Register Allocation:

Local and Global Allocations

Register Allocation

- Allocation and Assignment
- Local methods
- Global Allocation (Graph Coloring)
  - Webs – allocatable objects
  - Interference graph
  - Grpah pruning
  - Spilling
  - Priority based coloring
- New directions
  - Speculative Register Allocation

Register Allocation

Allocation: allocate frequently used objects to virtual (or pseudo) registers.
e.g.

frequently used local variables

CSE, special pointers such as $sp,$fp,$gp

Assignment: assign unlimited virtual registers to the limited physical registers

However, in compiler context, many register allocation algorithms are actually doing register assignments

Early Allocation Approach

- *Local* register allocation can be performed near optimal (replacement based, consider *next-use* information)
- *Local* allocation alone is insufficient, since there will be numerous saving/restoring at block boundaries. One simple extension is to allocate a dedicated register to an object.
- Some early *global* allocation strategies:
  - Loop nesting level based heuristics
  - Usage count based heuristics

Loop nesting level based heuristics

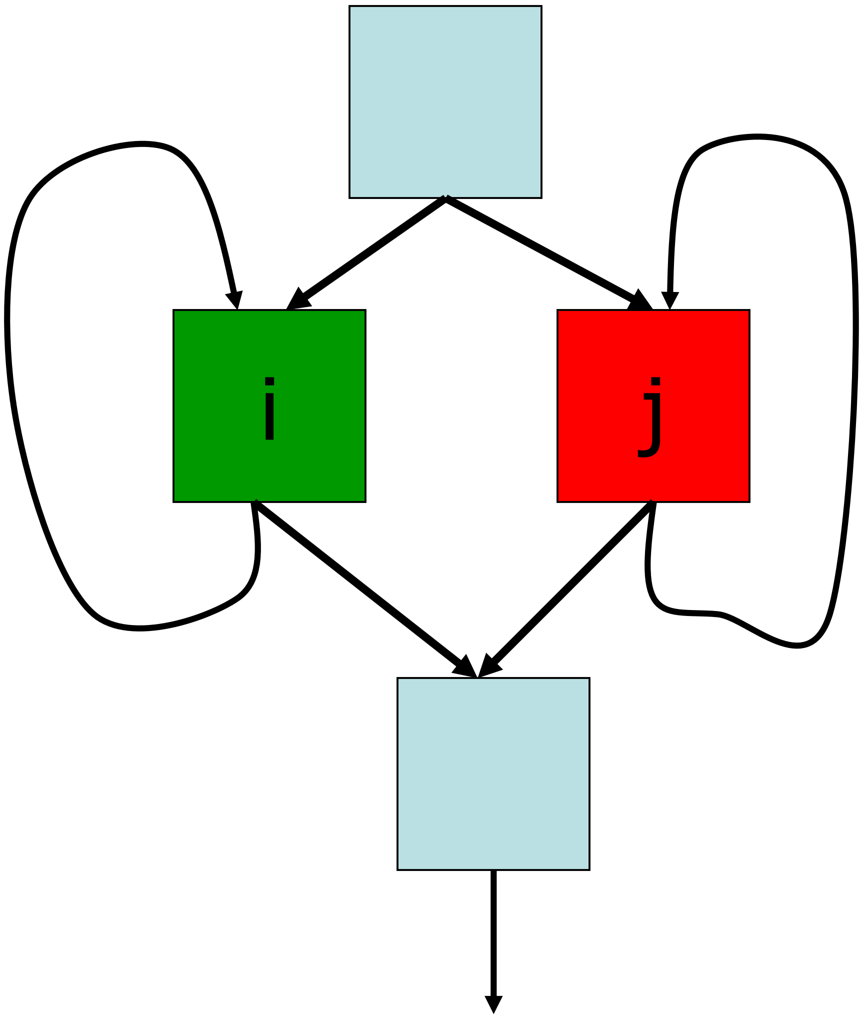
Cost model:

net_save(*v*,*i*): net saving of allocating a particular variable *v* in block *i*,

if block *i* is in a loop L, then increase the saving by 10*net_save(*v*,*i*).

Allocate the limited available registers to the objects with the greatest estimated saving.

Limitation of Old Methods



i and j could share the same register.

Need a way to allow objects with *disjoint* lifetime to share registers!

Graph Coloring

- If we model each Pseudo-Register (PR) as a node, and any two PRs that may live at the same time is connected by an arc, then register allocation becomes the graph coloring problem.
- Global register allocation has long been known as graph coloring problem (which is NP-hard). However, efficient heuristics have not been invented to make graph coloring algorithm practical until early 1980s.

Steps in Graph Coloring RA

- During code generation, use as many PRs as possible (e.g. each var assigned a PR).
- Determine what objects should also be candidates for registers. (e.g. web is better than PR)
- Construct an *interference graph* (based on liveness analysis).
- Color the graph with R colors, where R is the number of available registers
- Allocate each object to the register that has the same color.

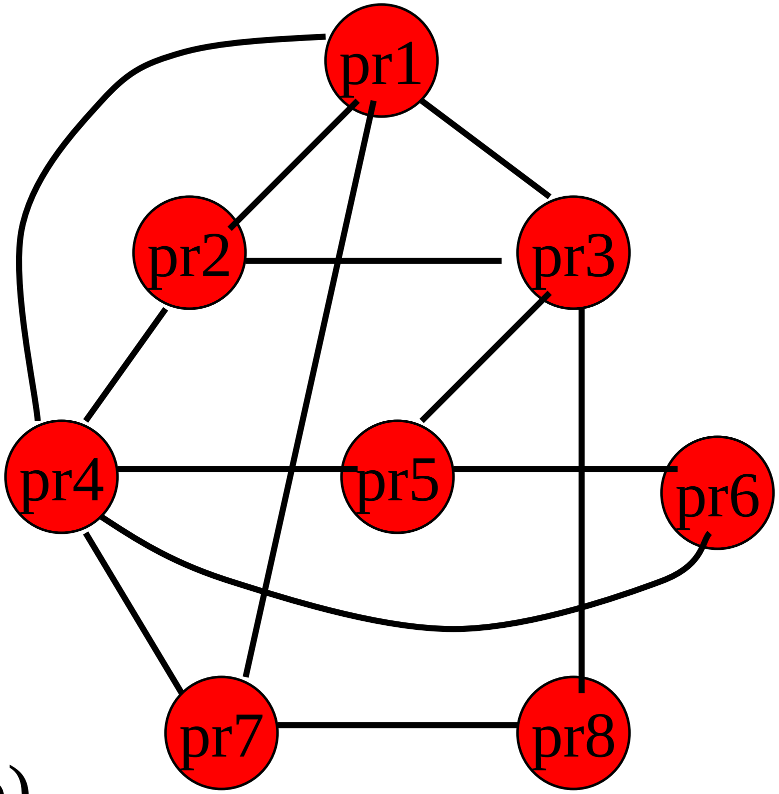
Coloring Register Allocation

- Graph coloring problems derive from the old mapmakers' rule that adjacent countries on a map should be colored with different colors where using as few colors as possible is the NP-hard problem.
- In practice, we simply want to know if the graph is *R*-colorable, where *R* is the number of registers available.
- Chaitin contributes to answering the question "is the graph R-colorable?"

Simple Example

A harder question:
what is the minimum number of colors required for coloring this graph?

An easier question:
Is this graph R-colorable?
Yes or No?
(Yes or Not-Yes – maybe or no)



Is a Graph R-colorable?

Graph Pruning Theory

Degree < R rule:

Given a graph that contains a node with degree less than R, the graph is R-colorable *iff* the graph without that node is R-colorable.

Implementation:

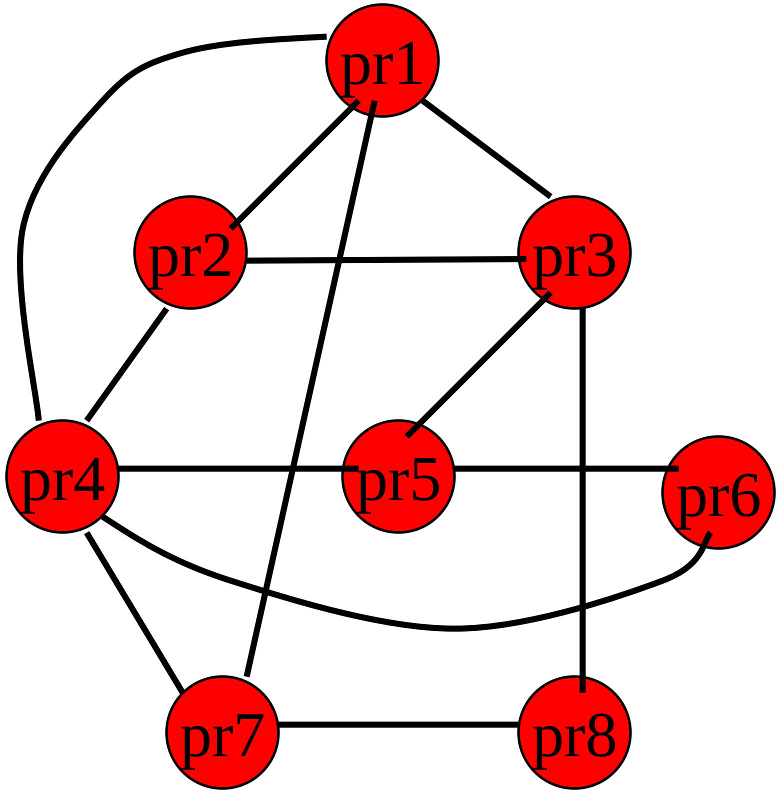
We repeatedly search the graph for nodes that have fewer than R neighbors and remove them. If we exhaust the graph, we have determined that R-coloring is possible for the graph.

Simple Example

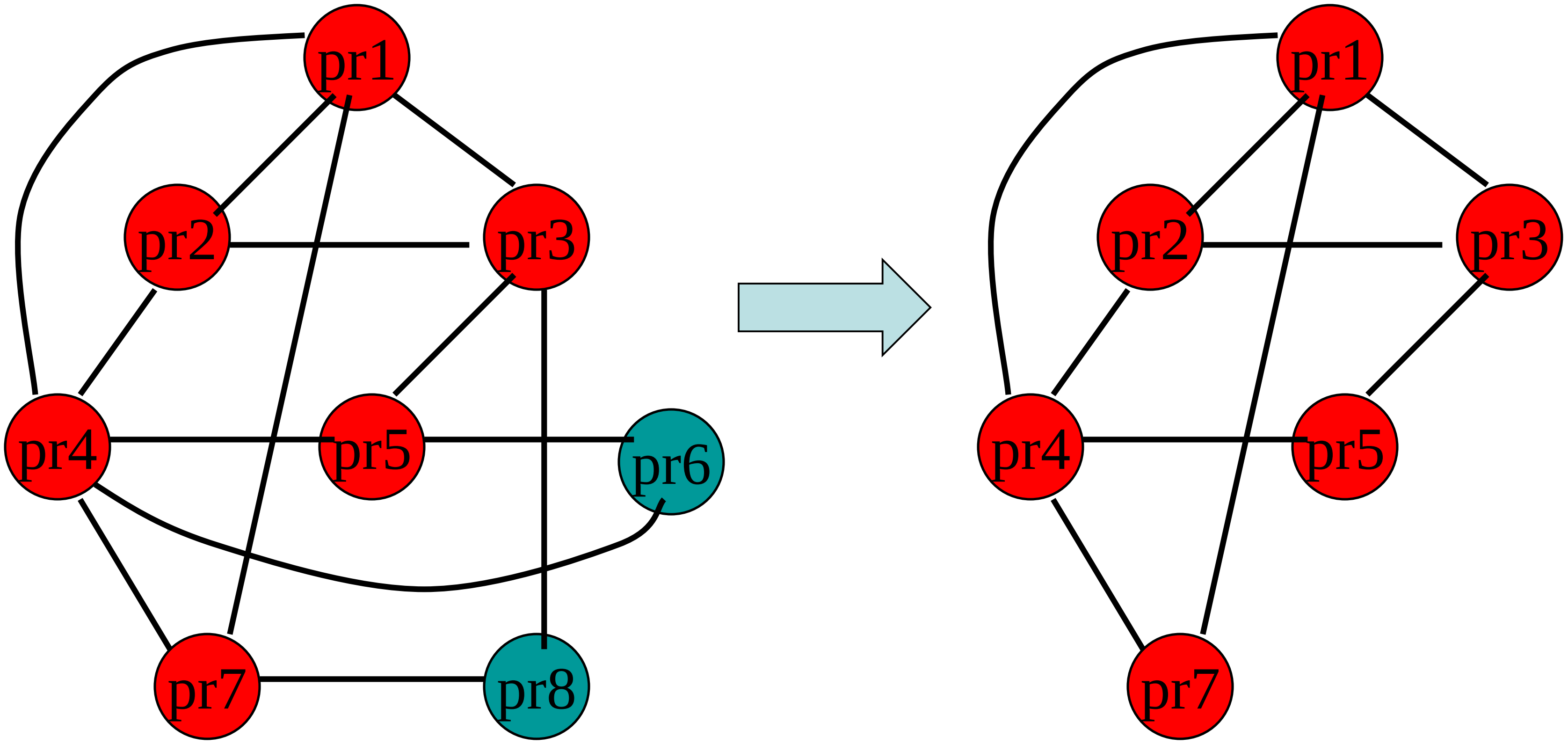
Is this graph 3-colorable?

The answer is “Yes”

But why?

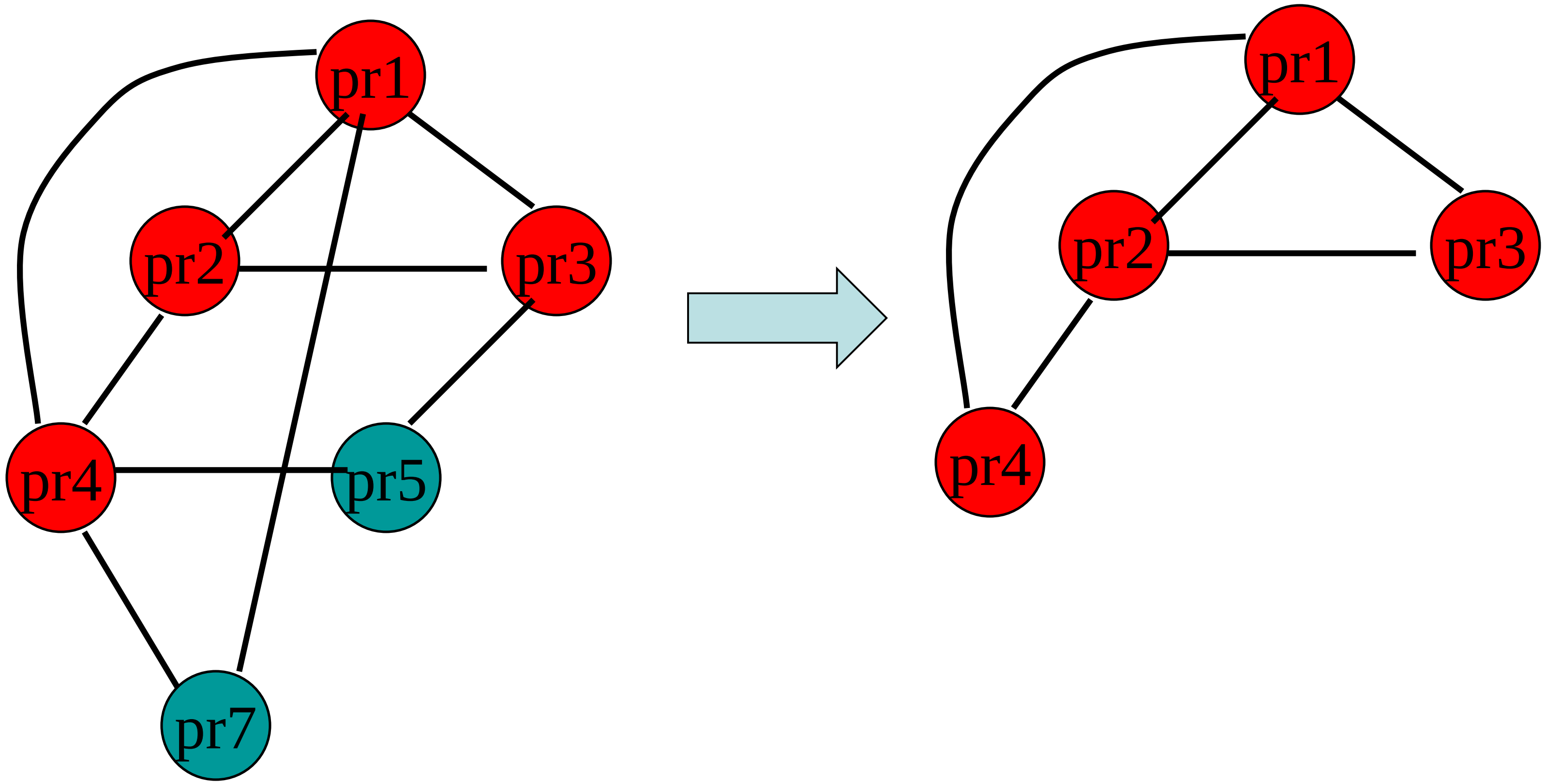


Pruning Process (3-coloring)



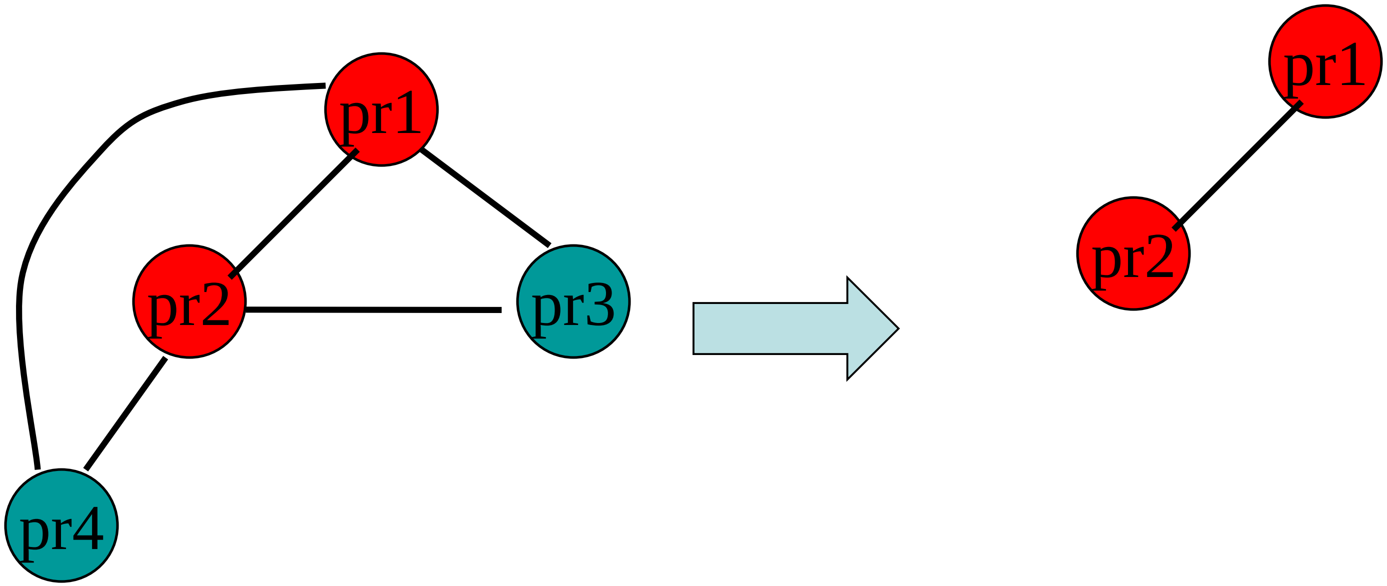
Stack: pr8,pr6

Pruning Process (3-coloring)



Stack: pr8,pr6,pr7,pr5

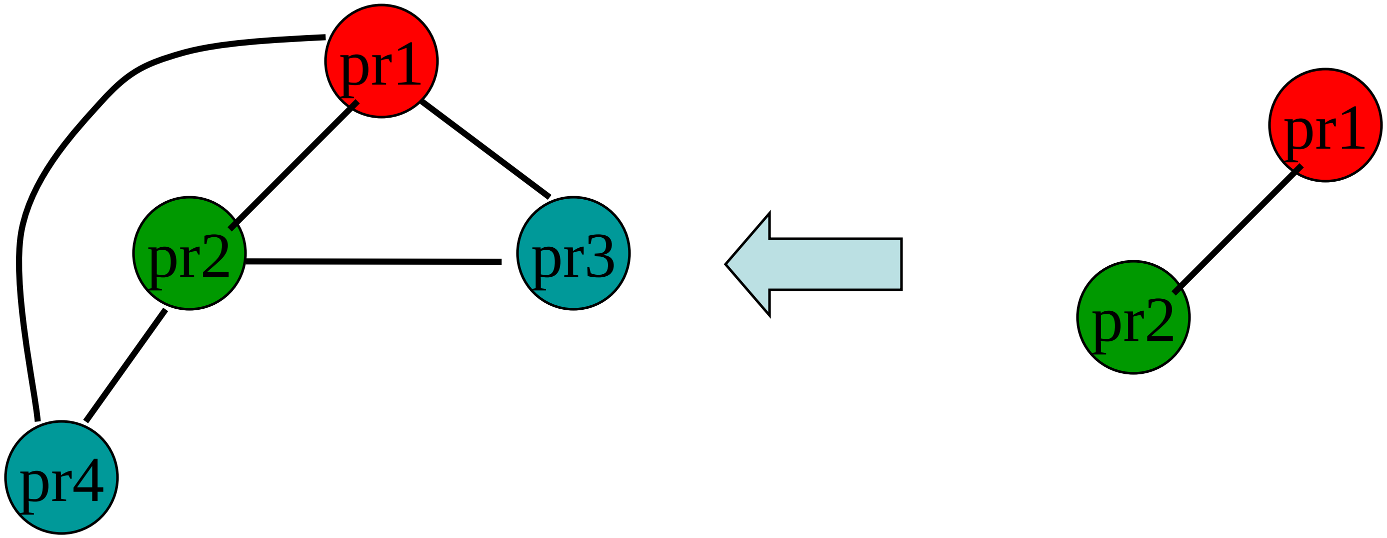
Pruning Process (3-coloring)



Stack: pr8,pr6,pr7,pr5,pr4,pr3,pr2,pr1

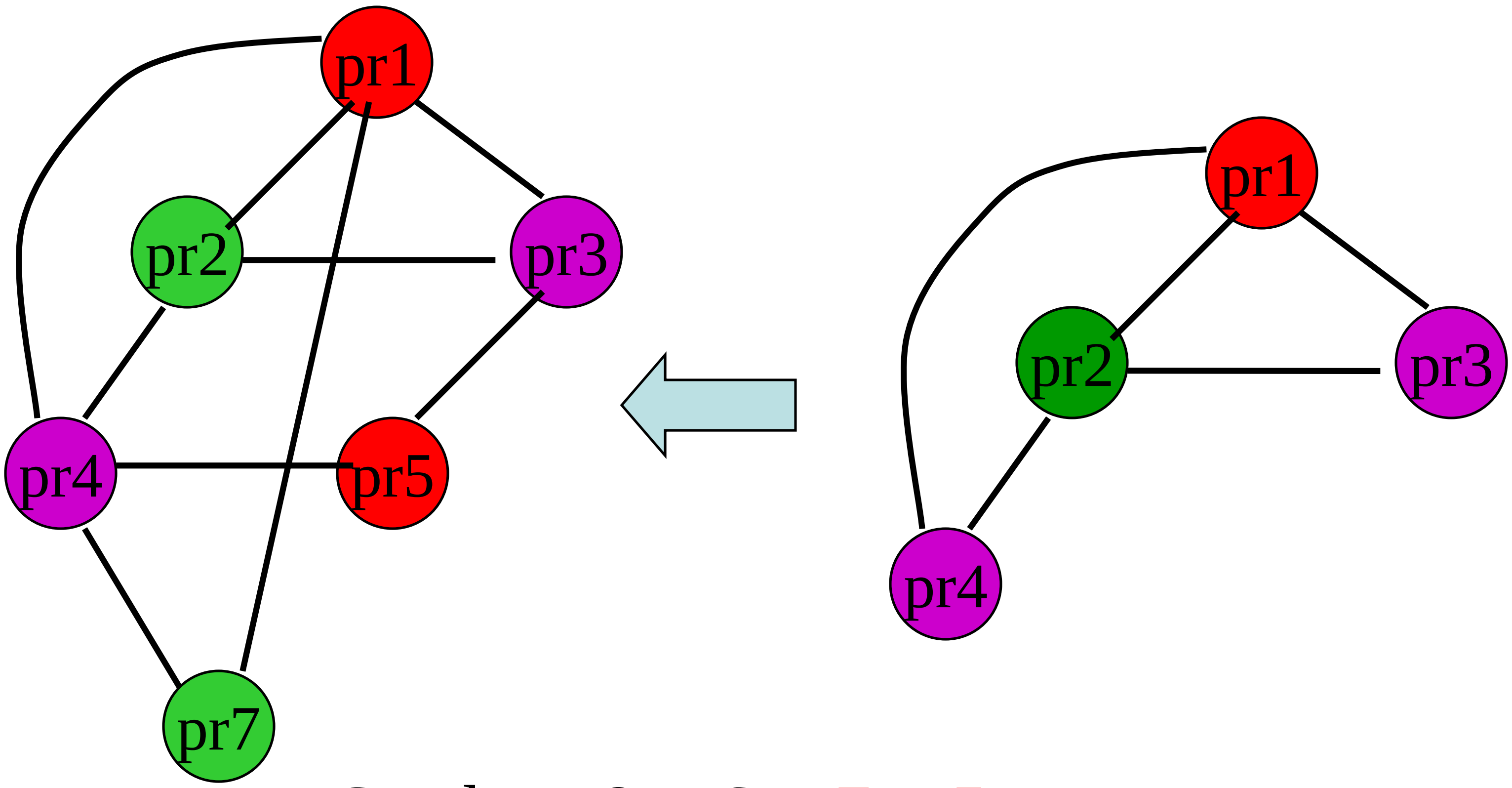
Coloring Process

Assume three available registers, i.e.
three colors: red ($8), green ($9), purple ($10)



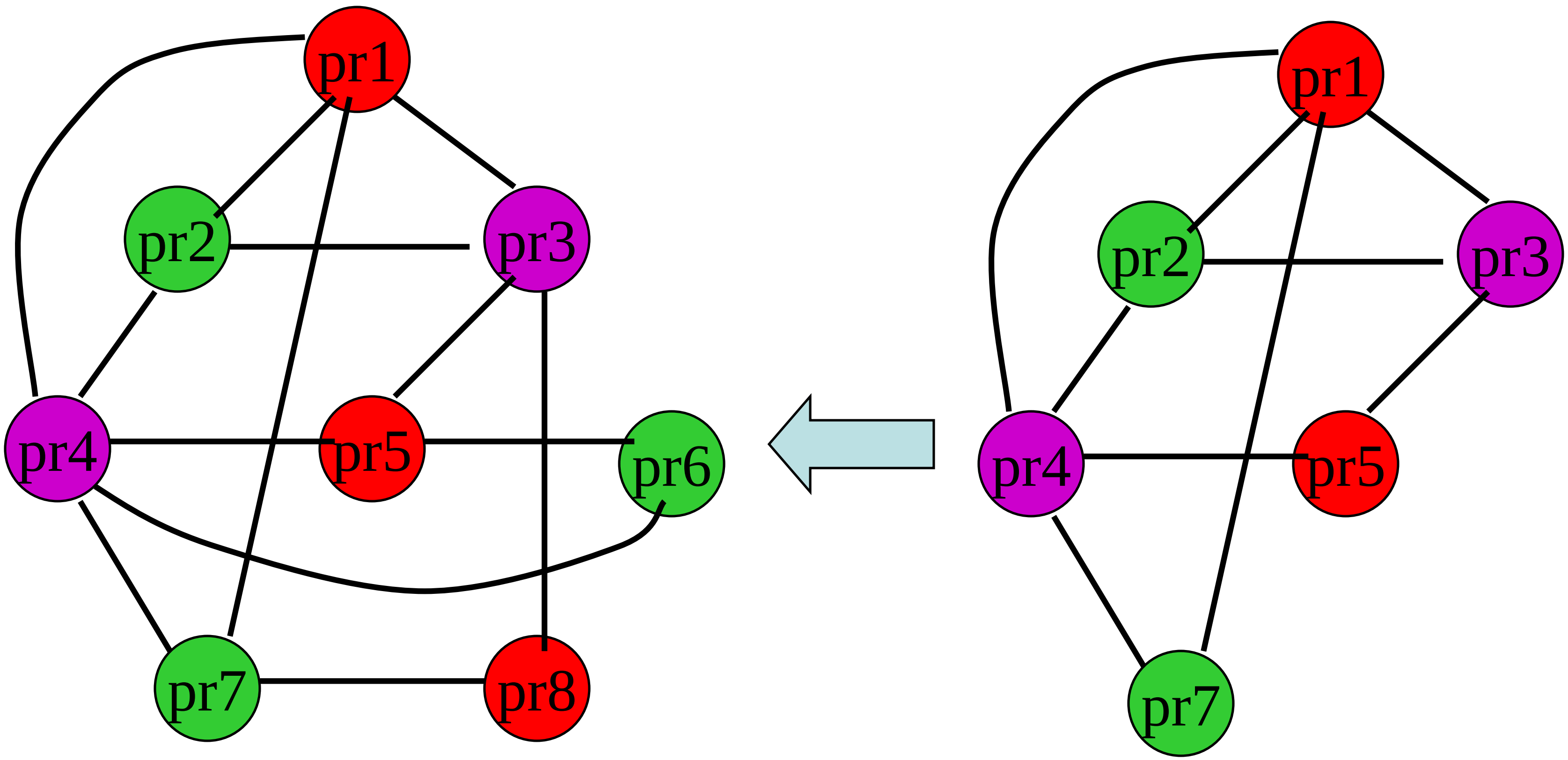
Stack: pr8,pr6,pr7,pr5,pr4,pr3,pr2,pr1

Coloring Process



Stack: pr8,pr6,pr7,pr5

Coloring Process

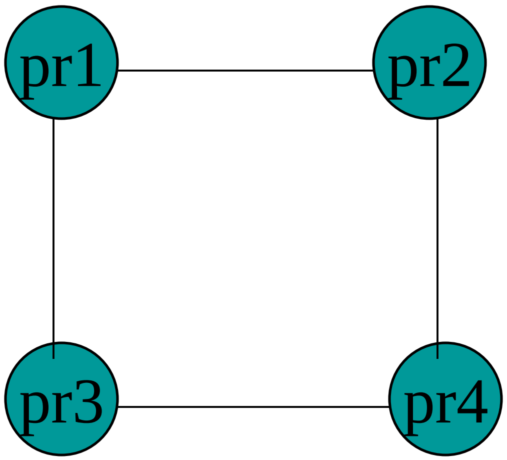


Stack: pr8,pr6

Minimum colors vs. R-colorable

While it is easy to answer the R-colorable question, the compiler may use more registers than it should. (*especially when the architecture provides a large number of registers*)

Instead of asking "is the graph 128 colorable, the compiler needs to try 32-colorable? then 16-colorable? 8-colorable….



Is this graph 2-colorable?

Yes, but our pruning process would say "NO"

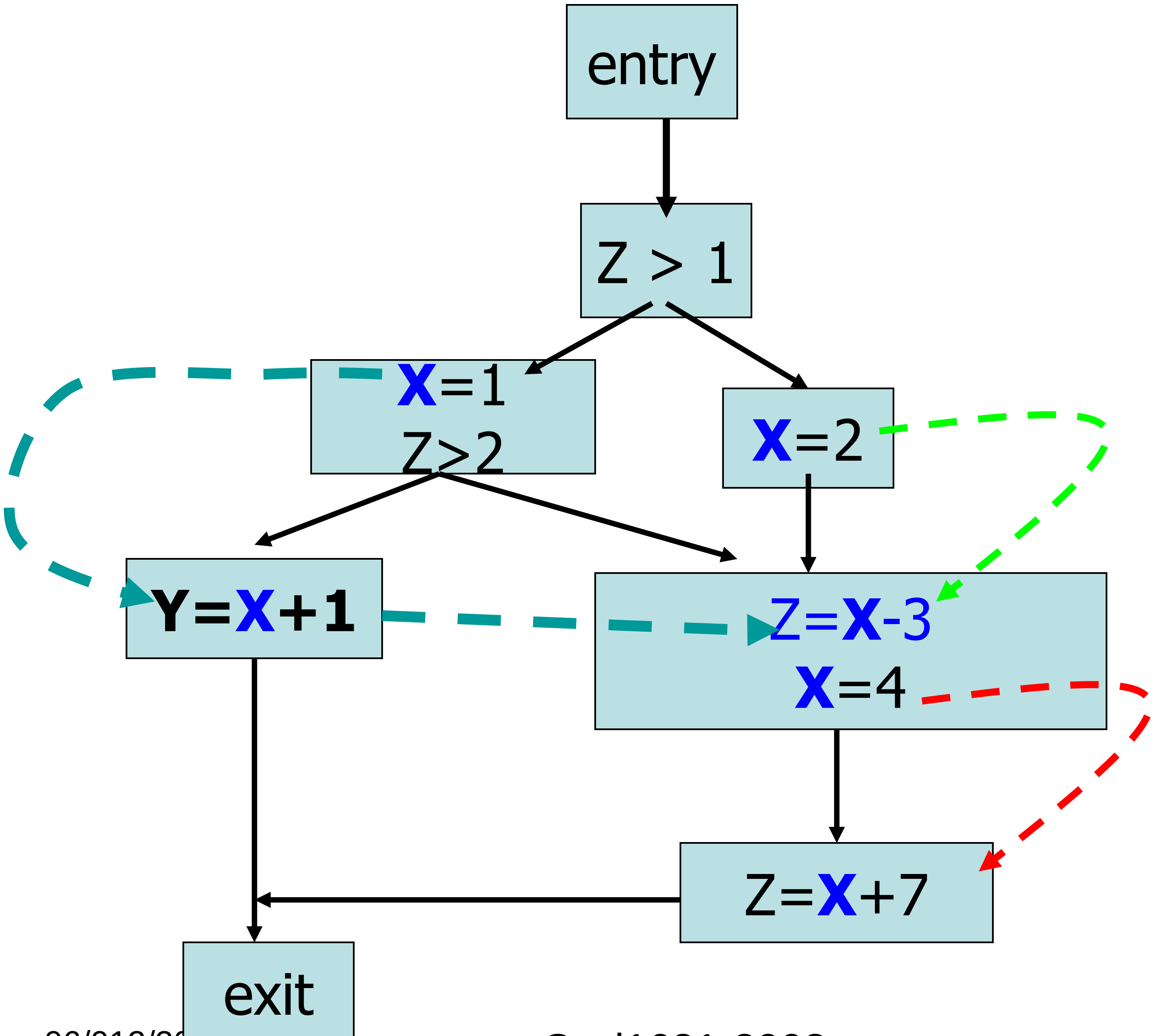
Interference: Liveness Analysis

- A variable is *live* if it holds a value that may be needed in the future.
- If two temporaries (or variables) a, b are never "in use" at the same time, they can be allocated to the same register.
- The *Liveness analysis* is also a typical *data flow analysis* problem.
- Definition of liveness: A variable is live on an edge if there is a directed path from that edge to a *use* of the variable that does not go through any *def.*

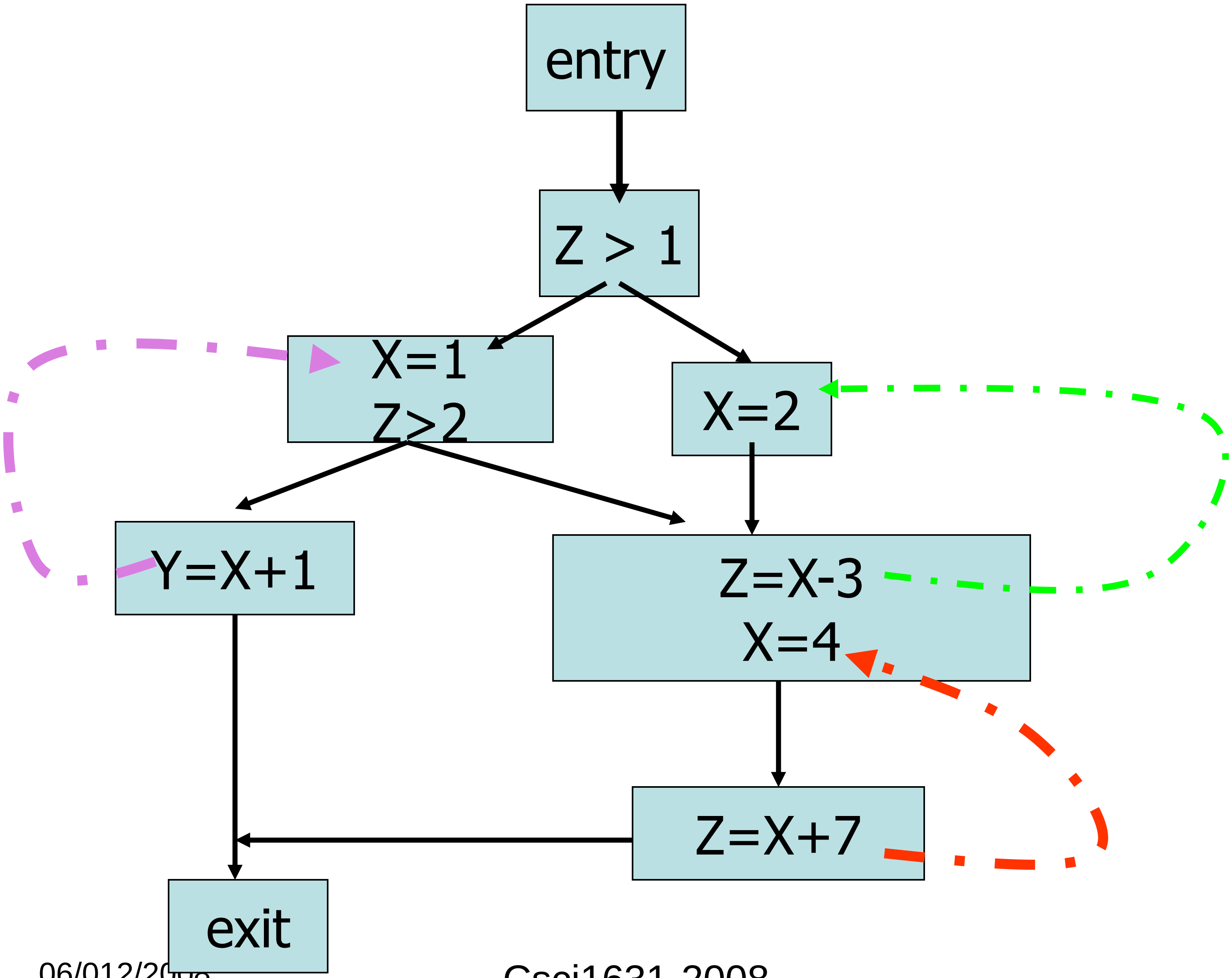
Du-Chains, Ud-chains

- DU and UD chains are a sparse representation of data flow information about variables
- A DU chain for a variable connects a definition of that variable to all the uses it may flow to.
- A UD chain connects a use to all the definitions that may flow to it

Du-Chains

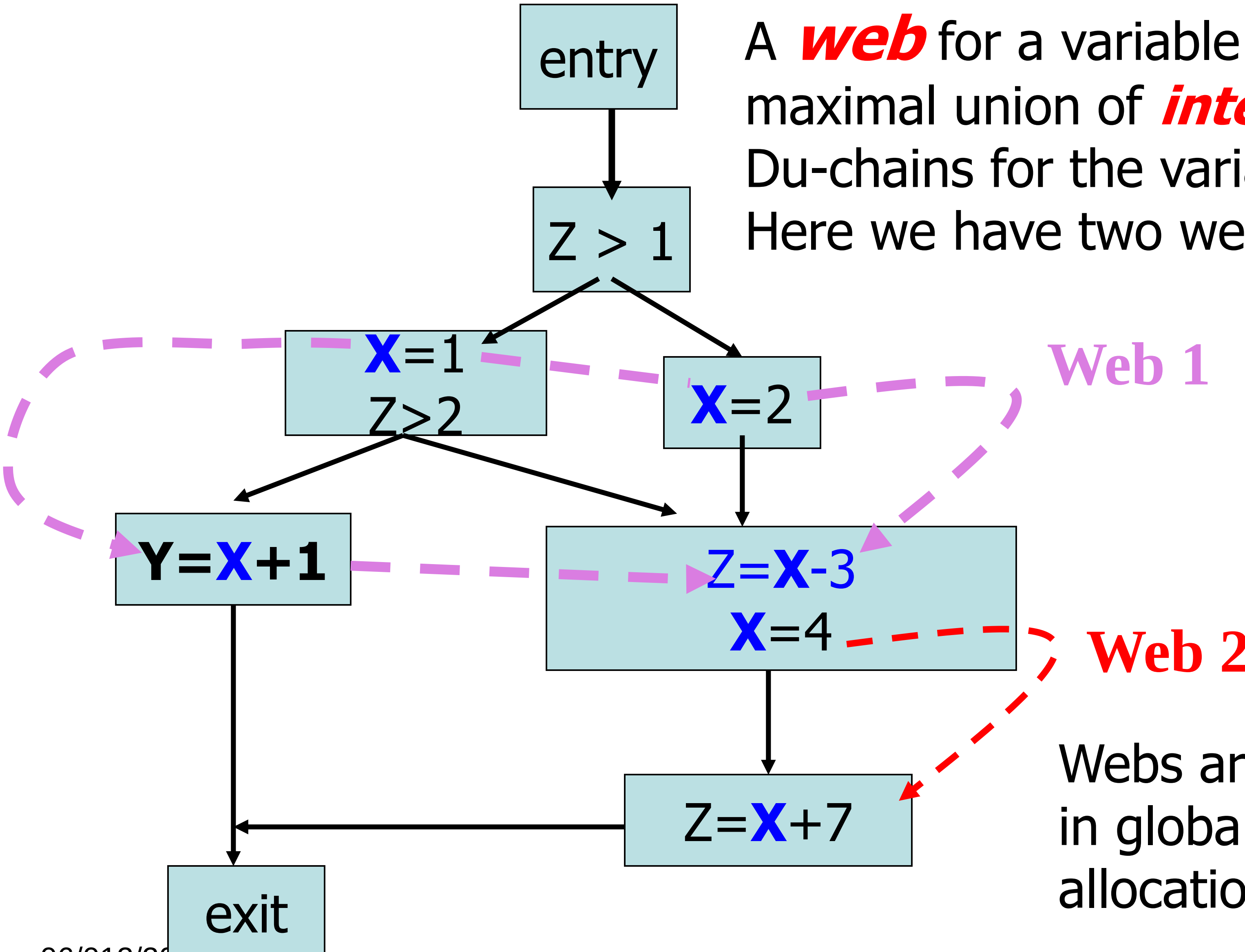


Ud-Chains



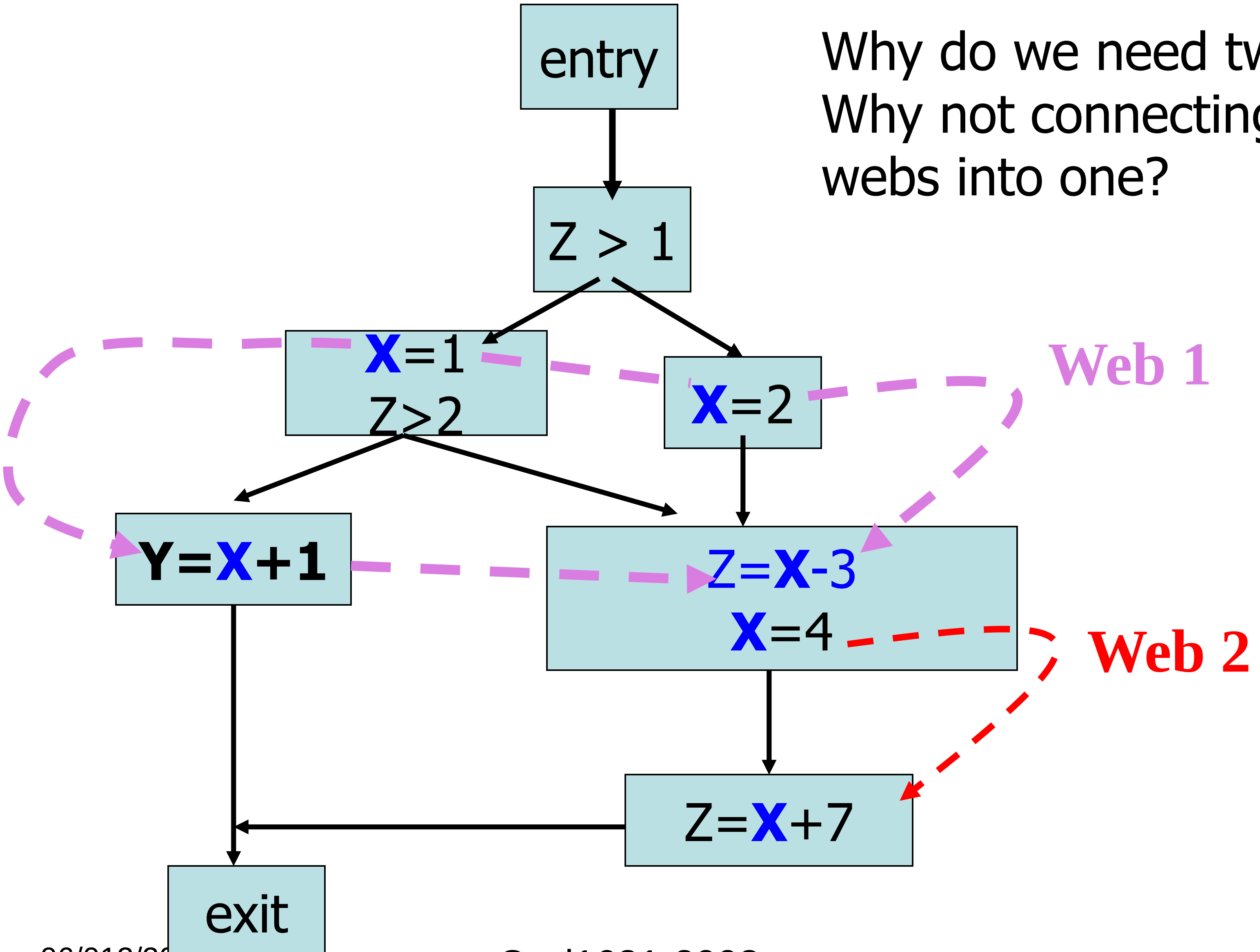
Webs

A ***web*** for a variable is the maximal union of ***intersecting*** Du-chains for the variable.
Here we have two webs for X



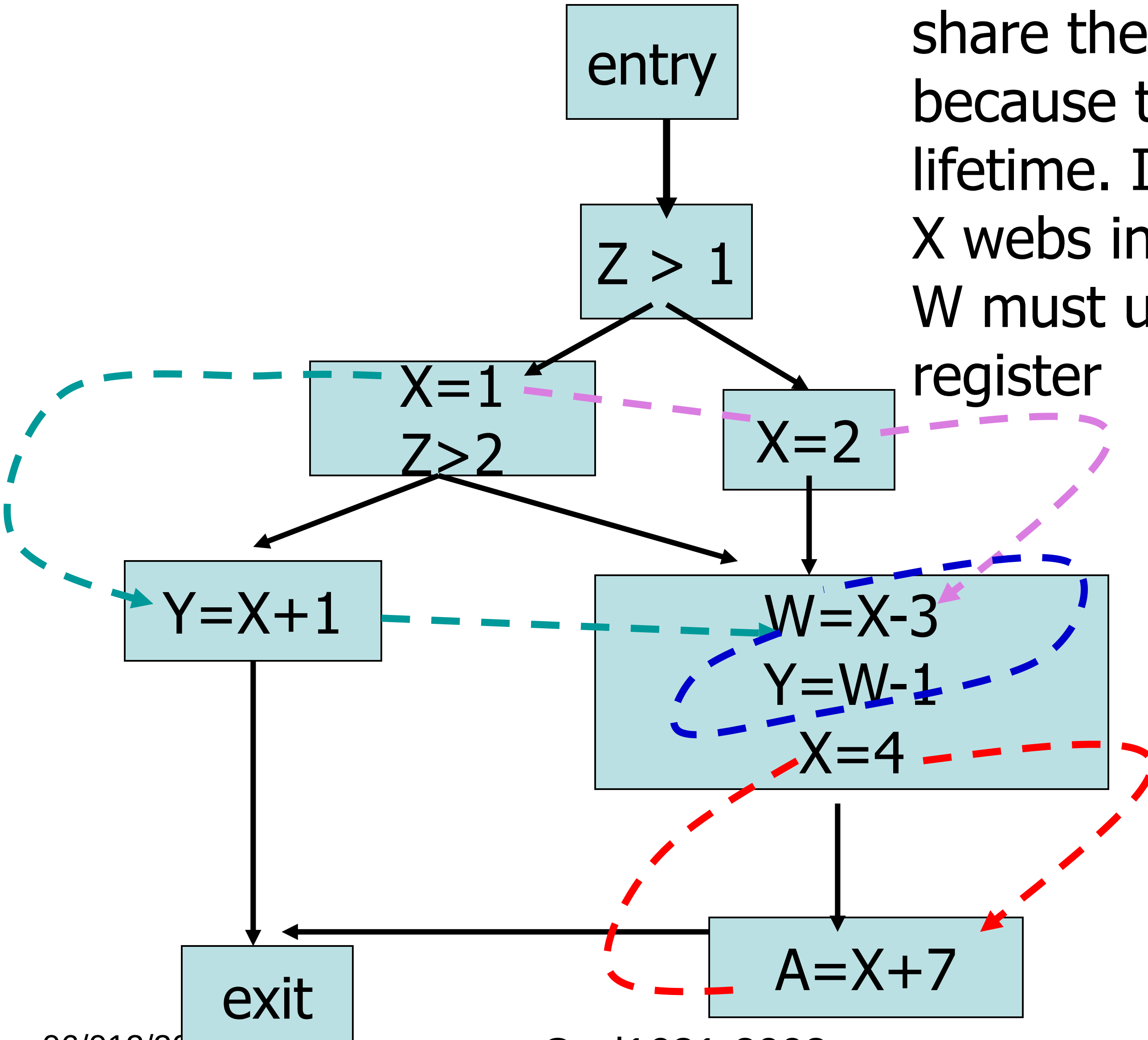
Webs are useful in global register allocation.

Webs



Why do we need two webs? Why not connecting the two webs into one?

Webs



The three webs can share the same register because they have disjoint lifetime. If we connect two X webs into one, the variable W must use a different register

Representing the Interference-graph

- Graph may be quite large
- Access time may be an issue
- How to represent an arc?
- How many nodes are adjacent to a given one?

Using an adjacency matrix and adjacency lists

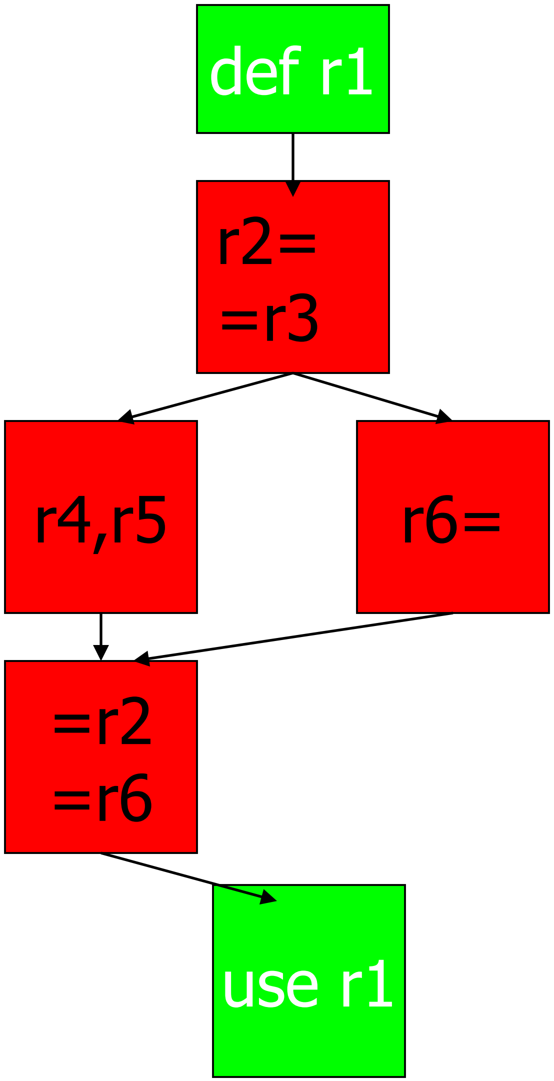
	web1	web2	…	webn
web1		x		
web2	x			
…				x
webn				

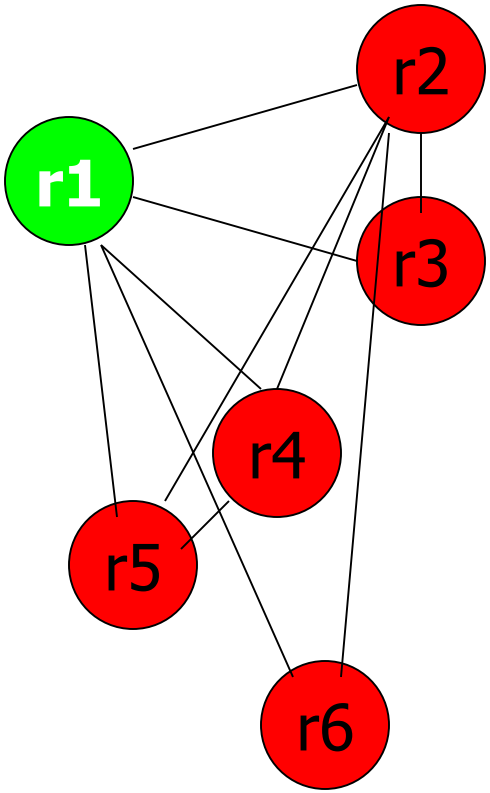
web1 and web2 are live simultaneously (or adjacent)

Representing the I-graph

- Using an adjacency matrix and adjacency lists
- Adjacency matrix can be used to answer "Does node i interfere with node j ?"
- Adjacency list is an array of records that tracks of the following information
  - the register (color) assigned to this node
  - the spill cost
  - the number of interferences
  - the list of interferences

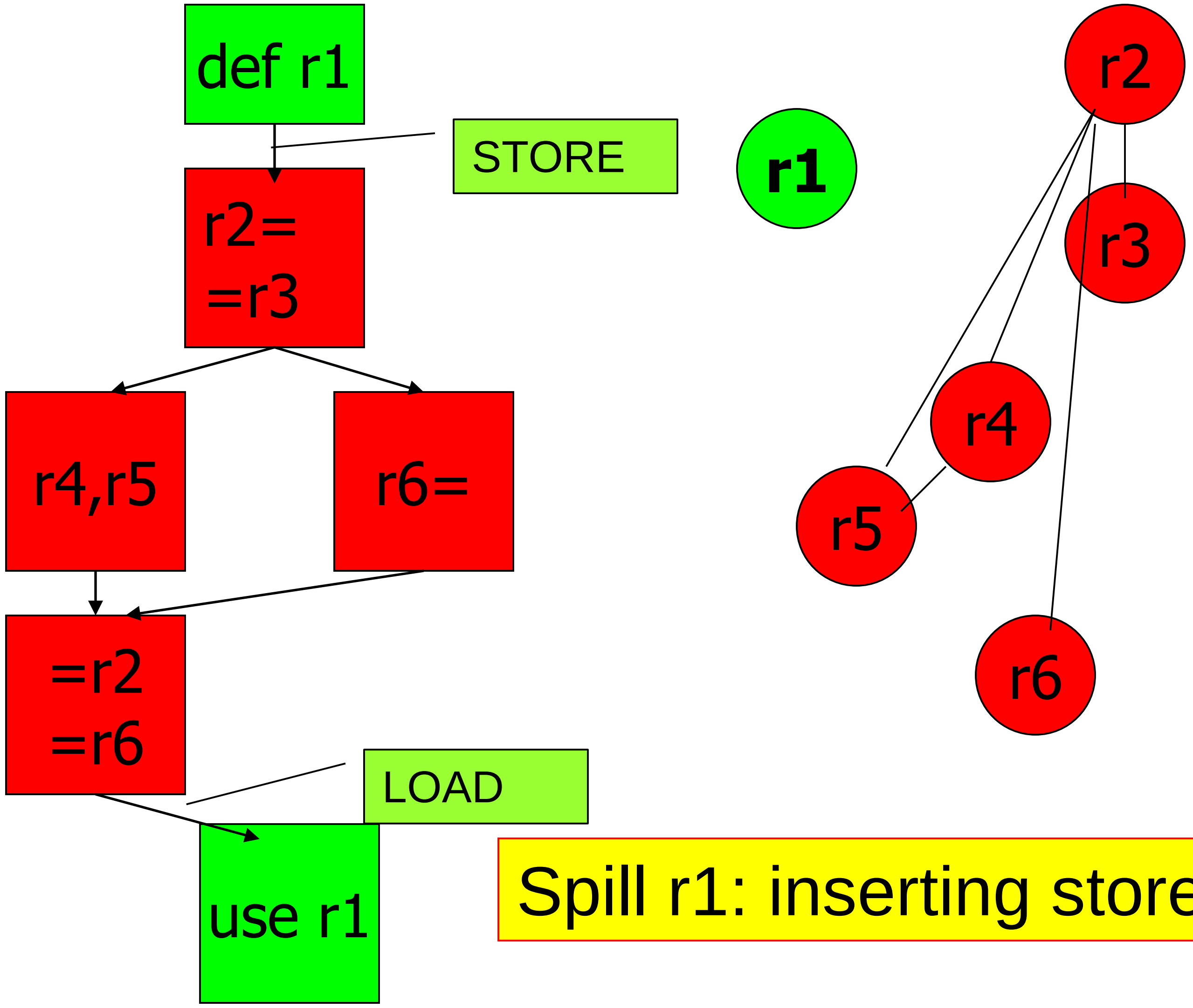
Spilling Registers





Not 3-colorable

Spilling Registers



Spill r1: inserting store and load

Spilling Cost

- How to determine which web (or PR) to spill?
  - ❖ The spill cost of each web is computed based on the number of defs, uses in the program
  - ❖ If a web's value can be more efficiently recomputed than reloaded, the cost of re-computing is used (e.g. the "li" instruction is cheaper than reload).
  - ❖ If a spilled value is used several times (*clustered*), only a single load is needed.
  - ❖ Profiles may be useful in spill cost computation

Speculative Register Promotion

- Example:

x = a->b + …

*p = …

y = a->b + …

Can we allocate a->b to a register?

What if *p modifies a->b? or a ?

- Example:

  x = a->b + …

  *p = …

  y = a->b + …

Accurate alias analysis would enable more effectively register allocation, but it is difficult and very costly.

- inter-procedural analysis
- dynamic allocated objects

- Example:

  x = a->b + …

  *p = …

  y = a->b + …

Compiler can speculatively promote *a*->*b* to a register as long as it can be verified at runtime that *p does not alias to *a* or *a*->*b*.

ALAT

r1	00011

Assume p=xxx..xxx00011

=*p+1;	**ld.a r1=[p]**
*q=…	add r3=r1,1
=*p+9;	st [q] = ….
	ld.c r1=[p]
	add r4=r1, 9

If q =…..00011, the ST inst will invalid the r1 entry and the ld.c will fail.

If q != …..00011, r1 entry will remain in the table, and ld.c will be handled as a NOP

Note: Check (ld.c) is much less expensive than load (ld.a)