Code Generation for x86 Machines

The x86 presents several special difficulties when generating code.

- There are only 8 architecturally visible registers, and only 6 of these are allocatable. Deciding what values to keep in registers, and for how long, is a difficult, but crucial, decision.
- Operands may be addressed directly from memory in some instructions. Such instructions avoid using a register, but are longer and add to I-cache pressure.

In "Optimal Spilling for CISC Machines with Few Registers," Appel and George address both of these difficulties.

They use Integer Programming techniques to directly and optimally solve the crucial problem of deciding which live ranges are to be register-resident at each program point. Stores and loads are automatically added to split long live ranges.

Then a variant of Chaitin-style register allocation is used to assign registers to live ranges chosen to be register-resident.

The presentation of this paper, at the 2001 PLDI Conference, is at

www.cs.wisc.edu/~fischer/cs701/cisc.spilling.pdf

Optimistic Coalescing

Given R allocatable registers, Appel and George guarantee that no more than R live ranges are marked as register resident.

This doesn't always guarantee that an R coloring is possible.

Consider the following program fragment:

```
x=0;
while (...) {
  y = x+1;
  print(x);
  z = y+1;
  print(y);
  x = z+1;
  print(z);
}
```

At any given point in the loop body only 2 variables are live, but 3 registers are needed (`x` interferes with `y`, `y` interferes with `z` and `z` interferes with `x`).

We know that we have enough registers to handle all live ranges marked as register-resident, but we may need to "shuffle" register allocations at certain points.

Thus at one point x might be allocated R1 and at some other point it might be placed in R2. Such shuffling implies register to register copies, so we'd like to minimize their added cost.

Appel and George suggest allowing changes in register assignments between program points. This is done by creating multiple variable names for a live range (x_1, x_2, x_3, ...), one for each program point. Variables are connected by assignments between points. Using coalescing, it is expected that most of the assignments will be optimized away.

Using our earlier example, we have the following code with each variable expanded into 3 segments (one for each assignment). Copies of dead variables are removed to simplify the example:

```
x3=0;
while (...) {
  x1 = x3;
  y1 = x1+1;
  print(x1);
  y2 = y1;
  z2 = y2+1;
  print(y2);
  z3 = z2;
  x3 = z3+1;
  print(z3);
}
```

Now a 2 coloring is possible:

x_1: R1, y_1: R2

z_2: R1, y_2: R2

z_3: R1, x_3: R2

(and only $x_1 = x_3$ is retained).

Appel and George found that iterated coalescing wasn't effective (too many copies, most of which are useless).

Instead they recommend *Optimistic Coalescing*. The idea is to first do Chaitin-style reckless coalescing of all copies, even if colorability is impaired.

Then we do graph coloring register allocation, using the cost of copies as the "spill cost." As we select colors, a coalesced node that can't be colored is simply split back to the original source and target variables. Since we always limit the number of live ranges to the number of colors, we know the live ranges must be colorable (with register to register copies sometimes needed).

Using our earlier example, we initially merge x_1 and x_3, y_1 and y_2, z_2 and z_3. We already know this can't be colored with two registers. All three pairs have the same costs, so we arbitrarily stack x_1-x_3, then y_1-y_2 and finally z_2-z_3.

When we unstack, z_2-z_3 gets R1, and y_1-y_2 gets R2. x_1-x_3 must be split back into x_1 and x_3. x_1 interferes with y_1-y_2 so it gets R1. x_3 interferes with z_2-z_3 so it gets R2, and coloring is done.

x_1: R1, y_1: R2

z_2: R1, y_2: R2

z_3: R1, x_3: R2

Reading Assignment

- Read pages 1–30 of "Automatic Program Optimization," by Ron Cytron. (Linked from the class Web page.)

Data Flow Frameworks

- Data Flow Graph:

  Nodes of the graph are basic blocks or individual instructions.

  Arcs represent flow of control.

  Forward Analysis:

    Information flow is the same direction as control flow.

  Backward Analysis:

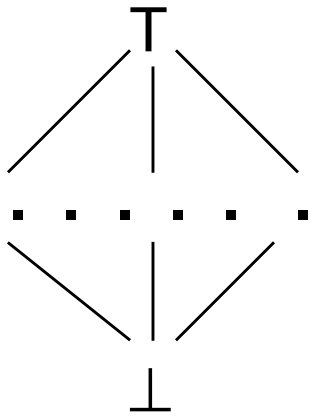
    Information flow is the opposite direction as control flow.

  Bi-directional Analysis:

    Information flow is in both directions. (Not too common.)

- Meet Lattice

  Represents solution space for the data flow analysis.



- Meet operation
  (And, Or, Union, Intersection, etc.)

  Combines solutions from predecessors or successors in the control flow graph.

- Transfer Function

  Maps a solution at the top of a node to a solution at the end of the node (forward flow)

  or

  Maps a solution at the end of a node to a solution at the top of the node (backward flow).

Example: Available Expressions

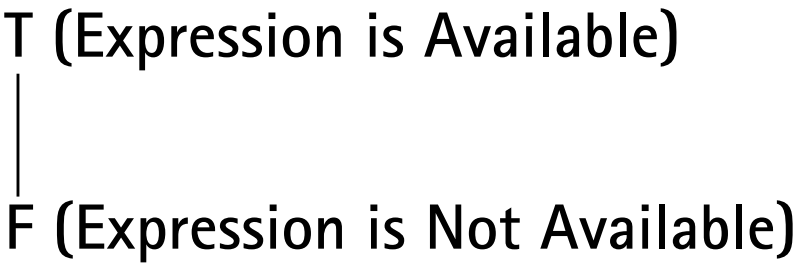
This data flow analysis determines whether an expression that has been previously computed may be reused.

Available expression analysis is a forward flow problem—computed expression values flow forward to points of possible reuse.

The best solution is True—the expression may be reused.

The worst solution is False—the expression may not be reused.

The Meet Lattice is:



As initial values, at the top of the start node, nothing is available. Hence, for a given expression,

$AvailIn(b_0) = F$

We choose an expression, and consider all the variables that contribute to its evaluation.

Thus for e_1=a+b–c, a, b and c are e_1's *operands*.

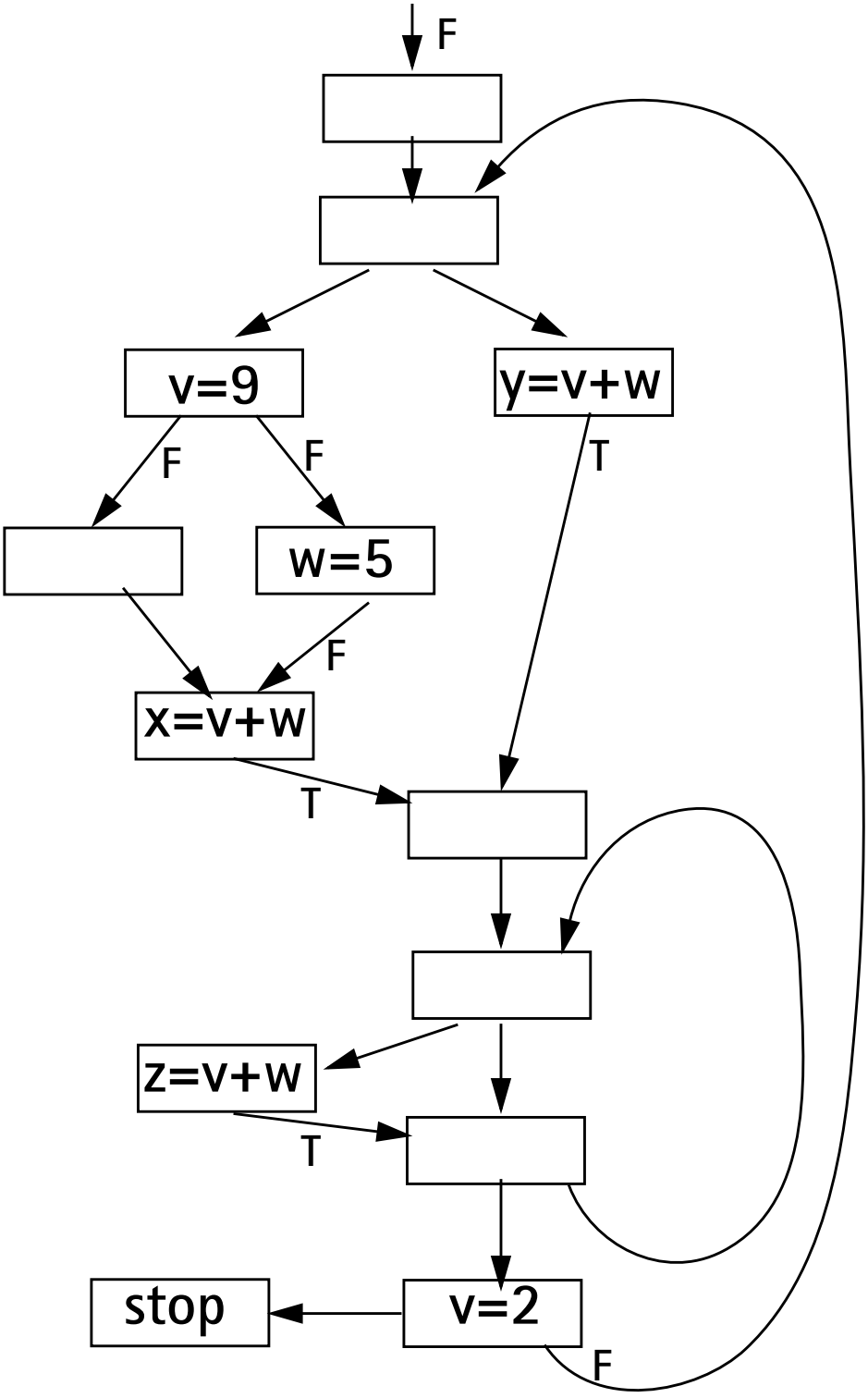
The transfer function for e_1 in block b is defined as:

If e_1 is computed in b after any assignments to e_1's operands in b
Then AvailOut(b) = T
Elsif any of e_1's operands are changed after the last computation of e_1 or e_1's operands are changed without any computation of e_1
Then AvailOut(b) = F
Else AvailOut(b) = AvailIn(b)

The meet operation (to combine solutions) is:

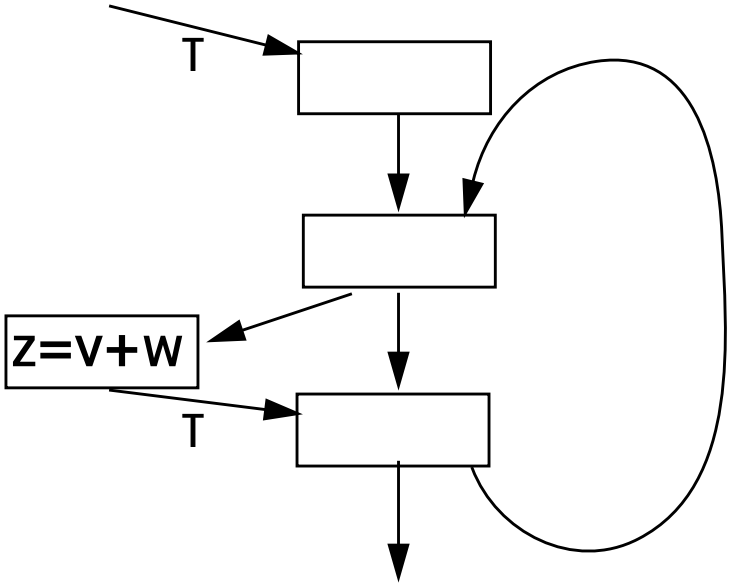
$$\text{AvailIn}(b) = \underset{p \in \text{Pred}(b)}{\text{AND}} \text{AvailOut}(p)$$

Example: e_1=v+w



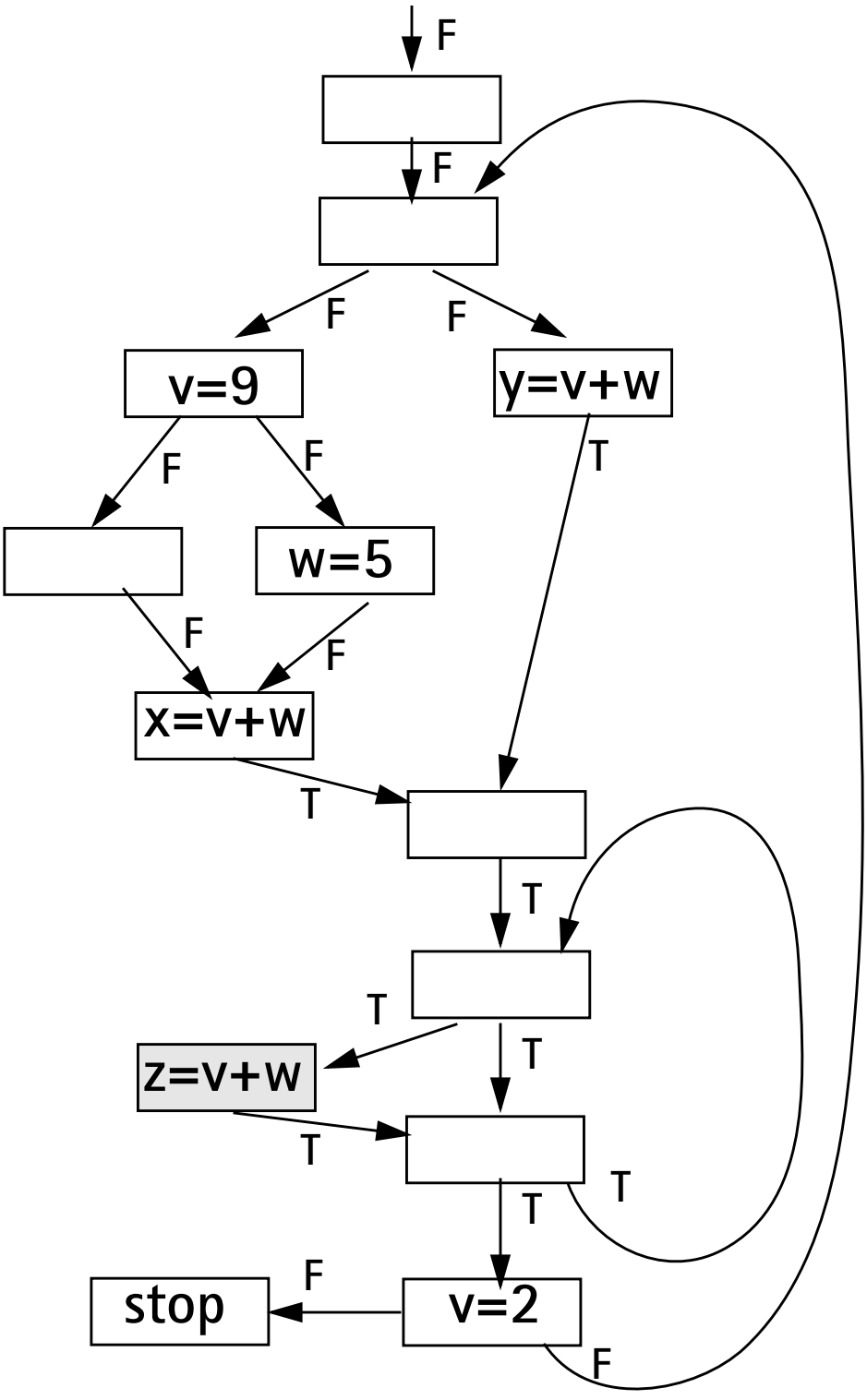
Circularities Require Care

Since data flow values can depend on themselves (because of loops), care is required in assigning initial "guesses" to unknown values. Consider



If the flow value on the loop backedge is initially set to false, it can never become true. (Why?)

Instead we should use True, the *identity* for the AND operation.


F
F
F
F
v=9
y=v+w
F
F
T
w=5
F
F
x=v+w
T
T
T
T
z=v+w
T
T
T
T
F
stop
v=2
F

Very Busy Expressions

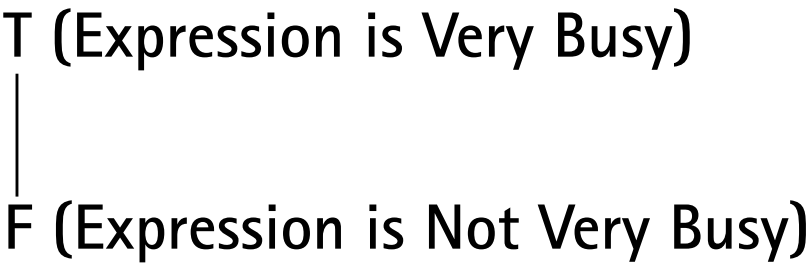
This is an interesting variant of available expression analysis.

An expression is *very busy* at a point if it is *guaranteed* that the expression will be computed at some time in the future.

Thus starting at the point in question, the expression must be reached before its value changes.

Very busy expression analysis is a backward flow analysis, since it propagates information about future evaluations backward to "earlier" points in the computation.

The meet lattice is:



As initial values, at the end of all exit nodes, nothing is very busy. Hence, for a given expression,

$\text{VeryBusyOut}(b_{last}) = F$

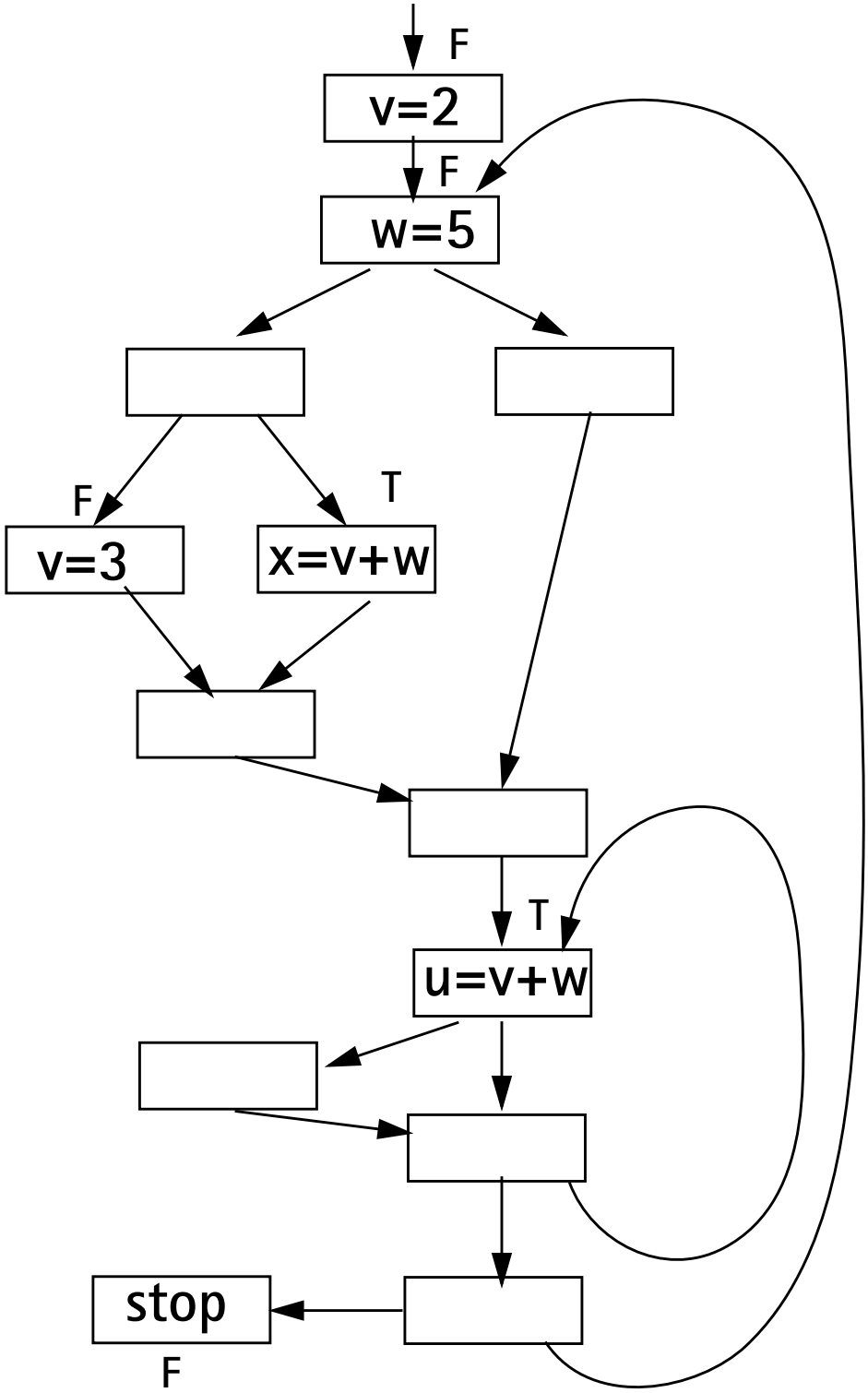
The transfer function for e_1 in block b is defined as:

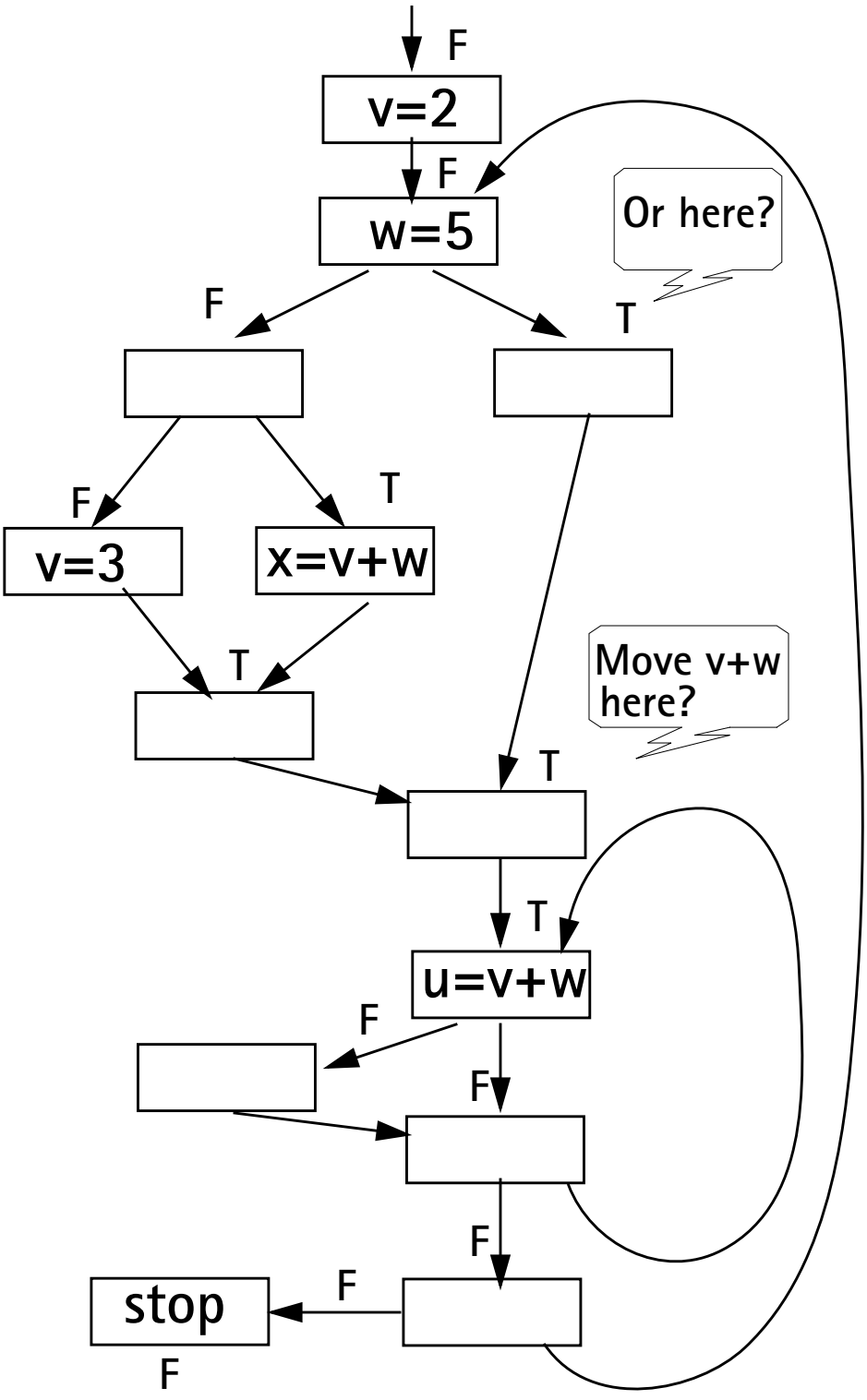
If e_1 is computed in b before any of its operands
Then VeryBusyIn(b) = T
Elsif any of e_1's operands are changed before e_1 is computed
Then VeryBusyIn(b) = F
Else VeryBusyIn(b) = VeryBusyOut(b)

The meet operation (to combine solutions) is:

$$\text{VeryBusyOut}(b) = \bigwedge_{s \in \text{Succ}(b)} \text{VeryBusyIn}(s)$$

Example: e_1=v+w




F
v=2
F
w=5
Or here?
F
T
F
T
v=3
x=v+w
T
Move v+w
here?
T
T
u=v+w
F
F
F
F
stop
F

Identifying Identical Expressions

We can hash expressions, based on hash values assigned to operands and operators. This makes recognizing potentially redundant expressions straightforward.

For example, if H(a) = 10, H(b) = 21 and H(+) = 5, then (using a simple product hash),
H(a+b) = 10×21×5 Mod TableSize

Effects of Aliasing and Calls

When looking for assignments to operands, we must consider the effects of pointers, formal parameters and calls.

An assignment through a pointer (e.g, `*p = val`) *kills* all expressions dependent on variables `p` might point too. Similarly, an assignment to a formal parameter kills all expressions dependent on variables the formal might be bound to.

A call kills all expressions dependent on a variable changeable during the call.

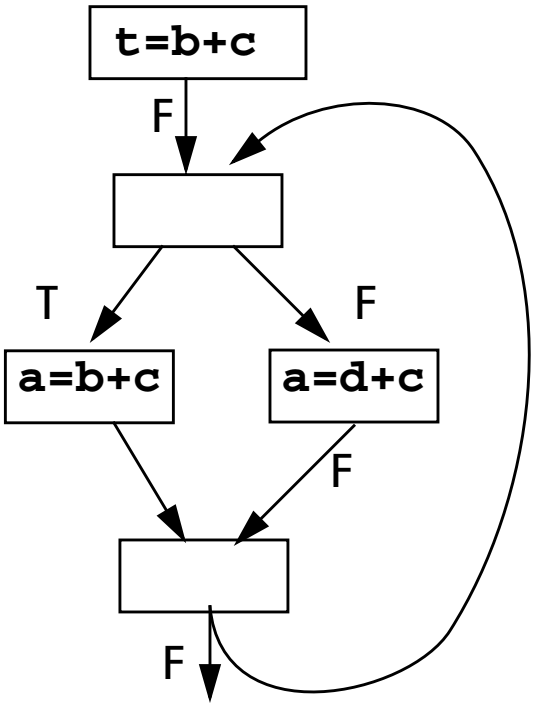
Lacking careful alias analysis, pointers, formal parameters and calls can kill all (or most) expressions.

Very Busy Expressions and Loop Invariants

Very busy expressions are ideal candidates for invariant loop motion.

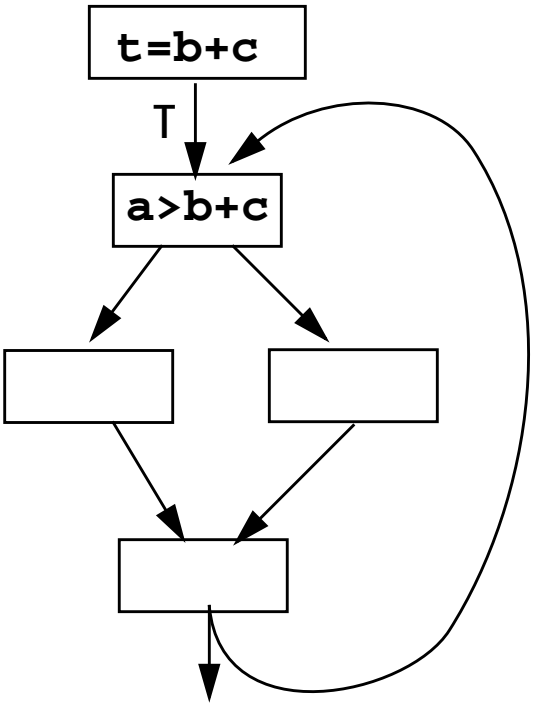
If an expression, invariant in a loop, is also very busy, we know it must be used in the future, and hence evaluation outside the loop must be worthwhile.

```
for (...) {
  if (...)
       a=b+c;
 else a=d+c;}
```



b+c is not very busy at loop entrance

```
for (...) {
  if (a>b+c)
       x=1;
 else x=0;}
```



b+c is very busy at loop entrance