Register allocation & liveness

- We have used a register for every non-leaf node in an expression tree.
- But the number of registers is limited in a real machine.
- How can we reduce the number of registers used?
- We need to know which variables are needed when.

Liveness analysis

Code generation:

- Uses an unlimited number of registers

- Unnecessarily copies between registers

If we know that a value (in a register) will not be used again, we can reuse its register for another variable:

- reduces number of registers needed
- reduces need for copying between registers

Defining and using variables

In an imperative program, each statement (instruction)

- **uses** some variables (registers)
- **defines** some variables (registers)

use(*s*) = set of variables used by statement *s*

def(*s*) = set of variables defined by statement *s*

E.g.

use(“x = y op z”) = {y, z}

def(“x = y op z”) = {x}

Live and dead variables

Each variable x can be defined and/or used many times

For each definition of x, there may be many uses.

A variable is *live* between a definition and its last use (before its next definition).

Live variable analysis: simple method

Works for basic block.

1. Decide which variables are live at the end of block

2. Work backward from end of block, checking whether each variable in each statement is live

Calculates for each statement (instruction):

in(*s*) = set of variables live before statement *s*

out(*s*) = set of variables live after statement *s*

Algorithm:

1. Initialize *out*(*last*) to contain all variables live at end of block
2. For each statement *n*, from last to first:

   if $n \neq last$, $out(n) = in(succ(n))$

   $in(n) = use(n) \cup (out(n) - def(n))$

Example:

```
1:  t1 = a * b
2:  t2 = t1 + a
3:  p = t2 + b
```

Assume that `a`, `b`, `p` are live at end of block.

```
out(3) = {a, b, p}
in(3) = {a, b, t2}
out(2) = {a, b, t2}
in(2) = {a, b, t1}
out(1) = {a, b, t1}
in(1) = {a, b}
```

Shows that `t1` and `t2` are never live at same time:

- Could allocate `t1` and `t2` to same register

Example:

```
1:  ADDI:  R1 ← FP + a
2:  LOAD:  R2 ← M[FP + i]
3:  ADDI:  R3 ← 4
4:  MUL:   R4 ← R2 * R3
5:  ADD:   R5 ← R1 + R4
6:  LOAD:  R6 ← M[FP + x]
7:  STORE: M[R5] ← R6
```

```
out(7) =          {}
out(6) = in(7) = {R5, R6}
out(5) = in(6) = {R5, FP}
out(4) = in(5) = {FP, R1, R4}
out(3) = in(4) = {FP, R1, R2, R3}
out(2) = in(3) = {FP, R1, R2}
out(1) = in(2) = {FP, R1}
in(1) =          {FP}
```

Shows that only 3 registers are necessary, not 6.

Live variable analysis: general method

Algorithm:

1. For each statement n:

   $out(n) = in(n) = \{\}$

2. Repeat

   For each statement n:

   $in'(n) = in(n)$

   $out'(n) = out(n)$

   $out(n) = \cup_{s \in succ(n)} in(s)$

   $in(n) = use(n) \cup (out(n) - def(n))$

   until $in'(n) == in(n)$ && $out'(n) == out(n)$ for all n

Example: Fibonacci program

```
1:  x = 0
2:  y = 1
3:  if (y > 1000) goto 8
4:  z = x + y
5:  x = y
6:  y = z
7:  goto 3
8:  write(y)
```

1st iteration:

out(8) = {}
in(8) = {y}
out(7) = in(3) = {}
in(7) = {}
out(6) = in(7) = {}
in(6) = {z}
out(5) = in(6) = {z}
in(5) = {z, y}
out(4) = in(5) = {z, y}
in(4) = {y, x}
out(3) = in(4) ∪ in(8) = {y, x} ∪ {y} = {y, x}
in(3) = {y, x}
out(2) = in(3) = {y, x}
in(2) = {x}
out(1) = in(2) = {x}
in(1) = {}

2nd iteration:

out(8) = {}
in(8) = {y}
out(7) = in(3) = {y, x}
in(7) = {y, x}
out(6) = in(7) = {y, x}
in(6) = {x, z}
out(5) = in(6) = {x, z}
in(5) = {z, y}
out(4) = in(5) = {z, y}
in(4) = {y, x}
out(3) = in(4) ∪ in(8) = {y, x} ∪ {y} = {y, x}
in(3) = {y, x}
out(2) = in(3) = {y, x}
in(2) = {x}
out(1) = {x}
in(1) = {}

	⇐ Which variables are live here?
t1 = x + 1	
	⇐ Which variables are live here?
t2 = t1 + y	
	⇐ Which variables are live here?
t3 = x + 1	
	⇐ Which variables are live here?
t4 = t2 * t1	
	⇐ Which variables are live here?
write(t4)	
	⇐ No variables are live here

Answer

x y	⇐ Which variables are live here?
t1 = x + 1	
t1 x y	⇐ Which variables are live here?
t2 = t1 + y	
t1 t2 x	⇐ Which variables are live here?
t3 = x + 1	
t1 t2	⇐ Which variables are live here?
t4 = t2 * t1	
t4	⇐ Which variables are live here?
write(t4)	
	⇐ No variables are live here