CS536

Register Allocation & Instruction Selection

A Sahu

CSE, IIT Guwahati

Outline

- Register Allocation
- Spilling
- Spilling Example with RA
- Instruction Selection

Example Answers

```
                live =  {b,c}      ← what does this mean???
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}
```

Register and Address Descriptors

- A ***register descriptor*** keeps track of what is currently stored in a register at a particular point in the code, e.g. a local variable, argument, global variable, etc.

  `MOV a,R0`   "**`R0`** contains **`a`**"

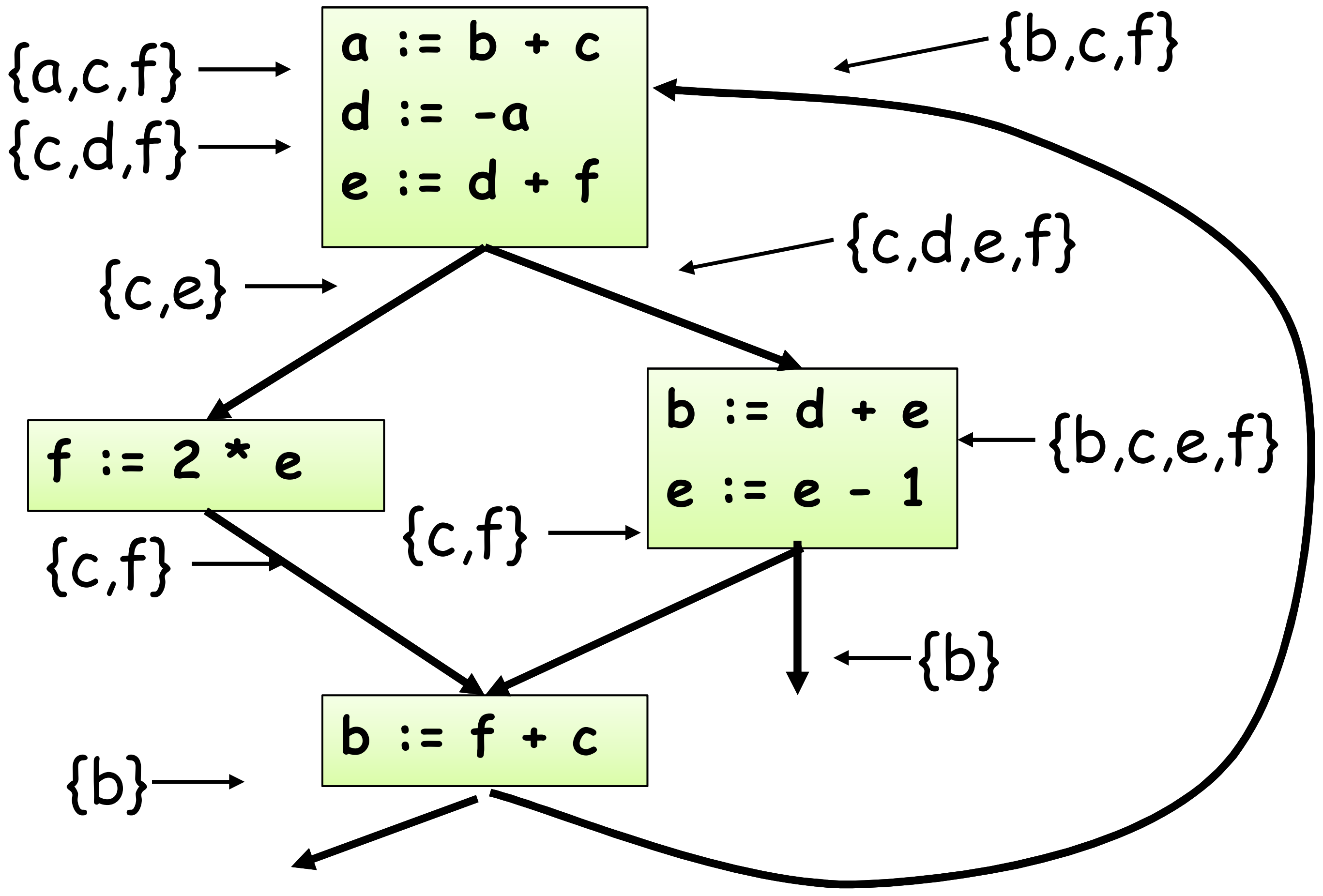
- An ***address descriptor*** keeps track of the location where the current value of the name can be found at run time, e.g. a register, stack location, memory address, etc.

  `MOV a,R0`

  `MOV R0,R1`   "**`a`** in **`R0`** and **`R1`**"

Algorithm: Part I

- Compute live variables for each point:



Spilling

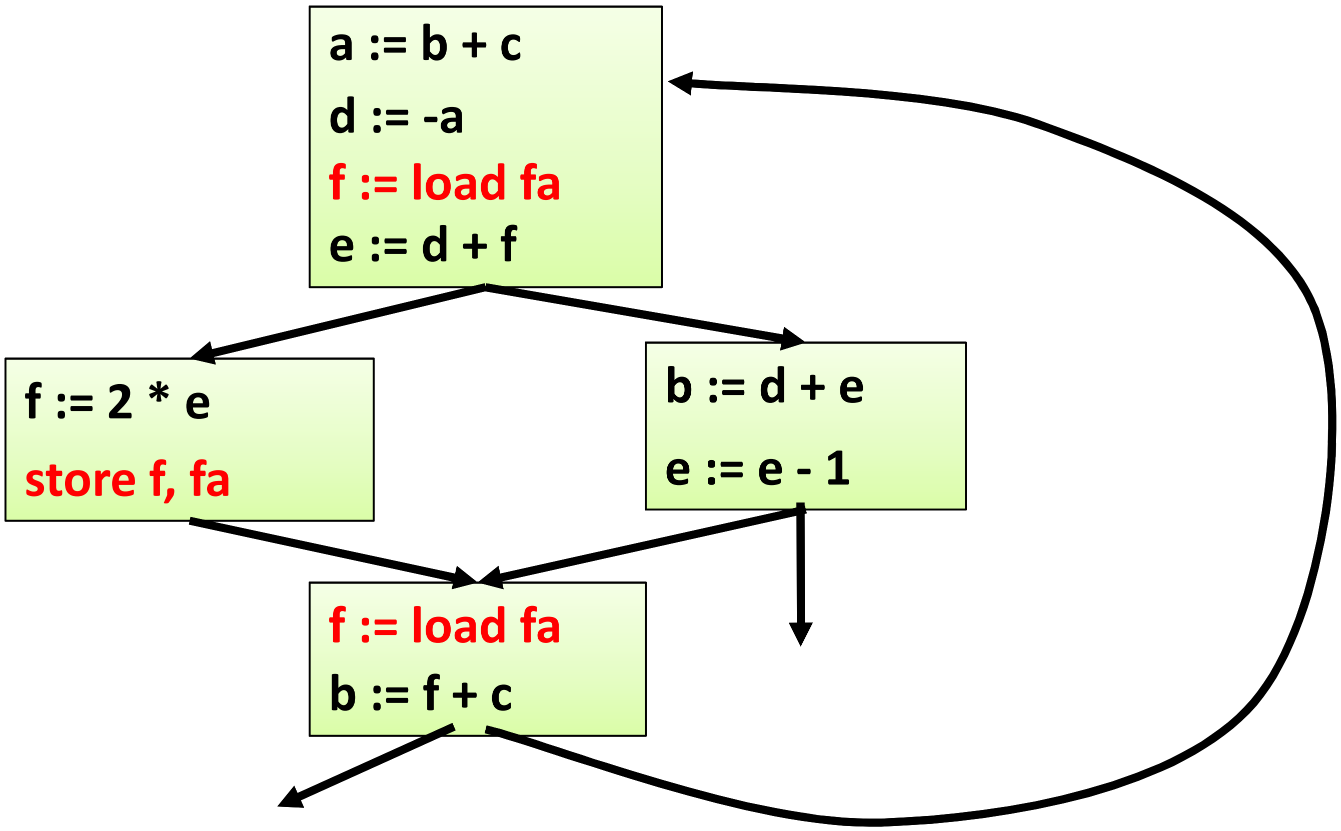
- Since optimistic coloring failed we must spill variable f
- We must allocate a memory location as the home of f
  - Typically this is in the current stack frame
  - Call this address **fa**
- Before each operation that uses **f**, insert

  f := load fa
- After each operation that defines **f**, insert

  store f, fa

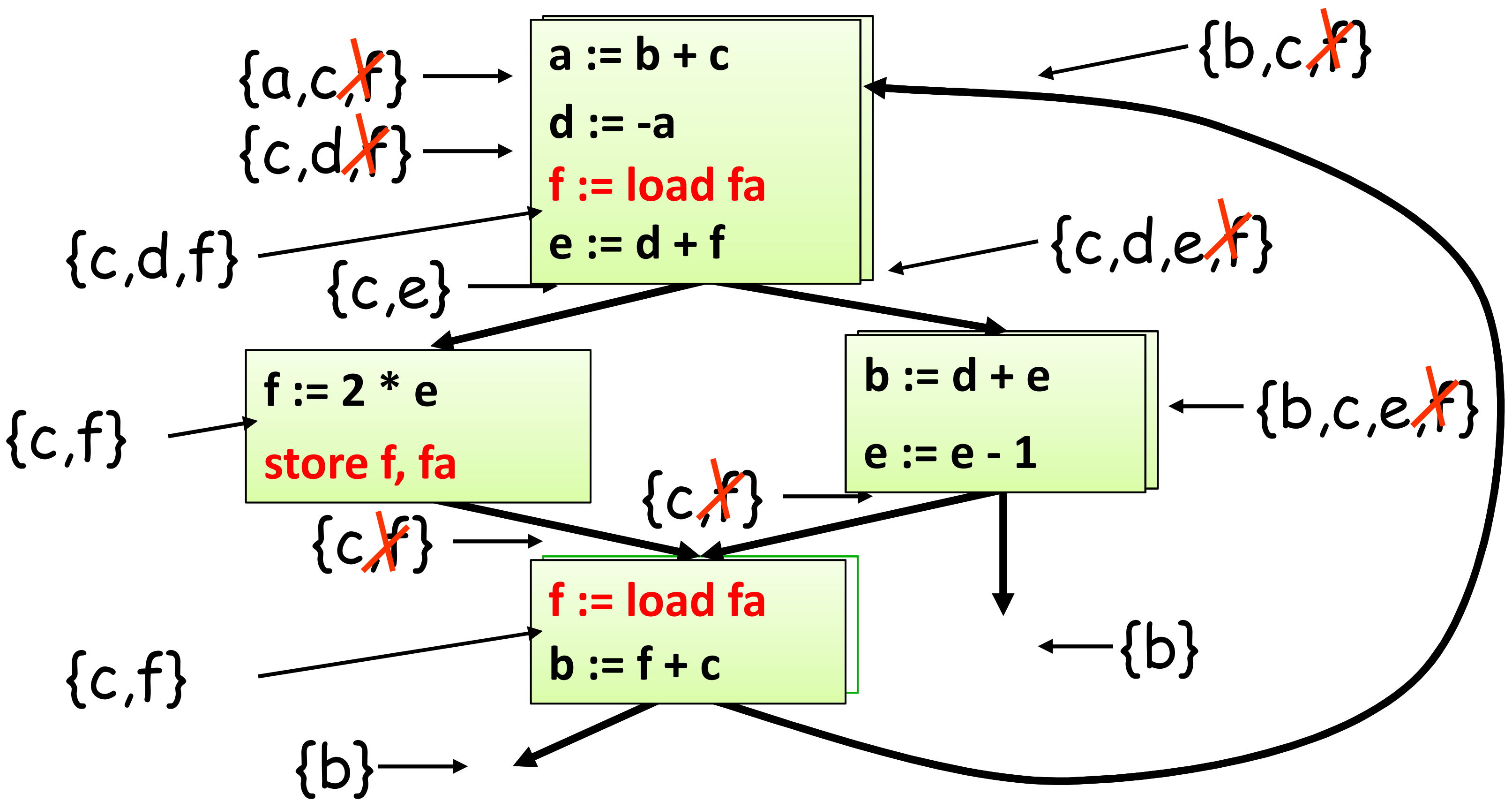
Spilling. Example.

- This is the new code after spilling **f**



Recomputing Liveness Information

- The new liveness information after spilling:



Global Register Allocation Using Graph Coloring

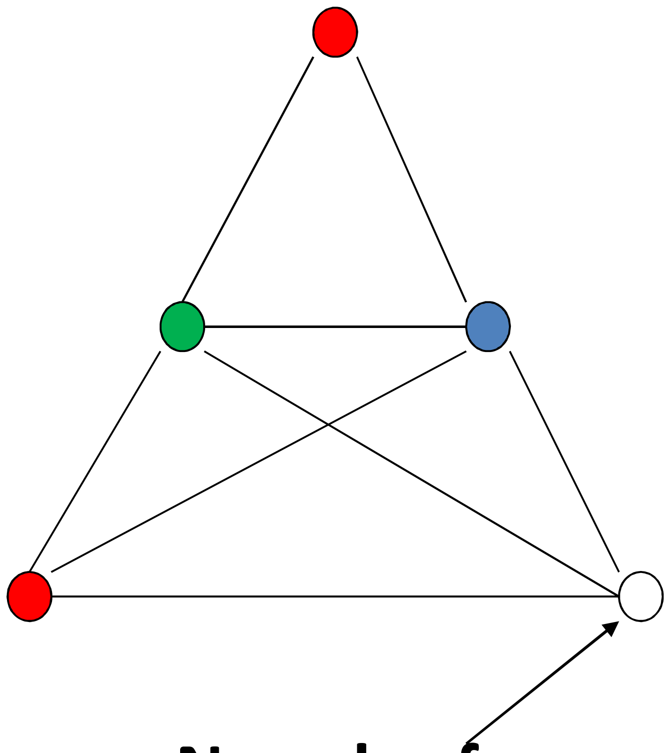
- When a register is needed but all available registers are in use, the content of one of the used registers **must be stored (spilled)** to free a register
- Graph coloring allocates registers and attempts to minimize the cost of spills
- Build a *conflict graph* (*interference graph*)
- Find a *k*-coloring for the graph, with *k* the number of registers

Graph Coloring

- The **vertex/default** 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).**
- Problem is NP (for more than 2 colors) ➔ no polynomial time solution.
- Fortunately there are approximation algorithms.
- Greedy Algo:
  - Order vertices in any order,
  - choose color for v_i with smallest available color not used by v_i neighbors among v_1 to v_{i-1}, otherwise choose a new color for v_i

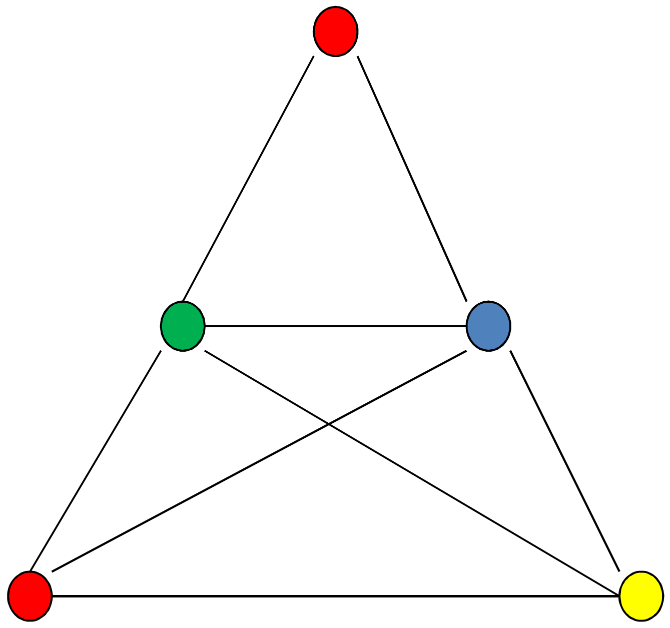
Coloring a graph with K colors

K = 3



No color for this node

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

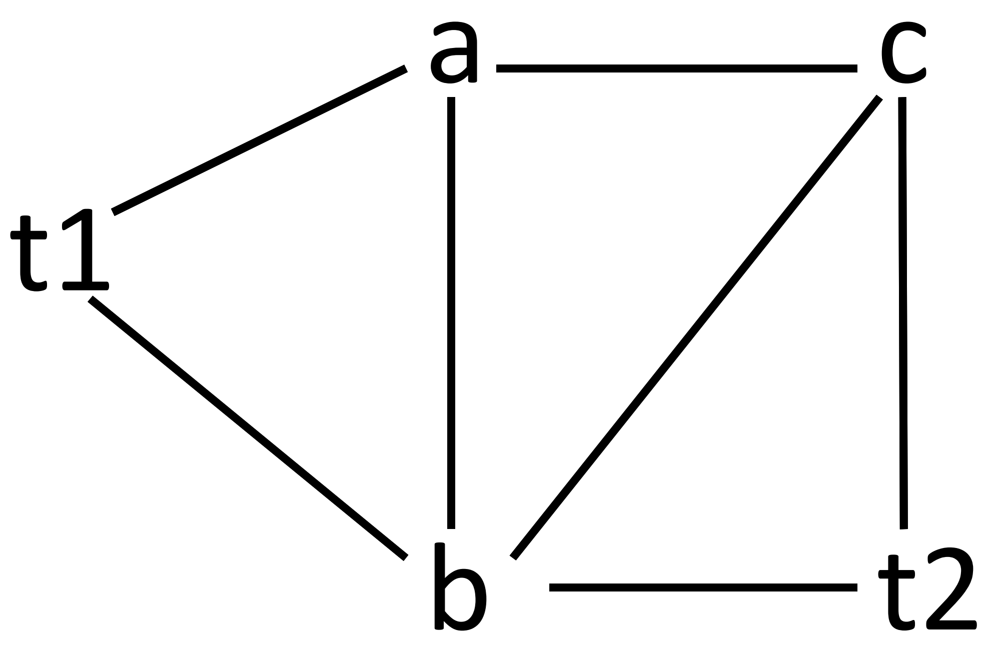
G is an ‘*interference graph*’

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, remove it from the graph and push it onto a stack. If G - m can be colored with K colors, so can G. If we reduce the entire graph, goto step 5.
4. **Spill** - If we get to the point where we are left with only nodes with degree >= K, mark some node for potential spilling. Remove and push onto stack. 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** - Starting with empty graph, rebuild graph by popping elements off the stack, putting them back into the graph and assigning them colors different from neighbors. Potential spill nodes may or may not be colorable.

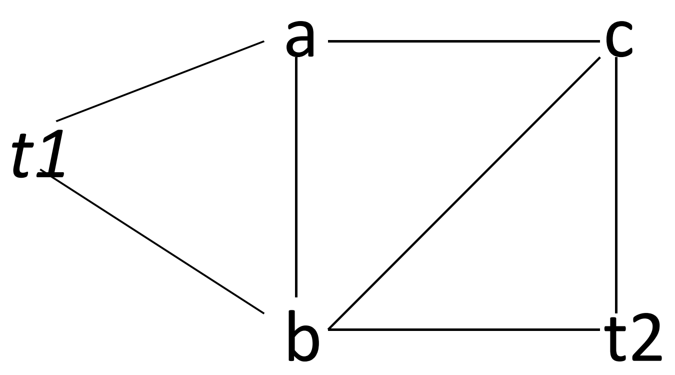
Process may require iterations and rewriting of some of the code to create more temporaries.

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

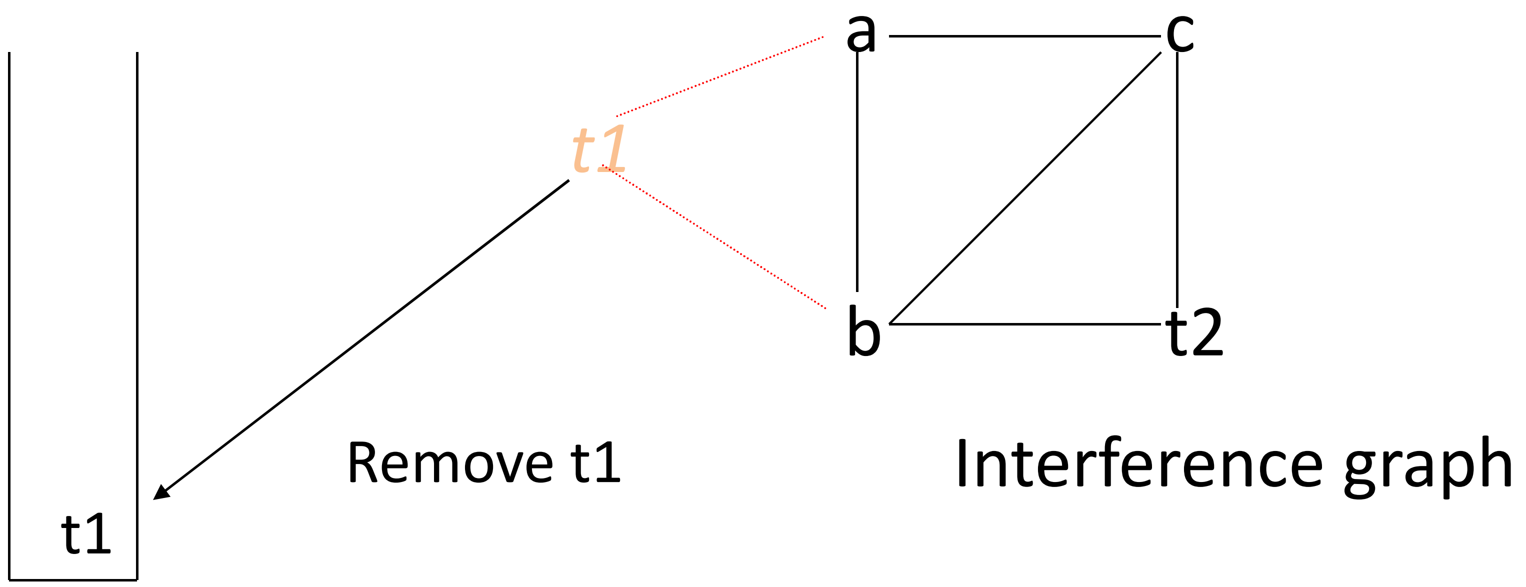
a := b + c	{b,c}
t1 := a * a	{a}
b := t1 + a	{t1,a}
c := t1 * b	{b,t1}
t2 := c + b	{b,c}
a := t2 + t2	{b,c,t2}
	{a,b,c}



Interference graph

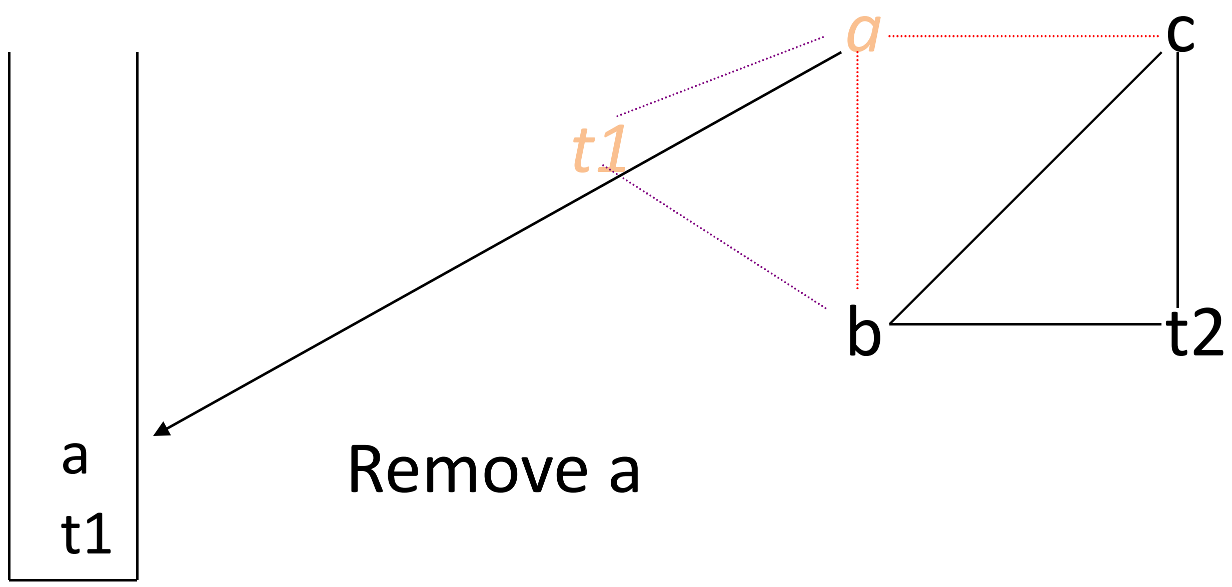
Example, k = 3

Assume k = 3



Example

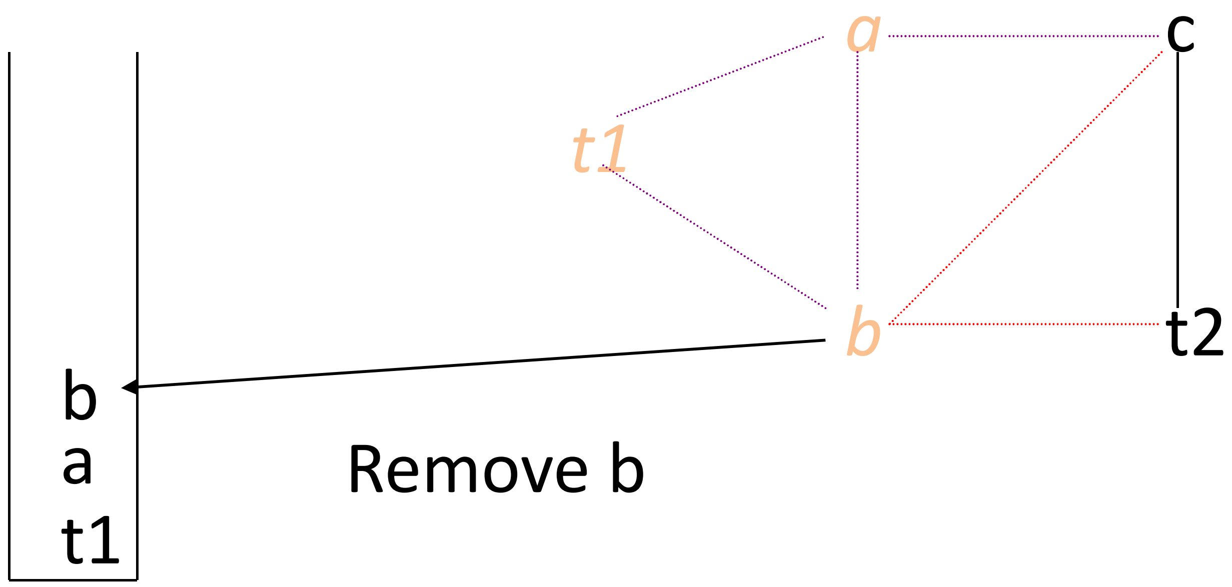
Assume k = 3



Remove a

Example

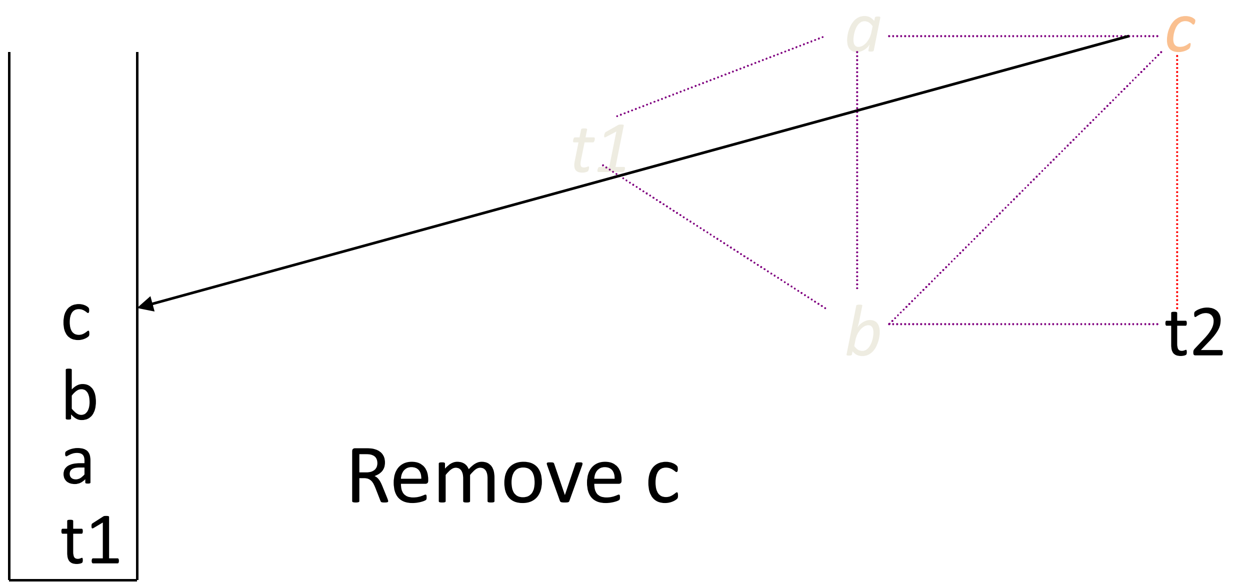
Assume k = 3



Remove b

Example

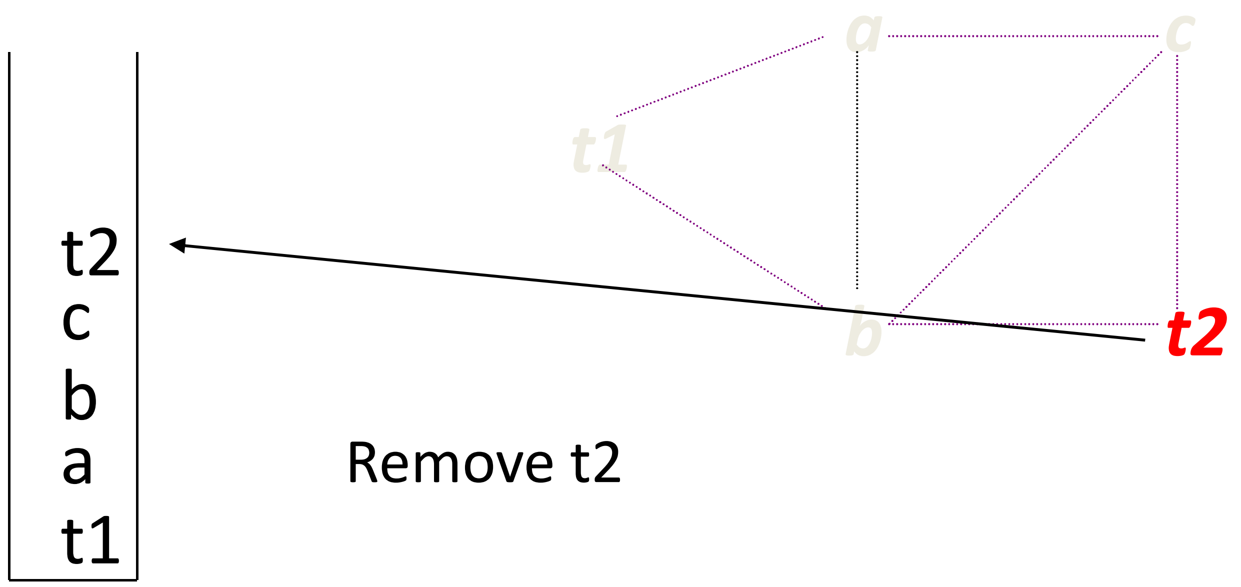
Assume k = 3



Remove c

Example

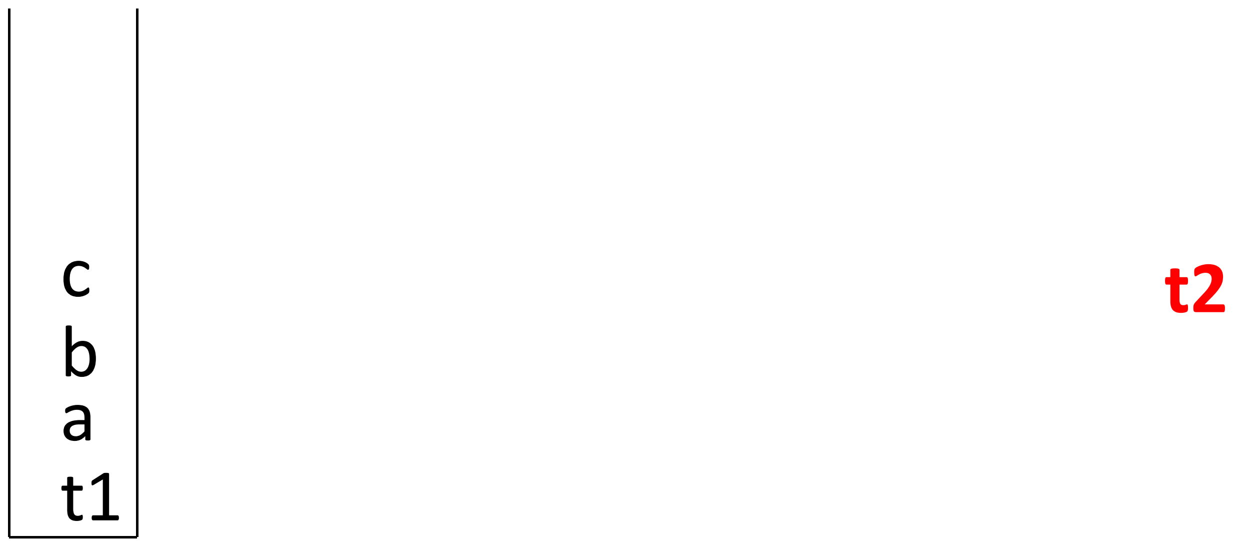
Assume k = 3



Remove t2

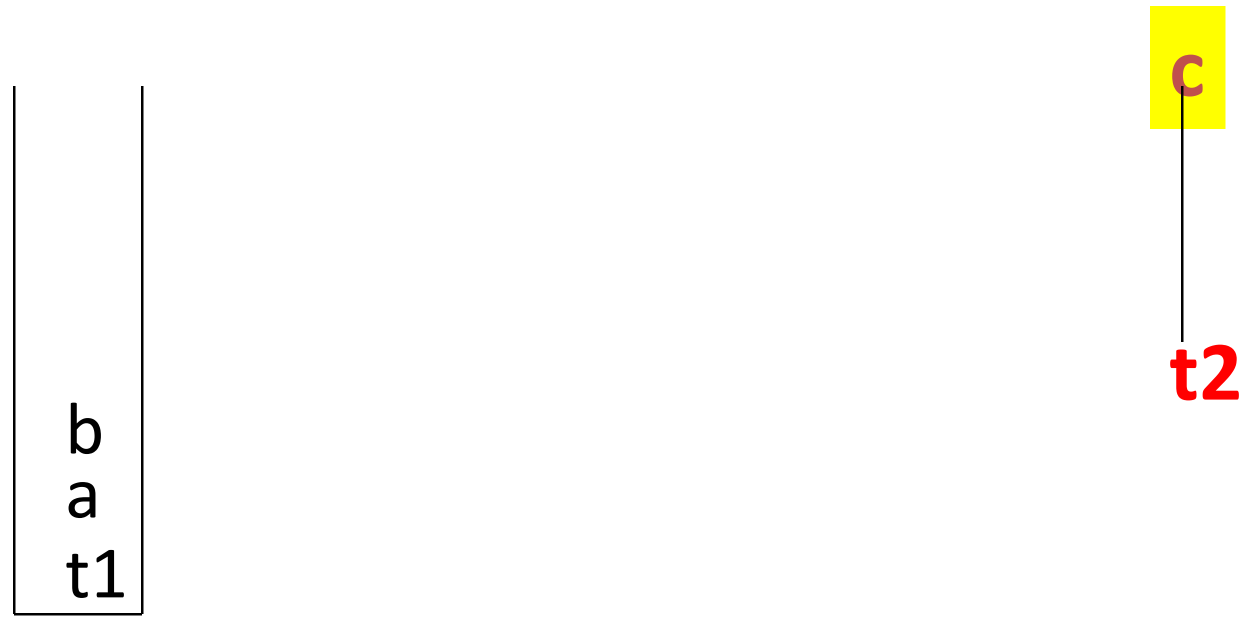
Rebuild the graph

Assume k = 3



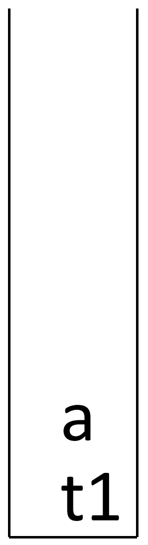
Example

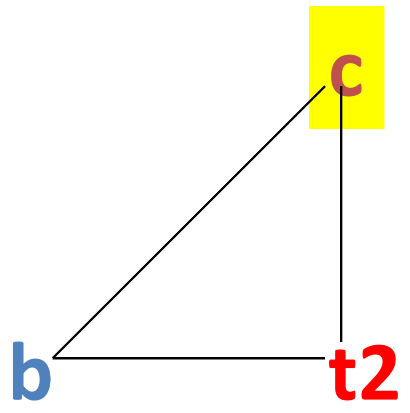
Assume k = 3



Example

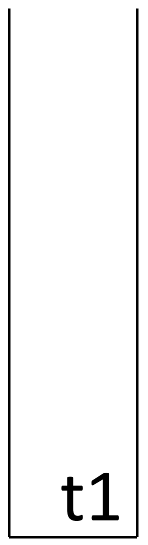
Assume k = 3

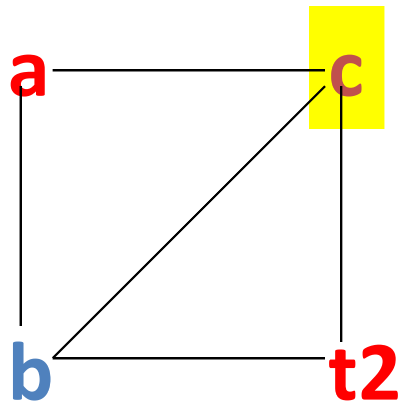




Example

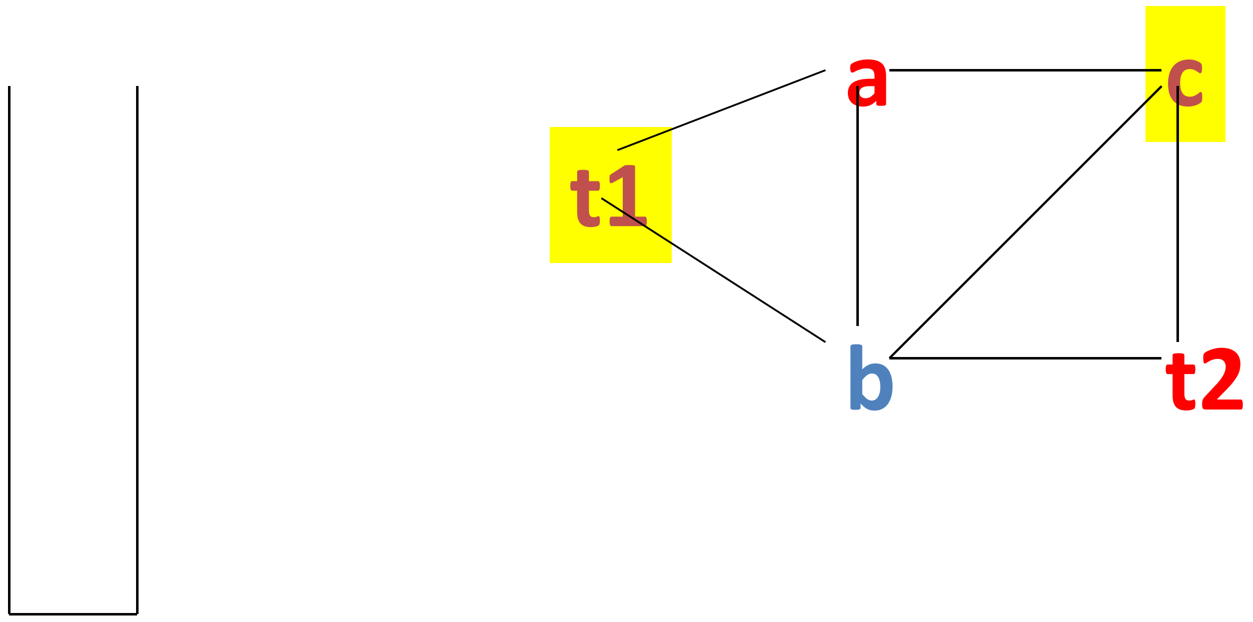
Assume k = 3





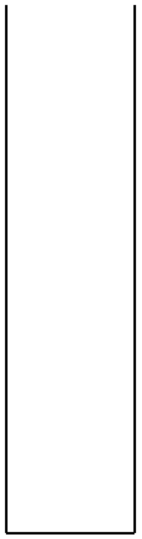
Example

Assume k = 3

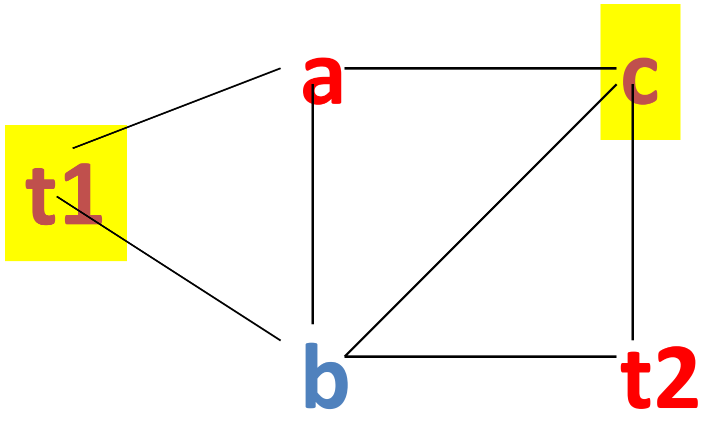


Example

Assume k = 3



a	R0
b	R1
c	R2
t1	R2
t2	R0



Back to example

```
a := b + c
t1 := a * a
b := t1 + a
c := t1 * b
t2 := c + b
a := t2 + t2
```

a	R0
b	R1
c	R2
t1	R2
t2	R0

```
lw $r1,b
lw $r2,c
add $r0,$r1,$r2
mul $r2,$r0,$r0
add $r1,$r2,$r0
mul $r2,$r2,$r1
add $r0,$r2,$r1
add $r0,$r0,$r0
sw $r0,a
sw $r1,b
sw $r2,c
```

Back to example

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
```

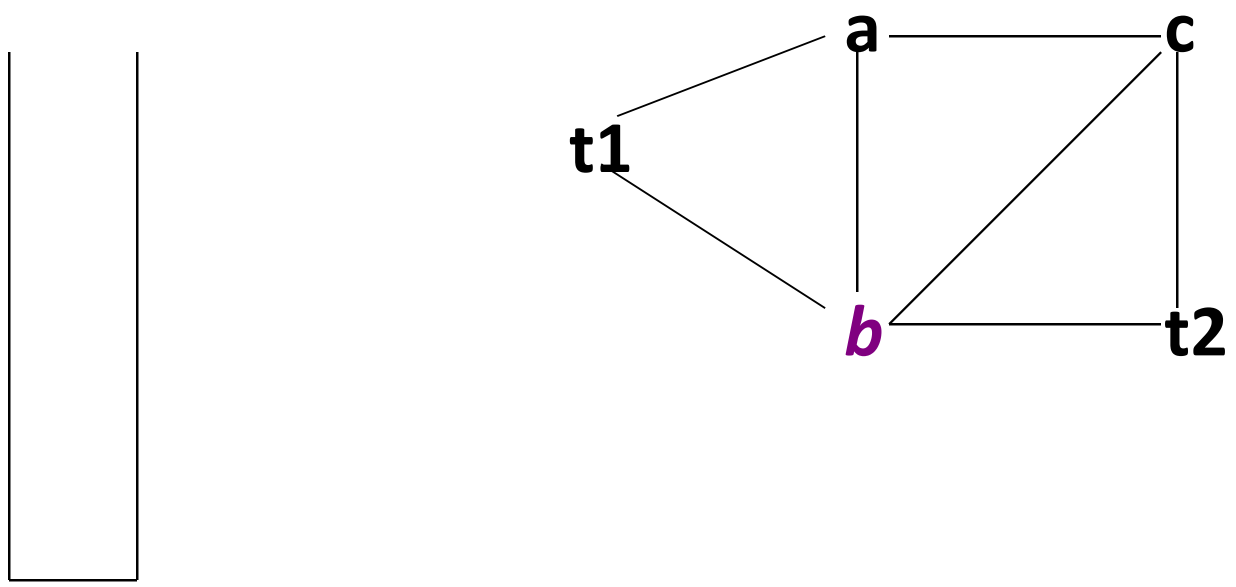
Generated Code: Coloring

```
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
```

R0, R1, R2 mapped to t0, t1, t2 respectively of earlier page: same meaning

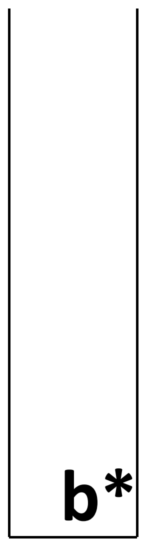
Example, k = 2

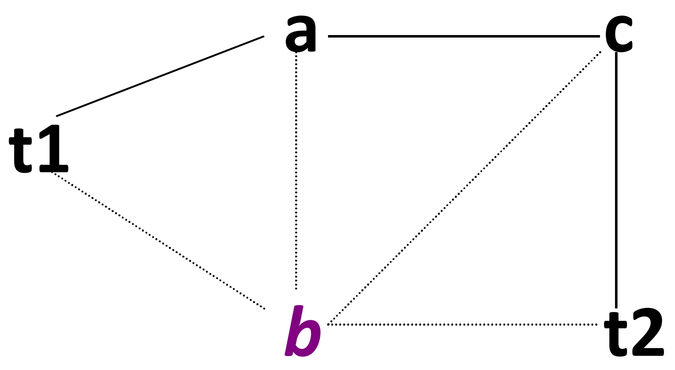
Assume k = 2



Example, k = 2

Assume k = 2

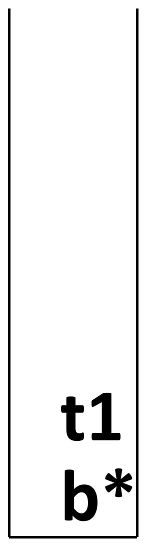


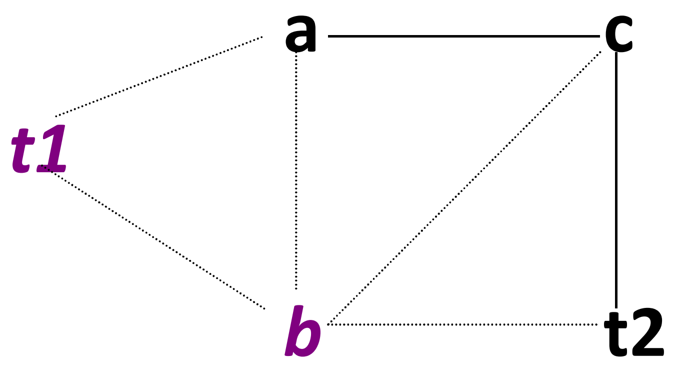


Remove b as spill

Example

Assume k = 2

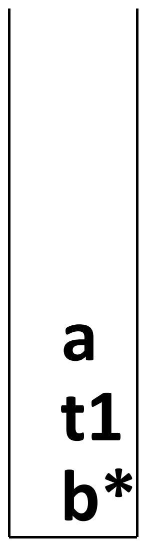




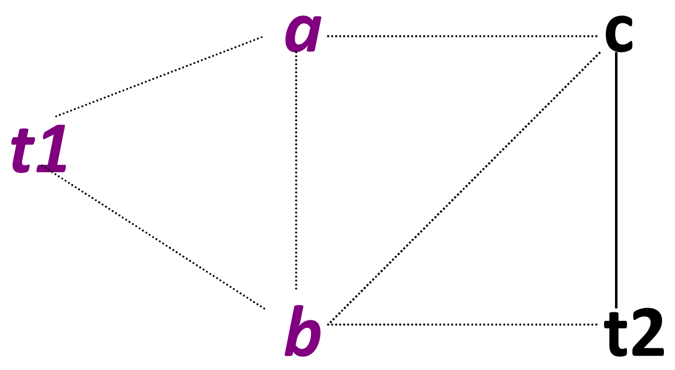
Remove t1

Example

Assume k = 2

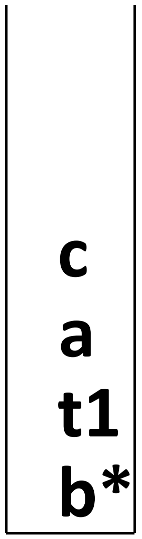


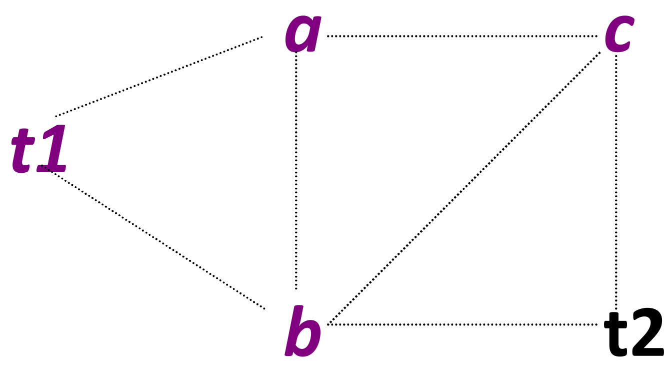
Remove a



Example

Assume k = 2

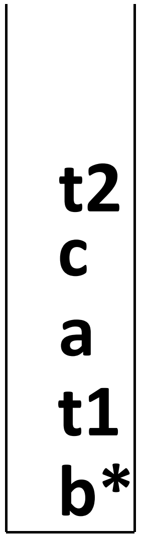


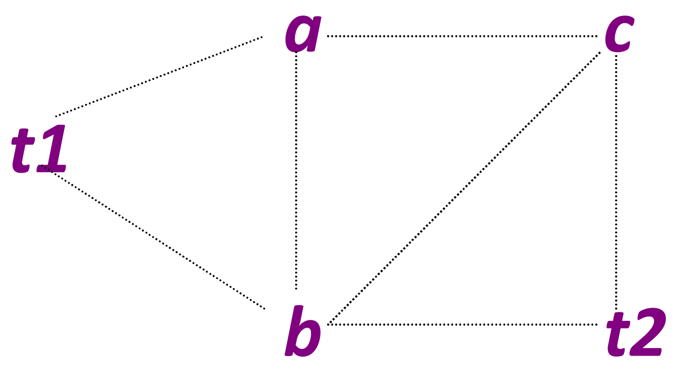


Remove c

Example

Assume k = 2

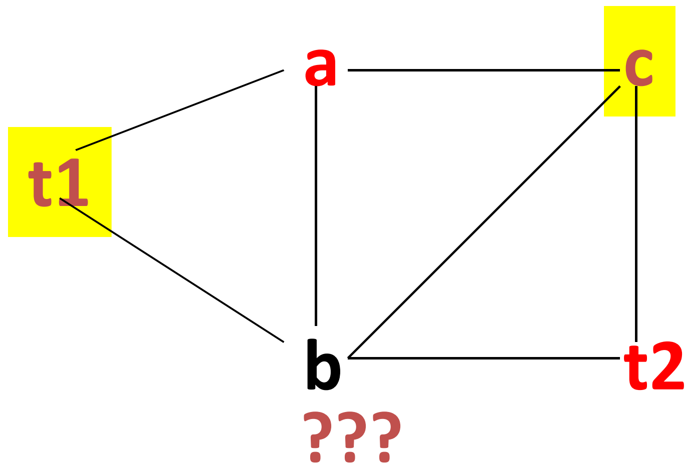




Remove t2

Example

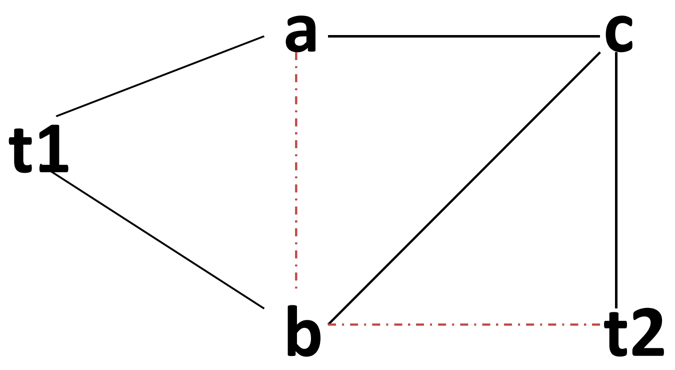
Assume k = 2



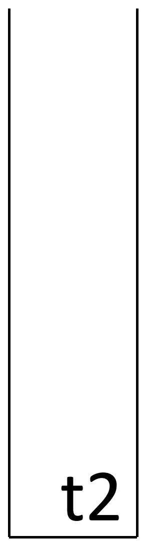
Can flush b out to memory, creating a smaller window

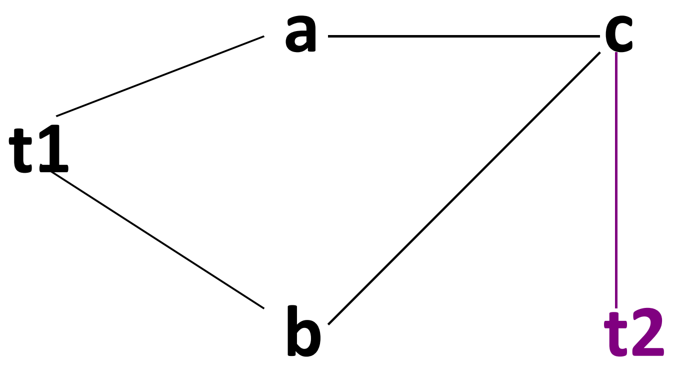
After spilling b:

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	{~~b~~, c,t2}
	{a, c, ~~b~~}

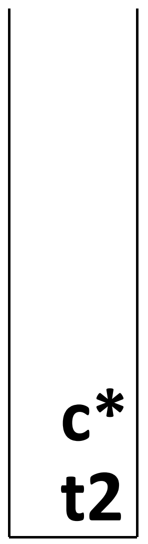


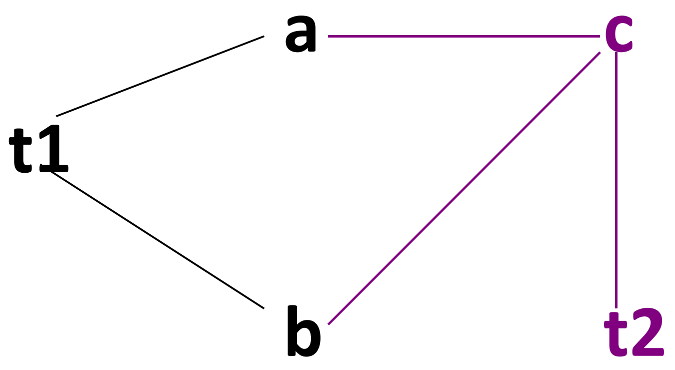
After spilling b:





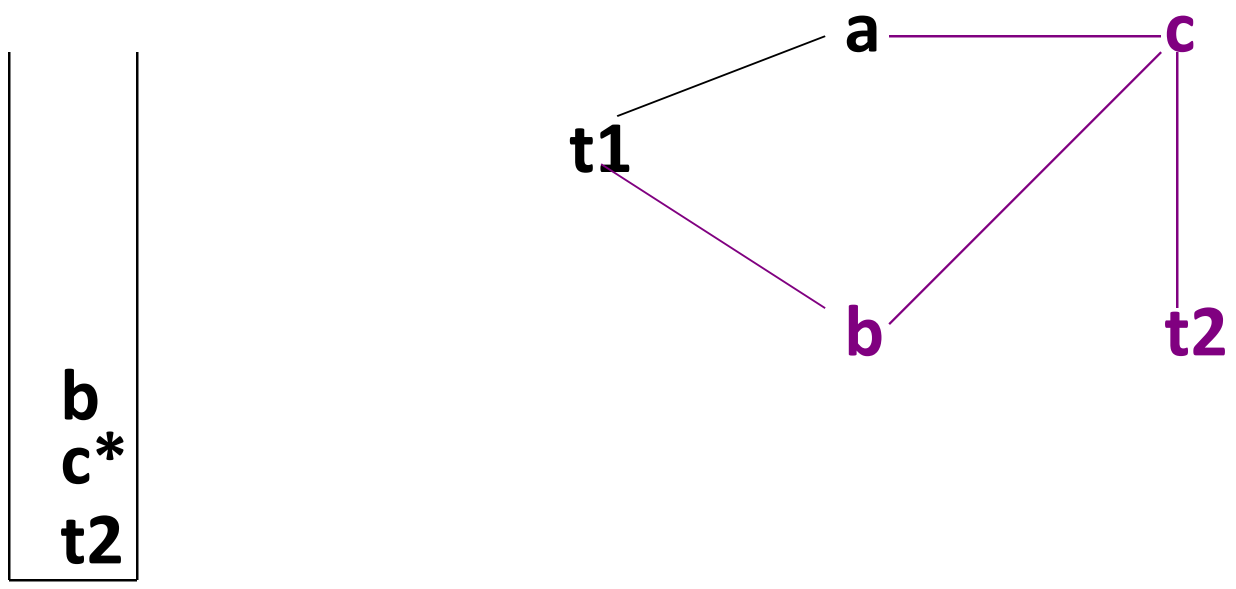
After spilling b:



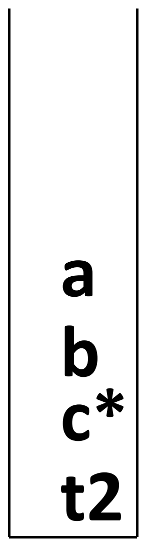


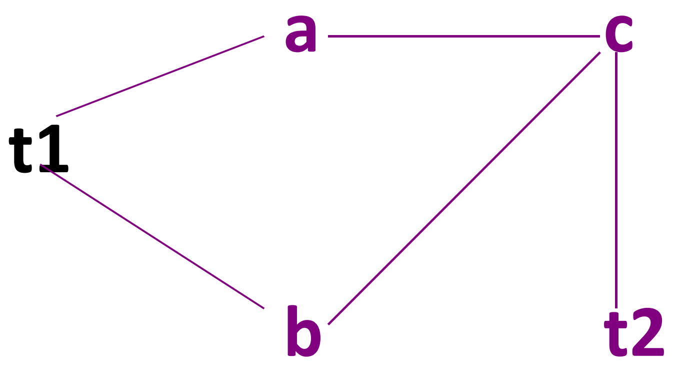
Have to choose c as a potential spill node.

After spilling b:

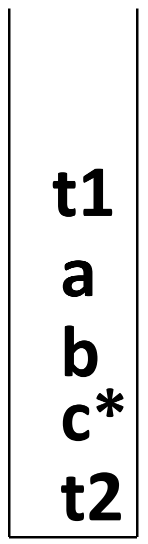


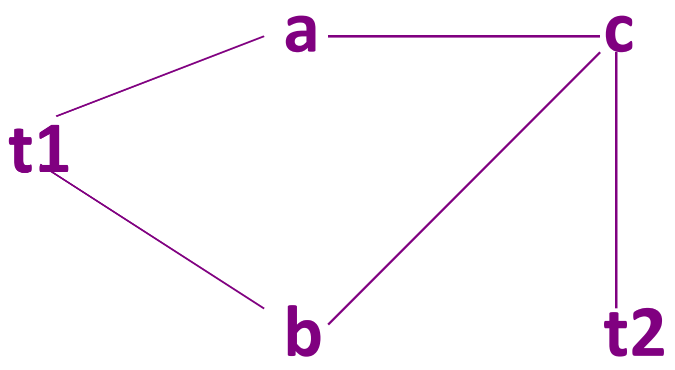
After spilling b:



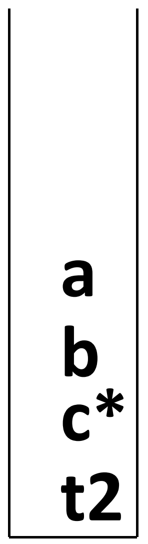


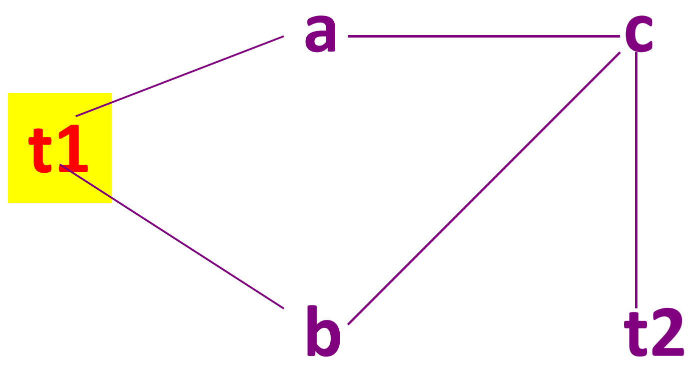
After spilling b:



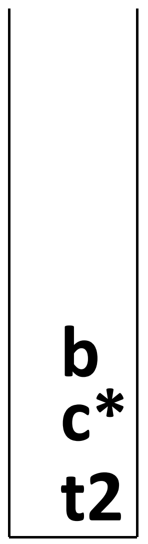


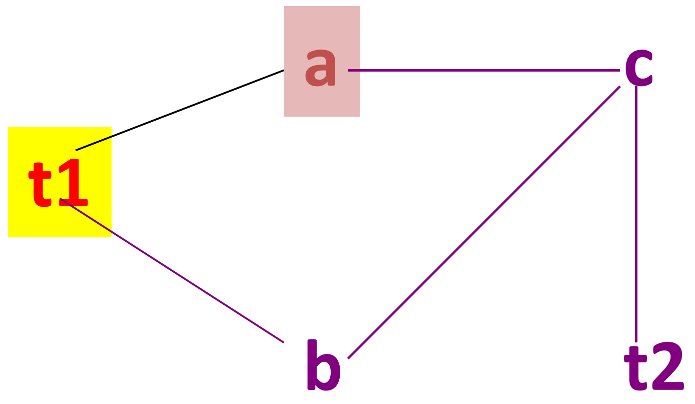
Now rebuild:



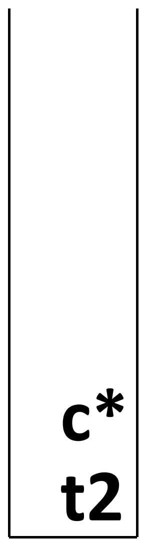


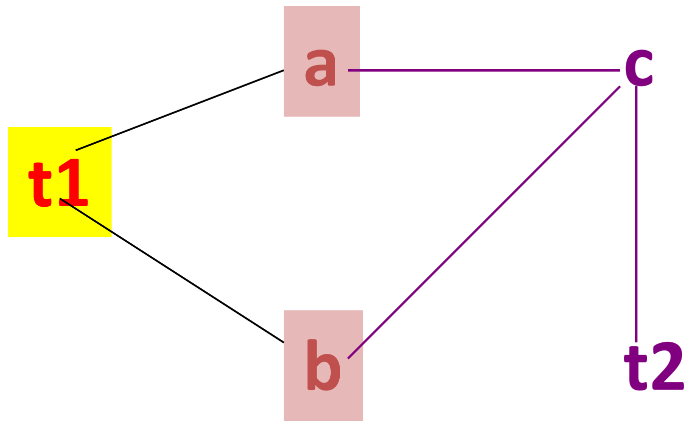
Now rebuild:



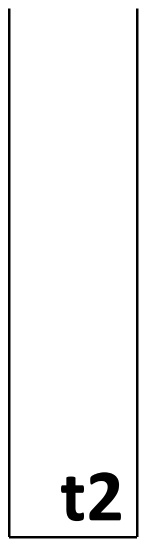


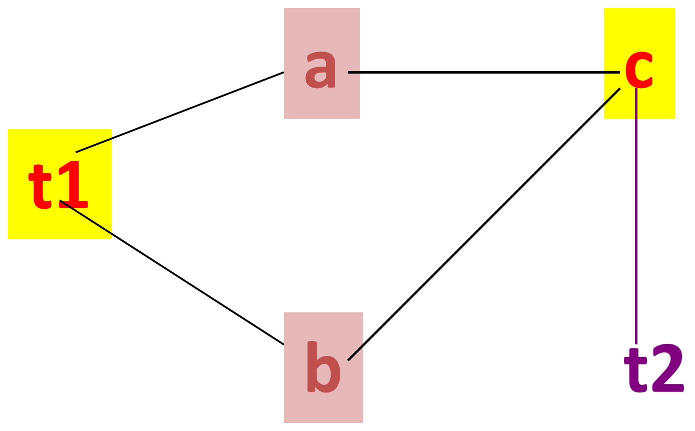
Now rebuild:





Now rebuild:

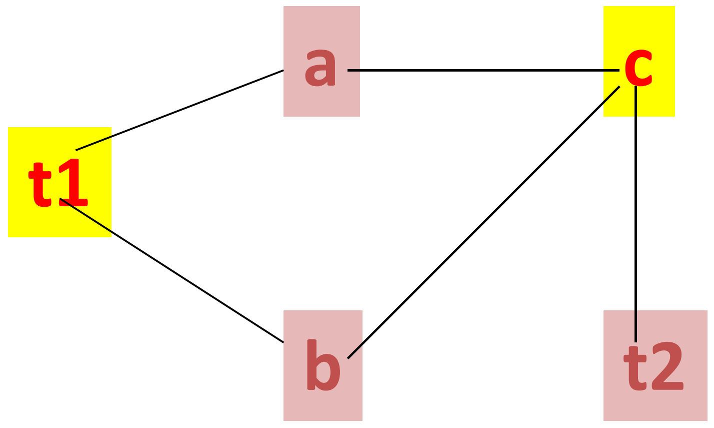




Fortunately, there is a color for c

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
```

Three Basic Back-End Optimization

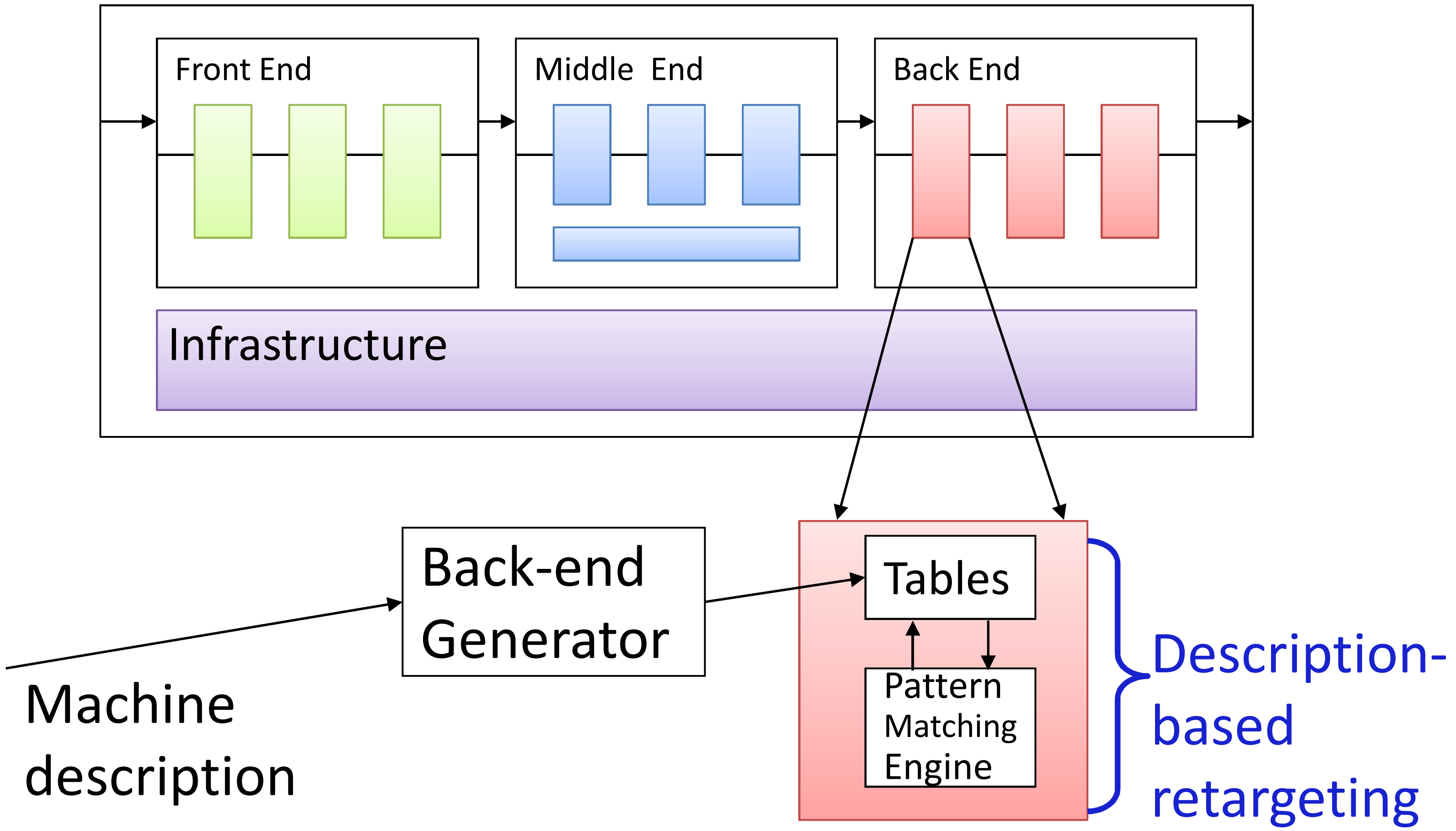
- Instruction selection
  - Mapping IR into assembly code
  - Assume a fixed storage mapping & code shape
  - Combining operations, using address modes
- Instruction scheduling
  - Reordering operations to hide latencies
  - Assume a fixed program (set of operations)
  - Changes demand for registers
- Register allocation
  - Deciding which values will reside in registers
  - Changes the storage mapping may add false sharing
  - Concerns about placement of data & memory operations

Instruction Selection

Objectives

- Introduce the complexity and importance of instruction selection
- Study practical issues and solutions

Instruction Selection: Retargetable



Machine description should also help with scheduling & allocation

This is simplistic but useful view

Processor Types

- CISC: X86, X86-64 : **3000 instructions types**
- RISC: 80 INS; simple addressing mode
  - Similar to compiler LLIR
  - MIPS, ARM, PowerPC
- Application Specific Instruction Processor (ASIP) or Domain Specific Inst. Processor
  - **Digital Signal Processor / MAC Ins**
  - **Image Processor**
  - **Network Processor //Match+Action, P4 Language**
  - **Crypto Processor**
  - **Media Processor (MMX)**
  - **Vector/Matrix Processor (Google TPU)**
  - **GPU**
  - **Tensor Core**

Complexity of Instruction Selection

Modern computers have many ways to do things.

Consider a register-to-register copy

- Obvious operation is: mv r_j, r_i
- Many others exist
  - add rj, ri,0 sub rj, ri, 0 rshiftI rj, ri, 0

    mul rj, ri, 1 or rj, ri, 0 divI rj, r, 1

    xor rj, ri, 0 others …

Complexity of Instruction Selection (Cont.)

- Multiple addressing modes
- Each alternate sequence has its cost
  - Complex ops (mult, div): several cycles
  - Memory ops: latency vary
- Sometimes, cost is context related
- Use under-utilized FUs
- Dependent on objectives: speed, power, code size

Complexity of Instruction Selection (Cont.)

- **Additional constraints on specific operations**
  - Load/store multiple words: contiguous registers
  - Multiply: need special register Accumulator
- **Interaction between instruction selection, instruction scheduling, and register allocation**
  - For scheduling, instruction selection predetermines latencies and function units
  - For register allocation, instruction selection pre-colors some variables. e.g. non-uniform registers (such as registers for multiplication)

Instruction Selection Techniques

- **Tree Pattern-Matching**
  - Tree-oriented IR suggests pattern matching on trees
  - Tree-patterns as input, matcher as output
  - Each pattern maps to a target-machine instruction sequence
  - Use dynamic programming or bottom-up rewrite systems
- **Peephole-based Matching**
  - Linear IR suggests using some sort of string matching
  - Inspired by peephole optimization
  - Strings as input, matcher as output
  - Each string maps to a target-machine instruction sequence
- **In practice, both work well; matchers are quite different.**

Thanks