Register Allocation

GRAPH COLORING

Graph Coloring Algorithm

Input: a register-interference graph *G*

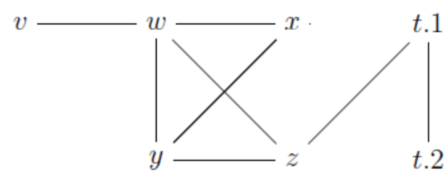
Output: an assignment color[*v*] for each node $v \in G$

$W \leftarrow vertices(G)$
while $W \neq \emptyset$ **do**
  pick a node u from W with the highest saturation,
    breaking ties randomly
  find the lowest color c that is not in $\{\text{color}[v] \ : \ v \in \text{adjacent}(v)\}$
  $\text{color}[u] \leftarrow c$
  $W \leftarrow W - \{u\}$

$$\text{saturation}(u) = \{c \mid \exists v.v \in \text{adjacent}(u) \text{ and } \text{color}(v) = c\}$$

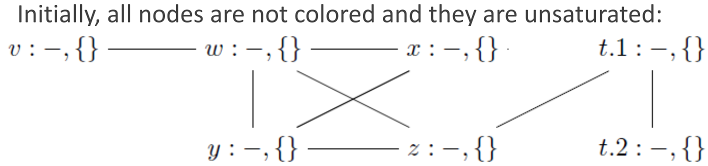
adjacent(*u*) is the set of nodes adjacent to *u*.

Interference Graph for our Example



Coloring the Interference Graph

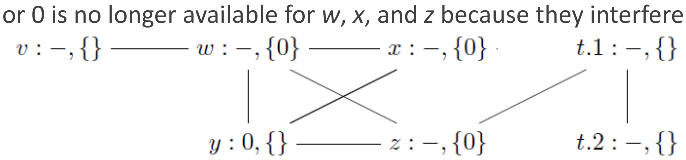
Initially, all nodes are not colored and they are unsaturated:



We select a maximally saturated node and color it 0.

In this case we have a 7-way tie, so we arbitrarily pick *y*.

Color 0 is no longer available for *w*, *x*, and *z* because they interfere with *y*:

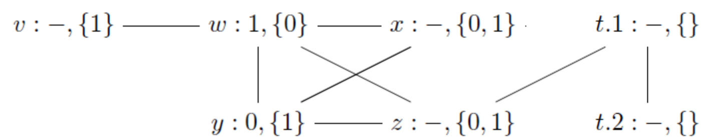


Coloring the Interference Graph

Repeat the process, selecting another maximally saturated node.

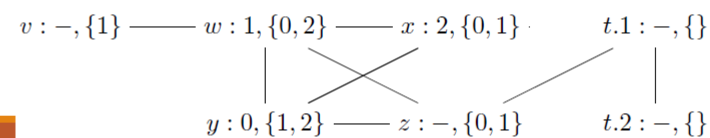
There is a three-way tie between *w*, *x*, and *z*.

We color *w* with 1:



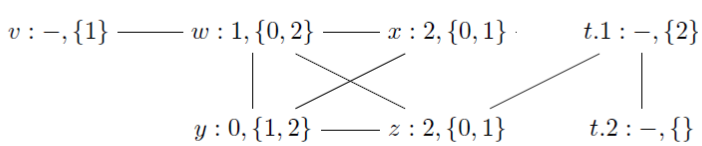
The most saturated nodes are now *x* and *z*.

We color *x* with the next available color which is 2:

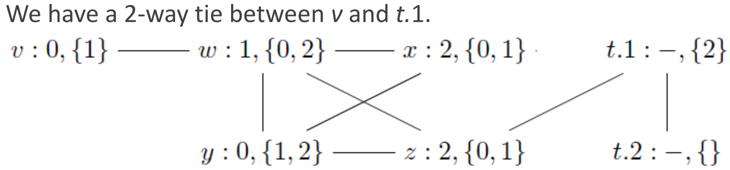


Coloring the Interference Graph

Node *z* is the next most highly saturated, so we color *z* with 2:

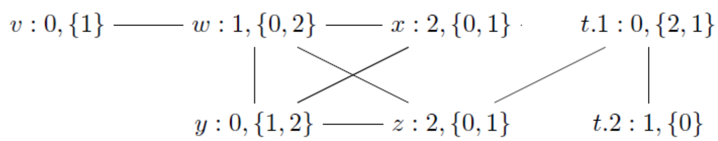


We have a 2-way tie between *v* and *t*.1.



Coloring the Interference Graph

In the last two steps of the algorithm, we color *t*.1 with 0 then *t*.2 with 1:



With the coloring complete, we can finalize the assignment of variables to registers and stack locations.

Coloring the Interference Graph

If we have *k* registers, map the first *k* colors to registers and the rest to stack locations.

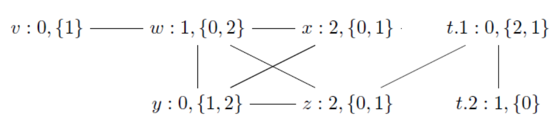
Suppose there is just one register: **rbx**.

We get the following mapping of colors to registers and stack allocations:

```
{0 -> %rbx, 1 -> -8(%rbp), 2 ->  -16(%rbp), ...}
```

And the following assignment:

```
{v -> %rbx, w -> -8(%rbp), x -> -16(%rbp), y -> %rbx,
 z -> -16(%rbp), t.1 -> %rbx, t.2 -> -8(%rbp)}
```



Applying Assignment to Example

```
(program (v w x y z)
  (movq (int 1) (var v))
  (movq (int 46) (var w))
  (movq (var v) (var x))
  (addq (int 7) (var x))
  (movq (var x) (var y))
  (addq (int 4) (var y))
  (movq (var x) (var z))
  (addq (var w) (var z))
  (movq (var y) (var t.1))
  (negq (var t.1))
  (movq (var z) (var t.2))
  (addq (var t.1) (var t.2))
  (movq (var t.2) (reg rax)))
```

⇒

```
(program 16
  (movq (int 1) (reg rbx))
  (movq (int 46) (deref rbp -8))
  (movq (reg rbx) (deref rbp -16))
  (addq (int 7) (deref rbp -16))
  (movq (deref rbp -16) (reg rbx))
  (addq (int 4) (reg rbx))
  (movq (deref rbp -16) (deref rbp -16))
  (addq (deref rbp -8) (deref rbp -16))
  (movq (reg rbx) (reg rbx))
  (negq (reg rbx))
  (movq (deref rbp -16) (deref rbp -8))
  (addq (reg rbx) (deref rbp -8))
  (movq (deref rbp -8) (reg rax)))
```

Patching the Example

```
(program (v w x y z)
  (movq (int 1) (var v))
  (movq (int 46) (var w))
  (movq (var v) (var x))
  (addq (int 7) (var x))
  (movq (var x) (var y))
  (addq (int 4) (var y))
  (movq (var x) (var z))
  (addq (var w) (var z))
  (movq (var y) (var t.1))
  (negq (var t.1))
  (movq (var z) (var t.2))
  (addq (var t.1) (var t.2))
  (movq (var t.2) (reg rax)))
```

⇒

```
(program 16
  (movq (int 1) (reg rbx))
  (movq (int 46) (deref rbp -8))
  (movq (reg rbx) (deref rbp -16))
  (addq (int 7) (deref rbp -16))
  (movq (deref rbp -16) (reg rbx))
  (addq (int 4) (reg rbx))
  (movq (deref rbp -16) (deref rbp -16))
  (addq (deref rbp -8) (deref rbp -16))
  (movq (reg rbx) (reg rbx))
  (negq (reg rbx))
  (movq (deref rbp -16) (deref rbp -8))
  (addq (reg rbx) (deref rbp -8))
  (movq (deref rbp -8) (reg rax)))
```

```
(movq (deref rbp -8) (reg rax)
(addq (reg rax) (deref rbp -16))
```

Register Allocation in a Structured Language

Let *use(x, B)* be the number of times *x* is used in block *B* prior to any definition of *x.*

Let *live(x, B)* be 1, if *x* is live on exit of block *B* and is assigned a value in *B* and 0 otherwise.

An approximate formula for the benefit of allocating a register to *x* within loop *L* is:

$$\sum_{\text{blocks } B \text{ in } L} use(x, B) + 2 * live(x, B)$$

Example

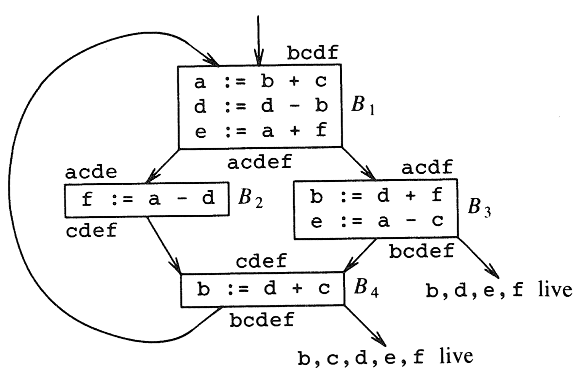


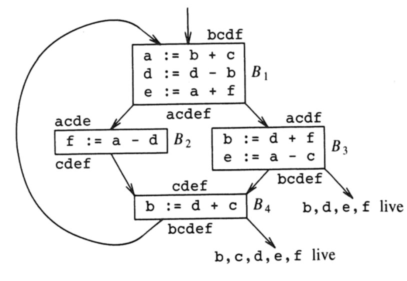
Fig. 9.13. Flow graph of an inner loop.

Example Cont'd

a: - Live on exit of B_1 and assigned a value there: 2
- Used in B_2 and $B_{3:}$: 2
- Total: 4

b: - Live on exit of B_3 and B_4 and assigned a value there: 4
- Used twice in B_1: 2
- Total: 6

c: - Not live anywhere: 0
- Used in B_1, B_3 and $B_{4:}$ 3
- Total: 3



$$\sum_{\text{blocks } B \text{ in } L} use(x, B) + 2 * live(x, B)$$

Fig. 9.13. Flow graph of an inner loop.

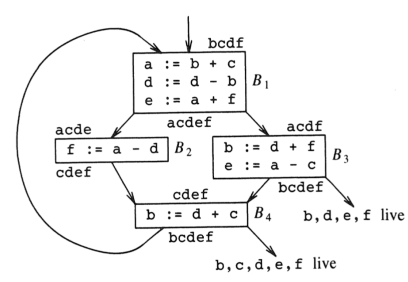
Example Cont'd

Class exercise: determine the values for the following variables:

d:

e:

f:



$$\sum_{\text{blocks } B \text{ in } L} use(x, B) + 2 * live(x, B)$$

Fig. 9.13. Flow graph of an inner loop.

Example Cont'd

d: - Live on exit of B_1 and assigned a value there: 2
- Used in B_1, B_2, B_3 and B_4 : 4
- Total: 6

e: - Live on exit of B_1 and B_3 and assigned a value there: 4
- Not used anywhere: 0
- Total: 4

f: - Live on exit of B_2 and assigned a value there: 2
- Used in B_1 and B_3: 2
- Total: 4

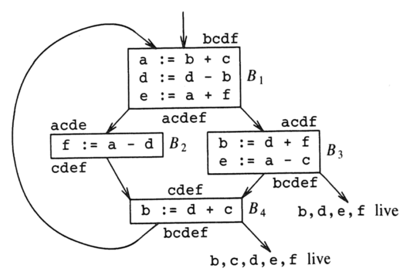


Fig. 9.13. Flow graph of an inner loop.

$$\sum_{\text{blocks } B \text{ in } L} use(x, B) + 2 * live(x, B)$$

Example Cont'd

If we have three registers: **`R0`**, **`R1`** and **`R2`**, then assign **`a`** to **`R0`**, **`b`** to **`R1`** and **`d`** to **`R2`**.

Variables **`e`** and **`f`** could have been used instead of **`a`**.

Example Cont'd

a to **R0**,
b to **R1**
d to **R2**.

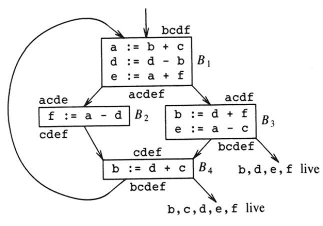


Fig. 9.13. Flow graph of an inner loop.