# LECTURE 8

# COMPILING LLVMLITE TO X86

## (WITH LLVM'S HELP)

```
long bar(long n);

long foo(long n) {
    return bar(n);
}
```

```
define  i64 @foo(i64 %0)  {
  %2 = alloca i64
  store i64 %0, i64* %2
  %3 = load i64, i64* %2
  %4 = call i64 @bar(i64 %3)
  ret i64 %4
}

declare i64 @bar(i64)
```

```
define  i64 @foo(i64 %0)  {
  %2 = alloca i64
  store i64 %0, i64* %2
  %3 = load i64, i64* %2
  %4 = call i64 @bar(i64 %3)
  ret i64 %4
}

declare i64 @bar(i64)
```

```
foo:
        pushq   %rbp
        movq    %rsp, %rbp
        subq    $16, %rsp
        movq    %rdi, -8(%rbp)
        movq    -8(%rbp), %rdi
        callq   bar
        addq    $16, %rsp
        popq    %rbp
        retq
```

```
define  i64 @foo(i64 %0)  {
  %2 = alloca i64
  store i64 %0, i64* %2
  %3 = load i64, i64* %2
  %4 = call i64 @bar(i64 %3)
  ret i64 %4
}

declare i64 @bar(i64)
```

```
foo:
        pushq   %rbp
        movq    %rsp, %rbp
        subq    $16, %rsp
        movq    %rdi, -8(%rbp)
        movq    -8(%rbp), %rdi
        callq   bar
                %rsp
                %rbp
        retq
```

alloca Ty →
subq sizeof(Ty), %rsp

```
define  i64 @foo(i64 %0)  {
  %2 = alloca i64
  store i64 %0, i64* %2
  %3 = load i64, i64* %2
  %4 = call i64 @bar(i64 %3)
  ret i64 %4
}

declare i64 @bar(i64)
```

```
foo:
        pushq   %rbp
        movq    %rsp, %rbp
        subq    $8, %rsp
        movq    %rdi, -8(%rbp)
        movq    -8(%rbp), %rdi
                                r
                                %rsp
                                p
                        retq
```

LLVM allocated more than needed here. Missed optimization.

```
define  i64 @foo(i64 %0)  {
  %2 = alloca i64
  store i64 %0, i64* %2
  %3 = load i64, i64* %2
  %4 = call i64 @bar(i64 %3)
  ret i64 %4
}

declare i64 @bar(i64)
```

```
foo:
        pushq   %rbp
        movq    %rsp, %rbp
        subq    $8, %rsp
        movq    %rdi, -8(%rbp)
        movq    -8(%rbp), %rdi
        call
```

arguments (here: %0) mapped to registers according to calling conventions (here: %rdi)

```
define  i64 @foo(i64 %0)  {
  %2 = alloca i64
  store i64 %0, i64* %2
  %3 = load i64, i64* %2
  %4 = call i64 @bar(i64 %3)
  ret i64 %4
}

declare i64 @b
```

```
foo:
        pushq   %rbp
        movq    %rsp, %rbp
        subq    $8, %rsp
        movq    %rdi, -8(%rbp)
        movq    -8(%rbp), %rdi
        callq    bar
                 %rsp
```

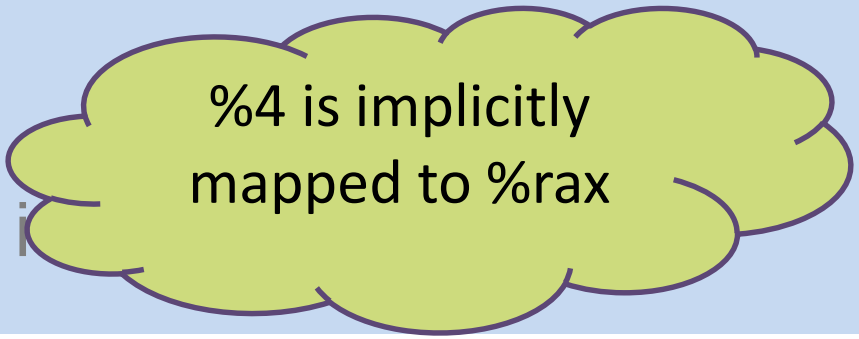
loads from/stores to stack slots →
movq & offset(%rbp)

```
define  i64 @foo(i64 %0)  {
  %2 = alloca i64
  store i64 %0, i64* %2
  %3 = load i64, i64* %2
  %4 = call i64 @bar(i64 %3)
  ret i64 %4
}

declare i64 @b
```

```
foo:
        pushq   %rbp
        movq    %rsp, %rbp
        subq    $8, %rsp
        movq    %rdi, -8(%rbp)
        movq    -8(%rbp), %rdi
        callq   bar
                %rsp
```

%3 is passed as argument to bar →
map it to %rdi

```
define  i64 @foo(i64 %0)  {
  %2 = alloca i64
  store i64 %0, i64* %2
  %3 = load i64, i64* %2
  %4 = call i64 @bar(i64 %3)
  ret i64 %4
}

declare i
```




```
foo:
        pushq   %rbp
        movq    %rsp, %rbp
        subq    $8, %rsp
        movq    %rdi, -8(%rbp)
        movq    -8(%rbp), %rdi
        callq    bar
        addq    $8, %rsp
        popq    %rbp
        retq
```

```
long bar(long n);

long foo(long n) {
    return bar(n);
}
```

```
define  i64 @foo(i64 %0)  {
  %2 = alloca i64
  store i64 %0, i64* %2
  %3 = load i64, i64* %2
  %4 = call i64 @bar(i64 %3)
  ret i64 %4
}

declare i64 @bar(i64)
```

```
long bar(long n);

long foo(long n) {
    return bar(n) + n;
}
```

```
define i64 @foo(i64 %0)  {
  %2 = alloca i64
  store i64 %0, i64* %2
  %3 = load i64, i64* %2
  %4 = call i64 @bar(i64 %3)
  %5 = load i64, i64* %2
  %6 = add nsw i64 %4, %5
  ret i64 %6
}
```

```
define i64 @foo(i64 %0)  {
  %2 = alloca i64
  store i64 %0, i64* %2
  %3 = load i64, i64* %2
  %4 = call i64 @bar(i64 %3)
  %5 = load i64, i64* %2
  %6 = add nsw i64 %4, %5
  ret i64 %6
}
```

```
foo:
        pushq   %rbp
        movq    %rsp, %rbp
        subq    $8, %rsp
        movq    %rdi, -8(%rbp)
        movq    -8(%rbp), %rdi
        callq   bar
        addq    -8(%rbp), %rax
        addq    $8, %rsp
        popq    %rbp
        retq
```

Storing args/temporaries to stack slots simplifies codegen:
no need to keep track if a register was overwritten; instead load before every use

```
define i64 @foo(i64 %
  %2 = alloca i64
  store i64 %0, i64* %2
  %3 = load i64, i64* %2
  %4 = call i64 @bar(i64 %3)
  %5 = load i64, i64* %2
  %6 = add nsw i64 %4, %5
  ret i64 %6
}
```

```
                %rsp
        movq    %rdi, -8(%rbp)
        movq    -8(%rbp), %rdi
        callq   bar
        addq    -8(%rbp), %rax
        addq    $8, %rsp
        popq    %rbp
        retq
```

# Compiling LLVM locals

- How do we manage storage for each %uid defined by an LLVM instruction?

- Option 1:
  - Map each %uid to a x86 register
  - Efficient!
  - Difficult to do effectively: many %uid values, only 16 registers
  - We will see how to do this later in the semester

- Option 2:
  - Map each %uid to a stack-allocated space
  - Less efficient!
  - Simple to implement

- For HW3 we will follow Option 2

```
struct Point {
    long x;
    long y;
};

void foo(){
    struct Point p;
    p.x = 1;
    p.y = 2;
}
```

```
%struct.Point = type { i64, i64 }

define  void @foo()  {
  %1 = alloca %struct.Point
  %2 = getelementptr,
          %struct.Point* %1, i32 0, i32 0
  store i64 1, i64* %2
  %3 = getelementptr,
          %struct.Point* %1, i32 0, i32 1
  store i64 2, i64* %3
  ret void
}
```

```
%struct.Point = type { i64, i64 }

define  void @foo()  {
  %1 = alloca %struct.Point
  %2 = getelementptr,
    %struct.Point* %1, i32 0, i32 0
  store i64 1, i64* %2
  %3 = getelementptr,
    %struct.Point* %1, i32 0, i32 1
  store i64 2, i64* %3
  ret void
}
```

```
foo:
        pushq   %rbp
        movq    %rsp, %rbp
        subq    $16, %rsp
        movq    $1, -16(%rbp)
        movq    $2, -8(%rbp)
        addq    $16, %rsp
        popq    %rbp
        retq
```

```
%struct.Point = type { i64, i64 }

define  void @foo()  {
  %1 = alloca %struct.Point
  %2 = getelementptr,
    %struct.Point* %1, i32 0, i32 0
  store i64 1, i64* %2
  %3 = getelementptr,
    %struct.Point* %1, i32 0, i32 1
  store i64 2, i64* %3
  ret void
}
```

```
foo:
        pushq   %rbp
        movq    %rsp, %rbp
        subq    $16, %rsp
        movq    $1, -16(%rbp)
        movq    $2, -8(%rbp)
        addq    $16, %rsp
        popq    %rbp
```

%1 in this case corresponds to -16(%rbp);
getelementptr → base address + offset;

# Compilation of GEP

1. Translate GEP's base pointer into an actual address (e.g., a stack slot).

2. Compute the offset specified by the indices and add it to the base address.
   – In case of non-constant indices, part of the computation must be done at runtime.

```
struct Point {
    long x;
    long y;
};

void foo(struct Point *ps,
                        long n){
    ps[n].y = 42;
}
```

```
%struct.Point = type { i64, i64 }


define  void @foo(%struct.Point*  %0, i64 %1)
{
  %3 = getelementptr,
            %struct.Point* %0, i64 %1, i32 1
  store i64 42, i64* %3
  ret void
}
```

```
%struct.Point = type { i64, i64 }

define  void @foo(%struct.Point*  %0, i64 %1)
{
  %3 = getelementptr,
            %struct.Point* %0, i64 %1, i32 1
  store i64 42, i64* %3
  ret void
}
```

Or imulq $16, %rsi

```
foo:
        shlq    $4, %rsi
        movq    $42, 8(%rdi,%rsi)
        retq
```

```
%struct.Point = type { i64, i64 }


define  void @foo(%struct.Point*  %0, i64 %1)
{
  %3 = getelementptr,
            %struct.Point* %0, i64 %1, i32 1
  store i64 42, i64* %3
  ret void
}
```

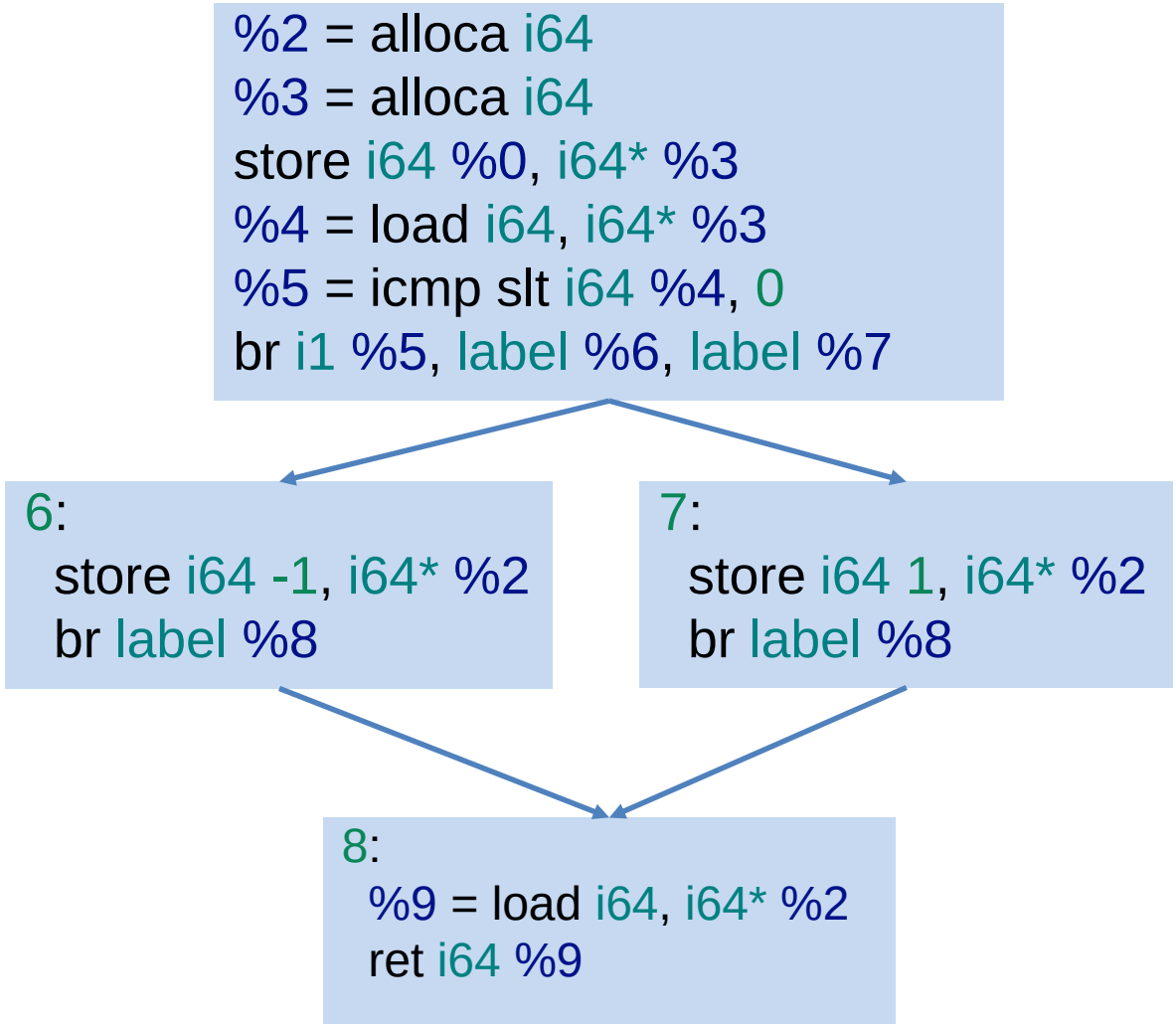
```
foo:
        imulq   $16, %rsi
        addq    %rsi, %rdi
        movq    $42, 8(%rdi)
        retq
```

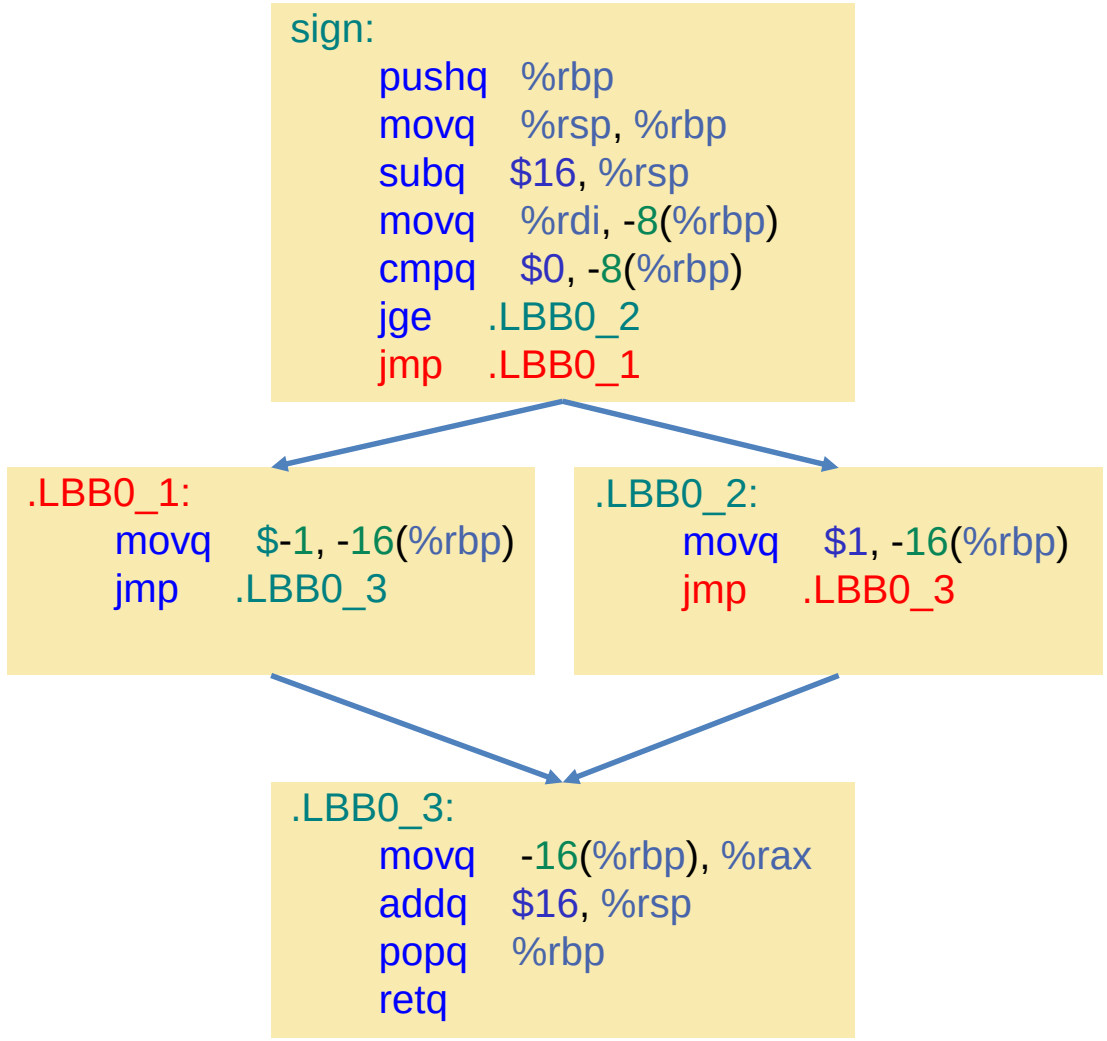
```
long sign(long n) {
    long s;
    if ( n < 0)
        s =  - 1;
    else
        s =  1;
    return s;
}
```

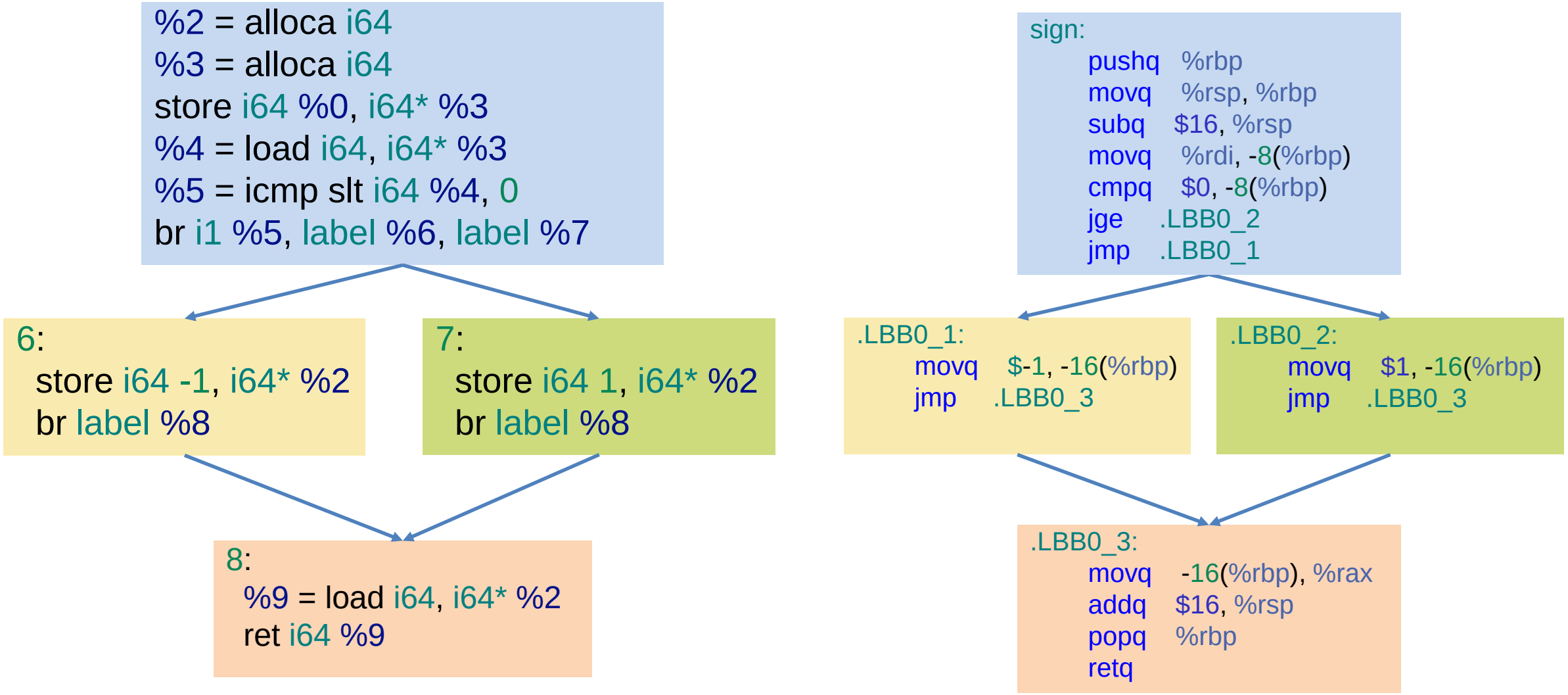
```
define  i64 @sign(i64 %0)  {
  %2 = alloca i64
  %3 = alloca i64
  store i64 %0, i64* %3
  %4 = load i64, i64* %3
  %5 = icmp slt i64 %4, 0
  br i1 %5, label %6, label %7
6:
  store i64 -1, i64* %2
  br label %8
7:
  store i64 1, i64* %2
  br label %8
8:
  %9 = load i64, i64* %2
  ret i64 %9
}
```

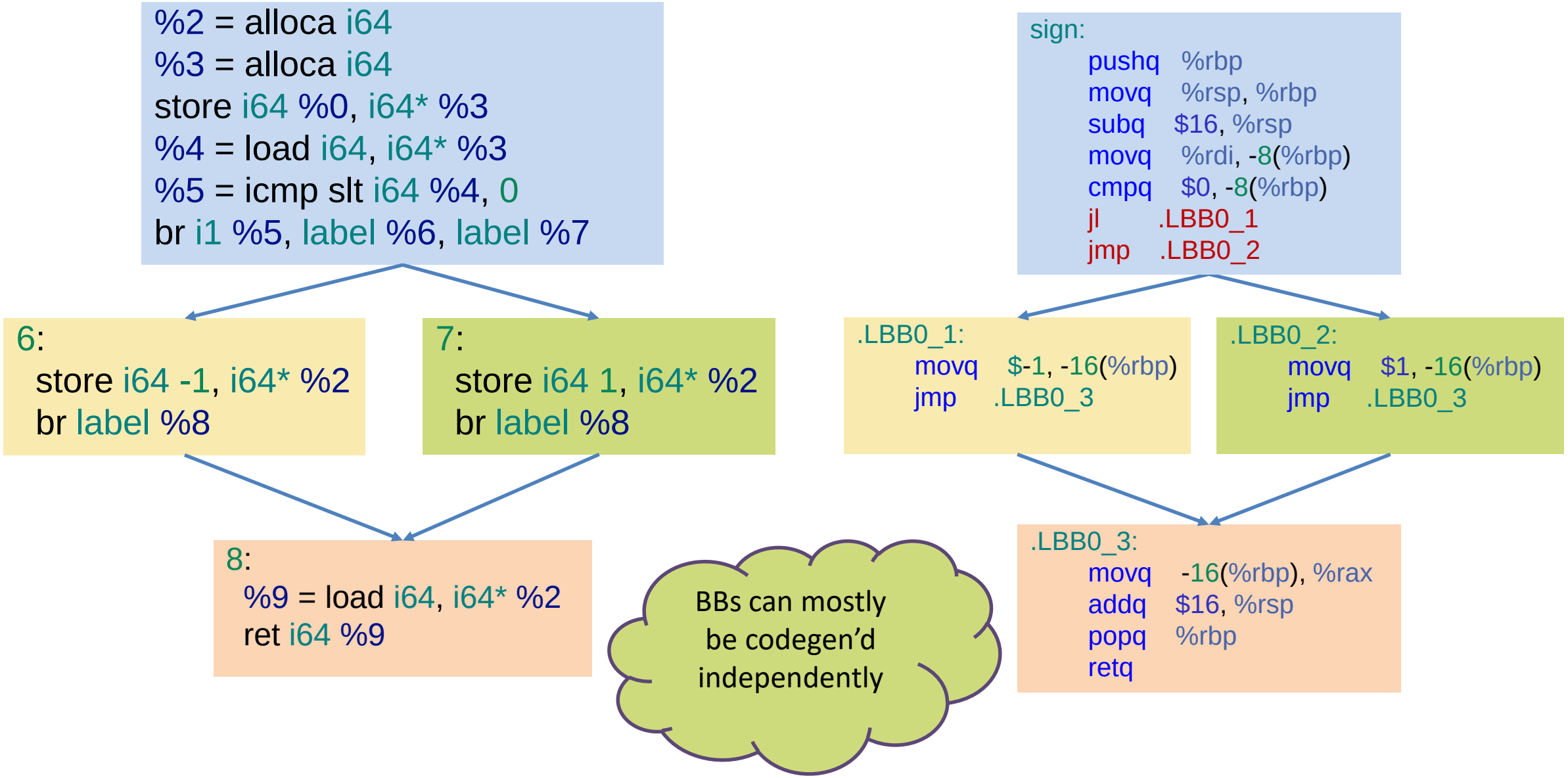
```
define  i64 @sign(i64 %0)  {
  %2 = alloca i64
  %3 = alloca i64
  store i64 %0, i64* %3
  %4 = load i64, i64* %3
  %5 = icmp slt i64 %4, 0
  br i1 %5, label %6, label %7
6:
  store i64 -1, i64* %2
  br label %8
7:
  store i64 1, i64* %2
  br label %8
8:
  %9 = load i64, i64* %2
  ret i64 %9
}
```

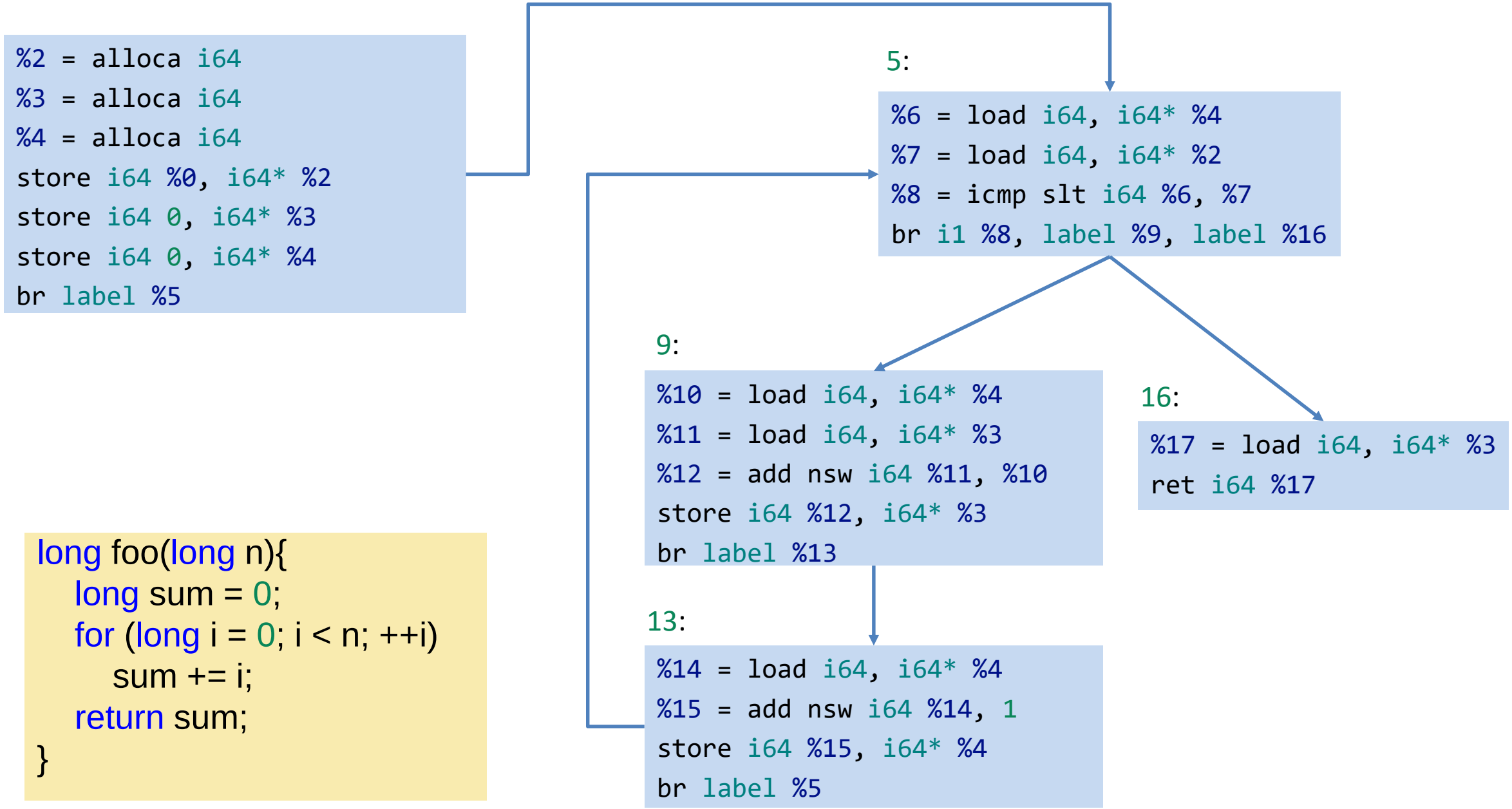
```
sign:
        pushq   %rbp
        movq    %rsp, %rbp
        subq    $16, %rsp
        movq    %rdi, -8(%rbp)
        cmpq    $0, -8(%rbp)
        jge     .LBB0_2
        movq    $-1, -16(%rbp)
        jmp     .LBB0_3
.LBB0_2:
        movq    $1, -16(%rbp)
.LBB0_3:
        movq    -16(%rbp), %rax
        addq    $16, %rsp
        popq    %rbp
        retq
```

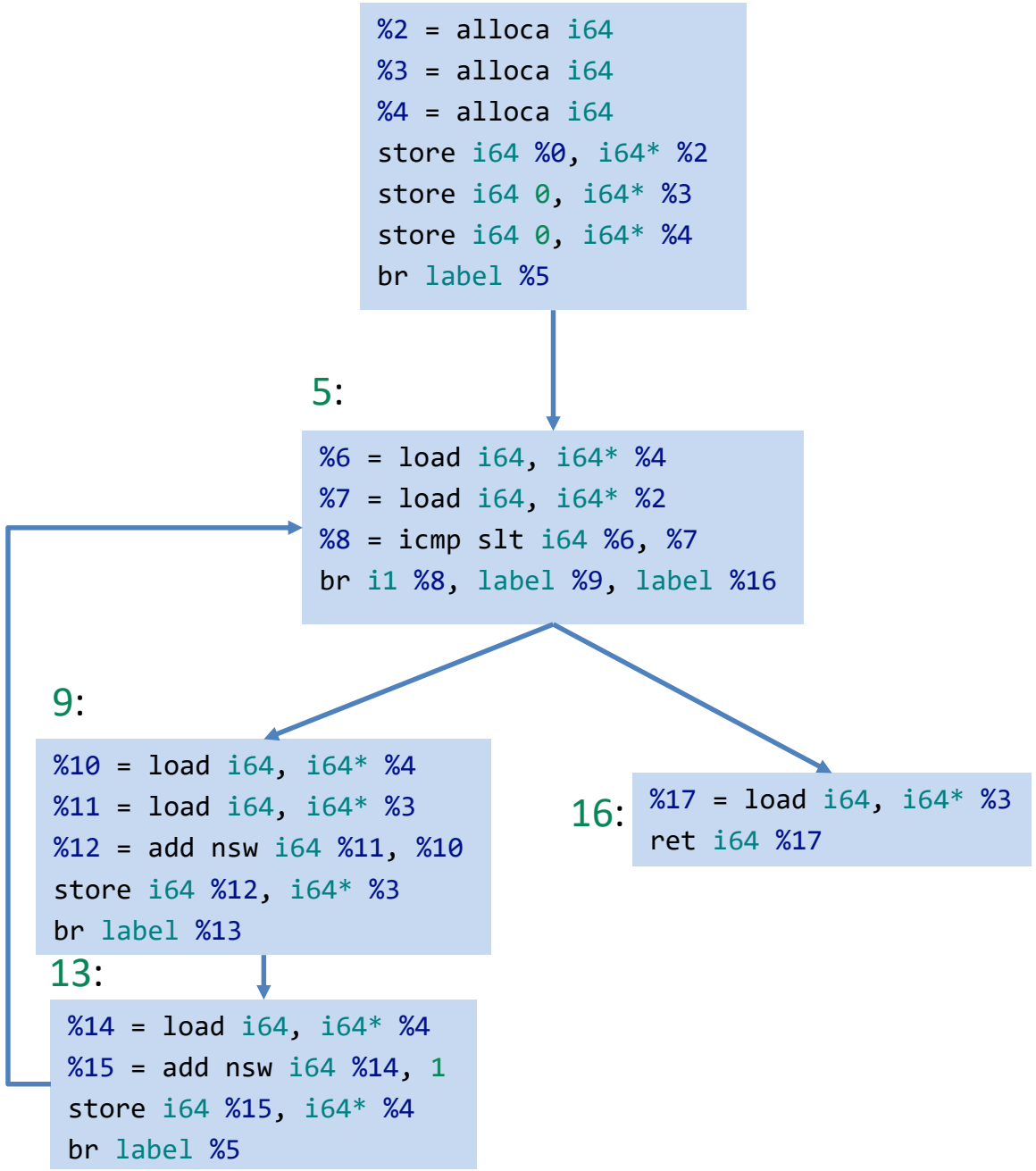

%2 = alloca i64
%3 = alloca i64
store i64 %0, i64* %3
%4 = load i64, i64* %3
%5 = icmp slt i64 %4, 0
br i1 %5, label %6, label %7
6:
store i64 -1, i64* %2
br label %8
7:
store i64 1, i64* %2
br label %8
8:
%9 = load i64, i64* %2
ret i64 %9

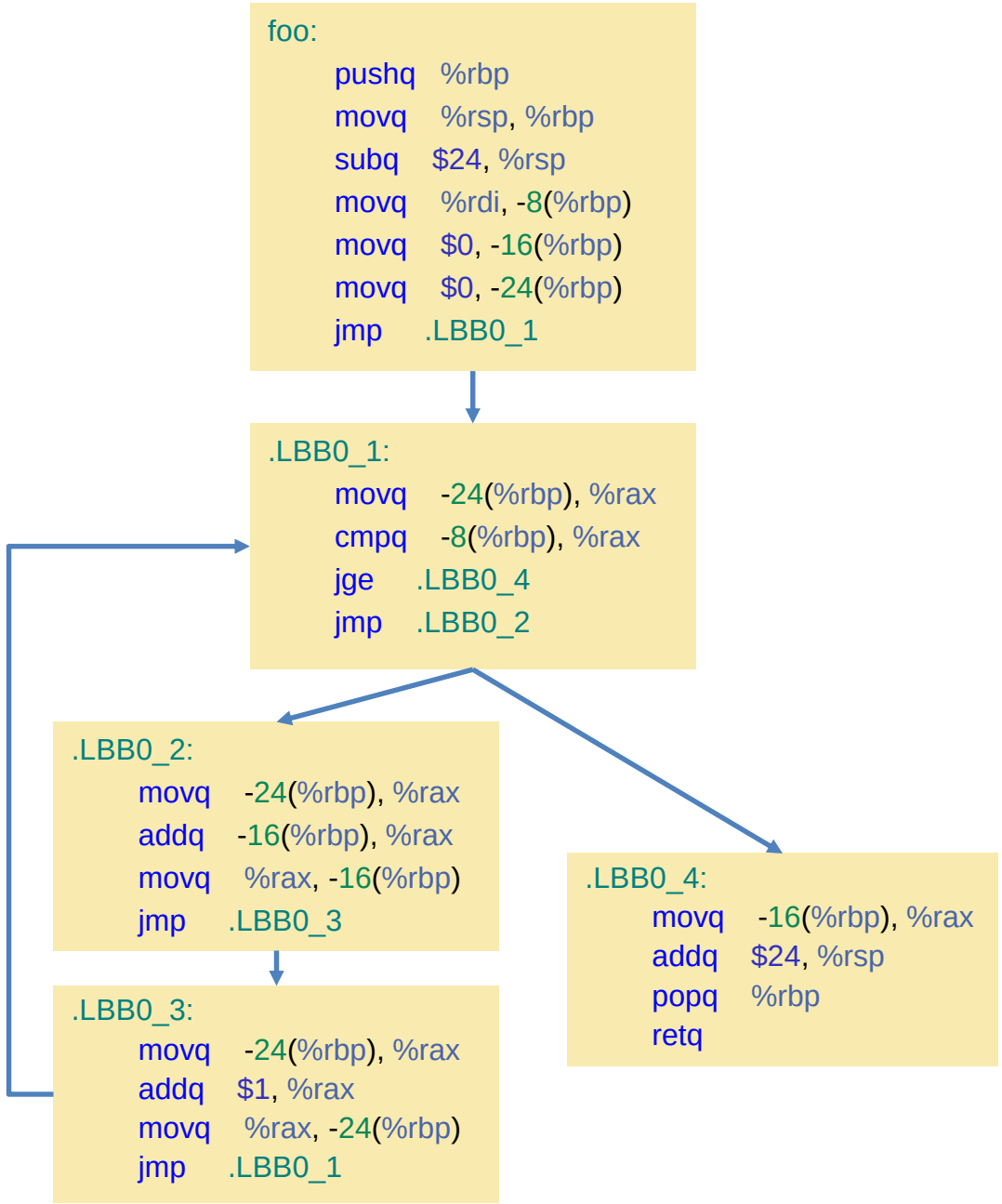

sign:
pushq %rbp
movq %rsp, %rbp
subq $16, %rsp
movq %rdi, -8(%rbp)
cmpq $0, -8(%rbp)
jge .LBB0_2
jmp .LBB0_1
.LBB0_1:
movq $-1, -16(%rbp)
jmp .LBB0_3
.LBB0_2:
movq $1, -16(%rbp)
jmp .LBB0_3
.LBB0_3:
movq -16(%rbp), %rax
addq $16, %rsp
popq %rbp
retq


%2 = alloca i64
%3 = alloca i64
store i64 %0, i64* %3
%4 = load i64, i64* %3
%5 = icmp slt i64 %4, 0
br i1 %5, label %6, label %7
6:
store i64 -1, i64* %2
br label %8
7:
store i64 1, i64* %2
br label %8
8:
%9 = load i64, i64* %2
ret i64 %9
sign:
pushq %rbp
movq %rsp, %rbp
subq $16, %rsp
movq %rdi, -8(%rbp)
cmpq $0, -8(%rbp)
jge .LBB0_2
jmp .LBB0_1
.LBB0_1:
movq $-1, -16(%rbp)
jmp .LBB0_3
.LBB0_2:
movq $1, -16(%rbp)
jmp .LBB0_3
.LBB0_3:
movq -16(%rbp), %rax
addq $16, %rsp
popq %rbp
retq


%2 = alloca i64
%3 = alloca i64
store i64 %0, i64* %3
%4 = load i64, i64* %3
%5 = icmp slt i64 %4, 0
br i1 %5, label %6, label %7
6:
store i64 -1, i64* %2
br label %8
7:
store i64 1, i64* %2
br label %8
8:
%9 = load i64, i64* %2
ret i64 %9
BBs can mostly be codegen'd independently
sign:
pushq %rbp
movq %rsp, %rbp
subq $16, %rsp
movq %rdi, -8(%rbp)
cmpq $0, -8(%rbp)
jl .LBB0_1
jmp .LBB0_2
.LBB0_1:
movq $-1, -16(%rbp)
jmp .LBB0_3
.LBB0_2:
movq $1, -16(%rbp)
jmp .LBB0_3
.LBB0_3:
movq -16(%rbp), %rax
addq $16, %rsp
popq %rbp
retq


%2 = alloca i64
%3 = alloca i64
%4 = alloca i64
store i64 %0, i64* %2
store i64 0, i64* %3
store i64 0, i64* %4
br label %5
5:
%6 = load i64, i64* %4
%7 = load i64, i64* %2
%8 = icmp slt i64 %6, %7
br i1 %8, label %9, label %16
9:
%10 = load i64, i64* %4
%11 = load i64, i64* %3
%12 = add nsw i64 %11, %10
store i64 %12, i64* %3
br label %13
13:
%14 = load i64, i64* %4
%15 = add nsw i64 %14, 1
store i64 %15, i64* %4
br label %5
16:
%17 = load i64, i64* %3
ret i64 %17
long foo(long n){
long sum = 0;
for (long i = 0; i < n; ++i)
sum += i;
return sum;
}


%2 = alloca i64
%3 = alloca i64
%4 = alloca i64
store i64 %0, i64* %2
store i64 0, i64* %3
store i64 0, i64* %4
br label %5
5:
%6 = load i64, i64* %4
%7 = load i64, i64* %2
%8 = icmp slt i64 %6, %7
br i1 %8, label %9, label %16
9:
%10 = load i64, i64* %4
%11 = load i64, i64* %3
%12 = add nsw i64 %11, %10
store i64 %12, i64* %3
br label %13
13:
%14 = load i64, i64* %4
%15 = add nsw i64 %14, 1
store i64 %15, i64* %4
br label %5
16:
%17 = load i64, i64* %3
ret i64 %17


foo:
pushq %rbp
movq %rsp, %rbp
subq $24, %rsp
movq %rdi, -8(%rbp)
movq $0, -16(%rbp)
movq $0, -24(%rbp)
jmp .LBB0_1
.LBB0_1:
movq -24(%rbp), %rax
cmpq -8(%rbp), %rax
jge .LBB0_4
jmp .LBB0_2
.LBB0_2:
movq -24(%rbp), %rax
addq -16(%rbp), %rax
movq %rax, -16(%rbp)
jmp .LBB0_3
.LBB0_3:
movq -24(%rbp), %rax
addq $1, %rax
movq %rax, -24(%rbp)
jmp .LBB0_1
.LBB0_4:
movq -16(%rbp), %rax
addq $24, %rsp
popq %rbp
retq

# Simple X86 code generation for functions

1. Write function prologue
2. Generate necessary stack slots
3. Translate Basic Blocks
   - Connect them with jumps
4. Write function epilogue

# DEMO

# Other LLVMlite Features

- Globals
  - must use %rip relative addressing
- Calls
  - Follow x64 AMD ABI calling conventions
  - Should interoperate with C programs
- getelementptr
  - trickiest part