CS443: Compiler Construction

Lecture 16: Static Single Assignment

Stefan Muller

Based on material by Steve Zdancewic

Midterm Exam: next Tuesday

- Normal lecture room, normal lecture time
- Open book, open notes (but no electronics)
- Reference material included in exam (no need for you to print+bring):
    - MiniIITRAN spec
    - MiniC spec
    - LLVM reference
- Practice exam on Blackboard
- Will be answering questions at OH (on Zoom) next Monday

Midterm Exam: next Tuesday

- 100 points, 75 minutes
    - If you’re close to spending X minutes on an X point q, move on

- 10-20%: MC, short answer
- 80-90%: 3-4 longer, multi-part questions

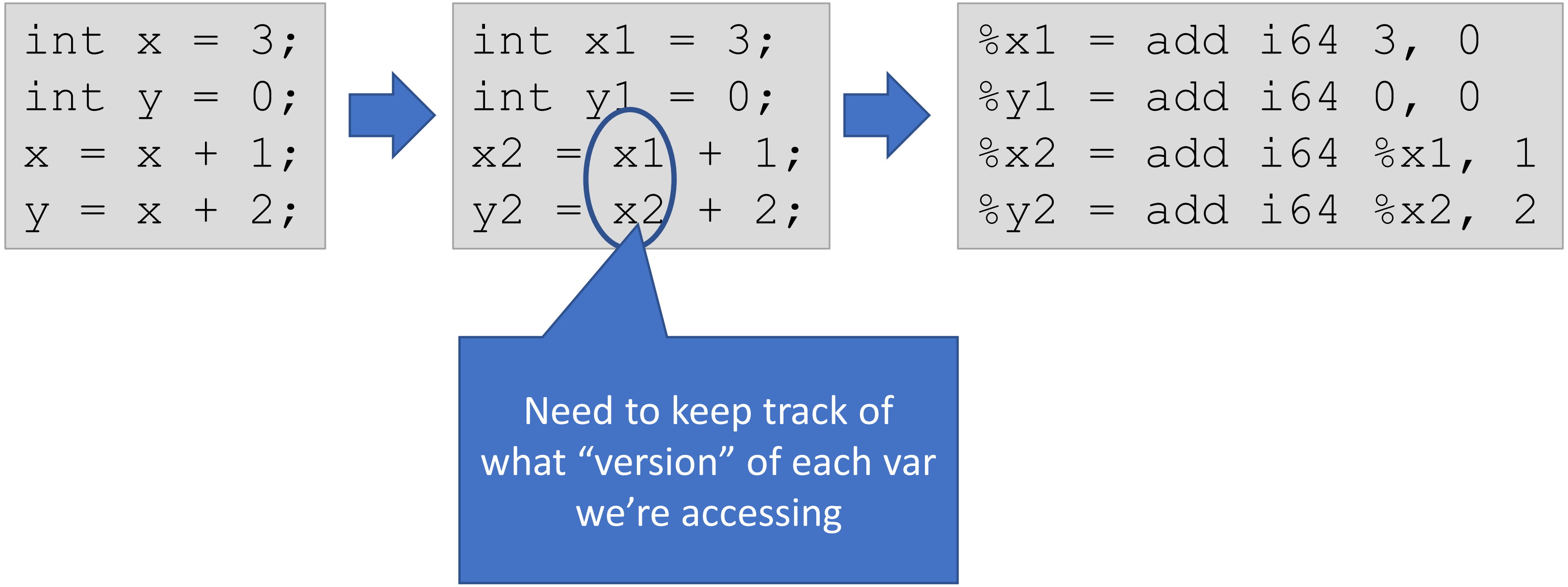
- Lectures: 0-12
- Projects: 0-4

Static Single Assignment: Local variables assigned only once

- Every variable name associated with one static value
  - Like FP with no shadowing!

- Great for optimizations!

For straight-line code, just number different instances of variables



```
int x = 3;
int y = 0;
x = x + 1;
y = x + 2;
```

```
int x1 = 3;
int y1 = 0;
x2 = x1 + 1;
y2 = x2 + 2;
```

```
%x1 = add i64 3, 0
%y1 = add i64 0, 0
%x2 = add i64 %x1, 1
%y2 = add i64 %x2, 2
```

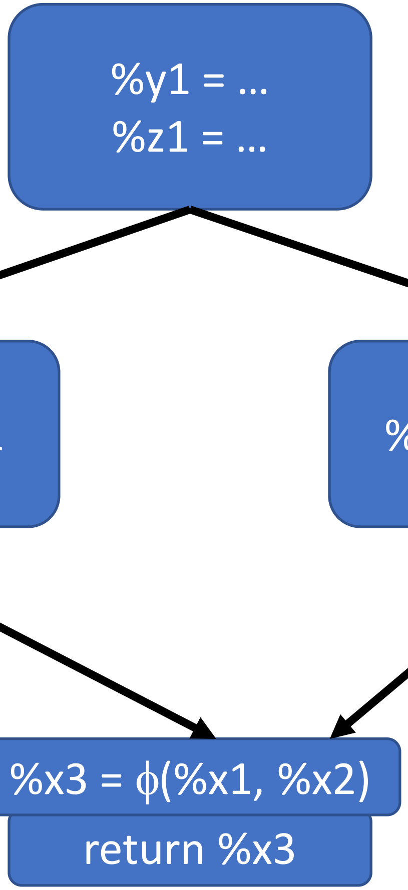
Need to keep track of what “version” of each var we’re accessing

Branching causes a problem

```
int y = …
int x = …
int z = …
if (p) {
  x = y + 1;
} else {
  x = y * 2;
}
z = x + 3;
```

```
entry:
  %y1 = …
  %x1 = …
  %z1 = …
  %p = icmp …
  br i1 %p, label %then, label %else
then:
  %x2 = add i64 %y1, 1
  br label %merge
else:
  %x3 = mul i64 %y1, 2
  br label %merge
merge:
  %z2 = %add i64 ???, 3
```

Phi (ϕ) functions "choose" which version of a variable based on where we came from



Phi in LLVM

```
%dest = phi <ty> [<val>, <label>]+
```

```
int y = …
int x = …
int z = …
if (p) {
  x = y + 1;
} else {
  x = y * 2;
}
z = x + 3;
```

```
  %y1 = …
  %x1 = …
  %z1 = …
  %p = icmp …
  br i1 %p, label %then, label %else
then:
  %x2 = add i64 %y1, 1
  br label %merge
else:
  %x3 = mul i64 %y1, 2
  br label %merge
merge:
  %x4 = phi i64 [%x2, %then], [%x3, %else]
  %z2 = %add i64 %x4, 3
```

Converting to inefficient SSA is pretty simple in theory

- Renumber all definitions of each variable
- Update uses of variables with correct number
- Insert phi nodes at join points

- This inserts way more phi functions than you need. The algorithm for doing better is complex and based on dominators

Phi functions are fictitious

- Q: How do we implement phi?
- A: Usually don’t
  - If we’re lucky/smart, %x2, %x3, and %x4 will all be in same reg anyway
  - Can also convert out of SSA before compiling to assembly
  - If all else fails, can implement it as a `mov` instruction before the jump

“If all else fails, can implement it as a mov instruction before the jump”

```
then:
  %x2 = add i64 %y1, 1
  br label %merge
else:
  %x3 = mul i64 %y1, 2
  br label %merge
merge:
  %x4 = phi i64 [%x2, then], [%x3, %else]
  %z2 = %add i64 %x4, 3
```

```
then:
  addi a2, a1, 1
  mv a4, a2
  jal zero, merge
else:
  addi t0, zero, 2
  mul a3, a1, t0
  mv a4, a3
merge:
  addi a5, a4, 3
```

“If we’re lucky/smart, %x2, %x3, and %x4 will all be in same reg anyway”

```
then:
  %x2 = add i64 %y1, 1
  br label %merge
else:
  %x3 = mul i64 %y1, 2
  br label %merge
merge:
  %x4 = phi i64 [%x2, then], [%x3, %else]
  %z2 = %add i64 %x4, 3
```

(actually)

```
then:
  addi a2, a1, 1
  jal zero, merge
else:
  addi t0, zero, 2
  mul a2, a1, t0
merge:
  addi a5, a2, 3
```

“Arguments” to phi functions may be defined “later”

```
int x = 0;
while (x < 10)
  x++;
```

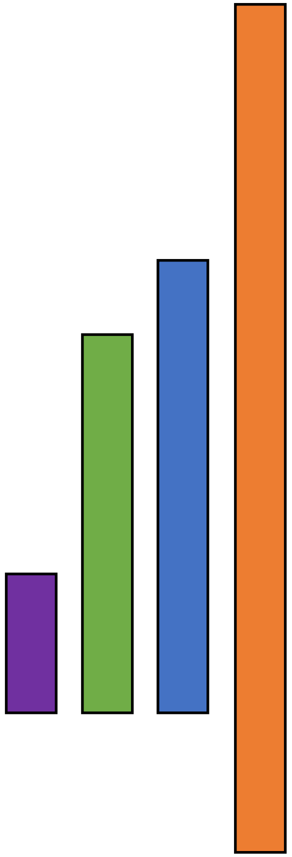
```
entry:
  %x1 = add i32 0 0
  br label %test

test:
  %x2 = phi i32 [%x1, %entry], [%x3, body]
  %p = icmp lt i32 %x2 10
  br i1 %p, label %body, label %end

body:
  %x3 = add i32 %x2 1
  br label %test

end: …
```

Aside: How is scope handled in LLVM?



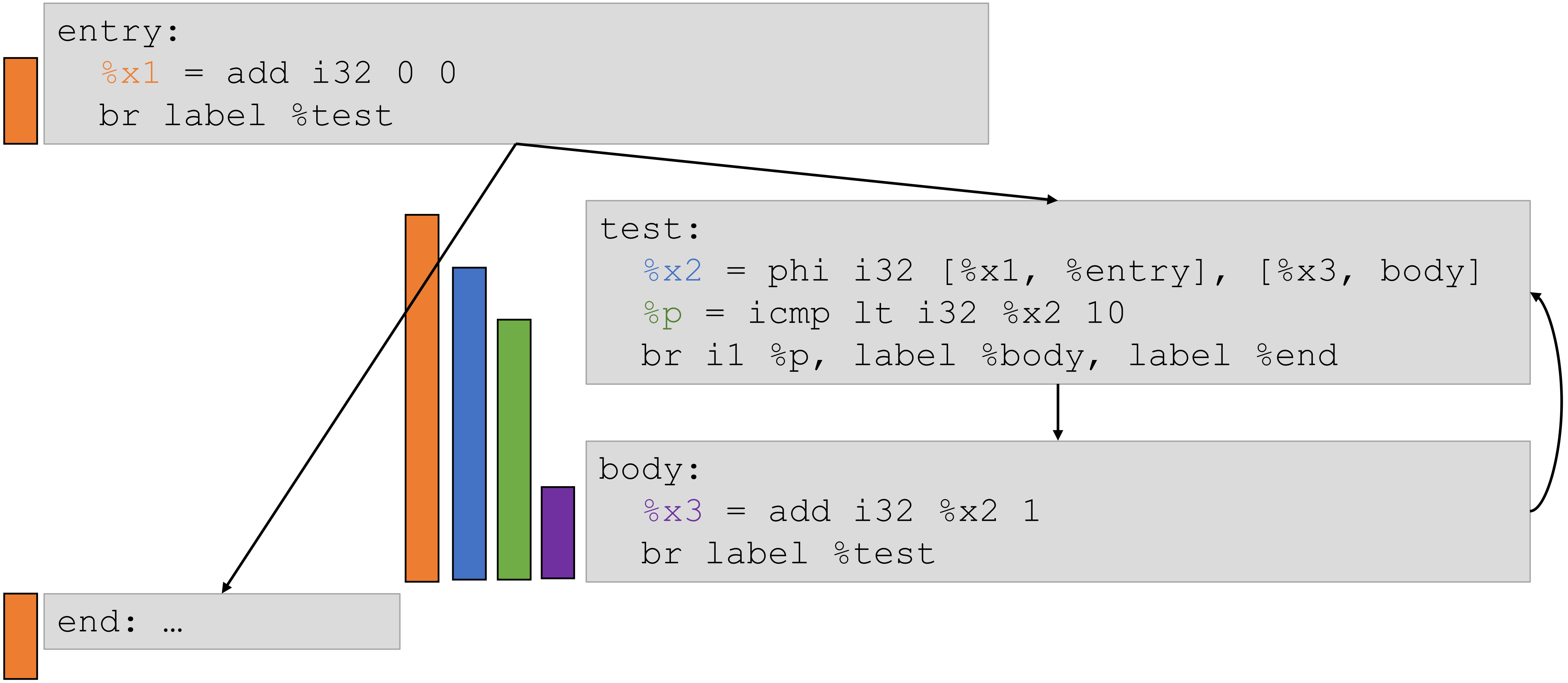
```
entry:
   %x1 = add i32 0 0
   br label %test

test:
   %x2 = phi i32 [%x1, %entry], [%x3, body]
   %p = icmp lt i32 %x2 10
   br i1 %p, label %body, label %end

body:
   %x3 = add i32 %x2 1
   br label %test

end: …
```

What if I show it this way?



```
entry:
  %x1 = add i32 0 0
  br label %test
```

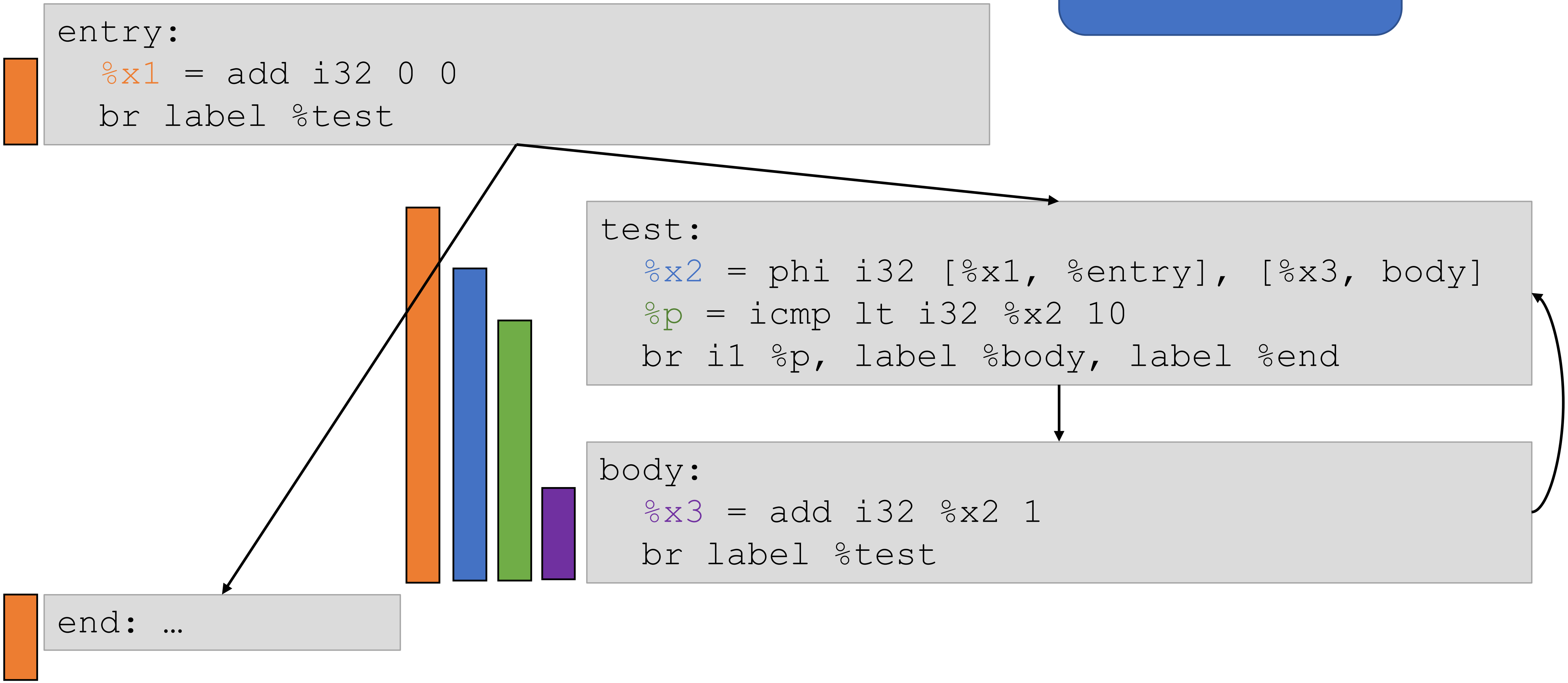
```
test:
  %x2 = phi i32 [%x1, %entry], [%x3, body]
  %p = icmp lt i32 %x2 10
  br i1 %p, label %body, label %end
```

```
body:
  %x3 = add i32 %x2 1
  br label %test
```

```
end: …
```

Scope is based on *dominance*

“Definition must dominate all uses”



```
entry:
   %x1 = add i32 0 0
   br label %test
```

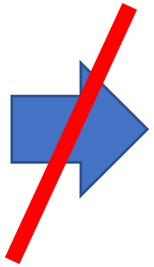
```
test:
   %x2 = phi i32 [%x1, %entry], [%x3, body]
   %p = icmp lt i32 %x2 10
   br i1 %p, label %body, label %end
```

```
body:
   %x3 = add i32 %x2 1
   br label %test
```

```
end: …
```

Aside #2: Why does LLVM not have a move instruction?

```
int sqrt(int n) {
  int i = n;
  while (i * i > n)
    i--;
  return i;
}
```



```
define @sqrt(i32 %n) {
  %i1 = mov i32 %n
  br label %test
test:
  %i2 = phi i32 [%i1, %entry], [%i3, %body]
  %sq = mul i32 %i2 %i2
  %p = icmp i32 gt %sq %n
  br i1 %p, label %body, label %end
body:
  %i3 = sub i32 %i2 1
  br label %test
end:
  ret i32 %i2
```

Declares %i1 to be %n. What’s the point?

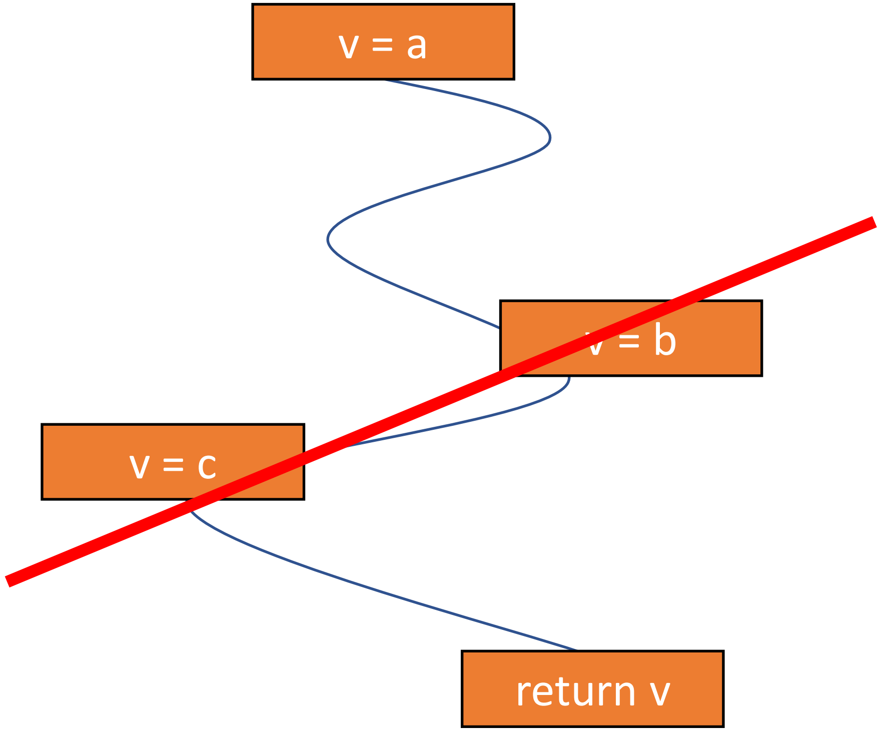
Aside #2: Why does LLVM not have a move instruction?

```
int sqrt(int n) {
  int i = n;
  while (i * i > n)
    i--;
  return i;
}
```

```
define @sqrt(i32 %n) {
  br label %test
test:
  %i2 = phi i32 [%n, %entry], [%i3, %body]
  %sq = mul i32 %i2 %i2
  %p = icmp i32 gt %sq %n
  br i1 %p, label %body, label %end
body:
  %i3 = sub i32 %i2 1
  br label %test
end:
  ret i32 %i2
```

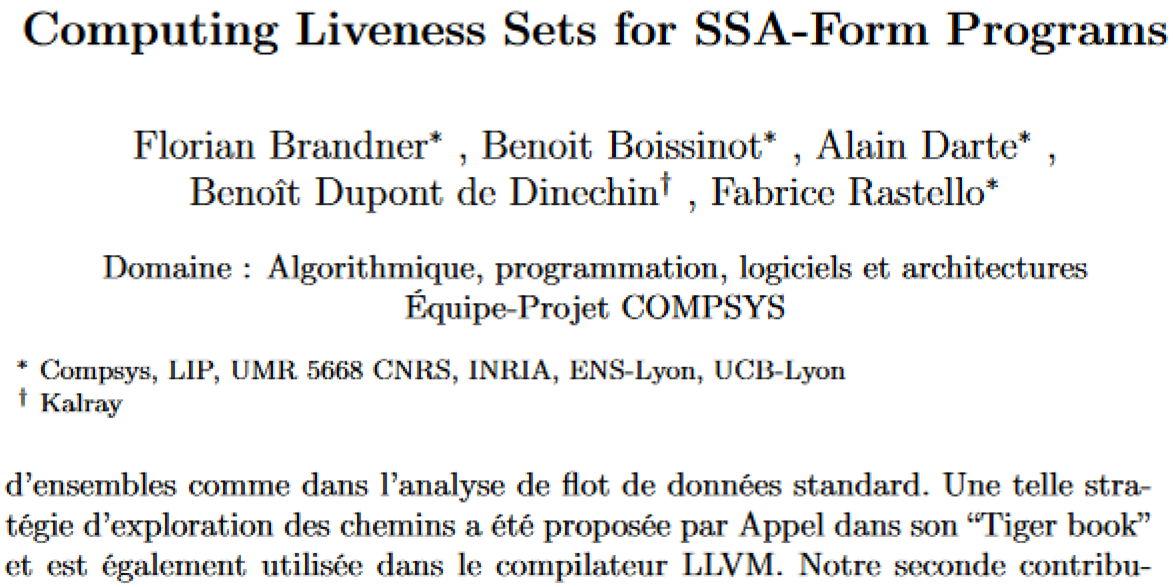
(Related to copy propagation optimization—next class)

Liveness, revisited: just propagate back from uses to (the only) def



Liveness no longer needs iterative dataflow!

- Appel, LLVM compiler: propagate from uses of each var back to def
- This paper: like a dataflow analysis, but just 2 passes!



Computing Liveness Sets for SSA-Form Programs

Florian Brandner* , Benoit Boissinot* , Alain Darte* ,
Benoît Dupont de Dinechin† , Fabrice Rastello*

Domaine : Algorithmique, programmation, logiciels et architectures
Équipe-Projet COMPSYS

* Compsys, LIP, UMR 5668 CNRS, INRIA, ENS-Lyon, UCB-Lyon
† Kalray

d'ensembles comme dans l'analyse de flot de données standard. Une telle stratégie d'exploration des chemins a été proposée par Appel dans son "Tiger book" et est également utilisée dans le compilateur LLVM. Notre seconde contribu-

Reaching definitions is essentially irrelevant!

- Is %x4 in scope? Then the (one) definition of %x4 reaches here
- (Not surprising, as you can think of converting to SSA as based on reaching definitions)