COS320: Compiling Techniques

Zak Kincaid

April 22, 2019

Register allocation

Motivation

- Your LLVMlite compiler places each uid in its own stack slot
- Every binary operation is compiled to 2 loads, the operation, and a store
- Loads and stores are *expensive*
- *Register allocation* is the problem of determining a mapping from IR-level "virtual registers" to machine registers

Live variables

- A variable x is live at a point n if there is some path starting from n that *reads* the value of x before *writing* it.
  - Intuition: a variable is live if its value might be needed later in some computation.
- If a variable x is *not* live, we can free/re-use the memory associated with x
- If two variables are not live at the same time, we can store them in the same memory (ideally, a register)

Live variables

- Live variables is a *backwards* dataflow analysis problem
  - Information flows from control flow *successors* to their *predecessors*

Forwards: compute *least* function f s.t.

1. $f(s) = \top$
2. For all $p \to n \in E$, $\textit{post}(p, f(p)) \sqsubseteq f(n)$

Backwards: compute *least* function g s.t.

1. $g(n) = \top$ for all n that terminate with a `return`
2. For all $n \to s \in E$, $\textit{pre}(s, g(p)) \sqsubseteq g(n)$

  - Backwards analyses work in essentially the same was as forwards analyses
- Live variables as a data flow analysis:

Live variables

- Live variables is a *backwards* dataflow analysis problem
  - Information flows from control flow *successors* to their *predecessors*

Forwards: compute *least* function f s.t.

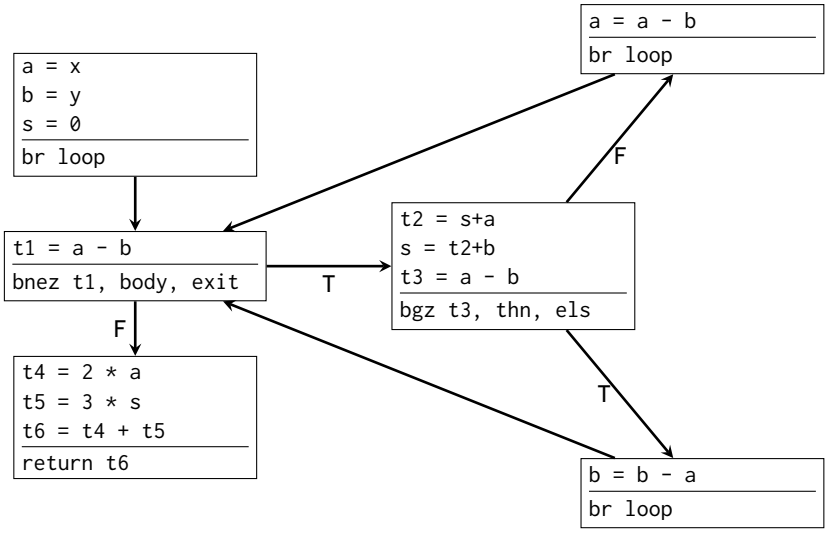
1. $f(s) = \top$
2. For all $p \to n \in E$, $\textit{post}(p, f(p)) \sqsubseteq f(n)$

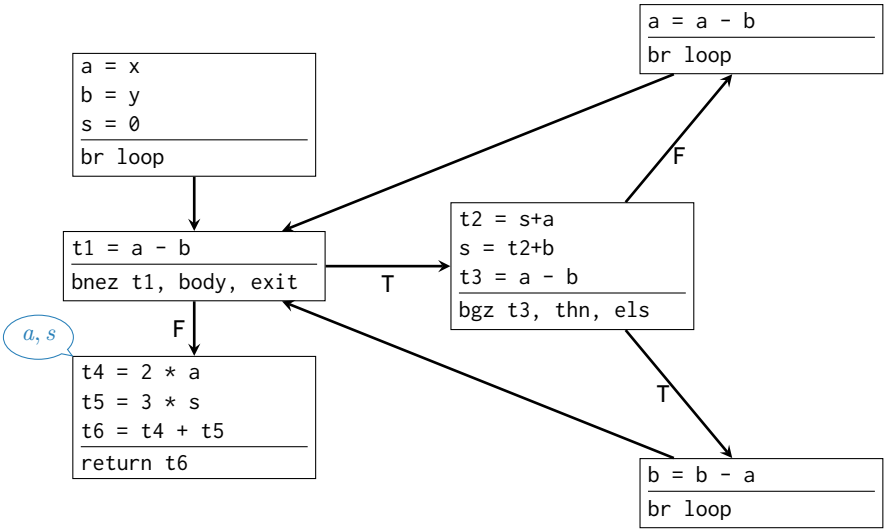
Backwards: compute *least* function g s.t.

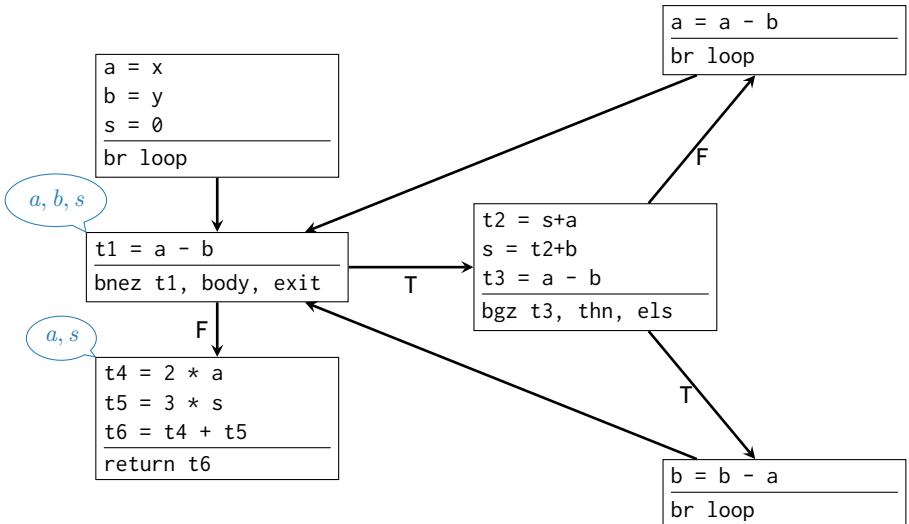
1. $g(n) = \top$ for all n that terminate with a `return`
2. For all $n \to s \in E$, $\textit{pre}(s, g(p)) \sqsubseteq g(n)$

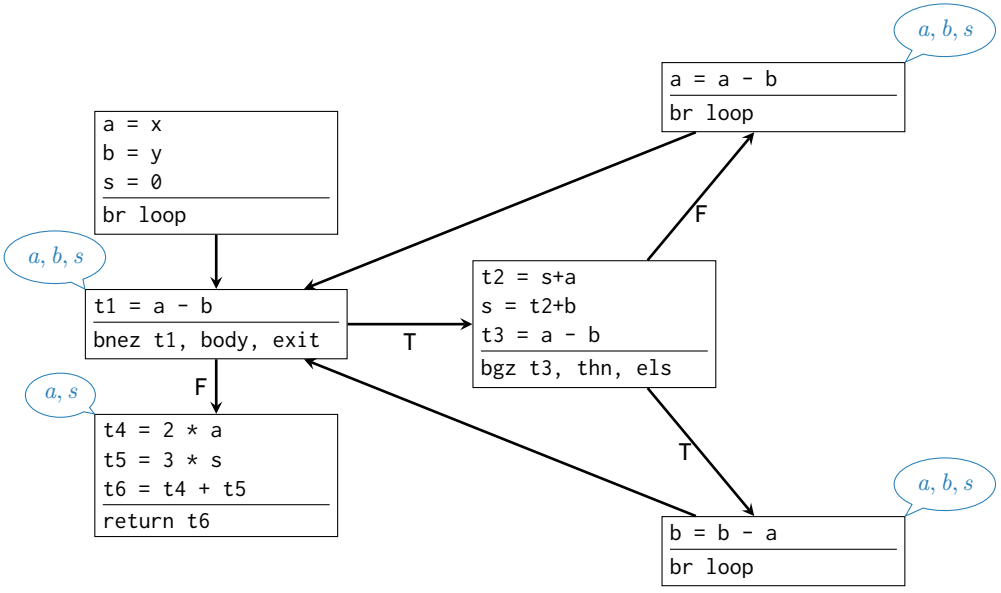
  - Backwards analyses work in essentially the same was as forwards analyses
- Live variables as a data flow analysis:
  - Abstract domain: 2^{Var}
    - *Existential* $\Rightarrow$ order is $\subseteq$, join is $\cup$, $\top$ is *Var*, $\bot$ is $\emptyset$
  - $\textit{kill}(x := e) = \{x\}$, $\textit{kill}(\texttt{cbr x, l1, l2}) = \emptyset$
  - $\textit{gen}(x := e) = \{y : y \textit{ in } e\}$, $\textit{gen}(\texttt{cbr x, l1, l2}) = \{x\}$
  - $\textit{pre}(elt, L) = (L \setminus \textit{kill}(elt)) \cup \textit{gen}(elt)$

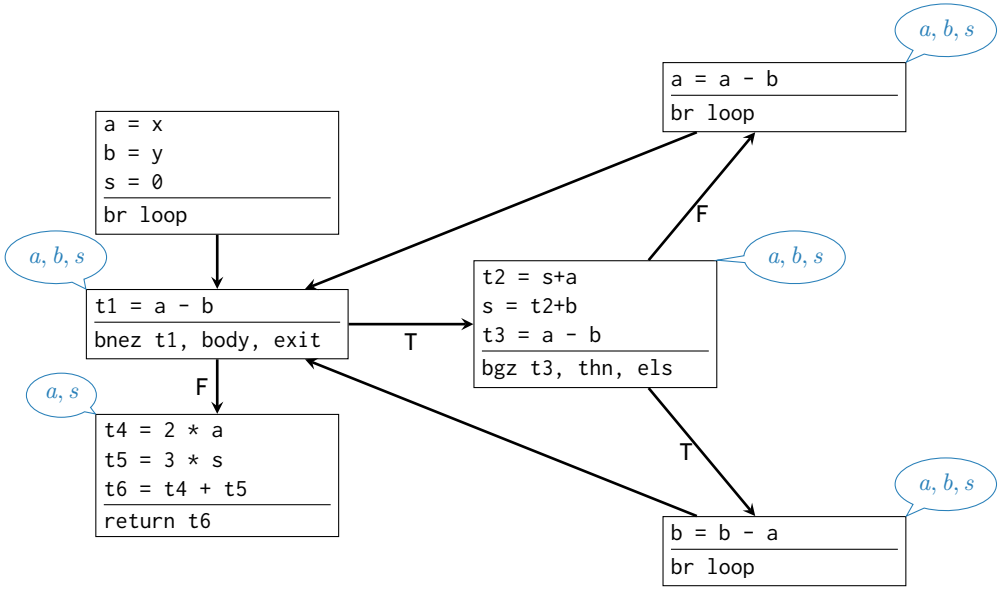
```
foo(int x, int y) {
  a := x;
  b := y;
  s := 0;
  while (a != b) {
    s := s + a + b;
    if (a > b) {
      a := a - b;
    } else {
      b := b - a;
    }
  }
  return 2 * a + 3 * s;
}
```

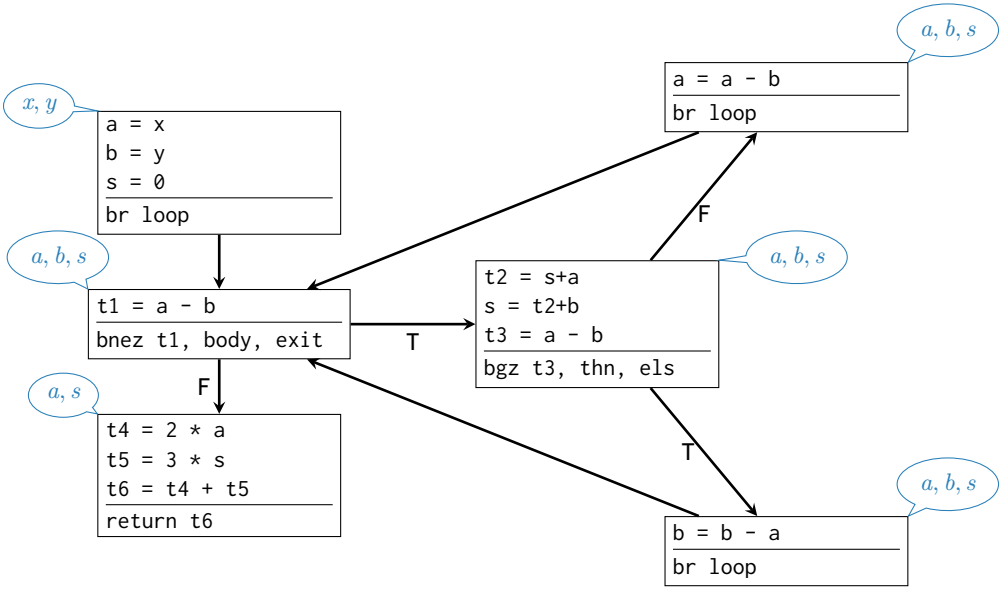

a = x
b = y
s = 0
br loop
t1 = a - b
bnez t1, body, exit
T
F
t4 = 2 * a
t5 = 3 * s
t6 = t4 + t5
return t6
t2 = s+a
s = t2+b
t3 = a - b
bgz t3, thn, els
F
T
a = a - b
br loop
b = b - a
br loop


a = x
b = y
s = 0
br loop
t1 = a - b
bnez t1, body, exit
T
F
a = a - b
br loop
F
t2 = s+a
s = t2+b
t3 = a - b
bgz t3, thn, els
T
a, s
t4 = 2 * a
t5 = 3 * s
t6 = t4 + t5
return t6
b = b - a
br loop


a = x
b = y
s = 0
br loop
a, b, s
t1 = a - b
bnez t1, body, exit
T
F
a, s
t4 = 2 * a
t5 = 3 * s
t6 = t4 + t5
return t6
t2 = s+a
s = t2+b
t3 = a - b
bgz t3, thn, els
F
T
a = a - b
br loop
b = b - a
br loop


a = x
b = y
s = 0
br loop
a, b, s
t1 = a - b
bnez t1, body, exit
T
F
a, s
t4 = 2 * a
t5 = 3 * s
t6 = t4 + t5
return t6
t2 = s+a
s = t2+b
t3 = a - b
bgz t3, thn, els
F
T
a, b, s
a = a - b
br loop
a, b, s
b = b - a
br loop


a = x
b = y
s = 0
br loop
a, b, s
t1 = a - b
bnez t1, body, exit
T
F
a, s
t4 = 2 * a
t5 = 3 * s
t6 = t4 + t5
return t6
t2 = s+a
s = t2+b
t3 = a - b
bgz t3, thn, els
a, b, s
F
T
a = a - b
br loop
a, b, s
b = b - a
br loop
a, b, s

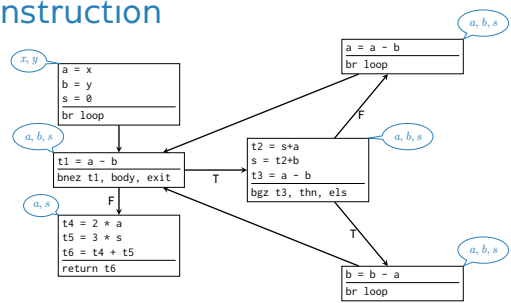

x, y
a = x
b = y
s = 0
br loop
a, b, s
t1 = a - b
bnez t1, body, exit
T
F
a, s
t4 = 2 * a
t5 = 3 * s
t6 = t4 + t5
return t6
a, b, s
t2 = s+a
s = t2+b
t3 = a - b
bgz t3, thn, els
F
T
a, b, s
a = a - b
br loop
a, b, s
b = b - a
br loop

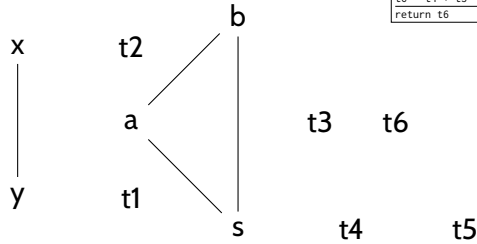
Interference graph

- An interference graph for a CFG is an undirected graph (V, I) where
  - Vertices V = program variables
  - Edges I connect variables x and y iff there is some program point[1] where x and y are simultaneously live
- Vertices that are adjacent in the interference graph cannot be stored in the same memory location

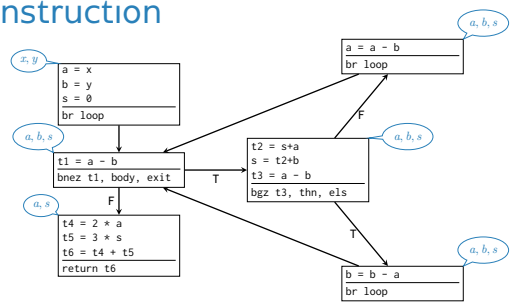
[1] "Program point" includes intermediate points within basic blocks

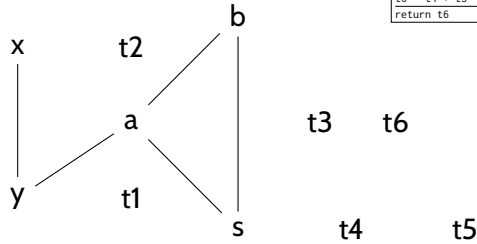
Interference graph construction



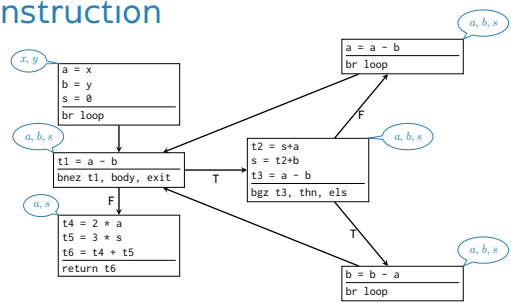


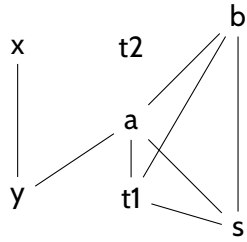
Interference graph construction





Interference graph construction

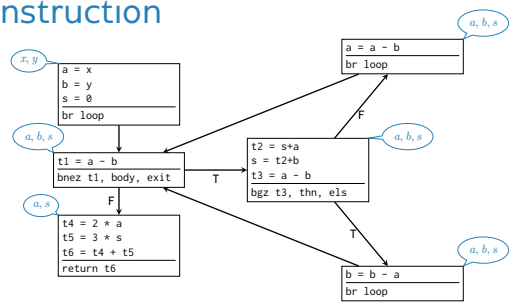


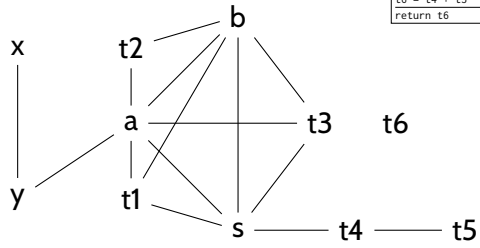


t3 t6

t4 t5

Interference graph construction





Interference graph coloring

- A K-coloring of the interference graph is a function $c : V \to \{1, ..., K\}$ such that if x and y are adjacent in I, then $c(x) \neq c(y)$.
- Basic idea (due to Chaitin): if a processor has K registers, then a K-coloring of its interference graph corresponds to a valid memory layout.

Interference graph coloring

- A K-coloring of the interference graph is a function $c : V \to \{1, ..., K\}$ such that if x and y are adjacent in I, then $c(x) \neq c(y)$.
- Basic idea (due to Chaitin): if a processor has K registers, then a K-coloring of its interference graph corresponds to a valid memory layout.
- Problem: Determining whether a graph is K-colorable is NP-complete
  - *But*: we don't need an optimal coloring – any coloring will do
  - If we use more colors than we have registers, can *spill*: place the variable in memory rather than a register
  - May need to reserve some registers for intermediate computations (e.g., accessing memory)

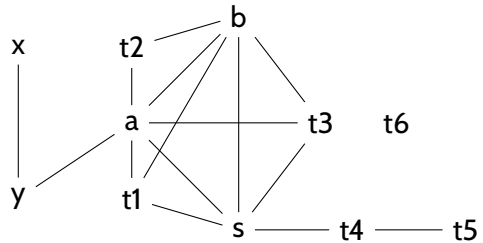
Greedy coloring

- Process:
  - Simplify: choose a node with $< K$ neighbors. Add it to a stack & remove it from the graph
  - Spill: if all nodes have $\geq K$ neighbors, choose one to *potentially* spill. Add it to a stack & remove it from the graph.
  - Color: traverse the stack, assigining colors to the *Simplified* vertices, and either color or spill *Spilled* vertices

Greedy coloring

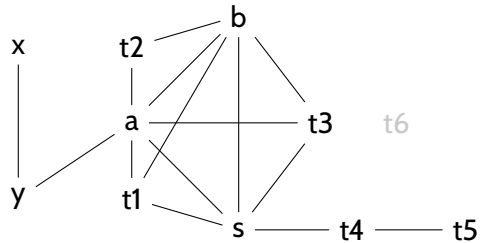
- Process:
  - Simplify: choose a node with $< K$ neighbors. Add it to a stack & remove it from the graph
  - Spill: if all nodes have $\geq K$ neighbors, choose one to *potentially* spill. Add it to a stack & remove it from the graph.
  - Color: traverse the stack, assigining colors to the *Simplified* vertices, and either color or spill *Spilled* vertices
- Not optimal: may use more colors than needed
  - but works well in practice.

3-coloring the interference graph



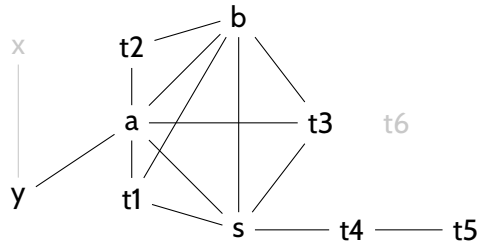
Stack:

3-coloring the interference graph



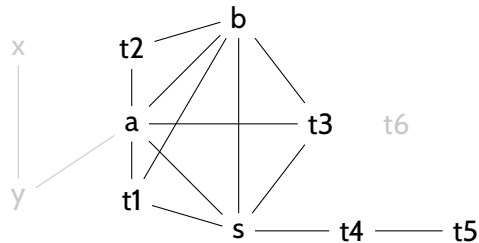
Stack: t6

3-coloring the interference graph



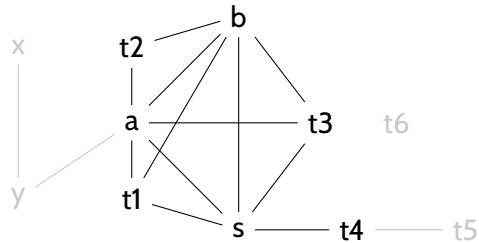
Stack: t6,x

3-coloring the interference graph



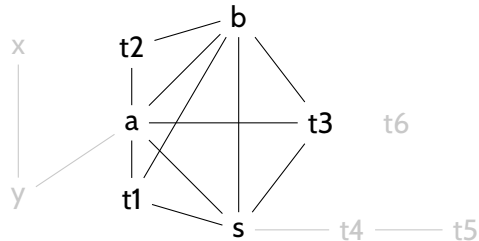
Stack: t6,x,y

3-coloring the interference graph



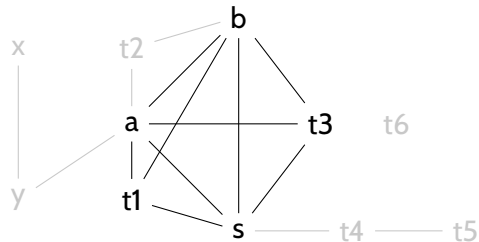
Stack: t6,x,y,t5

3-coloring the interference graph



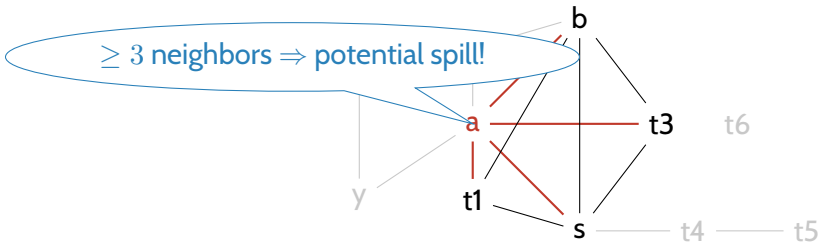
Stack: t6,x,y,t5,t4

3-coloring the interference graph



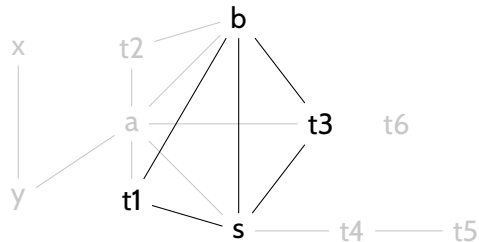
Stack: t6,x,y,t5,t4,t2

3-coloring the interference graph



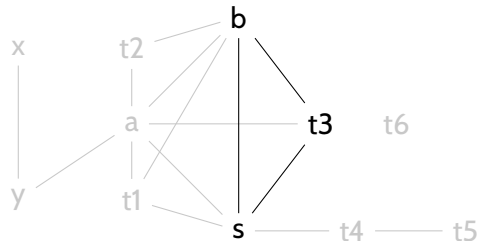
Stack: t6,x,y,t5,t4,t2

3-coloring the interference graph



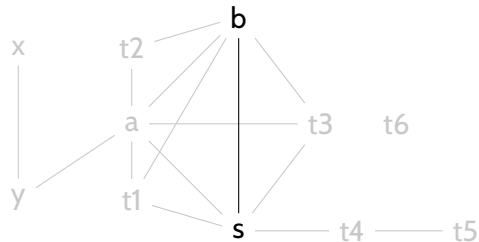
Stack: t6,x,y,t5,t4,t2,a

3-coloring the interference graph



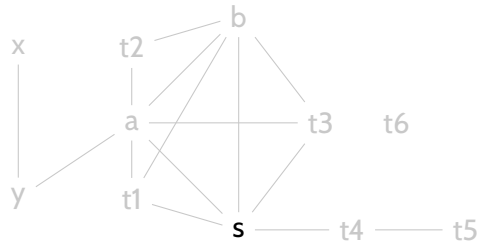
Stack: t6,x,y,t5,t4,t2,a,t1

3-coloring the interference graph



Stack: t6,x,y,t5,t4,t2,a,t1,t3

3-coloring the interference graph

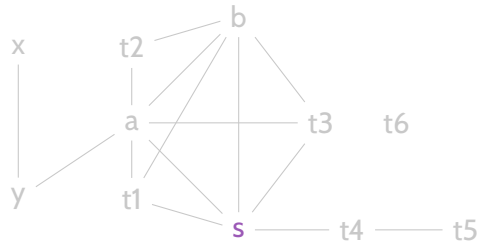


Stack: t6,x,y,t5,t4,t2,a,t1,t3,b

3-coloring the interference graph

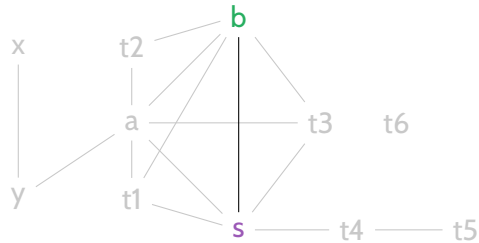
Stack: t6,x,y,t5,t4,t2,a,t1,t3,b,s

3-coloring the interference graph



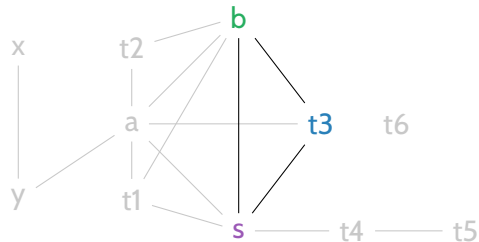
Stack: t6,x,y,t5,t4,t2,a,t1,t3,b,s

3-coloring the interference graph



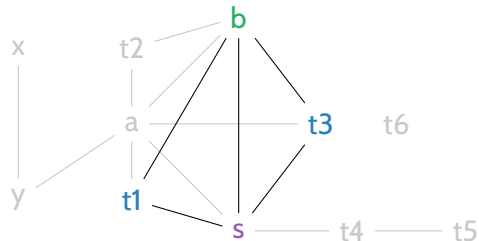
Stack: t6,x,y,t5,t4,t2,a,t1,t3,b,s

3-coloring the interference graph



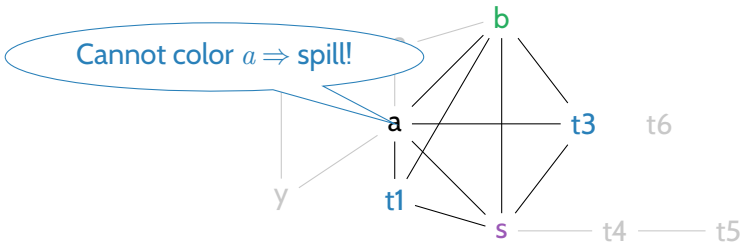
Stack: t6,x,y,t5,t4,t2,a,t1,t3,b,s

3-coloring the interference graph



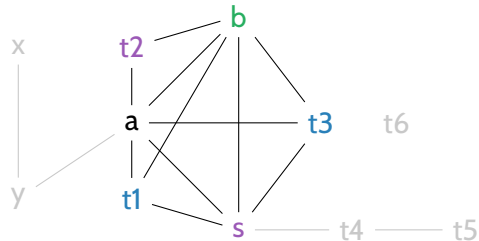
Stack: t6,x,y,t5,t4,t2,a,t1,t3,b,s

3-coloring the interference graph



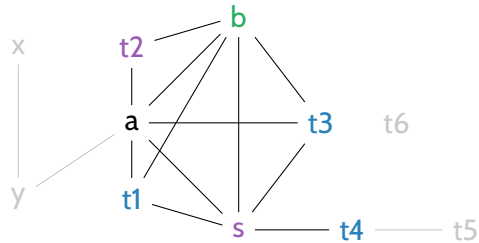
Stack: t6,x,y,t5,t4,t2,a,t1,t3,b,s

3-coloring the interference graph



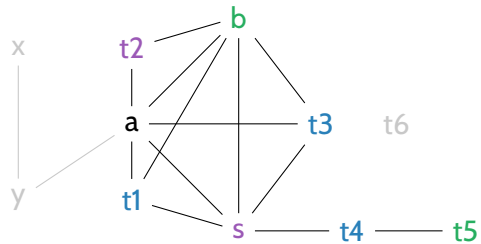
Stack: t6,x,y,t5,t4,t2,a,t1,t3,b,s

3-coloring the interference graph



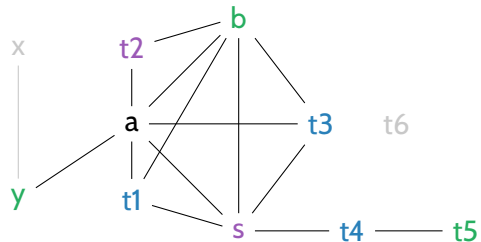
Stack: t6,x,y,t5,t4,t2,a,t1,t3,b,s

3-coloring the interference graph



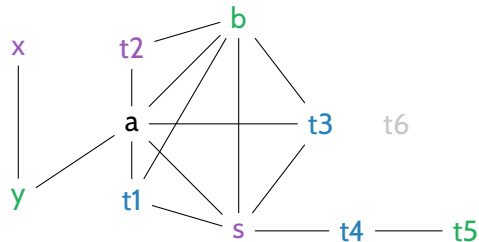
Stack: t6,x,y,t5,t4,t2,a,t1,t3,b,s

3-coloring the interference graph



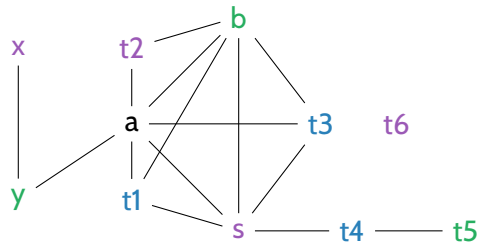
Stack: t6,x,y,t5,t4,t2,a,t1,t3,b,s

3-coloring the interference graph



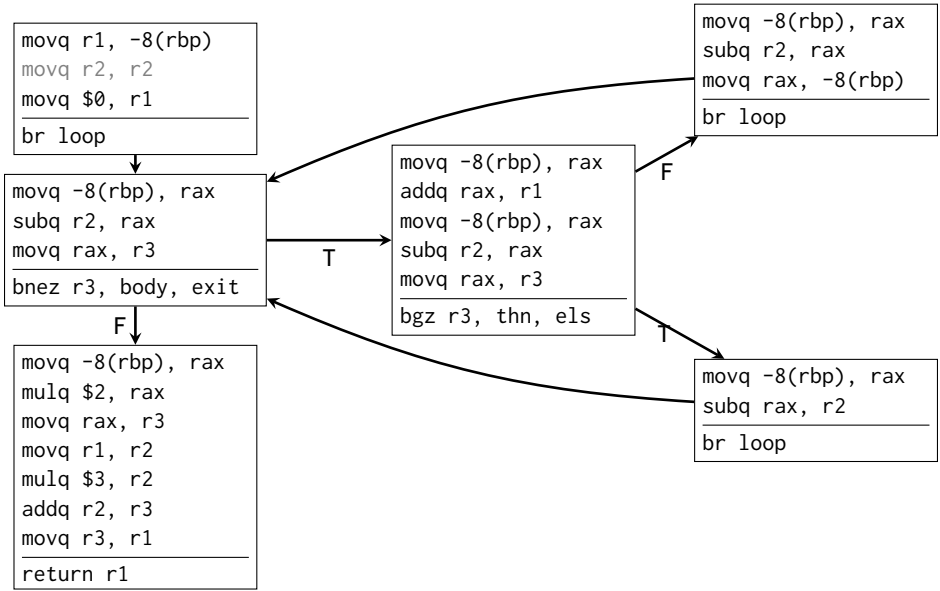
Stack: t6,x,y,t5,t4,t2,a,t1,t3,b,s

3-coloring the interference graph



Stack: t6,x,y,t5,t4,t2,a,t1,t3,b,s

Suppose we have two reserved registers `rax`,`rcx` and three available registers `r1`,`r2`,`r3`



Accessing spilled registers

Problem: we need to use registers to be able to access the stack slots that we use to store spilled virtual registers

- Easy option: reserve some registers for memory operation (`rax` and `rcx` in last slide)

Accessing spilled registers

Problem: we need to use registers to be able to access the stack slots that we use to store spilled virtual registers

- Easy option: reserve some registers for memory operation (`rax` and `rcx` in last slide)
- Better option: genererate spill code, then re-run register allocator
  - Spill code accesses memory using virtual registers
  - When we re-run the register allocator, we must allocate registers to these virtual registers, but
  - live range for new virtual register is very short
  - Eventually the process will converge with a K-colored graph an no spills.

Pre-colored nodes

- Some instructions require the use of certain registers
  - E.g., the `call` must pass parameters in `rdi`, `rsi`, `rdx`, `rcx`, `r08`, `r09`
- Virtual registers that must be assigned a particular register should be considered "pre-colored"
  - Not a target for *Simplify* or *Spill*
  - Terminate register allocator when no *uncolored* nodes remain

Graph coalescing

- May be desirable to place two variables in the same register
  - E.g., if we have an assignment $x := y$ and x and y are in the same register, we can elide the `mov` instruction
- Graph coalescing collapses two (non-adjacent) vertices into one vertex with the neighborhood of both
- Coalescing creates more register pressure
- Strategies to preserve K-colorability
  - Brigg's: coalesce only when the resulting node has $< K$ neighbors
  - George's: coalesce x and y only when each neighbor of x is either a neighbor of y or has degree $< k$.

More register allocation

Graph coloring is not the end of the story...

- Spill selection: if an interference graph cannot be simplified, which register should be spilled?
  - Priority based on # of edges, # of uses of the variable, ...
- Live range splitting
  - Might be desirable to allocate a single variable in different registers in different code sections
  - SSA already does some of this implicitly!
- See *Modern Compiler Implementation in ML* Ch 11 for (some) more details