Lecture 7
Redundancy elimination

Motivation

Some expressions in a program may cause redundant recomputation of values.

If such recomputation is safely eliminated, the program will usually become faster.

There exist several *redundancy elimination* optimisations which attempt to perform this task in different ways (and for different specific meanings of "redundancy").

Common subexpressions

Common-subexpression elimination is a transformation which is enabled by available-expression analysis (AVAIL), in the same way as LVA enables dead-code elimination.

Since AVAIL discovers which expressions will have been computed by the time control arrives at an instruction in the program, we can use this information to spot and remove redundant computations.

Common subexpressions

Recall that an expression is *available* at an instruction if its value has definitely already been computed and not been subsequently invalidated by assignments to any of the variables occurring in the expression.

If the expression e is available on entry to an instruction which computes e, the instruction is performing a redundant computation and can be modified or removed.

Common subexpressions

We consider this redundantly-computed expression to be a *common subexpression*: it is common to more than one instruction in the program, and in each of its occurrences it may appear as a subcomponent of some larger expression.

```
x = (a*b)+c;
⋮
print a * b;    a*b AVAILABLE
```

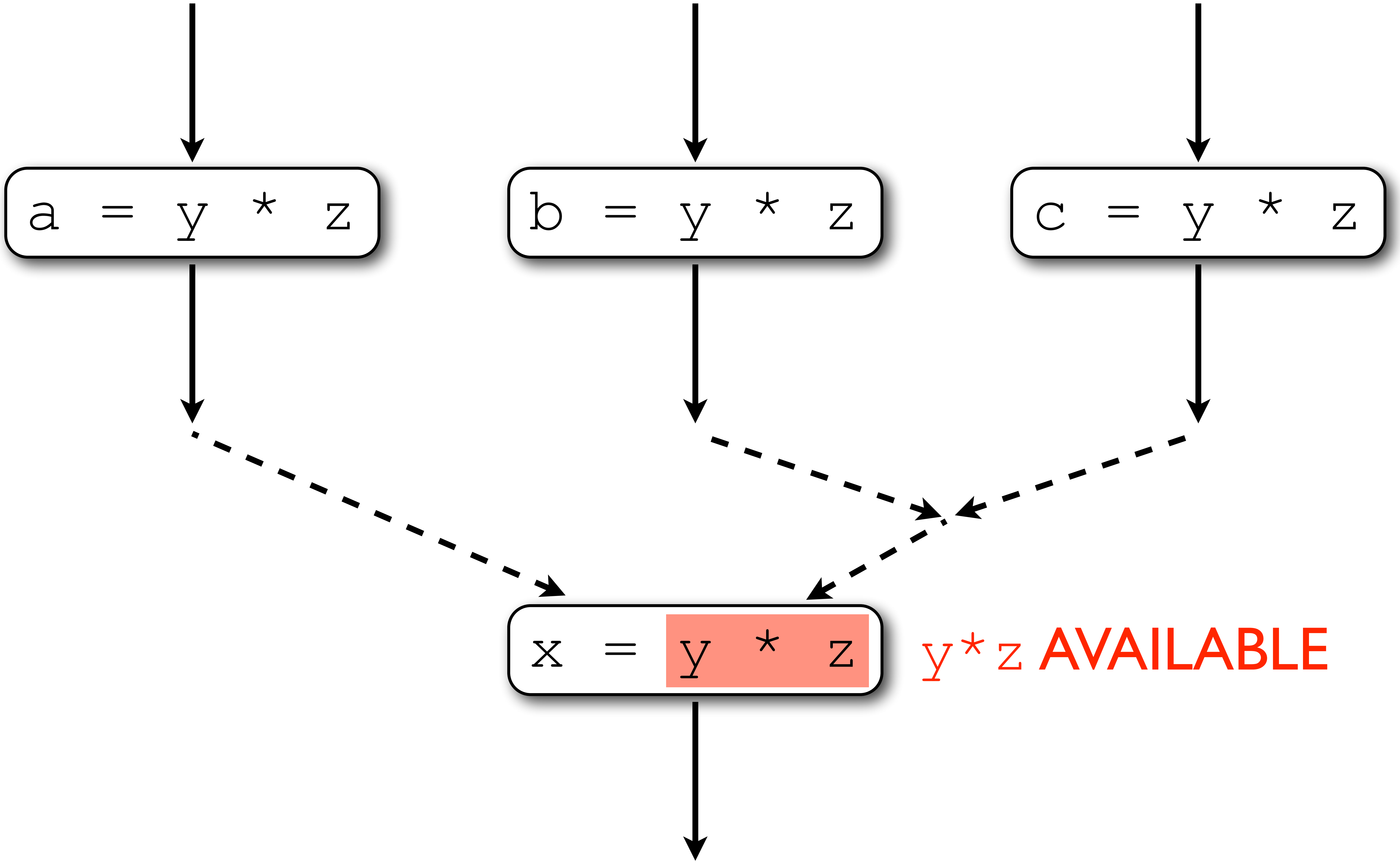
Common subexpressions

We can eliminate a common subexpression by storing its value into a new temporary variable when it is first computed, and reusing that variable later when the same value is required.

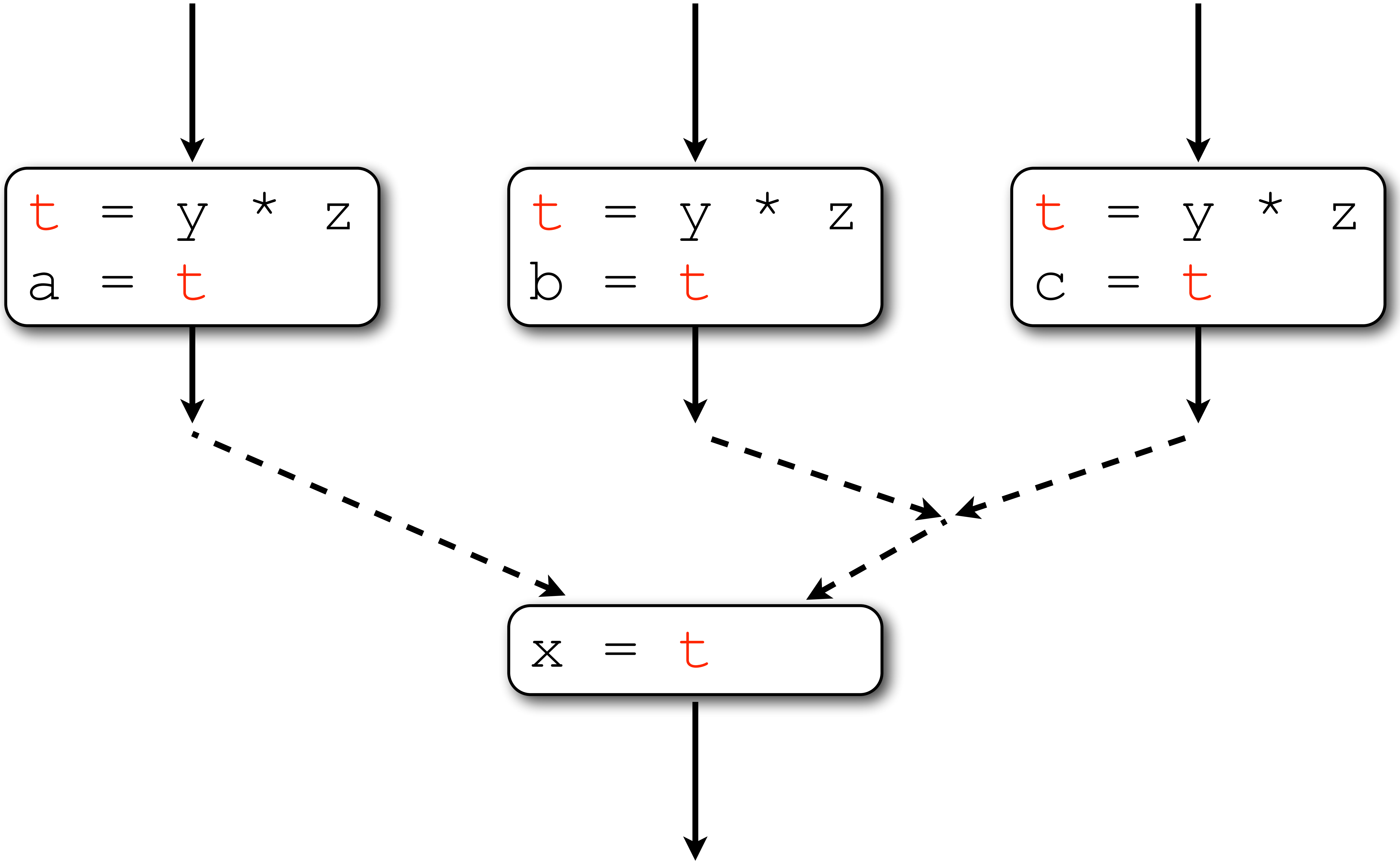
Algorithm

- Find a node *n* which computes an already-available expression *e*
  - Replace the occurrence of e with a new temporary variable *t*
  - On each control path backwards from *n*, find the first instruction calculating e and add a new instruction to store its value into *t*
- Repeat until no more redundancy is found

Algorithm



Algorithm



Common subexpressions

Our transformed program performs (statically) fewer arithmetic operations: `y*z` is now computed in three places rather than four.

However, three register copy instructions have also been generated; the program is now larger, and whether it is faster depends upon characteristics of the target architecture.

Common subexpressions

The program might have "got worse" as a result of performing common-subexpression elimination.

In particular, introducing a new variable increases register pressure, and might cause spilling.

Memory loads and stores are much more expensive than multiplication of registers!

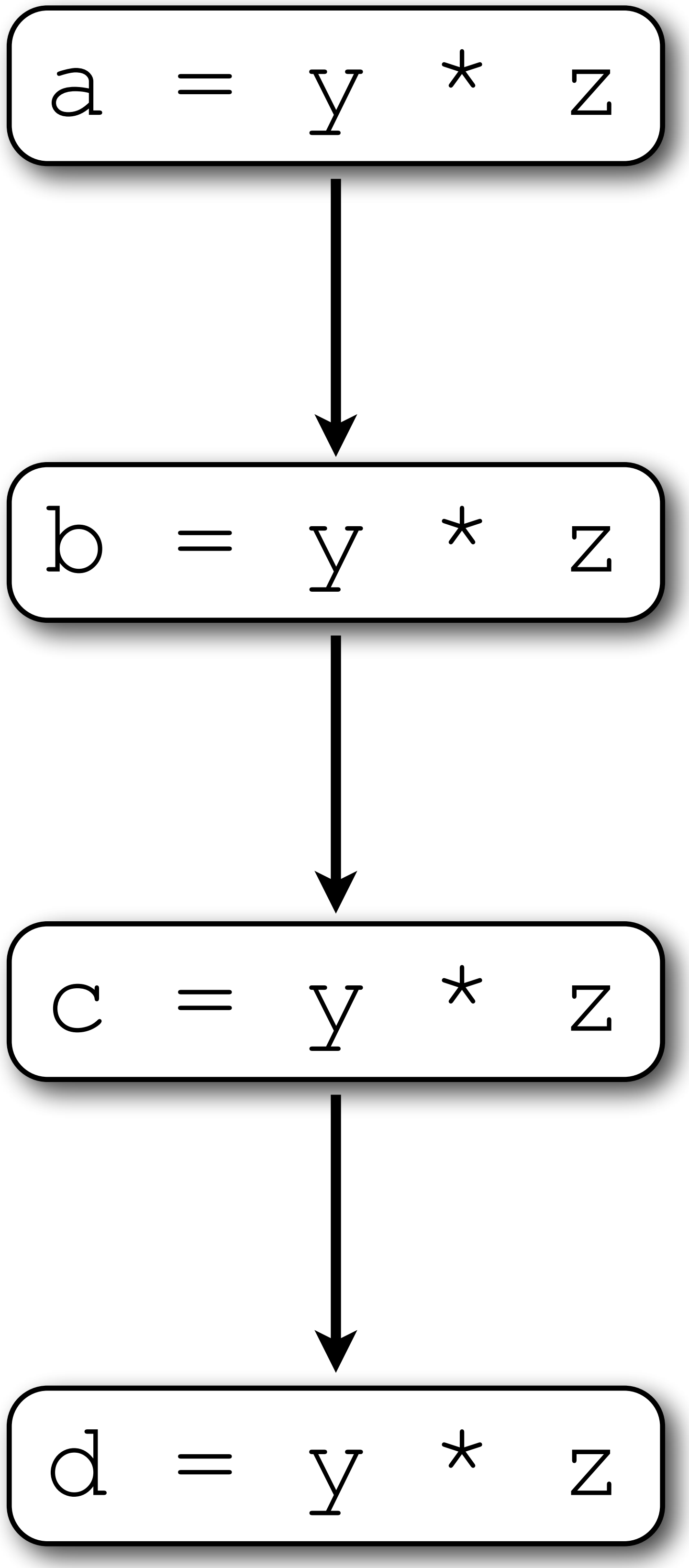
Copy propagation

This simple formulation of CSE is fairly careless, and assumes that other compiler phases are going to tidy up afterwards.

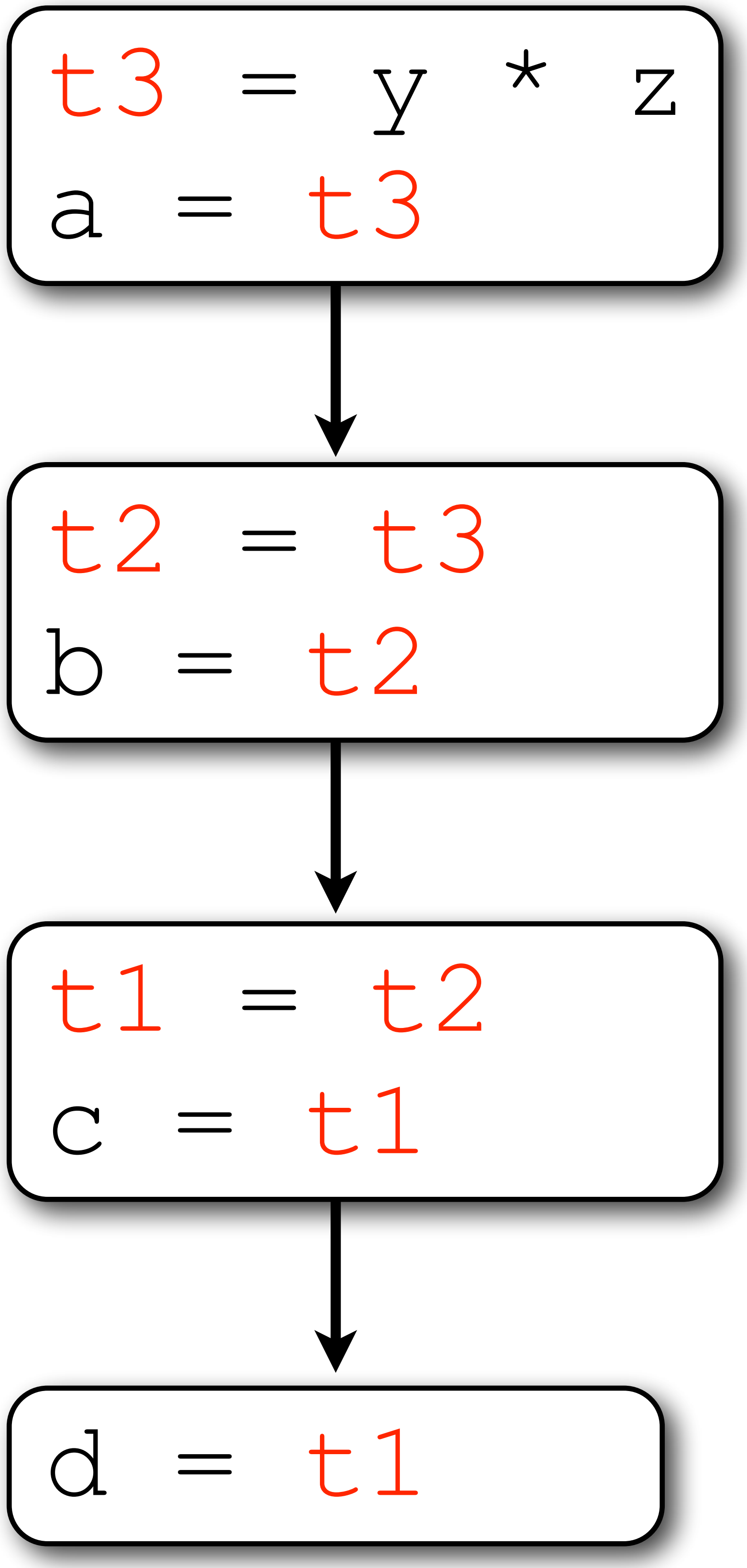
In addition to register allocation, a transformation called *copy propagation* is often helpful here.

In copy propagation, we scan forwards from an `x=y` instruction and replace `x` with `y` wherever it appears (as long as neither `x` nor `y` have been modified).

Copy propagation



Copy propagation



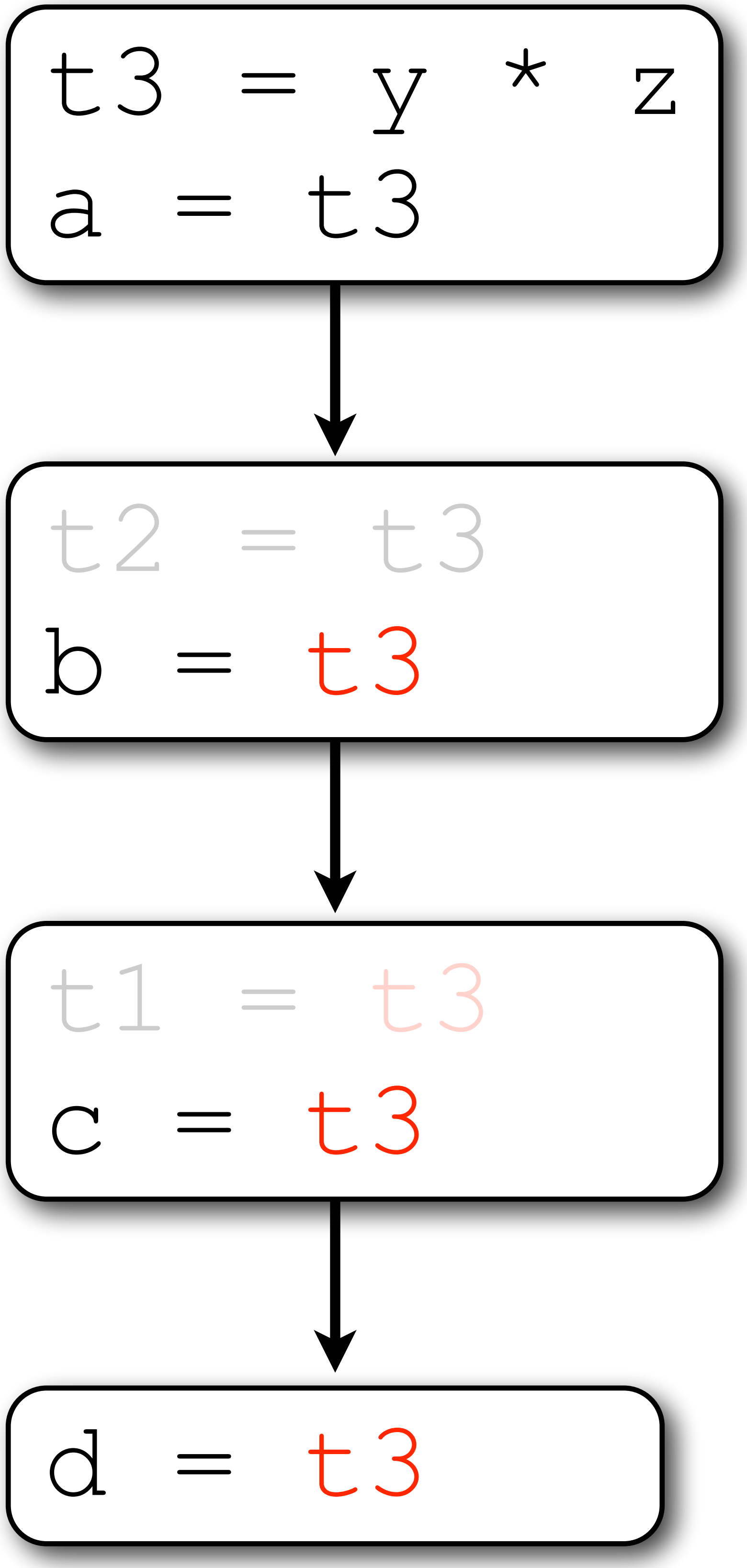
```
t3 = y * z
a = t3
```

```
t2 = t3
b = t2
```

```
t1 = t2
c = t1
```

```
d = t1
```

Copy propagation



```
t3 = y * z
a = t3
```

```
t2 = t3
b = t3
```

```
t1 = t3
c = t3
```

```
d = t3
```

Code motion

Transformations such as CSE are known collectively as *code motion* transformations: they operate by moving instructions and computations around programs to take advantage of opportunities identified by control- and data-flow analysis.

Code motion is particularly useful in eliminating different kinds of redundancy.

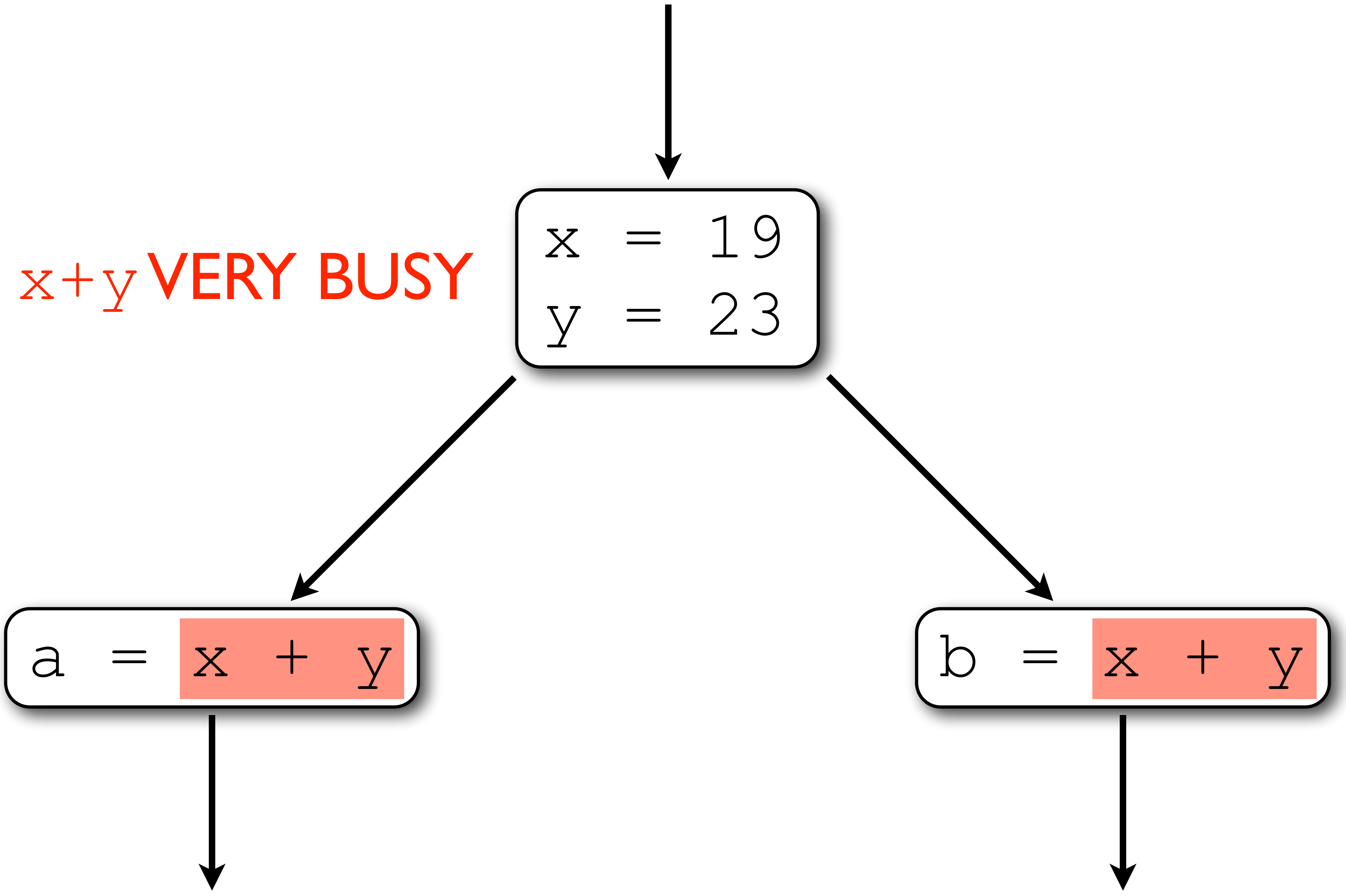
It’s worth looking at other kinds of code motion.

Code hoisting

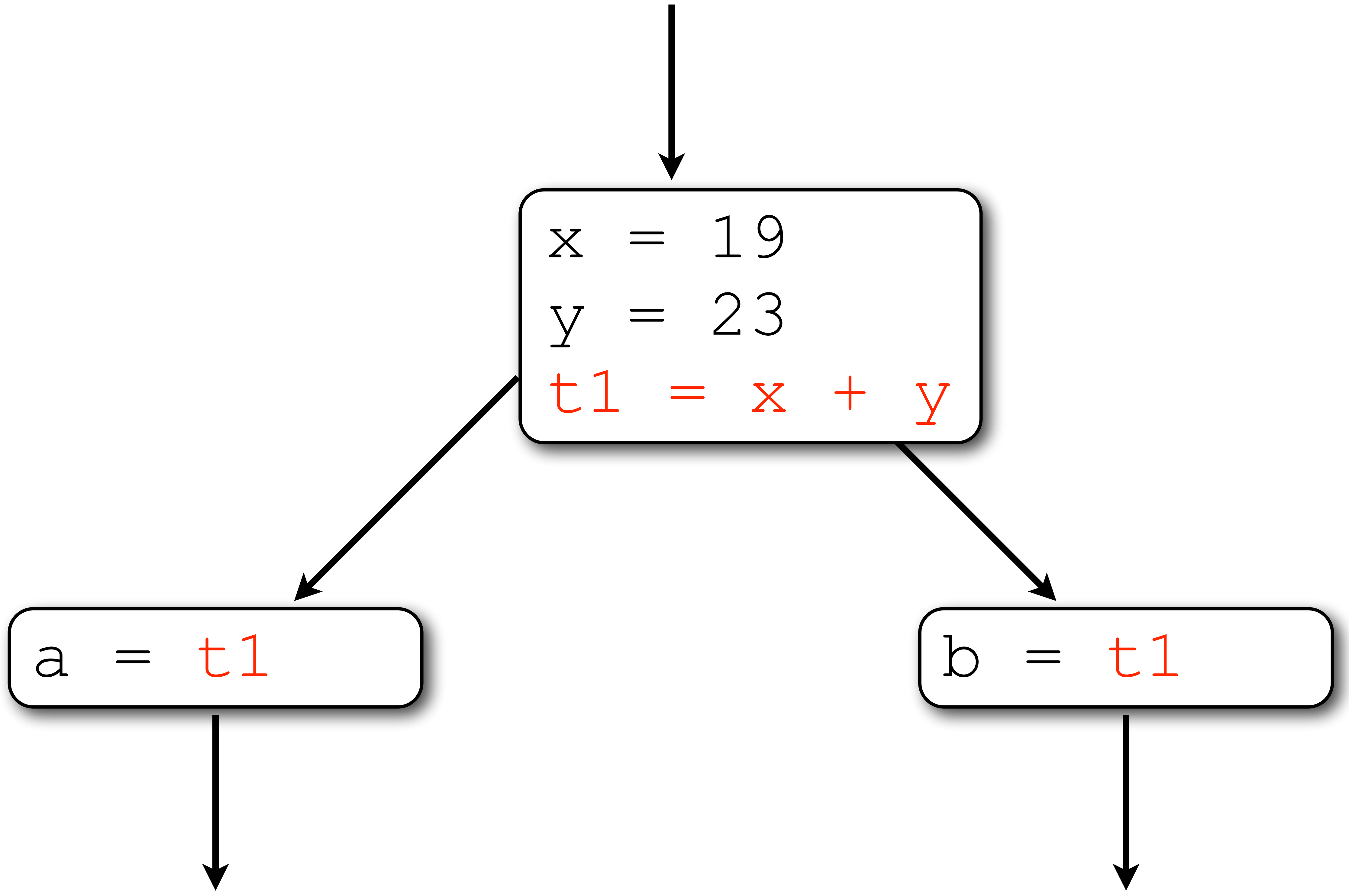
Code hoisting reduces the size of a program by moving duplicated expression computations to the same place, where they can be combined into a single instruction.

Hoisting relies on a data-flow analysis called *very busy expressions* (a backwards version of AVAIL) which finds expressions that are definitely going to be evaluated later in the program; these can be moved earlier and possibly combined with each other.

Code hoisting



Code hoisting



Code hoisting

Hoisting may have a different effect on execution time depending on the exact nature of the code. The resulting program may be slower, faster, or just the same speed as before.

Loop-invariant code motion

Some expressions inside loops are redundant in the sense that they get recomputed on every iteration even though their value never changes within the loop.

Loop-invariant code motion recognises these redundant computations and moves such expressions outside of loop bodies so that they are only evaluated once.

Loop-invariant code motion

```
a = ...;
b = ...;
while (...) {
  x = a + b;
  ...
}
print x;
```

Loop-invariant code motion

```
a = ...;
b = ...;
x = a + b;
while (...) {
    ...
}
print x;
```

Note: the loop must iterate at least once.

Loop-invariant code motion

This transformation depends upon a data-flow analysis to discover which assignments may affect the value of a variable ("reaching definitions").

If none of the variables in the expression are redefined inside the loop body (or are only redefined by computations involving other invariant values), the expression is invariant between loop iterations and may safely be relocated before the beginning of the loop.

Partial redundancy

Partial redundancy elimination combines common-subexpression elimination and loop-invariant code motion into one optimisation which improves the performance of code.

An expression is *partially redundant* when it is computed more than once on *some* (vs. all) paths through a flowgraph; this is often the case for code inside loops, for example.

Partial redundancy

```
a = ...;
b = ...;
while (...) {
  ... = a + b;
  a = ...;
  ... = a + b;
}
```

Partial redundancy

```
a = ...;
b = ...;
... = a + b;
while (...) {
  ... = a + b;
  a = ...;
  ... = a + b;
}
```

Partial redundancy

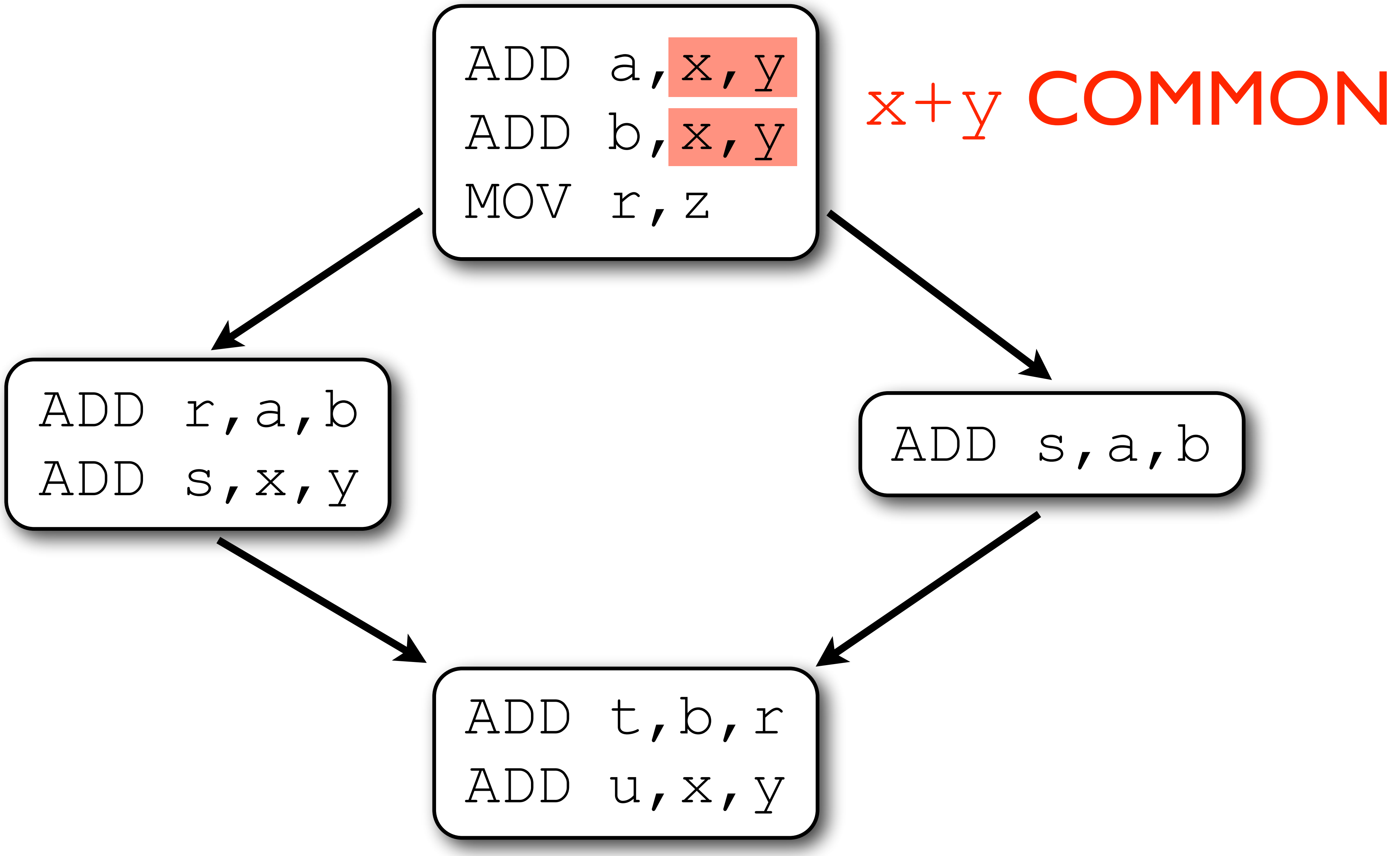
This example gives a faster program of the same size.

Partial redundancy elimination can be achieved in its own right using a complex combination of several forwards and backwards data-flow analyses in order to locate partially redundant computations and discover the best places to add and delete instructions.

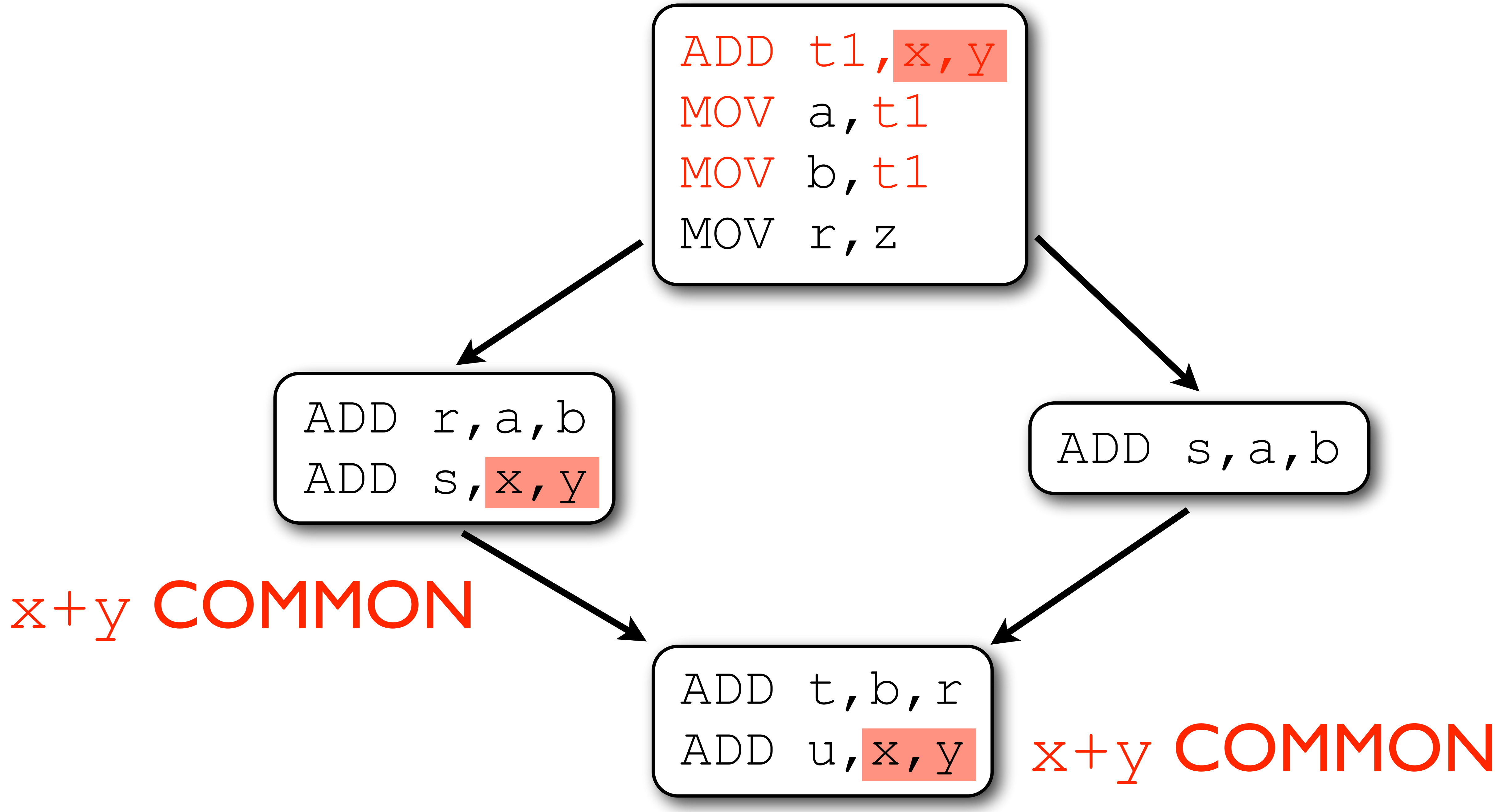
Putting it all together

```
a = x + y;                       ADD a,x,y
b = x + y;           →           ADD b,x,y
r = z;                           MOV r,z
if (a == 42) {
   r = a + b;        →           ADD r,a,b
   s = x + y;                    ADD s,x,y
} else {
   s = a + b;        →           ADD s,a,b
}
t = b + r;           →           ADD t,b,r
u = x + y;                       ADD u,x,y
⋮
return r+s+t+u;
```

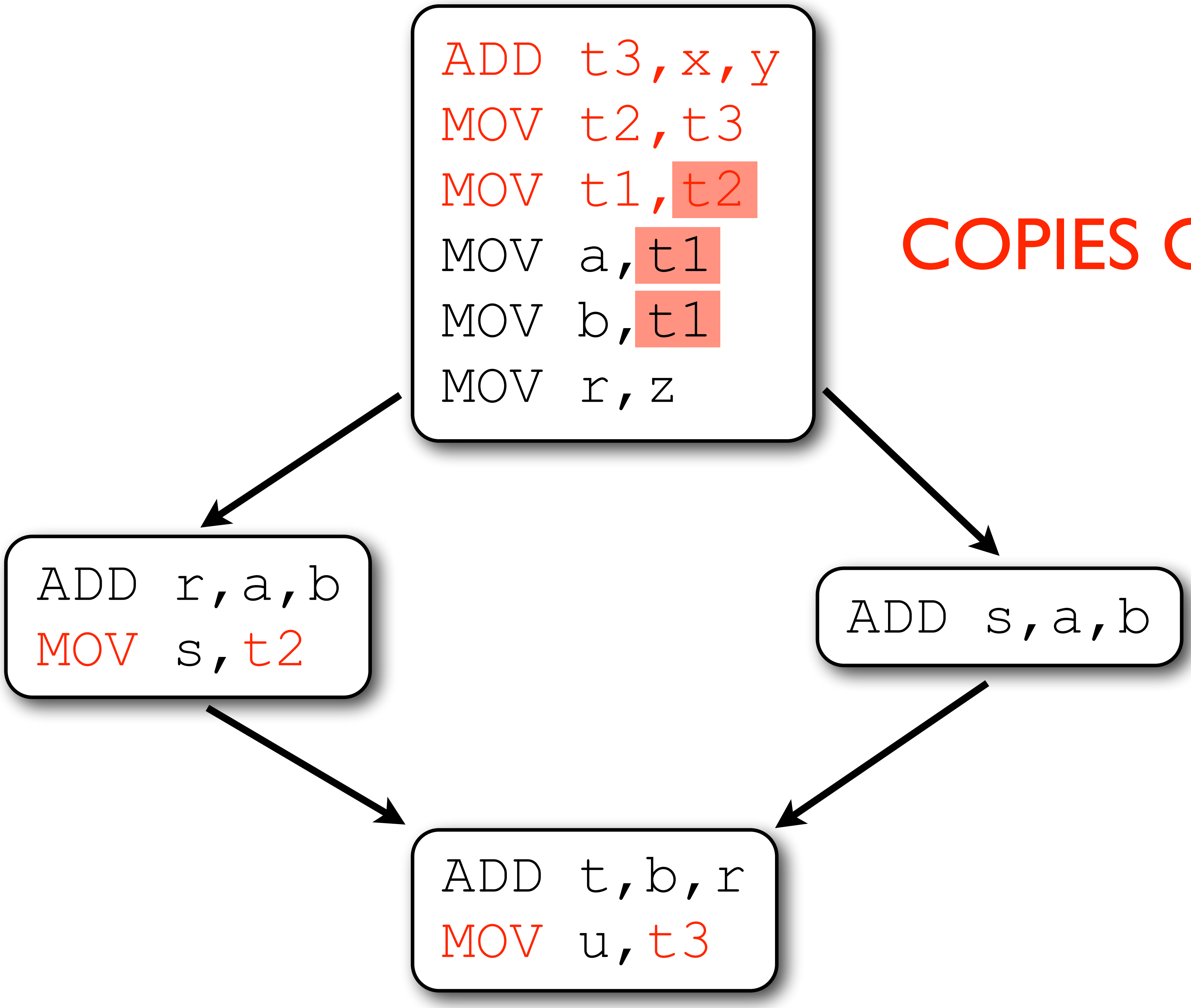
Putting it all together



Putting it all together



Putting it all together



```
ADD t3,x,y
MOV t2,t3
MOV t1,t2
MOV a,t1
MOV b,t1
MOV r,z
```

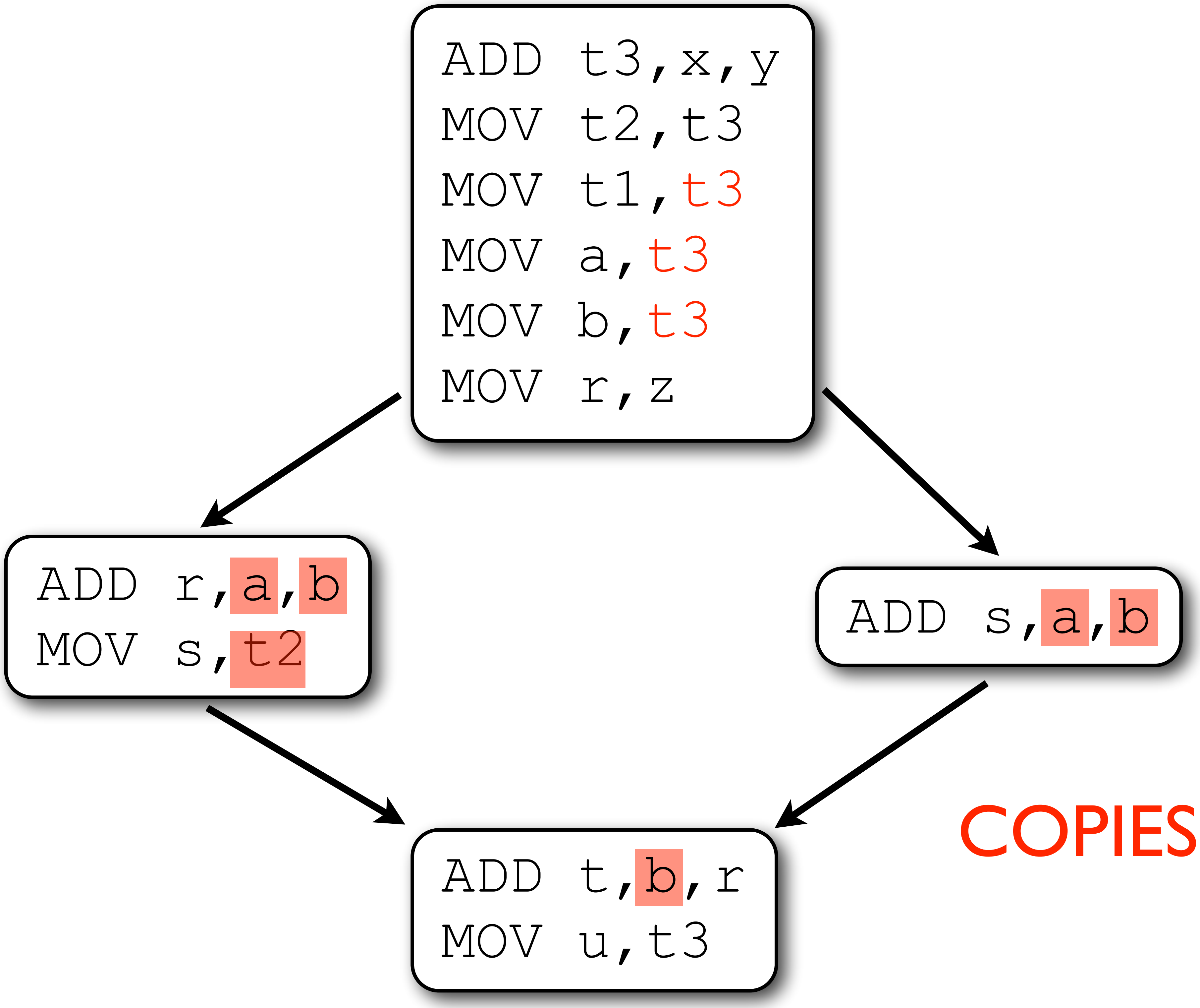
```
ADD r,a,b
MOV s,t2
```

```
ADD s,a,b
```

```
ADD t,b,r
MOV u,t3
```

COPIES OF `t3`

Putting it all together



```
ADD t3,x,y
MOV t2,t3
MOV t1,t3
MOV a,t3
MOV b,t3
MOV r,z
```

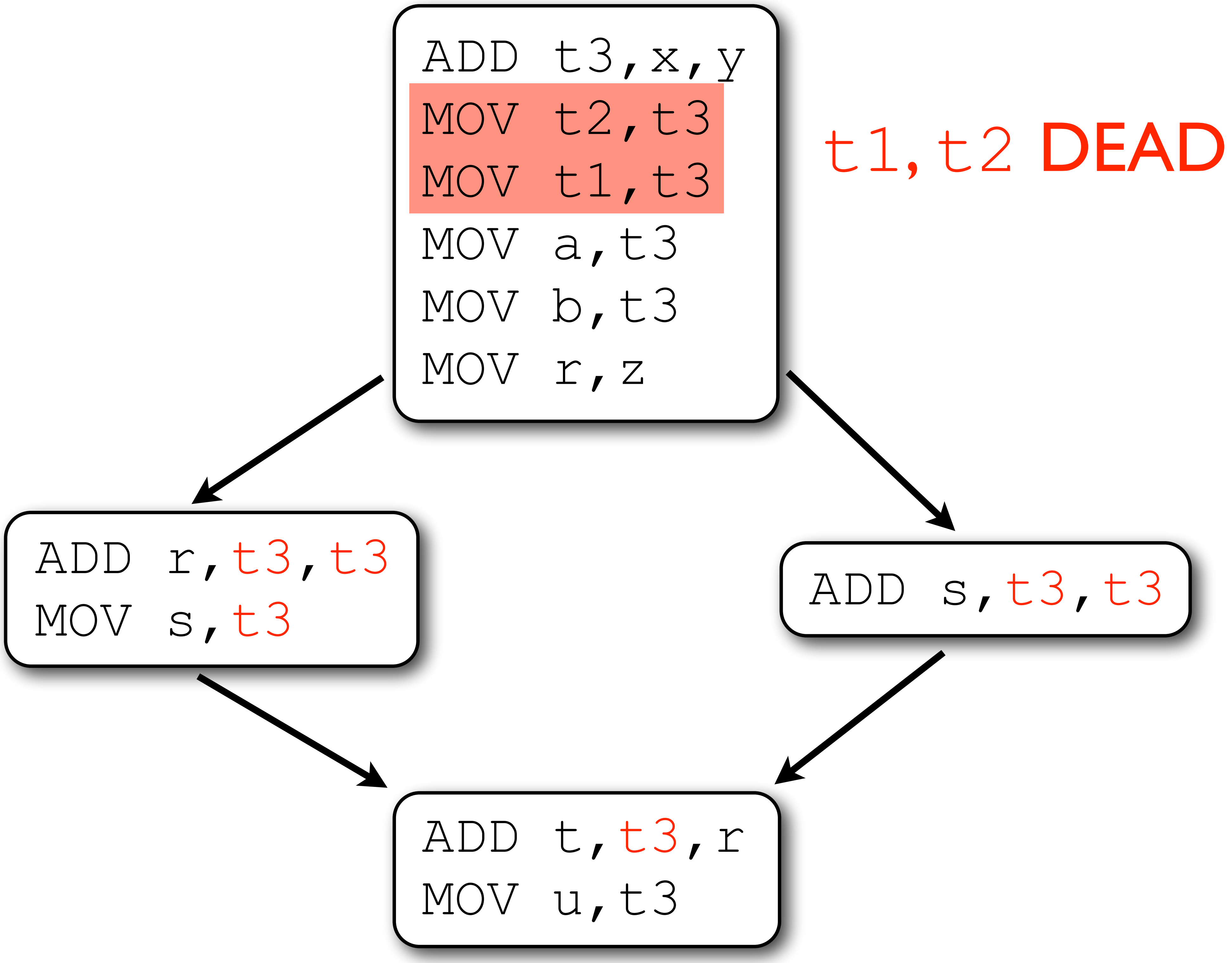
```
ADD r,a,b
MOV s,t2
```

```
ADD s,a,b
```

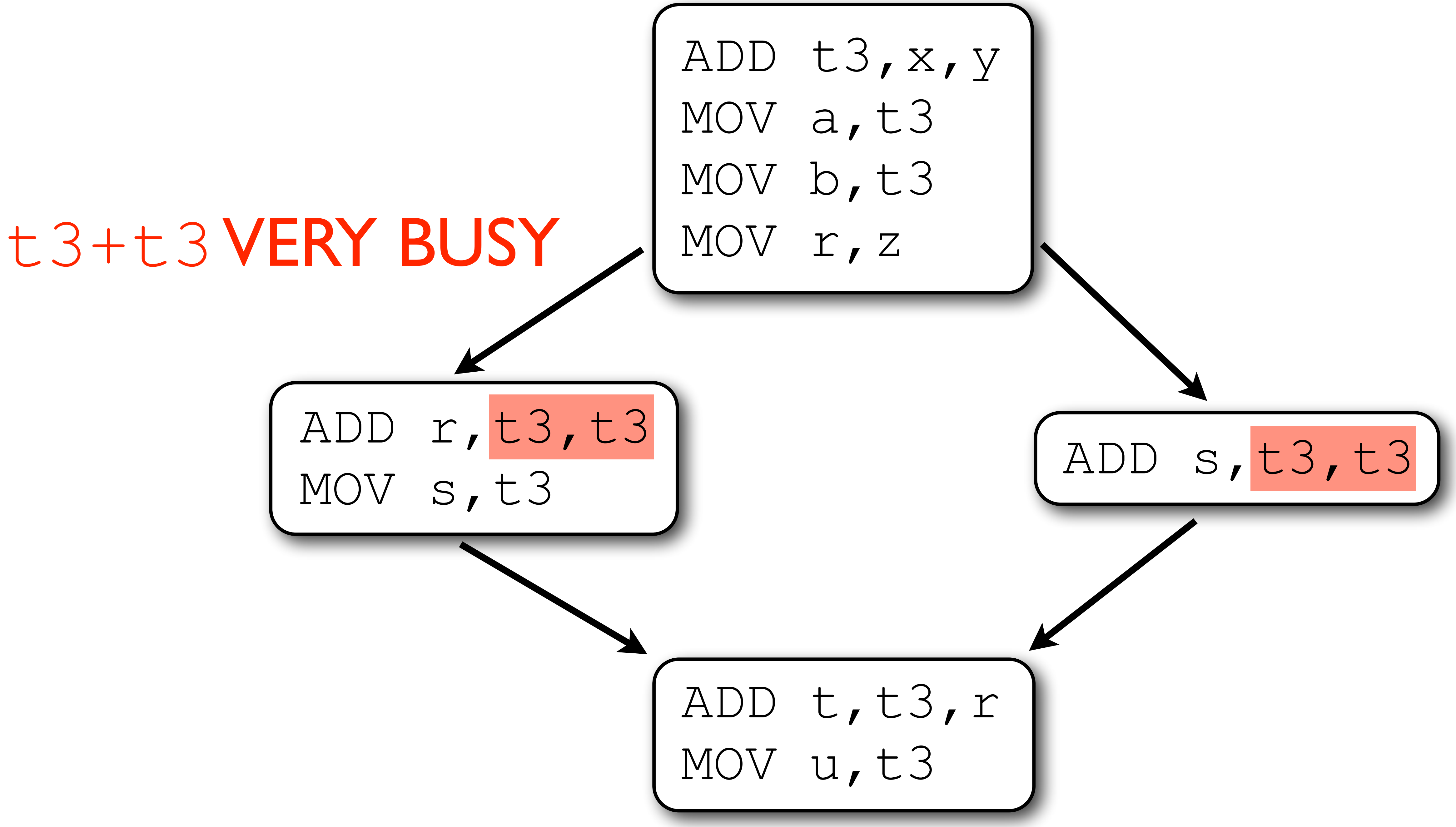
```
ADD t,b,r
MOV u,t3
```

COPIES OF `t3`

Putting it all together



Putting it all together



`t3+t3` VERY BUSY

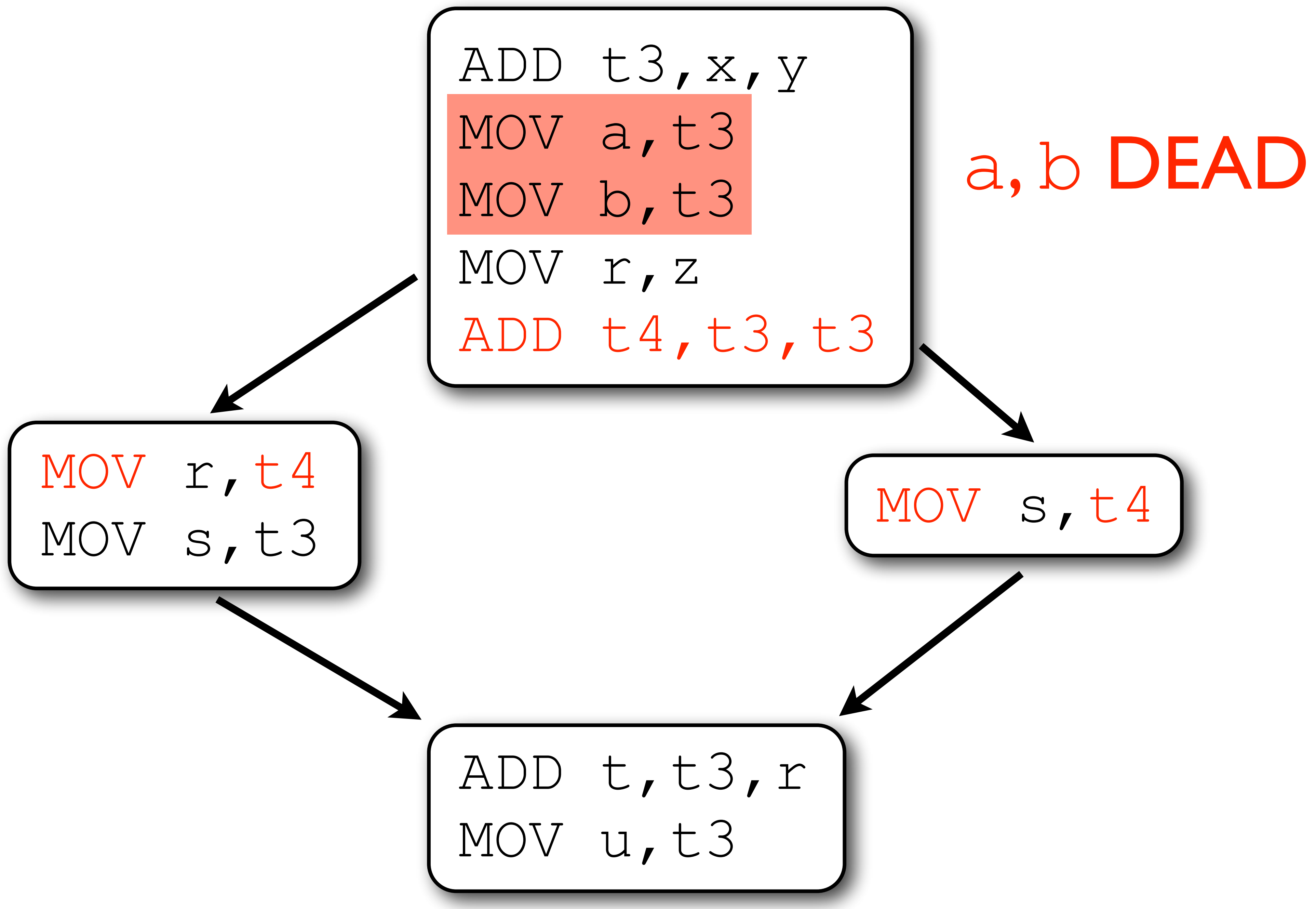
```
ADD t3,x,y
MOV a,t3
MOV b,t3
MOV r,z
```

```
ADD r,t3,t3
MOV s,t3
```

```
ADD s,t3,t3
```

```
ADD t,t3,r
MOV u,t3
```

Putting it all together



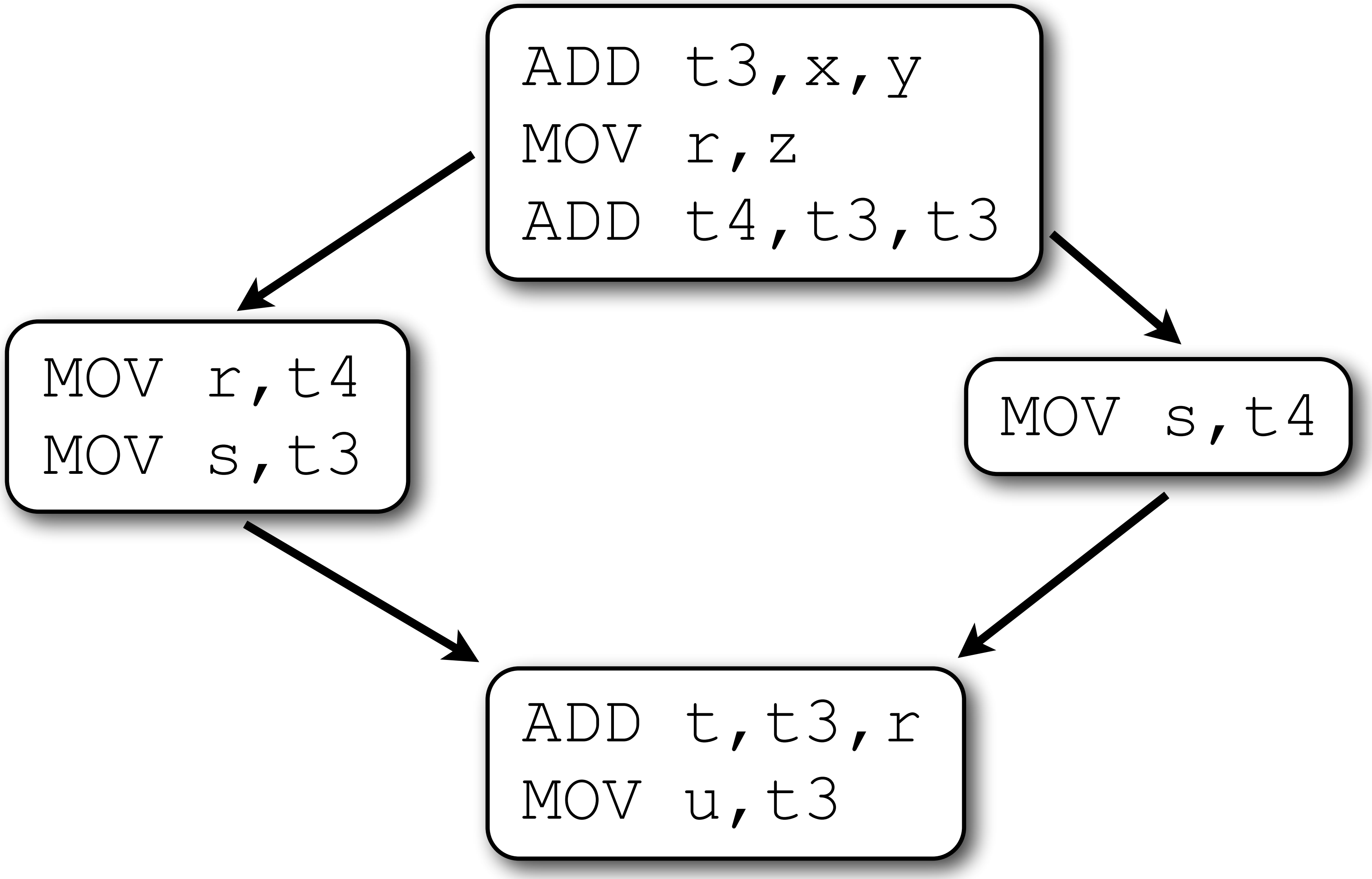
```
ADD t3,x,y
MOV a,t3
MOV b,t3
MOV r,z
ADD t4,t3,t3
```

```
MOV r,t4
MOV s,t3
```

```
MOV s,t4
```

```
ADD t,t3,r
MOV u,t3
```

Putting it all together



```
ADD t3,x,y
MOV r,z
ADD t4,t3,t3
```

```
MOV r,t4
MOV s,t3
```

```
MOV s,t4
```

```
ADD t,t3,r
MOV u,t3
```

Summary

- Some optimisations exist to reduce or remove redundancy in programs
- One such optimisation, common-subexpression elimination, is enabled by AVAIL
- Copy propagation makes CSE practical
- Other code motion optimisations can also help to reduce redundancy
- These optimisations work together to improve code