Lecture 16

Register Allocation: Coalescing and Spilling

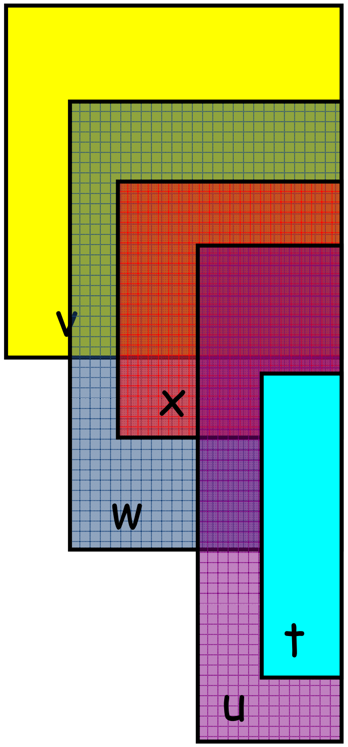
(Slides courtesy of Seth Goldstein and David Koes.)

Review: An Example, k=4

```
v <-  1
w <-  v + 3
x <-  w + v
u <-  v
t <-  u + x
  <-  w
  <-  t
  <-  u
```

Review: An Example, k=4

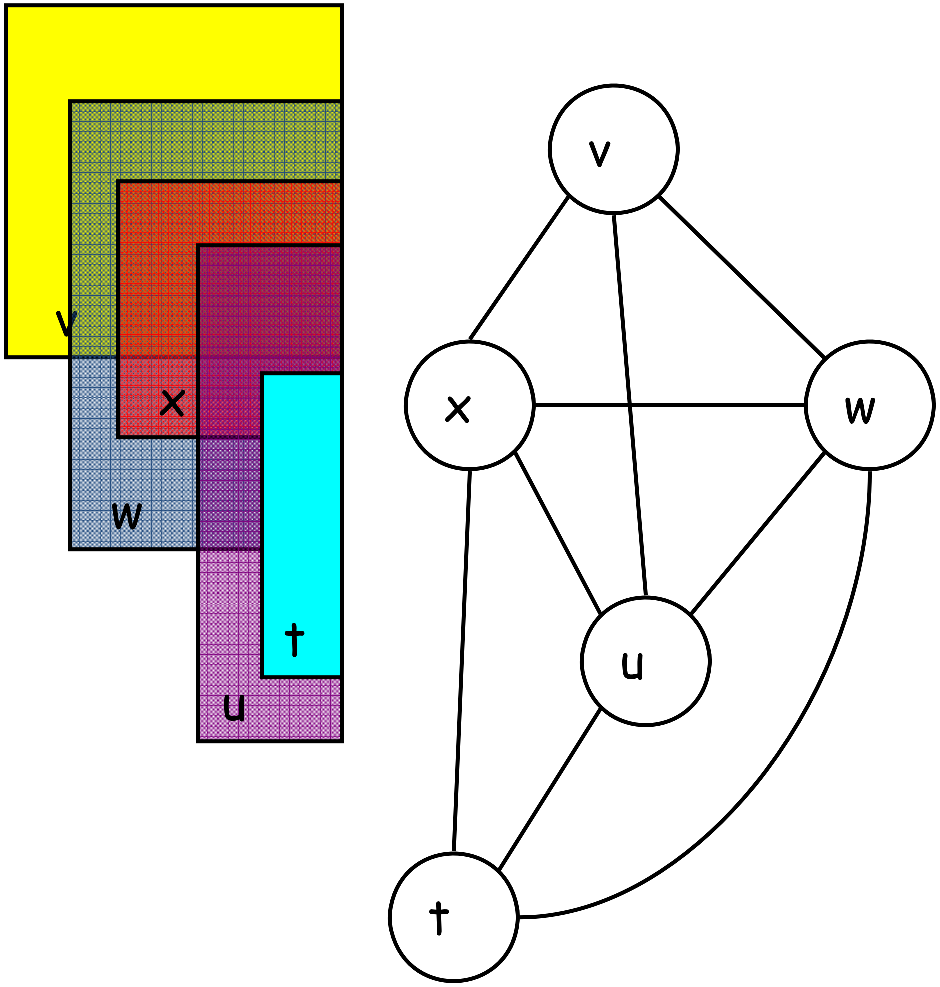
```
v <-  1
w <-  v + 3
x <-  w + v
u <-  v
t <-  u + x
  <-  w
  <-  t
  <-  u
```



Compute live ranges

Review: An Example, k=4

```
v <-  1
w <-  v + 3
x <-  w + v
u <-  v
t <-  u + x
  <-  w
  <-  t
  <-  u
```

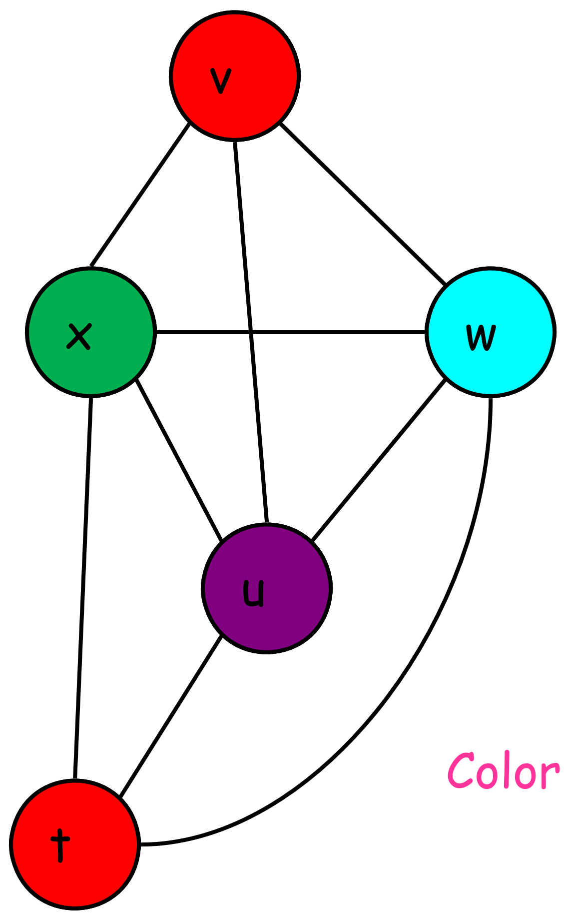


Construct the interference graph

Review: An Example, k=4

```
v <-  1
w <-  v + 3
x <-  w + v
u <-  v
t <-  u + x
  <-  w
  <-  t
  <-  u
```

Voila, registers are assigned!

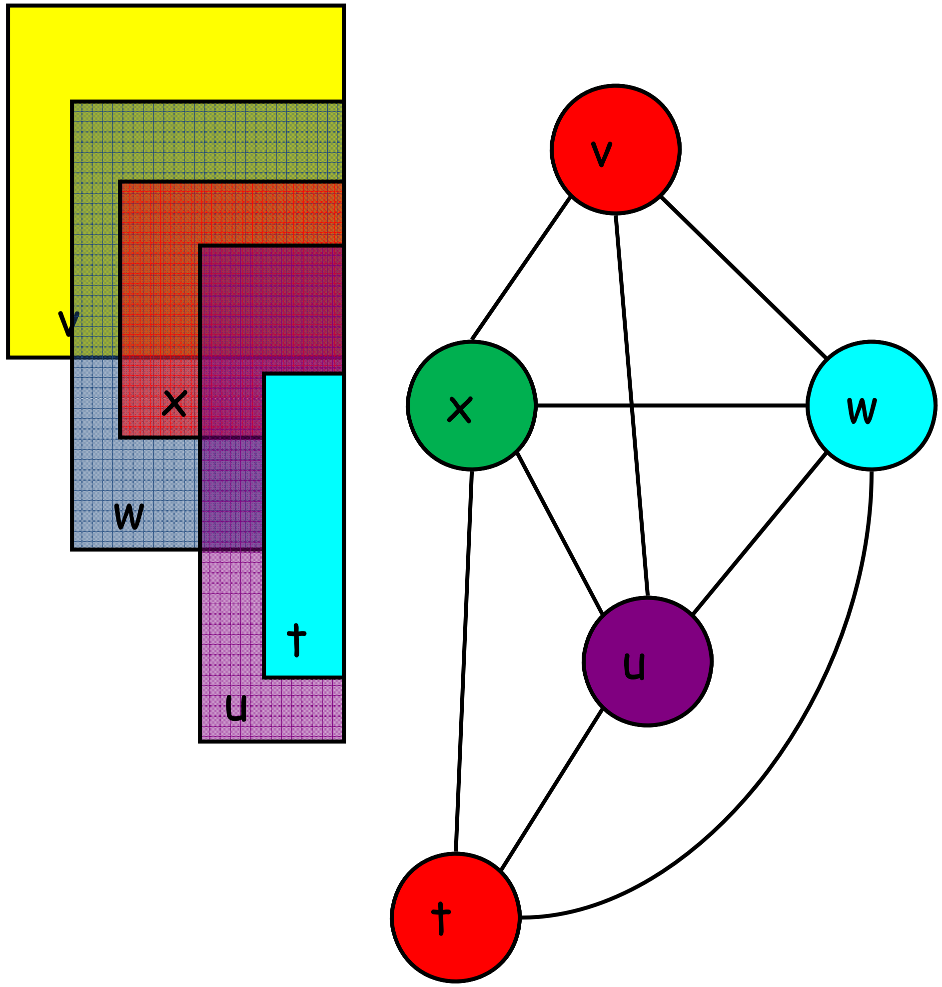


Color the graph

But, can we do better?

An Example, k=4

```
v <-  1
w <-  v + 3
x <-  w + v
u <-  v
t <-  u + x
  <-  w
  <-  t
  <-  u
```



u & v are special. They interfere, but only through a move!

An Example, k=4

uv <- 1

w <- uv + 3

x <- w + uv

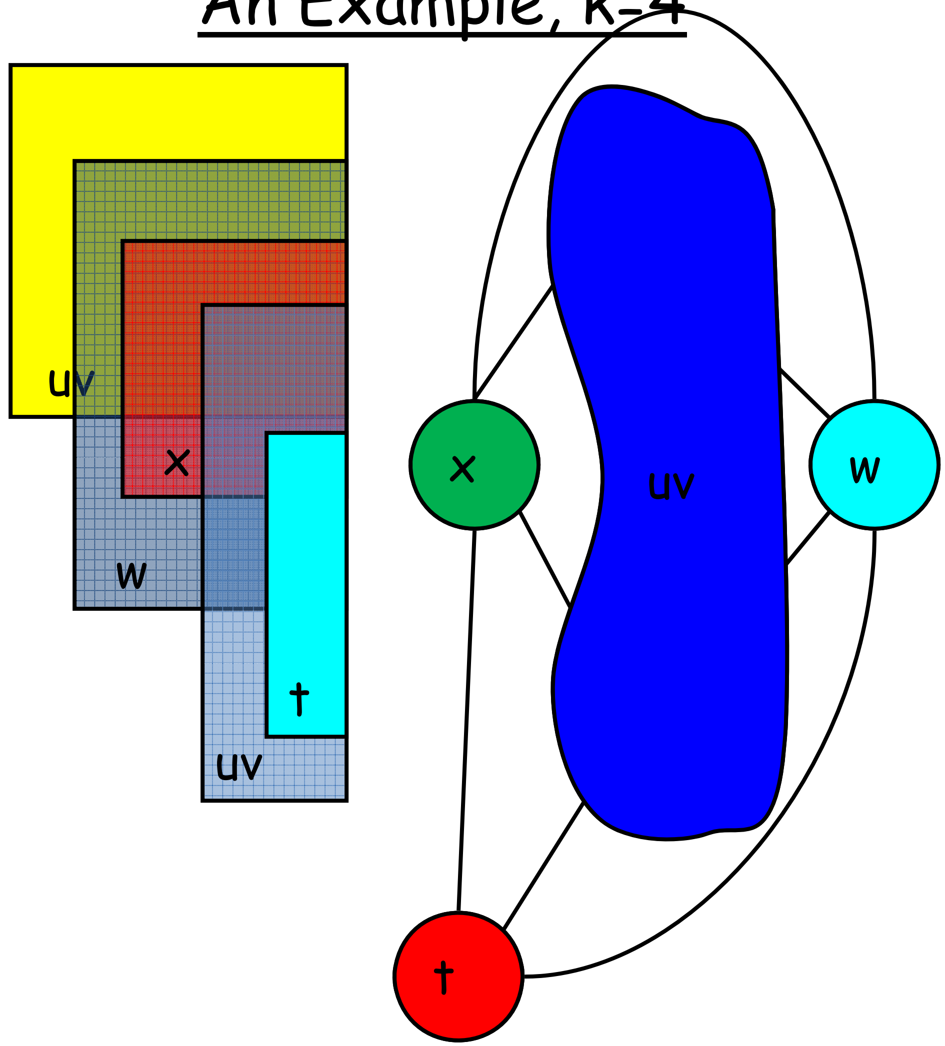
~~u <- v~~

t <- uv + x

<- w

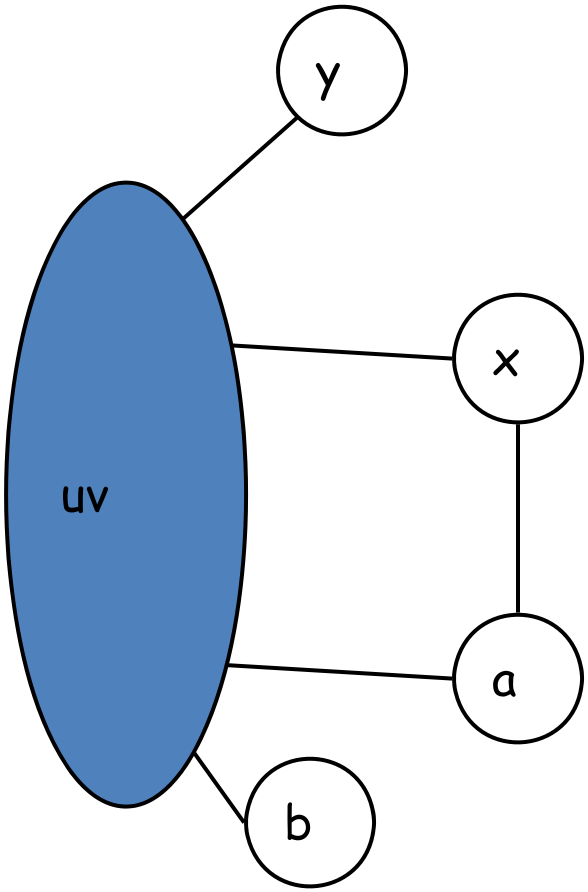
<- t

<- uv



Rewrite the code to coalesce u & v

Is Coalescing Always Good?

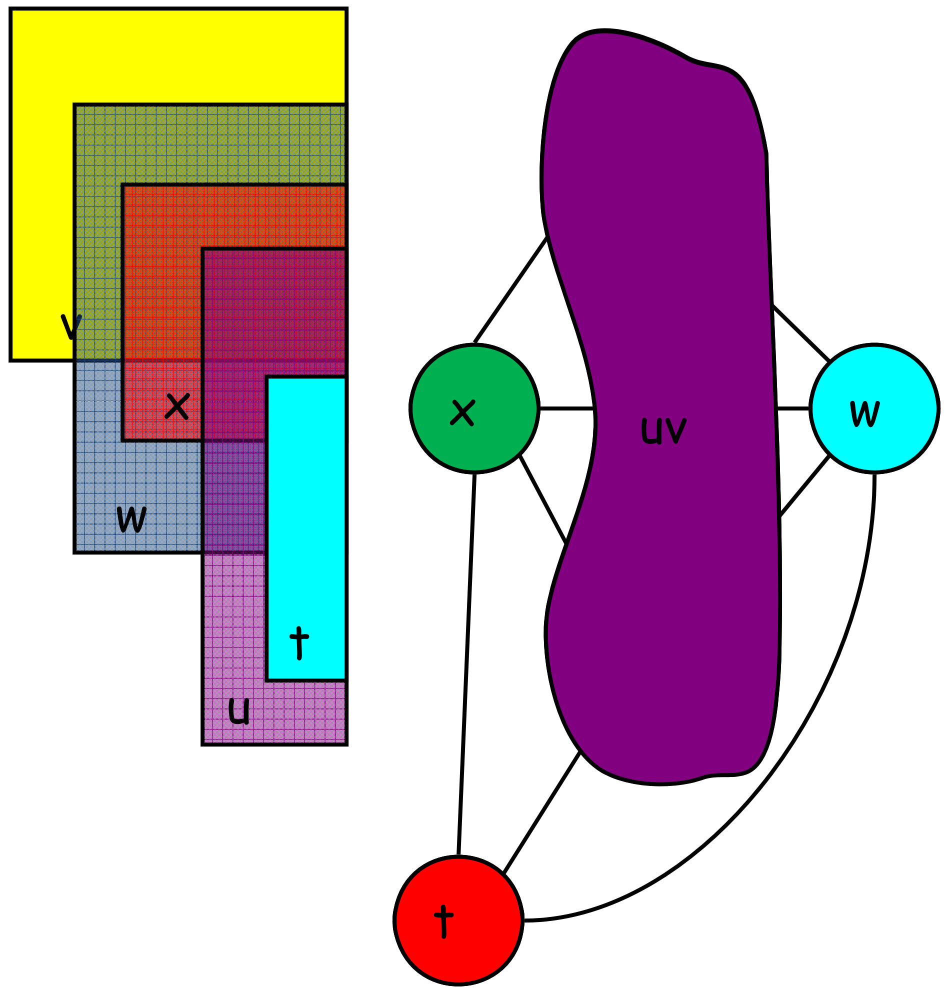


Was 2-colorable,
now it needs 3 colors

So, we treat moves specially.

An Example, k=4

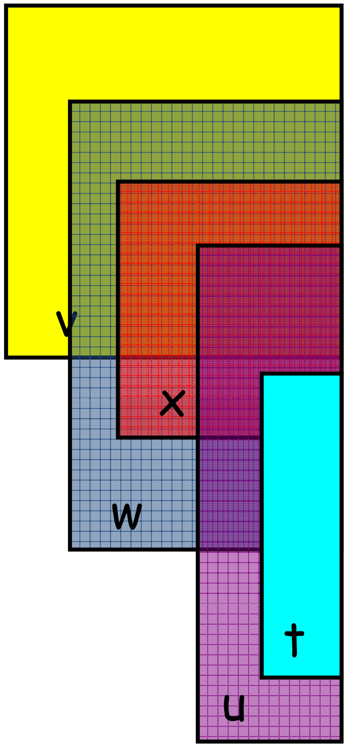
```
v <-  1
w <-  v + 3
x <-  w + v
u <-  v
t <-  u + x
  <-  w
  <-  t
  <-  u
```



Interference from moves become "move edges."

An Example, k=3

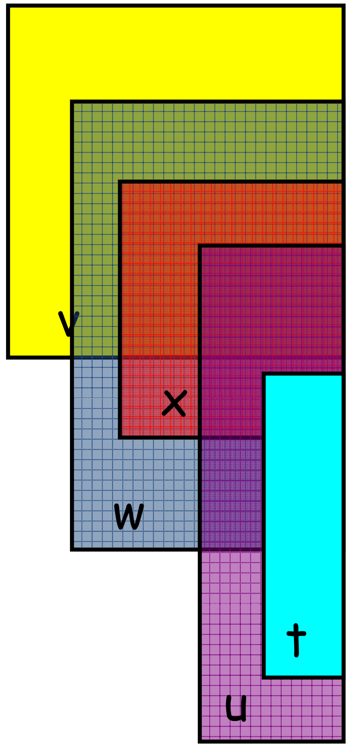
```
v <-  1
w <-  v + 3
x <-  w + v
u <-  v
t <-  u + x
  <-  w
  <-  t
  <-  u
```

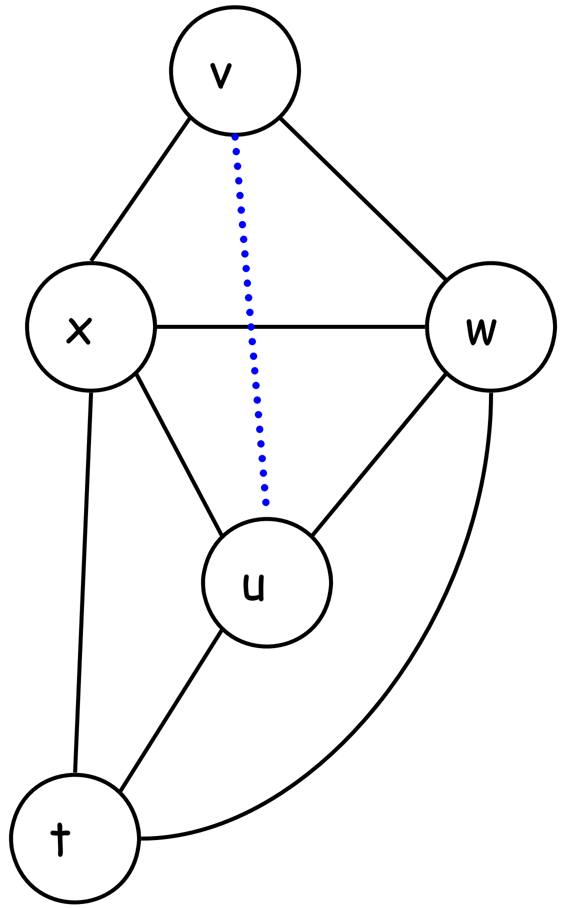


Compute live ranges

An Example, k=3

```
v <-  1
w <-  v + 3
x <-  w + v
u <-  v
t <-  u + x
  <-  w
  <-  t
  <-  u
```

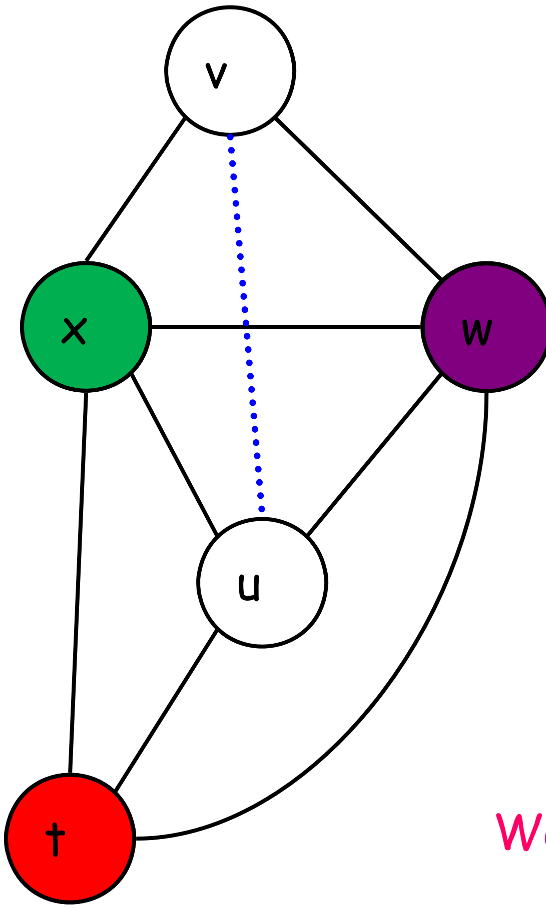




Construct the interference graph

An Example, k=3

```
v <-  1
w <-  v + 3
x <-  w + v
u <-  v
t <-  u + x
  <-  w
  <-  t
  <-  u
```



We need to spill

Color the interference graph

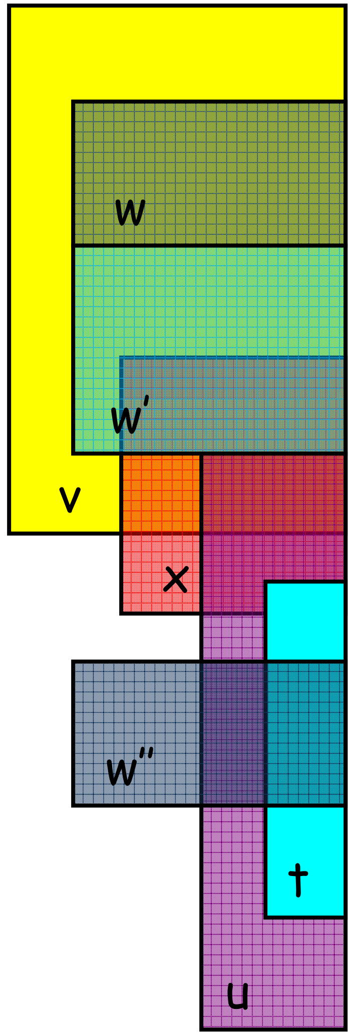
An Example, k=3

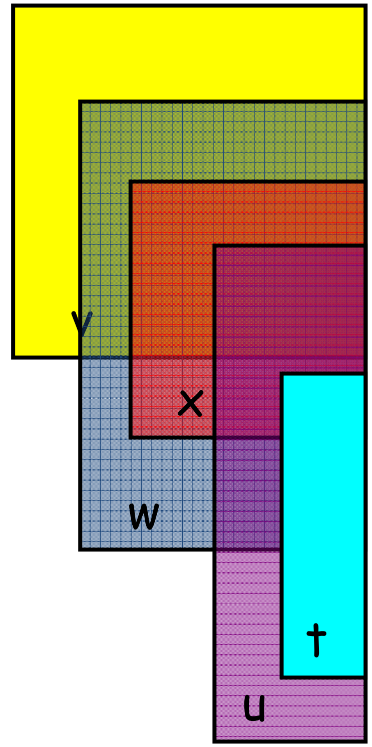
```
v <-   1
w <-   v + 3
M[]<-  w
w' <-  M[]
x <-   w' + v
u <-   v
t <-   u + x
w'' <- M[]
  <-   w''
  <-   t
  <-   u
```

Rewrite program

An Example, k=3 *(Old)*

```
v <-   1
w <-   v + 3
M[]<-  w
w' <-  M[]
x <-   w' + v
u <-   v
t <-   u + x
w'' <- M[]
  <-   w''
  <-   t
  <-   u
```



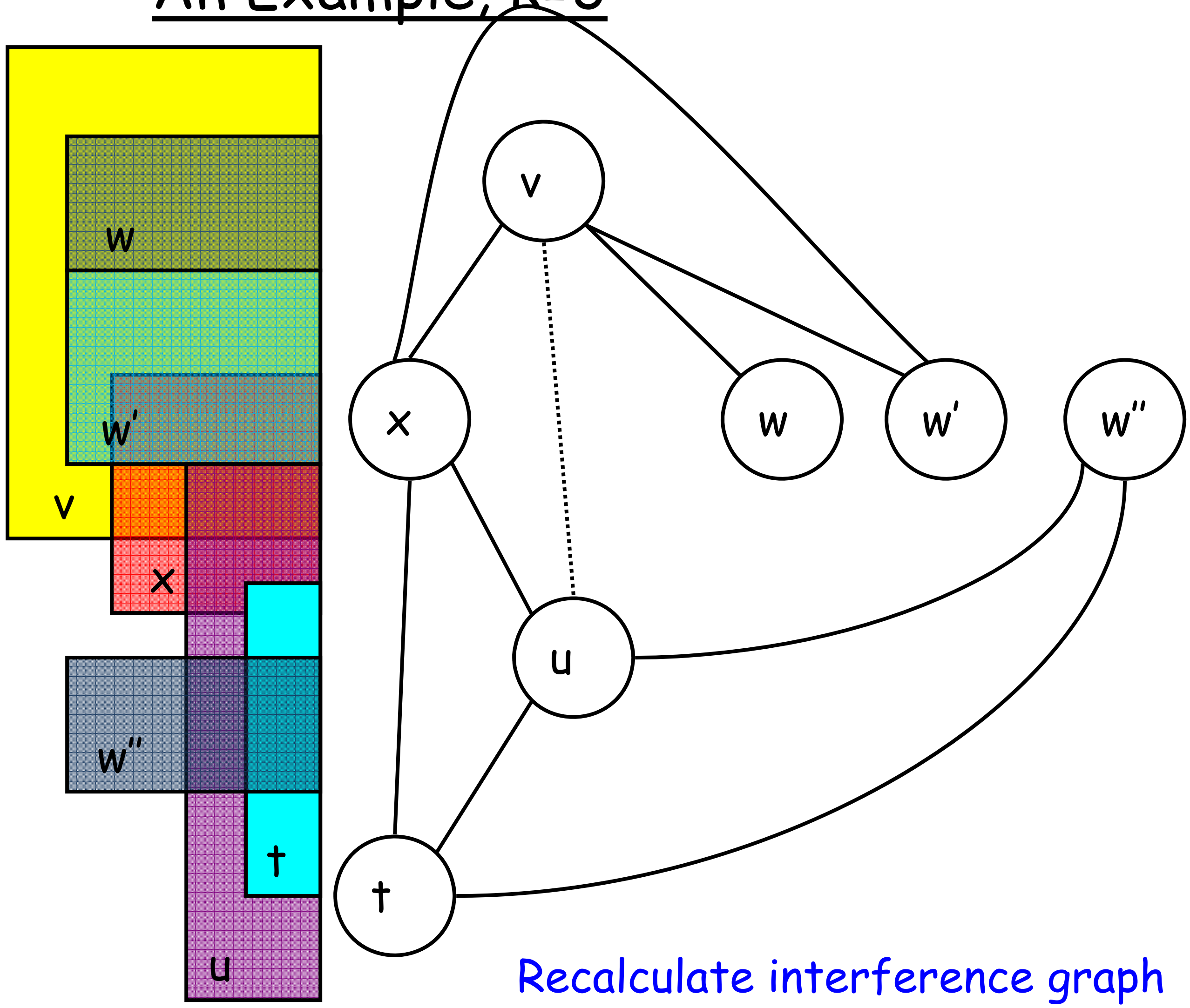


Recalculate live ranges

Spilling reduces live ranges, which decreases register pressure.

An Example, k=3

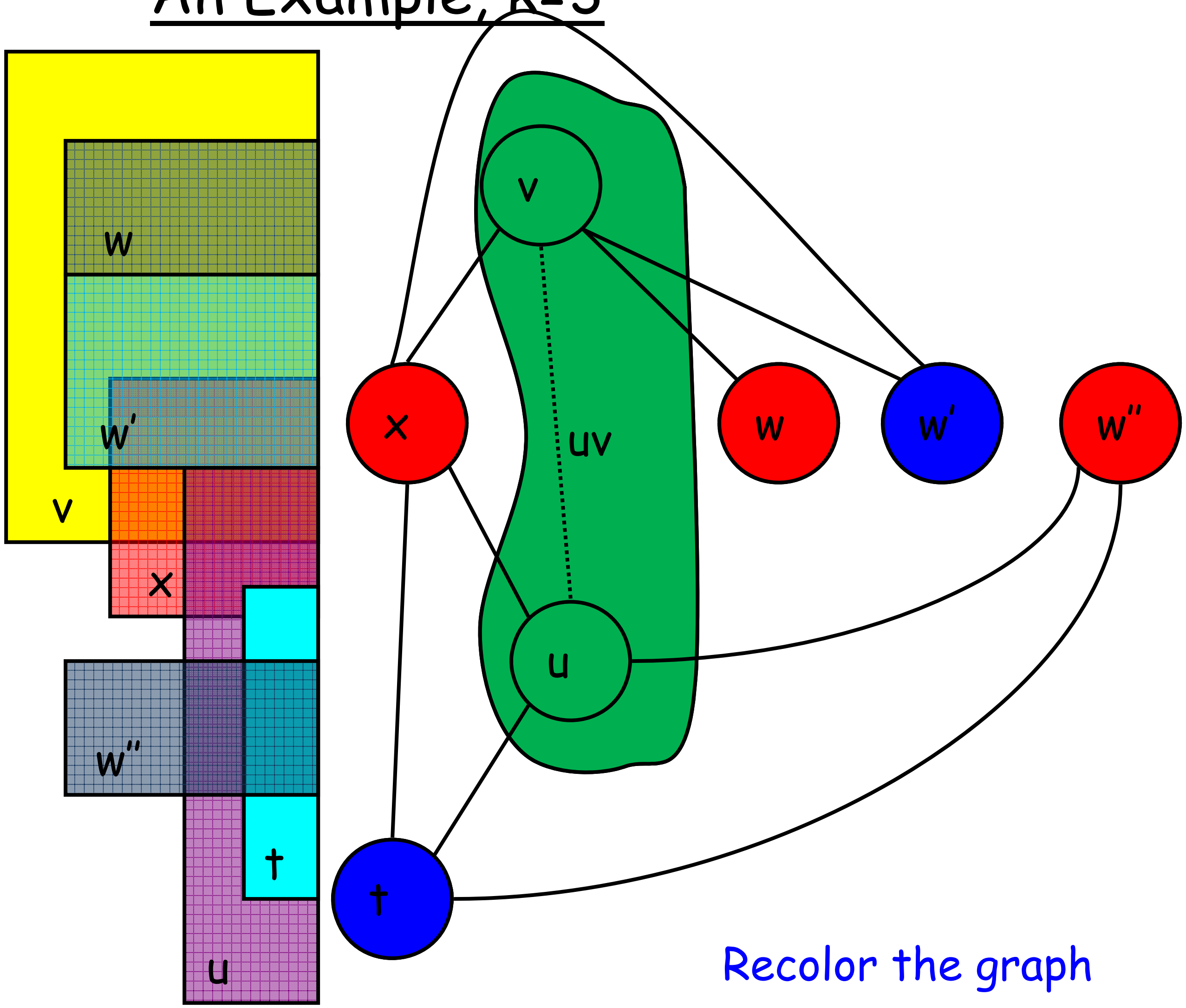
```
v <-   1
w <-   v + 3
M[]<-  w
w' <-  M[]
x <-   w' + v
u <-   v
t <-   u + x
w'' <- M[]
  <-   w''
  <-   t
  <-   u
```



Recalculate interference graph

An Example, k=3

```
v <-   1
w <-   v + 3
M[]<-  w
w' <-  M[]
x <-   w' + v
u <-   v
t <-   u + x
w'' <- M[]
  <-   w''
  <-   t
  <-   u
```



Recolor the graph

Things We Have Seen So Far

- Interference Graph
- Coalescing
- Coloring
- Spilling

General Plan

- Construct an interference graph
- Respect special registers:
  - avoid reserved registers
  - use registers properly
  - respect distinction between callee/caller save registers
- Map temporaries to registers
- Generate code to save & restore
- Deal with spills

Special Registers

- Which registers can be used?
  - Some registers have special uses.
    - Register 0 or 31 is often hardwired to contain 0.
    - Special registers to hold return address, stack pointer, frame pointer, global area, etc.
    - Reserved registers for operating system.
  - Typically, leaves about 20 or so registers for other general uses.
- Impact on register allocation:
  - Temps should be assigned only to the non-reserved registers.
  - Hard registers are pre-colored in the interference graph.

Register Usage Conventions

- Certain registers are used for specific purposes by standard calling convention.
  - 4-6 argument registers.
    - The first 4-6 arguments to procedures/functions are always passed in these registers.
  - ~8 callee-save registers.
    - These registers must be preserved across procedure calls. Thus, if a procedure wants to use a callee-save register, it must first save the old value and then restore it before returning.
  - The remainder are caller-save registers.
    - These are not preserved across procedure calls. Thus, a procedure is free to use them without saving first.
    - Includes the argument registers.

Spilling to Memory

- CISC architectures
  - can operate on data in memory directly
  - memory operations are slower than register operations
- RISC architectures
  - machine instructions can only apply to registers
  - Use
    - must first load data from memory to a register before use
  - Definition
    - must first compute RHS in a register
    - store to memory afterwards
  - Even if spilled to memory, needs a register at time of use/definition

Extending Coloring: Design Principles

- **A pseudo-register is**
  - Colored successfully: allocated a hardware register
  - Not colored: left in memory
- **Objective function**
  - Cost of an uncolored node:
    - proportional to number of uses/definitions (dynamically)
    - estimate by its loop nesting
  - Objective: minimize sum of cost of uncolored nodes
- **Heuristics**
  - Benefit of spilling a pseudo-register:
    - increases colorability of pseudo-registers it interferes with
    - can approximate by its degree in interference graph
  - Greedy heuristic
    - spill the pseudo-register with lowest cost-to-benefit ratio, whenever spilling is necessary

Coloring Algorithm (Without Spilling)

Build interference graph

Iterate until there are no nodes left:

  If there exists a node v with less than n neighbors

    place v on stack to register allocate

  else

    return (coloring heuristics fail)

  remove v and its edges from graph

While stack is not empty

  Remove v from stack

  Reinsert v and its edges into the graph

  Assign v a color that differs from all its neighbors

Chaitin: Coloring and Spilling

- **Identify spilling**

  Build interference graph
  Iterate until there are no nodes left
      If there exists a node v with less than n neighbor
          place v on stack to register allocate
      else
          v = node with highest degree-to-cost ratio
          mark v as spilled
      remove v and its edges from graph

- **Spilling may require use of registers; change interference graph**

  While there is spilling
      rebuild interference graph and perform step above

- **Assign registers**

  While stack is not empty
      Remove v from stack
      Reinsert v and its edges into the graph
      Assign v a color that differs from all its neighbors

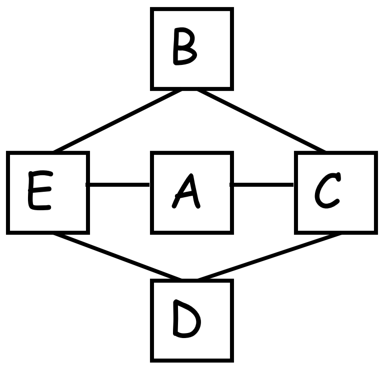
Spilling

- What should we spill?
  - Something that will eliminate a lot of interference edges
  - Something that is used infrequently
  - Maybe something that is live across a lot of calls?

- One Heuristic:
  - spill cheapest live range (aka "web")
  - Cost = $[(\#\ \text{defs} \ \&\ \text{uses}) * 10^{\text{loop-nest-depth}}]/\text{degree}$

Quality of Chaitin's Algorithm

- Giving up too quickly



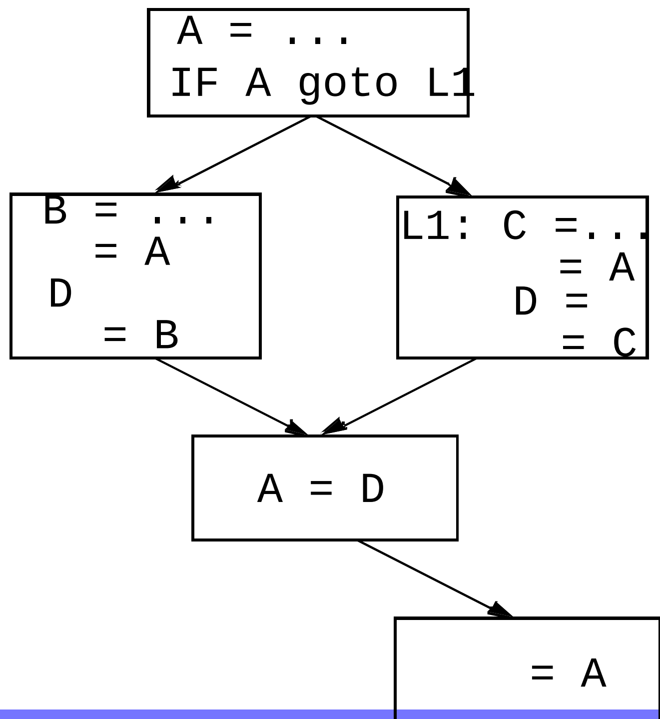
- An optimization: "Prioritize the coloring"
  - Still eliminate a node and its edges from graph
  - Do not commit to "spilling" just yet
  - Try to color again in assignment phase.

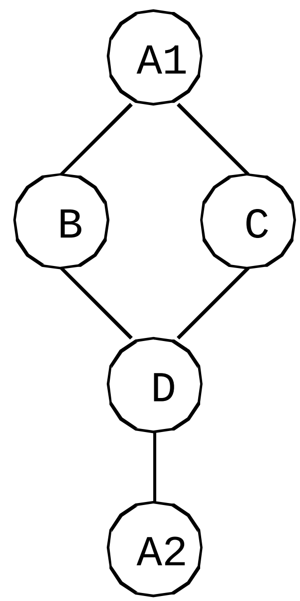
Setting Up For Better Spills

- We want variables that are not live across procedures to be allocated to caller-save registers. Why?
- We want variables live across many procedures to be in callee-save registers
- We want live ranges of pre-colored nodes to be short!
- We prefer to use callee-save registers last.

Splitting Live Ranges

- Recall: Split pseudo-registers into live ranges to create an interference graph that is easier to color
  - Eliminate interference in a variable's "dead" zones.
  - Increase flexibility in allocation:
    - can allocate same variable to different registers





Insight

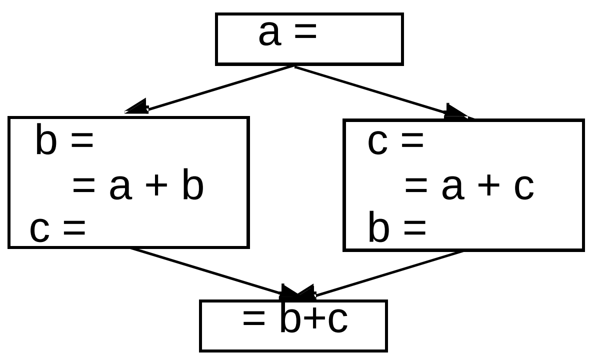
- Split a live range into smaller regions (by paying a small cost) to create an interference graph that is easier to color
  - Eliminate interference in a variable's "nearly dead" zones.
    - Cost: Memory loads and stores
      - Load and store at boundaries of regions with no activity
    - # active live ranges at a program point can be > # registers
  - Can allocate same variable to different registers
    - Cost: Register operations
      - a register copy between regions of different assignments
    - # active live ranges cannot be > # registers

Examples

Example 1:

```
FOR i = 0 TO 10
   FOR j = 0 TO 10000
      A = A + ...
      (does not use B)
   FOR j = 0 TO 10000
      B = B + ...
      (does not use A)
```

Example 2:



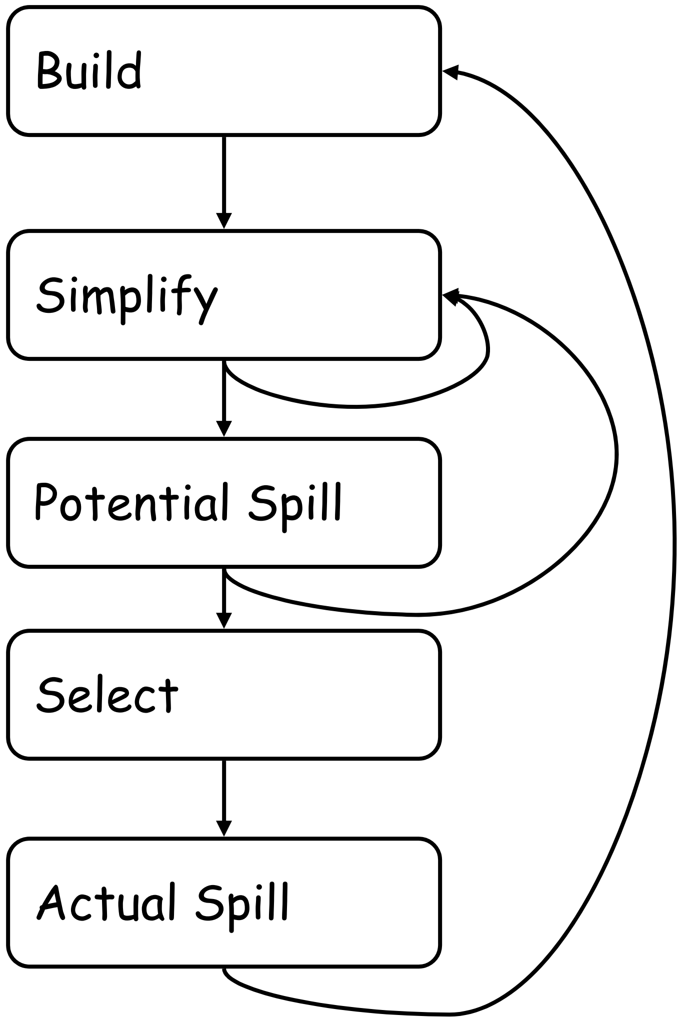
Live Range Splitting

- When do we apply live range splitting?
- Which live range to split?
- Where should the live range be split?
- How to apply live-range splitting with coloring?
  - Advantage of coloring:
    - defers arbitrary assignment decisions until later
  - When coloring fails to proceed, may not need to split live range
    - degree of a node >= n does not mean that the graph definitely is not colorable
  - Interference graph does not capture positions of a live range

One Algorithm

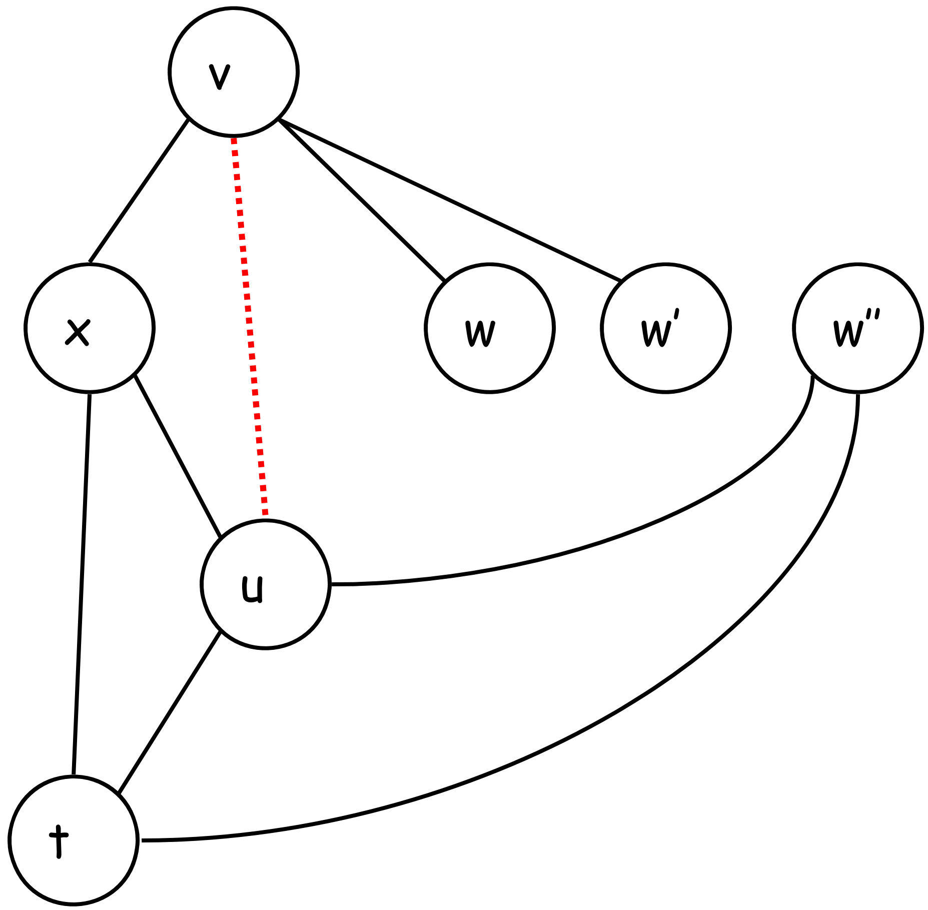
- Observation: spilling is absolutely necessary if
  - number of live ranges active at a program point > n
- Apply live-range splitting before coloring
  - Identify a point where number of live ranges > n
  - For each live range active around that point:
    - find the outermost “block construct” that does not access the variable
  - Choose a live range with the largest inactive region
  - Split the inactive region from the live range

The Big Picture



Coalescing

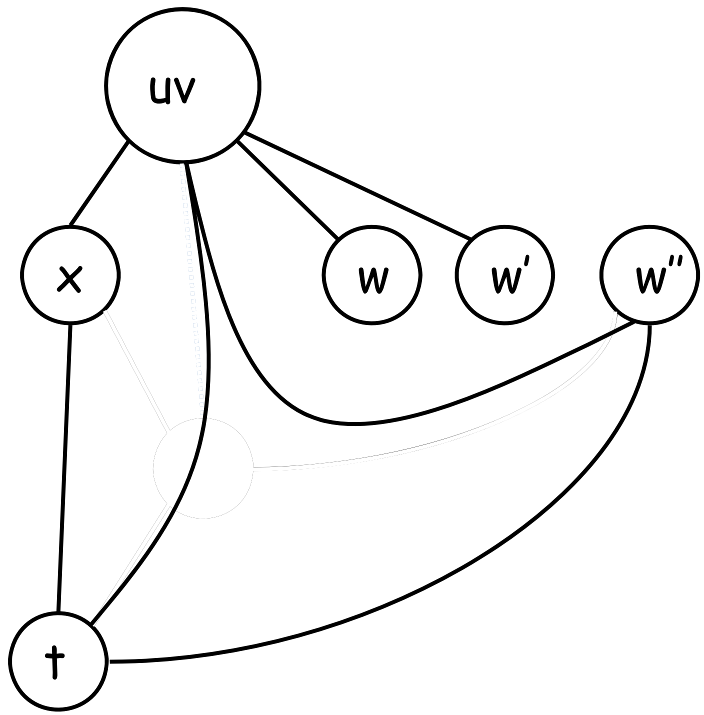
```
v <-   1
w <-   v + 3
M[]<-  w
w' <-  M[]
x <-   w' + v
u <-   v
t <-   u + x
w'' <- M[]
  <-   w''
  <-   t
  <-   u
```



Can u & v be coalesced?
Should u & v be coalesced?

Briggs: Conservative Coalescing

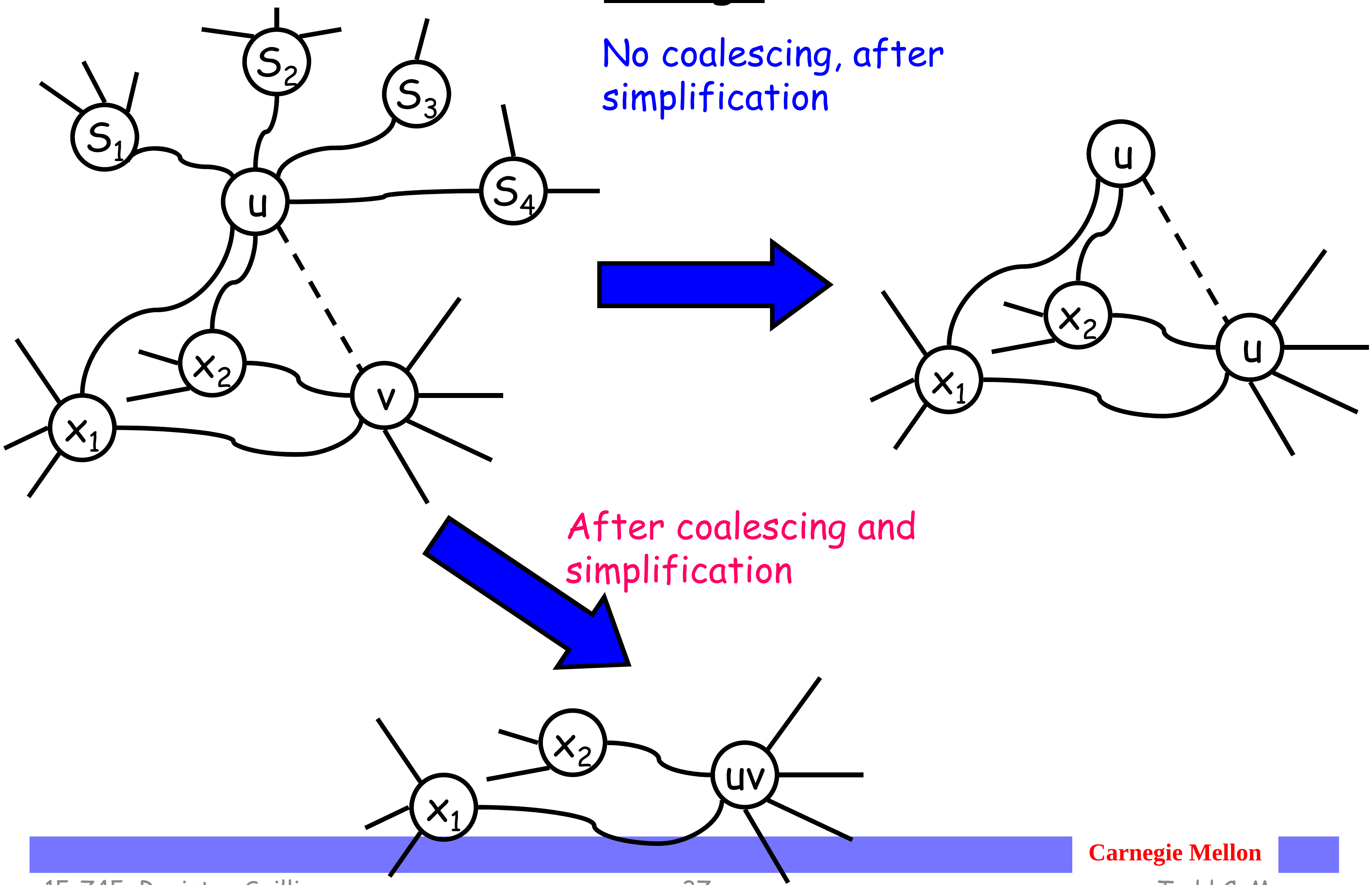
- Can coalesce u and v if:
  - (# of neighbors of uv with degree ≥ k) < k
- Why?
  - Simplify pass removes all nodes with degree < k
  - # of remaining nodes < k
  - Thus, uv can be simplified



George: Iterated Coalescing

- Can coalesce u and v if:
  - foreach neighbor t of u, either:
    - t interferes with v, or,
    - degree of t < k
- Why?
  - let S be set of neighbors of u with degree < k
  - If no coalescing, simplify removes all nodes in S; call that graph G_1
  - If we coalesce, we can still remove all nodes in S; call that graph G_2
  - G_2 is a subgraph of G_1

George

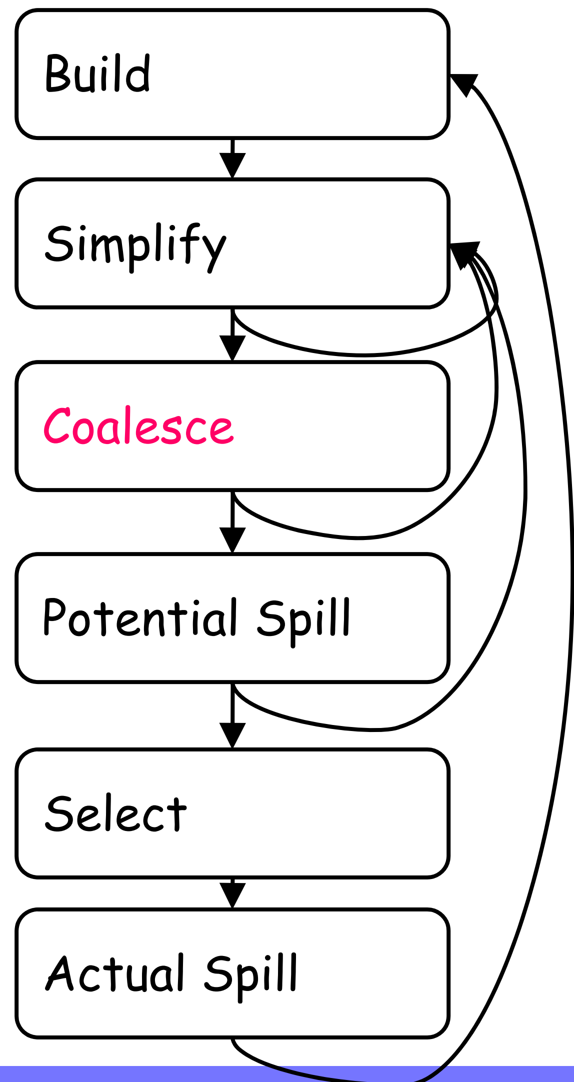


No coalescing, after simplification
After coalescing and simplification
S1
S2
S3
S4
u
x2
v
x1
u
x2
u
x1
x2
uv
x1

Why Two Methods?

- With Briggs, one needs to look at all neighbors of a & b
- With George, only need to look at neighbors of a.
- We need to insert hard registers in graph and they will have LARGE adjacency lists.
- Hence:
  - Precolored nodes have infinite degree
  - No other precolored nodes in adjacency list
  - Use George if one of a & b is precolored
  - Use Briggs if both are temps

Where We Are


Build
Simplify
Coalesce
Potential Spill
Select
Actual Spill