CS 473: COMPILER DESIGN
REGISTER ALLOCATION
Register Allocation Problem

• Given: an IR program that uses an unbounded number of variables

• Find: a mapping from variables to machine registers such that
  – the program behaves the same when temps are replaced with registers
  – as many variables as possible are in registers
  – moves between registers (like `move r1, r2`) are minimized
  – architecture requirements are obeyed (e.g., zero register always has the value 0, argument registers are overwritten on function calls)

• Stack Spilling
  – If there are $k$ registers available and $m > k$ variables are live at the same time, then not all of them will fit into registers
  – So we must "spill" the excess variables to the stack
Register Allocation by Graph Coloring

- Liveness analysis tells us which variables are live at each point in a program
- We can’t assign two variables to the same register if they’re live at the same time
- So we want to match every variable with a register such that no two variables that are live at the same time get the same register

- This is a graph coloring problem! (where colors are registers)
- Graph coloring: color all the nodes of a graph one of $k$ different colors so that no two adjacent nodes have the same color
- Register allocation: assign all the variables in a program to one of $k$ different registers so that no two variables live at the same time have the same register
Nodes of the graph are variables
Edges connect variables that *interfere* with each other
  - Two variables interfere if their live ranges intersect
Register assignment is a *graph coloring*
  - A graph coloring assigns each node in the graph a color (register)
  - Any two nodes connected by an edge must have different colors
Exercise: Color this graph so that no 2 connected nodes have the same color. How many colors are required?

```plaintext
// live = {a}
b1 = addi a, 2 // live = {a, b1}
c = mul b1, b1 // live = {a, c}
b2 = addi c, 1 // live = {a, b2}
ans = mul b2, a // live = {ans}
return ans
```

Interference Graph

2-Coloring of the graph
red = r8
yellow = r9
Register Allocation Questions

• How many colors do we have?
  – As many as there are registers

• Can we efficiently find a $k$-coloring of the graph whenever possible?
  – Answer: in general the problem is NP-complete
  – But we can do an efficient approximation using heuristics

• How do we assign registers to colors?
  – With a clever approach, we can eliminate some extra move instructions

• What do we do when there aren’t enough colors/registers?
  – We have to use stack space, but how do we do this effectively?
Algorithm for $k$-coloring a graph by Kempe [1879]

Recursive algorithm with three steps:

- **Step 1:** Find a node with degree $< k$ and cut it out of the graph
  - It has fewer neighbors than there are colors, so we’ll always have a color left for it once we color the rest of the graph
  - Remove the node and all its edges
  - This is called *simplifying* the graph

- **Step 2:** Recursively $k$-color the remaining subgraph

- **Step 3:** When remaining graph is colored, there must be at least one free color available for the deleted node (since its degree was $< k$). Color the node with one of those free colors.
Example: 3-coloring a graph

1 and 2: recurse down the simplified graphs
Example: 3-color this Graph

3: assign colors on the way back up
Failure of the Algorithm

• If the graph cannot be colored, it will simplify to a graph where every node has at least $k$ neighbors.
  – This might also happen even when the graph can be colored!
  – This is a symptom of NP-hardness – we’d need to try every possibility to always get the best answer

• Example: When trying to 3-color this graph:

![Graph diagram]
Questions
Spilling

• Idea: If we can’t $k$-color the graph, we need to store (at least) one temporary variable on the stack.

• Which variable to spill?
  – Pick one that isn’t used very frequently
  – Pick one that isn’t used in a (deeply nested) loop
  – Pick one that has high interference (since removing it will make the graph easier to color)

• In practice: some weighted combination of these criteria

• When coloring:
  – Mark the node as spilled
  – Remove it from the graph
  – Keep recursively coloring
Spilling Example

- If no nodes have degree $< k$, select a node to spill
- Mark it and remove it from the graph
- Continue coloring
Optimistic Coloring

- Recall that Kempe’s algorithm sometimes gets stuck even when the graph is \(k\)-colorable.
- So sometimes it’s actually possible to color a node marked for spilling!

Example: When 2-coloring this graph:

- Even though the node was marked for spilling, we can color it yellow.
- So: on the way down, mark for spilling, but don’t actually spill. On the way back up, if the marked node has a color available, color it instead!
• Suppose \( t \) is marked for spilling to stack slot \( fp + o \)

\[
t = a \, \text{op} \, b
\]
\[
\ldots
\]
\[
x = t \, \text{op} \, c
\]
\[
\ldots
\]
\[
y = d \, \text{op} \, t
\]
\[
\text{// def of } t
\]
\[
\text{// use 1 of } t
\]
\[
\text{// use 2 of } t
\]
Example Spill Code

• Suppose $t$ is marked for spilling to stack slot $fp + o$

\[
t = a \text{ op } b \\
[fp + o] = a \text{ op } b \quad // \text{ def of } t
\]

\[
x = t \text{ op } c \\
x = [fp + o] \text{ op } c \quad // \text{ use 1 of } t
\]

\[
y = d \text{ op } t \\
y = d \text{ op } [fp + o] \quad // \text{ use 2 of } t
\]

• But these instructions don’t exist in most assembly languages!
• Exercise: How would you write this code if we can’t access memory and do \text{ op} in a single instruction?
• We still need to use registers to load from/store to stack variables!
• Suppose $t$ is marked for spilling to stack slot $fp + o$

• Approach 1: reserve r1 and r2 for stack

\[ t = a \ op \ b \]
\[ [fp + o] = a \ op \ b \quad \// \ def \ of \ t \]
...

\[ x = t \ op \ c \]
\[ x = [fp + o] \ op \ c \quad \// \ use \ 1 \ of \ t \]
...

\[ y = d \ op \ t \]
\[ y = d \ op \ [fp + o] \quad \// \ use \ 2 \ of \ t \]
Example Spill Code

• Suppose \( t \) is marked for spilling to stack slot \( fp + o \)
• Approach 1: reserve \( r1 \) and \( r2 \) for stack

\[
\begin{align*}
t = a \text{ op } b \\
\ldots \\
r1 = a \text{ op } b \\
[fp + o] = r1 \\
x = t \text{ op } c \\
\ldots \\
\rightarrow r1 = [fp + o] \\
x = r1 \text{ op } c \\
y = d \text{ op } t \\
r1 = [fp + o] \\
y = d \text{ op } r1 \\
\end{align*}
\]
Example Spill Code

• Suppose \( t \) is marked for spilling to stack slot \( fp + o \)

• Approach 1: reserve \( r1 \) and \( r2 \) for stack

\[
\begin{align*}
t &= a \text{ op } b \\
&\vdots \\
x &= t \text{ op } c \\
&\vdots \\
y &= d \text{ op } t
\end{align*}
\]

\[
\begin{align*}
&\text{r1 = a \text{ op } b} \\
&[fp + o] = r1 \\
&\text{r1 = [fp + o]} \\
&x = r1 \text{ op } c \\
&\text{r1 = [fp + o]} \\
&\text{r2 = [fp + p]} \\
&y = r2 \text{ op } r1
\end{align*}
\]

• If \( d \) is also on the stack, at \( fp + p \)

• In general, we need a register for every operand an instruction can have (often just 2)
Example Spill Code

• Suppose \( t \) is marked for spilling to stack slot \( fp + o \)

• Approach 2: make a new variable for each access to \( t \)

\[
\begin{align*}
  t &= a \text{ op } b \\
  \quad \cdots \\
  t1 &= a \text{ op } b \\
  &\quad \text{// def of } t \\
  [fp + o] &= t1 \\
  x &= t \text{ op } c \\
  \quad \cdots \\
  t2 &= [fp + o] \\
  &\quad \text{// use 1 of } t \\
  x &= t2 \text{ op } c \\
  y &= d \text{ op } t \\
  t3 &= [fp + o] \\
  &\quad \text{// use 2 of } t \\
  y &= d \text{ op } t3
\end{align*}
\]

• Where \( t1, t2, t3 \) are freshly generated temporaries that replace \( t \) for different uses of \( t \)

• Why does this work? We’ve just introduced even more variables that we need to allocate to registers!
  – Because each one is only live for a very short time.
Accessing Spilled Registers

• We can’t usually do operations directly on values in memory, so we need to load stack variables into registers when we use them!

• Approach 1: Reserve specific registers for loading/storing to spilled variables
  – Pro: Only need to color the graph once
  – Con: Need at least two registers (one for each source operand of an instruction), so decreases total # of available registers by 2
  – Not good on x86 (especially 32-bit) because there are too few registers & too many constraints on how they can be used

• Approach 2: Rewrite the program to use a new temporary variable for each access to a spilled variable, and then do register allocation again
  – Pro: Need to reserve fewer registers
  – Con: Introducing a variable changes live ranges, so must recompute liveness & recolor graph
Questions
Precolored Nodes

• Some variables must be pre-assigned to registers
  – Most processors have a dedicated frame pointer register for fp
  – Most instruction sets reserve certain registers for passing function arguments (a0-a3 in MIPS)
  – We can still assign other variables to these registers, too, as long as they’re available when we need them!

• To properly allocate temporaries, we can treat registers as nodes in the interference graph with pre-assigned colors
  – Pre-colored nodes can’t be removed during simplification, and should never be spilled
  – Implementation trick: Treat pre-colored nodes as having infinite degree in the interference graph (so their degree is always > k)
  – When the graph is empty except for the pre-colored nodes, then we start coloring the rest of the nodes.
Picking Good Colors

• When choosing colors during the coloring phase, we can choose any color that isn’t assigned to an adjacent nodes, and some choices are better for performance than others.

• In particular, if we have move $t_1, t_2$ and assign $t_1$ and $t_2$ to the same register, the move is redundant and can be eliminated
  – Note that $t_1$ and $t_2$ probably don’t interfere with each other – there’s no reason to keep using both once they have the same value

• A simple color choosing strategy that helps eliminate such moves:
  – Add a new kind of “move-related” edge between the nodes for $t_1$ and $t_2$ in the interference graph
  – When choosing a color for $t_1$ (or $t_2$), if possible pick the color of an already-colored node reachable by a move-related edge
Example Color Choice

• Consider 3-coloring this graph, where the dashed edge indicates that there is a move from one temporary to another

• After coloring the rest, we have a choice
  – Picking yellow is better than red because it will eliminate a move
Coalescing Interference Graphs

• A more aggressive strategy is to coalesce nodes of the interference graph into a single node if they are connected by move-related edges.
  – Coalescing the nodes forces the two temporaries to be assigned the same register

![Diagram of coalescing interference graph nodes](image)

• Idea: interleave simplification and coalescing to maximize the number of moves that can be eliminated
• Problem: coalescing can sometimes increase the degree of a node

![Diagram of coalescing interference graph nodes](image)
Conservative Coalescing

• Two strategies are guaranteed to preserve the $k$-colorability of the interference graph:

  • **Brigg’s strategy**: It's safe to coalesce $x$ and $y$ if the resulting node will have fewer than $k$ neighbors (with degree $\geq k$).

  • **George’s strategy**: We can safely coalesce $x$ and $y$ if for every neighbor $t$ of $x$, either $t$ already interferes with $y$ or $t$ has degree $< k$. 
Two strategies are guaranteed to preserve the $k$-colorability of the interference graph:

- **Brigg’s strategy**: It's safe to coalesce $x$ and $y$ if the resulting node will have fewer than $k$ neighbors (with degree $\geq k$).

- **George’s strategy**: We can safely coalesce $x$ and $y$ if for every neighbor $t$ of $x$, either $t$ already interferes with $y$ or $t$ has degree $< k$. 
Two strategies are guaranteed to preserve the $k$-colorability of the interference graph:

- **Brigg’s strategy**: It's safe to coalesce $x$ and $y$ if the resulting node will have fewer than $k$ neighbors (with degree $\geq k$).

- **George’s strategy**: We can safely coalesce $x$ and $y$ if for every neighbor $t$ of $x$, either $t$ already interferes with $y$ or $t$ has degree $< k$. 
Questions

Top
Nodes of the graph are variables

Edges connect variables that \textit{interfere} with each other
  
  Two variables interfere if their live ranges intersect

Register assignment is a \textit{graph coloring}
  
  A graph coloring assigns each node in the graph a color (register)
  
  Any two nodes connected by an edge must have different colors

// live = \{a\}
b1 = addi a, 2
// live = \{a, b1\}
c = mul b1, b1
// live = \{a, c\}
b2 = addi c, 1
// live = \{a, b2\}
ans = mul b2, a
// live = \{ans\}
return ans

Interference Graph

2-Coloring of the graph
red = r8
yellow = r9
Complete Register Allocation Algorithm

1. Build interference graph from liveness information, with precolored nodes and move-related edges

2. Reduce the graph (building a stack of nodes to color)
   1. Simplify the graph by removing nodes with degree < $k$ that aren’t move-related; remaining nodes are high-degree or move-related
   2. Coalesce move-related nodes using Brigg’s or George’s strategy
   3. Repeat 2.1 and 2.2 until no node can be simplified or coalesced
   4. If no nodes can be coalesced, remove a move-related edge and keep trying to simplify/coalesce

3. If there are non-precolored nodes left, they have degree $\geq k$: mark one for spilling, remove it from the graph, and go back to step 2

4. When only pre-colored nodes remain, start coloring: pop simplified nodes off the stack and give each one a color its neighbors don’t have
   1. If a node must be spilled, insert spill code and rerun the whole register allocation algorithm starting at step 1

5. After register allocation, the compiler should do an optimization pass to remove redundant moves (like `move r1, r1`)
Register Allocation: Summary

• Once we have liveness information, we can build an *interference graph* showing which variables are live at the same time.

• Coloring the graph (so that no two connected nodes have the same color) with *k* colors corresponds to allocating the variables to *k* machine registers.

• If we can’t color the graph fully, we have to *spill* a variable to the stack and then try again.

• Graph coloring is NP-complete, so we might end up spilling more variables than necessary.

• There are a few tricks for getting efficient allocations (move-related edges, coalescing).

• Once we’ve done this to the output of instruction selection, we’ve translated all the way from source language to real assembly!
Questions