CS 335: Register Allocation

Swarnendu Biswas

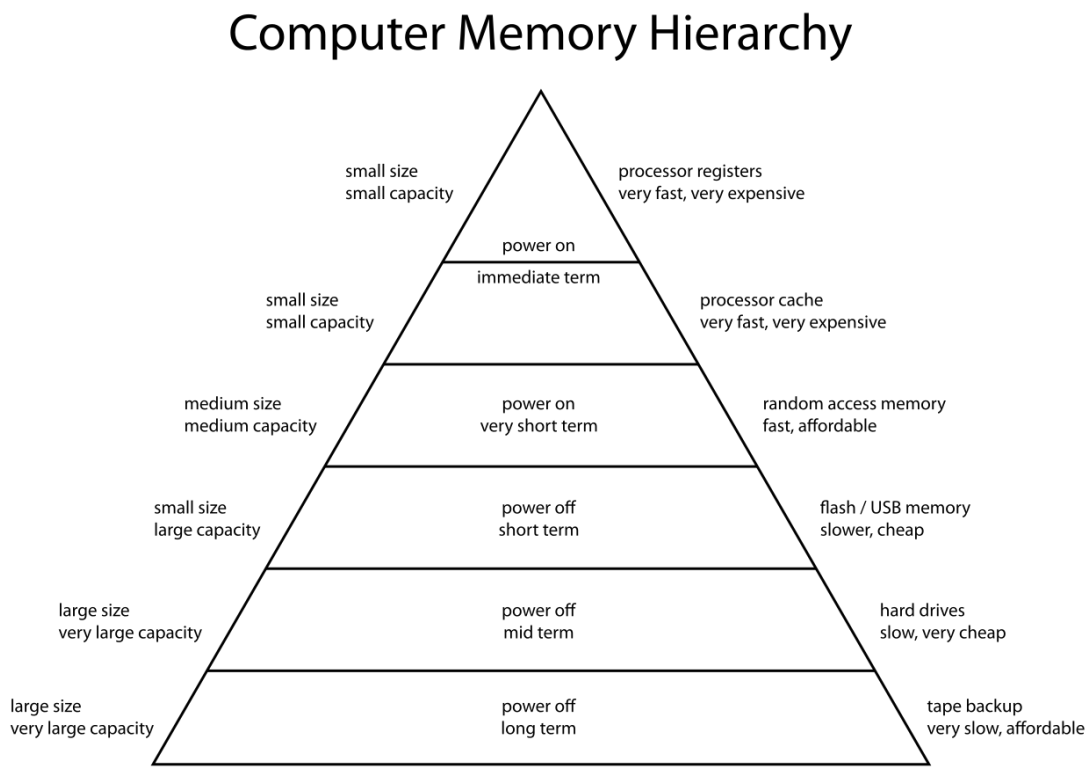
Semester 2019-2020-II

CSE, IIT Kanpur

Content influenced by many excellent references, see References slide for acknowledgements.

Impact of Register Operands

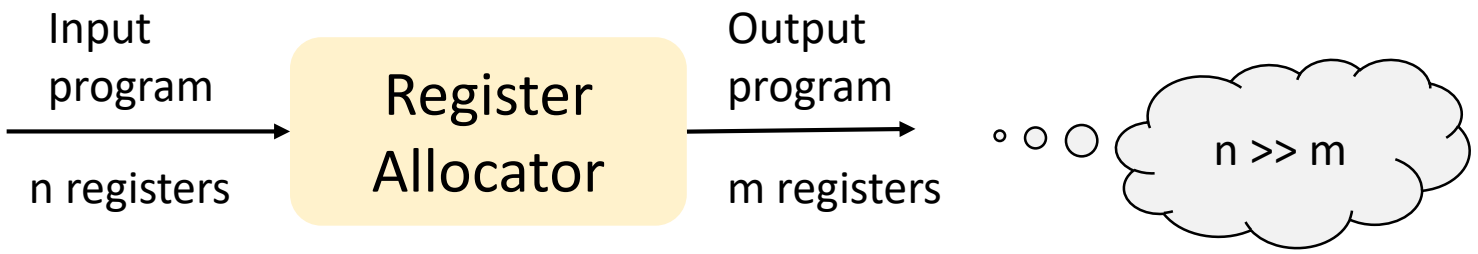
- Instructions involving register operands are faster than those involving memory operands
- Code is often smaller and hence is faster to fetch
- Efficient utilization of registers is important
  - Number of general-purpose registers is limited
    - ~16-32 64-bit general-purpose registers



Goals in Register Allocation

- All variables are not used (or live) at the same time

- Register allocator in a compiler helps with decision making
  - Which values will reside in registers?
  - Which register will hold each of those values?
  - At each program location, values stored in virtual registers in the IR are mapped to physical registers



Goals in Register Allocation

- Programs spend most of their time in loops
  - Natural to store values in innermost loops in registers
- When no registers are available to store a computation, the contents of one of the in-use registers must be stored into memory
  - This is called register spilling
    - Spilling requires generating load and store instructions
  - Concerns associated with spilling
    - Code and data overhead associated in spilling, and the overhead in execution time
- Register pressure measures the availability of free registers
  - High register pressure implies that a large fraction of registers are in use

Goals in Register Allocation

- Goal in register allocation is thus to minimize the impact of spills especially for performance-critical code
- Allocation
  - Maps an unlimited name space onto the register set of the target machine
  - For e.g., map virtual registers to the physical register set and spill values that do not fit in the physical register set
  - Or, map a subset of memory locations to a set of physical registers
- Assignment
  - Map an allocated name set to the physical registers of the target machine
  - Assumes that allocation has already been performed

An Analogy

- Register allocation is a bit like room scheduling
- Room scheduling
  - We have a set of rooms (registers)
  - We have a set of classes (variables) to fit into the rooms
  - Two classes that meet at the same time cannot be allocated to the same room
- The difference is that in room scheduling there can be no spilling; no one gets to have their lecture in the park!

Challenges in Register Allocation

- Architectures provide different register classes
  - General purpose registers, floating-point registers, predicate and branch target registers
  - General-purpose registers may be used for floating-point register spills, which implies an order for allocation
  - Registers may be aliased
    - x86 has 32-bit registers whose lower halves are used as 16-bit or 8-bits registers
    - Similarly, vector registers like zmm, ymm, and xmm
  - If different register classes overlap, the compiler must allocate them together
- PowerPC calling conventions requires parameters to be passed in R3-R10 and the return is in R3

Register Allocation Problem

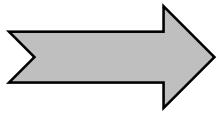
- General formulation of the problem is NP-Complete
  - For e.g., register allocation for a set of BBs, multiple control flow paths, multiple data types, non-uniform cost of memory access complicate the analysis
  - Optimal allocation can be done in polynomial time for very restricted versions with a single BB, with one data type,

Local Register Allocation

Local Register Allocation

- Assumptions
  - Considers only a single basic block
  - Loads values from memory and stores to memory
  - Single class of k general-purpose registers on the target machine

$$\begin{aligned} &\text{op}_1\ vr_{1_1}, vr_{1_2} \Rightarrow vr_{1_3} \\ &\text{op}_2\ vr_{2_1}, vr_{2_2} \Rightarrow vr_{2_3} \\ &\qquad\qquad \ldots \\ &\text{op}_n\ vr_{n_1}, vr_{n_2} \Rightarrow vr_{n_3} \end{aligned}$$



$$\begin{aligned} &\text{op}_1\ ?, ? \Rightarrow ? \\ &\text{op}_2\ ?, ? \Rightarrow ? \\ &\qquad \ldots \\ &\text{op}_n\ ?, ? \Rightarrow ? \end{aligned}$$

Top-Down Allocation with Frequency Counts

- Idea
  - Count the frequency of occurrence of virtual registers
  - Map virtual registers to physical registers in descending order of frequency
- If the BB uses fewer than k virtual registers, then mapping is trivial
- A few registers ($F \cong 2{-}4$) are required to execute spill code
- Assign the top $(k - F)$ virtual registers to physical registers
- Rewrite the code and replace virtual registers with physical registers
- For unassigned virtual registers, generate code sequence to spill code using the F reserved registers

Drawback of Top-Down Allocation

- Top-down local allocation keeps heavily used virtual registers in physical registers
  - Allocates a physical register to one virtual register for the entire BB
- Allocation can be suboptimal if values show phased behavior
  - A value heavily-used in the first half of the BB and no use in the second half of the BB still stays in the physical register

Bottom-Up Allocation

- Iterates over the operations in the BB and makes decisions

- Assume that registered are grouped in classes
  - `Size`: # physical registers
  - `Name`: virtual register name
  - `Next`: distance to next reuse
  - `Free`: flag to indicate whether currently in use

```
struct Class {
  int Size;
  int Name[Size];
  int Next[Size];
  int Free[Size];
  int Stack[Size];
  int StackTop;
}
```

Bottom-Up Algorithm

```
for each operation i = {1 … n}
  r_x = Ensure(vr_i1, class(vr_i1))
  r_y = Ensure(vr_i2, class(vr_i2))
  if vr_i1 is not needed after i
    Free(r_x, class(r_x))
  if vr_i2 is not needed after i
    Free(r_y, class(r_y))
  r_z = Allocate(vr_i3, class(vr_i3))
  rewrite i as op_i r_x,r_y ⇒ r_z
  if vr_i1 is needed after i
    class.next[r_x]= Dist(vr_i1)
  if vr_i2 is needed after i
    class.next[r_y]= Dist(vr_i2)
  class.next[r_z]= Dist(vr_i3)
```

Bottom-Up Algorithm

```
Ensure(vr, class)
  if vr is already in class
    result = vr's physical register
  else
    result = Allocate(vr, class)
    emit code to move vr into result
  return result
```

```
Allocate(vr, class)
  if class.StackTop >= 0
    i = pop(class)
  else
    i = j that maximizes class.Next[j]
    store contents of j
  class.Name[i] = vr
  class.Next[i] = -1
  class.Free[i] = false
  return i
```

Challenges in Bottom-Up Allocation

- A store on a spill is unnecessary if the data is clean
  - Register contains a constant value
  - Register contains a return from a load
- A spill should be stored only if the data is dirty

- Nice!

But is it so straightforward?

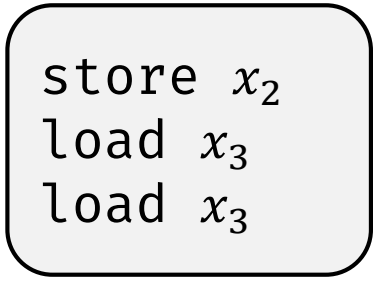
- Assume a two-register machine
- x_1 is clean and x_2 is dirty
- Assume reference stream for the rest of the BB is $x_3 x_1 x_2$

```
store x2
load x3
load x3
```

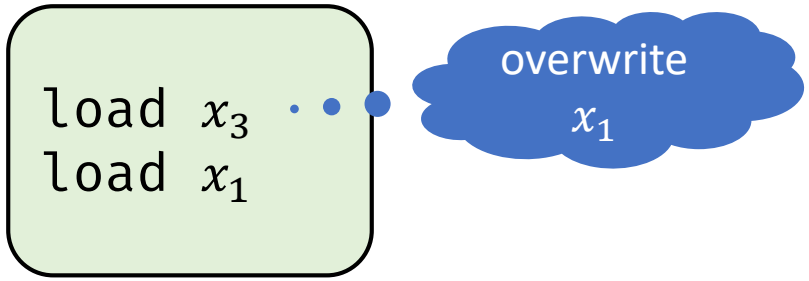
spill dirty value

But is it so straightforward?

- Assume a two-register machine
- x_1 is clean and x_2 is dirty
- Assume reference stream for the rest of the BB is $x_3 x_1 x_2$



spill dirty value



spill clean values

But is it so straightforward?

- Assume a two-register machine
- x_1 is clean and x_2 is dirty
- Assume reference stream for the rest of the BB is $x_3 x_1 x_3 x_1 x_2$

```
load x3
load x1
load x3
load x1
```

spill clean values

```
store x2
load x3
load x2
```

spill dirty values

Global Register Allocation

Global Register Allocation

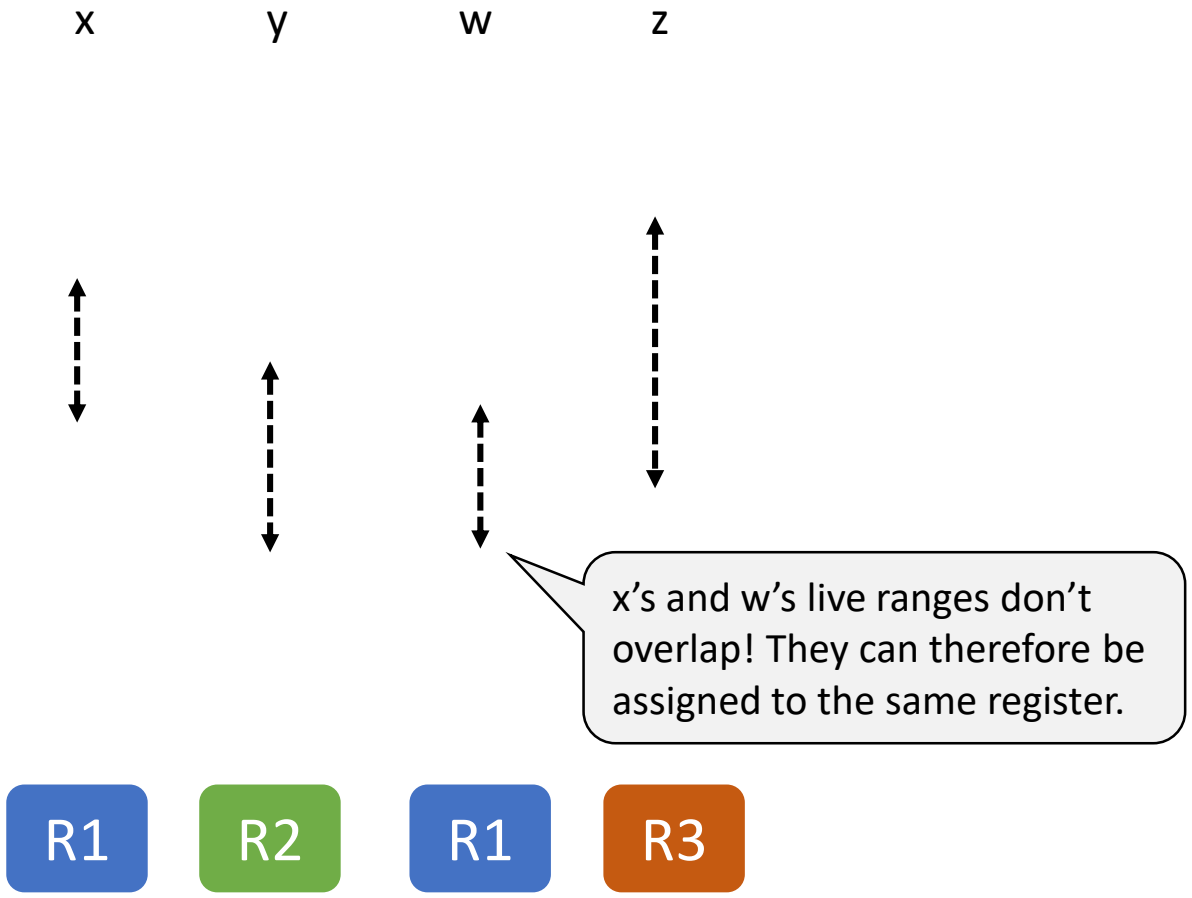
- Scope is either multiple BBs or a whole procedure
- Fundamentally a more complex problem than local register allocation
  - Need to consider def-use across multiple blocks
  - Cost of spilling may not be uniform since it depends on the execution frequency of the block where a spill happens

Live Ranges

- A live range for a variable is a region of code
- A global live range is defined as
  - For a use u in live range LR_i, LR_i must include every definition d that reaches u
  - For each definition d in LR_i, LR_i must include every use u that d reaches
- A variable's live range
  - Starts at the point in the code where the variable receives a value
  - Ends where that value is used for the last time

Example

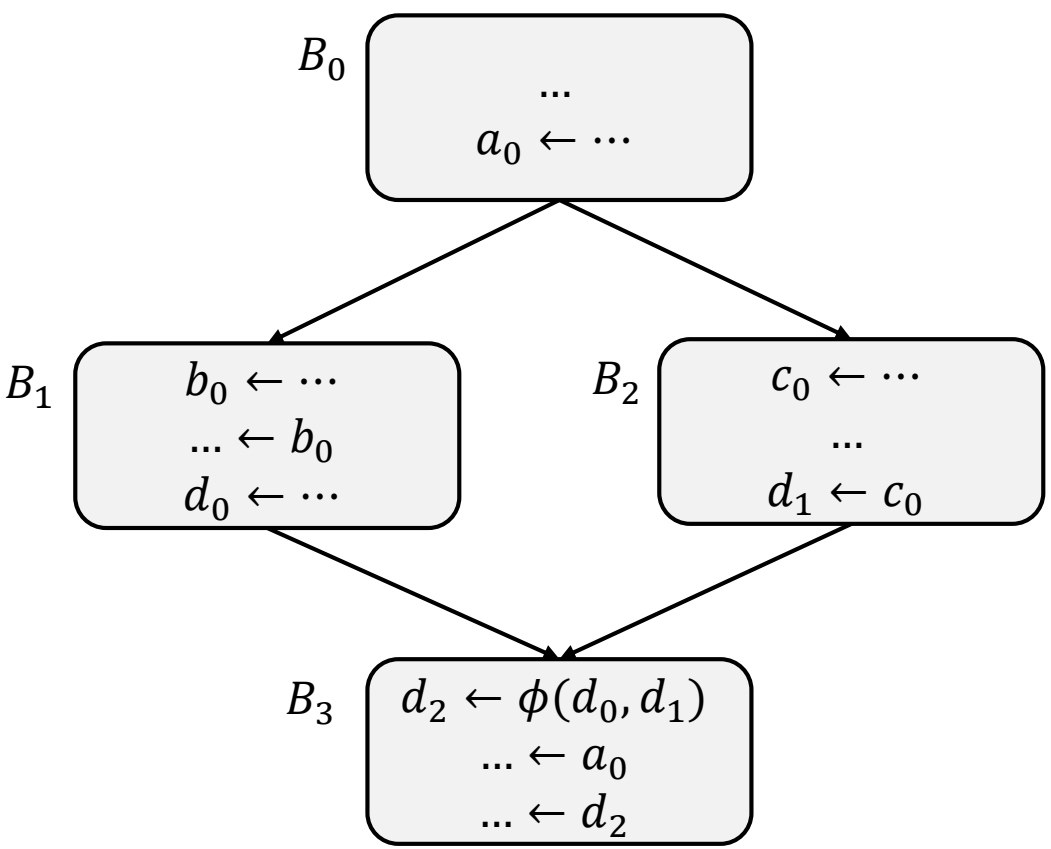
```
z = 1
x = 2 * x
y = 3 * z
w = x + y
print y + z
x = y * w
```

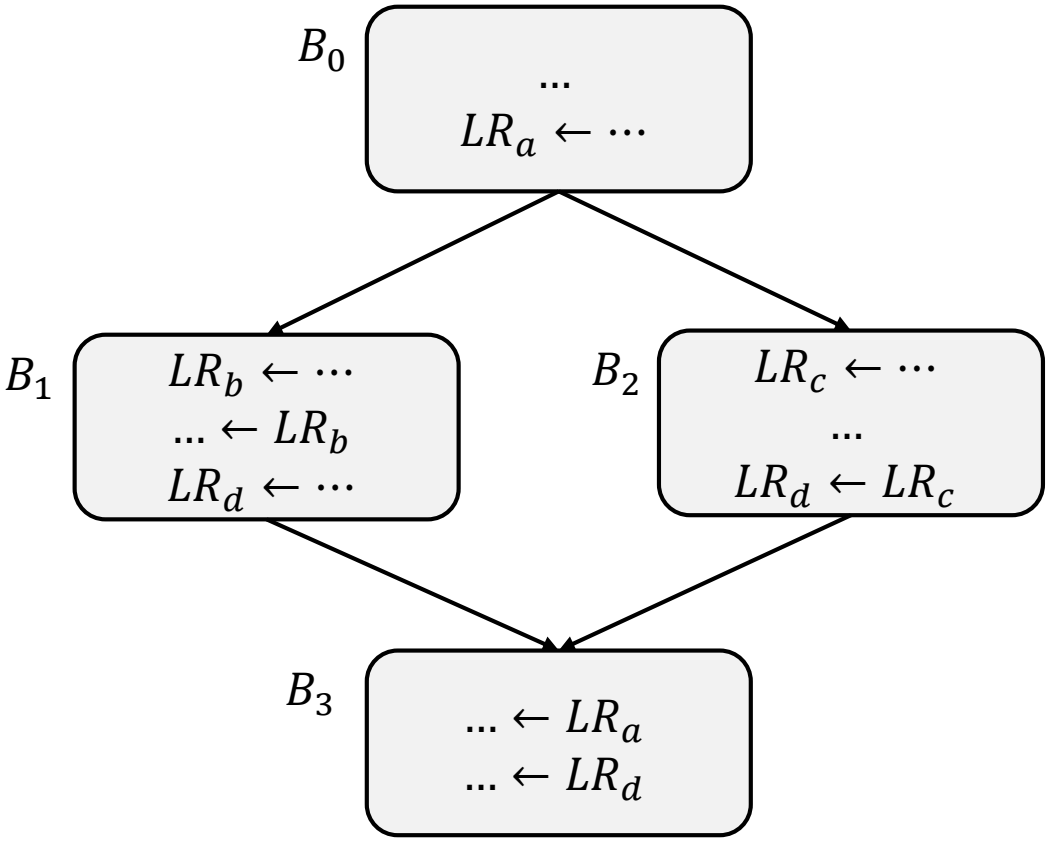


Identifying Global Live Ranges

- Requirement:
  - Group all definitions that reach a single use
  - Group all uses that a single definition can reach
- Assumption: Register allocation operates on the SSA form
  - In SSA, each name is defined once, and each use refers to one definition
  - ϕ functions are used at control flow merge points

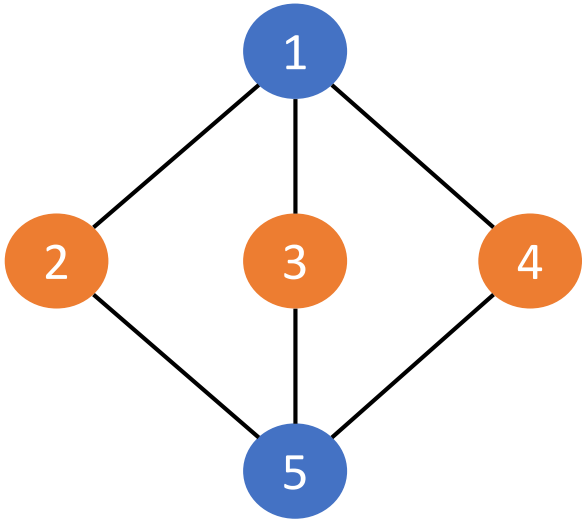
Example: Discovering Live Ranges

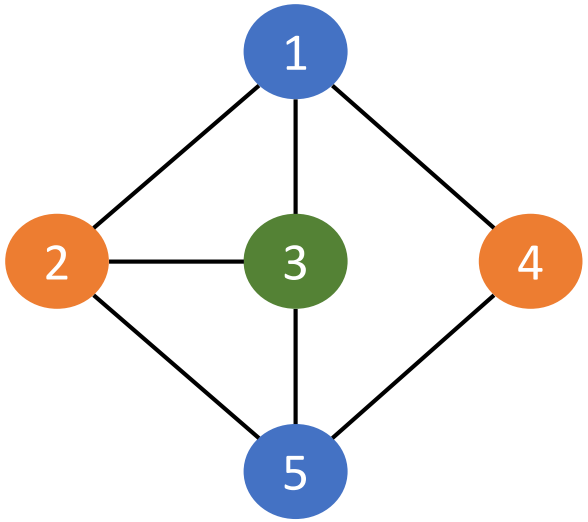




The Graph Coloring Problem

- For an arbitrary graph G, a coloring of G assigns a color to each node in G so that no pair of adjacent nodes have the same color
- A coloring that uses k colors is termed a k-coloring
- The smallest possible k for a given graph is called the graph's chromatic number





Complexity of Graph Coloring

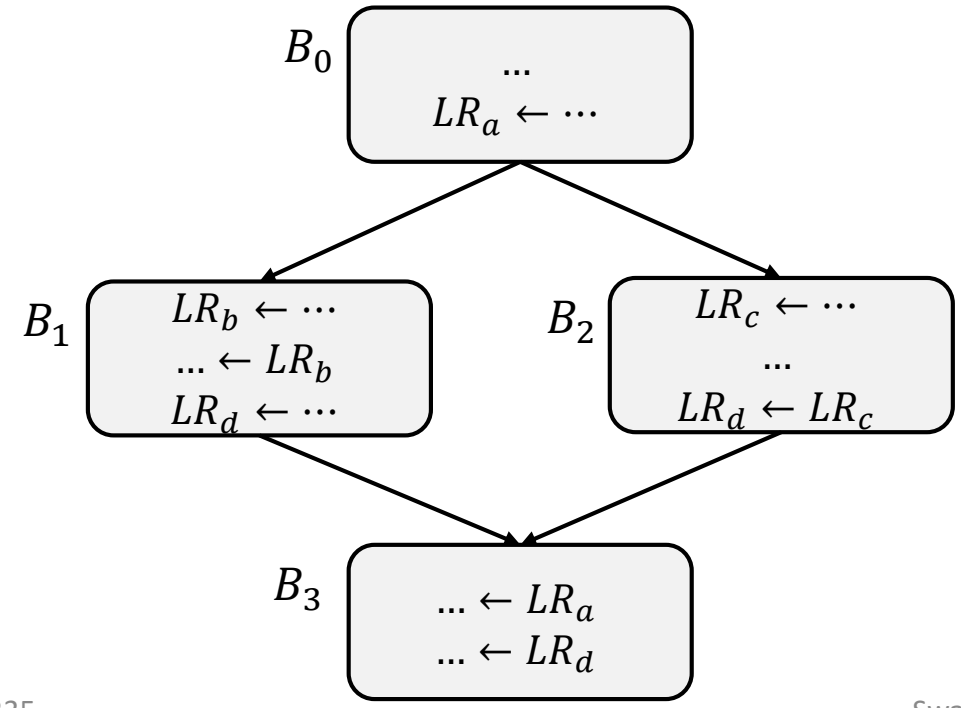
- For a given graph G, the problem of finding its chromatic number is NP-complete
- Determining if a graph G is k-colorable, for some fixed k, is NP-complete

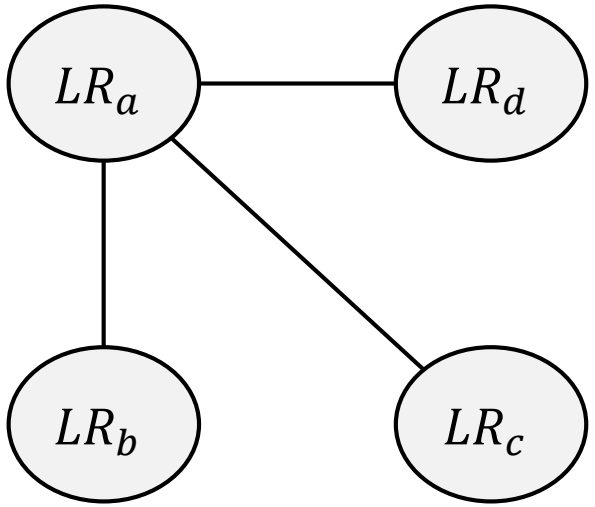
Register Assignment with Graph Coloring

- Compilers model register allocation through coloring on an interference graph
  - Each color represents an available register
- An interference graph models conflicts in live regions
  - Nodes in an interference graph represent live ranges (LR) for a variable
  - If variables a and b are active (live) at the same point, they cannot be assigned to the same register
  - An edge (i, j) indicates LR_i and LR_j cannot share a register
  - Interference graph is undirected

Interferences and the Interference Graph

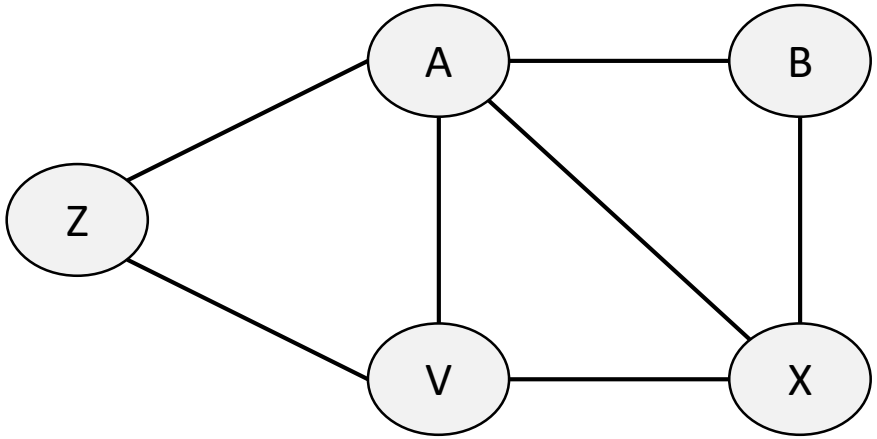
- If there is an operation during which both LR_i and LR_j are live, they cannot reside in the same register
  - Two live ranges LR_i and LR_j interfere is one is live at the definition of the other and they have different values





Another Example

```
a = 5
d = 9 + a
e = a + d
b = d + a
f = e + 6
c = b + f
```



Connect a and b if a is live at a point where b is defined

Register Assignment with Graph Coloring

- An k-coloring of the interference graph indicates possible register assignment to k physical registers
- A failed attempt implies compiler needs to generate spill code
- This iterative process will terminate since spilling minimizes the number of values to be kept in registers

Estimating Cost of Global Spills

- Cost of spilling in global allocation depends on the location
  - The cost is uniform for local spills
- Global allocators annotate each reference with an estimated execution frequency
  - Information is derived through static analysis, heuristics, or from profile information
- Annotations are used to guide decisions about both allocation and spilling

Top-Down Coloring

- Top-down allocators color live ranges in an order determined by some ranking function
- Priority-based allocators assign each node a rank that is the estimated runtime savings that accrue from keeping that live range in a register
- Uses registers for important variables as defined by the ranking function

Bottom-Up Coloring

- Top-down coloring involves high-level information for ranking
- Bottom-up coloring uses low-level structural knowledge from the inference graph

References

- K. Cooper and L. Torczon. Engineering a Compiler, 2nd edition, Chapter 13.
- A. Aho et al. Compilers: Principles, Techniques, and Tools, 2nd edition, Chapter 8.8.
- Register Allocation, Wikipedia. https://en.wikipedia.org/wiki/Register_allocation
- Christian Collberg. CSc 553: Register Allocation, Department of Computer Science, University of Arizona.