Global Register Allocation
drawbacks of local register allocation

- Why do we have to store all live/dirty registers at the end of a basic block?
  - Because we only consider per-basic block register allocation, and information may not match across basic blocks
- Consider the CFG

  ![CFG Diagram]

- In BB1, x is mapped to r1, in BB2, x is mapped to r2
- What should x be mapped to in BB3?
global register allocation

• To make sure that a temporary/local/global has consistent mapping across basic blocks, we want to assign that variable to a register for the entire function

• Isn’t this kind of like our naïve register allocation approach?

• Key: a register might have multiple variables assigned to it

• All variables with the same color can be assigned to the same register in this code

1:  \( T_1 = A + B \)
2:  \( T_2 = A + T_1 \)
3:  \( T_3 = A + T_2 \)
4:  \( D = A + T_3 \)
5:  \( T_4 = C + B \)
6:  \( T_5 = T_4 + C \)
7:  \( E = T_5 + D \)
global register allocation

• Issues:
  • How do we know that two variables can be assigned to the same register?
  • How do we find the right assignment of variables to registers?
  • What do we do if we don’t have enough registers to make the assignment?

1: \( T_1 = A + B \)
2: \( T_2 = A + T_1 \)
3: \( T_3 = A + T_2 \)
4: \( D = A + T_3 \)
5: \( T_4 = C + B \)
6: \( T_5 = T_4 + C \)
7: \( E = T_5 + D \)
co-locating variables

- Two variables can be assigned to the same register if they are not live at the same time
- They don’t have values we need at the same time

1: \( T1 = A + B \)
2: \( T2 = A + T1 \)
3: \( T3 = A + T2 \)
4: \( D = A + T3 \)
5: \( T4 = C + B \)
6: \( T5 = T4 + C \)
7: \( E = T5 + D \)
co-locating variables

- Two variables can be assigned to the same register if they are not live at the same time.
- They don’t have values we need at the same time.

\[
\begin{align*}
1: \quad T1 &= A + B \quad [A, B] \\
2: \quad T2 &= A + T1 \quad [A, B, T1] \\
3: \quad T3 &= A + T2 \quad [A, B, T2] \\
4: \quad D &= A + T3 \quad [B, C, D] \\
5: \quad T4 &= C + B \quad [T4, C, D] \\
6: \quad T5 &= T4 + C \quad [T5, D] \\
7: \quad E &= T5 + D \quad [E]
\end{align*}
\]
making the right assignments

- Just because you make sure that variables that are not live at the same time do not go in the same register doesn’t mean you make the right assignments

1: $T_1 = A + B$ [A, B, T1]
2: $T_2 = A + T_1$ [A, B, T2]
3: $T_3 = A + T_2$ [A, B, T3]
4: $D = A + T_3$ [B, C, D]
5: $T_4 = C + B$ [T4, C, D]
6: $T_5 = T_4 + C$ [T5, D]
7: $E = T_5 + D$ [E]
making the right assignments

• Just because you make sure that variables that are not live at the same time do not go in the same register doesn’t mean you make the right assignments.

• If we put all the temporaries in the same register, then we need an extra register for C.

\[
\begin{align*}
1: & \quad T1 = A + B \quad [A, B] \\
2: & \quad T2 = A + T1 \quad [A, B, T1] \\
3: & \quad T3 = A + T2 \quad [A, B, T2] \\
4: & \quad D = A + T3 \quad [B, C, D] \\
5: & \quad T4 = C + B \quad [T4, C, D] \\
6: & \quad T5 = T4 + C \quad [T5, D] \\
7: & \quad E = T5 + D \quad [E]
\end{align*}
\]
making the right assignments

- Just because you make sure that variables that are not live at the same time do not go in the same register doesn’t mean you make the right assignments.
- If we put all the temporaries in the same register, then we need an extra register for C.

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>$T1 = A + B$</td>
</tr>
<tr>
<td>2</td>
<td>$T2 = A + T1$</td>
</tr>
<tr>
<td>3</td>
<td>$T3 = A + T2$</td>
</tr>
<tr>
<td>4</td>
<td>$D = A + T3$</td>
</tr>
<tr>
<td>5</td>
<td>$T4 = C + B$</td>
</tr>
<tr>
<td>6</td>
<td>$T5 = T4 + C$</td>
</tr>
<tr>
<td>7</td>
<td>$E = T5 + D$</td>
</tr>
</tbody>
</table>
Graph Coloring
Two variables can be assigned to the same register if they are not live at the same time.

They don’t have values we need at the same time.

Graph $G = (V, E)$ where

- $V =$ variable/temporary
- $E = (v_1, v_2)$ if $v_1$ and $v_2$ are live at the same time

1. $T_1 = A + B$
2. $T_2 = A + T_1$
3. $T_3 = A + T_2$
4. $D = A + T_3$
5. $T_4 = C + B$
6. $T_5 = T_4 + C$
7. $E = T_5 + D$
1: \( T_1 = A + B \) [A, B]
2: \( T_2 = A + T_1 \) [A, B, T1]
3: \( T_3 = A + T_2 \) [A, B, T2]
4: \( D = A + T_3 \) [B, C, D]
5: \( T_4 = C + B \) [T4, C, D]
6: \( T_5 = T_4 + C \) [T5, D]
7: \( E = T_5 + D \) [E]
graph coloring

- Assign variables to registers by *coloring* the graph, one color per register

1: \( T_1 = A + B \) [A, B, T1]
2: \( T_2 = A + T_1 \) [A, B, T2]
3: \( T_3 = A + T_2 \) [A, B, T3]
4: \( D = A + T_3 \) [B, C, D]
5: \( T_4 = C + B \) [T4, C, D]
6: \( T_5 = T_4 + C \) [T5, D]
7: \( E = T_5 + D \) [E]
graph coloring

- No vertices that share an edge get the same color

1: \( T_1 = A + B \) [A, B]
2: \( T_2 = A + T_1 \) [A, B, T1]
3: \( T_3 = A + T_2 \) [A, B, T2]
4: \( D = A + T_3 \) [A, B, T3]
5: \( T_4 = C + B \) [B, C, D]
6: \( T_5 = T_4 + C \) [T4, C, D]
7: \( E = T_5 + D \) [T5, D]
graph coloring

- Multiple valid coloring. Looking for **minimal coloring**

```
1: T1 = A + B
2: T2 = A + T1
3: T3 = A + T2
4: D = A + T3
5: T4 = C + B
6: T5 = T4 + C
7: E = T5 + D
```

---

A
B
C
D
E
T1
T2
T3
T4
T5
Graph Coloring Theory
graph coloring theory

• How do we color graphs?

1: $T_1 = A + B$ [A, B]
2: $T_2 = A + T_1$ [A, B, T1]
3: $T_3 = A + T_2$ [A, B, T2]
4: $D = A + T_3$ [B, C, D]
5: $T_4 = C + B$ [T4, C, D]
6: $T_5 = T_4 + C$ [T5, D]
7: $E = T_5 + D$ [E]
graph coloring theory

- Can we find a coloring of a graph whenever possible?
- Can we efficiently find the optimum coloring of a graph?
Can we find a coloring of a graph whenever possible?

Can we efficiently find the optimum coloring of a graph?

Problem: optimal graph coloring is **NP-hard**

(Decision problem: can a graph be colored with K or fewer colors?)
**kempe’s algorithm**

- Algorithm from 1879 for finding a $K$-coloring of a graph

- **Step 1: simplify**
  - Find a node with at most $K-1$ edges and remove it from the graph
  - Remember this node on the stack

- Observation: if smaller graph can be colored, bigger graph can be colored too (why?)
kempe’s algorithm

• Algorithm from 1879 for finding a K-coloring of a graph
• Step 1: simplify
  • Find a node with at most K-1 edges and remove it from the graph
  • Remember this node on the stack

• Observation: if smaller graph can be colored, bigger graph can be colored too (why?)
kempe’s algorithm

- Algorithm from 1879 for finding a K-coloring of a graph
- Step 2: **color**
  - Once smaller graph has been colored, add node back in
  - Assign a color
kempe’s algorithm

- Algorithm from 1879 for finding a K-coloring of a graph
- Step 2: **color**
  - Once smaller graph has been colored, add node back in
  - Assign a color
kempe’s algorithm

• Algorithm from 1879 for finding a K-coloring of a graph

• Step 2: **color**
  • Once smaller graph has been colored, add node back in
  • Assign a color
kempe’s algorithm

- Algorithm from 1879 for finding a K-coloring of a graph

- Apply steps 1 and 2 recursively:
  - Reduce graph
  - Color reduced graph if fewer than K vertices
  - Add nodes back into graph in reverse order they were removed
• Algorithm from 1879 for finding a $K$-coloring of a graph
• Apply steps 1 and 2 recursively:
  • Reduce graph
  • Color reduced graph if fewer than $K$ vertices
  • Add nodes back into graph in reverse order they were removed
Example

- Algorithm from 1879 for finding a K-coloring of a graph
- Apply steps 1 and 2 recursively:
  - Reduce graph
  - Color reduced graph if fewer than K vertices
  - Add nodes back into graph in reverse order they were removed
• Algorithm from 1879 for finding a K-coloring of a graph

• Apply steps 1 and 2 recursively:
  • Reduce graph
  • Color reduced graph if fewer than K vertices
  • Add nodes back into graph in reverse order they were removed
• Algorithm from 1879 for finding a K-coloring of a graph

• Apply steps 1 and 2 recursively:
  • Reduce graph
  • Color reduced graph if fewer than K vertices
  • Add nodes back into graph in reverse order they were removed
• Algorithm from 1879 for finding a K-coloring of a graph
• Apply steps 1 and 2 recursively:
  • Reduce graph
  • Color reduced graph if fewer than K vertices
  • Add nodes back into graph in reverse order they were removed
• Algorithm from 1879 for finding a K-coloring of a graph
• Apply steps 1 and 2 recursively:
  • Reduce graph
  • Color reduced graph if fewer than K vertices
  • Add nodes back into graph in reverse order they were removed
• Algorithm from 1879 for finding a \( K \)-coloring of a graph
• Apply steps 1 and 2 recursively:
  • Reduce graph
  • Color reduced graph if fewer than \( K \) vertices
  • Add nodes back into graph in reverse order they were removed

![Graph example]
- Algorithm from 1879 for finding a K-coloring of a graph
- Apply steps 1 and 2 recursively:
  - Reduce graph
  - Color reduced graph if fewer than K vertices
  - Add nodes back into graph in reverse order they were removed
example

- Algorithm from 1879 for finding a K-coloring of a graph
- Apply steps 1 and 2 recursively:
  - Reduce graph
  - Color reduced graph if fewer than K vertices
  - Add nodes back into graph in reverse order they were removed
Algorithm from 1879 for finding a K-coloring of a graph

Apply steps 1 and 2 recursively:

- Reduce graph
- Color reduced graph if fewer than K vertices
- Add nodes back into graph in reverse order they were removed
• Algorithm from 1879 for finding a K-coloring of a graph
• Apply steps 1 and 2 recursively:
  • Reduce graph
  • Color reduced graph if fewer than K vertices
  • Add nodes back into graph in reverse order they were removed
example

- Algorithm from 1879 for finding a $K$-coloring of a graph
- Apply steps 1 and 2 recursively:
  - Reduce graph
  - Color reduced graph if fewer than $K$ vertices
  - Add nodes back into graph in reverse order they were removed
• Algorithm from 1879 for finding a K-coloring of a graph
• Apply steps 1 and 2 recursively:
  • Reduce graph
  • Color reduced graph if fewer than K vertices
  • Add nodes back into graph in reverse order they were removed
• Algorithm from 1879 for finding a K-coloring of a graph
• Apply steps 1 and 2 recursively:
  • Reduce graph
  • Color reduced graph if fewer than K vertices
  • Add nodes back into graph in reverse order they were removed
• Algorithm from 1879 for finding a K-coloring of a graph
• Apply steps 1 and 2 recursively:
  • Reduce graph
  • Color reduced graph if fewer than K vertices
  • Add nodes back into graph in reverse order they were removed
• Algorithm from 1879 for finding a K-coloring of a graph
• Apply steps 1 and 2 recursively:
  • Reduce graph
  • Color reduced graph if fewer than K vertices
  • Add nodes back into graph in reverse order they were removed
does this always work?

- What if there isn’t a node to remove in step 1?
- Doesn’t mean the graph can’t be colored
does this always work?

• What if there isn’t a node to remove in step 1?
• Doesn’t mean the graph can’t be colored
does this always work?

• Modified algorithm:
  • If no node can be safely removed, pick one anyway, mark it as a potential spill
  • Keep going

• If graph still can’t be colored, need to deal with potential spill
Dealing with spills
does this always work?

- Modified algorithm:
  - If no node can be safely removed, pick one anyway, mark it as a potential spill
  - Keep going

- If graph still can’t be colored, need to deal with potential spill
does this always work?

- Modified algorithm:
  - If no node can be safely removed, pick one anyway, mark it as a **potential spill**
  - Keep going

- If graph *still* can’t be colored, need to deal with potential spill
what do we do?

- If a variable cannot be assigned to a register, it needs to be placed on the stack
- Need to generate extra instructions to load/store from stack — those instructions need registers too!
- Naïve approach: reserve registers for managing spills
- Better approach: rewrite code
code rewriting

• Assign spilled temporary to memory location (e.g., T2)
• Introduce a new temporary for each instruction that uses T2

T2 = T0 + T1

becomes

T19 = T0 + T1
SW T19, [stack location of T2]
code rewriting

• Assign spilled temporary to memory location (e.g., T2)
• Introduce a new temporary for each instruction that uses T2

T1 = T2 + T3

becomes

LW T37, [stack location of T2]
T1 = T37 + T3
code rewriting

- Assign spilled temporary to memory location (e.g., T2)
- Introduce a new temporary for each instruction that uses T2
- Rerun liveness, register allocation algorithm
code rewriting

• Why does this help?
• T2 is eliminated from the graph entirely
• Newly introduced temporaries have very short live ranges, so not too many edges!
• Less likely to have spills

• This is an example of live range splitting
  • Lots of refinement to reduce loads/stores
Global register allocation allows for variables to be mapped to the same register across basic blocks.

Live range splitting allows for efficient generation of spill code.

Graph coloring-based allocation is effective but potentially slow:
- Iterative algorithm that keeps rewriting code, recomputing liveness, redoing allocation.

Many modern compilers, especially JITs, use simpler, but potentially less-efficient register allocators (e.g., linear-scan register allocation).
what do we have?

• We now have a full-featured language:
  • Arithmetic operations
  • Control flow
  • Functions
• And compiler:
  • Code generation
  • Register allocation

• Good base to keep adding features!