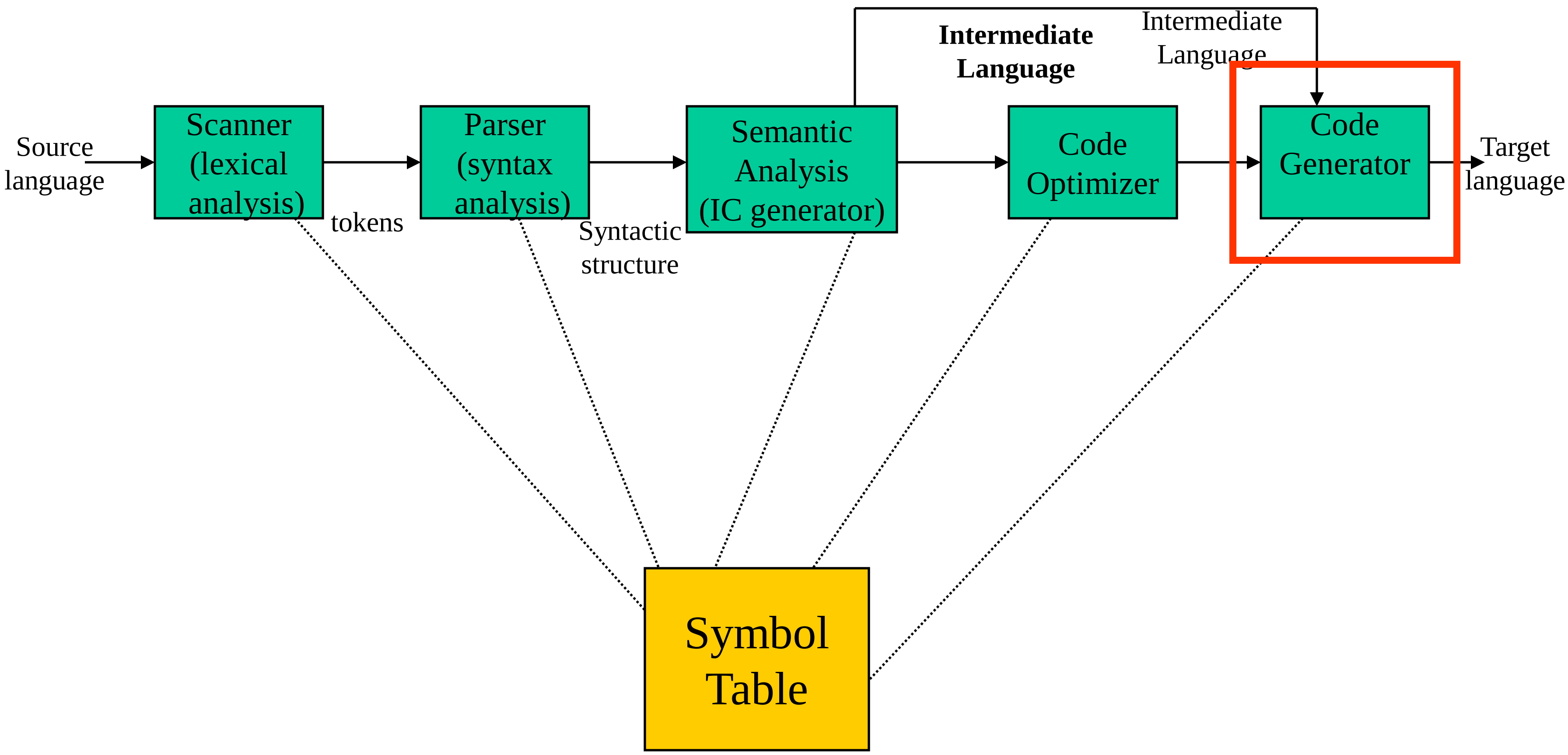
Code Generation-II

CS 346

Compiler Architecture



Register Allocation

How to best use the bounded number of registers?

- Reducing load/store operations
- What are the best values to keep in registers?
- When can we ‘free’ registers?

Complications:

- special purpose registers
- operators requiring multiple registers

Register Allocation Algorithms

- Local (basic block level):
  - **Basic - using liveness information**
  - **Register Allocation using graph coloring**
- Global (CFG)
  - Need to use global liveness information

Basic Code Generation

- Deal with each basic block individually
- Compute liveness information for the block
- Using liveness information, generate code that uses registers as best as possible
- At end, generate code that saves any live values left in registers
-

Concept: Variable Liveness

- For some statement *s*, variable *x* is **live** if
  - there is a statement *t* that uses *x*
  - there is a path in the CFG from *s* to *t*
  - there is no assignment to *x* on some path from *s* to *t*
- **A variable is *live* at a given point in the source code if it could be used before it is defined.**
- Liveness tells us whether we care about the value held by a variable

Example: When is a live?

a := b + c

t1 := a * a

a is live

b := t1 + a

c := t1 * b

t2 := c + b

a := t2 + t2

Assume a,b and c are used after this basic block

Example: When is b live?

a := b + c

t1 := a * a

b := t1 + a

c := t1 * b

t2 := c + b

a := t2 + t2

Assume a,b and c are used after this basic block

Computing live status in basic blocks

Input: A basic block.

Output: For each statement, set of live variables

1. Initially all non-temporary variables go into live set (L).
2. for i = *last* statement to *first* statement:

   for statement i: x := y op z

   1. Attach L to statement i.
   2. Remove x from set L.
   3. Add y and z to set L.

Example

live = {

a := b + c

live = {

t1 := a * a

live = {

b := t1 + a

live = {

c := t1 * b

live = {

t2 := c + b

live = {

a := t2 + t2

live = {a,b,c}

Example Answers

live = {}

a := b + c

live = {}

t1 := a * a

live = {}

b := t1 + a

live = {}

c := t1 * b

live = {}

t2 := c + b

live = {b,c,t2}

a := t2 + t2

live = {a,b,c}

Example Answers

live = {}

a := b + c

live = {}

t1 := a * a

live = {}

b := t1 + a

live = {}

c := t1 * b

live = {b,c}

t2 := c + b

live = {b,c,t2}

a := t2 + t2

live = {a,b,c}

Example Answers

live = {}

a := b + c

live = {}

t1 := a * a

live = {}

b := t1 + a

live = { b,t1}

c := t1 * b

live = {b,c}

t2 := c + b

live = {b,c,t2}

a := t2 + t2

live = {a,b,c}

Example Answers

live = {}

a := b + c

live = {}

t1 := a * a

live = {a,t1}

b := t1 + a

live = { b,t1}

c := t1 * b

live = {b,c}

t2 := c + b

live = {b,c,t2}

a := t2 + t2

live = {a,b,c}

Example Answers

live = {}

a := b + c

live = {a}

t1 := a * a

live = {a,t1}

b := t1 + a

live = { b,t1}

c := t1 * b

live = {b,c}

t2 := c + b

live = {b,c,t2}

a := t2 + t2

live = {a,b,c}

Example Answers

live = {b,c}

a := b + c

live = {a}

t1 := a * a

live = {a,t1}

b := t1 + a

live = { b,t1}

c := t1 * b

live = {b,c}

t2 := c + b

live = {b,c,t2}

a := t2 + t2

live = {a,b,c}

Basic Code Generation

- Deal with each basic block individually
- Compute liveness information for the block
- Using liveness information, generate code that uses registers as much as possible
- At end, generate code that saves any live values left in registers

Basic Code Generation

Idea: Deal with the instructions from beginning to end. For each instruction,

- Use registers whenever possible
- A non-live value in a register can be discarded, freeing that register

Data Structures:

- Register descriptor- register status (empty, full) and contents (one or more "values"): keeps track of what is there in registers
- Address descriptor-the location (or locations) where the current value for a variable can be found (*register, stack, memory*)

•

Instruction type: `x := y op z`

1. Choose R_x, the register where the result (x) will be kept.
   1. If y (or z) is in a register t alone and not live, choose R_x = t
   2. Else if there is a free register t, choose R_x = t
   3. Else must free up a register for R_x
2. Find R_y. If y is not in a register, generate load into a free register (or R_x)
3. Find R_z. If z is not in a register, generate load into a free register (can use R_x if not used by y).
4. Generate: OP R_x, R_y, R_z

Instruction type: `x := y op z`

5. Update information about the current best location of x
6. If x is in a register, update that register's information
7. If y and/or z are not live after this instruction, update register and address descriptors accordingly.

Example Code

live = {b,c}

a := b + c

live = {a}

t1 := a * a

live = {a,t1}

b := t1 + a

live = { b,t1}

c := t1 * b

live = {b,c}

t2 := c + b

live = {b,c,t2}

a := t2 + t2

live = {a,b,c}

Returning to live Example

- Initially

  Three Registers: (-, -, -) all empty

  current values: (a,b,c,t1,t2) = (m,m,m, -, -)

- instruction 1: a := b + c, Live = {a }

  R_a = \$t0, R_b = \$t0 , R_c = \$t1

```
lw $t0,b
lw $t1,c
add $t0,$t0,$t1
```

Don't need to keep track of b or c since aren't live.

Registers: (a, -, -)   current values: (\$t0,m,m, -, -)

- instruction 2: t1 := a * a, Live = {a,t1}

  R_{t1} = \$t1 (since a is live)

  ```
  mul $t1,$t0,$t0
  ```

  Registers: (a,t1, -)   current values: (\$t0,m,m,\$t1, -)

- instruction 3: b := t1 + a, Live = {b,t1}

  Since a is not live after call, R_b = \$t0

  ```
  add $t0,$t1,$t0
  ```

  Registers: (b,t1, -)   current values: (m,\$t0,m,\$t1, -)

- instruction 4: c := t1 * b, Live = {b,c }

  Since t1 is not live after call R_c = $t1

  ```
  mul $t1,$t1,$t0
  ```

  Registers: (b,c, -)   current values: (m,$t0,$t1, -, -)

- instruction 5: t2 := c + b, Live = {b,c,t2 }

  R_{t2} = $t2

  ```
  add $t2,$t1,$t0
  ```

  Registers: (b,c,t2)   current values: (m,$t0,$t1, -,$t2)

- instruction 6: a := t2 + t2, Live = {a,b,c}

```
add $t2,$t2,$t2
```

Registers: (b,c,a) current values: ($t2,$t0,$t1, -,-)

- Since end of block, move live variables:

```
sw $t2,a
sw $t0,b
sw $t1,c
```

all registers available

all live variables moved to memory

Generated code

```
lw $t0,b
lw $t1,c
add $t0,$t0,$t1
mul $t1,$t0,$t0
add $t0,$t1,$t0
mul $t1,$t1,$t0
add $t2,$t1,$t0
add $t2,$t2,$t2
sw $t2, a
sw $t0,b
sw $t1,c
```

a := b + c

t1 := a * a

b := t1 + a

c := t1 * b

t2 := c + b

a := t2 + t2

Cost = 16

How does this compare to naïve approach?

Register Allocation with Graph Coloring

Local register allocation-*graph coloring problem*

Uses *liveness* information

Allocate *K* registers where each register is associated with one of the *K* colors

Graph Coloring

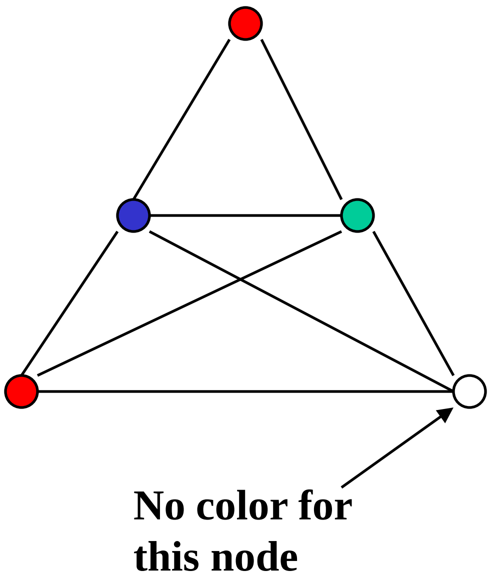
- The coloring of a graph G = (V,E) is a mapping C: V→ S, where S is a finite set of colors, such that if edge vw is in E, C(v) <> C(w)

- NP complete problem (for more than 2 colors)→ no polynomial time solution

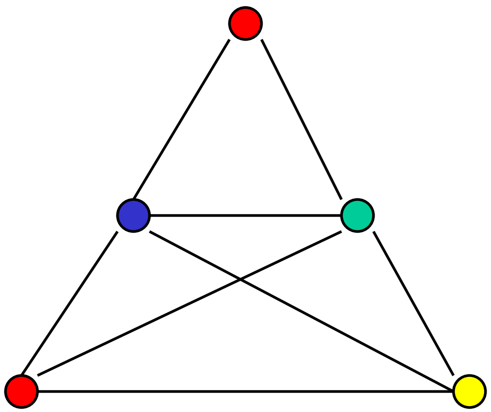
- Fortunately there are approximation algorithms!

Coloring a graph with K colors

K = 3



K = 4



Register Allocation and Graph K-Coloring

K = number of available registers

G = (V,E) where

- Vertex set: V= {V_s | s is a program variable}
- Edge: $V_s V_t$ in E if s and t can be live at the same time

G is an '*interference graph*'

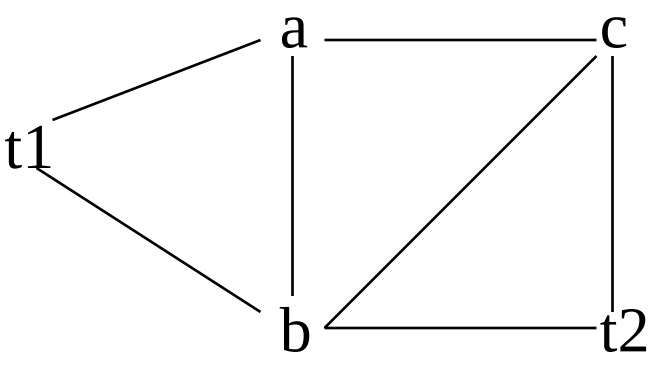
Algorithm: K registers

1. Compute liveness information for the basic block

2. Create interference graph G - one node for each variable, an edge connecting any two variables alive simultaneously

Example: Interference Graph

a := b + c	{b,c}
t1 := a * a	{a}
b := t1 + a	{t1,a}
c := t1 * b	{b,t1}
t2 := c + b	{b,c}
a := t2 + t2	{b,c,t2}
	{a,b,c}



Algorithm: K registers

3. **Simplify**-For any node *m* with fewer than *K* neighbors,

    a. remove it from the graph

    b. push it onto a stack

    c. If G-m can be colored with K colors, so can G

    d. If we reduce the entire graph, goto step 5

Algorithm: K registers

4. **Spill**- If we get to the point where we are left with only nodes with degree $>=$ K,

   a. mark some node for potential spilling
   b. remove and push onto stack
   c. back to step 3

Choosing a Spill Node

Potential criteria:

- Random
- Most neighbors
- Longest live range (in code)
  - with or without taking the access pattern into consideration

Algorithm: K registers

5. Assign colors-

a. start with empty graph
b. rebuild graph by popping elements off the stack
c. put them back into the graph
d. assign them colors different from neighbors

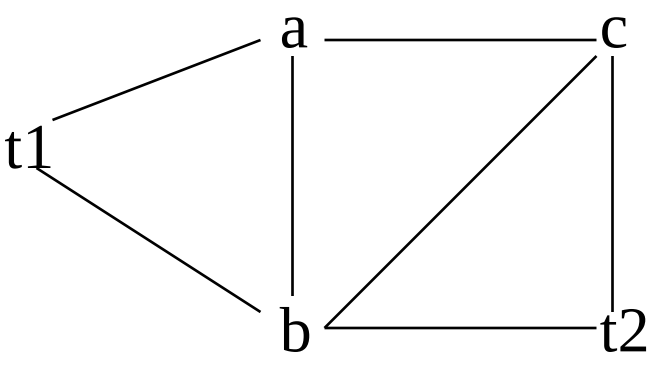
Potential spill nodes may or may not be colorable

Rewriting the code

- Want to be able to remove some edges in the interference graph
  - write variable to memory earlier
  - compute/read in variable later

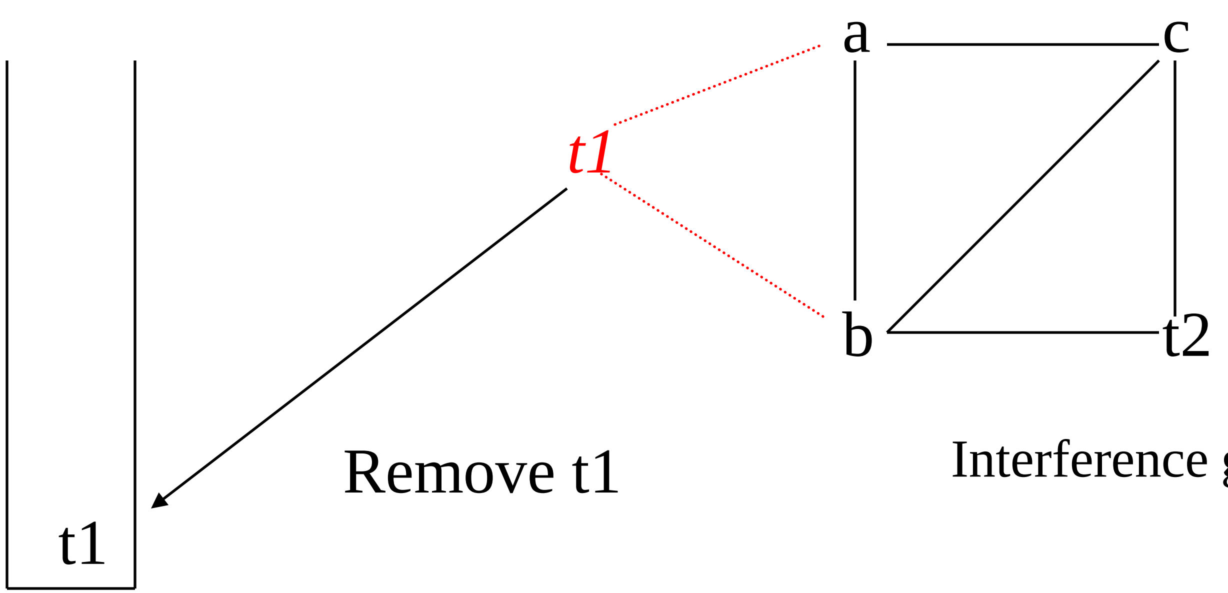
Back to example

a := b + c	{b,c}
t1 := a * a	{a}
b := t1 + a	{t1,a}
c := t1 * b	{b,t1}
t2 := c + b	{b,c}
a := t2 + t2	{b,c,t2}
	{a,b,c}



Example, k = 3

Assume k = 3

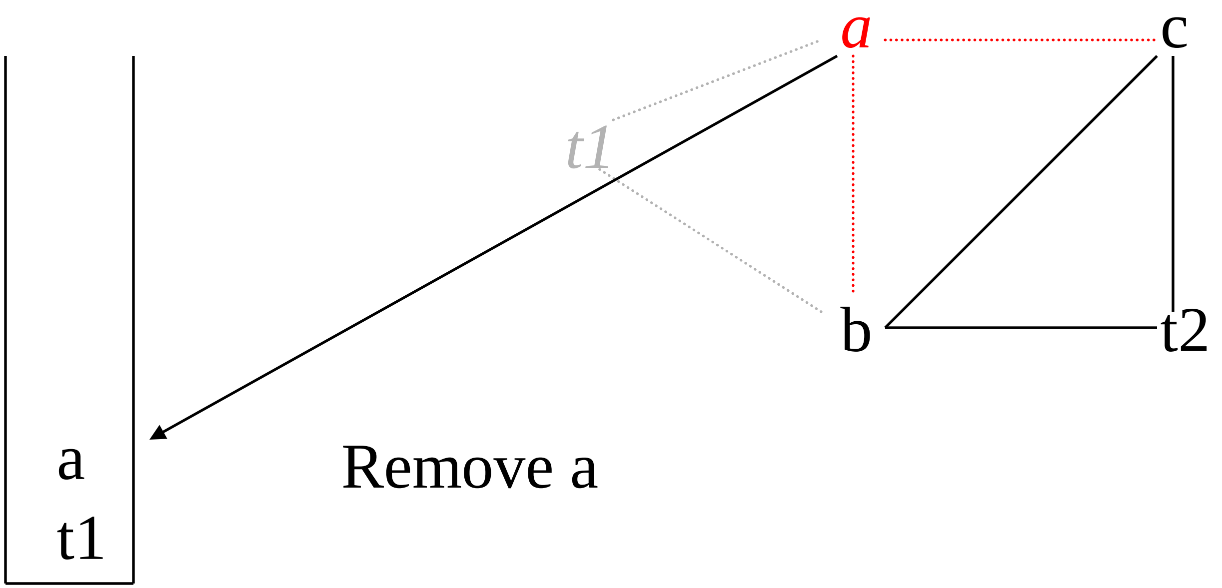


Remove t1

Interference graph

Example

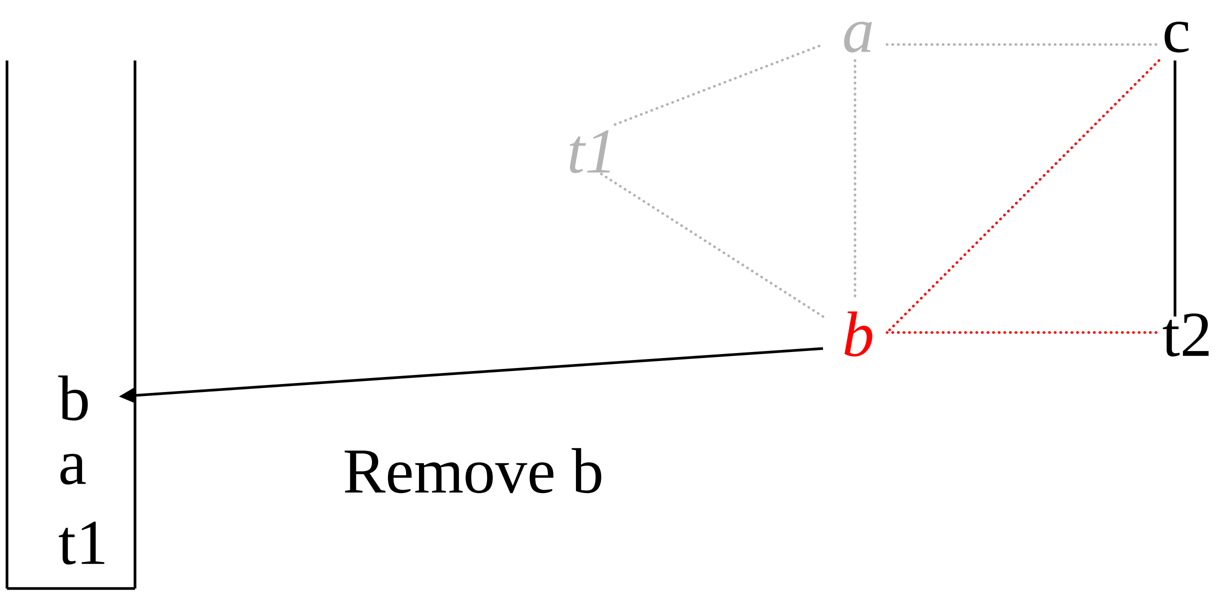
Assume k = 3



Remove a

Example

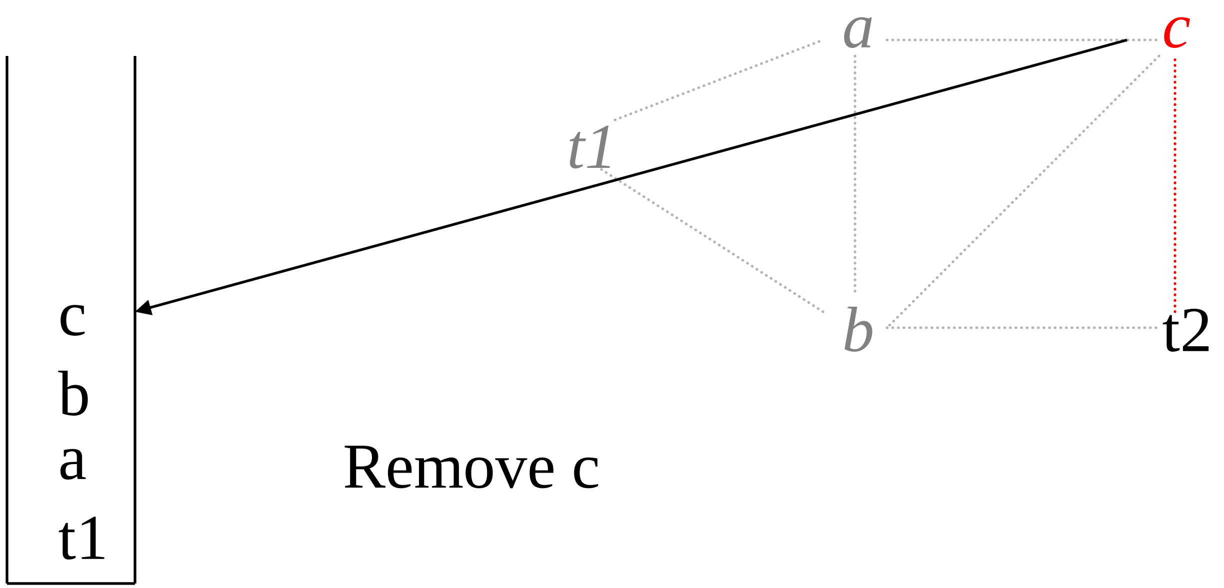
Assume k = 3



Remove b

Example

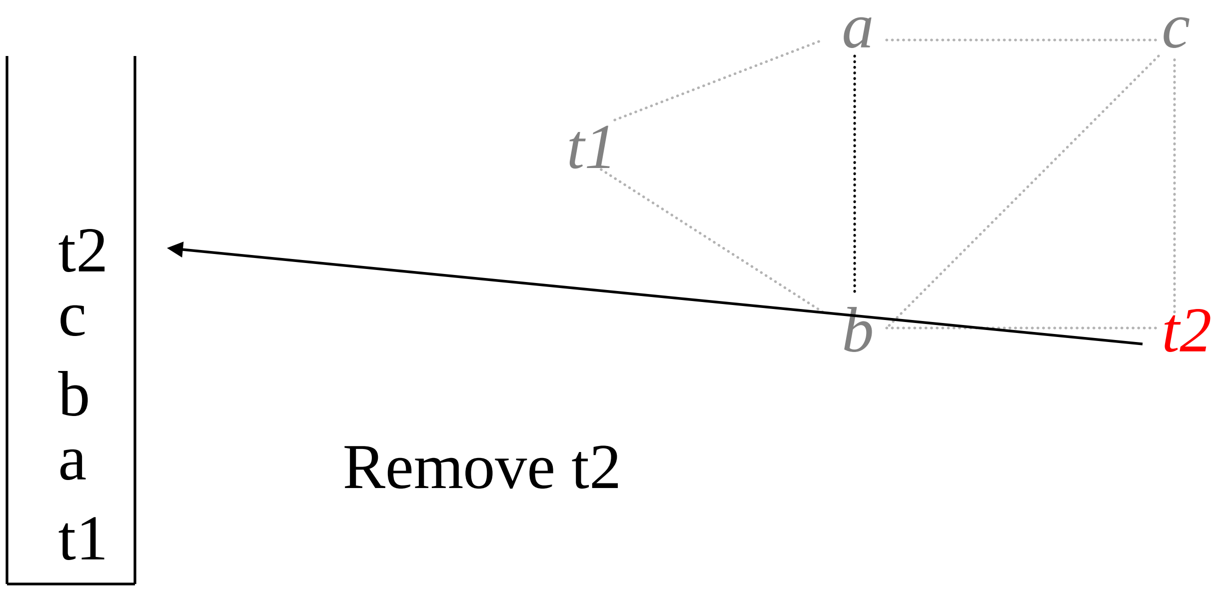
Assume k = 3



Remove c

Example

Assume k = 3



Rebuild the graph

Assume k = 3

c
b
a
t1

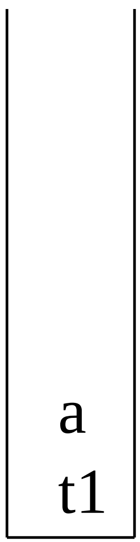
t2

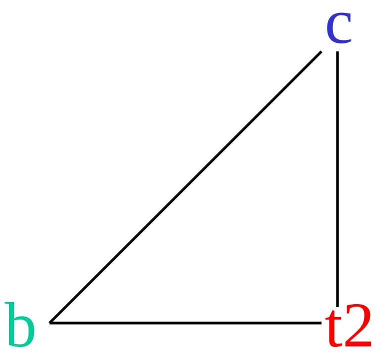
Example

Assume k = 3

Example

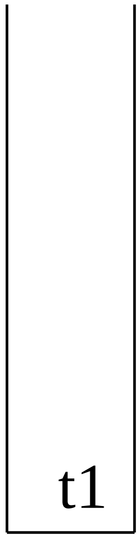
Assume k = 3

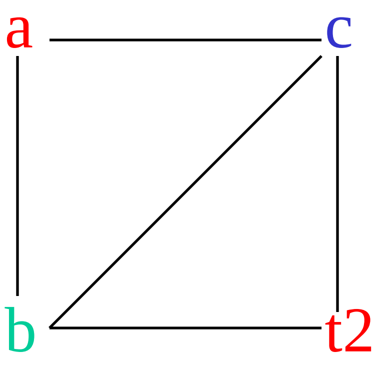




Example

Assume k = 3

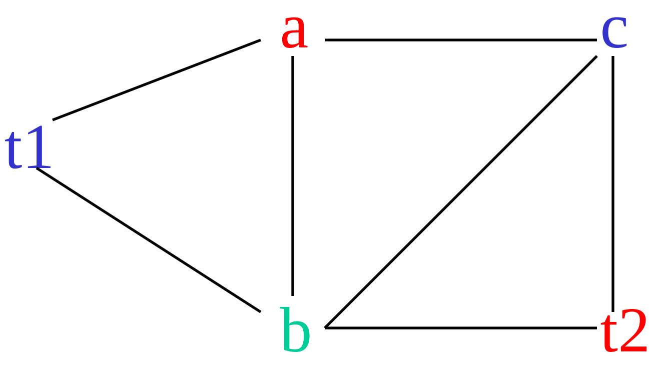




Example

Assume k = 3

a	t0
b	t1
c	t2
t1	t2
t2	t0



Back to example

a := b + c

t1 := a * a

b := t1 + a

c := t1 * b

t2 := c + b

a := t2 + t2

a	t0
b	t1
c	t2
t1	t2
t2	t0

```
lw $t1,b
lw $t2,c
add $t0,$t1,$t2
mul $t2,$t0,$t0
add $t1,$t2,$t0
mul $t2,$t2,$t1
add $t0,$t2,$t1
add $t0,$t0,$t0
sw $t0,a
sw $t1,b
sw $t2,c
```

Generated code: Basic

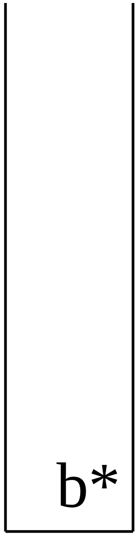
```
lw $t0,b
lw $t1,c
add $t0,$t0,$t1
mul $t1,$t0,$t0
add $t0,$t1,$t0
mul $t1,$t1,$t0
add $t2,$t1,$t0
add $t2,$t2,$t2
sw $t2, a
sw $t0,b
sw $t1,c
```

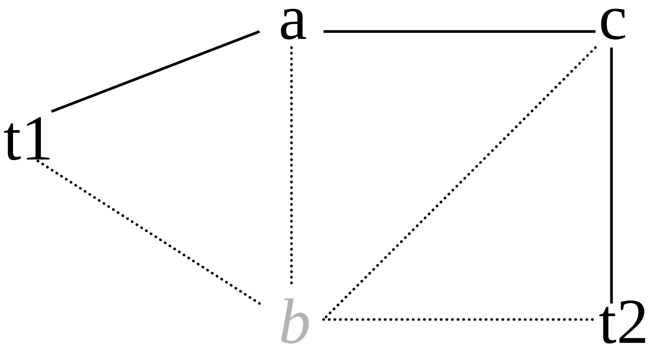
Generated Code: Coloring

```
lw $t1,b
lw $t2,c
add $t0,$t1,$t2
mul $t2,$t0,$t0
add $t1,$t2,$t0
mul $t2,$t2,$t1
add $t0,$t2,$t1
add $t0,$t0,$t0
sw $t0,a
sw $t1,b
sw $t2,c
```

Example, k = 2

Assume k = 2

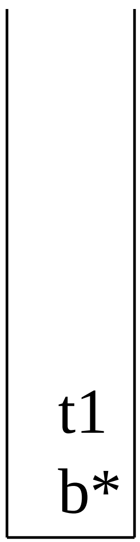


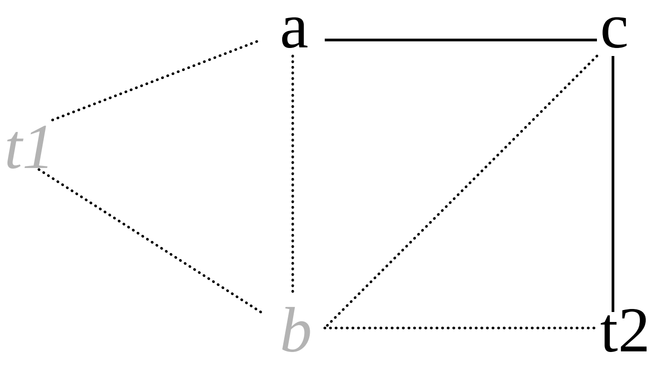


Remove b as spill

Example

Assume k = 2





Remove t1

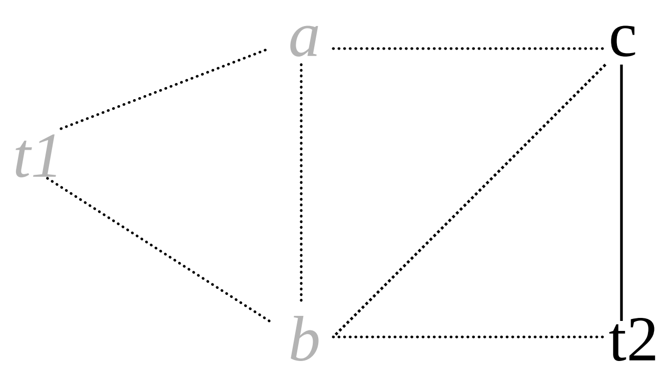
Example

Assume k = 2

a

t1

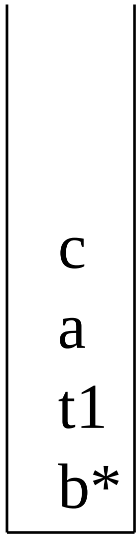
b*



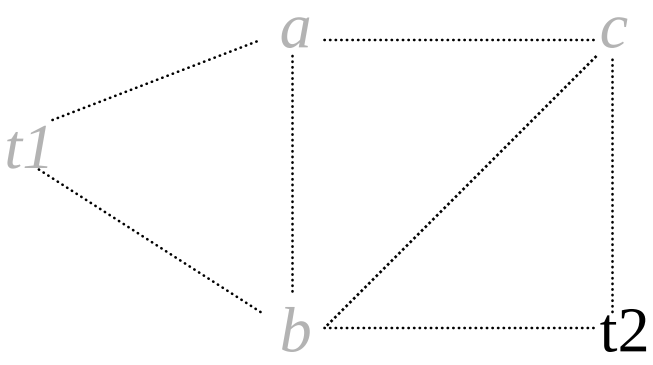
Remove a

Example

Assume k = 2



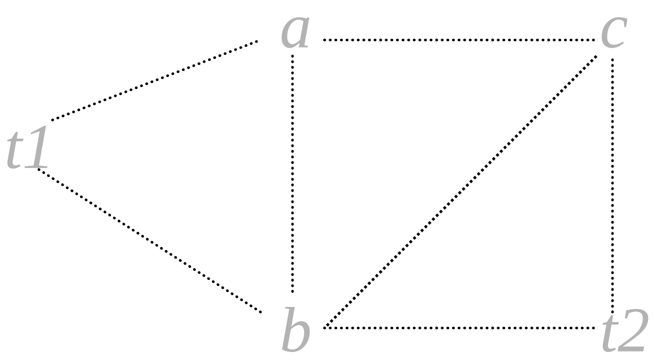
Remove c



Example

Assume k = 2

t2
c
a
t1
b*



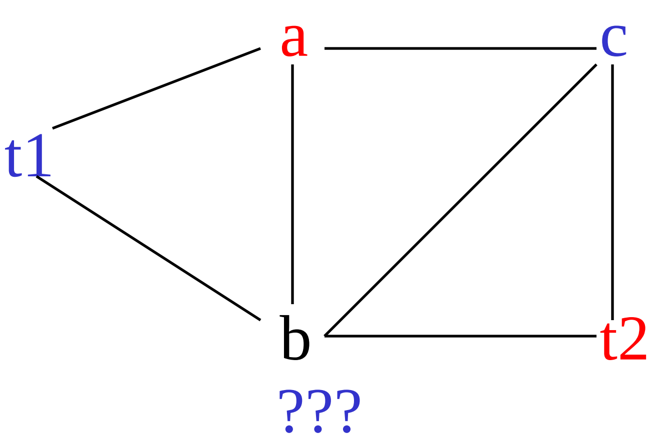
Remove t2

Example

Assume k = 2

Can flush b out to memory, creating a smaller window

a := b + c	{b,c}
t1 := a * a	{a}
b := t1 + a	{t1,a}
c := t1 * b	{b,t1}
t2 := c + b	{b,c}
a := t2 + t2	{b,c,t2}
	{a,b,c}



After spilling b:

a := b + c {b,c}

t1 := a * a {a}

b := t1 + a {t1,a}

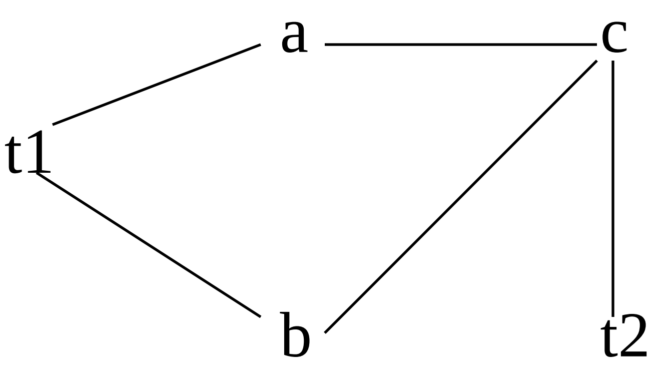
c := t1 * b {b,t1}

b to memory

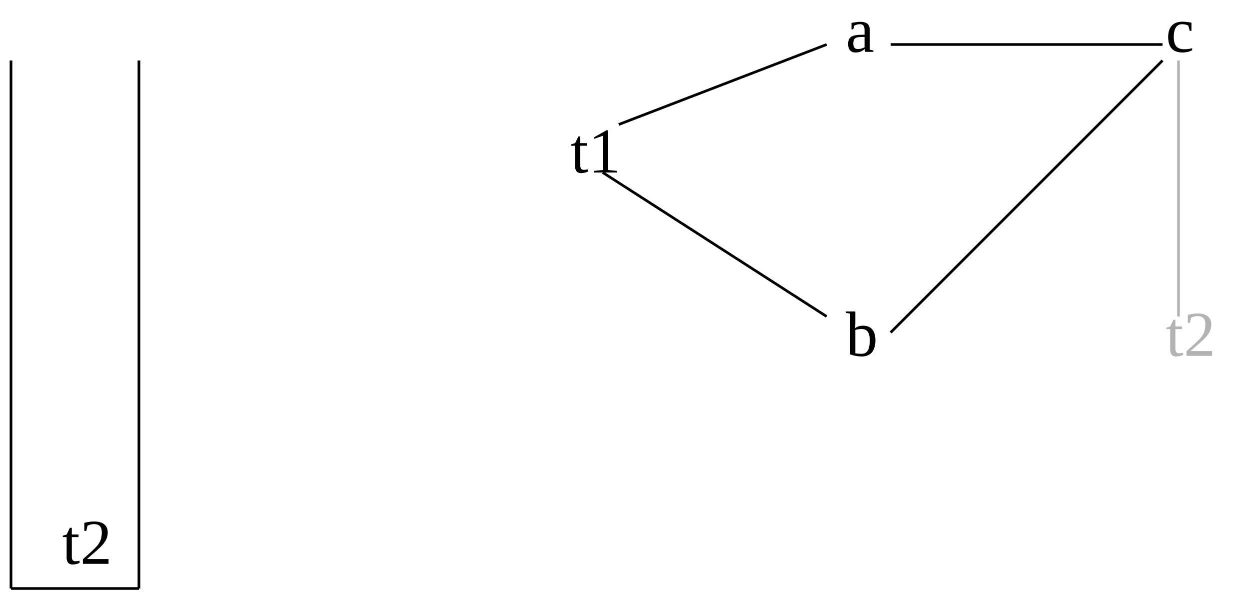
t2 := c + b {b,c}

a := t2 + t2 {c,t2}

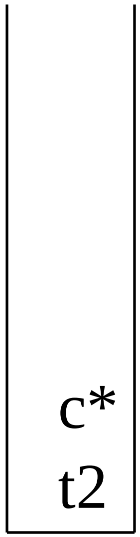
{a, c}

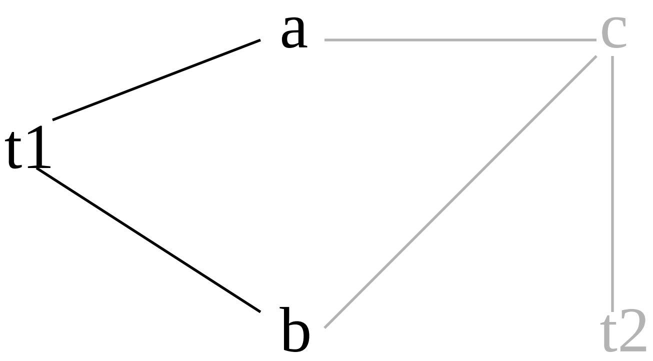


After spilling b:



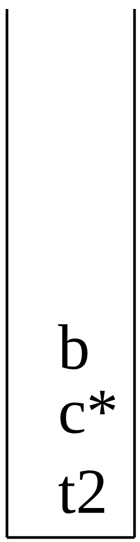
After spilling b:

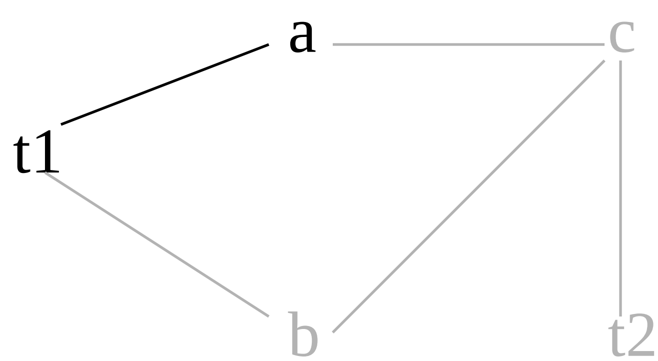




Have to choose c as a potential spill node.

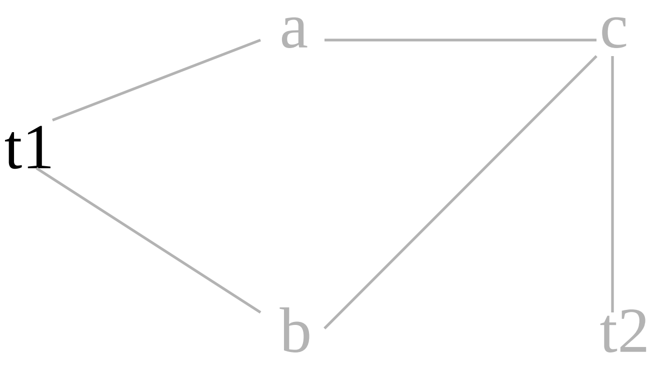
After spilling b:



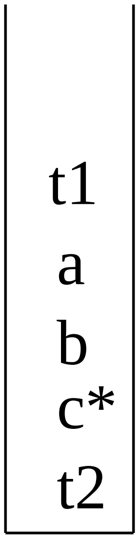


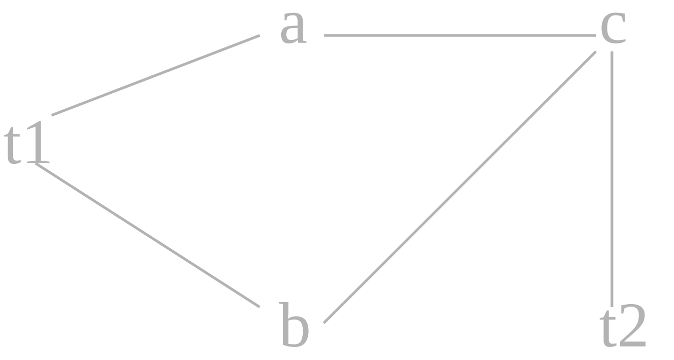
After spilling b:

a
b
c*
t2

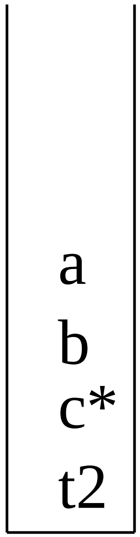


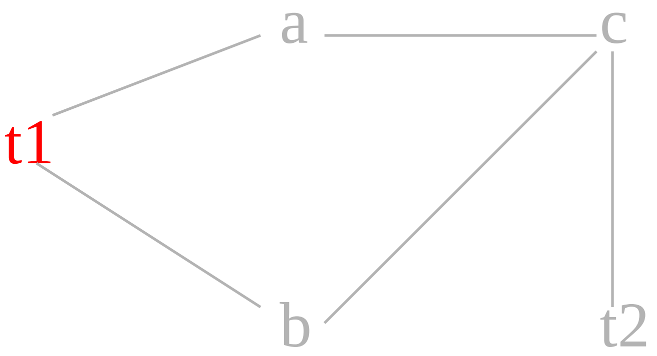
After spilling b:



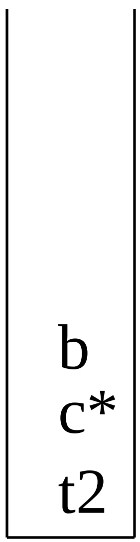


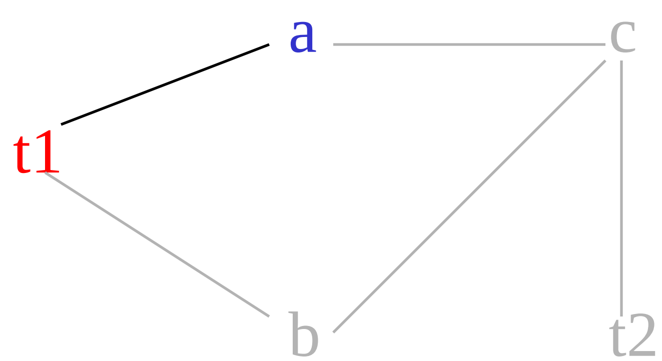
Now rebuild:



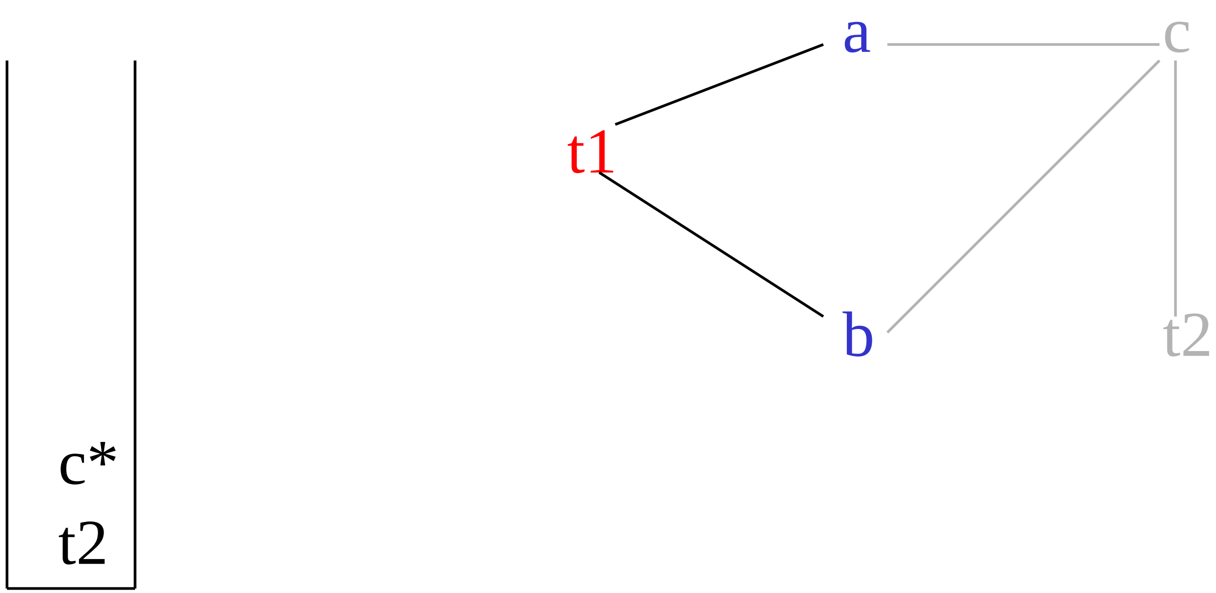


Now rebuild:

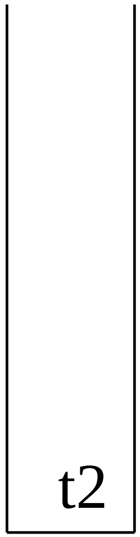


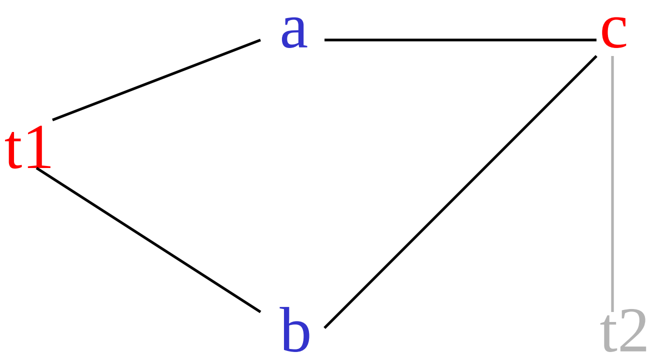


Now rebuild:



Now rebuild:

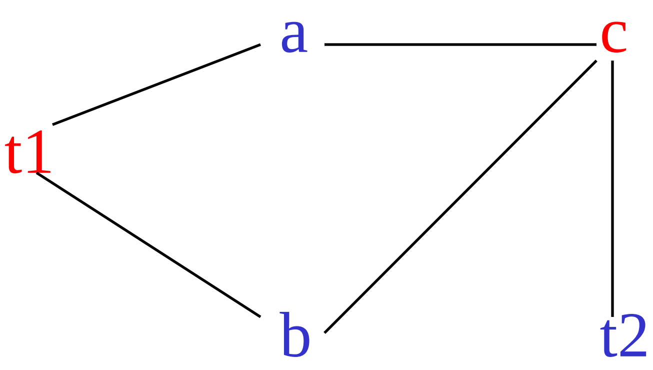




Fortunately, there is a color for c

Now rebuild:

a	t0
b	t0
c	t1
t1	t1
t2	t0



The graph is 2-colorable now

The code

a := b + c

t1 := a * a

b := t1 + a

c := t1 * b

b to memory

t2 := c + b

a := t2 + t2

a	t0
b	t0
c	t1
t1	t1
t2	t0

```
lw $t0,b
lw $t1,c
add $t0,$t0,$t1
mul $t1,$t0,$t0
add $t0,$t1,$t0
mul $t1,$t1,$t0
sw $t0,b
add $t0,$t1,$t0
add $t0,$t0,$t0
sw $t0,a
sw $t1,c
```

Spilling (more!)

- Spilling: once all nodes have K or more neighbors, pick a node for spilling
  - Storage on the stack
- There are many heuristics that can be used to pick a node
  - not in an inner loop

Spilling code

- We need to generate extra instructions to load variables from stack and store them
- These instructions use registers themselves What to do?
  - Stupid approach: always keep extra registers handy for shuffling data in and out: what a waste!
  - Better approach: ?

Spilling code

- We need to generate extra instructions to load variables from stack and store them
- These instructions use registers themselves. What to do?
  - Stupid approach: always keep extra registers handy for shuffling data in and out: what a waste!
  - Better approach: rewrite code introducing a new temporary; rerun liveness analysis and register allocation

Rewriting code

- Consider: add t1 t2
  - Suppose t2 is selected for spilling and assigned to stack location [ebp-24]
  - Invent new temporary t35 for just this instruction and rewrite:
    - mov t35, [ebp – 24]; add t1, t35
  - Advantage: t35 has a very short live range and is much less likely to interfere.
  - Rerun the algorithm; fewer variables will spill