CS322 Languages and Compiler Design II

Spring 2012

Lecture 10

MACHINE CODE GENERATION

- Instruction Selection
- Register Allocation and Assignment
- Optimization

Issues:

- Complexity of Target Machine
- Level of Translation: expression, statement, basic block, routine, program?
- Management of Scarce Resources

Approaches to Instruction Selection

For **RISC** targets, translate one IR instruction to one or more target instructions.

For **CISC** targets, translate several IR instructions to one target instruction.

```
Example Source: a := b  (assuming a,b in frame)

3-addr IR: t1 = fp-12
           t2 = *t1
           t3 = fp+8
           *t3 = t2

Typical Tree IR:          mov
                          / \
                         /   \
                       mem   mem
                        |     |
                        +     +
                       / \   / \
                      fp  8 fp -12

extreme RISC: add %fp,-12,%r3
              ld [%r3], %r7
              add %fp,8,%r4
              st %r7,[%r4]

moderate RISC:  ld [%fp-12], %r7
                st %r7, [%fp + 8]

CISC:   move [%fp-12],[%fp+8]
```

X86-64 INSTRUCTION SELECTION FOR FAB

- Assume we have permanently allocated each IR NamedReg and TempReg to an X86 register or a stack slot.
- So when generating code for an IR operation, need to be able to cope with either register or memory operand in each position
- X86 instructions are much more constrained:

`mov` s,d	d can be reg or mem; s can be reg, imm, or mem (but not if d is)
`add` s,d	$d := d + s$; d can be reg or mem, s can be reg, imm, or mem (but not if d is)
`sub` s,d	$d := d - s$; d can be reg or mem, s can be reg, imm, or mem (but not if d is)
`imul` s,d	$d := d * s$; d must be reg, s can be reg, imm, or mem
`idiv` s	`%rax` := `%rdx:%rax div` s and `%rdx` := `%rdx:%rax mod` s; s must be reg or mem
`cmp` s_1,s_2	test sense is "backwards"; s_1 can be reg, imm, or mem; s_s can be reg or mem (but not if s_1 is)

- Can use these utility functions:

```
X86.Operand gen_source_operand (IR.Source rand, X86.Size size,
                                boolean imm_ok, X86.Reg temp)
X86.Operand gen_dest_operand (IR.Dest rand, X86.Size size)
```

• Remember to keep track of sizes:

IR.Type.BOOL → X86.Size.B = 1 byte (registers `%al`, etc.)
IR.Type.INT → X86.Size.L = 4 bytes (registers `%eax`, etc.)
IR.Type.PTR → X86.Size.Q = 8 bytes (registers `%rax`, etc.)

Register names must match instruction suffixes, or you get an assembler error. Immediates and memory operands are sized automatically.

• One pattern for generating 2-addr code from 3-addr code:

```
IR.sub a,b,c        X86.mov a,c    can omit if a = c
                    X86.sub b,c
```

But what if b = c ?!?

```
IR.sub a,b,b        X86.mov a,b    can omit if a = b
                    X86.sub b,b    Oops!
```

Must be smarter in this case, or dumber in the regular case.

Hint: Use %r10 and %r11 as temporaries within IR instructions.

• There are many possible improvements, especially when dealing with constants or commutative operators.

Register Allocation and Assignment

Task: Manage scarce resources (registers) in environment with imperfect information (static program text) about dynamic program behavior.

General aim is to keep frequently-used values in registers as much as possible, to lower memory traffic. Can have a **large** effect on program performance.

Variety of approaches are possible, differing in sophistication and in scope of analysis used.

Allocator may be unable to keep every "live" variable in registers; must then "spill" variables to memory. Spilling adds new instructions, which often affects the allocation analysis, requiring a new iteration.

If spilling is necessary, what should we spill? Some heuristics:

- Don't spill variables used in inner loops.
- Spill variables not used again for "longest" time.
- Spill variables which haven't been updated since last read from memory.

LIVENESS

To assign registers effectively for a whole procedure, we need to look at the uses of each variable across expressions and statements.

To see how long to keep a given variable (or temporary) in a register, need to know the range of instructions for which the variable is **live**.

A variable or temporary is **live** immediately following an instruction if its current value will be needed in the future (i.e., it will be used again, and it won't be changed before that use).

Example:

```
                        ! temps live after instruction:
movI 3, $T2             ! $T2
movI $T2, $T3           ! $T2 $T3
addI $T3, 4, $T4        ! $T2     $T4
addI $T2, $T4, $T4      !         $T4
movI $T4, $RET          ! (nothing)
```

It's easy to calculate liveness for a consecutive series of instructions without branches, just by working backwards.

LIVENESS (CONTINUED)

But if a value can stay in a register over a jump, things get harder, e.g.:

```
                                ! temps live after instruction:
1     movI 0, $T1               ! $T1     $T3
2 L1: addI $T1, 1, $T2          !     $T2 $T3
3     addI $T3, $T2, $T3        !     $T2 $T3
4     mulI $T2, 2, $T1          ! $T1     $T3
5     cmpI $T1, 1000            ! $T1     $T3
6     jl L1                     ! $T1     $T3
7     movI $T3, $RET            ! (nothing)
```

To calculate liveness in this case requires **iterative data flow analysis** and the result is only **conservative approximation** to true liveness.

The **live range** of a variable is the set of instructions which leave it live. E.g. here live range of `$T1` is $\{1, 4, 5, 6\}$; live range of `$T3` is $\{1, \ldots, 6\}$.

Basic idea: If two variables have disjoint live ranges, they can occupy the same physical register.

So in both examples, 2 physical registers suffice to allocate all temporaries without spilling.

LINEAR SCAN ALLOCATION

Using live ranges turns out to be computationally expensive (more later).

A simple alternative is to approximate each live range by a **live interval**. This is the consecutive interval of instructions between the first and last use of each temporary. Example:

```
                              ! temps live after instruction:
1     movI 0, $T1             ! $T1     $T3
2 L1: addI $T1, 1, $T2        !     $T2 $T3
3     addI $T3, $T2, $T3      !     $T2 $T3
4     mulI $T2, 2, $T1        ! $T1     $T3
5     cmpI $T1, 1000          ! $T1     $T3
6     jl L1                   ! $T1     $T3
7     movI $T3, $RET          ! (nothing)

Live ranges: $T1: 1,4,5,6  $T2:2,3 $T3:1,2,3,4,5,6

Live intervals: $T1: [1,6]  $T2: [2,3]  $T3: [1,6]
```

(Revised) Basic idea: if two temporaries have non-overlapping live intervals, they can occupy the same physical register.

LINEAR SCAN ALLOCATION ALGORITHM DETAILS

1. Compute *startpoint[i]* and *endpoint[i]* of live interval i for each temporary. Store the intervals in a list in order of increasing start point.
2. Initialize set *active* := $\emptyset$ and pool of free registers = all usable registers.
3. For each live interval i in order of increasing start point:
   (a) For each interval j in *active*, in order of increasing end point
      - If *endpoint[j]* $\geq$ *startpoint[i]* break to step 3b.
      - Remove j from *active*.
      - Add *register[j]* to pool of free registers.

   (b) Set *register[i]* := next register from pool of free registers, and remove it from pool. (If pool is already empty, need to spill.)

   (c) Add i to *active*, sorted by increasing end point.

LINEAR SCAN ALLOCATION FOR FAB

• For HW4, the live interval computation code is already provided for you. It maps each IR.Reg operand to its live interval.

• Note that if an operand is defined but not used, its live interval is empty, and it does not appear in the map.

• When allocating a register in step 3(b), need to consider different register classes (callee-save vs. caller-save). Simple strategy: use callee-save register if operand's live range includes a call instruction; otherwise use caller-save register if available. (This way, we never have to worry about performing caller saves at all.)

• The Arg and RetReg operands must be pre-allocated to fixed registers. These registers can still be used to store other operands with non-overlapping ranges, but at step 3(b) we must **look ahead** to make sure that the interval doesn't overlap a pre-allocated interval.

• Once a spill is necessary, simply commit the spilled operand to a stack slot once and for all.

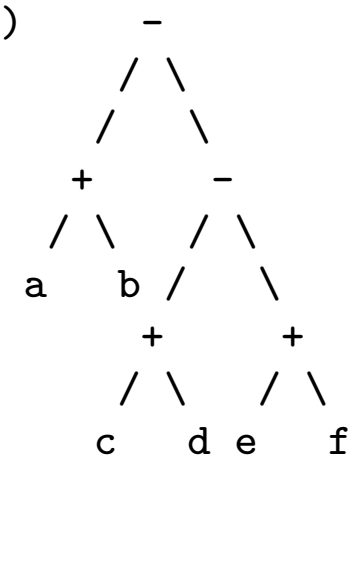
MINIMIZING DEMAND FOR REGISTERS

A compiler can sometimes make choices that reduce the **register pressure**, i.e. the number of registers needed simultaneously.

One example is choice of evaluation order when linearizing expression trees.

Example: Assume a RISC-like load-store instruction set. If we compute left child first, need 4 regs, but doing right child first needs only 3.

```
(a+b) - ((c+d) - (e+f))
```




```
load a,r1
load b,r2
add r1,r2,r1
load c,r2
load d,r3
add r2,r3,r2
load e,r3
load f,r4
add r3,r4,r3
sub r2,r3,r2
sub r1,r2,r1
```

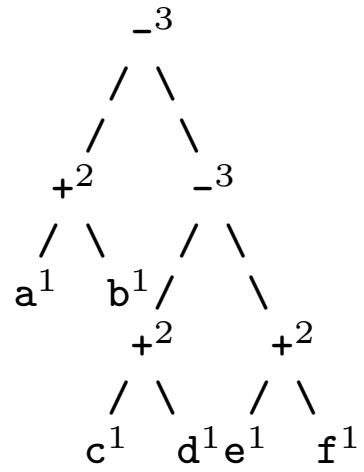
```
load c,r1
load d,r2
add r1,r2,r1
load e,r2
load f,r3
add r2,r3,r2
sub r1,r2,r1
load a,r2
load b,r3
add r2,r3,r2
sub r2,r1,r1
```

Minimizing Registers Needed to Evaluate Expression Trees

Key idea (Sethi & Ullman): At each node, **first** evaluate subtree requiring largest number of registers to evaluate. Can then save result of this evaluation in a register while doing other subtree.

1. Label each node with minimum number of registers needed to evaluate subtree.

```
risc_label(t)
  if isLeaf(t) then
    t->label = 1  (depends on machine architecture)
  else
    label(t->left)
    label(t->right)
    if (t->left->label == t->right->label)
      t->label = t->left->label + 1
    else
      t->label = max(t->left->label,
                     t->right->label)
```



2. Use labels to guide order of code emission; emit code for higher-numbered subtree **first**.

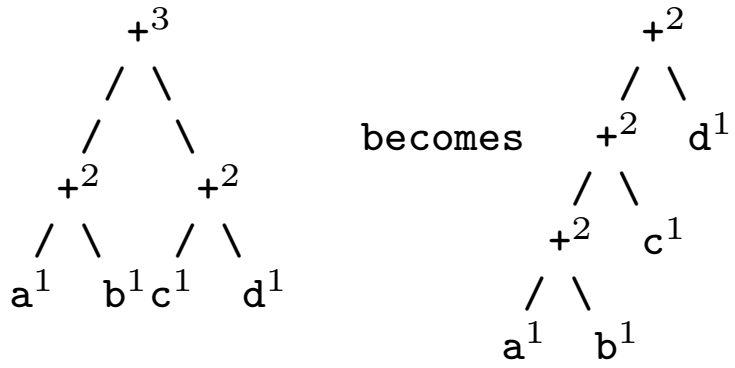
OTHER ISSUES IN TREE EVALUATION ORDER

• Some machines allow one operand to be a complex expression, while the other must be a register, which also holds the result ("accumulator" style):

```
add 37,r0      ;  r0 <- r0 + 37
add [b], r1    ;  r1 <- r1 + *b
sub r0,r1      ;  r1 <- r1 - r0
```

These machines have different Sethi-Ullman numbering, e.g., right **leaves** might require no temporary registers at all.

• Can use associativity to make trees "less bushy, " e.g.



CONTROL-FLOW GRAPHS

To compute liveness and other properties of an entire procedure, we use a **control-flow** graph.

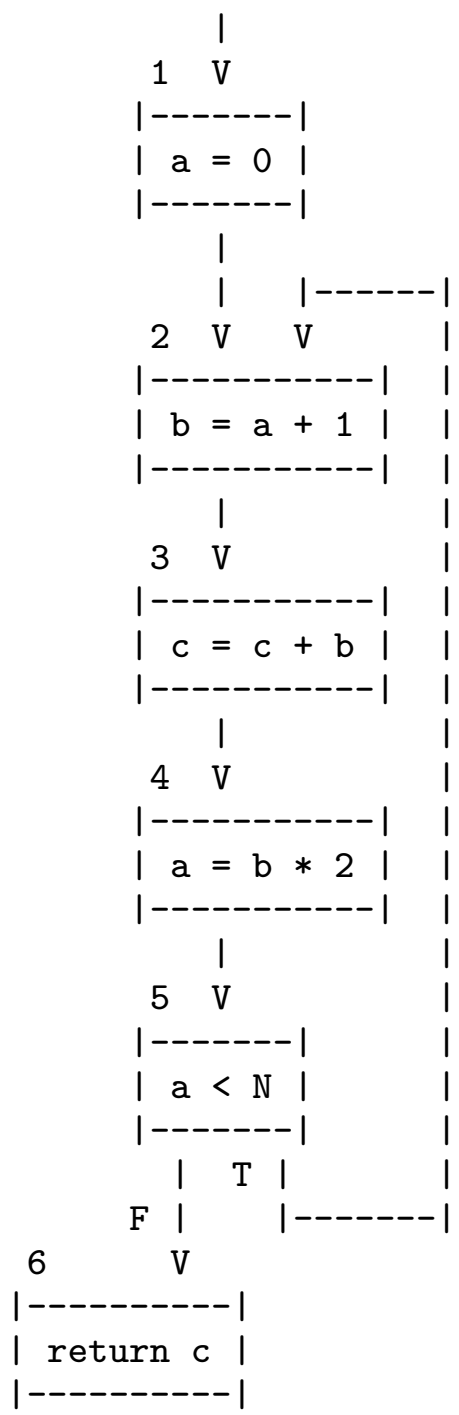
In simplest form, control flow graph has one node per statement, and an edge from n_1 to n_2 if control can ever flow directly from statement 1 to statement 2.

We write *pred*$[n]$ for the set of predecessors of node n, and *succ*$[n]$ for the set of successors.

(In practice, usually build control-flow graphs where each node is a **basic block**, rather than a single statement. More later.)

Example....

```
   a = 0
L: b = a + 1
   c = c + b
   a = b * 2
   if a < N goto L
   return c
```



LIVENESS ANALYSIS USING DATAFLOW ANALYSIS

Working from the future to the past, we can determine the **edges** over which each variable is live.

In the example:

`b` is live on $2 \rightarrow 3$ and on $3 \rightarrow 4$.

`a` is live from on $1 \rightarrow 2$, on $4 \rightarrow 5$, and on $5 \rightarrow 2$ (but not on $2 \rightarrow 3 \rightarrow 4$).

`c` is live throughout (including on entry $\rightarrow 1$).

Can see that two registers suffice to hold `a`,`b`,`c`.

DATAFLOW EQUATIONS

We can do liveness analysis (and many other analyses) via **dataflow** analysis.

A node **defines** a variable if its corresponding statement assigns to it.

A node **uses** a variable if its corresponding statement mentions that variable in an expression (e.g., on the rhs of assignment).

For any variable v, define

- *def*$[v]$ = set of graph nodes that define v
- *use*$[v]$ = set of graph nodes that use v

Similarly, for any node n, define

- *def*$[n]$ = set of variables defined by node n;
- *use*$[v]$ = set of variables used by node n.

SETTING UP EQUATIONS

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**.

A variable is **live-in** at a node if it is live on any in-edge of that node; it is **live-out** if it is live on any out-edge.

Then the following equations hold:

$$\mathit{in}[n] = \mathit{use}[n] \cup (\mathit{out}[n] - \mathit{def}[n])$$

$$\mathit{out}[n] = \bigcup_{s \in \mathit{succ}[n]} \mathit{in}[s]$$

We want the **least fixed point** of these equations: the smallest *in* and *out* sets such that the equations hold.

SOLVING FIXED-POINT EQUATIONS

We can find this solution by **iteration**:

- Start with empty sets
- Use equations to add variables to sets, one node at a time.
- Repeat until sets don't change any more.

Adding additional variables to the sets is safe, as long as the sets still obey the equations, but inaccurately suggests that more live variables exist than actually do.

EXAMPLE SOLUTION

For correctness, order in which we take nodes doesn't matter, but it turns out to be fastest to take them in roughly reverse order:

			1st		2nd		3rd	
node	*use*	*def*	*out*	*in*	*out*	*in*	*out*	*in*
6	c			c		c		c
5	a		c	ac	ac	ac	ac	ac
4	b	a	ac	bc	ac	bc	ac	bc
3	bc	c	bc	bc	bc	bc	bc	bc
2	a	b	bc	ac	bc	ac	bc	ac
1		a	ac	c	ac	c	ac	c

Implementation issues:

- Algorithm always terminates, because each iteration must enlarge at least one set, but sets are limited in size (by total number of variables).
- Time complexity is $O(N^4)$ worst-case, but between $O(N)$ and $O(N^2)$ in practice.

BASIC BLOCKS

The control flow between adjacent non-branching instructions is trivial. We can use this fact to reduce the number of nodes required for a control flow graph, and hence speed up (by a constant factor) the algorithms that work on it.

- **Basic Block** = sequence of instructions with single entry & exit.
- If first instruction of BB is executed, so is remainder of block (in order).
- To calculate basic blocks:

(1) Determine BB **leaders** ($\rightarrow$) :

(a) First statement in routine
(b) Target of any jump (conditional or unconditional).
(c) Statement following any jump.

(What about subroutine calls?)

(2) Basic block extends from leader to (but not including) next leader (or end of routine).

Basic Block Example

```
prod := 0;
i := 1;
while i <= 20 do
  prod := prod + a[i] * b[i];
  i := i + 1
end
```

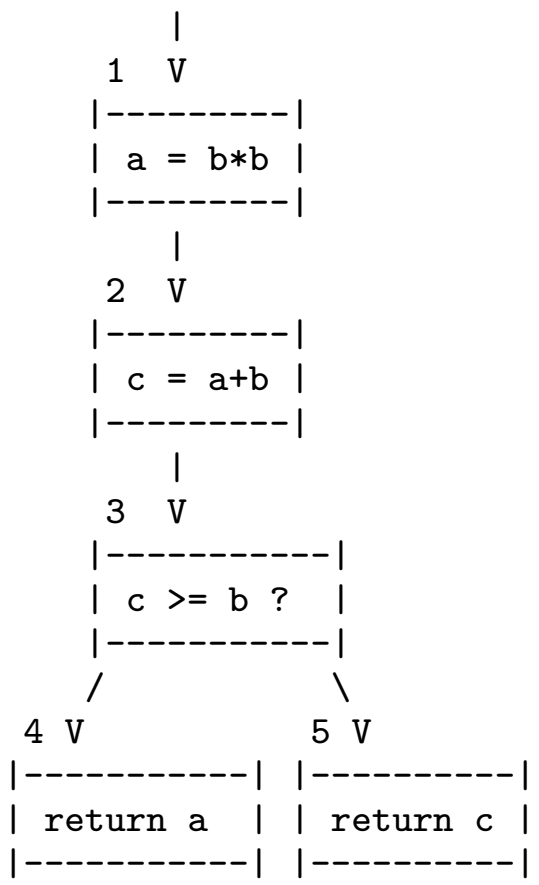
```
→   1. prod := 0
    2. i := 1
→   3. if i > 20 goto 14
→   4. t1 := i * 4
    5. t2 := addr a
    6. t3 := *(t2+t1)
    7. t4 := i * 4
    8. t5 := addr b
    9. t6 := *(t5+t4)
   10. t7 := t3 * t6
   11. prod := prod + t7
   12. i := i + 1
   13. goto 3
→  14. ---
```

Once we've discovered the basic blocks, we build a CFG in which each node is an entire basic block.

Most dataflow algorithms reduce to very simple form within a block. For example, for liveness, it is easy to compute the *in* and *out* sets of each block by a simple backward pass over its instructions.

STATIC VS. DYNAMIC LIVENESS

Consider the following graph:



Is a live-out at node 2? It depends on whether control flow ever reaches node 4.

STATIC LIVENESS (CONTINUED)

A smart compiler could answer no.

A smarter compiler could answer similar questions about more complicated programs.

But **no** compiler can ever **always** answer such questions correctly. This is a consequence of the **uncomputability** of the **Halting Problem**.

So we must be content with **static** liveness, which talks about paths of control-flow edges, and is just a **conservative** approximation of **dynamic liveness**, which talks about actual execution paths.

HALTING PROBLEM

Theorem There is no program H that takes an input any program P and input X, and (without infinite-looping) returns true if $P(X)$ halts and false if $P(X)$ infinite-loops.

Proof Suppose there were such an H. From it, construct the function

$F(Y) =$ `if` $H(Y,Y)$ `then (while true do ()) else 1`

Now consider $F(F)$.

• If $F(F)$ halts, then, by the definition of H, $H(F,F)$ is true, so the `then` clause executes, so $F(F)$ does not halt.

• But,if $F(F)$ loops forever, then $H(F,F)$ is false, so the `else` clause is taken, so $F(F)$ halts.

Hence $F(F)$ halts if any only if it doesn't halt.

Since we've reached a contradiction, the initial assumption is wrong: there can be no such H.

REACHABILITY PROBLEM

Corollary No program $H'(P, X, L)$ can tell, for any program P, input X, and label L within P, whether L is ever reached on an execution of P on X.

Proof If we had H', we could construct H. Consider a program transformation T that, from any program P constructs a new program by putting a label L at the end of the program, and changing every `halt` to `goto` L. Then $H(P, X) = H'(T(P), X, L)$.

REGISTER INTERFERENCE GRAPHS

Recall that linear scan allocation uses a very conservative approximation of each register's lifetime by reducing it to a single live interval.

A more precise approach is to use the live range set, e.g.:

```
                                Live after instr.
ld a,t0              ; a:t0     t0
ld b,t1              ; b:t1     t0 t1
sub t0,t1,t2         ; t:t2     t0    t2
ld c,t3              ; c:t3     t0    t2 t3
sub t0,t3,t4         ; u:t4           t2    t4
add t2,t4,t5         ; v:t5                 t4 t5
add t5,t4,t6         ; d:t6                       t6
st t6,d
```

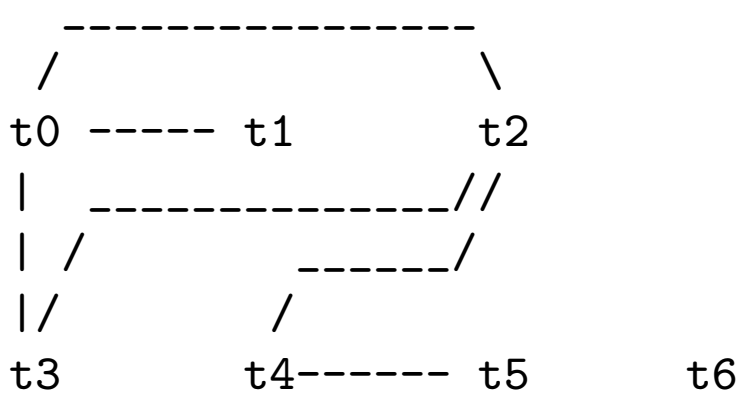
The resulting demand on registers can be described by a **register interference graph**, which has

- a node for each logical register.

- an edge between two nodes if the corresponding registers are simultaneously live.

COLORING INTERFERENCE GRAPHS

Interference Graph Example:



A **coloring** of a graph is an assignment of colors to nodes such that no two connected nodes have the same color. (Like coloring a map, where nodes=countries and edges connect countries with common border.)

Suppose we have k physical registers available. Then aim is to color interference graph with k or fewer colors. This implies we can allocate logical registers to physical registers without spilling.

In general case, determining whether a graph can be k-colored is hard (N.P. Complete, and hence probably exponential).

But a simple heuristic will **usually** find a k-coloring if there is one.

GRAPH COLORING HEURISTIC

1. Choose a node with fewer than k neighbors.

2. Remove that node. Note that if we can color the resulting graph with k colors, we can also color the original graph, by giving the deleted node a color different from all its neighbors.

3. Repeat until **either**

• there are no nodes with fewer than k neighbors, in which case we must spill; **or**

• the graph is gone, in which case we can color the original graph by adding the deleted nodes back in one at a time and coloring them.

In our example, heuristic finds a 3-coloring. There cannot be a 2-coloring (why not?).

Each “color” corresponds to a physical register, so 3 registers will do for this example.