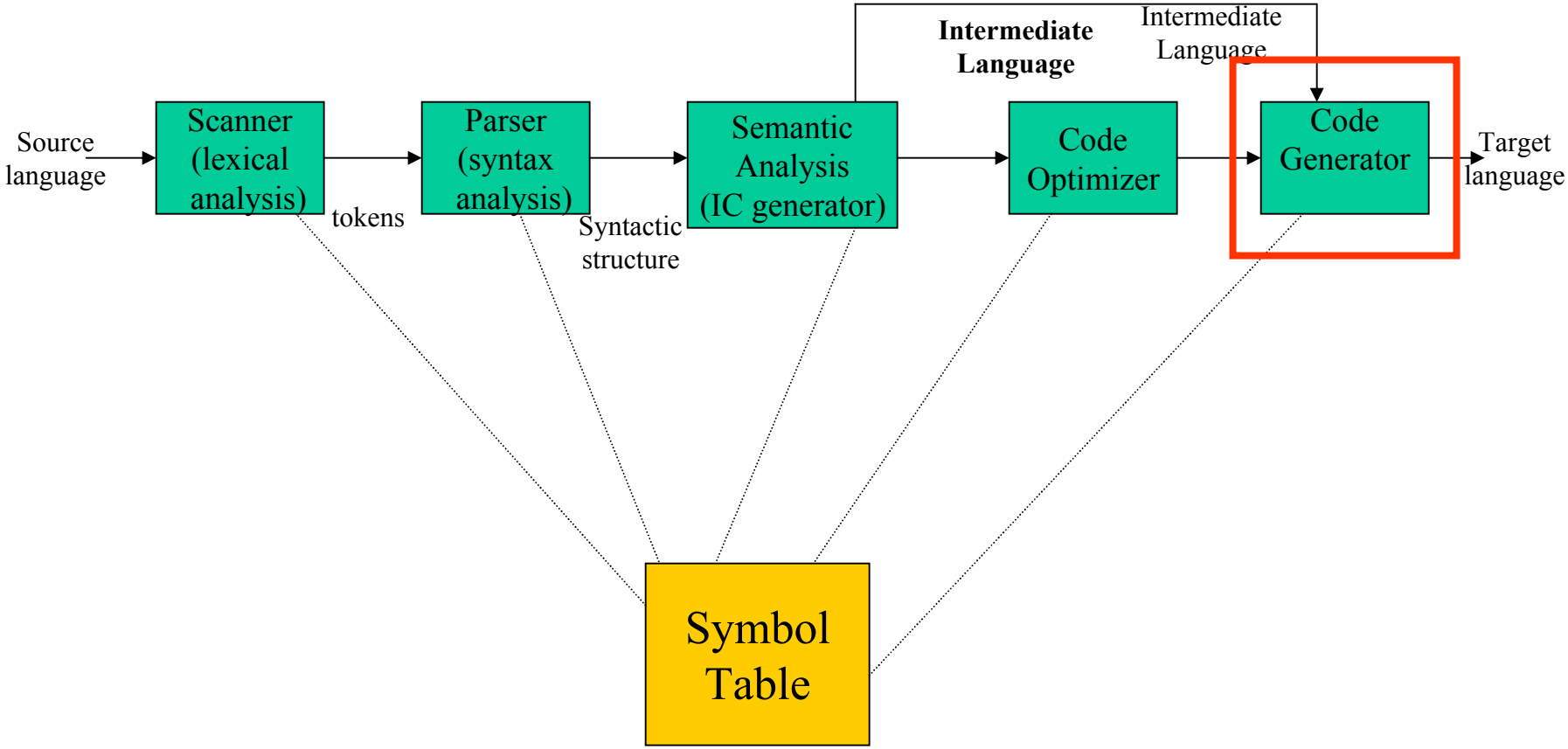
Code Generation

CS 540

George Mason University

Compiler Architecture



Code Generation

The code generation problem is the task of mapping intermediate code to machine code.

Machine Dependent Optimization!

Requirements:

- Correctness
- Efficiency

Issues:

- Input language: intermediate code (optimized or not)
- Target architecture: must be **well** understood
- Interplay between
  - Instruction Selection
  - Register Allocation
  - Instruction Scheduling

Instruction Selection

- There may be a large number of ‘candidate’ machine instructions for a given IC instruction
  - each has own cost and constraints
  - cost may be influenced by surrounding context
  - different architectures have different needs that must be considered: speed, power constraints, space …

Instruction Scheduling

- Choosing the order of instructions to best utilize resources
- Architecture
  - RISC (pipeline)
  - Vector processing
  - Superscalar and VLIW
- Memory hierarchy
  - Ordering to decrease memory fetching
  - Latency tolerance – doing something when data does have to be fetched

Register Allocation

How to best use the bounded number of registers.

Complications:

- special purpose registers
- operators requiring multiple registers.

Naïve Approach to Code Generation

Simple code generation algorithm:

- Define a target code sequence to each intermediate code statement type.

This is basically what we did earlier when creating the intermediate code (i.e. SPIM in project 3)

Why is this not sufficient?

Example Target: SPIM Assembly Language

General Characteristics

- Byte-addressable with 4-byte words
- N general -purpose registers
- Three-address instructions: op destination, source1, source2

Mapping from Intermediate code

Simple code generation algorithm:

Define a target code sequence to each intermediate code statement type.

Intermediate	becomes…	Intermediate	becomes…

a := b	lw $t0,b
	sw $t0,a
		a := b + c	lw $t0,b
			lw $t1,c
			add $t0,$t0,$t1
			sw $t0,a

a := b[c]	la $t0,b
	lw $t1,c
	add $t0,$t0,$t1
	lw $t0,($t0)
	sw $t0,a
		a[b] := c	la $t0,a
			lw $t1,b
			add $t0,$t0,$t1
			lw $t1,c
			sw $t1,($t0)

Consider the C statement: a[i] = d[c[k]];

```
t1 := c[k]          la $t0,c
                    lw $t1,k
                    add $t0,$t0,$t1
                    lw $t0,($t0)
                    sw $t0,t1
t2 := d[t1]         la $t0,d
                    lw $t1,t1
                    add $t0,$t0,$t1
                    lw $t0,($t0)
                    sw $t0,t2
a[i] := t2          la $t0,a
                    lw $t1,i
                    add $t0,$t0,$t1
                    lw $t1,t2
                    sw $t1,($t0)
```

We use 15 instructions (12 load/store + 3 arithmetic) and allocate space for two temporaries (but only use two registers).

Problems with this approach

- Local decisions do not produce good code.
- Does not take temporary variables into account

Get rid of the temporaries (reduce load/store):

```
la $t0,c
lw $t1,k
add $t0,$t0,$t1   # address of c[k]
lw $t0,($t0)
la $t1,d
add $t1,$t1,$t0   # address of d[c[k]]
lw $t1,($t1)
la $t0,a
lw $t2,i
add $t0,$t0,$t2  # address of a[i]
sw $t1,($t0)
```

Need a way to generate machine code based on past and future use of the data.

- Analyze the code
- Use results of analysis

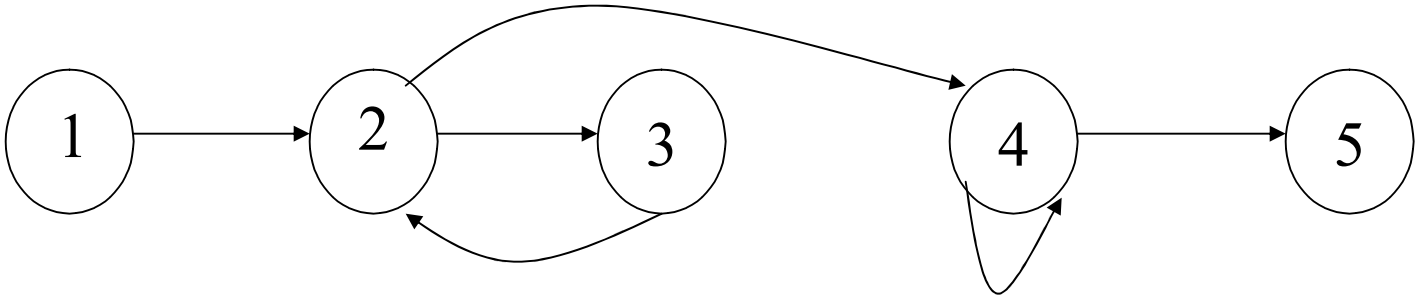
Representing Intermediate Code: Control Flow Graph - CFG

CFG = < V, E, Entry >, where

V = vertices or nodes, representing an instruction or basic block (group of statements).

E = (V x V) edges, potential flow of control

Entry is an element of V, the unique program entry



Basic Blocks

A basic block is a sequence of consecutive statements with single entry/single exit:

- flow of control only enters at the beginning
- Flow of control only leaves at the end
- Variants: single entry/multiple exit, multiple entry/single exit

Generating CFGs from Intermediate Code

- Partition intermediate code into basic blocks
- Add edges corresponding to control flow between blocks
  - Unconditional goto
  - Conditional goto – multiple edges
  - No goto at end – control passes to first statement of next block

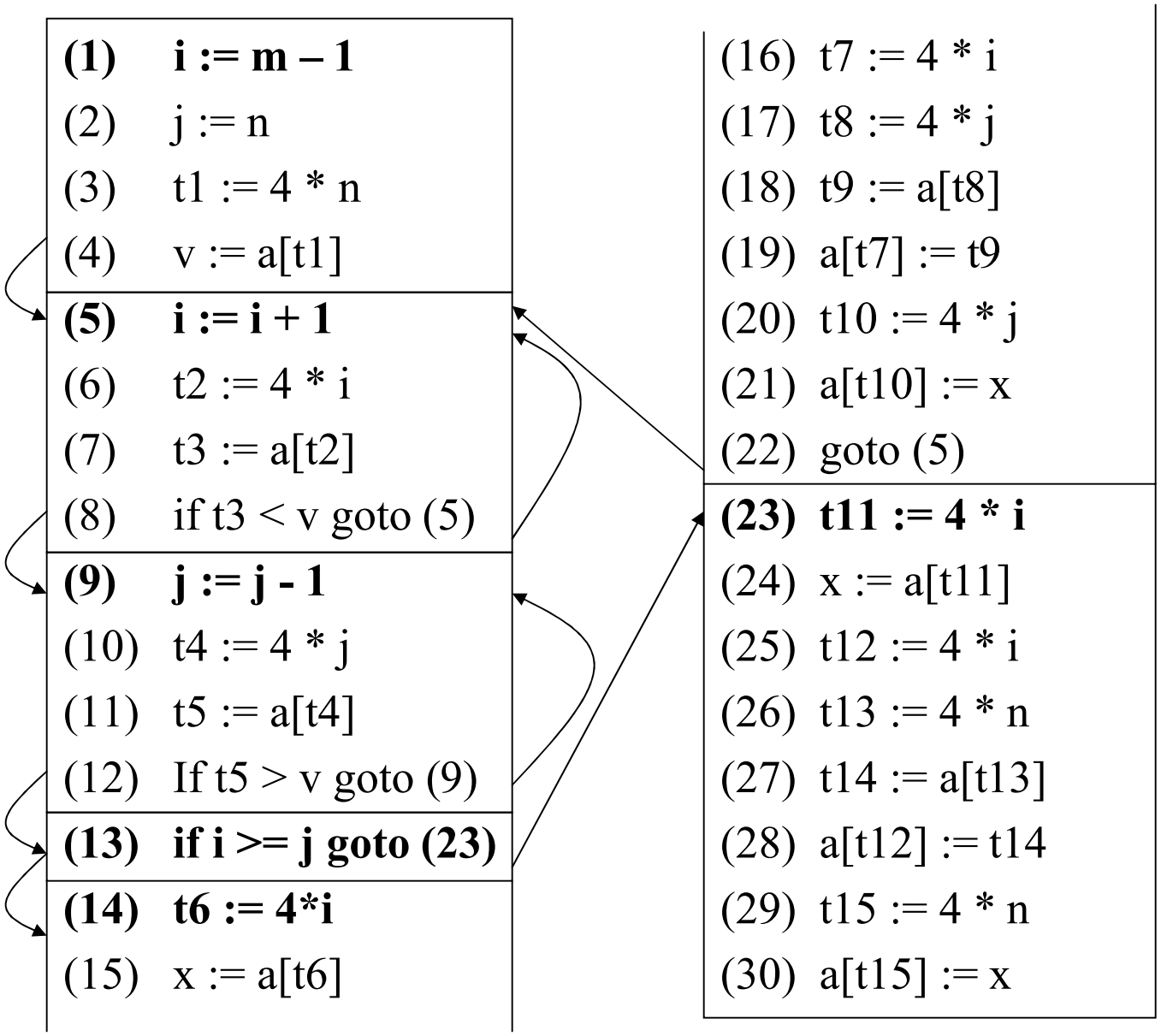
Partitioning into basic blocks

Input: A sequence of intermediate code statements

1. Determine the leaders, the first statements of basic blocks.
   - The first statement in the sequence is a leader.
   - Any statement that is the target of a goto (conditional or unconditional) is a leader.
   - Any statement immediately following a goto (conditional or unconditional) is a leader.
2. For each leader, its basic block is the leader and all statements up to, but not including, the next leader or the end of the program.

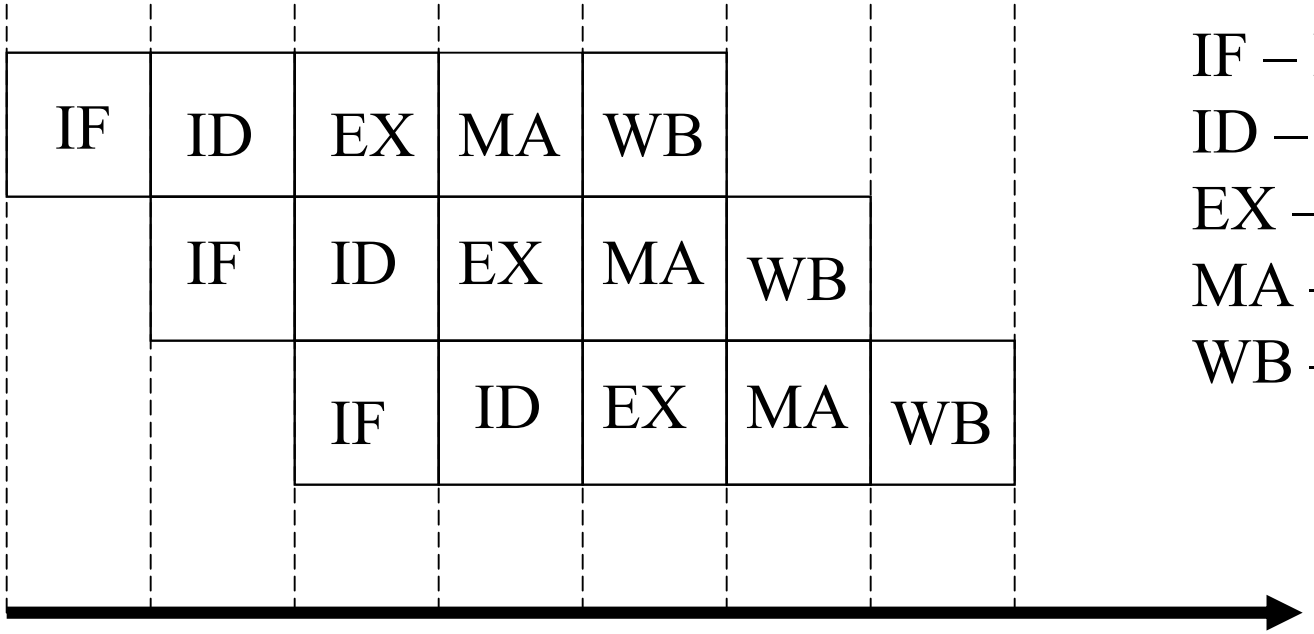
```
(1)   i := m – 1
(2)   j := n
(3)   t1 := 4 * n
(4)   v := a[t1]
(5)   i := i + 1
(6)   t2 := 4 * i
(7)   t3 := a[t2]
(8)   if t3 < v goto (5)
(9)   j := j - 1
(10)  t4 := 4 * j
(11)  t5 := a[t4]
(12)  If t5 > v goto (9)
(13)  if i >= j goto (23)
(14)  t6 := 4*i
(15)  x := a[t6]
(16)  t7 := 4 * i
(17)  t8 := 4 * j
(18)  t9 := a[t8]
(19)  a[t7] := t9
(20)  t10 := 4 * j
(21)  a[t10] := x
(22)  goto (5)
(23)  t11 := 4 * i
(24)  x := a[t11]
(25)  t12 := 4 * i
(26)  t13 := 4 * n
(27)  t14 := a[t13]
(28)  a[t12] := t14
(29)  t15 := 4 * n
(30)  a[t15] := x
```

```
(1)   i := m – 1
(2)   j := n
(3)   t1 := 4 * n
(4)   v := a[t1]
(5)   i := i + 1
(6)   t2 := 4 * i
(7)   t3 := a[t2]
(8)   if t3 < v goto (5)
(9)   j := j - 1
(10)  t4 := 4 * j
(11)  t5 := a[t4]
(12)  If t5 > v goto (9)
(13)  if i >= j goto (23)
(14)  t6 := 4*i
(15)  x := a[t6]
(16)  t7 := 4 * i
(17)  t8 := 4 * j
(18)  t9 := a[t8]
(19)  a[t7] := t9
(20)  t10 := 4 * j
(21)  a[t10] := x
(22)  goto (5)
(23)  t11 := 4 * i
(24)  x := a[t11]
(25)  t12 := 4 * i
(26)  t13 := 4 * n
(27)  t14 := a[t13]
(28)  a[t12] := t14
(29)  t15 := 4 * n
(30)  a[t15] := x
```


(1) i := m – 1
(2) j := n
(3) t1 := 4 * n
(4) v := a[t1]
(5) i := i + 1
(6) t2 := 4 * i
(7) t3 := a[t2]
(8) if t3 < v goto (5)
(9) j := j - 1
(10) t4 := 4 * j
(11) t5 := a[t4]
(12) If t5 > v goto (9)
(13) if i >= j goto (23)
(14) t6 := 4*i
(15) x := a[t6]
(16) t7 := 4 * i
(17) t8 := 4 * j
(18) t9 := a[t8]
(19) a[t7] := t9
(20) t10 := 4 * j
(21) a[t10] := x
(22) goto (5)
(23) t11 := 4 * i
(24) x := a[t11]
(25) t12 := 4 * i
(26) t13 := 4 * n
(27) t14 := a[t13]
(28) a[t12] := t14
(29) t15 := 4 * n
(30) a[t15] := x

Instruction Scheduling

- Choosing the order of instructions to best utilize resources (CPU, registers, …)
- Consider RISC pipeline Architecture:



IF – Instruction Fetch
ID – Instruction Decode
EX – Execute
MA – Memory access
WB – Write back

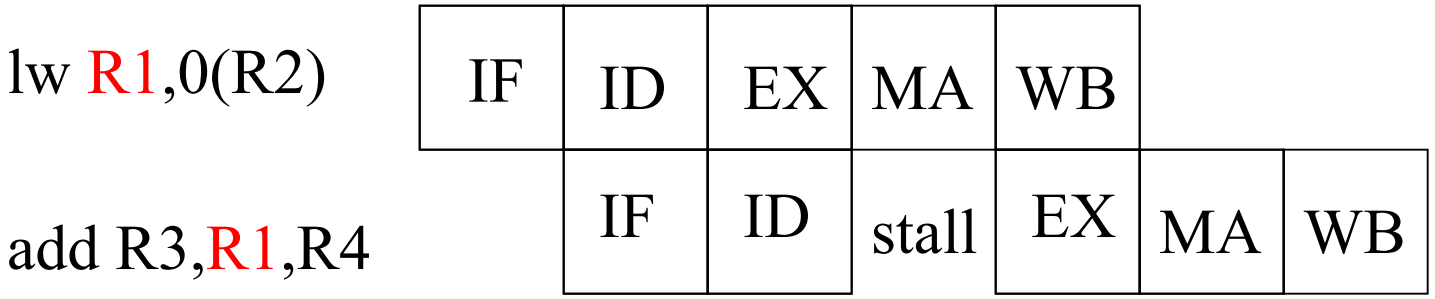
Hazards

1. Structural hazards – machine resources limit overlap
2. Data hazards – output of instruction needed by later instruction
3. Control hazards – branching

Pipeline stalls!

Data Hazards

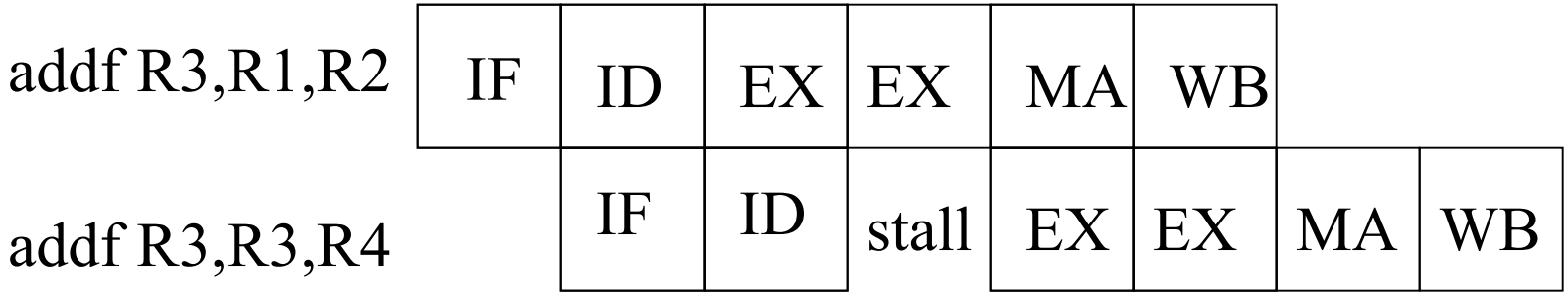
Memory latency:



Can't add until register R1 is loaded.

Data Hazards

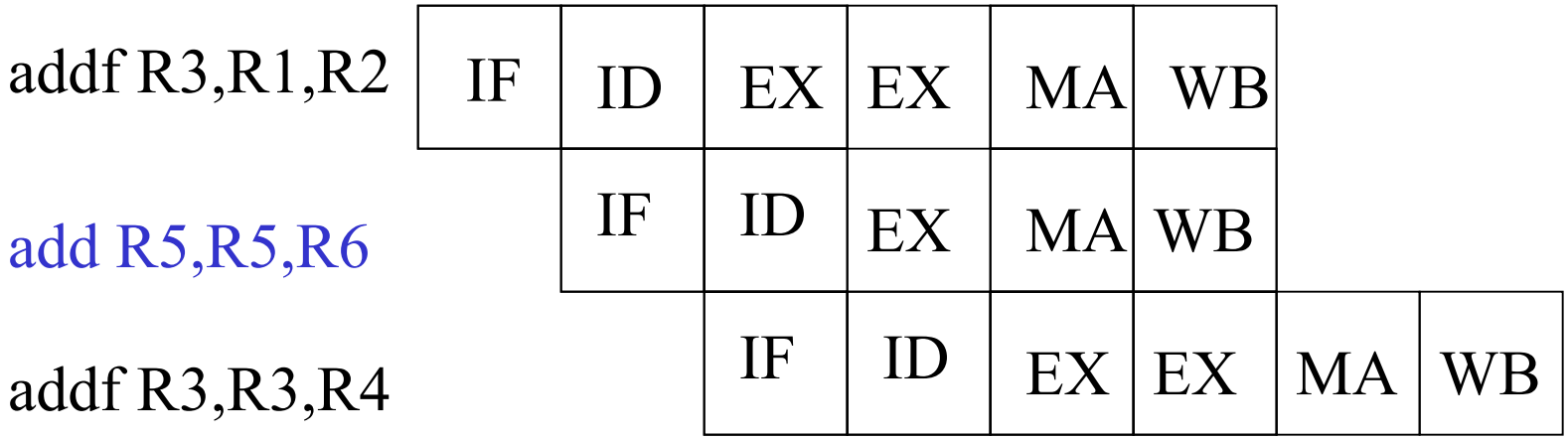
Instruction latency:



Assumes floating point ops take 2 execute cycles

Dealing with Data Hazards

- Typical solution is to re-order statements.
- To do this without changing the outcome, need to understand the relationship (dependences) between statements.



Instruction Scheduling

- Many operations have non-zero latencies
- Execution time is *order-dependent*

Assumed latencies *(conservative)*

Operation	Cycles
load	3
store	3
loadI	1
add	1
mult	2
fadd	1
fmult	2
shift	1
branch	0 to 8

- **Loads & stores may or may not block**
  - **> Non-blocking ⇒fill those issue slots**
- **Scheduler should hide the latencies**

w ← w * 2 * x * y * z

- Schedule 1

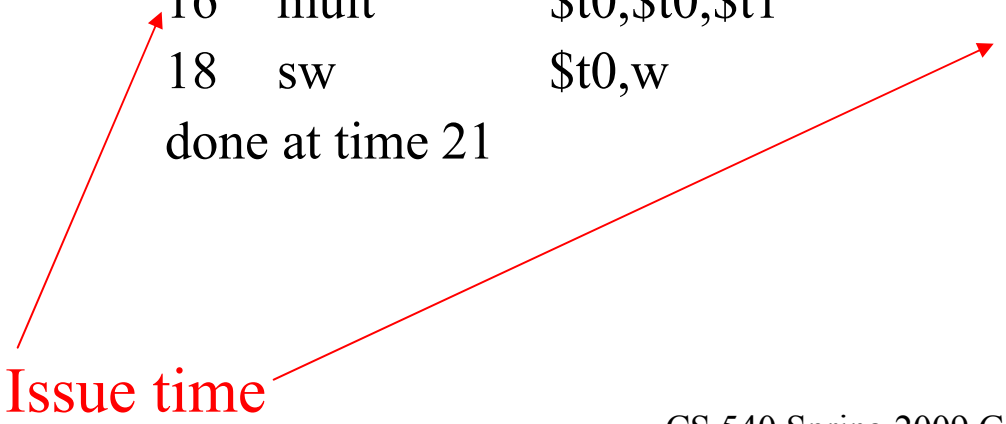
1	lw	$t0,w
4	add	$t0,$t0,$t0
5	lw	$t1,x
8	mult	$t0,$t0,$t1
9	lw	$t1,y
12	mult	$t0,$t0,$t1
13	lw	$t1,z
16	mult	$t0,$t0,$t1
18	sw	$t0,w

done at time 21

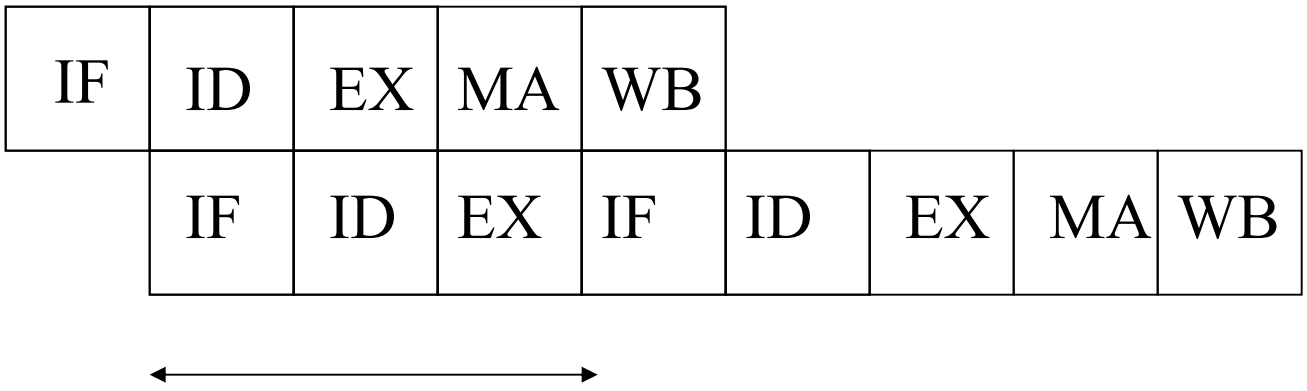
- Schedule 2

1	lw	$t0,w
2	lw	$t1,x
3	lw	$t2,y
4	add	$t0,$t0,$t0
5	mult	$t0,$t0,$t1
6	lw	$t1,z
7	mult	$t0,$t0,$t2
9	mult	$t0,$t0,$t1
11	sw	$t0,w

done at time 14



Control Hazards



Branch Scheduling

Problem:

- Branches often take some number of cycles to complete, creating delay slots.
- Can be a delay between *a compare b* and its associated branch.
- Even unconditional branches have delay slots

A compiler will try to fill these delay slots with valid instructions (rather than *nop*).

Example

Assume loads take 2 cycles and branches have a delay slot.

7 cycles

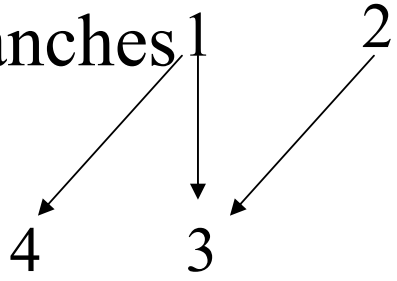
instruction	start time
lw $t2,$t1(4)	1
lw $t3,$(t1)(8)	2
add $t4, $t2, $t3	4
add $t5, $t2,1	5
b L1	6
nop	7

Example

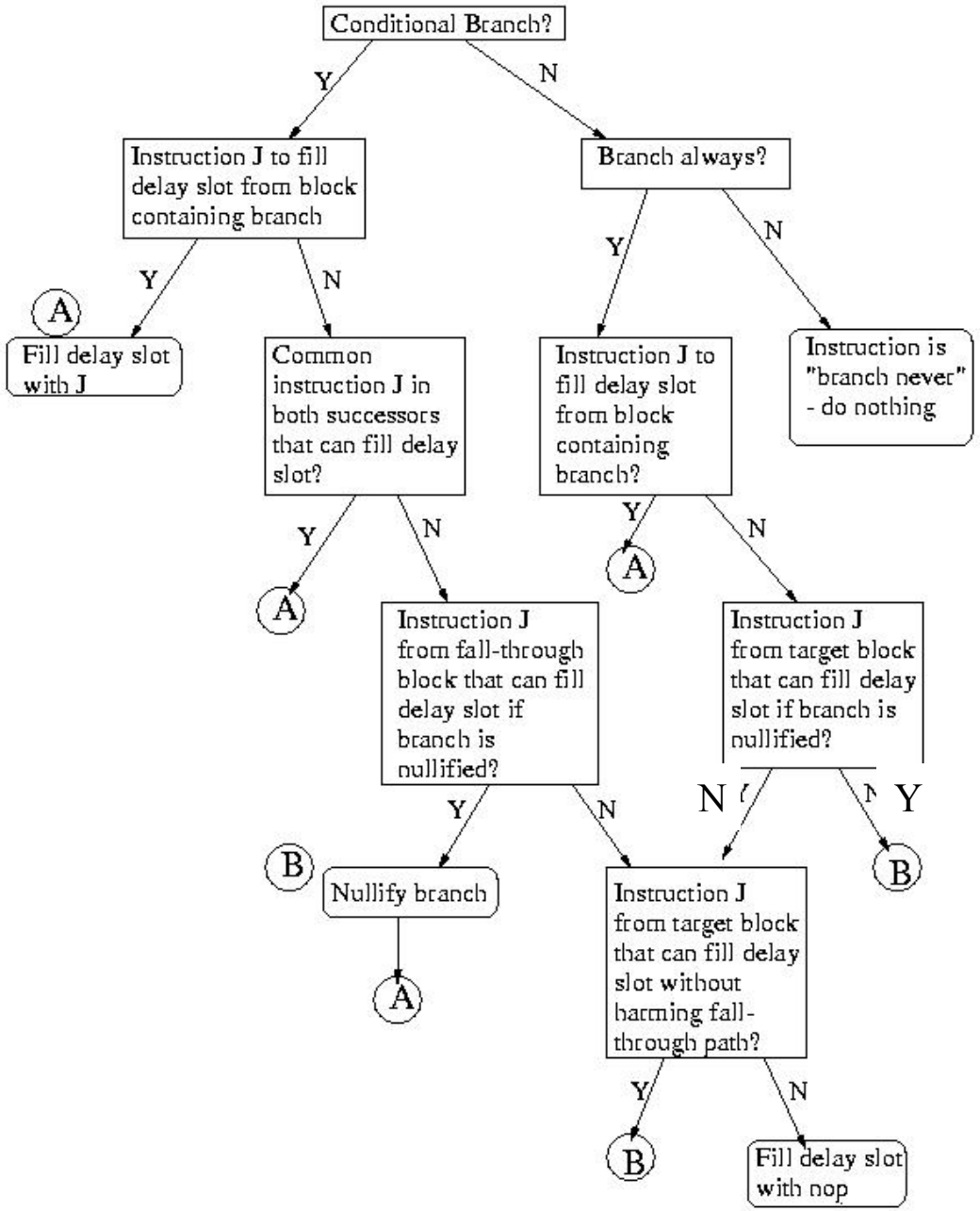
Can look at the dependencies between the statements and move a statement into the delay slot.

instruction	start time
lw $t2,$t1(4)	1
lw $t3,$(t1)(8)	2
add $t5, $t2,1	3
b L1	4
add $t4, $t2, $t3	5

5 cycles Filling Delay branches



Filling the delay slot in the SPARC architecture



Register Allocation

How to best use the bounded number of registers.

- Reducing load/store operations
- What are 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

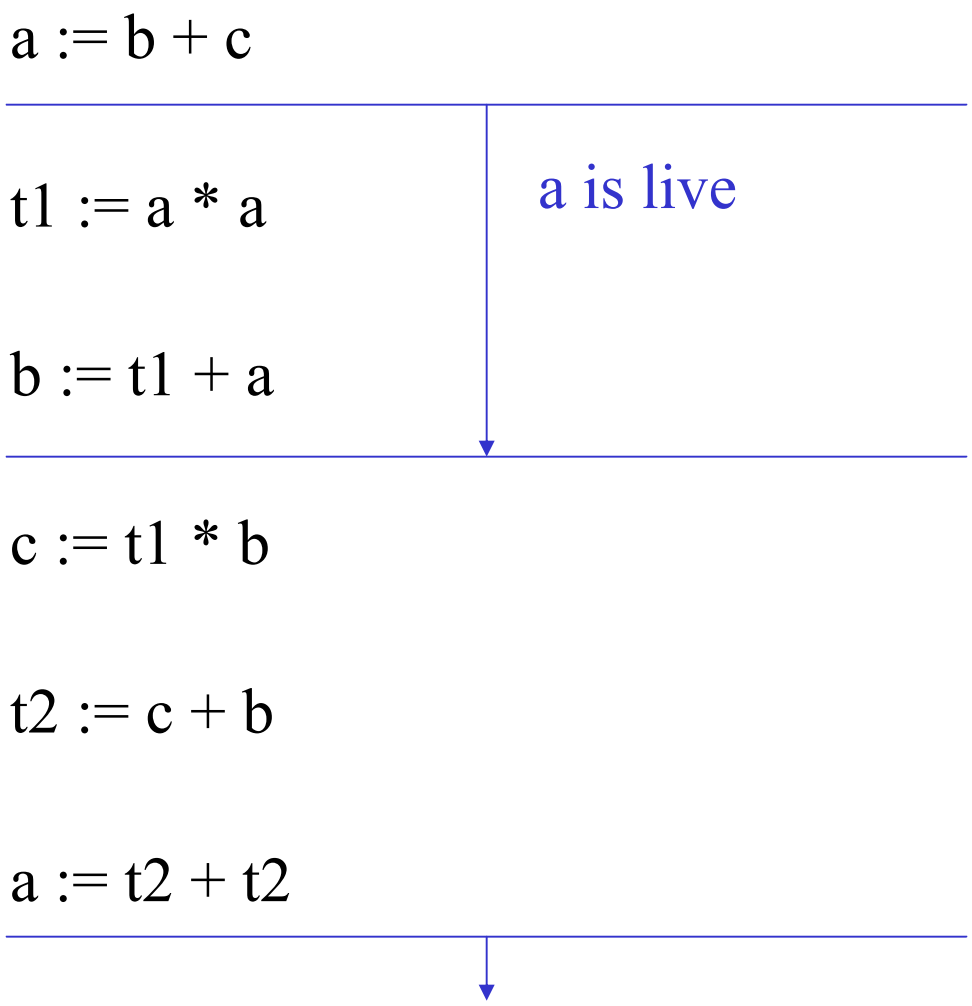
- Deal with each basic block individually.
- Compute liveness information for the block.
- Using liveness information, generate code that uses registers as well as possible.
- At end, generate code that saves any live values left in registers.

.

Concept: Variable Liveness

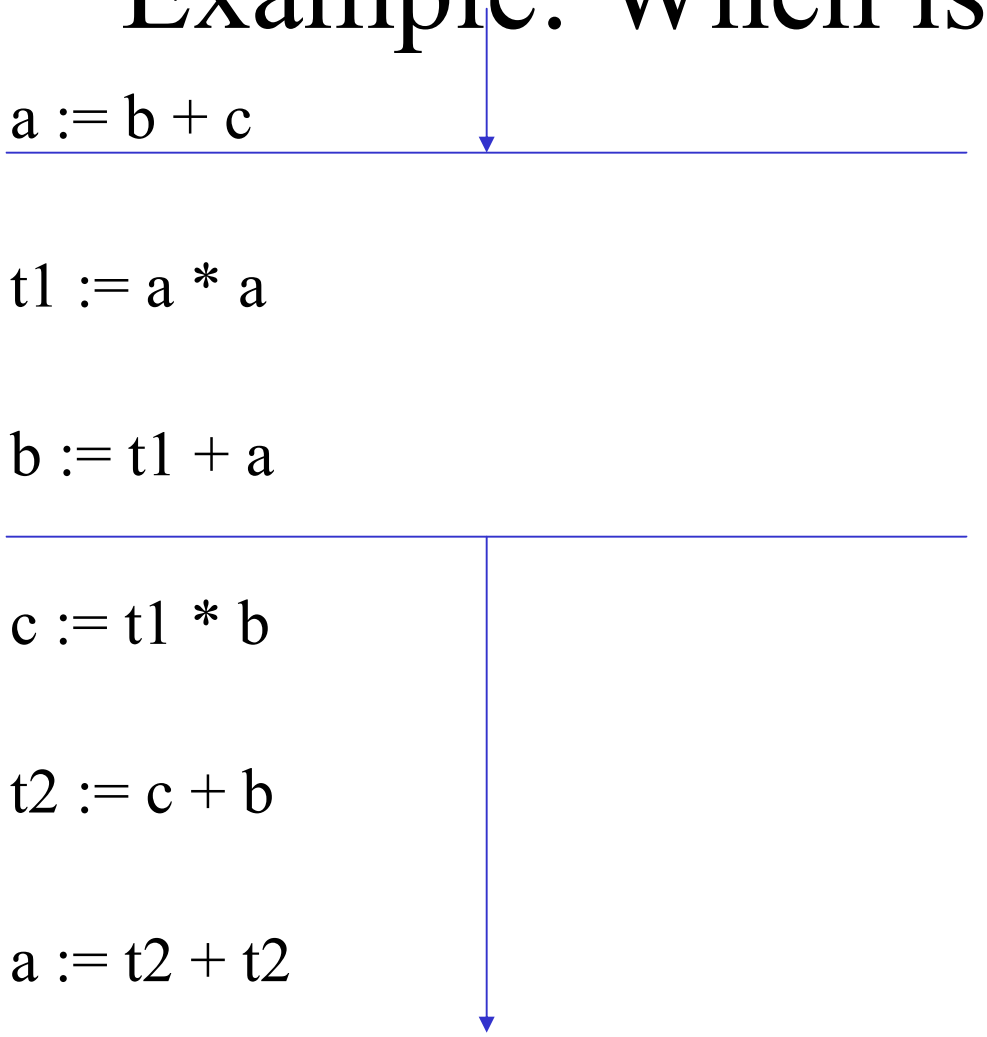
- For some statement *s*, variable *x* is **live** if
  - there is a statement *t* that uses *x*
  - there is a path in the CFG from *s* to *t*
  - there is no assignment to *x* on some path from *s* to *t*
- **A variable is *live* at a given point in the source code if it could be used before it is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

Example: When is a live?



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

Example: When is b live?



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} ← 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}

Basic Code Generation

- Deal with each basic block individually.
- Compute liveness information for the block.
- Using liveness information, generate code that uses registers as well 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")
- 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 according.

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_c = $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?

- Liveness information allows us to keep values in registers if they will be used later (efficiency)
- Why do we assume all variables are live at the end of blocks? Can we do better?
- Why do we need to save live variables at the end? We might have to reload them in the next block.

Register Allocation with Graph Coloring

Local register allocation - graph coloring problem.

Uses liveness information.

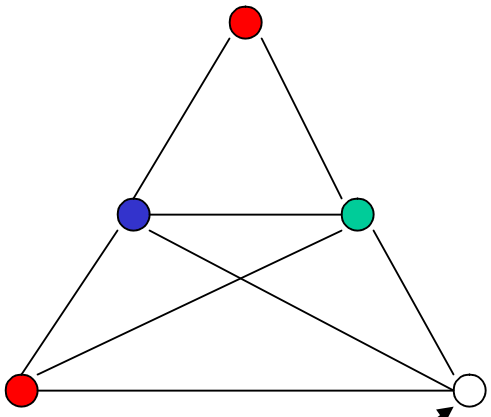
Allocate K registers where each register is associated with one of the K colors.

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).
- Problem is NP (for more than 2 colors) ➔ no polynomial time solution.
- Fortunately there are approximation algorithms.

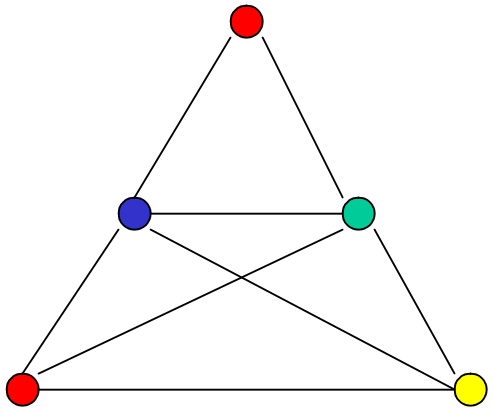
Coloring a graph with K colors

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

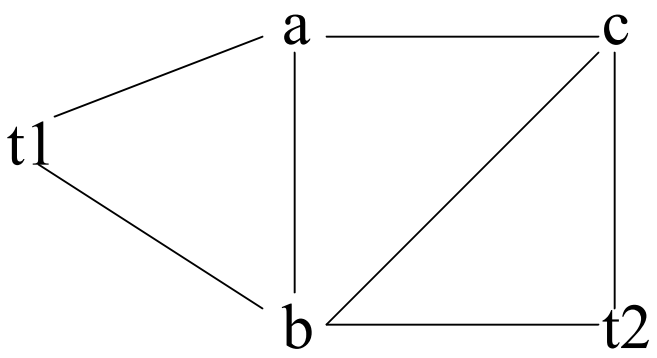
G is an '*interference graph*'

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

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.

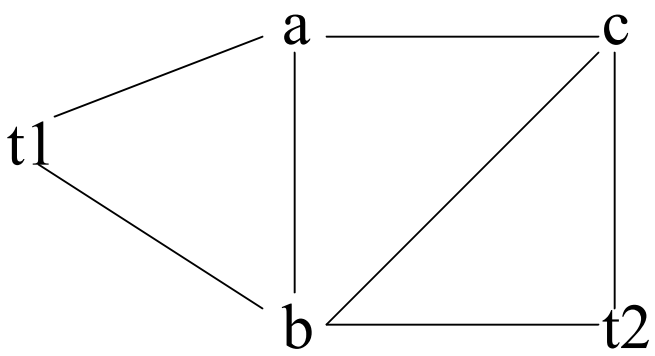
Process may require iterations and rewriting of some of the code to create more temporaries.

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

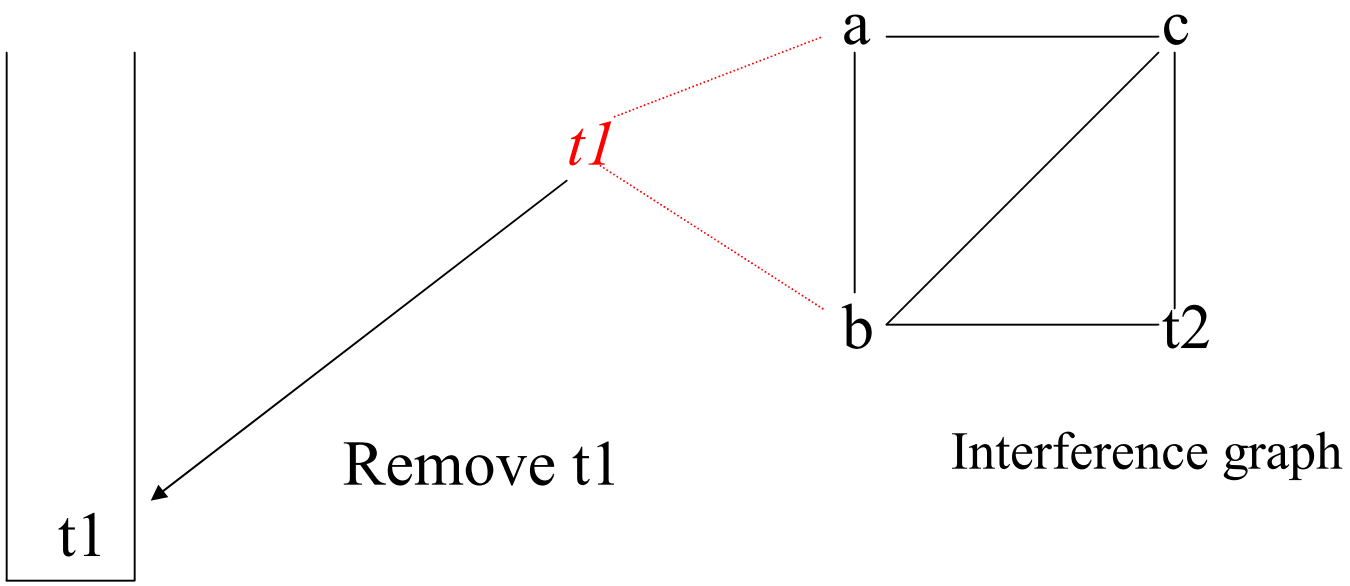
Back to example

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}



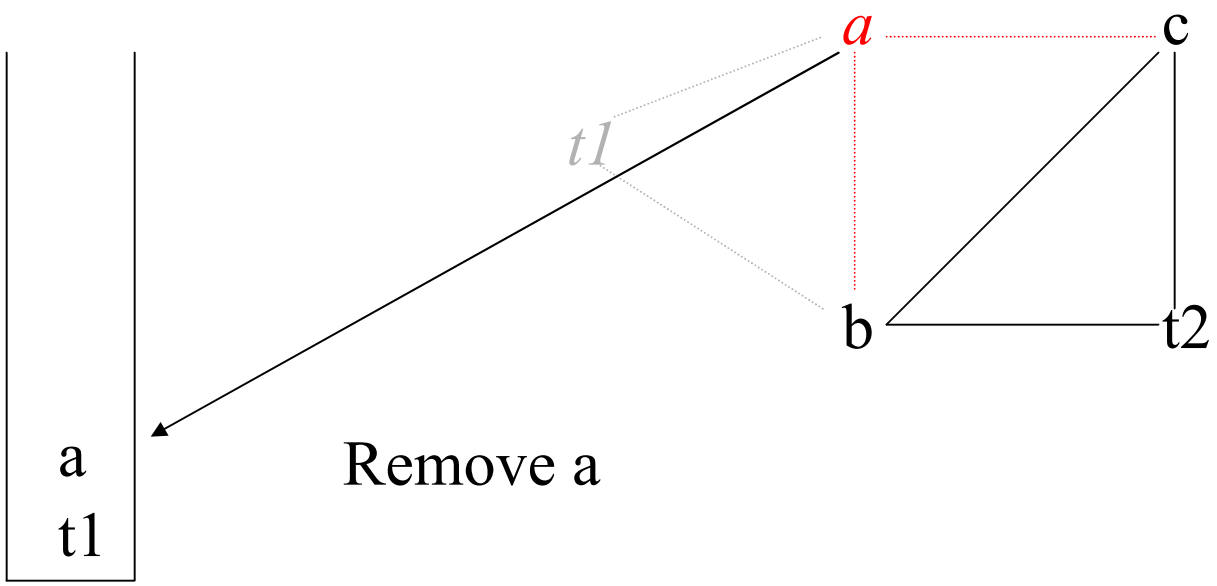
Example, k = 3

Assume k = 3



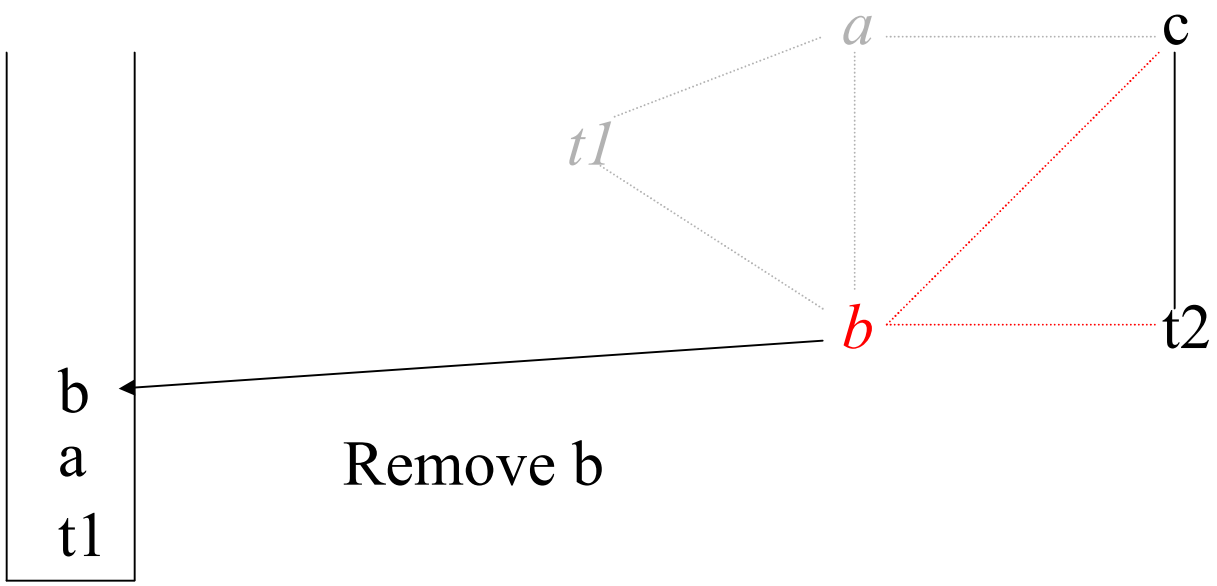
Example

Assume k = 3



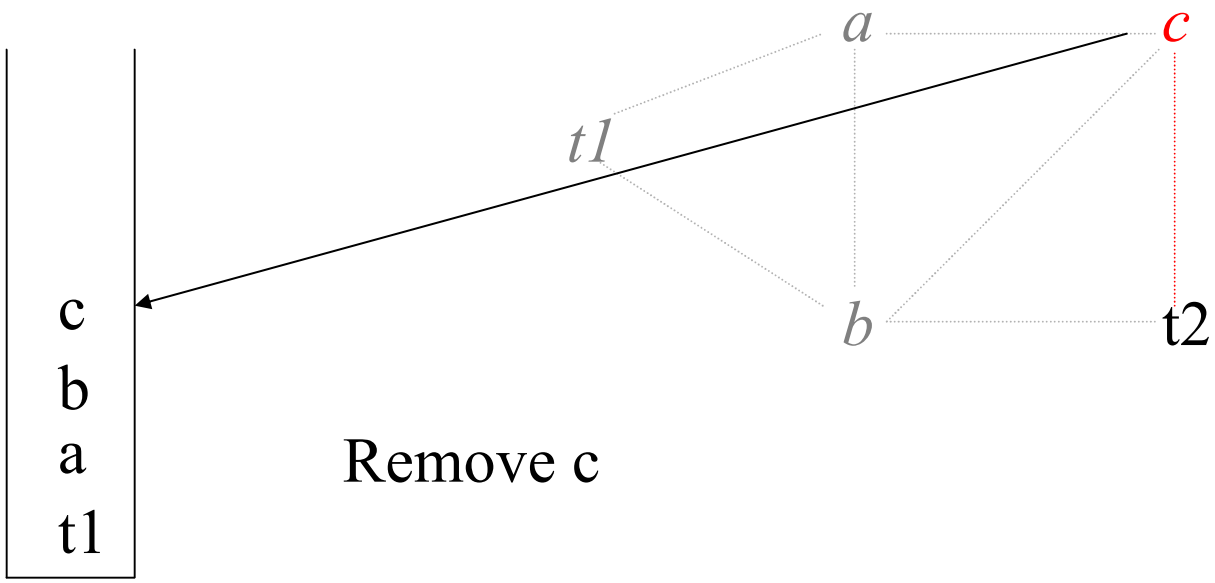
Example

Assume k = 3



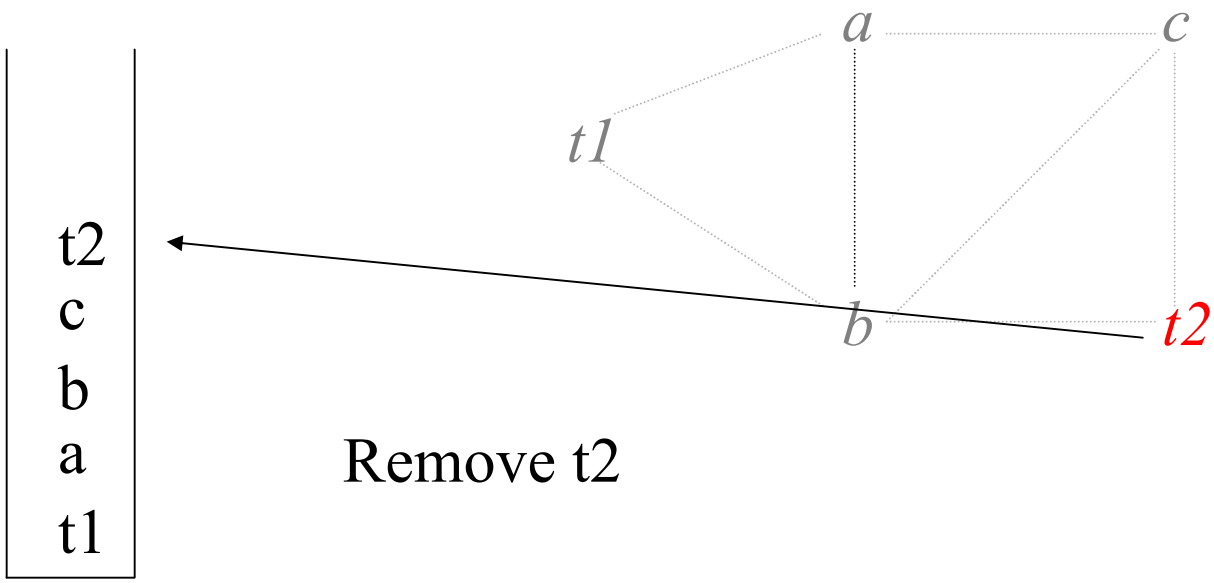
Example

Assume k = 3



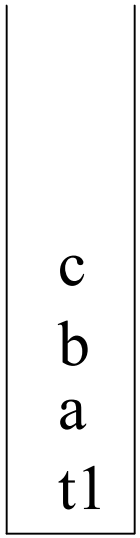
Example

Assume k = 3



Rebuild the graph

Assume k = 3



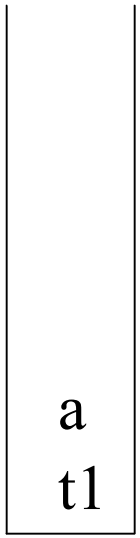
t2

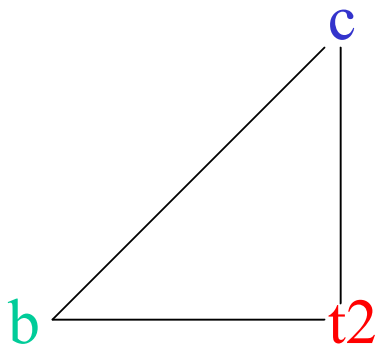
Example

Assume k = 3

Example

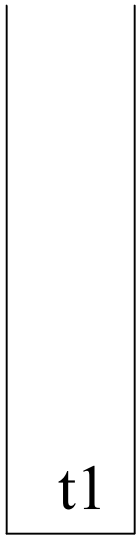
Assume k = 3

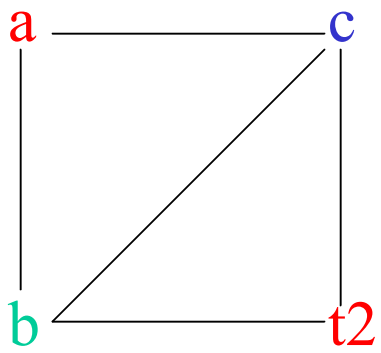




Example

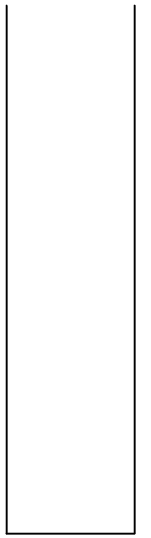
Assume k = 3



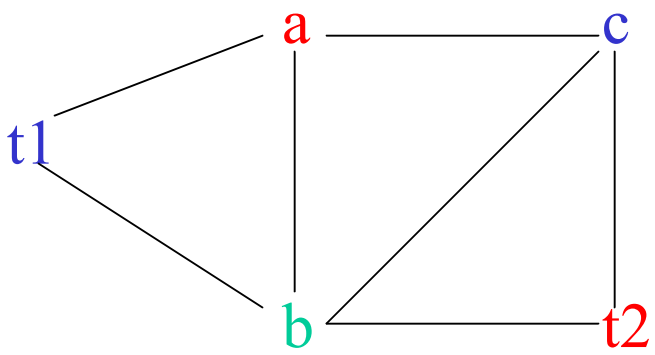


Example

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

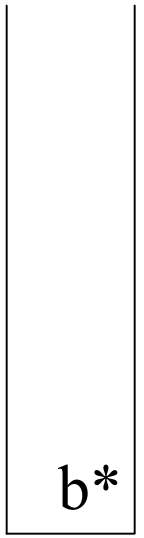
```
lw $t0,b
lw $t1,c
add $t0,$t0,$t1
mul $t1,$t0,$t0
add $t0,$t1,$t0
mul $t1,$t1,$t0
add $t2,$t1,$t0
add $t2,$t2,$t2
sw $t2, a
sw $t0,b
sw $t1,c
```

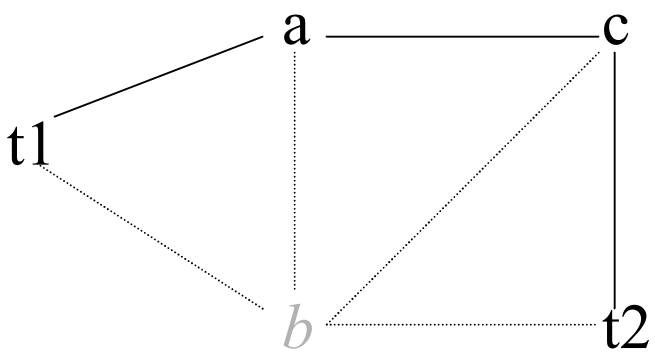
Generated Code: Coloring

```
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

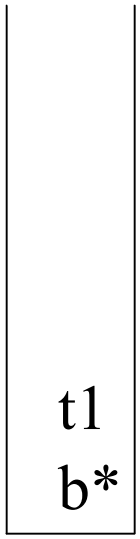




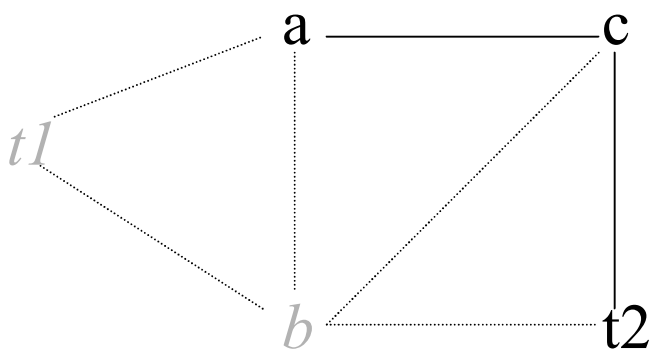
Remove b as spill

Example

Assume k = 2

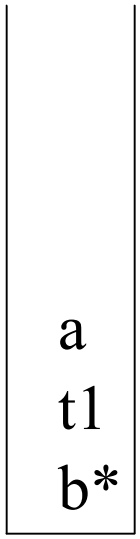


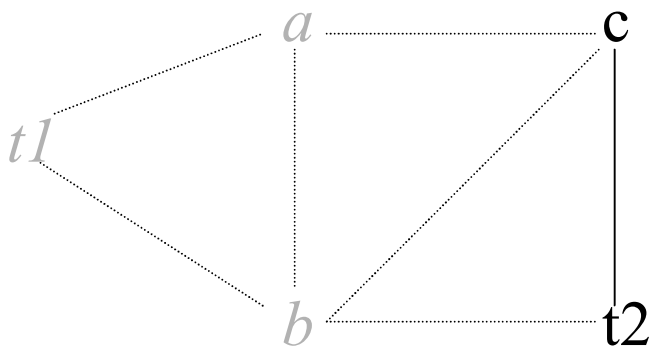
Remove t1



Example

Assume k = 2

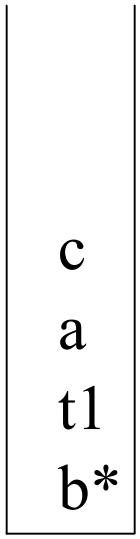


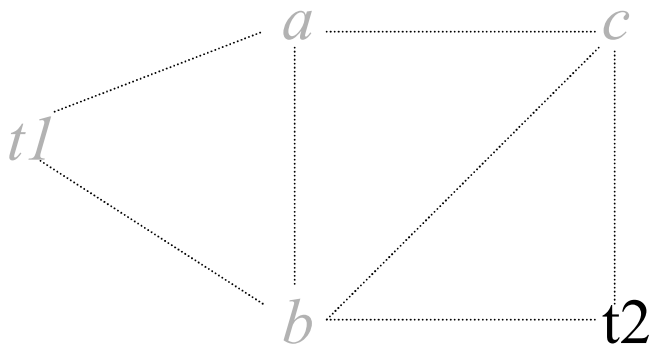


Remove a

Example

Assume k = 2

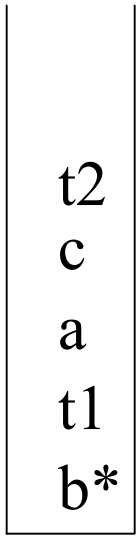


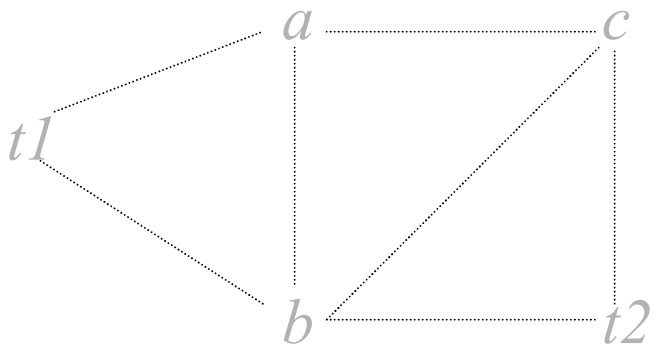


Remove c

Example

Assume k = 2

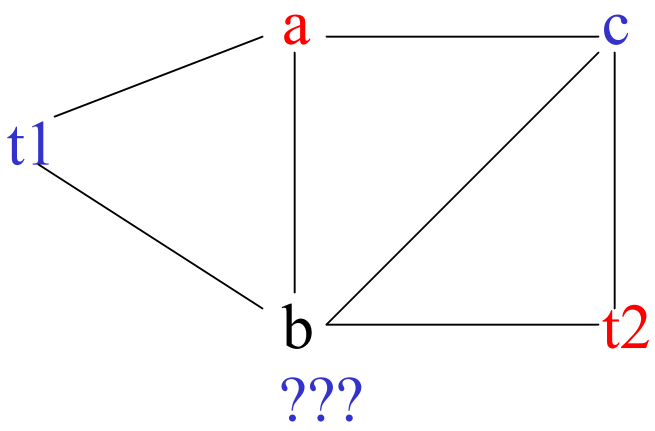




Remove t2

Example

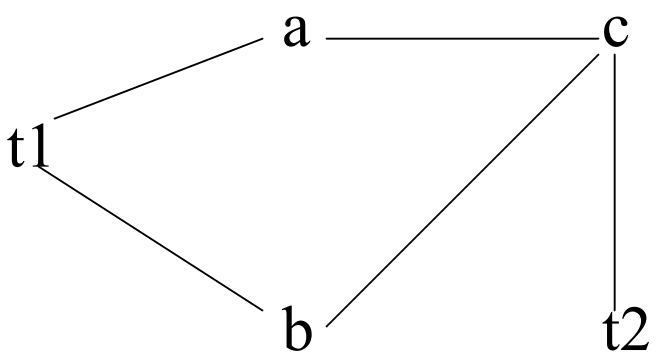
Assume k = 2



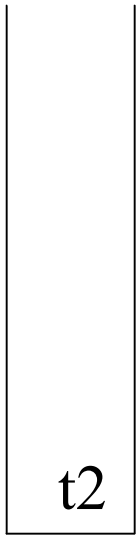
Can flush b out to memory, creating a smaller window

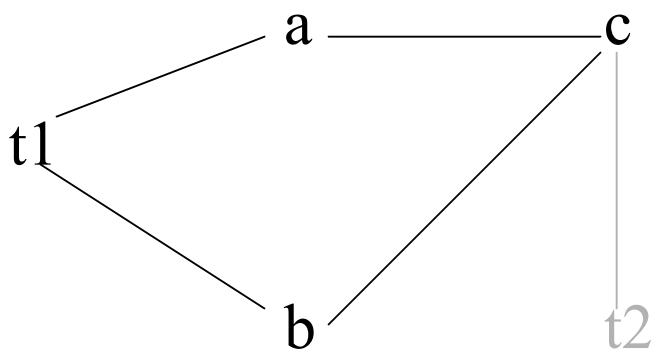
After spilling b:

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	{c,t2}
	{a, c}

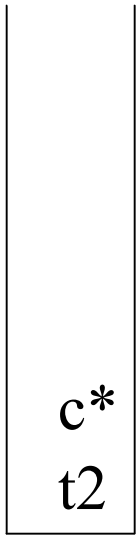


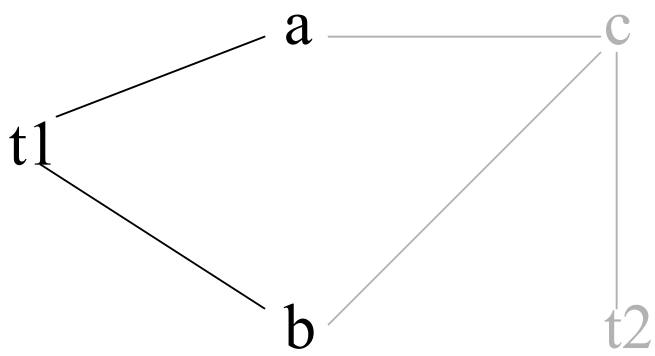
After spilling b:





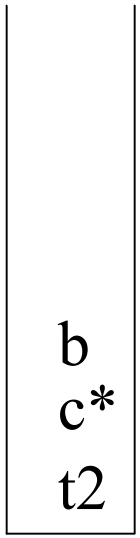
After spilling b:

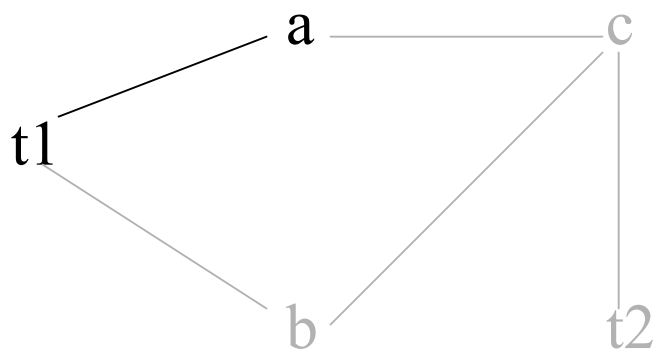




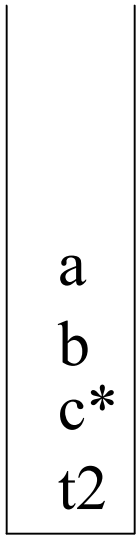
Have to choose c as a potential spill node.

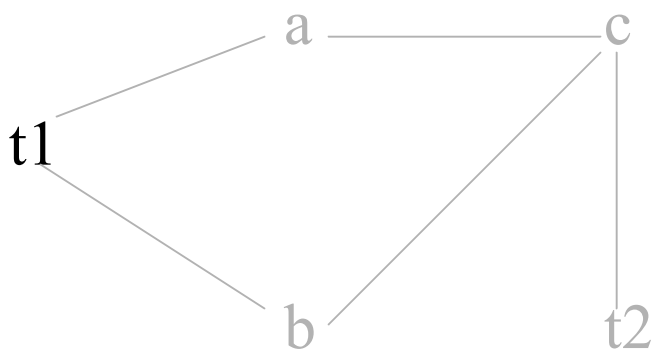
After spilling b:





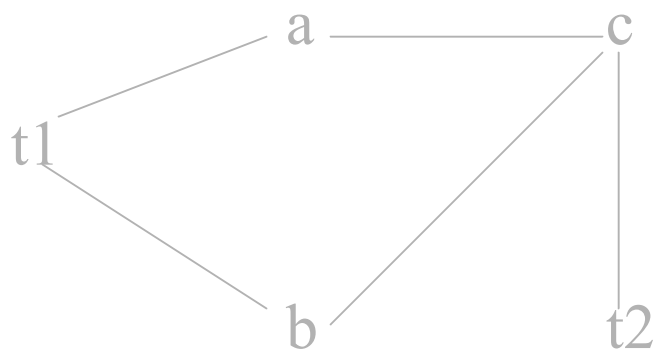
After spilling b:



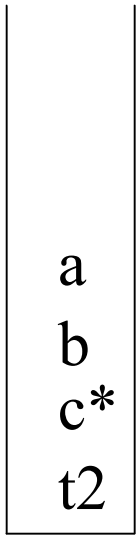


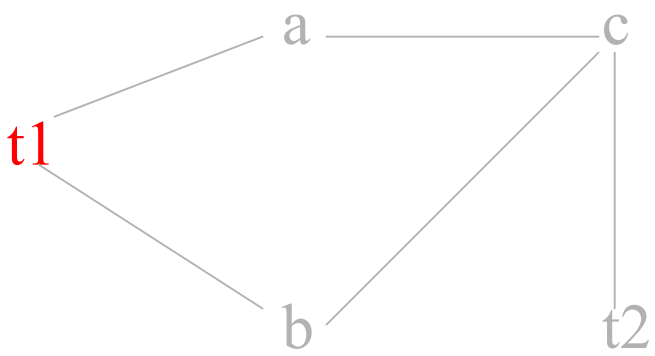
After spilling b:

t1
a
b
c*
t2

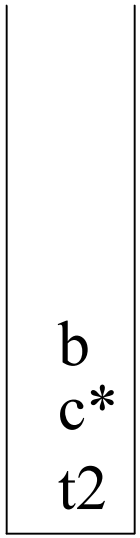


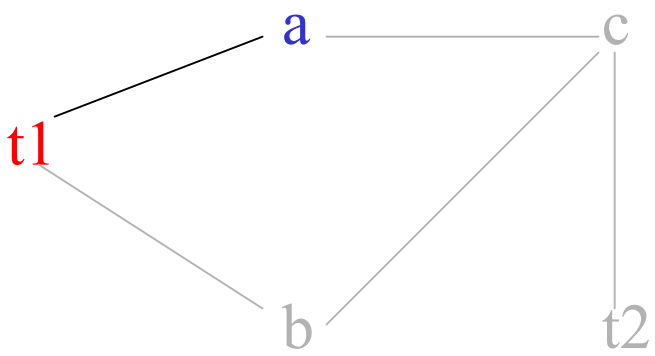
Now rebuild:



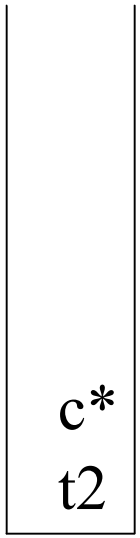


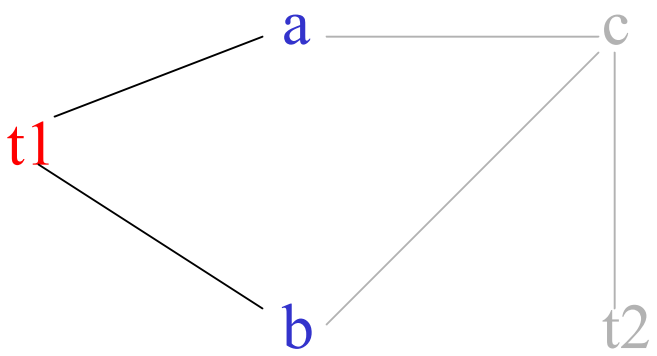
Now rebuild:



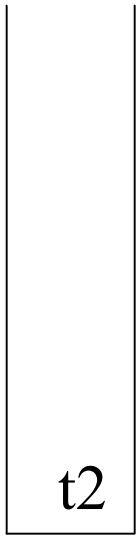


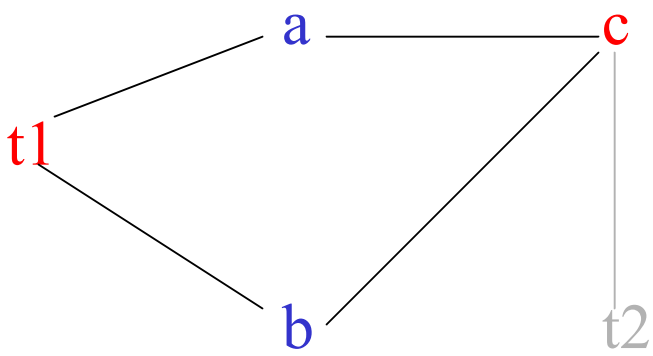
Now rebuild:





Now rebuild:

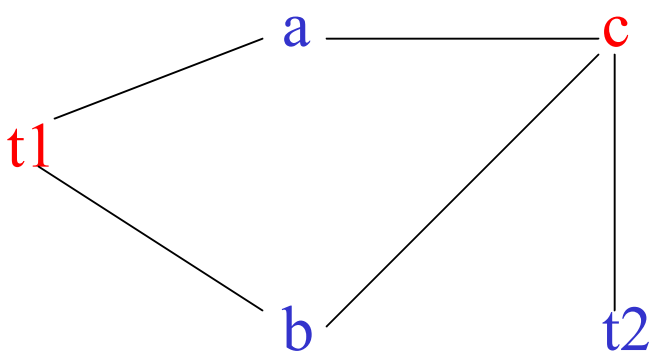




Fortunately, there is a color for c

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
```