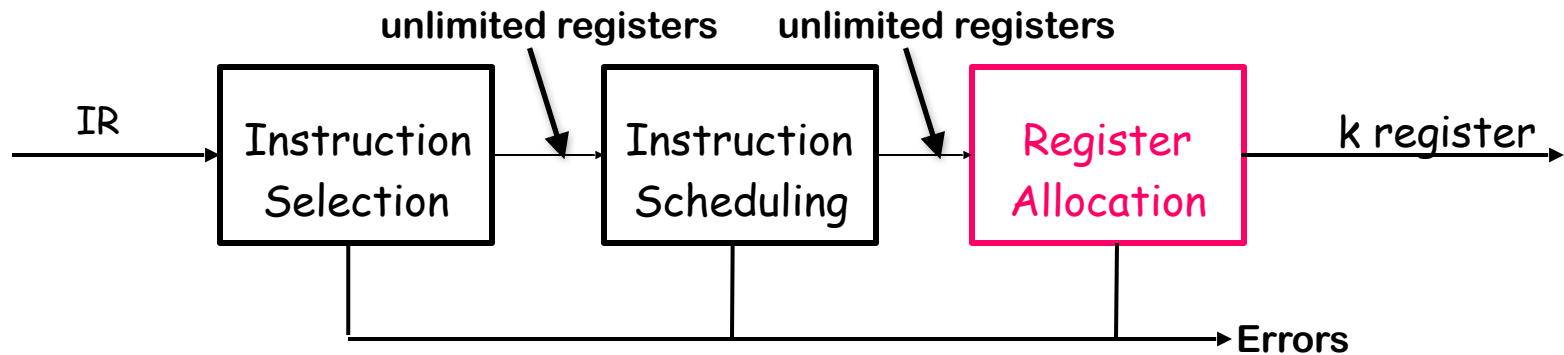


Local Register Allocation

Copyright 2010, Keith D. Cooper & Linda Torczon, all rights reserved.

Register Allocation

Part of the compiler's back end



Spill code: Loads and Stores
inserted by the register allocator

Critical properties

- Produce correct code that uses no more than k registers
- Minimize added work from loads and stores that spill values
- Minimize space used to hold spilled values
- Operate efficiently
 $O(n)$, $O(n \log_2 n)$, maybe $O(n^2)$, but not $O(2^n)$

Notation: The literature on register allocation consistently uses k as the number of registers available on the target system.

r0 holds base address for local variables

Register Allocation

@x is constant offset of x from r0

Consider a fragment of assembly code (or ILOC)

```
loadI  2      ⇒ r1    // r1 ← 2
loadAI r0, @b ⇒ r2    // r2 ← b
mult   r1, r2 ⇒ r3    // r3 ← 2 · b
loadAI r0, @a ⇒ r4    // r4 ← a
sub    r4, r3 ⇒ r5    // r5 ← a - (2 · b)
```

From the allocation perspective, these registers are virtual or pseudo-registers

The Problem

- At each instruction, decide which **values** to keep in registers
 - Note: each pseudo-register in the example is a value
- Simple if $|\text{values}| \leq |\text{registers}|$
- Harder if $|\text{values}| > |\text{registers}|$
- The compiler must automate this process

Register Allocation

The Task

- At each point in the code, pick the values to keep in registers
- Insert code to move values between registers & memory
 - No transformations (leave that to optimization & scheduling)
- Minimize inserted code — both dynamic & static measures
- Make good use of any extra registers

Allocation versus assignment

- **Allocation** is deciding which values to keep in registers
- **Assignment** is choosing specific registers for values
- This distinction is often lost in the literature

The compiler must perform both allocation & assignment

Background issues

- The register allocator takes as input a code that is almost completely compiled
- It has been scanned, parsed, checked, analysed, optimised, rewritten as target machine code, and, perhaps, scheduled
- Many previously made decisions influence the task of the allocator:
 - Memory-to-memory versus register-to-register memory model

Additional complexity:

- Allocation vs Assignment
 - Register Classes

Register-to-register vs. memory-to-memory

- With a register-to-register earlier phases in the compiler directly encode the knowledge about ambiguous memory references: with this model unambiguous values are kept into virtual registers
- In a register-to-register the code produced by the previous step is not legal
- In a memory-to-memory model, the code is legal before allocation; allocation improve performance
- In a memory-to-memory model the allocator does not have any knowledge and this can limit its ability

Allocation

Allocation is an hard problem that in its general formulation is NP-complete.

The allocation of a **single basic** block with **one size data** value can be done in **polynomial time under strong hypothesis**:

- each value have to be stored to memory at the end of its lifetime (no constant,...)
- the spilling of value has **uniform cost**

any additional complexity makes the problem NP-complete

Allocation vs. Assignment

- Once we have reduced the demand for registers, the assignment can be done in polynomial time for a machine with one kind of registers

Register Classes

- General purpose registers
 - Integer values and memory addresses
 - Floating-point registers (single and double precision)
 - On some architectures also condition code, predicate registers or branch target registers
-
- If the compiler uses different kind of registers for different kinds of data, it can allocate each class independently: the problem can be simplified
 - If the different kinds of data overlap, the compiler must allocate them together: the allocation can become more complex (single and double precision registers for floating-point)

Basic Blocks in Assembly Code (or ILOC)

Definition

- A **basic block** is a maximal length segment of straight-line (i.e., branch free) code

Importance

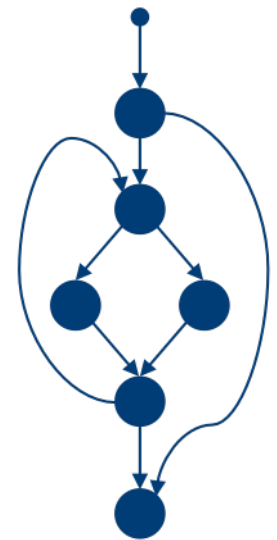
- Strongest facts are provable for branch-free code
 - If any statement executes, they all execute
 - Execution is totally ordered
- Ignore, for the moment, exceptions

Role of Basic Blocks in Optimization

- Many techniques for improving basic blocks
- Simplest problems
- Strongest methods

Local Register Allocation

- What is “local” ? (different from “regional” or “global”)
 - A local transformation operates on basic blocks
 - Many optimizations are done on a local scale or scope
- Does local allocation solve the problem?
 - It produces good register use inside a block
 - Inefficiencies can arise at boundaries between blocks
- How many passes can the allocator make?
 - This is an off-line problem
 - As many passes as it takes, within reason
 - You can do a fine job in a couple of passes



Blocks in a Control-flow Graph (CFG)

Register Allocation

Optimal register allocation is hard

Local Allocation

- Simplified cases $\Rightarrow O(n)$
- Real cases \Rightarrow NP-Complete

Global Allocation

- NP-Complete for 1 register
- NP-Complete for k registers
(most sub-problems are NPC, too)

Local Assignment

- Single size, no spilling $\Rightarrow O(n)$
- Two sizes \Rightarrow NP-Complete

Global Assignment

- NP-Complete

Real compilers face real problems

ILOC

- Pseudo-code for a simple, abstracted RISC machine
 - generated by the instruction selection process
- Simple, compact data structures

a - 2 x b

loadI 2 $\Rightarrow r_1$

loadAI $r_0, @b$ $\Rightarrow r_2$

mult r_1, r_2 $\Rightarrow r_3$

loadAI $r_0, @a$ $\Rightarrow r_4$

sub r_4, r_3 $\Rightarrow r_5$

Nearly assembly code

- simple three-address code
- RISC-like addressing modes
 - I, AI, AO
- unlimited virtual registers
(**register-to-register** vs
memory-to-memory)

ILOC

- Pseudo-code for a simple, abstracted RISC machine
 - generated by the instruction selection process
- Simple, compact data structures

$a - 2 \times b$

loadI	2		r_1
loadAI	r_0	@b	r_2
add	r_1	r_2	r_3
loadAI	r_0	@a	r_4
sub	r_4	r_3	r_5

Quadruples:

- table of $k \times 4$ small integers
- simple record structure
- easy to reorder
- all names are explicit

Observations

The Register Allocator does not need to “understand” the code

- It needs to distinguish definitions from uses
 - Definitions might need to store a spilled value
 - Uses might need to load a spilled value
 - ILOC makes definitions and uses pretty clear
 - The assignment arrow, \Rightarrow , separates uses from definitions
 - Except on the store operation, which uses all its register operands
- ```
store r8 \Rightarrow r1 // MEM(r1) \leftarrow r8
```
- That is the point of the arrow!
- Your allocator needs to know, by opcode, how many definitions and how many uses it should see
  - Beyond that, the meaning of the ILOC is somewhat irrelevant to the allocator

# Observations

---

A value is *live* between its *definition* and its *uses*

- Find definitions ( $x \leftarrow \dots$ ) and uses ( $y \leftarrow \dots x \dots$ )
- From definition to last use is its *live range*
  - How does a *second definition* affect this?
- Can represent live range as an interval  $[i,j]$  (in block)

Let  $\text{MAXLIVE}$  be the maximum, over each instruction  $i$  in the block, of the number of values (pseudo-registers) live at  $i$ .

- If  $\text{MAXLIVE} \leq k$ , allocation should be easy:  
no need to reserve  $F$  registers for spilling
- If  $\text{MAXLIVE} > k$ , some values must be spilled to memory:  
need to reserve  $F$  registers for spilling

Finding live ranges is harder in the global case



# Concrete Example of MAXLIVE

## Sample code sequence

```
loadI 1028 => r1 // r1 ← 1028
load r1 => r2 // r2 ← MEM(r1)
mult r1, r2 => r3 // r3 ← 1028 · y
load x => r4 // r4 ← x
sub r4, r2 => r5 // r5 ← x - y
load z => r6 // r6 ← z
mult r5, r6 => r7 // r7 ← z · (x - y)
sub r7, r3 => r8 // r8 ← z · (x - y) - (1028 · y)
store (r8) => r1 // MEM(r1) ← z · (x - y) - (1028 · y)
```

Store uses this register & defines a memory location.

The code uses 1028 as both an address and as a constant in the computation.

The intent is to create a long live range for pedagogical purposes. Remember, the allocator does not need to understand the computation. It just needs to preserve the computation.

# Concrete Example of MAXLIVE

---

Live ranges in the example

```
loadI 1028 => r1 // r1
load r1 => r2 // r1 r2
mult r1, r2 => r3 // r1 r2 r3
load x => r4 // r1 r2 r3 r4
sub r4, r2 => r5 // r1 r3 r5
load z => r6 // r1 r3 r5 r6
mult r5, r6 => r7 // r1 r3 r7
sub r7, r3 => r8 // r1 r8
store r8 => r1 //
```

Remember, r1 is a use,  
not a definition

A pseudo-register is live  
after an operation if it has  
been defined & has a use in  
the future

# Concrete Example of MAXLIVE

Live ranges in the example

|       |        |      |                |    |
|-------|--------|------|----------------|----|
| loadI | 1028   | ⇒ r1 | // r1          |    |
| load  | r1     | ⇒ r2 | // r1 r2       |    |
| mult  | r1, r2 | ⇒ r3 | // r1 r2 r3    |    |
| load  | x      | ⇒ r4 | // r1 r2 r3 r4 |    |
| sub   | r4, r2 | ⇒ r5 | // r1 r3 r5    |    |
| load  | z      | ⇒ r6 | // r1 r3 r5 r6 |    |
| mult  | r5, r6 | ⇒ r7 | // r1 r3 r7    |    |
| sub   | r7, r3 | ⇒ r8 | // r1          | r8 |
| store | r8     | ⇒ r1 | //             |    |

MAXLIVE is 4

Remember, r1 is a use,  
not a definition

Compute these "live" sets in a backward pass over the code.  
Start with live as the empty set.  
At each op, remove target & add operands

# Local Allocation: Top-down Versus Bottom-up

---

## Top-down allocator

- Work from external notion of what is important
- Assign registers in priority order
- Save some registers for the values relegated to memory

## Bottom-up allocator

- Work from detailed knowledge about problem instance
- Incorporate knowledge of partial solution at each step
- Handle all values uniformly

# Top-down Allocator

---

## The idea

- The **most heavily used values should reside in a register**
- Reserve registers for use in spills, say  $r$  registers

## Algorithm

- Count the number of occurrences of each virtual register in the block (from 2 to  $\text{maxlength}(\text{block})$ )
  - Sort the registers according to the previous info
  - Allocate first  $k - r$  values to registers
  - Rewrite code to reflect these choices
- Move values with no register into memory  
(add LOADs & STOREs)

Programmers applied this idea by hand in the 70's & early 80's

# Top-down Allocator

---

How many registers must the allocator reserve?

- Need registers to compute spill addresses & load values
- Number depends on target architecture
  - Typically, must be able to load 2 values
- Reserve these registers for spilling

What if  $k - r < |\text{values}| < k$ ?

- Remember that the underlying problem is NP-Complete
- The allocator can either
  - Check for this situation
  - Adopt a more complex strategy
  - Accept the fact that the technique is an approximation

# Back to the Example

---

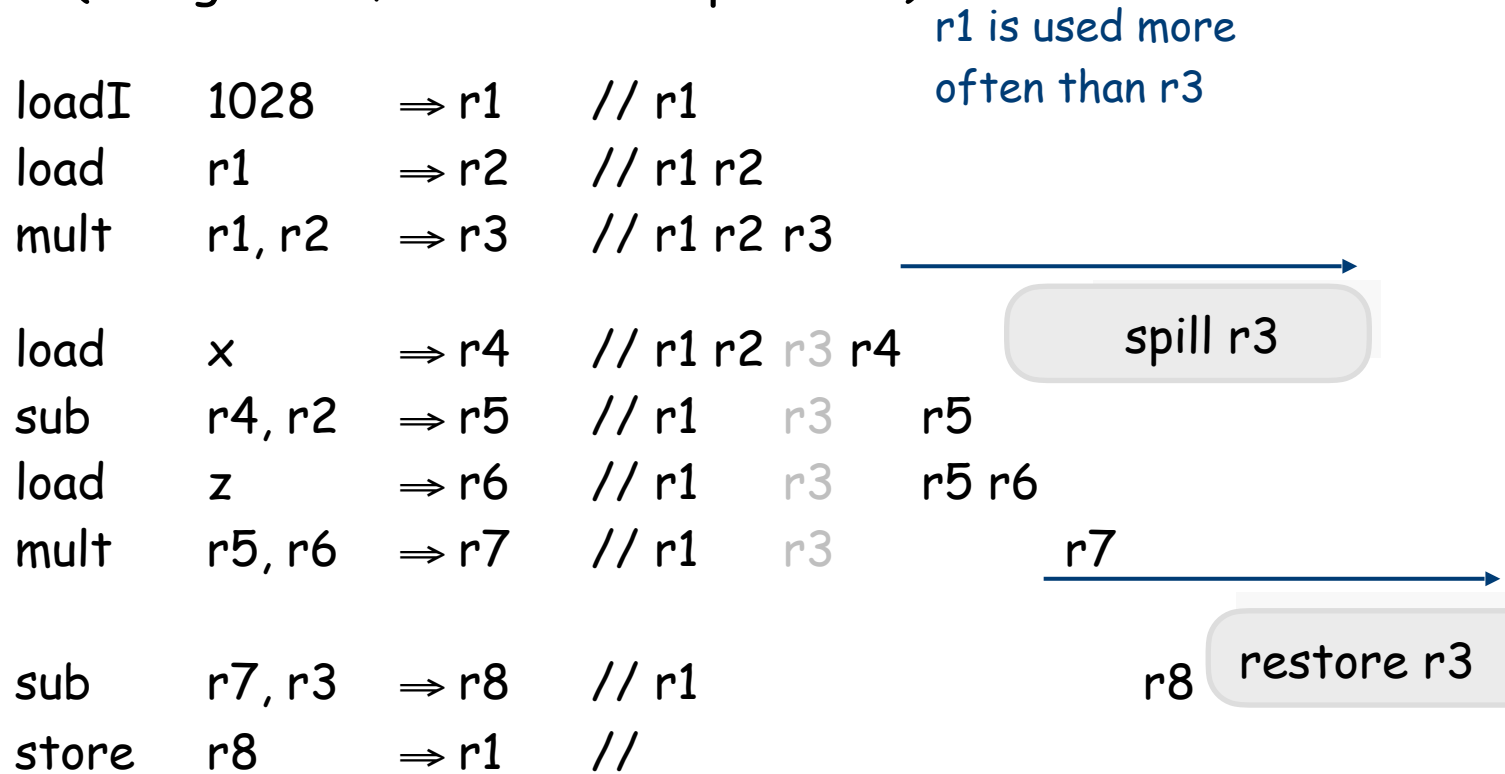
Top down (3 registers)

|       |        |      |                |  |  |  |  |                                  |
|-------|--------|------|----------------|--|--|--|--|----------------------------------|
| loadI | 1028   | ⇒ r1 | // r1          |  |  |  |  | r1 is used more<br>often than r3 |
| load  | r1     | ⇒ r2 | // r1 r2       |  |  |  |  |                                  |
| mult  | r1, r2 | ⇒ r3 | // r1 r2 r3    |  |  |  |  |                                  |
| load  | x      | ⇒ r4 | // r1 r2 r3 r4 |  |  |  |  |                                  |
| sub   | r4, r2 | ⇒ r5 | // r1 r3 r5    |  |  |  |  |                                  |
| load  | z      | ⇒ r6 | // r1 r3 r5 r6 |  |  |  |  |                                  |
| mult  | r5, r6 | ⇒ r7 | // r1 r3 r7    |  |  |  |  |                                  |
| sub   | r7, r3 | ⇒ r8 | // r1 r8       |  |  |  |  |                                  |
| store | r8     | ⇒ r1 | //             |  |  |  |  |                                  |

Note that this assumes that no extra register is needed for spilling

# Back to the Example

Top down (3 registers, need 2 for operands)



Note that this assumes that no extra register is needed for spilling



# An Example

---

Top down (3 registers, need 2 for operands)

|       |        |      |             |       |    |
|-------|--------|------|-------------|-------|----|
| loadI | 1028   | ⇒ r1 | // r1       |       |    |
| load  | r1     | ⇒ r2 | // r1 r2    |       |    |
| mult  | r1, r2 | ⇒ r3 | // r1 r2 r3 |       |    |
| store | r3     | ⇒ 16 | // r1 r2    |       |    |
| load  | x      | ⇒ r4 | // r1 r2    | r4    |    |
| sub   | r4, r2 | ⇒ r5 | // r1       | r5    |    |
| load  | z      | ⇒ r6 | // r1       | r5 r6 |    |
| mult  | r5, r6 | ⇒ r7 | // r1       |       | r7 |
| load  | 16     | ⇒ r3 | // r1       | r3    | r7 |
| sub   | r7, r3 | ⇒ r8 | // r1       |       | r8 |
| store | r8     | ⇒ r1 | //          |       |    |

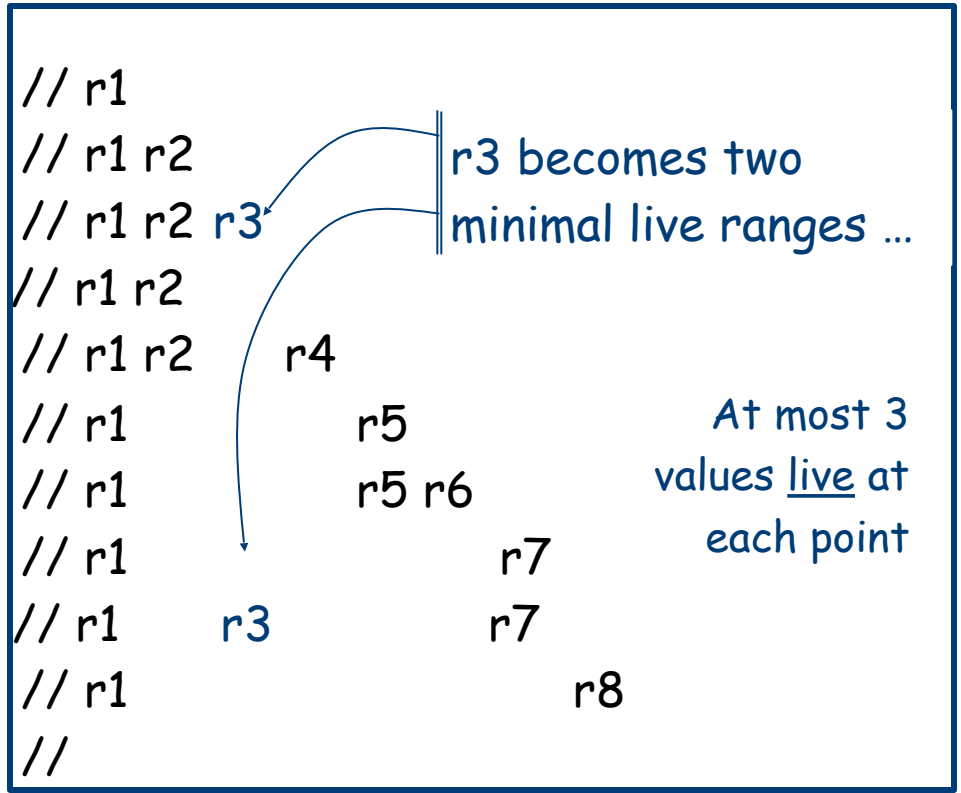
r3 becomes two minimal live ranges ...

“spill” and “restore” become stores and loads

# An Example

Top down (3 registers)

```
loadI 1028 => r1
load r1 => r2
mult r1, r2 => r3
store r3 => 16
load x => r4
sub r4, r2 => r5
load z => r6
mult r5, r6 => r7
load 16 => r3
sub r7, r3 => r8
store r8 => r1
```



The two short versions of r3 each overlap with fewer values, which simplifies the allocation problem. Such "spilling" will (eventually) create a code where the allocator can succeed.

# An Example

---

Top down (3 registers)

```
loadI 1028 ⇒ r1 // r1
load r1 ⇒ r2 // r1 r2
mult r1, r2 ⇒ r3 // r1 r2 r3
store r3 ⇒ 16 // r1 r2
loadI x ⇒ r4 // r1 r2 r4
sub r4, r2 ⇒ r5 // r1 r5
loadI z ⇒ r6 // r1 r5 r6
mult r5, r6 ⇒ r7 // r1 r7
load 16 ⇒ r3 // r1 r3 r7 possible delay
sub r7, r3 ⇒ r8 // r1 r8
store r8 ⇒ r1 //
```

This code is slower than the original, but it works correctly on a target machine with only three (available) registers.

Correctness is a virtue.

## Weakness of the top down approach to allocation

- A physical register is dedicated to a virtual register for an entire block

# Bottom-up Allocator

---

The idea:

- Focus on replacement rather than allocation
- Keep **values used "soon" in registers**

Algorithm (not optimal!):

- Start with empty register set
- Load on demand
- When no register is available, free one

Replacement:

- Spill the value whose **next use is farthest in the future**
- Prefer **clean values** (not to be stored that are constant or values already in memory) to **dirty values** (that need to be stored).


# An Example

---

Bottom up (3 registers)

|       |        |      |                |  |  |  |  |
|-------|--------|------|----------------|--|--|--|--|
| loadI | 1028   | ⇒ r1 | // r1          |  |  |  |  |
| load  | r1     | ⇒ r2 | // r1 r2       |  |  |  |  |
| mult  | r1, r2 | ⇒ r3 | // r1 r2 r3    |  |  |  |  |
| loadI | x      | ⇒ r4 | // r1 r2 r3 r4 |  |  |  |  |
| sub   | r4, r2 | ⇒ r5 | // r1 r3 r5    |  |  |  |  |
| loadI | z      | ⇒ r6 | // r1 r3 r5 r6 |  |  |  |  |
| mult  | r5, r6 | ⇒ r7 | // r1 r3 r7    |  |  |  |  |
| sub   | r7, r3 | ⇒ r8 | // r1 r8       |  |  |  |  |
| store | r8     | ⇒ r1 | //             |  |  |  |  |

All registers are used  
at this point



Note that this assumes that no extra register is needed for spilling

# An Example

---

Bottom up (3 registers; need 2 for operands)

|       |        |      |                |  |  |            |  |
|-------|--------|------|----------------|--|--|------------|--|
| loadI | 1028   | ⇒ r1 | // r1          |  |  |            |  |
| load  | r1     | ⇒ r2 | // r1 r2       |  |  |            |  |
| mult  | r1, r2 | ⇒ r3 | // r1 r2 r3    |  |  |            |  |
| loadI | x      | ⇒ r4 | // r1 r2 r3 r4 |  |  |            |  |
| sub   | r4, r2 | ⇒ r5 | // r1 r3 r5    |  |  | store r1   |  |
| loadI | z      | ⇒ r6 | // r1 r3 r5 r6 |  |  |            |  |
| mult  | r5, r6 | ⇒ r7 | // r1 r3 r7    |  |  |            |  |
| sub   | r7, r3 | ⇒ r8 | // r1 r8       |  |  |            |  |
| store | r8     | ⇒ r1 | //             |  |  | restore r1 |  |

Note that this assumes that no extra register is needed for spilling

# An Example

Bottom up (3 registers; need 2 for operands)

|       |        |      |             |                       |
|-------|--------|------|-------------|-----------------------|
| loadI | 1028   | ⇒ r1 | // r1       |                       |
| load  | r1     | ⇒ r2 | // r1 r2    |                       |
| mult  | r1, r2 | ⇒ r3 | // r1 r2 r3 |                       |
| store | r1     | ⇒ 20 | // r2 r3    |                       |
| loadI | x      | ⇒ r4 | // r2 r3 r4 |                       |
| sub   | r4, r2 | ⇒ r5 | // r3 r5    | At most 3             |
| loadI | z      | ⇒ r6 | // r3 r5 r6 | values <u>live</u> at |
| mult  | r5, r6 | ⇒ r7 | // r3 r7    | each point            |
| sub   | r7, r3 | ⇒ r8 | // r8       |                       |
| load  | 20     | ⇒ r1 | // r1 r8    |                       |
| store | r8     | ⇒ r1 | //          |                       |

The two short versions of r1 each overlap with fewer values, which simplifies the allocation problem. Such "spilling" will (eventually) create a code where the allocator can succeed.



# Live Ranges in a single block

---

Your allocator is not bound by the names used in its input

- Every computed value is part of some live range
  - Even if it has no name in the source code (e.g.,  $2 * y$  in  $x - 2 * y$ )
- A live range usually has a single name, such as  $r_{17}$
- A single name (SSA!) with multiple values can be renamed into distinct live ranges

| Operation |               |                             |                                    | Live Ranges |    |
|-----------|---------------|-----------------------------|------------------------------------|-------------|----|
| loadI     | @base         | $\Rightarrow$ $r_{arp}$     | none                               | [1,11]      | 1  |
| loadAI    | $r_{arp}, @a$ | $\Rightarrow$ $r_a$         | $r_{arp}, r_a$                     | [2,7]       | 2  |
| loadI     | 2             | $\Rightarrow$ $r_2$         | $r_{arp}, r_a, r_2$                | [3,7]       | 3  |
| loadAI    | $r_{arp}, @b$ | $\Rightarrow$ $r_b$         | $r_{arp}, r_a, r_2$                | [4,8]       | 4  |
| loadAI    | $r_{arp}, @c$ | $\Rightarrow$ $r_c$         | $r_{arp}, r_a, r_b, r_2$           | [5,9]       | 5  |
| loadAI    | $r_{arp}, @d$ | $\Rightarrow$ $r_d$         | $r_{arp}, r_a, r_c, r_b, r_2$      | [6,10]      | 6  |
| mult      | $r_a, r_2$    | $\Rightarrow$ $r_a$         | $r_{arp}, r_a, r_d, r_c, r_b, r_2$ | [7,8]       | 7  |
| mult      | $r_a, r_b$    | $\Rightarrow$ $r_a$         | $r_{arp}, r_a, r_d, r_c, r_b$      | [8,9]       | 8  |
| mult      | $r_a, r_c$    | $\Rightarrow$ $r_a$         | $r_{arp}, r_a, r_d, r_c$           | [9,10]      | 9  |
| mult      | $r_a, r_d$    | $\Rightarrow$ $r_a$         | $r_{arp}, r_a, r_d$                | [10,11]     | 10 |
| storeAI   | $r_a$         | $\Rightarrow$ $r_{arp}, @w$ |                                    |             | 11 |

$MEM(w) \leftarrow (a * 2 * b * c * d)$

There are five distinct values, or live ranges, named  $r_a$

# Live Ranges

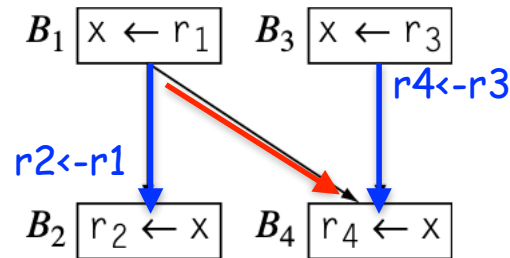
---

Your allocator is not bound by the names used in its input

- Every computed value is part of some live range
  - Even if it has no name in the source code (e.g.,  $2 * y$  in  $x - 2 * y$ )
- A live range usually has a single name, such as  $r_{17}$
- A single name with multiple values can be renamed into distinct live ranges
  
- Renaming distinct live ranges with distinct names can simplify the implementation of the allocator

# From local algorithms to regional algorithms

- Extending local algorithms to regional ones can be difficult



- the only solution is to store back in memory the value of  $x$  at the end of  $B_1$  and  $B_2$
- while we could add register to register operation for blue arrow no possibility exists for the red arrow