At a High level i

SSA Construction i

- SSA stands for Static Single Assignment
- Each variable can only be assigned exactly once (similar to linear variables in functional programming)
- SSA allows for simplifying and improving a variety of compiler optimizations.

Simple Example i

- What's wrong with this code?

```
y = 1
y = 2
x = y
print(x, y)
```

Simple Example Cont. i

- The first assignment is **dead** before it can be **used**.
- An equivalent program would therefore be:

```
y = 2
x = y
print(x, y)
```

Enter SSA i

I've rewritten this in SSA form:

```
y1 = 1
y2 = 2
x1 = y2
print(x1, y2)
```

- Note that you can see the uses of all the variables: only x1 and y2 are ever used
- Thus you can **delete** all variables that are unused.
- This is a trivial Dead Code Elimination (DCE) pass.

Handling conditionals i

```
x = random() # 1
if x < 0.5:
    result = "heads" # 2
else:
    result = "tails" # 3
print(result) # 4
```

- Note that 1 must always run and that 2, 3, 4 all come after
- 2 and 3 must come before 4, since print has to print either 2 or 3
- Thus, 1 dominates 2,3,4 and 2 and 3 dominate 4.
- We can handle this with a `phi`, which takes as input all of its possible dominating nodes and explicitly graphs to data flow of functions:

Handling conditionals ii

```
entry:
  %x = call double @random()
  %cond = fcmp olt double %x, 5.000000e-01
  br i1 %cond, label %then, label %else
then:
  %heads.ptr = getelementptr inbounds [6 x i8], ptr @.heads, i64 0, i64 0
  br label %merge
else:
  %tails.ptr = getelementptr inbounds [6 x i8], ptr @.tails, i64 0, i64 0
  br label %merge
merge:
  %result = phi ptr [ %heads.ptr, %then ], [ %tails.ptr, %else ]
  call void @print_string(ptr %result)
  ret void
```

Handling conditionals (cont). i

That gets us this asm:

```
.LCPI0_0:
        .quad   0x3fe0000000000000
flip_assume:
        movsd   xmm1, qword ptr [rip + .LCPI0_0]
        ucomisd xmm1, xmm0
        lea     rax, [rip + .L.str.1]
        lea     rdi, [rip + .L.str]
        cmovb   rdi, rax
        jmp     puts@PLT
.L.str:
        .asciz  "heads"
.L.str.1:
        .asciz  "tails"
```

Handling conditionals (cont). ii

Compiler Explorer Link for coin flip

Dead Code Elimination Again i

Let's take this example:

- The if is always true

```
x = random() # 1
if x <= 1:
    result = "heads" # 2
else:
    result = "tails" # 3
print(result) # 4
```

The LLVM IR emitted is the same:

Dead Code Elimination Again ii

```
entry:
  %x = call double @random()
  %cond = fcmp olt double %x, 5.000000e-01
  br i1 %cond, label %then, label %else
then:
  %heads.ptr = getelementptr inbounds [6 x i8], ptr @.heads, i64 0, i64 0
  br label %merge
else:
  %tails.ptr = getelementptr inbounds [6 x i8], ptr @.tails, i64 0, i64 0
  br label %merge
merge:
  %result = phi ptr [ %heads.ptr, %then ], [ %tails.ptr, %else ]
  call void @print_string(ptr %result)
```

But let's see how we'd delete it:

Dead Code Elimination Again Cont. i

```
entry:
  %x = call double @random() # we can label %x as [0.0, 1.0]
  %cond = fcmp ole double %x, 1.0 # thus this is true
```

```
entry:
  %x = call double @random() # we can label %x as [0.0, 1.0]
  %cond = true # transform to true
  br i1 %cond, label %then, label %else # now else is dead
```

So we can delete the else and set %result equal to %heads and emit this program:

Dead Code Elimination Again Cont. ii

```
entry:
  %x = call double @random()
  %result = %heads.ptr
  call void @print_string(ptr %result)
```

A real example i

```c
#include <stdio.h>

void flip_assume(double x) {
  __builtin_assume(x >= 0.0 && x <= 1.0); // prove x has to be [0, 1]

  const char *result;
  if (x <= 1.0) {
    result = "heads";
  } else {
    result = "tails";
  }

  puts(result);
}
```

Emits this code:

A real example ii

```
@.str = private unnamed_addr constant [6 x i8] c"heads\00", align 1

define dso_local void @flip_assume(double %0) {
  %2 = tail call i32 @puts(ptr @.str)
  ret void
}
```

Compiler Explorer Link for dead branch

```
flip_assume:
    leaq    .L.str(%rip), %rdi
    jmp puts@PLT
.L.str:
    .asciz  "heads"
```

Some other control flow optimizations i

Hoisting:

```
if (cond) {
    foo();
    a();
} else {
    foo();
    b();
}
```

Some other control flow optimizations ii

```
foo();
if (cond) {
    a();
} else {
    b();
}
```

Some other control flow optimizations cont: i

Switching to a lookup table:

```
switch (x) {
    case 0:
        return 1;
    case 1:
        return 2;
    case 2:
        return 3;
    case 3:
        return 4;
    default:
        return 5;
}
```

Some other control flow optimizations cont: ii

```
int table[] = {1, 2, 3, 4};
if (x < 4) {
    return table[x];
} else {
    return 5;
}
```

Common Subexpression Elimination i

```
res1 = x + y;
res2 = x + y;
print(res1);
print(res1);
```

- delete res2 since it's unnecessary

```
res1 = x + y;
print(res1);
print(res1);
```

LLVM Architecture i

- There are 3 main parts of the toolchain:
- The frontend (clang, rustc, swiftc, go) emits unoptimized LLVM IR with as much information as it can get
- The middle-end, which does target-independent optimizations that map LLVM IR -> optimized LLVM IR (this program is fittingly called `opt`).
- The back-end, which implements target dependent optimizations that turn LLVM IR -> assembly. (this program is called `llc`).

Frontend Responsibilities i

- The frontend should emit as much metadata as possible to aid `opt` and `llc` in optimizations:

```
pub fn base128_length(n: u128) -> usize {
    let bits = if n != 0 { 128 - n.leading_zeros() } else { 1 };
    let bytes = bits.div_ceil(7);
    bytes as usize
}
```

Frontend Responsibilities ii

```
define noundef range(i64 0, 6) i64 @src(i32 noundef %n) unnamed_addr #0 {
start:
  %0 = or i32 %n, 1
  %self.i = zext i32 %0 to i128
  ; calls llvm's count leading zeroes intrinsic
  ; with the information that the result fits in the range [0, 128]
  %1 = tail call range(i128 0, 128) i128 @llvm.ctlz.i128(i128 %self.i, i1 true)
  ; now truncate the result to an i8 from a i128, there is no
  ; signed or unsigned wrapping
  %2 = trunc nuw nsw i128 %1 to i8
  ; sub cannot unsigned wrap
  %bits.i = sub nuw i8 -128, %2
  ; div here
  %d1.i = udiv i8 %bits.i, 7
  ; zero extend the i8 to i64
  %d.zext.i = zext nneg i8 %d1.i to i64
  ; call remainder
  %r2.i = urem i8 %bits.i, 7
  %_8.not.i = icmp ne i8 %r2.i, 0
  %3 = zext i1 %_8.not.i to i64
```

Range Analysis i

Range analysis involves passing the lower and upper bound of integer variables during program execution. This is useful for code optimization.

```
int ret_0(unsigned x) {
    x %= 7;       // x is now in [0, 6]
    if (x < 7)    // this has to be true
        return 0;
    return 1;     // this branch is dead
}
```

```
ret_0:
        xor     eax, eax
        ret
```

Compiler explorer link

Range Analysis Cont. i

This emits something like this:

Range Analysis Cont. ii

```
define i32 @g(i32 %x) {
entry:
  %r = urem i32 %x, 7
  %cmp = icmp ult i32 %r, 7
  br i1 %cmp, label %then, label %else

then:
  br label %merge

else:
  br label %merge

merge:
  %ret = phi i32 [ 0, %then ], [ 1, %else ]
  ret i32 %ret
}
```

Range Analysis Cont. iii

Range analysis helps simplify to this:

```
define i32 @g(i32 %x) {
entry:
  br i1 true, label %then, label %else

then:
  br label %merge

else:
  br label %merge

merge:
  ret i32 0
}
```

SimplifyCFG will delete all the code and turn it to this:

Range Analysis Cont. iv

```
define i32 @g(i32 %x) {
entry:
  ret i32 0
}
```

Instcombine i

- One of the most common middle-end passes. You generally write proofs (with a program called `alive`) to prove that algebraic transformations are valid.

Take this program:

```
int min_branch(int a, int b) {
  if (a < b)
    return a;
  return b;
}
```

Instcombine ii

```
%cmp = icmp slt i32 %a, %b
br i1 %cmp, label %then, label %else
then:
  br label %merge
else:
  br label %merge
merge:
  %r = phi i32 [ %a, %then ], [ %b, %else ]
  ret i32 %r
```

- InstCombine will turn this branch into a select and find out this is actually a min function and turn it into an intrinsic for min

Instcombine iii

```
define dso_local noundef i32 @min_branch(i32 noundef %0, i32 noundef %1) local_unnamed_addr
  %3 = tail call i32 @llvm.smin.i32(i32 %0, i32 %1)
  ret i32 %3
}
```

And fold to a `cmov`:

```
min_branch:
        mov     eax, esi
        cmp     edi, esi
        cmovl   eax, edi
        ret
```

Backend Passes i

Backend passes tend to be target dependent:

For example this LLVM IR:

```
%addr = add ptr %base, %idx_scaled
%v = load i32, ptr %addr
```

can be optimized to a scaled `mov` in x86_64:

```
movl 8(%rdi,%rsi,4), %eax
```

Strength Reduction i

Take this print function:

Compiler Explorer

```
void print_array(const int *restrict a, size_t n) {
    for (size_t i = 0; i < n; ++i)
        printf("%d\n", a[i]);
}
```

In pseudocode, this is:

```
copy the ptr var
for all the items in the array:
    var += i * 4
    print(a[var])
```

Strength Reduction ii

Note that this multiplication is expensive and this is the same thing, replace the pointer fiddling with an addition instead:

```
copy the ptr var
for all the items in the array:
    var += 4
    print(arr[var])
```

Or even:

```
copy the ptr var
for all the items in the array:
    print(arr[var *= 4])
```

Which is expressed in the code:

Strength Reduction iii

```
mov     esi, dword ptr [r14 + 4*r12] ; the load and add are folded
                                     ; mul is used here but it's a scale
```

Case Studies i

C++26 constexpr structured bindings generate bad code

Case Studies ii

```cpp
#include <array>
#include <cstdint>
using u8 = std::uint8_t;
template<u8 N>
struct Range
{
    template<std::size_t I>
    consteval friend u8 get(Range) noexcept { return I; }
};
namespace std
{
    template<u8 N>
    struct tuple_size<Range<N>> {static constexpr std::size_t value = N;};
    template<std::size_t I, u8 N>
    struct tuple_element<I, Range<N>> {using type = u8;};
}
template<u8 x>constexpr u8 constant{x};
template<std::size_t L, std::size_t R>
__attribute__((always_inline)) inline constexpr std::array<u8, L+R> concat(const std::ar
{
```

Case 1: i

- It emits this (150+ loc)

Case 1: ii

```
test(std::array<unsigned char, 16ul> const&, std::array<unsigned char, 16ul> const&):
  movzx ecx, byte ptr [rip + reference temporary for std::array<unsigned char, 16ul + 16ul>
  mov rax, rdi
  movzx ecx, byte ptr [rsi + rcx]
  mov byte ptr [rdi], cl
  movzx ecx, byte ptr [rip + reference temporary for std::array<unsigned char, 16ul + 16ul>
  movzx ecx, byte ptr [rsi + rcx]
  mov byte ptr [rdi + 1], cl
  movzx ecx, byte ptr [rip + reference temporary for std::array<unsigned char, 16ul + 16ul>
  movzx ecx, byte ptr [rsi + rcx]
  mov byte ptr [rdi + 2], cl
  ... (30 more times)
  reference temporary for std::array<unsigned char, 16ul + 16ul> concat<16ul, 16ul>(std::ar
    .byte 0
  reference temporary for std::array<unsigned char, 16ul + 16ul> concat<16ul, 16ul>(std::ar
    .byte 1
  reference temporary for std::array<unsigned char, 16ul + 16ul> concat<16ul, 16ul>(std::ar
    .byte 2
  ... (30 more times)
```

Case 2: i

Strength Reduce lock xadd to lock sub in Instcombine

```
bool fn_add(std::atomic<uint64_t> &a, uint64_t n) noexcept {
  return a.fetch_add(-n, std::memory_order_relaxed) == n;
}
```

Lowered to this:

```
fn_add:
    movq    %rsi, %rax
    negq    %rax
    lock    xaddq   %rax, (%rdi)
    cmpq    %rsi, %rax
    sete    %al
    retq
```

Case 2: ii

But the ideal is this, which is far better:

```
fn_add:
    lock    subq    %rsi, (%rdi)
    sete    %al
    retq
```

Case 3: i

Remove two unnecessary movs when %rdx is an input to mulx

```
void mul64_u128(uint64_t *hi, uint64_t *lo, uint64_t a, uint64_t b) {
  unsigned __int128 r = (unsigned __int128)a * b;
  *lo = (uint64_t)r;
  *hi = (uint64_t)(r >> 64);
}
```

Generates this:

Case 3: ii

```
mul64_u128:
    movq    %rdx, %rax
    movq    %rcx, %rdx
    mulxq   %rax, %rcx, %rax
    movq    %rcx, (%rsi)
    movq    %rax, (%rdi)
    retq
```

When this is the correct codegen (2 less movs):

```
mul64_u128:
    mulx    %rcx, %rax, %rdx
    movq    %rax, (%rsi)
    movq    %rdx, (%rdi)
    ret
```

Case 4: i

Strength Reduce Constant Division

```
int udiv_7(unsigned short x) {
  return x / 7;
}
```

Emits this currently (note there's no division here)

Case 4: ii

```
udiv_16_7:                              # @udiv_16_7
    movzwl  %di, %eax
    imull   $9363, %eax, %ecx
    shrl    $16, %ecx
    subl    %ecx, %edi
    movzwl  %di, %eax
    shrl    %eax
    addl    %ecx, %eax
    shrl    $2, %eax
    retq
```

The ideal is this:

Case 4: iii

```
udiv_16_7:                              # @udiv_16_7
    movzwl %di, %eax
    imulq  $613572608, %rax, %rax
    shrq   $32, %rax
    retq
```

Case 4: Continued i

You might be expecting something like this (this is how a non-optimizing compiler would compile this code)

Case 4: Continued ii

```
udiv_16_7:
  push %rbp
  mov %rsp, %rbp
  sub $16, %rsp
  mov %rsp, -8(%rbp)
  mov %di, -10(%rbp)
  mov $7, %rax
  push %rax
  lea -10(%rbp), %rax
  movzwl (%rax), %eax
  pop %rdi
  cdq
  idiv %edi
  jmp .L.return.udiv_7
.L.return.udiv_7:
  mov %rbp, %rsp
  pop %rbp
  ret
```

The reason why is division is **extremely** expensive. On my computer, this non-optimized version is about

Case 4: Strength Reduction i

This is the division by constant strength reduction: For an unsigned division where you don't know X but you know C: We compute a magic reciprocal m and then compute a quotient q:

We want to replace:

$$q = \left\lfloor \frac{x}{d} \right\rfloor$$

With:

$$q \approx \left\lfloor \frac{x \cdot m}{2^k} \right\rfloor$$

We choose m with the formula:

$$\frac{2^k}{d}$$

Case 4: Strength Reduction ii

So that:

$$\frac{x}{d} \approx \frac{x \cdot m}{2^k}$$

If this equation is approximate, not exact, then a fixup phase is done. If the equation is exact, then we can replace the instruction sequence with this formula, one multiplication and one right shift.

Case 4: Continued i

The reason why clang currently emits a fixup sequence is because the magic, m does not have enough precision to precisely express the quotient in a 32-bit space. However, on most 32-bit and 64-bit architectures, 64-bit multiply is not that bad.

We can thus change the formula a bit:

We still calculate:

$$q \approx \left\lfloor \frac{x \cdot m}{2^k} \right\rfloor$$

We choose m with the formula:

$$\frac{2^k}{d}$$

Prove it: i

However we calculate three formulas to prove that

$$\frac{x}{d} = \frac{x \cdot m}{2^k}$$

Error Check:

$$Error = Magic \cdot C - 2^{Shift}$$

No overflow (fits in the space we alloted):

$$MaxX \cdot Magic < 2^{DestWidth}$$

And exactness (the equality is true):

Prove it: ii

$$MaxX \cdot Error < 2^{Shift}$$

Case 5: Ceiling Division i

Div Ceil Optimizations

Let's do another optimization:

How would you implement ceiling division (instead of floor division)

If there is no remainder, e.g. $\mathrm{div\_ceil}(21, 7) = 3$ ceiling division acts like normal division.

If there is a remainder, you add one to the quotient: $\mathrm{div\_ceil}(22, 7) = 4$

How do we translate this to code?

Ceiling Division Continued i

There's a formula to compute ceiling division with only one division (instead of two, which we stated before):

The general formula is:

$$\text{div\_ceil}(X, Y) = \left\lfloor \frac{X}{Y} \right\rfloor + \begin{cases} 1 & \text{if } X \bmod Y > 0 \\ 0 & \text{otherwise} \end{cases}$$

However we can exploit the fact that we have cheap floor divide. Instead of doing a remainder calculation, we can fold the remainder into the division.

$$\text{div\_ceil}(X, Y) = \left\lceil \frac{X}{Y} \right\rceil = \left\lfloor \frac{X + Y - 1}{Y} \right\rfloor$$
$$\text{for } Y > 0$$

This actually fails in alive:

Ceiling Division Continued ii

Failing Proof

Thankfully alive tells us the reasoning (we have made a variable have undefined behavior).

Ceiling Division Continued i

The counter case is that if x is 250, we add 6 to it and we overflow an i8 (0-255).

So we need to find a way to either extend (zero-extend) or constrain $X + Y - 1 < 255$

Alive Proof for div_ceil

The proof works and our reward is better code:

Ceiling Division Continued ii

```
div_ceil_without_assume:                # @div_ceil_without_assume
        movl    %edi, %eax
        movabsq $2635249153617166336, %rcx
        mulq    %rcx
        leal    (,%rdx,8), %eax
        movl    %edx, %ecx
        subl    %eax, %ecx
        xorl    %eax, %eax
        addl    %edi, %ecx
        setne   %al
        addl    %edx, %eax
        retq
```

Ceiling Division Continued iii

```
div_ceil_improved:
        leal    6(%rdi), %eax
        movabsq $2635249153617166336, %rcx
        mulq    %rcx
        movq    %rdx, %rax
        retq
```

Case 6: Improving Range Analysis i

range handling doesn't propagate range information through udiv, sdiv, urem, srem, mul, add, sub, trunc.

Improving Constant Range

This change propagates information recursively for integers that are created that use these operations.

In this example, the icmp is deleted because %x + 4 < -6

```
%add = add nsw i32 %x, 4
%cmp = icmp ult i32 %add, -6
ret i1 %cmp
```

turns into:

```
ret i1 true
```

Another Example i

This changes the following (poison/undefined behavior)

```
%a = add nsw <2 x i8> %o, <i8 -1, i8 -1>
%b = zext <2 x i8> %a to <2 x i32>
%c = add nuw nsw <2 x i32> %b, <i32 1, i32 poison>
```

And just deletes it:

```
%c = zext <2 x i8> %o to <2 x i32>
```

Another Example ii

```
dec_zext_add_nonzero_vec_poison2:       # @dec_zext_add_nonzero_vec_poison2
        por     xmm0, xmmword ptr [rip + .LCPI0_0]
        pcmpeqd xmm1, xmm1
        paddb   xmm0, xmm1
        pxor    xmm1, xmm1
        punpcklbw       xmm0, xmm1
        punpcklwd       xmm0, xmm1
        mov     eax, 1
        movd    xmm1, eax
        paddd   xmm0, xmm1
        ret
```

Another Example iii

```
dec_zext_add_nonzero_vec_poison2:       # @dec_zext_add_nonzero_vec_poison2
        por     xmm0, xmmword ptr [rip + .LCPI0_0]
        pxor    xmm1, xmm1
        punpcklbw       xmm0, xmm1
        punpcklwd       xmm0, xmm1
        ret
```

Popcount i

Popcount detection:

```
while (x != 0) {
  x &= x - 1;
}
```

Clang currently emits:

Popcount ii

```
xorl  %eax, %eax
testl %edi, %edi
je    .Lexit
.Lloop:
  incl  %eax
  blsrl %edi, %edi
  jne   .Lloop
retq
```

This code isn't properly optimized to the best:

```
popcntl %edi, %eax
retq
```

No more popcount detection i

- In any programming language, there are infinite ways to write any program.
- Thus, there are infinite ways to actually write popcount.
- The compiler writer cannot write each pattern by hand and lower them to popcount as there are infinite such programs.
- We've instead decided to give an intrinsic and have frontends have better abstractions for popcounting:
- C++ has `std::popcount`, C has had `__builtin_popcount` for 2+ decades, rust has `.count_ones()`. These all lower automatically to `popcount` without llvm needing to handle detecting infinite programs.
- Sometimes it's better to offer an intrinsic and allow people to use better constructs rather than optimize every given program.

Extras i

```
int sum_up_to(int value) {
    int result = 0;
    for (int x = 0; x < value; ++x) {
        result += x;
    }
    return result;
}
```

Link

Global Value Numbering i

```
int f(int a, int b, int c) {
    int x = a + b;

    if (c > 0) {
        return (a + b) * x;
    }

    return x - (a + b);
}
```

Global Value Numbering ii

```
f:
    lea     ecx, [rdi + rsi] ; ecx = a + b
    imul    ecx, ecx         ; ecx = (a + b) * (a + b)
    xor     eax, eax         ; eax = 0
    test    edx, edx         ; c > 0?
    cmovg   eax, ecx         ; if yes, return square, else eax, 0
    ret
```

Link

Load Elimination i

```
int f(int *p, int *q) {
    int x = *p;
    *q = 42;
    int y = *p;

    return x + y;
}
```

Link

Questions? i