SWEN430 - Compiler Engineering

Lecture 18 - Machine Code IV

Lindsay Groves and David J. Pearce

School of Engineering and Computer Science
Victoria University of Wellington

Compiler Optimisation

"In computing, an **optimizing compiler** *is a compiler that tries to minimize or maximize some attributes of an executable computer program. "* *–Wikipedia*

```
pushq %rbp
movq %rsp, %rbp
movl %edi, -4(%rbp)
movl %esi, -8(%rbp)
movl -4(%rbp), %edx
movl -8(%rbp), %eax
addl %edx, %eax
popq %rbp
ret
```

(before)

```
leal (%rdi,%rsi), %eax
ret
```

(after)

Peephole Optimisation

> *"**Peephole optimization** is an optimization technique performed on a small set of compiler-generated instructions; the small set is known as the* **peephole** *or window"* –*Wikipedia*

```
...
pushq %rax
popq  %rax
...
```

```
...
movq %eax, %eax
...
```

```
   ...
   jmp lab
lab:
   ...
```

- Identify **patterns** in peephole, ignoring other instructions
- Typically only for relatively **simple** optimisations

Constant Folding & Propagation

"Constant folding is the process of recognizing and evaluating constant expressions at **compile time** *rather than computing them at runtime"*
–Wikipedia

```
...
movl $1, %eax
movl $2, %ebx
addl %ebx,%eax
...
```

(before)

```
...
movl $3, %eax
...
```

(after)

"Constant **propagation** *is the process of substituting the values of known constants in expressions at compile time"*
–Wikipedia

Common Subexpression Elimination

*"**common subexpression elimination** (CSE) is a compiler optimization that searches for instances of identical expressions (i.e., they all evaluate to the same value)" –Wikipedia*

```
...
movl -4(%rbp), %eax
movl -8(%rbp), %ebx
addl %ebx,%eax
movl -4(%rbp), %ecx
movl -8(%rbp), %edx
addl %edx, %ecx
addl %ecx, %eax
...
```

(before)

```
...
movl -4(%rbp), %eax
movl -8(%rbp), %ebx
addl %ebx,%eax
addl %eax,%eax
...
```

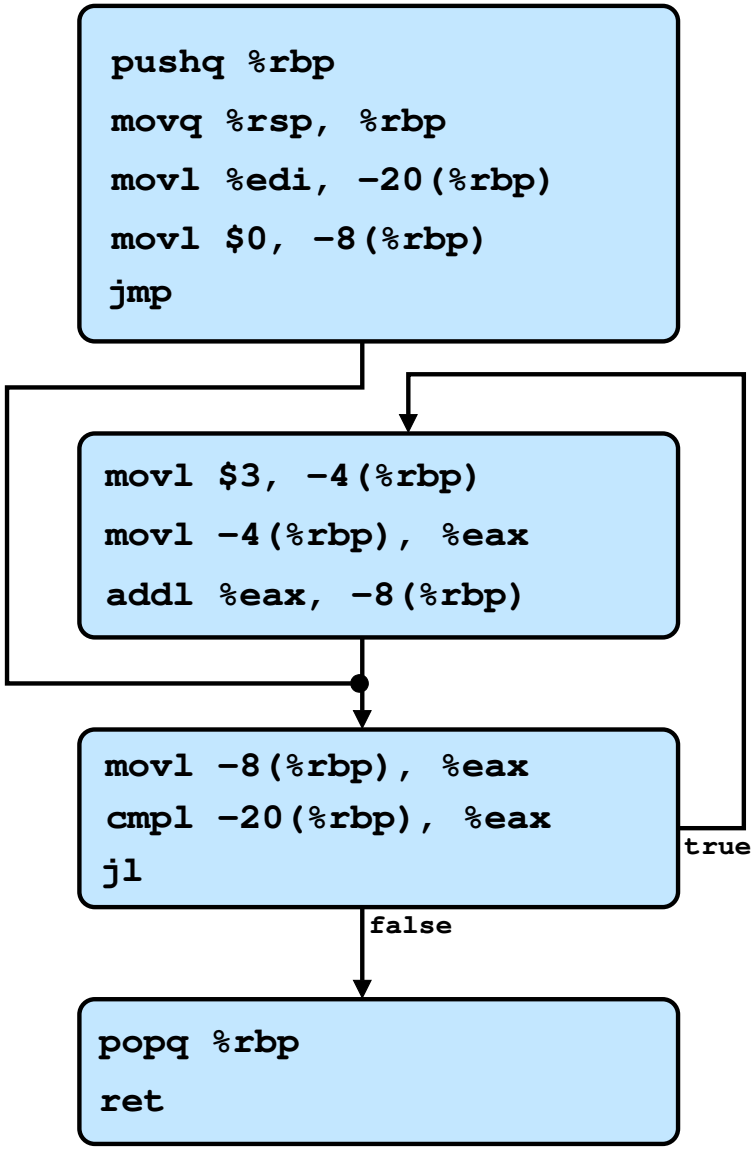
(after)

Loop Optimisation

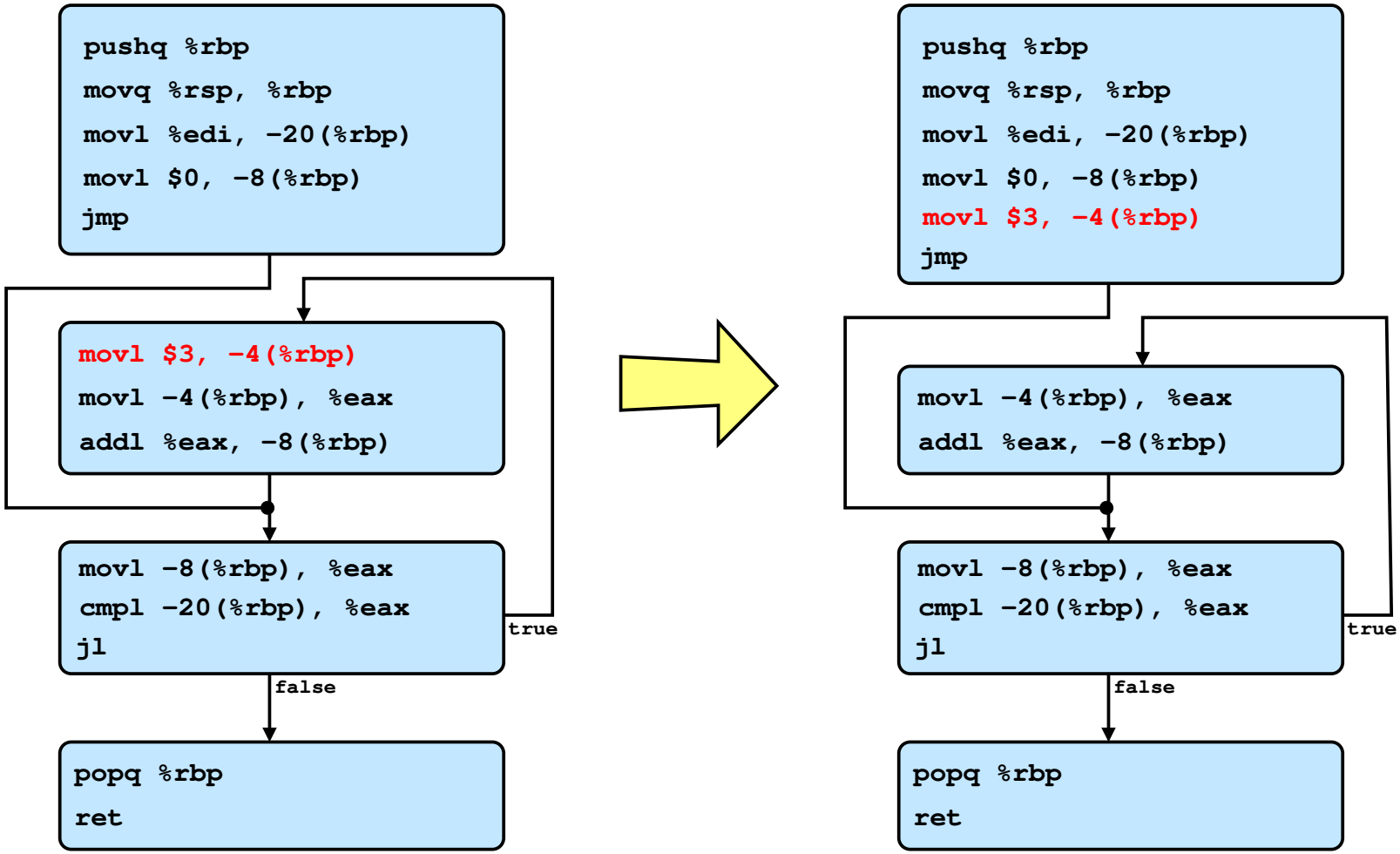
- Loops **execute** many iterations!
- Small changes have **big** effects
- e.g. loop invariant code **motion** ...

  ... moves code **out** of loops
- e.g. loop **unrolling** for loops ...

  ... with **fixed** number of iterations

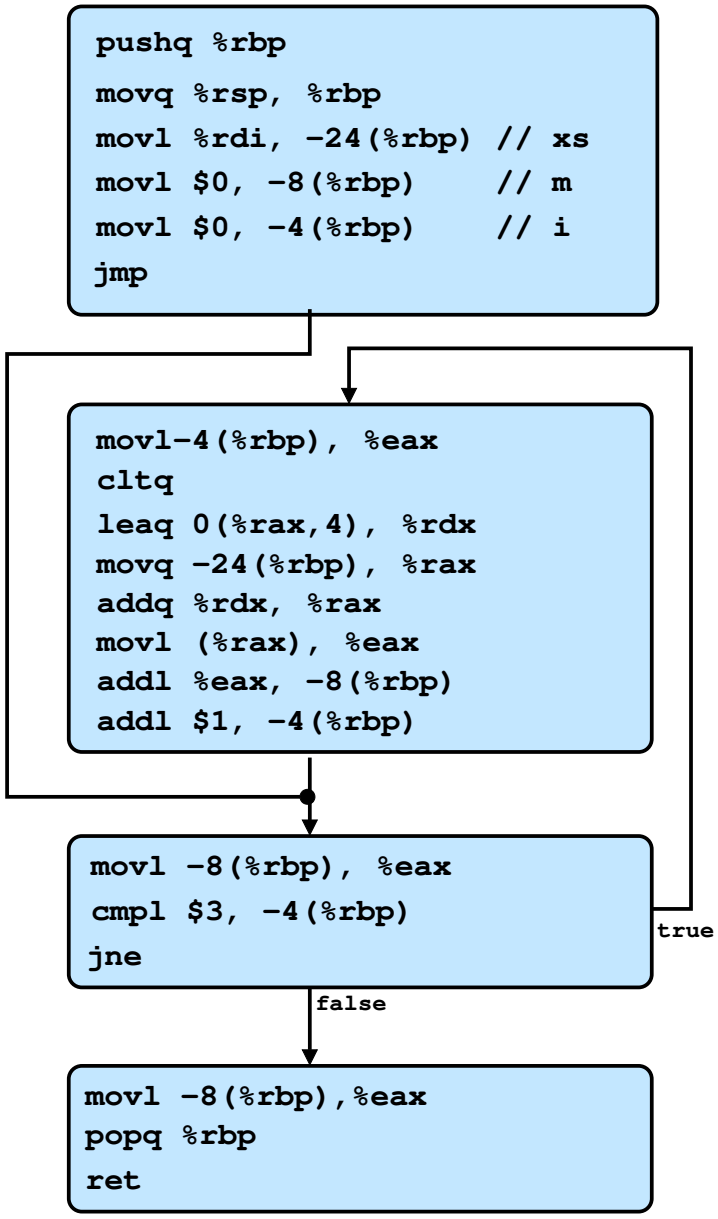


Loop Invariant Code Motion



- Could apply **lots** of peephole optimisations here as well!

Loop Unrolling




```
int f(int xs[]) {
  int m = 0;
  for(int i=0;i!=3;++i) {
    m = m + xs[i];
  }
  return m
}
```

(source code)

```
movl 4(%rdi), %eax
addl (%rdi), %eax
addl 8(%rdi), %eax
ret
```

Static Single Assignment (SSA)

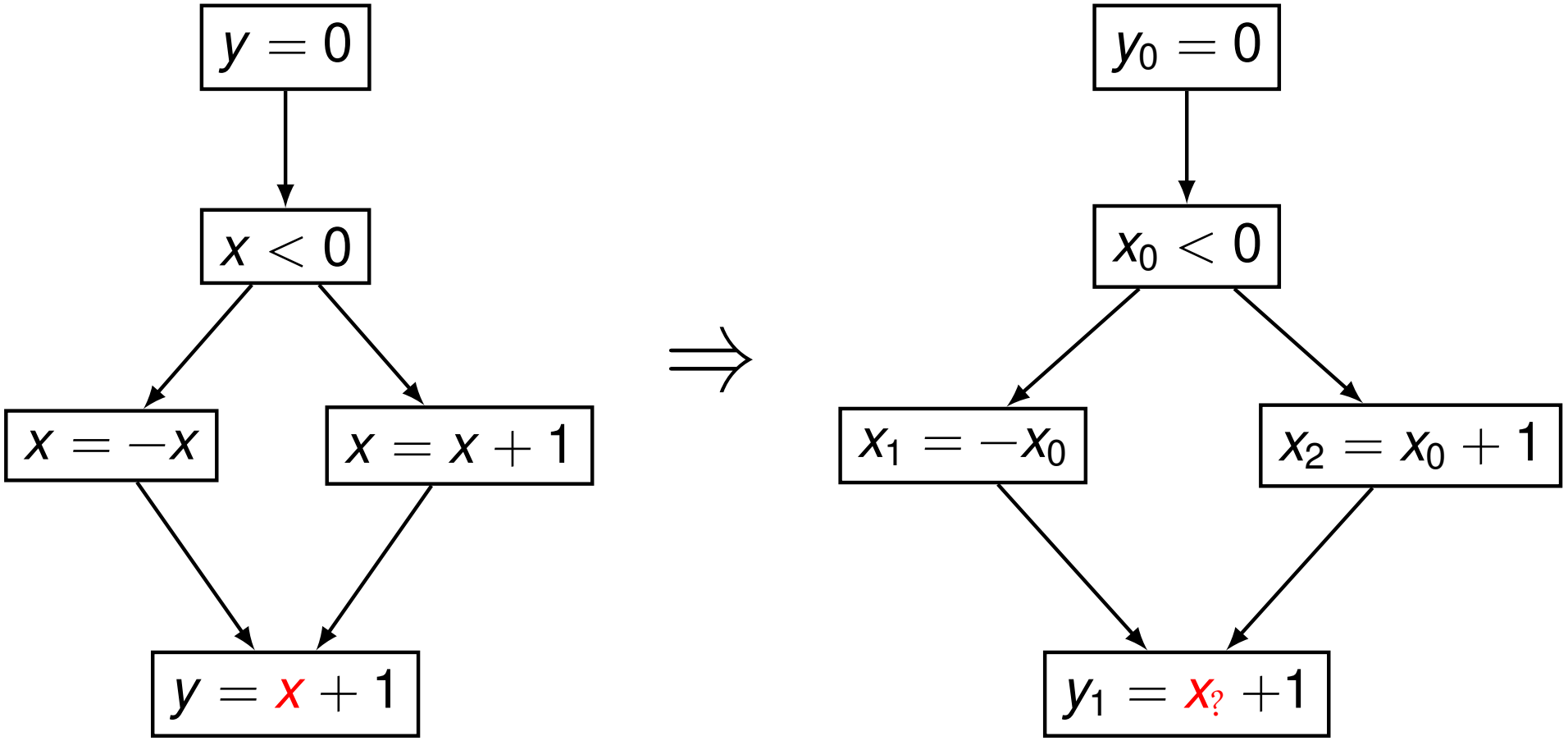
- Common intermediate representation
  E.g. *shimple* in soot and *tree-ssa* in GCC.

- **Idea**: adjust program so each variable has at most one definition
  A variable X becomes X_n, where n increases with each definition:

$$\begin{aligned} x &= 0 \\ y &= x \\ x &= x + 1 \end{aligned} \quad \Longrightarrow \quad \begin{aligned} x_0 &= 0 \\ y_0 &= x_0 \\ x_1 &= x_0 + 1 \end{aligned}$$

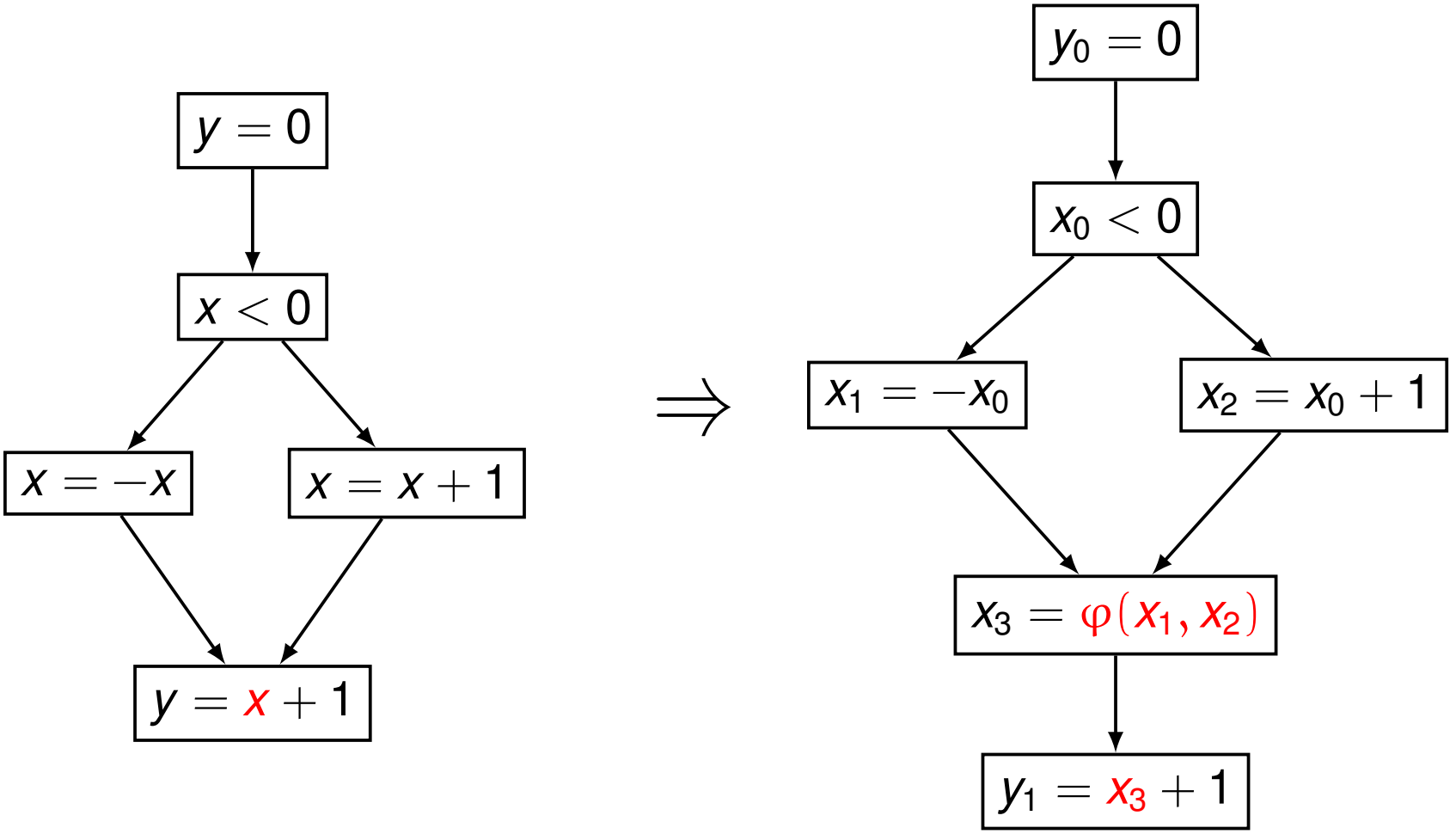
- This helps program analyses in many ways
  - Simplifies many analyses (e.g. constant propagation)
  - Reduces storage requirements

Another Example



But, what subscript do we give x?

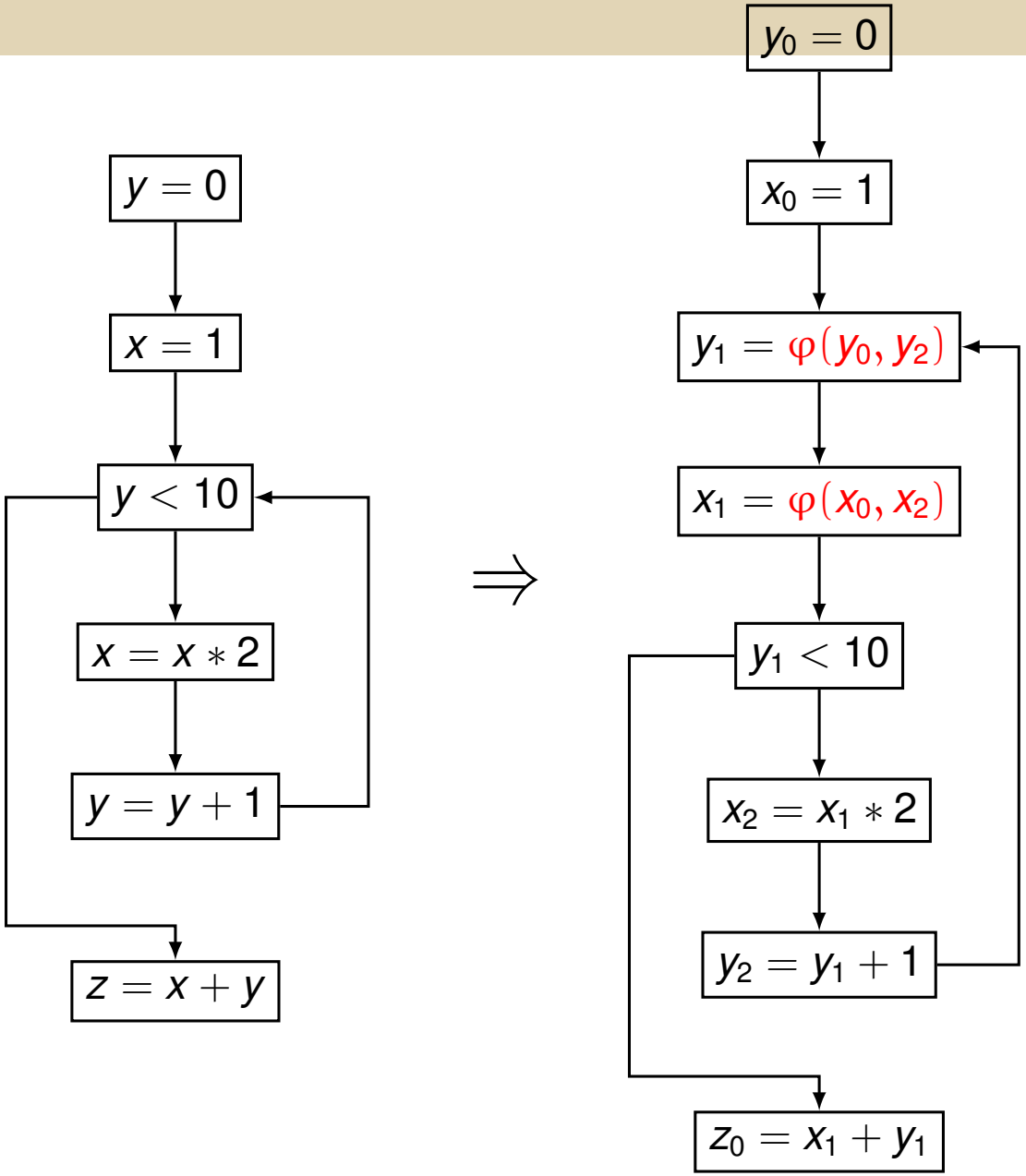
Another Example (continued)



But, what subscript do we give x?

SSA introduces special φ-nodes which merge values together

Loop Example



SSA Form and Dependence

Another interesting observation: SSA form encodes *dependence information*.

Definition (Dependence)

If statement S uses variable x, then S *depends* upon all statements which define x and where the value defined reaches S.

E.g.

$$
\begin{aligned}
S1: &\quad x = 0 \\
S2: &\quad y = x + 1 \\
S3: &\quad x = 3 \\
S4: &\quad y = 123 \\
S5: &\quad z = x + y
\end{aligned}
$$

Here $S2$ depends upon $S1$ and $S5$ depends upon $S4$ and $S3$

- But, $S5$ does **not** depend on $S1$
- Since that defintion of x does not *reach* $S5$

SSA Form and Dependence

Another interesting observation: SSA version of previous example:

$$
\begin{aligned}
x_0 &= 0 \\
y_0 &= x_0 + 1 \\
x_1 &= 3 \\
y_1 &= 123 \\
z_0 &= x_1 + y_1
\end{aligned}
$$

Here, we can see the dependencies immediately

- y_0 depends upon x_0
- z_0 depends upon x_1 and y_1

Problems with SSA – Arrays

Consider translating the following program to SSA:

```
int foobar(int x){
  int A[10];
  A[0] = 1;
  A[0] = 2;
  A[1] = 3;
  A[x] = 4;
  return A[0];
}
```

$\Longrightarrow$

```
int foobar(int x){
  int A[10]
  A[0]_0 = 1
  A[0]_1 = 2
  A[1]_0 = 3
  A[x]_? = 4
  return A[0]_?
}
```

What subscript should we use for definition of `A[x]`?

- It depends on the value of *x*!!
- To know this requires evaluating foobar(...)
  - This is likely to be expensive and maybe impossible if its parameters are parameters to the program itself

Problems with SSA — Pointers

Pointers are another problem for SSA

int foobar(**int** x){
    int a,b,*p
    if (f(...) $>$ 0) {p_0 = &a}
    else { p_1 = &b }
    $p_2 = \varphi(p_0, p_1)$
    $a_0 = 0$
    $b_0 = 3$
    $^*p_2 = 1$
    return $a_? + b_?$
}

Again, variable actually defined depends upon evaluation of f(...).
Why is this not a problem in java?

Additional Readings on SSA

- **Efficiently computing static single assignment form and the control dependence graph.** R. Cytron, J. Ferrante, B.K. Rosen and M. N. Wegman. In Transactions on Programming Languages and Systems (TOPLAS), 1991.

- **Extended SSA Numbering: introducing SSA properties to languages with multi-level pointers**. C. Lapkowski and Laurie J. Hendren. In Proceedings of the Conference of the Centre for Advanced Studies on Collaborative Research, 1996.

- **Array SSA form and its use in parallelization**, Kathleen Knobe and Vivek Sarkar. In Proceedings of the Symposium on the Principles of Programming Languages (POPL), 1998.