Compilers I

Introduction to Compiler Optimization

Dr. William L. Harrison
harrisonwl@missouri.edu

Today's Class

Today: Optimizations

- Basic Optimizations
- Static Single-Assignment

What does an optimizing compiler do?

Optimizing compiler attempts to:

- Eliminate language abstractions
- Map source program to target machine efficiently.
  - Use the hardware well!
- Equal the efficiency of a good assembly programmer.

Dimensions of Optimization

You can optimize for a number of things.

- machine cycles (speed)
- binary size
- heap usage
- to use special hardware features
  - parallel computing

Minimizing runtime cycles is the typical measure to optimize.

Proebsting's Law

- Moore’s Law: Computing power doubles every 18 months.
  - This has been true for 40 years!
- Proebsting's Law: **Compiler Advances** double computing power every 18 *Years*
  - Better optimizations
  - Better use of new hardware features

Optimizing compilers allow efficient abstractions

- We can engineer our software applications using abstractions like OO-style objects, etc.
  - Ease of maintenance.
  - Almost all design patterns are abstractions.
- The compiler can remove much of the overheads associated with using these abstractions.

Appel's Compiler Theorem

Theorem: For any optimizing compiler, there exists a better one.

"*full employment theorem for compiler writers…*"

Appel's point was you can always add a new optimizations to make any compiler better.

What is an optimization?

- The output program must have the same **observable behavior**, as defined in the semantics of the source language.

```
int foo(int i) {
  int j = 9;
  return i + j;
}
```

≈

```
int foo(int i) {
  return i+9;
}
```

*

Optimizations can make your working program fail!

- The output program must have the same **observable behavior**, as defined in the semantics of the source language.
- If a program steps outside the semantically defined part of the language, all bets are off.
  - Example: using an un-initialized variable in C.
    - The value might be different between optimized and non-optimized.
- Your program was faulty to start with!
  - It just happened to work…

Optimizations

- Optimizations change the program internally
- This enables more optimizations…
  - Some optimizations are ***enabling*** optimizations
  - Some optimizations are ***true*** optimizations

Optimization Camps

Optimizations that move computation from runtime to compile time.

- *Examples: #ifdef, asserts, constant folding*

Optimizations that replace some instructions with less expensive instructions.

- *Examples: peephole optimizations*

Optimizations that move a computation to a less expensive place.

- *Examples: loop invariant hoisting*

Compile Time Evaluation

Sometimes constants can be evaluated at compile time.

Caveats

x must be constant.

```
final int x = 9;
if (x < 10) {
    …
}
```

```
final int x = 9;
if (true) {
    …
}
```

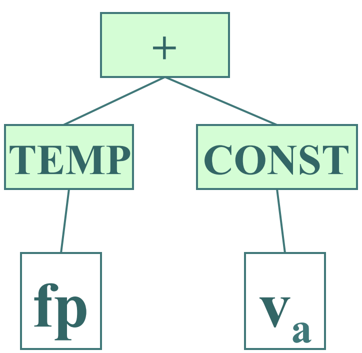
Constant folding

- Constant folding is a <u>true</u> optimization
  - The work that would have been done at runtime is no longer needed.
- Constant folding is **also** an enabling optimization.
  - On previous page, we can eliminate the if/then test.
  - Constant folding can give rise to **dead-code.**

Peephole Optimizations

Consider these instructions

ADDI	$r_1 \leftarrow \mathbf{V_a}$
ADD	$r_2 \leftarrow \mathbf{fp} + r_1$
LOAD	$r_3 \leftarrow M[r_2 + 0]$



But

- **V_a is a constant.**
- **fp is a register.**

→ so we could write

LOAD $r_3 \leftarrow M[\mathbf{fp} + \mathbf{V_a}]$

Works inside basic blocks

Need to know about usage of r_1 and r_2.

Procedure inlining

Two steps to inlining

- Substitute procedure body at call site
- Watch for name clashes
  - Fix variable names if clash

```
int foo(int x) {
      return x * x;
}

…
      v := foo(y);
…
```

Saves cost of function call

```
…
      v := y * y;
…
```

*

Loop invariants

Assuming **v** does not change inside this loop

Can "lift" the computation of `v * 2` out of the loop.

```
v := …
…
while(x < 10)
   {
   y = v * 2;
   x = x + y;
}
```

```
v := …
. . .
y := v * 2;
while (x < 10) {
     x = x + y;
}
```

It is almost always good to move things out of loops.

Common sub-expression elimination

We can cache the value of v * 3.

Assuming **v** doesn't change

- depends on data flow analysis!

```
x := foo(v * 3);


if (v * 3 < 9) {
      …
}
```

```
t := v * 3;
x := foo (t);

if (t < 9) {
      …
}
```

*

Where is the pattern?

We have seen some optimizations.

- All seem ad-hoc.
- All depend on knowing something about their **context**…

- Some optimizations are local.
- Some optimizations act over multiple basic blocks.

Most optimizing compilers now use SSA-form to unify many optimizations.

*

A survey of optimizations

Optimizing compilers

There's more to performance than asymptotic complexity

- Constant factors matter too!
  - Easily see 10:1 performance range depending on how code is written
  - Must optimize at multiple levels:
    - algorithm, data representations, procedures, and loops
- Understand target system to optimize performance
  - How programs are compiled and executed
  - How to measure program performance and identify bottlenecks
  - How to improve performance without destroying code modularity and generality

Optimizing compilers

- Provide efficient mapping of program to machine
  - register allocation
  - code selection and ordering (scheduling)
  - dead code elimination
  - eliminating minor inefficiencies
- Don’t (usually) improve asymptotic efficiency
  - up to programmer to select best overall algorithm
  - big-O savings are (often) more important than constant factors (but constant factors can make a big difference)
- Have difficulty overcoming “optimization blockers”
  - potential memory aliasing
  - potential procedure side-effects

Limitations of optimizing compilers

- Fundamental Constraint
  - Must not cause any change in observable program behavior under any possible condition
    - Can prevent making optimizations when would only affect behavior under pathological conditions.
- Behavior that may be obvious to the programmer can be obfuscated by languages and coding styles
  - e.g., Data ranges may be more limited than variable types suggest
- Most analysis is performed only within procedures
  - Whole-program analysis is too expensive in most cases
- Most analysis is based only on *static* information
  - Compiler has difficulty anticipating run-time inputs

When in doubt, the compiler must be conservative

What to optimize?

You can optimize for a number of things:

- machine cycles (speed)
- binary size
- heap usage
- to use special hardware features
  - parallel computing
- compile time (?!)

What to optimize?

You can optimize for a number of things:

- machine cycles (speed)
- binary size
- heap usage
- to use special hardware features
  - parallel computing
- compile time (?!)

(what we usually care about most)

Proebsting's Law

- Moore's Law: computing power (transistor density) doubles every 18 months.
- Proebsting's Law: advances in compiler optimizations double computing power every 18 *years.*

Optimization: a transformation that (hopefully) improves performance

The output program must have the same **observable behavior**, as defined in the semantics of the source language.

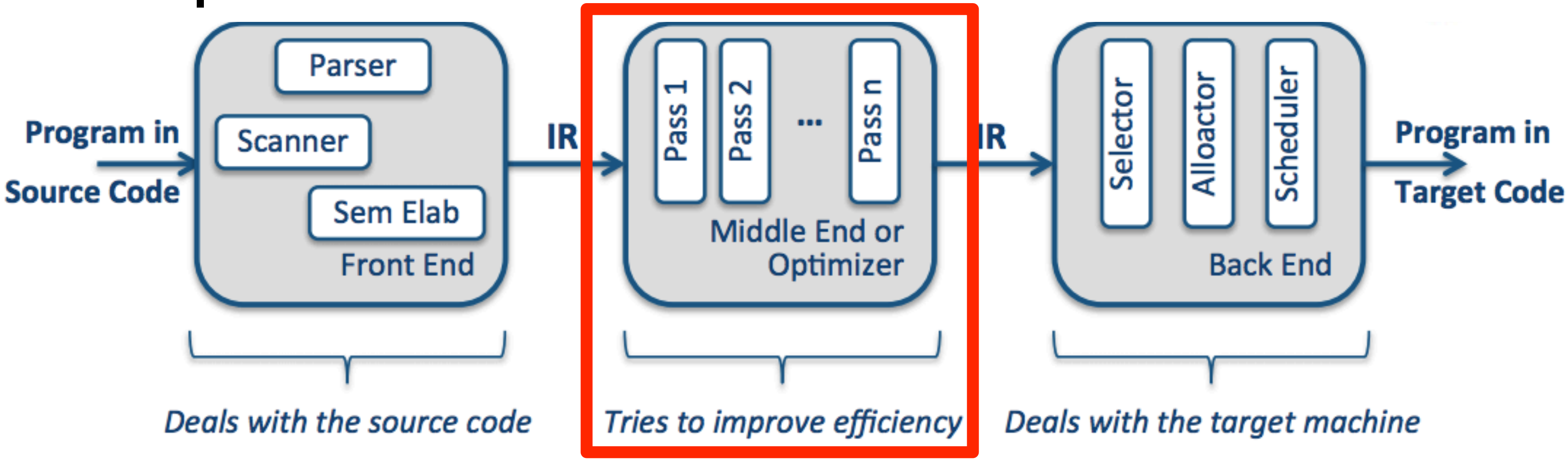
```
int foo(int i) {
  int j = 9;
  return i + j;
}
```

≈

```
int foo(int i) {
  return i + 9;
}
```

*

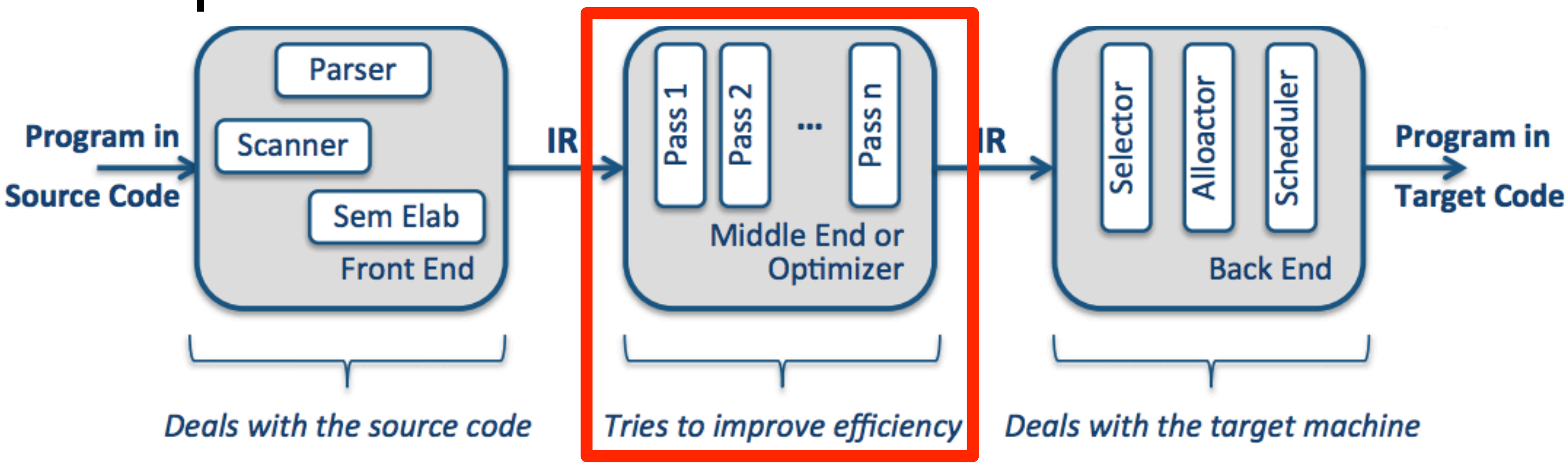
Middle end optimizations



Optimizations in the middle end are backend-agnostic and usually operate on the CFG, taking advantage of various dataflow analyses.

*

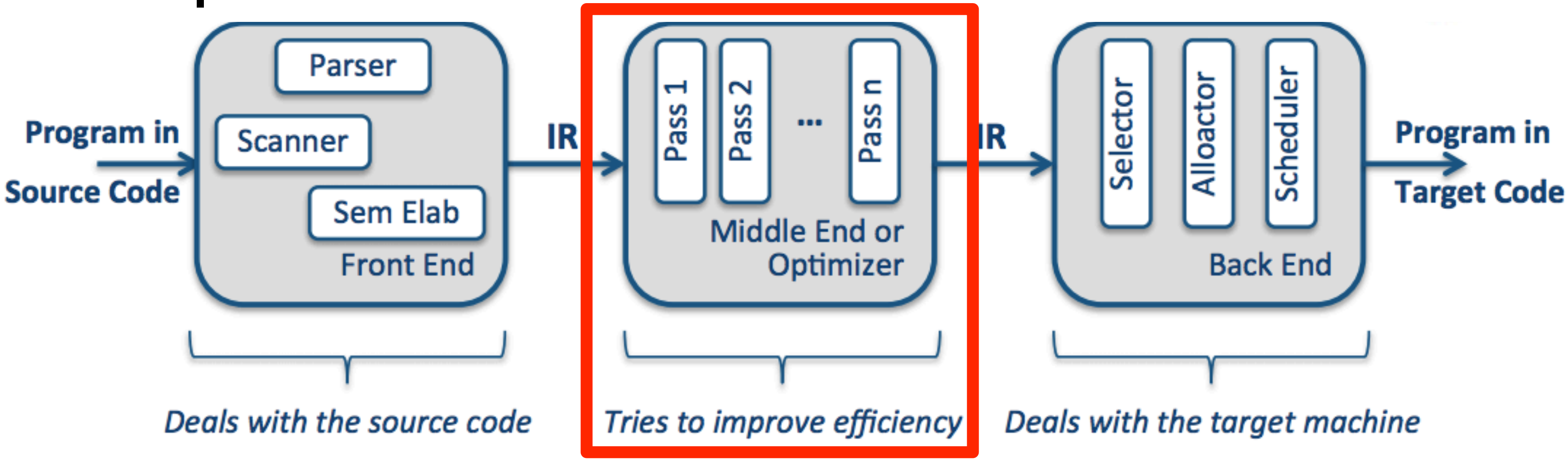
Middle end optimizations



- Constant folding/propagation
- Copy propagation
- Algebraic simplification
- Common subexpression elimination
- Dead code elimination
- Loop optimizations
- Function inlining

*

Middle end optimizations



- Constant folding/propagation
- Copy propagation
- Algebraic simplification
- Common subexpression elimination
- Dead code elimination
- Loop optimizations
- Function inlining

Constant folding

- Evaluate constant expressions at compile time
- Only possible when side-effect freeness guaranteed

```
c ← 1 + 3
```

➡

```
c ← 4
```

```
not true
```

➡

```
false
```

Constant propagation

Variables that have constant value, e.g. c := 3

- Later uses of c can be replaced by the constant
- If no change of c between!

```
b ← 3
c ← 1 + b
d ← b + c
```

```
b ← 3
c ← 1 + 3
d ← 3 + c
```

Analysis needed, as b can be assigned more than once!

Constant propagation

Variables that have constant value, e.g. c := 3

- Later uses of c can be replaced by the constant
- If no change of c between!

Ready for constant folding now.

```
b ← 3
c ← 1 + b
d ← b + c
```

```
b ← 3
c ← 1 + 3
d ← 3 + c
```

Analysis needed, as b can be assigned more than once!

Copy propagation

- for a statement x ← y
- replace later uses of x with y, if x and y have not been changed.

```
x ← y
c ← 1 + x
d ← x + c
```

```
x ← y
c ← 1 + y
d ← y + c
```

Analysis needed, as y and x can be assigned more than once!

Algebraic simplification

Use algebraic properties to simplify expressions

`-(-i)`	➡	`i`
`b or true`	➡	`true`

*

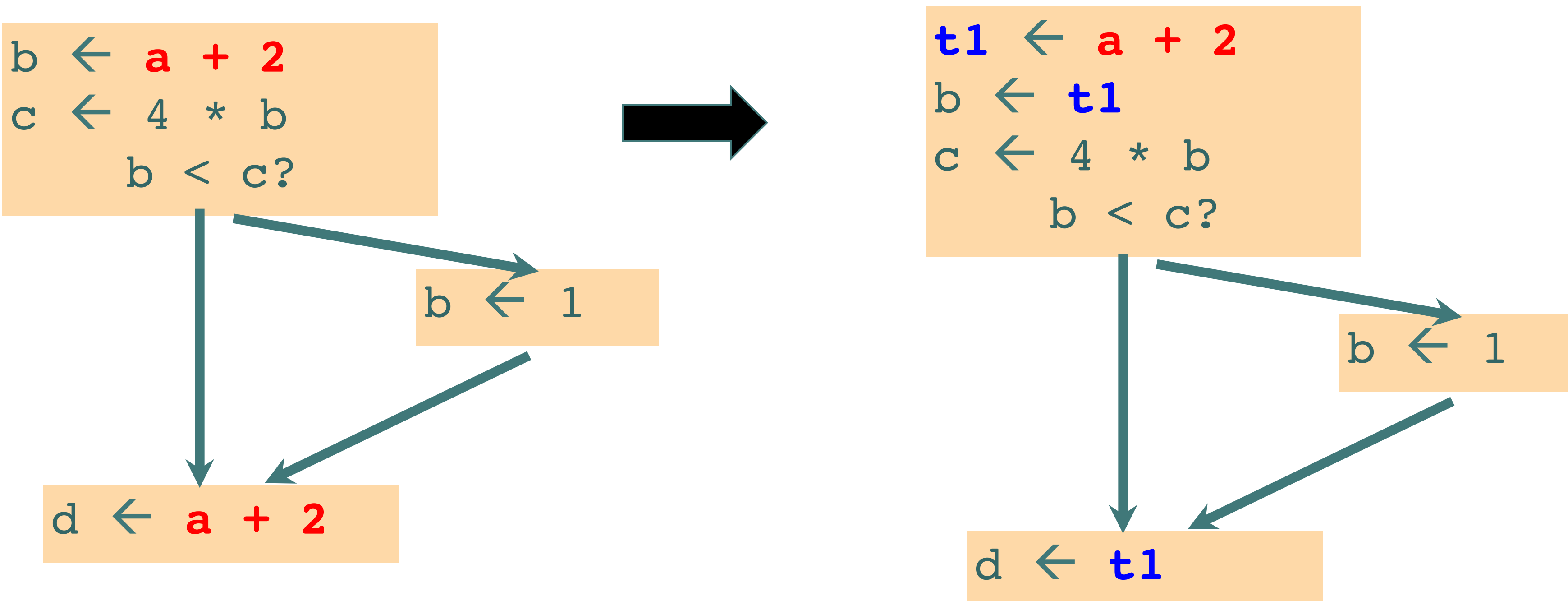
Common subexpression elimination

- "Cache" an expression to avoid re-evaluation.
- Only works if the registers used in the expression remain unchanged between uses!

```
x = foo(v * 3);


if (v * 3 < 9) {
      …
}
```

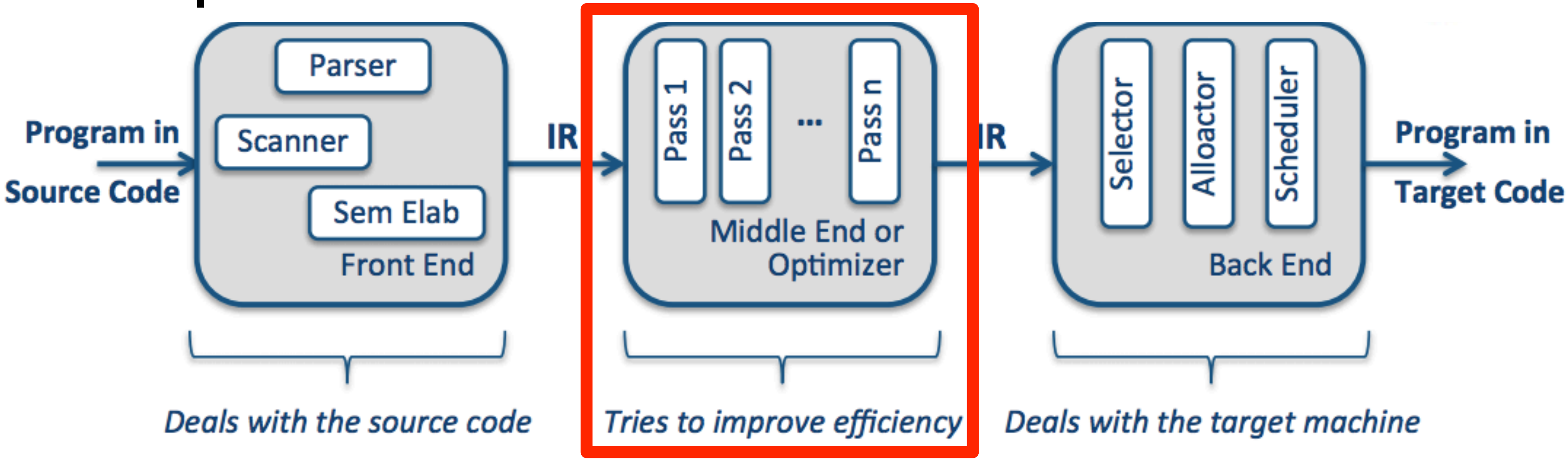
```
t = v * 3;
x = foo(t);

if (t < 9) {
      …
}
```

Common subexpression elimination



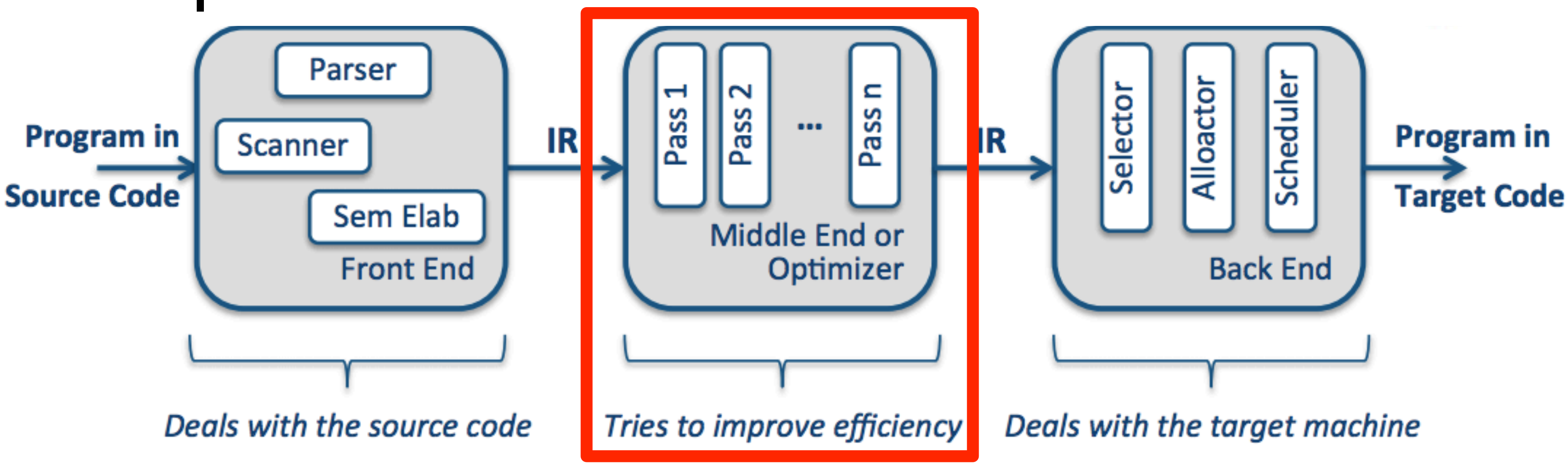
The Moral: inter-block CSE trickier than intra-block CSE

Middle end optimizations



- Constant folding/propagation
- Copy propagation
- Algebraic simplification
- Common subexpression elimination
- Dead code elimination
- Loop optimizations
- Function inlining

Middle end optimizations



- Constant folding/propagation
- Copy propagation
- Algebraic simplification
- Common subexpression elimination
- Dead code elimination
- Loop optimizations
- Function inlining

*

Dead code elimination

- Remove unnecessary code
  - e.g. variables assigned but never read:

```
b ← 3
c ← 1 + 3
d ← 3 + c
```

➡

```
c ← 1 + 3
d ← 3 + c
```

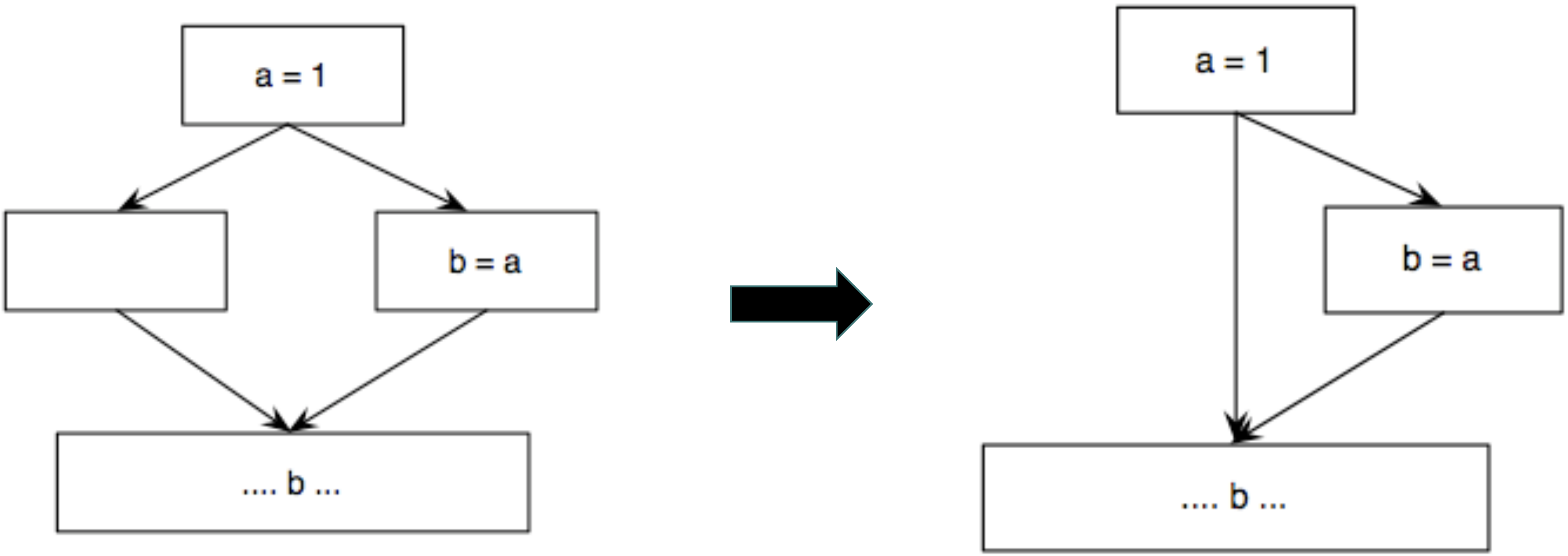
– blocks never reached:

```
if (false) {
      a ← 5
}
```

➡

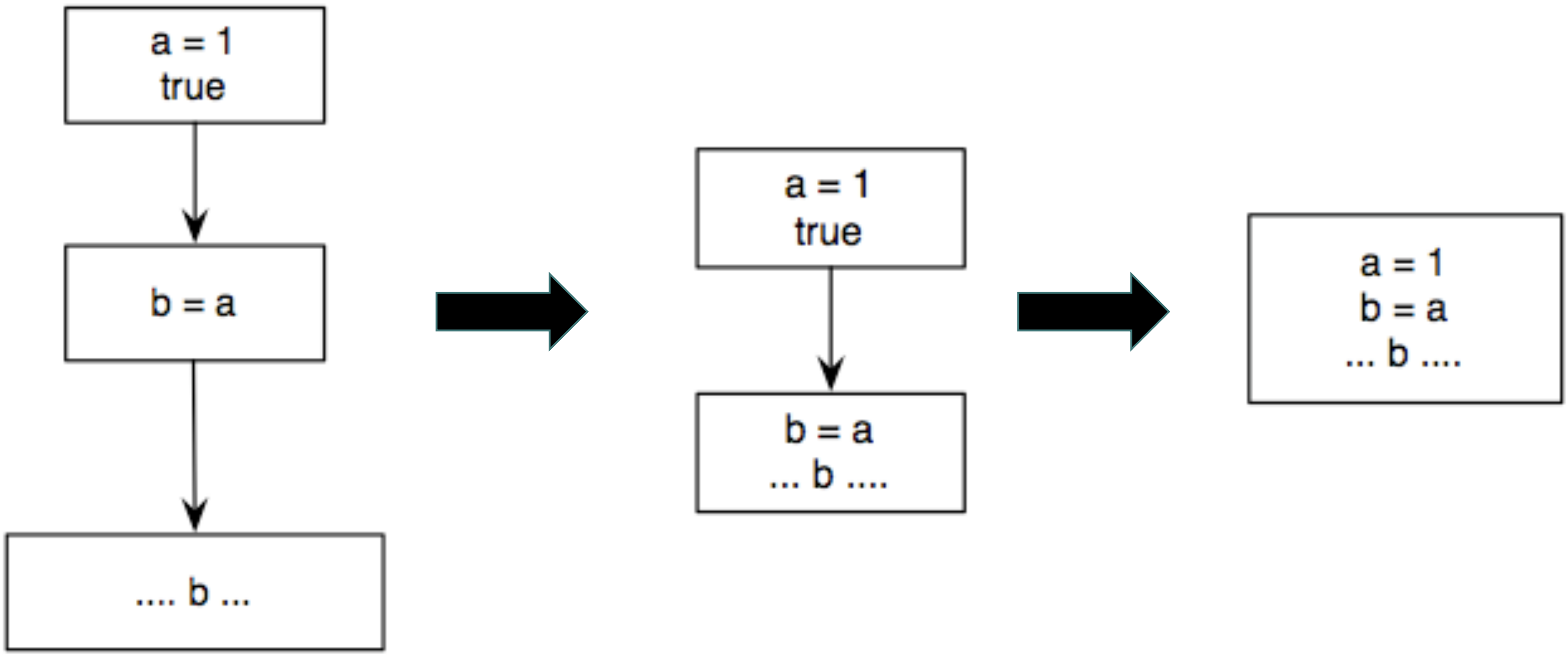
```
if (false) {}
```

Dead code elimination



*

Other CFG simplifications: "Fusion" of basic blocks



AKA, "Code Straightening"

*

DU, UD chains

- DU chain = “definition use” chain
  - directed arc(s) from each variable definition to the use(s) of that variable
- UD chain = “use definition”
  - directed arc(s) from a variable use to the instruction defining that variable
- Both are implemented as graphs
  - common technique before SSA

Example: DU, UD chains

def-use

$x \leftarrow \ldots$

$\ldots$

$z \leftarrow x + y$

$\ldots$

$z' \leftarrow x$

no def's of x

use-def

$x \leftarrow \ldots$

$\ldots$

$z \leftarrow x + y$

$\ldots$

$z' \leftarrow x$

Static Single-Assignment

- Invariant on instruction stream
  - Every virtual register has **one** (static) definition site
  - **Never** re-assign a virtual register.

This is straightforward for straight-line code.

$$
\begin{aligned}
a &\leftarrow x * y \\
b &\leftarrow a - 1 \\
a &\leftarrow y * b \\
b &\leftarrow x * 4 \\
a &\leftarrow a + b
\end{aligned}
$$

$$
\begin{aligned}
a_1 &\leftarrow x * y \\
b_1 &\leftarrow a_1 - 1 \\
a_2 &\leftarrow y * b_1 \\
b_2 &\leftarrow x * 4 \\
a_3 &\leftarrow a_2 + b_2
\end{aligned}
$$

*

SSA (2)

$$\begin{aligned} a &\leftarrow x * y \\ b &\leftarrow a - 1 \\ a &\leftarrow y * b \\ b &\leftarrow x * 4 \\ a &\leftarrow a + b \end{aligned}$$

$$\begin{aligned} a_1 &\leftarrow x * y \\ b_1 &\leftarrow a_1 - 1 \\ a_2 &\leftarrow y * b_1 \\ b_2 &\leftarrow x * 4 \\ a_3 &\leftarrow a_2 + b_2 \end{aligned}$$

- a_1, a_2, a_3 are distinct virtual registers.
- They *may* map to the same physical register.

SSA (3)

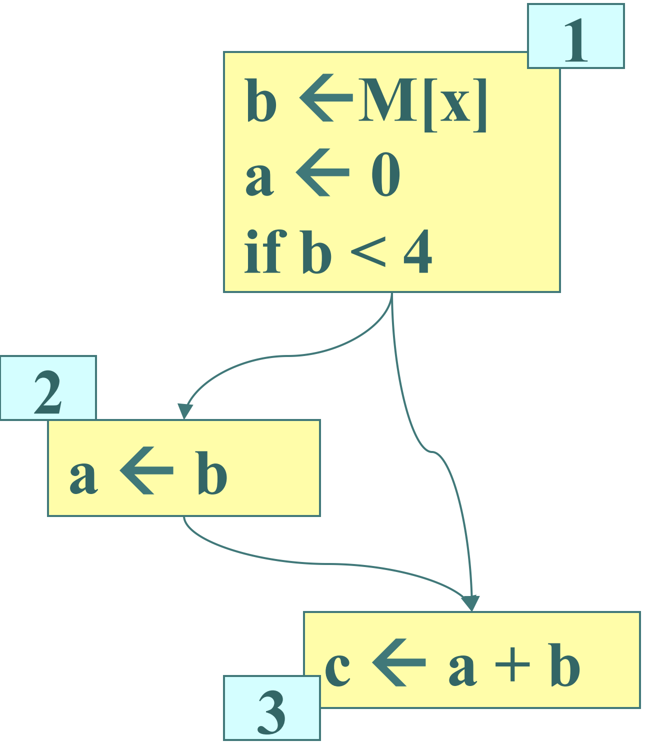
$a \leftarrow x * y$
$b \leftarrow a - 1$
$a \leftarrow y * b$
$b \leftarrow x * 4$
$a \leftarrow a + b$

$a_1 \leftarrow x * y$
$b_1 \leftarrow a_1 - 1$
$a_2 \leftarrow y * b_1$
$b_2 \leftarrow x * 4$
$a_3 \leftarrow a_2 + b_2$

- We now **know** the value of a_1 **for all time.**
- It is **never** reassigned.
  - ➔ constant folding, etc become easier.
  - if (a_1 < 10) { …}

*

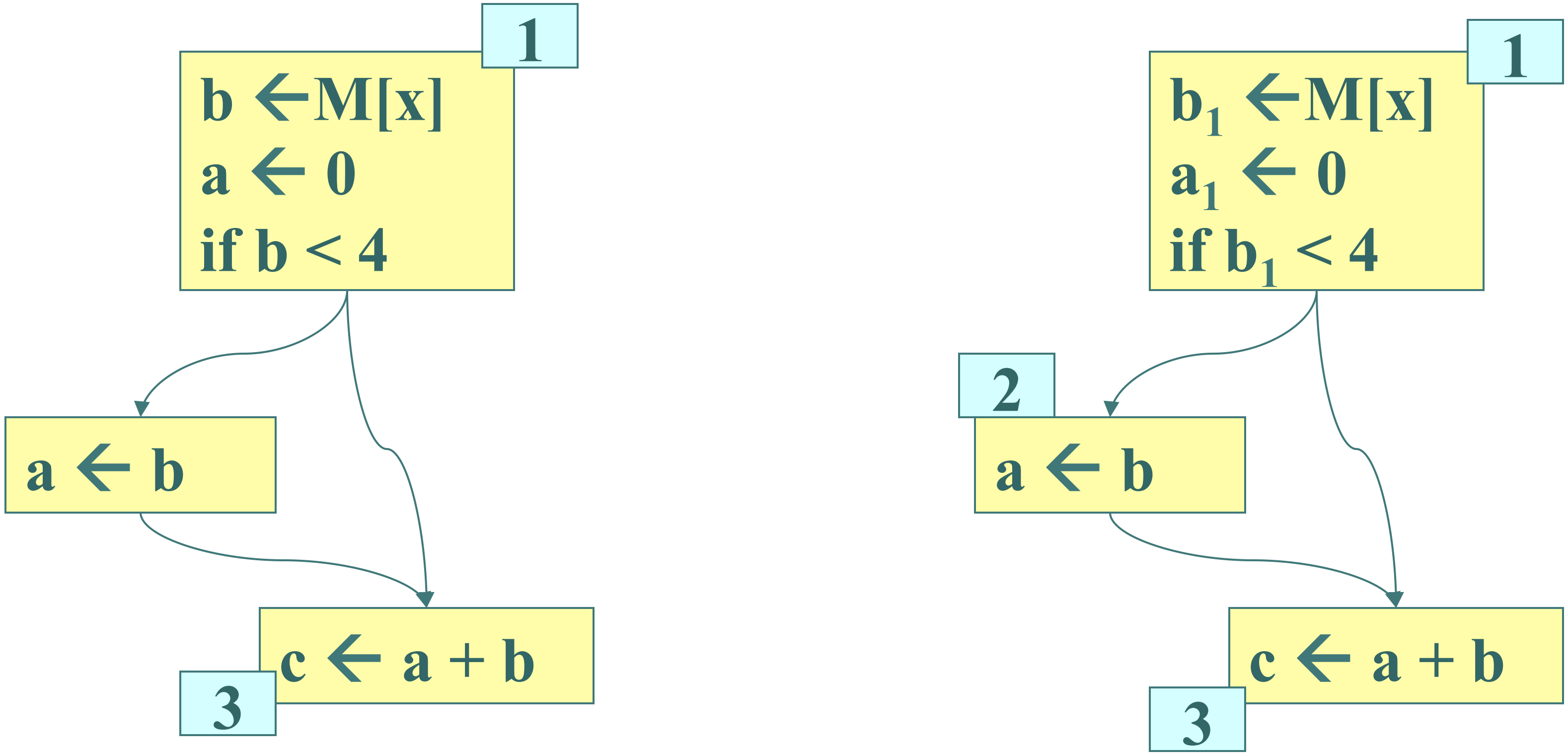
SSA Control Flow (1)



What about control flow?

- How do we give SSA definitions for a and c?

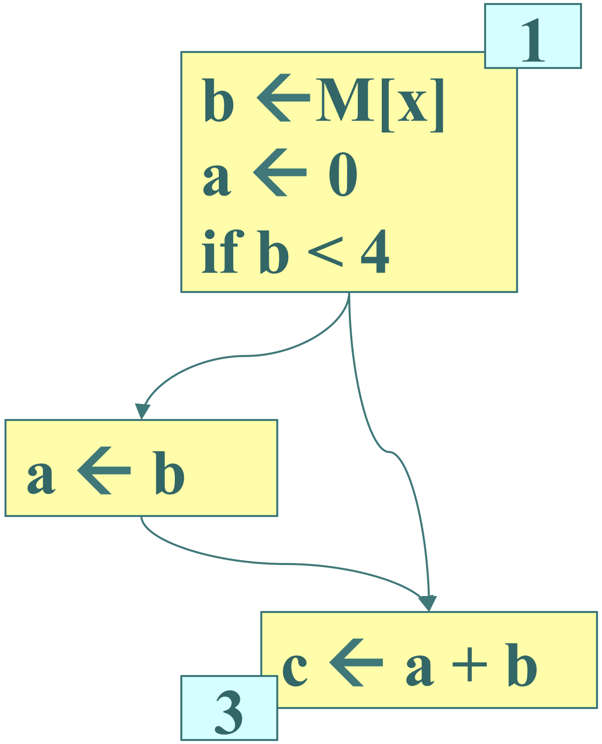
SSA Control Flow (2)

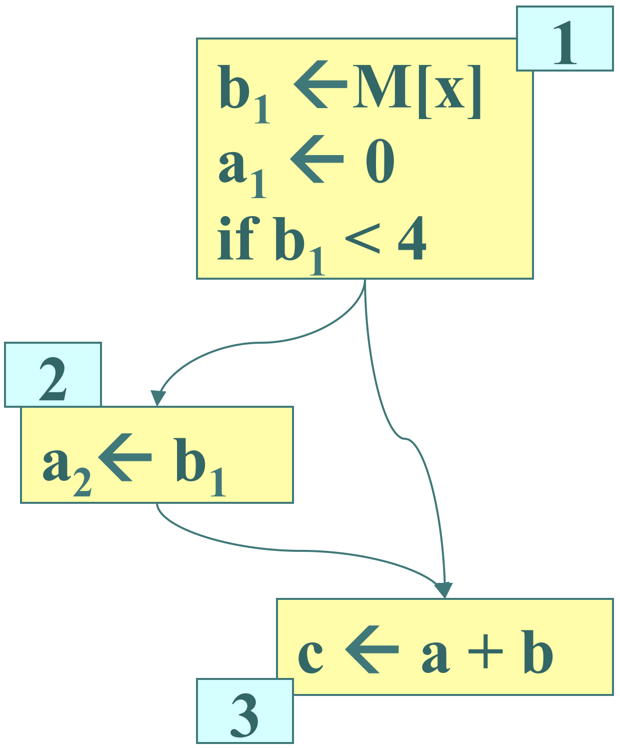


Block 1 is straightforward…

*

SSA Control Flow (3)

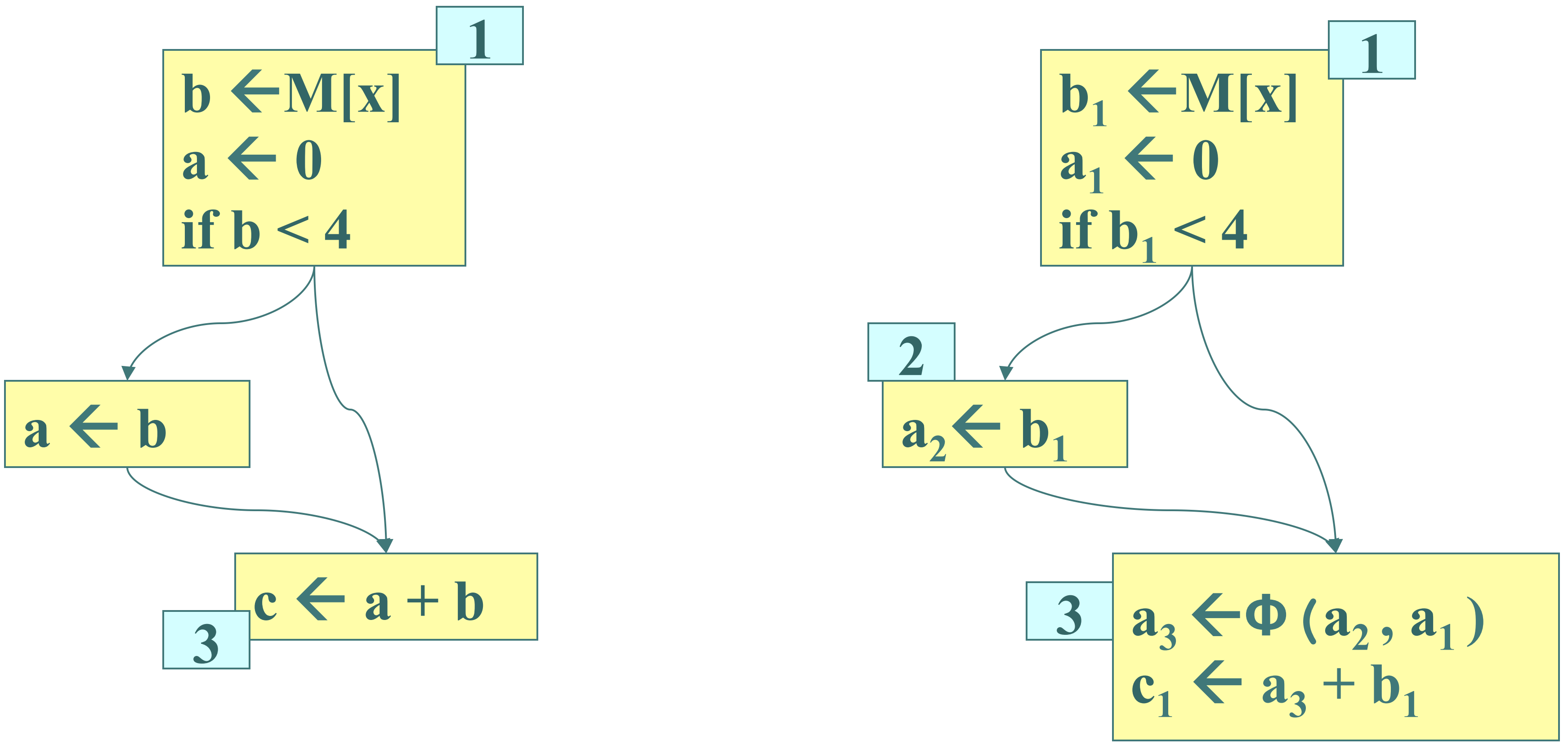




In block 2, we define a new a.

Which 'a' do we use in block 3?

SSA Control Flow (4)

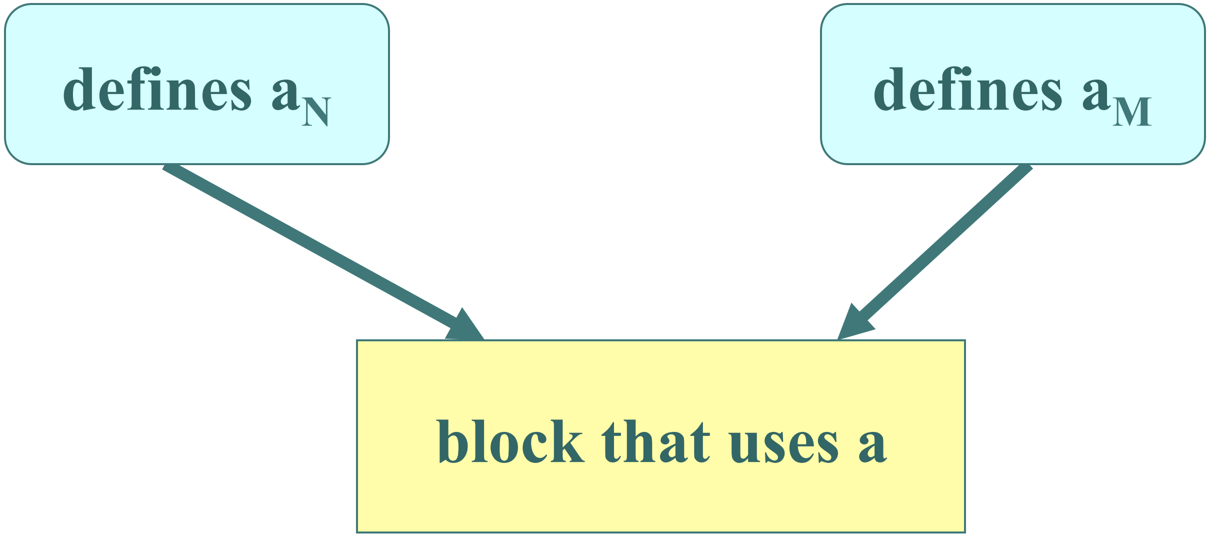


Use a **Φ**-node…

*

Φ-node?

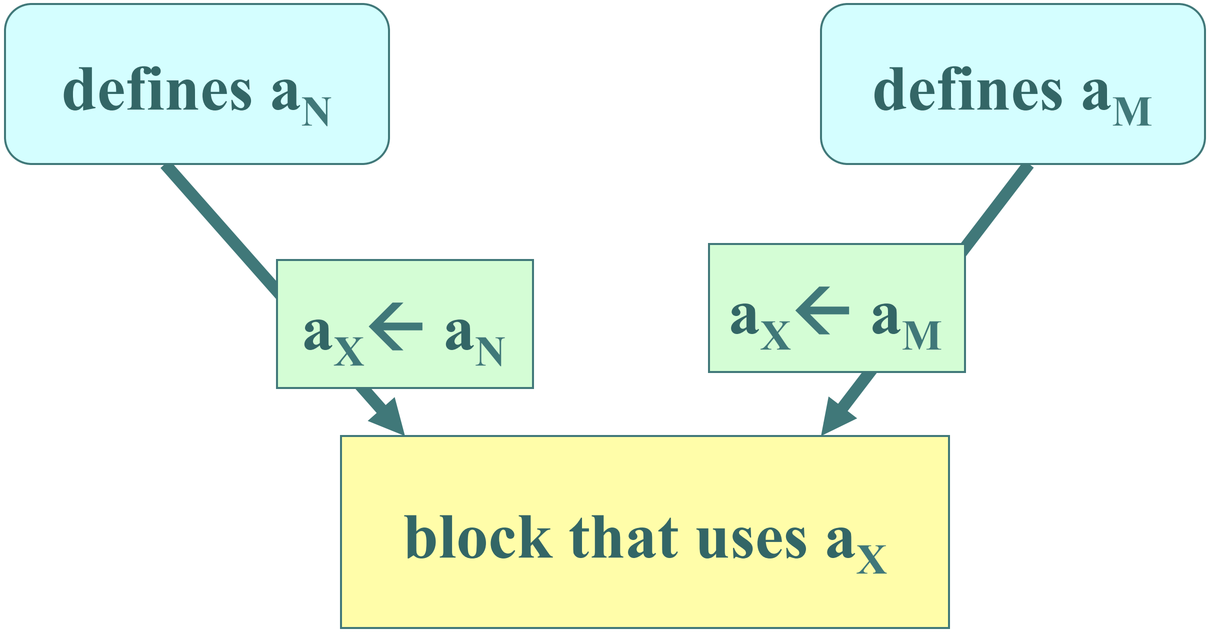
Φ-nodes: means something like a "choice" between values a_N and a_M



*

The Φ-node

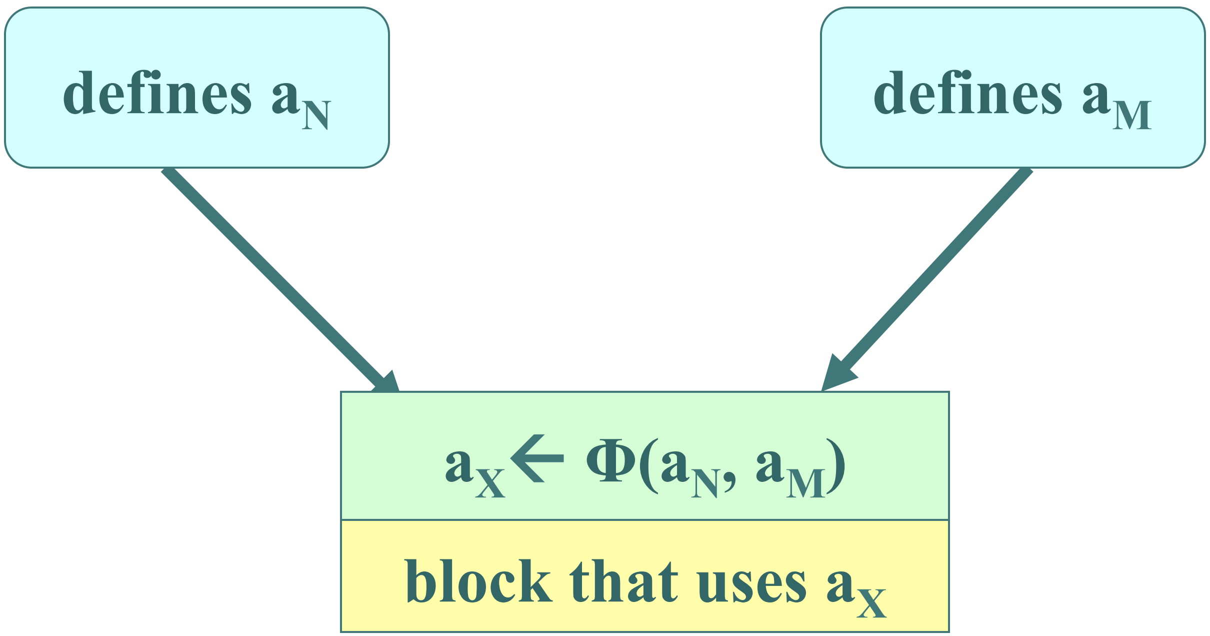
Φ-nodes: means something like a "choice" between values a_N and a_M



*

The Φ-node

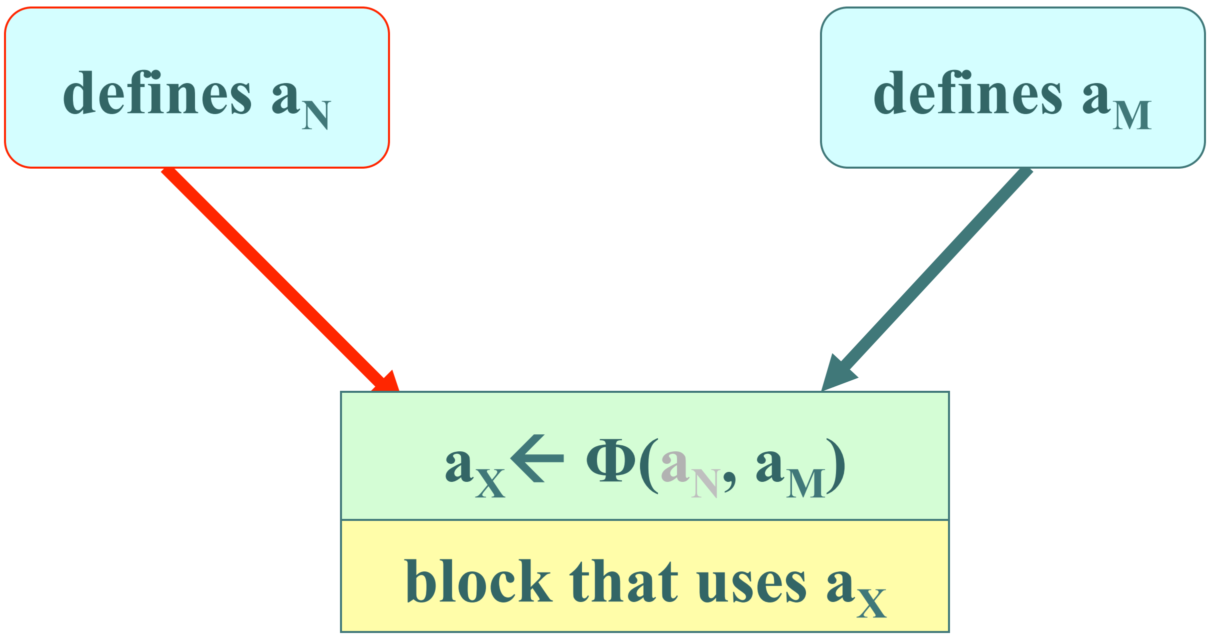
SSA is one of the most important new idioms for compiler construction in the last 20 years!



*

The Φ-node

If you come from the left-hand basic block, the **Φ** operation is a copy from a_N to a_X



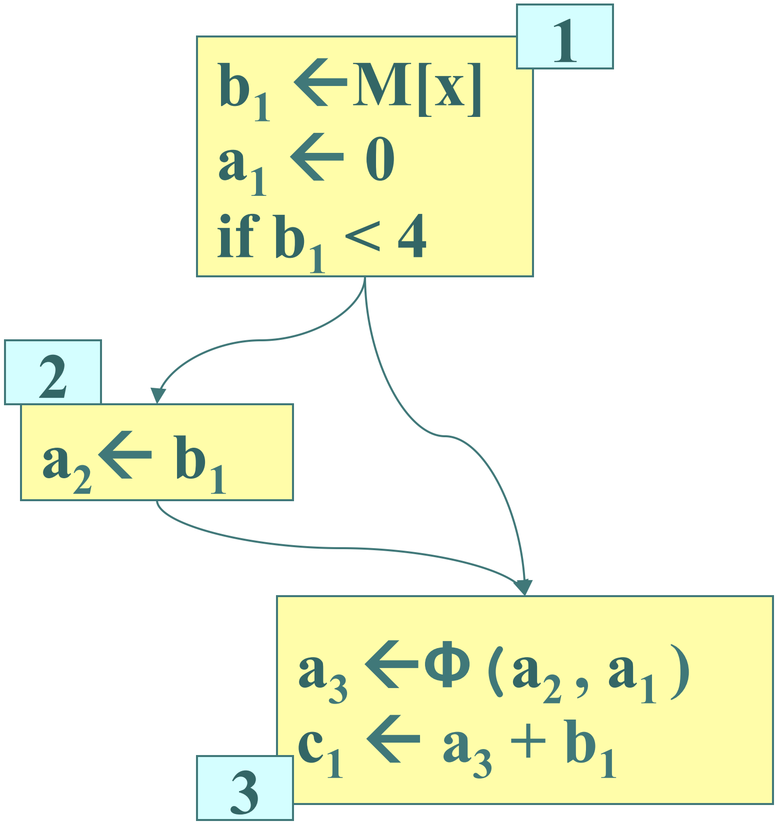
The Φ-node

Think of the **Φ** operation as a "magic move" operation, that copies one of its operands to its destination.

$a_X \leftarrow \Phi(a_N, a_M)$
block that uses a_X

*

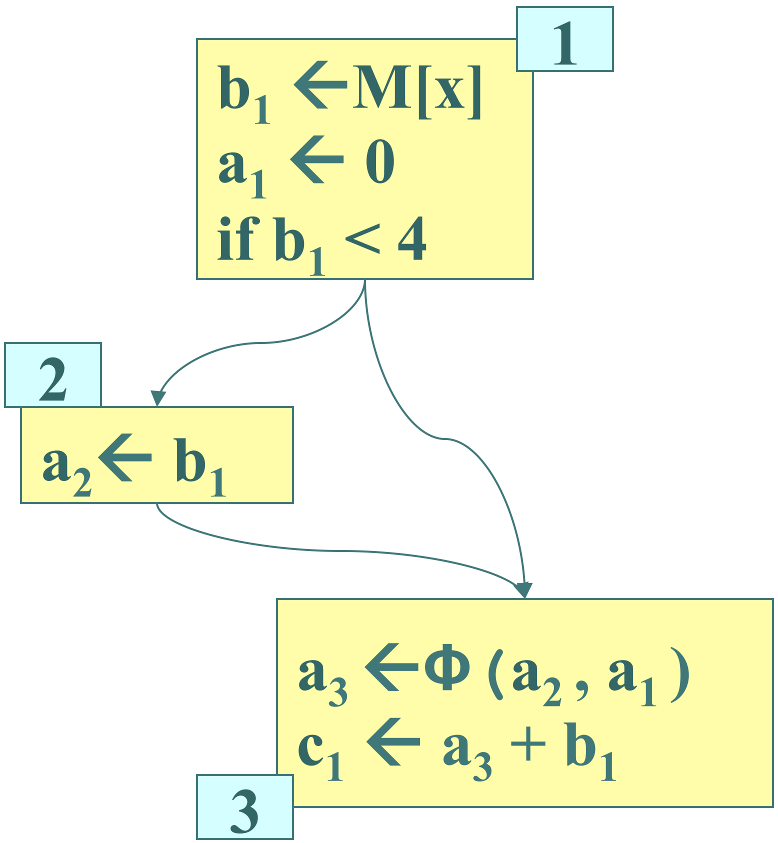
SSA Control Flow (5)



What about instruction generation?

*

SSA Control Flow (6)



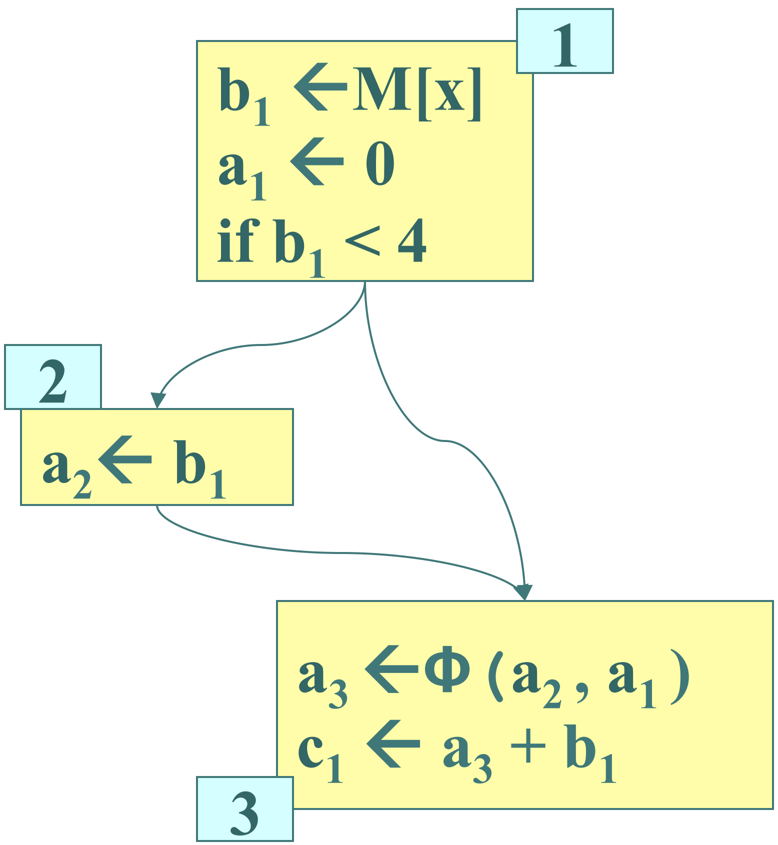

```
    r1 ← M[x]
    r2 ← 0
    if (r1 < 4) goto L1:
    goto L2:
L1:
    r3 ← r1
    goto L3:
L2:
    goto L3:
L3:
    r5 ← ?? + r1
```

may be r2 or r3

What about instruction generation?

*

SSA Control Flow (7)




```
    r1 ← M[x]
    r2 ← 0
    if (r1 < 4) goto L1:
    goto L2:
L1:
    r3 ← r1
    r4 ← r3
    goto L3:
L2:
    r4 ← r2
    goto L3:
L3:
    r5 ← r4 + r1
```

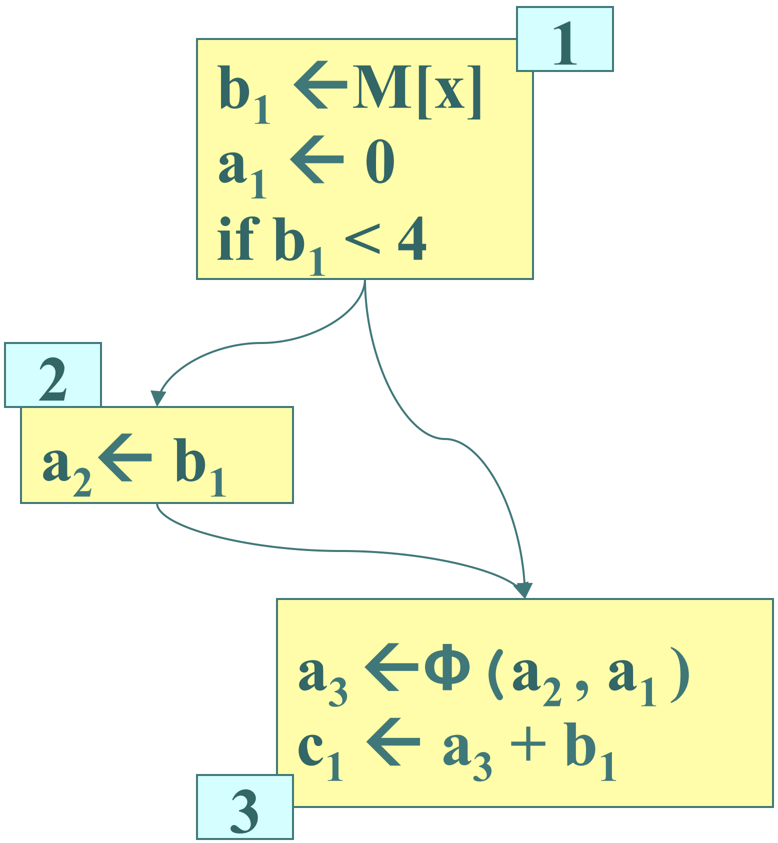
Φ implemented via moves.

Insert extra moves…

Note: r4 is new

*

SSA Control Flow (8)



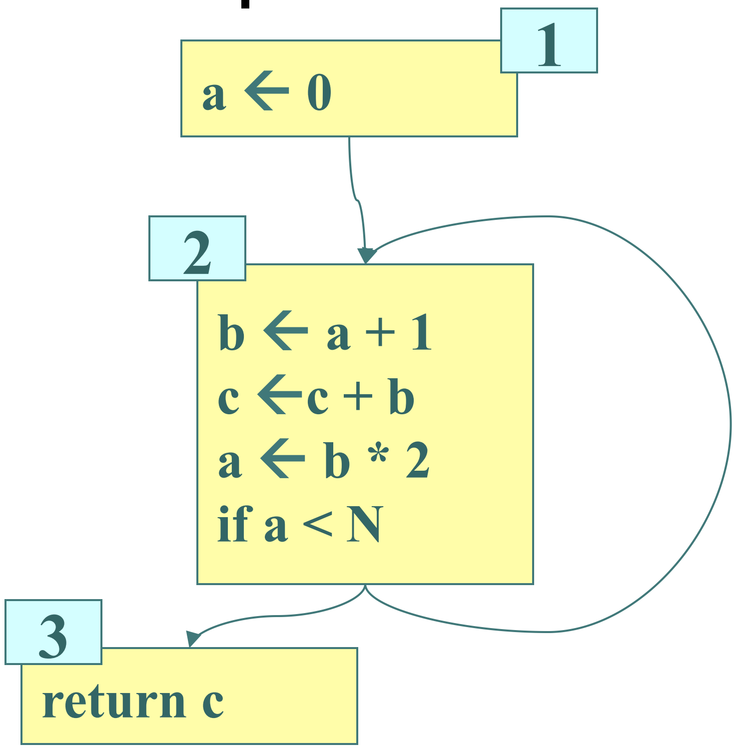

```
    r1 ← M[x]
    r2 ← 0
    if (r1 < 4) goto L1:
    goto L2:
L1:
    r2 ← r1
    r2 ← r2
    goto L3:
L2:
    r2 ← r2
    goto L3:
L3:
    r5 ← r2 + r1
```

Best case: if all ‘a’ s map to r2 (depends on register allocation)

Φ Summary

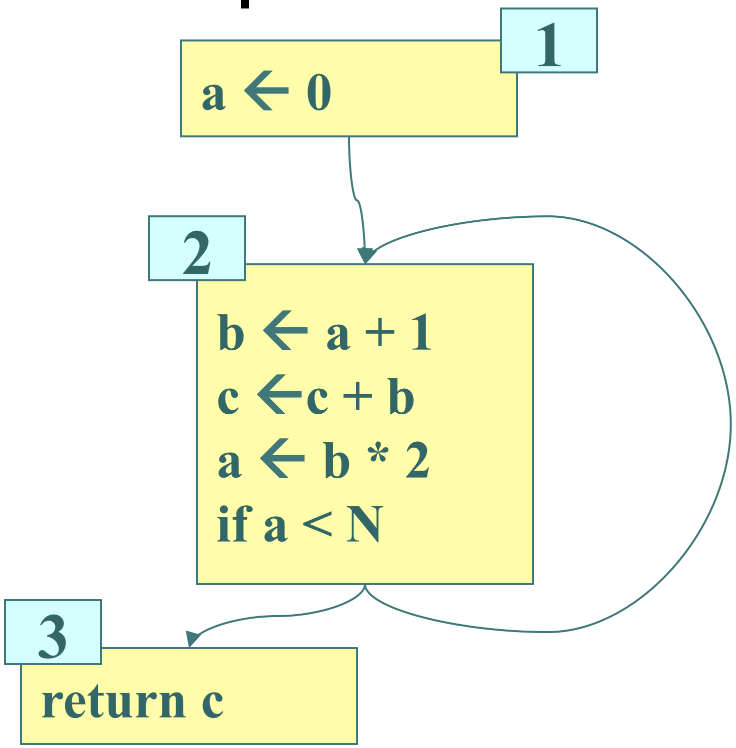
- For optimization purposes, assume **Φ** instructions are just normal instructions
  - e.g., $\mathbf{a_3 \leftarrow \Phi\,(a_2 , a_1)}$ is like an ordinary instruction for the purposes of data flow
- Typically they can be removed at register allocation time.
  - e.g., if $\mathbf{a_2 , a_1}$ are placed in the same register, then the above node may be removed
- If not, extra moves need to be inserted in previous basic blocks to implement **Φ** node.

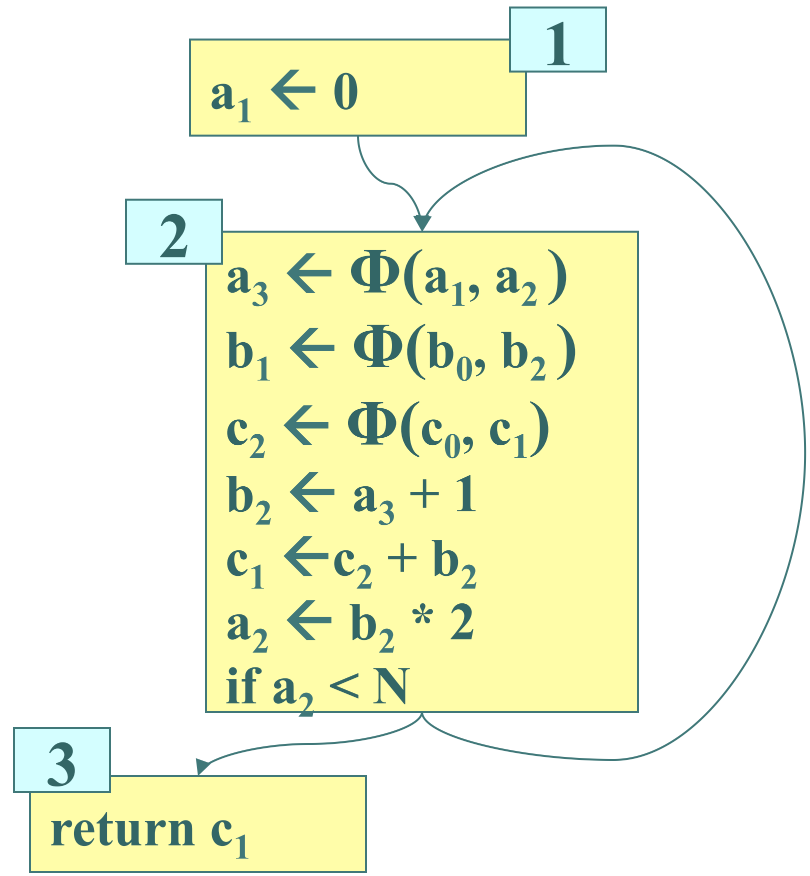
Loops and Φ nodes



This is translated just like the straight line example…

Loops and Φ nodes





This is translated just like the straight line example…

*

Is this really Static Single Assignment?

Each time round a loop, the same virtual register is re-assigned.

- But it is defined **statically** only once.
- Multiple **dynamic** definitions are allowed.

Using SSA: dead code elimination

Consider

v1 ← x3 + y2

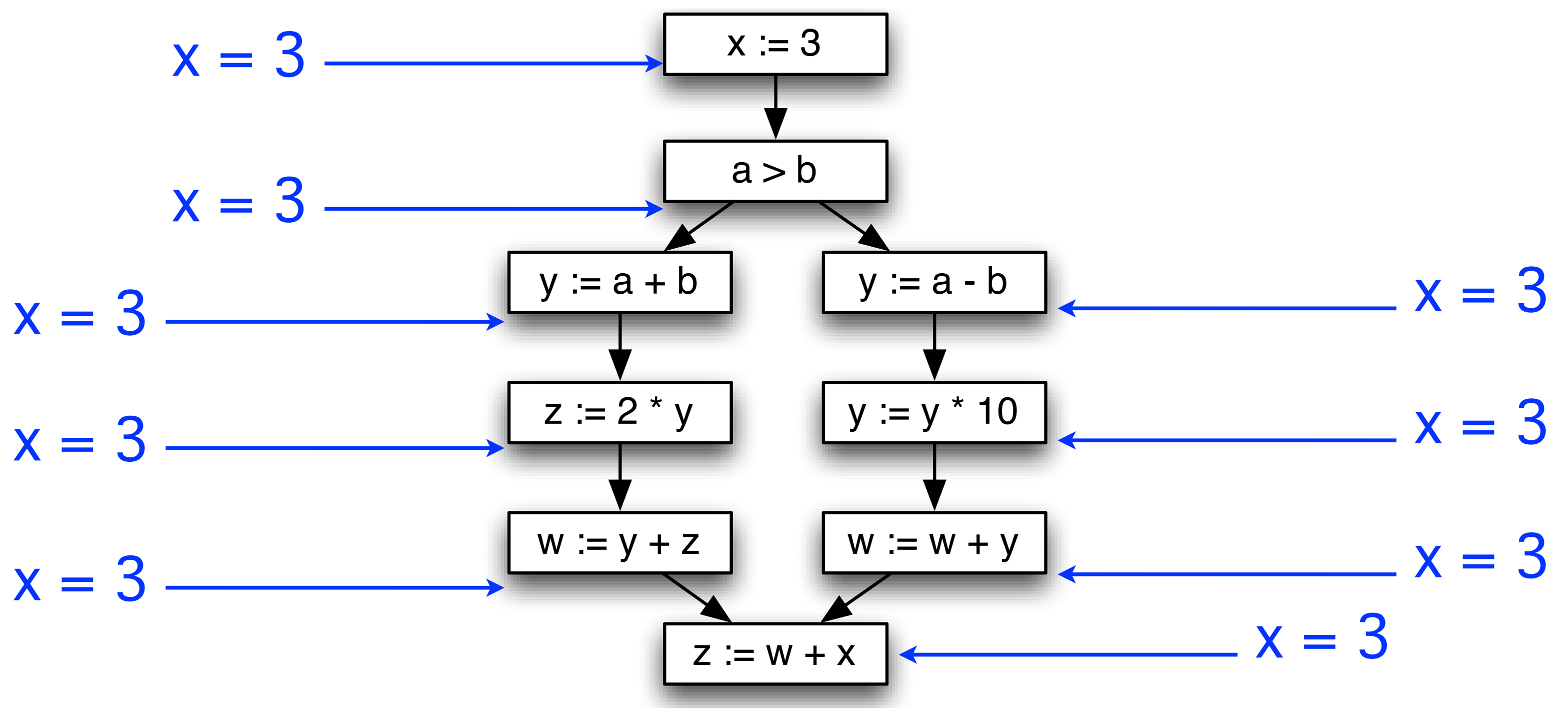
If v1 has no uses, then

- easy to tell by inspecting SSA code
- This must be the only definition of v1.
- So v1 is dead.
- This assignment to v1 can be removed!
- The operation must not have side-effects.
  - i.e., it can't be print, writing memory, etc.
- Removing this assignment can cause other instruction to also become dead.
  - i.e., deletes uses of x,y

Happens in practice inside optimizing compilers.

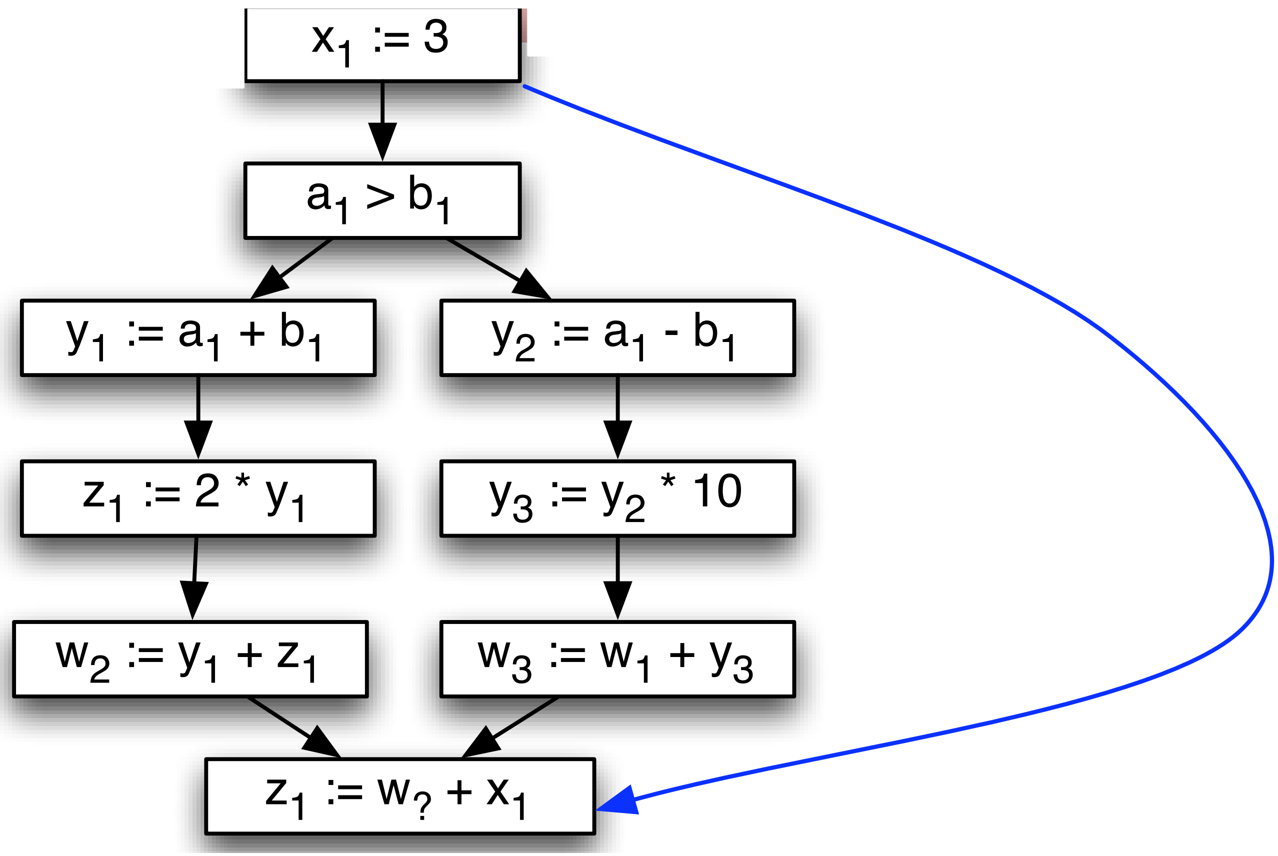
Using SSA: Constant propagation

Before SSA



Using SSA: Constant propagation

After SSA Why is this a big win over the previous slide?



*