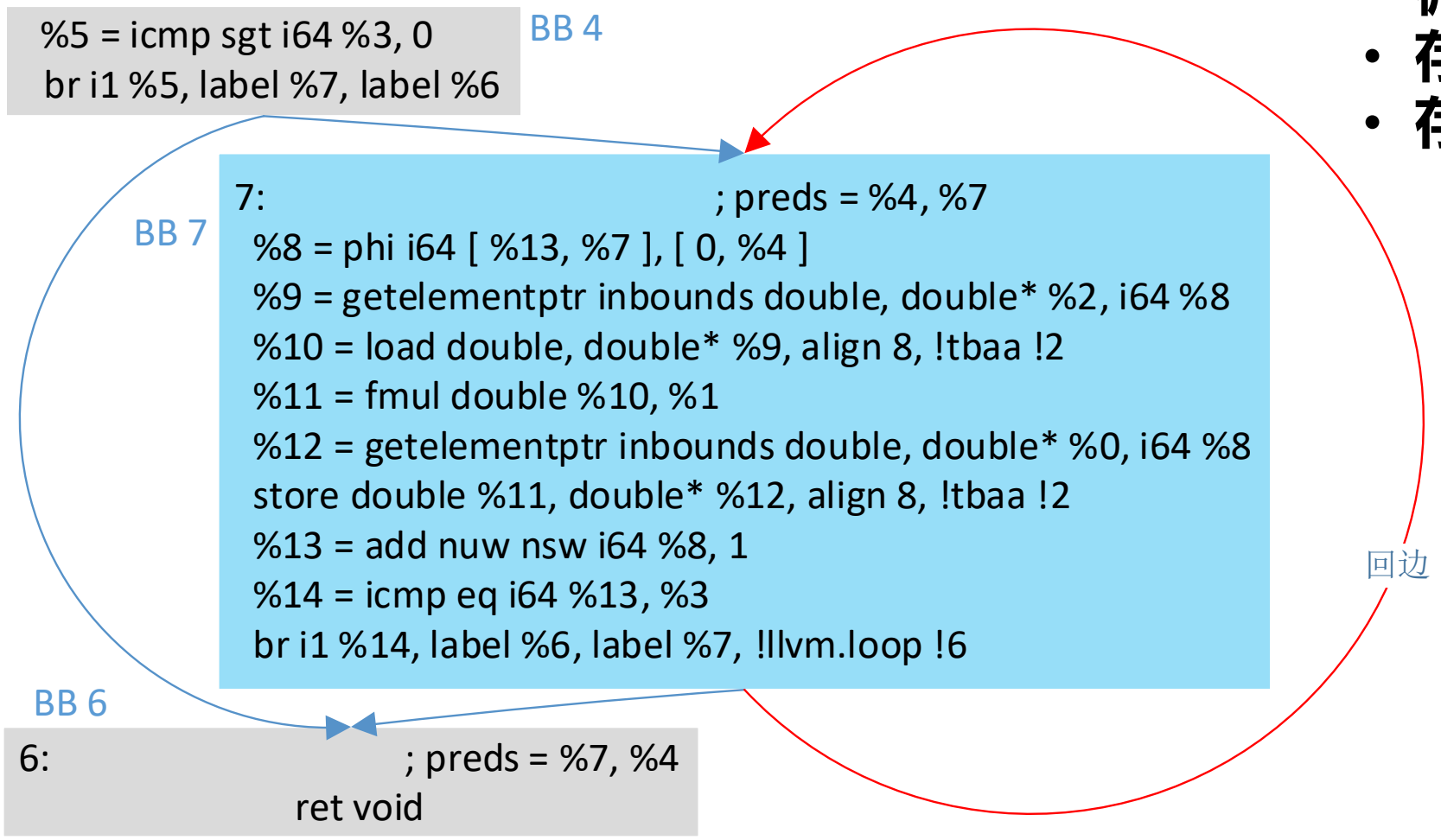
```
void dax(
    double *restrict y, double a,
    const double *restrict x,
    int64_t n){
    for (int64_t i = 0; i < n; ++i)
        y[i] = a * x[i];
}
```

循环体

循环控制

```
define dso_local void @dax(double* noalias nocapture %0, double %1,
double* noalias nocapture readonly %2, i64 %3) local_unnamed_addr #0 {
  %5 = icmp sgt i64 %3, 0
  br i1 %5, label %7, label %6

6:                                                ; preds = %7, %4
  ret void

7:                                                ; preds = %4, %7
  %8 = phi i64 [ %13, %7 ], [ 0, %4 ]
  %9 = getelementptr inbounds double, double* %2, i64 %8
  %10 = load double, double* %9, align 8, !tbaa !2
  %11 = fmul double %10, %1
  %12 = getelementptr inbounds double, double* %0, i64 %8
  store double %11, double* %12, align 8, !tbaa !2
  %13 = add nuw nsw i64 %8, 1
  %14 = icmp eq i64 %13, %3
  br i1 %14, label %6, label %7, !llvm.loop !6
}
```

# 控制流图中的循环

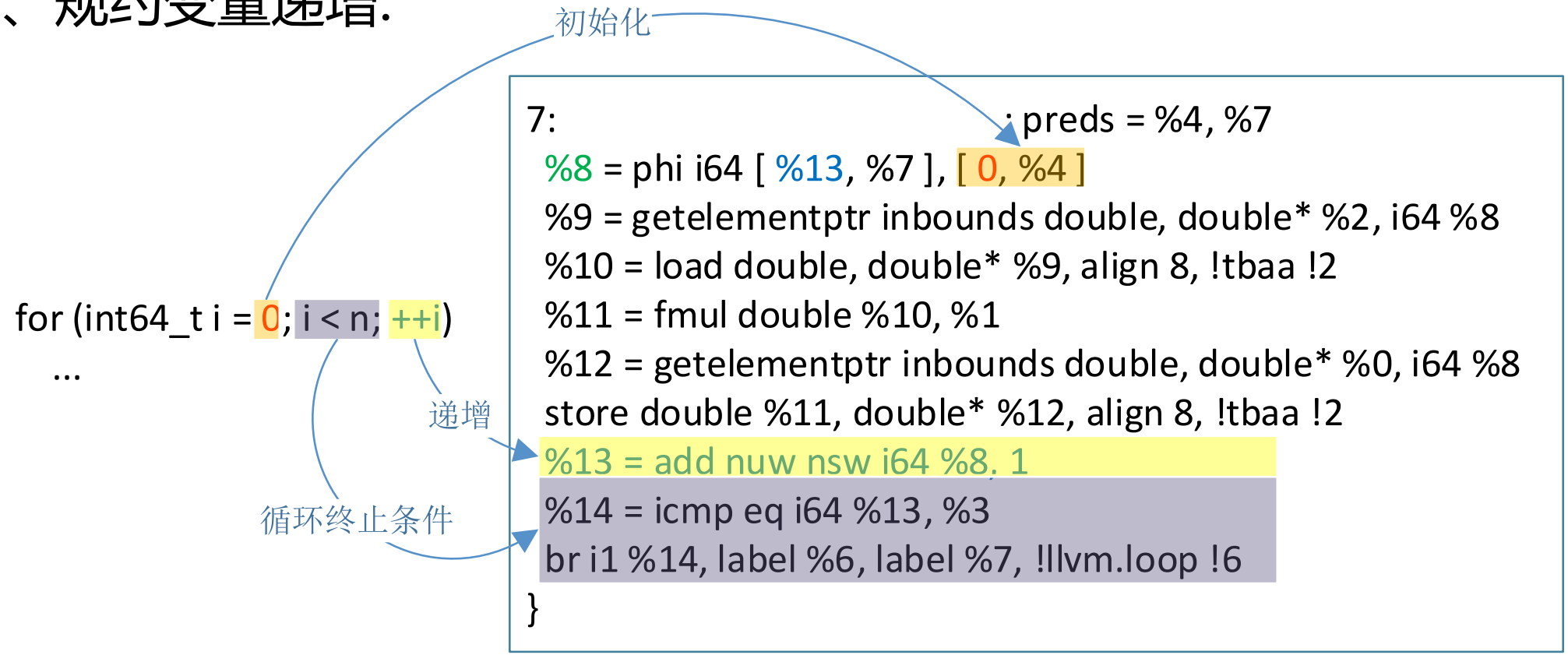



- **循环体一般有两个入口**
- **存在一个循环出口点**
- **存在回边（back edge）**

Q: 什么是回边?

# 循环控制

for循环控制：循环规约变量、规约变量初始化、循环终止条件判断、规约变量递增.




Q: 哪个虚拟寄存器对应的是循环规约变量?

# 循环规约变量

规约变量在递增时改变了寄存器!

```
for (int64_t i = 0; i < n; ++i)
  ...
```

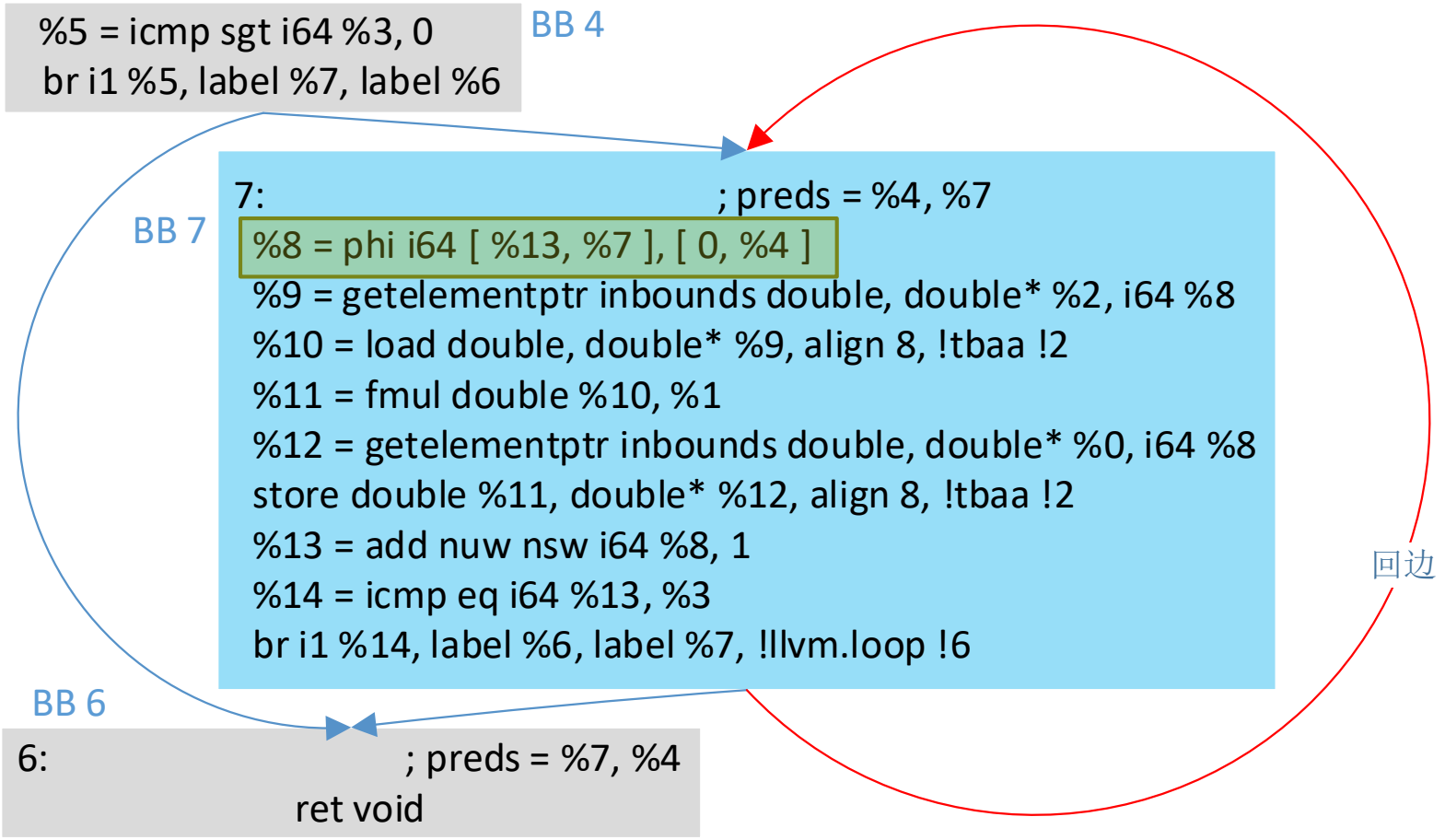
递增

```
7:                                                ; preds = %4, %7
  %8 = phi i64 [ %13, %7 ], [ 0, %4 ]
  %9 = getelementptr inbounds double, double* %2, i64 %8
  %10 = load double, double* %9, align 8, !tbaa !2
  %11 = fmul double %10, %1
  %12 = getelementptr inbounds double, double* %0, i64 %8
  store double %11, double* %12, align 8, !tbaa !2
  %13 = add nuw nsw i64 %8, 1
  %14 = icmp eq i64 %13, %3
  br i1 %14, label %6, label %7, !llvm.loop !6
}
```

Q:为什么需要改变寄存器?

# 静态单赋值（SSA）

任何一个LLVM IR中的寄存器只能最多在一条指令中被定义。



# phi指令

- phi指令其实是一个实现值选取的指令；
- 对于一个特定的基本块B以及B内的一个LLVM IR寄存器%r，根据进入B的前序基本块的不同而对%r选取不同的值；
- 只有当一个基本块具有多个前序基本块时才需要phi指令；
- phi指令不是一条真实存在的指令，在最后代码生成时编译器需要根据具体的目标指令集架构在合适的位置上把phi指令转换成一条或多条目标指令集架构上的真实指令。

# LLVM IR中的属性(Attributes)

LLVM IR中的各语言结构（比如指令、操作数、函数、函数参数等）可能带有修饰这些语言结构的属性

```
void dax (double *restrict y,
          double a,
          const double *restrict x,
          int64_t n) { … }
```

---

```
define dso_local void @dax(
        double* noalias nocapture %0,
        double %1,
        double* noalias nocapture readonly %2,
        i64 %3) local_unnamed_addr #0 {…}
```

Q: 这些属性各表示什么意思?

# 这些属性是从哪里来的?

```
void dax (
        double *restrict y,
        double a,
        const double *restrict x,
        int64_t n) { … }
```

```
define dso_local void @dax(
        double* noalias nocapture %0,
        double %1,
        double* noalias nocapture readonly %2,
        i64 %3) local_unnamed_addr #0 {…}
```

- dso_local、nocapture和local_unnamed_addr是编译器通过分析生成的属性
- noalias属性是从C源程序中的restrict属性导出的，readonly是从C源程序中的const限定字中导出的

---

```
%10 = load double, double* %9, align 8, !tbaa !2
```

属性“align 8”表示这条load指令在从内存中读取一个双精度浮点数时，要求地址是8字节对齐的，这个对齐属性也是编译器通过分析生成的

# 总结：C程序转换成LLVM IR

- 把被编译的程序转换成LLVM IR是LLVM编译器生成汇编代码的第一步；
- LLVM IR跟汇编程序相似，但相对简单；
- 在LLVM IR中，所有计算的值都被存储到LLVM IR的寄存器中，并且每个寄存器在LLVM IR中只能被赋值一次；
- LLVM IR中的每一个函数都可以用控制流图来建模，在控制流图中，图的节点是基本块，连接基本块的边代表程序执行过程中控制流的可能走向；
- 跟C相比，LLVM IR中所有的操作都是显式的，比如说C程序中的隐式类型转换在LLVM IR中都变成了显式操作，所有数据类型的位宽都明确表示。

# LLVM IR到汇编代码的转换

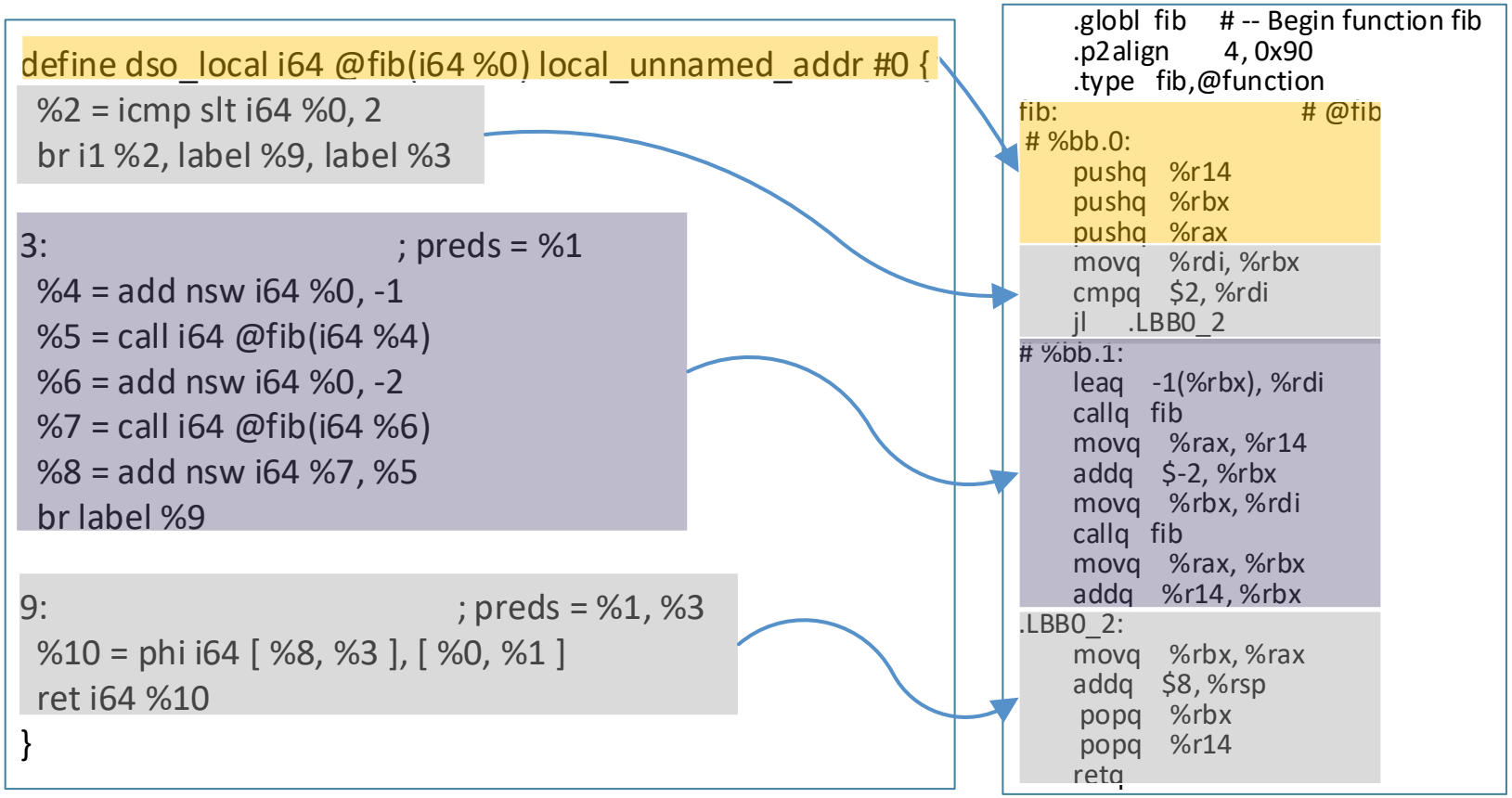
通常来说，编译器需要完成下面三件任务以成功地把LLVM IR转换成X86-64汇编代码：

- **指令选择**：选择合适的X86-64汇编指令去实现LLVM IR中的指令；
- **分配寄存器**：确保指令能充分利用X86-64中的寄存器（而不是内存）去保存值；
- **协调函数调用**：按照X86-64的函数调用规范，确保函数调用者和被调用函数之间的正确协作。

我们的侧重点

# 从LLVM IR到汇编代码的映射

在结构上，LLVM IR和汇编程序非常相似，LLVM IR指令与汇编指令之间的对应也比较清楚。



# 汇编器指导指令

汇编程序中通常包含一些制导指令（directives）告诉汇编器或链接器该如何操作汇编程序中相应的部分

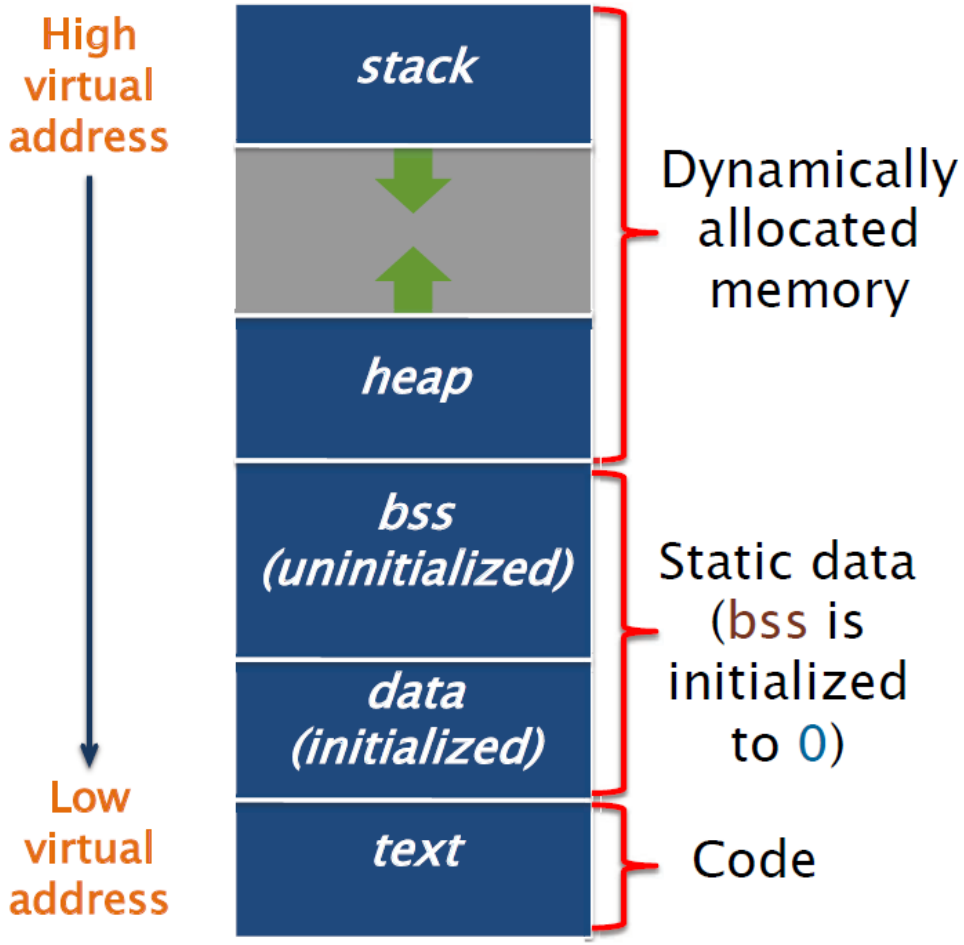
- **段制导指令**（Segment directives）：把汇编文件中的内容组织成不同的段
  - "*.text*"：代表代码段
  - "*.data*"代表数据段
  - "*.bss*"代表bss段

Q: .data段和.bss段有什么区别?

- **存储制导指令**（Storage directives）：跟当前段中的数据存储相关，e.g.
  - x: .space 20：表示在x这个地址空间分配20个字节
  - y: .long 172：表示把常量172L存储在位置y中
  - z: .asciz "6.172"： 表示在位置z中存储字符串常量"6.172\0"
  - .align 8：表示内存中的下一内容需要按8字节对齐
- **范围和链接制导指令**（Scope and linkage directives）：这类制导指令主要是控制链接器的行为.
  - ".global fib"表示fib函数对别的目标文件是可见的

# 程序的内存布局（Linux/X86-64）

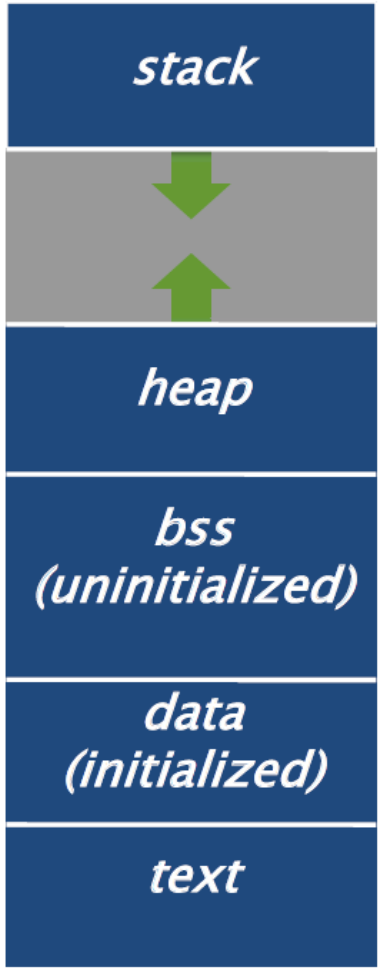
当一个程序运行时，其虚拟内存被组织成不同的段



# 函数调用栈

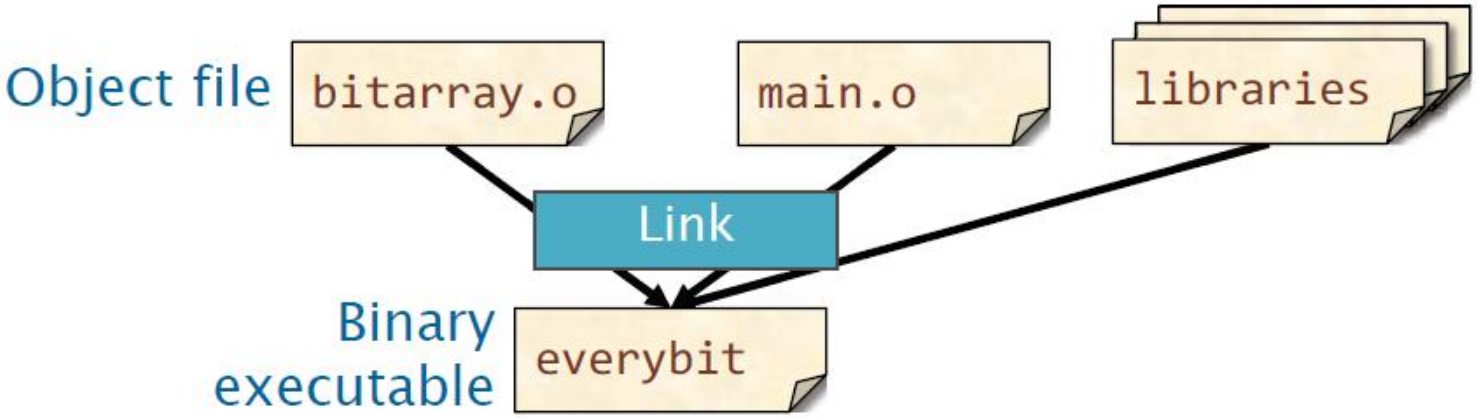
存储着程序运行时函数执行过程中的函数内部数据以及函数调用/返回时跨函数的数据传递，更加具体地说，栈中可能存有如下各种数据：

- **返回地址：**在栈中保留函数调用时的返回地址，以便被调用函数执行完毕后能够返回到函数调用点的下一条指令所在地；
- **寄存器状态：**根据调用函数和被调用函数使用寄存器的情况，在栈中保留寄存器的状态以便不同函数之间能够复用相同的寄存器而不影响程序的语义；
- **函数参数**：有些函数实参如果不能通过寄存器来传给被调用函数，需要通过栈来传递；
- **局部变量**：函数的局部变量或者编译器生成的临时变量如果不能存放在合适的寄存器内，必须存放在栈内。



# 函数调用/返回时的信息传递

问题: 位于不同目标文件中的函数在发生函数调用时协同调用函数与被调用函数之间的信息传递?




函数调用规范.

System V ABI, https://wiki.osdev.org/System_V_ABI

# 函数调用规范（Linux/X86-64）

在X86-64上Linux操作系统的函数调用规范中，栈被组织成一个个栈帧的形式，每个函数的运行实例都有一个自己的唯一栈帧：

- %rbp寄存器指向当前栈帧的顶部；
- %rsp寄存器指向当前栈帧的底部。

函数调用指令（call）和函数返回指令（ret）通过利用栈和指令指针%rip来管理每一个函数调用的返回地址.

- 当执行call指令时，%rip的值（指向call指令的下一条地址，也就是call指令执行完毕后的返回地址）被存到栈帧底部（注意栈帧是从高地址往地址值延伸，%rsp指向栈帧的最底部），然后控制流跳转到call指令的操作数所指向的地址（也就是被调用函数的地址）去继续执行
- 当执行ret指令时，栈底的内容（就是call指令执行时存储在栈上的返回地址）从栈上取出赋给%rip，这样执行控制流就回到了调用函数

# 跨函数调用时寄存器状态的维护

跨函数调用寄存器状态的一致性保证是通过调用函数和被调用函数的协同来共同完成的：

- %rbx、%rbp、%r12-%r15这些寄存器由被调用函数负责保存/恢复（callee-saved）
- 所有其它寄存器由调用函数负责保存/恢复 (caller-saved)

# 参数传递、函数返回值

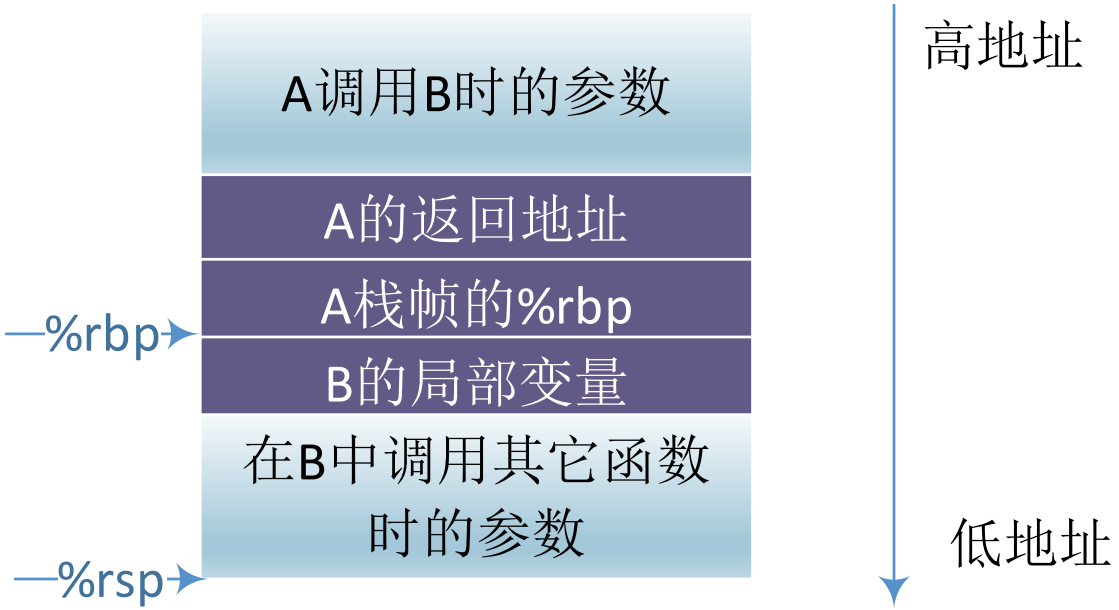
- 如果通过寄存器传递的整数参数超过6个，则前6个整数参数使用寄存器传递，剩余的整数参数通过栈来传递；
- 压栈的顺序是从右往左依次压栈；
- 对于可以用SSE寄存器来存储的浮点参数，可以通过%xmm0-%xmm7寄存器来传递；
- 函数返回值通过%rax寄存器来传递。

| 寄存器 | 描述 |
| --- | --- |
| %rdi | 传递第一个整数参数 |
| %rsi | 传递第二个整数参数 |
| %rdx | 传递第三个整数参数 |
| %rcx | 传递第四个整数参数 |
| %r8 | 传递第五个整数参数 |
| %r9 | 传递第六个整数参数 |

# 函数调用栈帧示意图

- 图中刻画的是函数A调用了函数B，函数B又将调用其它函数时的栈帧示意图；
- 对于A调用B时通过栈传递过来的参数，在函数B的执行过程中可以通过“%rbp+Offset” 的方式来访问；
- 同理，对于函数B中的局部变量，可以通过“%rbp-Offset” 的方式来访问。

**Offset是正整数*



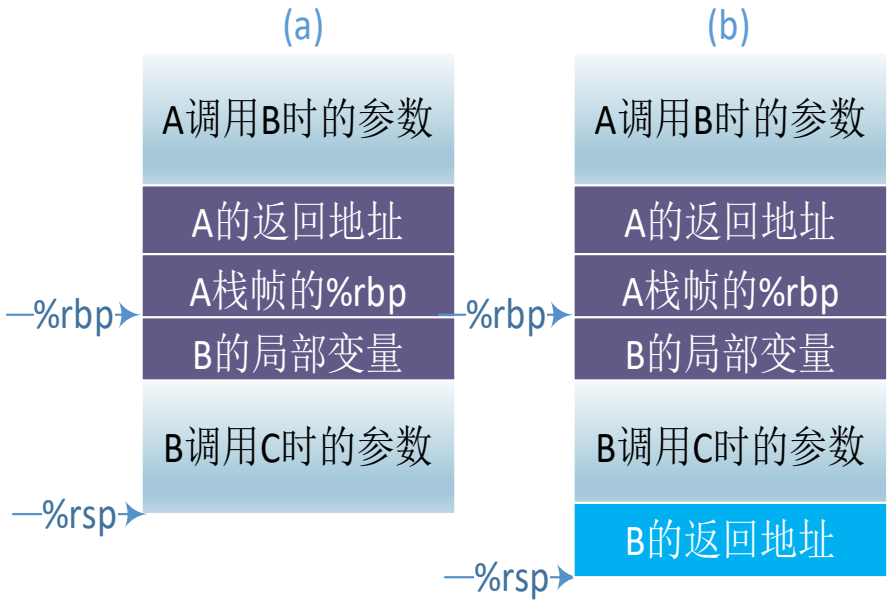
# 执行函数调用是的栈帧变化示意图

- B在执行的过程中即将调用C，（a）表示了在B中把不能通过寄存器传递的参数压栈后的情况，压栈时B对这些实参的访问可以通过“%rbp-Offset”的方式来访问；

**Offset是正整数*

# 执行函数调用是的栈帧变化示意图

- B在执行的过程中即将调用C，（a）表示了在B中把不能通过寄存器传递的参数压栈后的情况，压栈时B对这些实参的访问可以通过“%rbp-Offset”的方式来访问；
- (b)表示了在B中执行了对C的调用指令（call）但控制流又没有执行到C时的情况，这时B的返回地址已经被压入栈内；

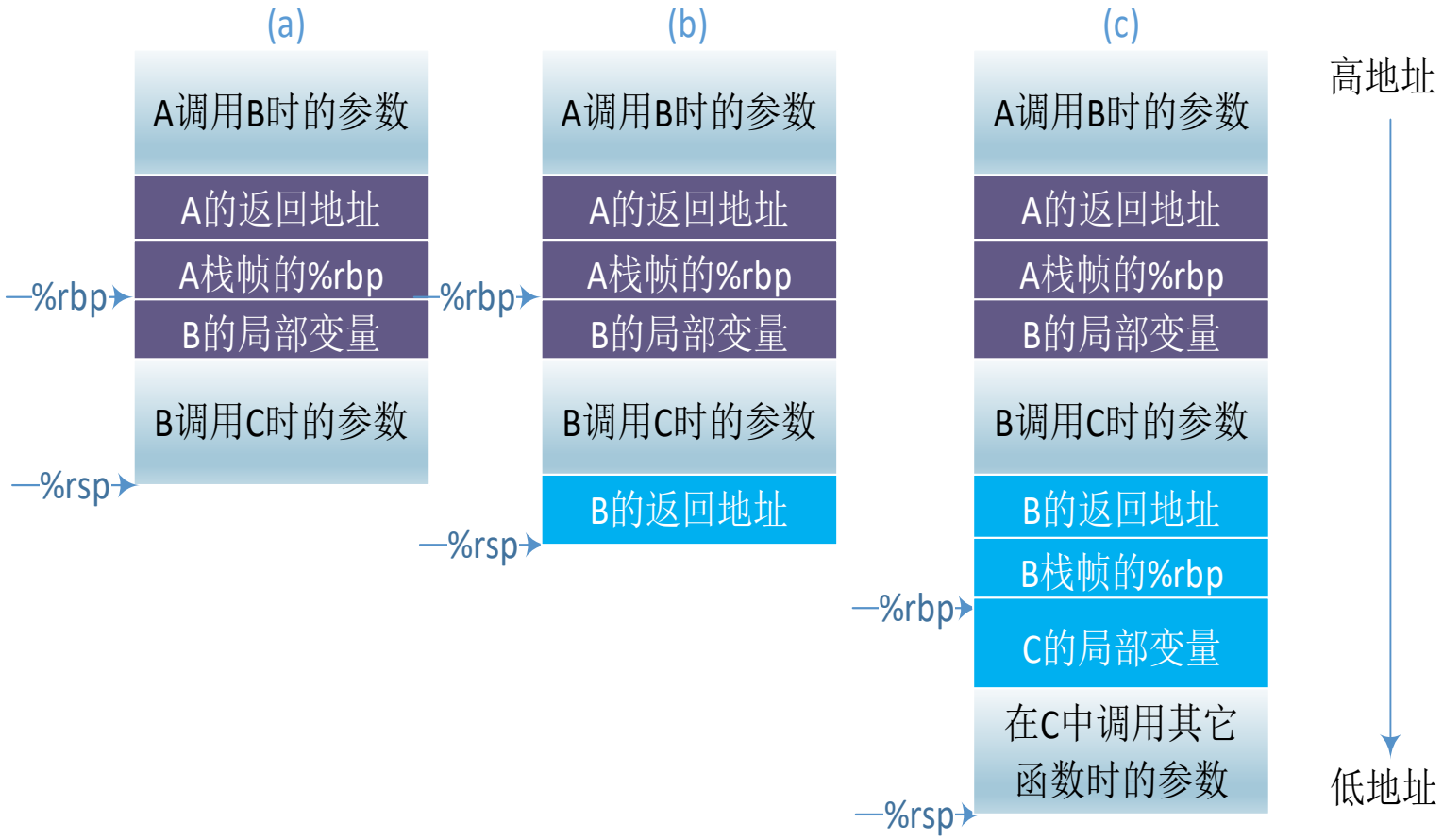



**Offset是正整数*

# 执行函数调用是的栈帧变化示意图

- B在执行的过程中即将调用C，（a）表示了在B中把不能通过寄存器传递的参数压栈后的情况，压栈时B对这些实参的访问可以通过“%rbp-Offset”的方式来访问；
- (b)表示了在B中执行了对C的调用指令（call）但控制流又没有执行到C时的情况，这时B的返回地址已经被压入栈内；
- 从(c)中可以看到：B栈帧的%rbp已经被保存在栈上；寄存器%rbp已经调整成C栈帧的栈顶；C中的局部变量以及C调用其它函数时传递参数所需的栈空间都已经分配完毕

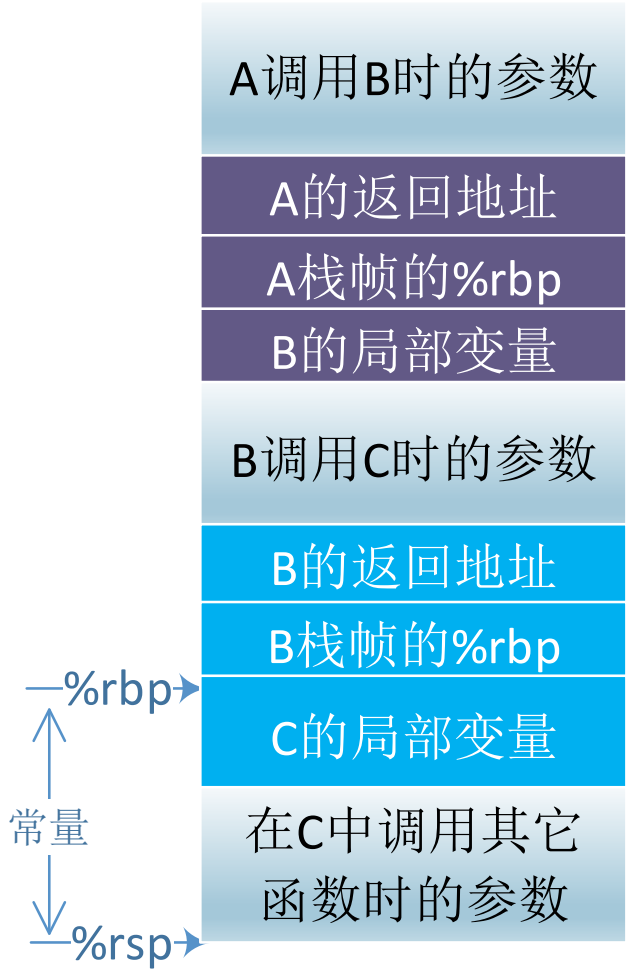
**Offset是正整数*



# 栈内变量访问优化

- 对于任意一个函数f（比如右图中的C）来说，如果f只在调用其它函数时需要有栈空间分配的操作，除此之外没有任何栈空间分配操作，那函数f栈帧上“%rsp-%rbp”的值在编译阶段就可以确定。如果上述条件成立的话，则所有对栈内变量的访问都可以变成“%rsp+Offset”（Offset是非负数）的形式，

Q: 这有什么好处?



# 本次课程总结

在LLVM编译器中从C程序到汇编代码的转换过程。整个转换分为两步：第一步把用C语言编写的源程序转换成LLVM IR，然后第二步把LLVM IR转换成汇编代码。

- 在LLVM IR中，函数被组织成控制流图的形式.
  - 控制流图中的每一个节点对应于LLVM IR中的一个基本块，基本块内的指令序列相当于一个直线型程序
  - 基本块的最后一条指令是控制流跳转指令，这些指令“形成” 的边把控制流图中相关的基本块连接在一起
- 编译器后端通过指令选择、指令调度与寄存器分配，并按照目标指令集架构和目标操作系统上的函数调用规范利用目标指令集架构中的寄存器和栈来协调函数调用，最终实现从LLVM IR到汇编代码的转换。


Q
Q
&
&
A
A