Register Allocation

Introduction
Local Register Allocators

Copyright 2016, Pedro C. Diniz, all rights reserved.
Students enrolled in the Compilers class at the University of Southern California have explicit permission to make copies of these materials for their personal use.
Outline

• What is Register Allocation and Its Importance
• Simple Register Allocators
• Webs
• Interference Graphs
• Graph Coloring
• Splitting
• More Transformations
What is Register Allocation?

Part of the Compiler’s Back End

Critical Properties

- Produce **Correct** Code that Uses $k$ (or fewer) Registers
- Minimize Added Loads and Stores
- Minimize Space Used to Hold *Spilled Values*
- Operate Efficiently
  
  $O(n)$, $O(n \log_2 n)$, maybe $O(n^2)$, but not $O(2^n)$
Register Allocation & Assignment

At Each Point in the Code
1. Allocation: Determine which Values will reside in Registers
2. Assignment: Select a Register for each such value

The Goal:
Allocation that “Minimizes” Running Time

Optimal global allocation is NP-Complete, under almost any assumptions.
Importance of Register Allocation

• **Optimally** Use of one of the Most Critical Processor Resources
  – Affects almost every statement of the program
  – Register accesses are much faster than memory accesses
    • Eliminates expensive memory instructions
    • Wider gap in faster newer processors
    • Number of instructions goes down due to direct manipulation of registers (no need for load and store instructions)

• **Probably** is the optimization with the most impact!

• **Common Trade-Off:**
  – Registers: Fast Storage with Small Capacity (say 32, 64, 128)
  – Main Memory: Slow Storage with High Capacity (say Giga Bytes)
Importance of Register Allocation

• What Can Be Put in Registers?
  – Scalar Variables
  – Big Constants
  – Some Array Elements and Record Fields
  – Register set depending on the data-type
    • Floating-point in fp registers
    • Fixed-point in integer registers

• Allocation of Variables (including temporaries) up-to-now stored in Memory to Hardware Registers
  – Pseudo or Virtual Registers
    • unlimited number of registers
    • space is typically allocated on the stack with the stack frame
  – Hard Registers
    • Set of Registers Available in the Processor
    • Usually need to Obey some Usage Convention
# Register Usage Convention in MIPS

<table>
<thead>
<tr>
<th>Name</th>
<th>Number</th>
<th>Use</th>
<th>Preserved Across a Function Call?</th>
</tr>
</thead>
<tbody>
<tr>
<td>$zero</td>
<td>0</td>
<td>The Constant Value 0</td>
<td>Yes</td>
</tr>
<tr>
<td>$at</td>
<td>1</td>
<td>Assembler Temporary</td>
<td>No</td>
</tr>
<tr>
<td>$v0, $v1</td>
<td>2,3</td>
<td>Values for Function Results and Expression Evaluation</td>
<td>No</td>
</tr>
<tr>
<td>$a0-$a3</td>
<td>4-7</td>
<td>Function Arguments</td>
<td>No</td>
</tr>
<tr>
<td>$t0-$t7</td>
<td>8-15</td>
<td>Temporaries</td>
<td>No</td>
</tr>
<tr>
<td>$s0-$s7</td>
<td>16-23</td>
<td>Saved Temporaries</td>
<td>Yes</td>
</tr>
<tr>
<td>$t8,$t9</td>
<td>24,25</td>
<td>Temporaries</td>
<td>No</td>
</tr>
<tr>
<td>$k0,$k1</td>
<td>26,27</td>
<td>Reserved for OS Kernel</td>
<td>No</td>
</tr>
<tr>
<td>$gp</td>
<td>28</td>
<td>Global Pointer</td>
<td>Yes</td>
</tr>
<tr>
<td>$sp</td>
<td>29</td>
<td>Stack Pointer</td>
<td>Yes</td>
</tr>
<tr>
<td>$fp</td>
<td>30</td>
<td>Frame Pointer</td>
<td>Yes</td>
</tr>
<tr>
<td>$ra</td>
<td>31</td>
<td>Return Address</td>
<td>Yes</td>
</tr>
</tbody>
</table>
Register Allocation Approaches

• Local Allocators: use instruction-level knowledge
  – Top-Down: Use Frequency of Variables Use for Allocation
  – Bottom-Up: Evaluate Instructions Needs and Reuse Registers

• Global Allocators: use a Graph-Coloring Paradigm
  – Build a “conflict graph” or “interference graph”
  – Find a $k$-coloring for the graph, or change the code to a nearby problem that it can $k$-color

• Common Algorithmic Trade-Off
  – Local Allocators are Fast
  – Some Problems with the Generated Code as they lack more Global Knowledge
Local Register Allocation

• In General Hard Problem (*still*)
  – Code Generation for more than a single Register is NP-Complete

• Use Simple Strategies:
  – Top-Down: Just put in Register Names that Occur more Often
  – Bottom-Up: Evaluate each Instruction and Keep Track of When
    Values are Needed Later On.

• Extension to Multiple Basic Blocks
  – Using Profile Data to Determine Frequently Executed Paths
  – Use Nesting Depth of Code
Top-Down Local Register Allocator

• Estimate the Benefits of Putting each Variable in a Register in a Particular Basic Block
  – \( \text{cost}(V, B) = \text{Number of uses anddefs of the var V in basic block B} \)

• Estimate the Overall Benefit
  – \( \text{TotCost}(V) = \text{cost}(V, B) \times \text{freq}(B) \) for all basic block \( B \)
  – If \( \text{freq}(B) \) is not known, use \( 10^{\text{depth}} \) where \( \text{depth} \) represents the nesting depth of \( B \) in the CFG of the code.

• Assign the (R-feasible) Highest-payoff Variables to Registers
  – Reserve feasible registers for basic calculations and evaluation.
  – Rewrite the code inserting load/store operation where appropriate.
Example

write z
read z

write x
write y
read x
read x
read y

write z
read z

bb1

bb2

bb3
Example

- **Cost for basic blocks**
  - \( \text{cost}(x, \text{bb1}) = 0 \)
  - \( \text{cost}(x, \text{bb2}) = 3 \)
  - \( \text{cost}(x, \text{bb3}) = 0 \)
  - \( \text{cost}(y, \text{bb1}) = 0 \)
  - \( \text{cost}(y, \text{bb2}) = 2 \)
  - \( \text{cost}(y, \text{bb3}) = 0 \)
  - \( \text{cost}(z, \text{bb1}) = 2 \)
  - \( \text{cost}(z, \text{bb2}) = 0 \)
  - \( \text{cost}(z, \text{bb3}) = 2 \)
Example

• Cost for basic blocks
  - $\text{cost}(x, \text{bb1}) = 0$
  - $\text{cost}(x, \text{bb2}) = 3$
  - $\text{cost}(x, \text{bb3}) = 0$
  - $\text{cost}(y, \text{bb1}) = 0$
  - $\text{cost}(y, \text{bb2}) = 2$
  - $\text{cost}(y, \text{bb3}) = 0$
  - $\text{cost}(z, \text{bb1}) = 2$
  - $\text{cost}(z, \text{bb2}) = 0$
  - $\text{cost}(z, \text{bb3}) = 2$

• Frequency
  - $\text{freq}(\text{bb1}) = 1$
  - $\text{freq}(\text{bb2}) = 100$
  - $\text{freq}(\text{bb3}) = 1$
Example

- **Cost for basic blocks**
  - $\text{cost}(x, \ bb1) = 0$
  - $\text{cost}(x, \ bb2) = 3$
  - $\text{cost}(x, \ bb3) = 0$
  - $\text{cost}(y, \ bb1) = 0$
  - $\text{cost}(y, \ bb2) = 2$
  - $\text{cost}(y, \ bb3) = 0$
  - $\text{cost}(z, \ bb1) = 2$
  - $\text{cost}(z, \ bb2) = 0$
  - $\text{cost}(z, \ bb3) = 2$

- **Total cost**
  - $\text{TotCost}(x) = 0*1 + 3*100 + 0*1 = 30$
  - $\text{TotCost}(y) = 0*1 + 2*100 + 0*1 = 20$
  - $\text{TotCost}(z) = 2*1 + 0*100 + 2*1 = 4$

- **Frequency**
  - $\text{freq}(bb1) = 1$
  - $\text{freq}(bb2) = 10$
  - $\text{freq}(bb3) = 1$
Example

• **Total cost**
  - \( \text{TotCost(x)} = 30 \)
  - \( \text{TotCost(y)} = 20 \)
  - \( \text{TotCost(z)} = 4 \)

• **Assume 2 Registers are Available**
  - Assign \( x \) and \( y \) to Registers

```plaintext
write z
read z
```

```plaintext
write x
write y
read x
read x
read y
```

```plaintext
write z
read z
```
Problem 1

• Allocation is same as above
  – x, and y get registers, but not z
• The variables need to occupy the registers even when it does not need it
• All x, y and z can have registers
Problem 1

- Allocation is same as above
  - x, and y get registers, but not z
- The variables need to occupy the registers even when it does not need it
- All x, y and z can have registers
Problem 1

- Allocation is same as above
  - x, and y get registers, but not z
- The variables need to occupy the registers even when it does not need it
- All x, y and z can have registers

```
write z
read z
```
```
write x
write y
read x
read x
read y
```
```
write z
read z
```
Problem 2

• Even non-interfering variables don’t share registers
Problem 2

- Even non-interfering variables don’t share registers
Problem 2

- Even non-interfering variables don’t share registers

- x and y can use the same registers
Problem 3

• Different phases of the program behave differently

• One register available
  – Register for \( x \) in the first loop
  – Register for \( y \) in the second loop
  – Don’t care too much about the rest

• Need to spill
  – Top-Down “All or Nothing” will not work

write & read \( x \)
write & read \( y \)

read & write \( x \)

read & write \( y \)

read \( x \)
read \( y \)
**Bottom-Up Local Allocator**

- **Basic Ideas:**
  - Focus on the Needs of Each Instructions in a Basic Block
    - Ensure Each Instruction Can Execute
    - Instruction Operands and Results in Registers
  - Transitions Between Instructions
    - Observe Which Values are Used Next; in the Future

- **On-Demand Allocation:**
  - Iterate Through the Instructions of a Basic Block
  - Allocate the Value of the Operand, if Not Already in a Register
  - Allocate Register for Result
  - When Out of Registers:
    - Release Register Whose Value is to be Used Farthest into the Future
    - Dirty Register Value Requires Memory Operation to Update Storage
Bottom-Up Local Allocator

• Details:
  – Instructions in format: \( vr_x \leftarrow vr_x \text{ op } vr_y \) using virtual registers
  – Data Structures: A Class of Registers
    • The Number of Registers in Each Class
    • The Virtual Name for Each Register in the Class
    • For Each Virtual Name a Distance to the Next Use in the Basic Block
    • A Flag Indicating if the Corresponding Physical Register is in Use
    • A Stack of Free Physical Registers with a Stack Pointer (Integer Index)

– Functions:
  • \( \text{class}(vr_x) \) defines the set of Registers the Value in \( vr_x \) can be Stored into
  • \( \text{ensure}(vr_x), \text{free}(vr_x) \) and \( \text{allocate}(vr_x) \) functions.
  • \( \text{dist}(vr_x) \) returns the distance to the next reference to \( vr_x \)

```plaintext
initialize(class, size)
  class.Size ← size;
  for i ← size-1 to 0 do
    class.Name[i] ← -1;
    class.Next[i] ← ∞;
    class.Free[i] ← true;
    push(i, class);
  class.StackTop = size-1;
```


Bottom-Up Local Allocator

**Bottom-Up Local Allocation Algorithm**

*Input*: Basic Block B  
*Output*: Rewritten Instruction in B

```plaintext
foreach instr i: vr₁₃ ← vr₁₁ op vr₁₂ ∈ B do

  \( rₓ \leftarrow \) ensure(vr₁₁, class(vr₁₁));
  \( rᵧ \leftarrow \) ensure(vr₁₂, class(vr₁₂));
  if(vr₁₁ is not needed after i) then
    free(vr₁₁, class(vr₁₁));
  if(vr₁₂ is not needed after i) then
    free(vr₁₂, class(vr₁₂));
  \( rz \leftarrow \) allocate(vr₁₃, class(vr₁₃));

  rewrite i as \( rz \leftarrow rₓ \) op \( rᵧ \)
  if (vr₁₁ is needed after i) then
    class.Next[rₓ] = dist(vr₁₁);
  else
    class.Next[rₓ] = \( \infty \);
  if (vr₁₂ is needed after i) then
    class.Next[rᵧ] = dist(vr₁₂);
  else
    class.Next[rᵧ] = \( \infty \);
  class.Next[rz] = dist(vr₁₃);

end
```

**void free(i,class)**

```plaintext
if(class.Free[i]≠ true) then
  push(i, class);
  class.Name[i] ← -1;
  class.Next[i] ← \( \infty \);
  class.Free[i] ← true;
end
```

**reg ensure(vr, class)**

```plaintext
r ← find(vr, class);
if(r exists) then
  result ← r;
else
  result ← allocate(vr, class);
  emit code to move vr into r;
end
return result;
```

**reg allocate(vr, class)**

```plaintext
if(class.StackTop ≥ 0) then
  r ← pop(class);
else
  r ← findMaxNext(class);
  if(r is dirty) then
    save r in memory;
  class.Name[r] ← vr;
  class.Next[r] ← -1;
  class.Free[r] ← false;
end
```
Bottom-Up Local Allocator

**Bottom-Up Local Allocation Algorithm**

*Input*: Basic Block \(B\)

*Output*: Rewritten Instruction in \(B\)

```
foreach instr i: \(vr_{i3} \leftarrow vr_{i1} \text{ op } vr_{i2} \in B\) do

\(r_x \leftarrow\) ensure(\(vr_{i1}\), class(\(vr_{i1}\)));
\(r_y \leftarrow\) ensure(\(vr_{i2}\), class(\(vr_{i2}\)));
if(\(vr_{i1}\) is not needed after \(i\)) then
  free(\(vr_{i1}\), class(\(vr_{i1}\)));
if(\(vr_{i2}\) is not needed after \(i\)) then
  free(\(vr_{i2}\), class(\(vr_{i2}\)));
\(r_z \leftarrow\) allocate(\(vr_{i3}\), class(\(vr_{i3}\)));

**rewrite** i as \(r_z \leftarrow r_x \text{ op } r_y\)
if (\(vr_{i1}\) is needed after \(i\)) then
  class.Next[\(r_x\)] = dist(\(vr_{i1}\));
else
  class.Next[\(r_x\)] = \(\infty\);
if (\(vr_{i2}\) is needed after \(i\)) then
  class.Next[\(r_y\)] = dist(\(vr_{i2}\));
else
  class.Next[\(r_y\)] = \(\infty\);
class.Next[\(r_z\)] = dist(\(vr_{i3}\));
```

**void free(i,class)**

```
if(class.Free[i]≠ true) then
  push(i,class);
  class.Name[i] ← -1;
  class.Next[i] ← \(\infty\);
  class.Free[i] ← true;
end
```

**reg ensure(vr,class)**

```
r ← find(vr,class);
if(r exists) then
  result ← r;
else
  result ← allocate(vr,class);
  emit code to move vr into r;
end
return result;
```

**reg allocate(vr,class)**

```
if(class.StackTop ≥ 0) then
  r ← pop(class);
else
  r ← findMaxNext(class);
  if(r is dirty) then
    save r in memory;
  class.Name[r] ← vr;
  class.Next[r] ← -1;
  class.Free[r] ← false;
return r;
```

Next field is temporarily set to -1 so that a possible second allocate will not bump this register on the same instruction.
Bottom-Up Allocator Example

vr₃ ← vr₁ op vr₂
vr₅ ← vr₄ op vr₁
vr₆ ← vr₅ op vr₆
vr₇ ← vr₃ op vr₂
## Bottom-Up Allocator Example

Size = 3

vr₃ ← vr₁ op vr₂
vr₅ ← vr₄ op vr₁
vr₆ ← vr₅ op vr₆
vr₇ ← vr₃ op vr₂

<table>
<thead>
<tr>
<th>Name</th>
<th>0</th>
<th>1</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td>vr₂</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Next</th>
<th>0</th>
<th>1</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Free</th>
<th>0</th>
<th>1</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td>T</td>
<td>T</td>
<td>T</td>
<td></td>
</tr>
</tbody>
</table>

Stack: 2

Top = 2
Bottom-Up Allocator Example

Size = 3

```
mem → r_0 // vr1’s value
vr_3 ← r_0 op vr_2
vr_5 ← vr_4 op vr_1
vr_6 ← vr_5 op vr_6
vr_7 ← vr_3 op vr_2
```

```
<table>
<thead>
<tr>
<th>Name</th>
<th>vr_1</th>
<th>-1</th>
<th>-1</th>
</tr>
</thead>
<tbody>
<tr>
<td>Next</td>
<td>1</td>
<td>∞</td>
<td>∞</td>
</tr>
<tr>
<td>Free</td>
<td>F</td>
<td>T</td>
<td>T</td>
</tr>
<tr>
<td>Stack</td>
<td>2</td>
<td>1</td>
<td></td>
</tr>
</tbody>
</table>

Top = 1
Bottom-Up Allocator Example

Size = 3

<table>
<thead>
<tr>
<th>Name</th>
<th>vr1</th>
<th>vr2</th>
<th>T</th>
</tr>
</thead>
<tbody>
<tr>
<td>Next</td>
<td>1</td>
<td>3</td>
<td>∞</td>
</tr>
<tr>
<td>Free</td>
<td>F</td>
<td>F</td>
<td>T</td>
</tr>
<tr>
<td>Stack</td>
<td>2</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Top = 0

mem → r₀ // vr₁’s value
mem → r₁ // vr₂’s value
vr₃ ← r₀ op r₁
vr₅ ← vr₄ op vr₁
vr₆ ← vr₅ op vr₆
vr₇ ← vr₃ op vr₂
Bottom-Up Allocator Example

Size = 3

mem → r₀  // vr₁’s value
mem → r₁  // vr₂’s value
r₂ ← r₀ op r₁
vr₅ ← vr₄ op vr₁
vr₆ ← vr₅ op vr₆
vr₇ ← vr₃ op vr₂

Name  vr₁  vr₂  vr₃
Next   1    3    3
Free   F    F    F
Stack  Top  =  -1
Bottom-Up Allocator Example

### Example:

Size = 3

<table>
<thead>
<tr>
<th>Name</th>
<th>vr₁</th>
<th>vr₂</th>
<th>vr₃</th>
</tr>
</thead>
<tbody>
<tr>
<td>Next</td>
<td>1</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>Free</td>
<td>F</td>
<td>F</td>
<td>F</td>
</tr>
<tr>
<td>Stack</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Top = -1

Run Out of Registers for vr₄

Allocate will find register with maximum value of Next to Spill to Memory, e.g., vr₂
### Bottom-Up Allocator Example

Size = 3

<table>
<thead>
<tr>
<th>Name</th>
<th>vr1</th>
<th>vr4</th>
<th>vr3</th>
</tr>
</thead>
<tbody>
<tr>
<td>Next</td>
<td>1</td>
<td>-1</td>
<td>3</td>
</tr>
<tr>
<td>Free</td>
<td>F</td>
<td>F</td>
<td>F</td>
</tr>
<tr>
<td>Stack</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Top</td>
<td>-1</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

```
mem → r₀  // vr1’s value
mem → r₁  // vr2’s value
r₂ ← r₀ op r₁
r₁ → mem  // vr2’s value
mem → r₁  // vr4’s value
vr₅ ← r₁ op vr₁
vr₆ ← vr₅ op vr₆
vr₇ ← vr₃ op vr₂
```
Bottom-Up Allocator Example

Size = 3

<table>
<thead>
<tr>
<th>Name</th>
<th>vr₁</th>
<th>-1</th>
<th>vr₃</th>
</tr>
</thead>
<tbody>
<tr>
<td>Next</td>
<td>1</td>
<td>∞</td>
<td>3</td>
</tr>
<tr>
<td>Free</td>
<td>F</td>
<td>T</td>
<td>F</td>
</tr>
<tr>
<td>Stack</td>
<td></td>
<td>1</td>
<td></td>
</tr>
</tbody>
</table>

mem $\rightarrow r₀$ // vr₁’s value
mem $\rightarrow r₁$ // vr₂’s value
r₂ $\leftarrow r₀ \text{ op } r₁$
vr₅ $\leftarrow r₁ \text{ op } vr₁$
vr₆ $\leftarrow vr₅ \text{ op } vr₆$
vr₇ $\leftarrow vr₃ \text{ op } vr₂$

mem $\rightarrow r₁$ // vr₂’s value
mem $\rightarrow r₁$ // vr₄’s value
Bottom-Up Allocator Example

Size = 3

mem → r₀ // vr₁’s value
mem → r₁ // vr₂’s value
r₂ ← r₀ op r₁
r₁ → mem // vr₂’s value
mem → r₁ // vr₄’s value
vr₅ ← r₁ op r₀
vr₆ ← vr₅ op vr₆
vr₇ ← vr₃ op vr₂

<table>
<thead>
<tr>
<th>Name</th>
<th>-1</th>
<th>-1</th>
<th>vr₃</th>
</tr>
</thead>
<tbody>
<tr>
<td>Next</td>
<td>∞</td>
<td>∞</td>
<td>3</td>
</tr>
<tr>
<td>Free</td>
<td>T</td>
<td>T</td>
<td>F</td>
</tr>
</tbody>
</table>

Stack

<table>
<thead>
<tr>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
</tr>
</tbody>
</table>

Top = 1
Bottom-Up Allocator Example

Size = 3

<table>
<thead>
<tr>
<th>Name</th>
<th>1</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td>vr₅</td>
<td>-1</td>
<td>vr₃</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Next</th>
<th>1</th>
<th>3</th>
<th>∞</th>
</tr>
</thead>
<tbody>
<tr>
<td>vr₇</td>
<td>vr₆</td>
<td>vr₅</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Free</th>
<th>F</th>
<th>T</th>
<th>F</th>
</tr>
</thead>
<tbody>
<tr>
<td>mem</td>
<td>r₁</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Stack</td>
<td>1</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Bottom-Up Allocator Example

Size = 3

<table>
<thead>
<tr>
<th>Name</th>
<th>vr₅</th>
<th>-1</th>
<th>vr₃</th>
</tr>
</thead>
<tbody>
<tr>
<td>Next</td>
<td>-1</td>
<td>∞</td>
<td>3</td>
</tr>
<tr>
<td>Free</td>
<td>F</td>
<td>T</td>
<td>F</td>
</tr>
<tr>
<td>Stack</td>
<td>1</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

mem → r₀ // vr₁’s value
mem → r₁ // vr₂’s value
r₂ ← r₀ op r₁
r₁ → mem // vr₂’s value
mem → r₁ // vr₄’s value
r₀ ← r₁ op r₀
vr₆ ← r₀ op vr₆
vr₇ ← vr₃ op vr₂
Bottom-Up Allocator Example

Size = 3

\[
\begin{align*}
\text{mem} & \rightarrow r_0 \quad // \text{vr1’s value} \\
\text{mem} & \rightarrow r_1 \quad // \text{vr2’s value} \\
r_2 & \leftarrow r_0 \text{ op } r_1 \\
r_1 & \rightarrow \text{mem} \quad // \text{vr2’s value} \\
\text{mem} & \rightarrow r_1 \quad // \text{vr4’s value} \\
r_0 & \leftarrow r_1 \text{ op } r_0 \\
\text{vr}_6 & \leftarrow r_0 \text{ op } r_1 \\
\text{vr}_7 & \leftarrow \text{vr}_3 \text{ op } \text{vr}_2
\end{align*}
\]

<table>
<thead>
<tr>
<th>Name</th>
<th>Next</th>
<th>Free</th>
<th>Stack</th>
<th>Top = -1</th>
</tr>
</thead>
<tbody>
<tr>
<td>vr_5</td>
<td>-1</td>
<td>F</td>
<td>F</td>
<td></td>
</tr>
<tr>
<td>vr_6</td>
<td>-1</td>
<td>F</td>
<td>F</td>
<td></td>
</tr>
<tr>
<td>vr_3</td>
<td>3</td>
<td>F</td>
<td>F</td>
<td></td>
</tr>
</tbody>
</table>
Bottom-Up Allocator Example

Size = 3

<table>
<thead>
<tr>
<th>Name</th>
<th>vr₀</th>
<th>vr₁</th>
<th>vr₂</th>
</tr>
</thead>
<tbody>
<tr>
<td>Next</td>
<td>-1</td>
<td>-1</td>
<td>3</td>
</tr>
<tr>
<td>Free</td>
<td>F</td>
<td>F</td>
<td>F</td>
</tr>
<tr>
<td>Stack</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Top</td>
<td>-1</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Bottom-Up Allocator Example

Size = 3

<table>
<thead>
<tr>
<th>Name</th>
<th>Next</th>
<th>Free</th>
<th>Stack</th>
</tr>
</thead>
<tbody>
<tr>
<td>-1</td>
<td>∞</td>
<td>T</td>
<td>0</td>
</tr>
<tr>
<td>-1</td>
<td>∞</td>
<td>T</td>
<td></td>
</tr>
<tr>
<td>vr3</td>
<td>3</td>
<td>F</td>
<td>1</td>
</tr>
</tbody>
</table>

mem → r₀  // vr1’s value
mem → r₁  // vr2’s value
r₂ ← r₀ op r₁
r₁ → mem  // vr2’s value
mem → r₁  // vr4’s value
r₀ ← r₁ op r₀
r₁ ← r₀ op r₁
vr₇ ← vr₃ op vr₂
Bottom-Up Allocator Example

Size = 3

```
mem \rightarrow r_0 // vr1's value
mem \rightarrow r_1 // vr2's value
r_2 \leftarrow r_0 \text{ op } r_1
r_1 \rightarrow \text{mem} // vr2's value
mem \rightarrow r_1 // vr4's value
r_0 \leftarrow r_1 \text{ op } r_0
r_1 \leftarrow r_0 \text{ op } r_1
r_1 \rightarrow \text{mem} // vr6's value
r_0 \leftarrow r_2 \text{ op } r_1
```

<table>
<thead>
<tr>
<th>Name</th>
<th>vr_7</th>
<th>vr_6</th>
<th>vr_3</th>
</tr>
</thead>
<tbody>
<tr>
<td>Next</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
</tr>
<tr>
<td>Free</td>
<td>F</td>
<td>F</td>
<td>F</td>
</tr>
</tbody>
</table>

Stack | Top = -1

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Bottom-Up Allocator Example

\[
\begin{align*}
vr_3 & \leftarrow vr_1 \text{ op } vr_2 \\
vr_5 & \leftarrow vr_4 \text{ op } vr_1 \\
vr_6 & \leftarrow vr_5 \text{ op } vr_6 \\
vr_7 & \leftarrow vr_3 \text{ op } vr_2 \\
\end{align*}
\]

\[
\begin{align*}
\text{mem} & \rightarrow r_0 \quad // vr_1\text{’s value} \\
\text{mem} & \rightarrow r_1 \quad // vr_2\text{’s value} \\
r_2 & \leftarrow r_0 \text{ op } r_1 \\
r_1 & \rightarrow \text{mem} \quad // vr_2\text{’s value} \\
\text{mem} & \rightarrow r_1 \quad // vr_4\text{’s value} \\
r_0 & \leftarrow r_1 \text{ op } r_0 \\
r_1 & \leftarrow r_0 \text{ op } r_1 \\
r_1 & \rightarrow \text{mem} \quad // vr_6\text{’s value} \\
r_0 & \leftarrow r_2 \text{ op } r_1 \\
\end{align*}
\]
Dirty and Clean Registers

- **Choosing which Register to Reuse:**
  - Dirty - Need to Update Memory
  - Clean - Just Reuse the Physical Register

- **Idea: Give Preference to Clean Registers**
  - No Need to Save Contents to Memory

- **Not Always the Best Approach:**
  - When Dirty Value is Reused Far Way
  - Better to Restore it to Memory
  - Rather than Holding on to the Register

```
mem → r₀ // vr1’s value
mem → r₁ // vr2’s value
r₂ ← r₀ op r₁
r₁ → mem // vr2’s value
mem → r₁ // vr4’s value
r₀ ← r₁ op r₀
r₁ ← r₀ op r₁
r₁ → mem // vr6’s value
r₀ ← r₂ op r₁
```
What a Smart Allocator Needs to Do

• Determine ranges for each variable can benefit from using a register (webs)
• Determine which of these ranges overlap (interference)
• Find the benefit of keeping each web in a register (spill cost)
• Decide which webs get a register (allocation)
• Split webs if needed (spilling and splitting)
• Assign hard registers to webs (assignment)
• Generate code including spills (code generation)
Summary

• Register Allocation and Assignment
  – Very Important Transformations and Optimization
  – In General Hard Problem (NP-Complete)

• Many Approaches
  – Local Methods: Top-Down and Bottom-Up
  – Quick but not Necessarily Very Good