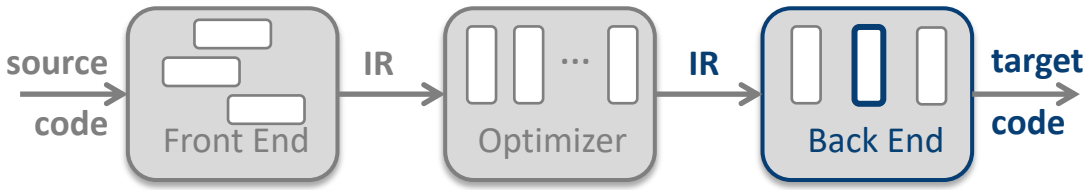
RICE

COMP 412
FALL 2018

Lab 2 Tutorial

Comp 412



Software Tools for Lab 2

A number of tools to help you build and debug Lab 2

- An **ILOC** Simulator
  - Essentially, an interpreter for Lab 2 ILOC
  - Matches the Lab 2 specs
    - → Single functional unit, latencies as specified, **ILOC** Lab 1 subset enforced
  - For code check, implements **–x** command line option
- Lab 2 Reference Implementation
  - A functioning Lab 2 allocator, available as an executable black box
  - Has additional functionality relative to your allocator
  - Runs **only** on CLEAR
- Various scripts, in the scripts directory
- A library of **ILOC** programs

All of these tools are found on **CLEAR** in ~comp412/students/lab2

Software Tools for Lab 2

Lab 2 ILOC Simulator

- Takes as input an **ILOC** program, interprets it, and produces the results
- Can initialize memory from the command line
- Can use **ILOC**'s output operation to print values from memory to stdout

```
% cat ex1.iloc
// add two numbers
  loadI  314    => r0
  loadI  0      => r1
  load   r1     => r1
  add    r0,r1  => r2
  loadI  0      => r0
  store  r2     => r0
  output 0
% sim -i 0 14 17 < ex1.iloc
328

Executed 7 instructions and 7 operations in 11 cycles.
```

Loads 14 and 17 into memory, starting at address 0

Software Tools for Lab 2

Lab 1 ILOC Simulator

```
% cat ex1.iloc
// add two numbers
  loadI 314    => r0
  loadI   0    => r1
  load  r1     => r1
  add   r0,r1  => r2
  loadI 0      => r0
  store r2     => r0
  output 0
%
```

```
% sim -t –i 0 14 17  ex1.iloc
ILOC Simulator, Version 412-2015-2
Interlock settings     memory registers branches

0:    [loadI 314 => r0 (314)]
1:    [loadI 0 => r1 (0)]
2:    [load r1 (addr: 0) => r1 (14)]
3:    [ stall ]
4:    [ stall ] *2
5:    [add r0 (314), r1 (14) => r2 (328)]
6:    [loadI 0 => r0 (0)]
7:    [store r2 (328) => r0 (addr: 0)]
8:    [ stall ]
9:    [ stall ] *7
10:   [output 0 (328)]
output generates => 328

Executed 7 instructions and 7 operations in 11 cycles.
```

Software Tools for Lab 2

Lab 2 Reference Allocator

- The reference implementation shows you what a reasonably good allocator might produce for a given input block
- lab2_ref implements the algorithms from class
  - Implementation in C, so it runs quickly
  - Uses the linked list data structure shown in class (singly linked)
  - Backward walk (renaming) is done with recursion (as in **foldr**)
  - Forward walk is a simple pointer-chasing loop
- lab2_ref produces commented output that may be helpful

You can beat lab2_ref in allocation quality

- Last year, several students beat it on several of the report blocks

Software Tools for Lab 2

The Scripts directory has scripts to run and test an allocator

- **AllocAndRunAll**
  - Applies an allocator to an entire directory of code and produces the results of running the resulting allocated code in the simulator
  - Relies on the **SIM INPUT** and **OUTPUT** comments for test blocks
  - Only run one instance at a time; the scripts use scratch files in your home directory that are naively named

- **CodeCheck**
  - Scripts for the CodeChecks are in /clear/courses/comp412/students/lab2/code_check_1 and /clear/courses/comp412/students/lab2/code_check_2
  - They will run five specific **ILOC** blocks
  - You should run your allocator inside the scripts to test it

What Matters?

In Lab 2, 75% of the points depend on the correctness of your allocator and the quality of the allocated code that your lab produces

- We measure performance by looking at the total number of cycles that the allocated code uses in the **Lab 2 ILOC Simulator**[1]
  - Take the unallocated code & run it
  - Take the allocated code & run it
  - Make sure the answers are identical
  - Difference between the two runs is the cost of spill code
- The objective of Lab 2 is to minimize the cost of the spill code
  - The credit for performance is based on how your lab does relative to the labs of other students & relative to the reference implementation
  - Register allocation can introduce a **lot** of spill code
  - Attention to detail can reduce spill costs
  - **But**, you cannot always win
- Allocator speed will be graded as part of your report

SO, focus **1st** on the quality of the allocated code & **2nd** on allocator speed

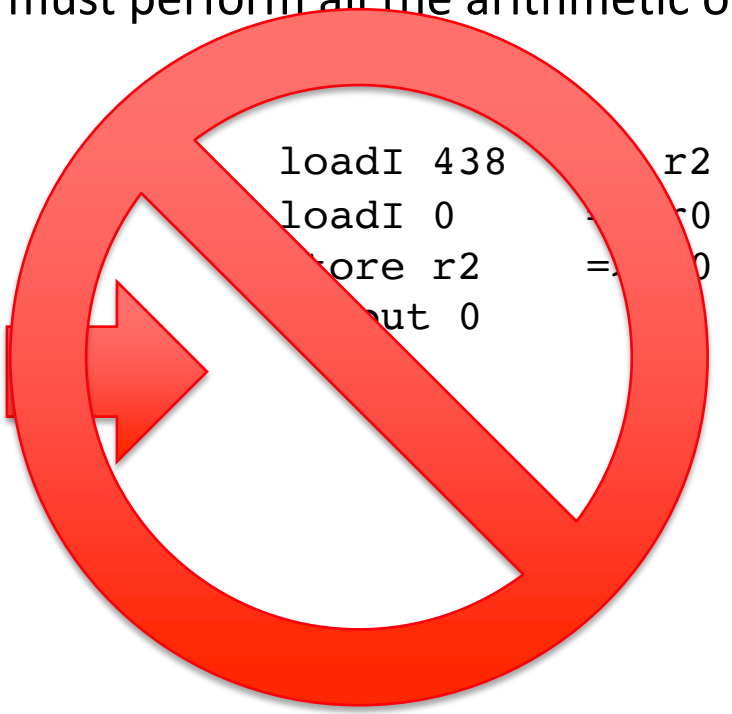
[1] Lab 3 will use a different simulator, with different latencies & execution order constraints.

Optimizations

The only optimizations allowed are register allocation and NOP elimination

- No other optimizations allowed!
- Specifically, no constant propagation, value numbering etc. You might be tempted, as some code blocks define the input values. Don't do it!
- The code you generate must perform all the arithmetic operations that the original code does

```
loadI 412    => r0
loadI  26    => r1
add   r0,r1 => r2
loadI 0      => r0
store r2     => r0
output 0
```



Data structures and algorithms

Think ahead

- What works for Code Check 1 may not work for Code Check 2 or the final submission
  - Forward pass is perfectly fine for renaming, but you can't compute next use easily
  - Backward pass is a bit tricky with a single-linked list
    - → Comp215 **foldr** for the win!
- What works for Lab 2 may not work for Lab 3
- For Lab2, you need to be able to insert spill and restore instructions
- For Lab3, you need to be able to move instructions around
- Pointer chasing is much faster than hashtable lookup
- Array lookup is much faster than a hashtable lookup
- Make reasonable assumptions and document them
  - i.e. "We don't expect the number of registers in the source code to be larger than 2 Billion"

Renaming Example from Lecture 11

An Example Block in ILOC

```
loadI   128      => r0   // r0 ← addr(a)
load    r0       => r1   // r1 ← a
loadI   132      => r2   // r2 ← addr(b)
load    r2       => r3   // r3 ← b
loadI   136      => r4   // r4 ← addr(c)
load    r4       => r5   // r5 ← c
mult    r3, r5   => r3   // r3 ← b * c
add     r1, r3   => r1   // r1 ← a – b * c
store   r1       => r0   // a  ← a – b * c
```

Allocator decides which values are in registers, and chooses a physical register for each value

- Easy if |*values*| ≤ |*registers*|, otherwise hard
- Must discover "values" or "live ranges"

Assumptions:

- Computes a ⇐ a + b * c
- Uses Lab2 **ILOC** subset
- a is stored at **128**, b at **132**, and c at **136**
- // denotes the start of a comment; the scanner can ignore // and any characters to the end of the line
- Slides will use **r** for a register in the input, **vr** for a virtual register and **pr** for a physical register
- Code reuses registers

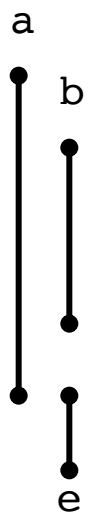
Finding Live Ranges

"Renaming" for Lab 2 Code Check 1

A value is *live* from its *definition* (x ← …) to its *last use* (y ← … x …)

- In a basic block, the notion is straightforward
  - *The live range is the interval from a definition to the last use of that value*
- In a setting with control flow, the situation is more complex (♣ 13.4.1)

#	Operation
0	a ← …
1	b ← …
2	c ← … a …
3	d ← … b …
4	e ← … a …
5	f ← … e …



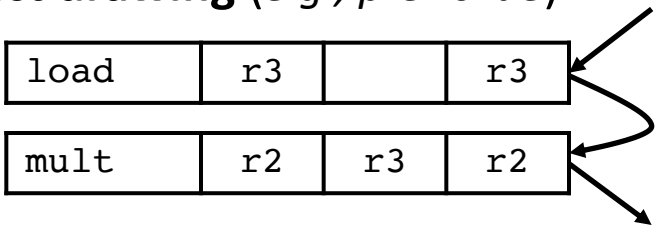
Simple Example

- a's live range is [0,4]
- b's live range is [1,3]
- e's live range is [4,5]
- Live ranges may, of course, overlap
  - a & b are simultaneously live, so they cannot occupy the same **PR**
  - a & e can occupy the same **PR**, as could b & e

Representing **ILOC**

To make the exposition of the algorithms more detailed, we need to be a little more concrete, particularly with regard to the IR

Abstract drawing (*e.g., prev slide*)



Implementation decision:
have separate fields for the **SR**, **VR**, & **PR**.

Concrete version

INDEX	OPCODE	OP1				OP2				OP3				NEXT OP
		SR	VR	PR	NU	SR	VR	PR	NU	SR	VR	PR	NU	
5	load	r4	—	—	∞	—	—	—	∞	r5	—	—	∞	6
6	mult	r3	—	—	∞	r5	—	—	∞	r5	—	—	∞	7

- Each operand has names for its **SR**, **VR**, **PR**, and Next Use
- Front end initializes all fields

In general, implementation should begin with data structure design.

Note: *SRToVR and LU are maps over relatively compact sets — use a vector.*

Renaming

(Lab 2 Code Check 1)

```
Update(OP, index) {
   if SRToVR[OP.SR] == -1     // SR has no VR
       then SRToVR[OP.SR] = VRName++
   OP.VR = SRToVR[OP.SR]
   OP.NU = LU[OP.SR]
   LU[OP.SR] = index
}
VRName = 0 //  execution starts here
for i = 0 to max SR number
    SRToVR[i] = invalid
    LU[i] = ∞
for i = n to 0 by -1  {
    Update(OPS[i].OP3, i)      // update
    if (OPS[i].OP3 is a DEF){   //  kill if DEF
       SRToVR[OPS[i].OP3.SR] =  -1
       LU[OPS[i].OP3.SR] = ∞
    }
    Update(OPS[i].OP1, i)      // update one use
    Update(OPS[i].OP2, i)      // & the other use
}
```

Finding live ranges, next uses, and renaming to VRs

- Algorithm walks the basic block from last op to first op
- If **SR** has no **VR**, assigns one (indicates a last use …)
- Renames **SR** to **VR** at each mention of the **SR** in the code
- Tags each mention with the index of the **VR**'s next use
- Assumes that all ops are three address ops, but easy to adapt to other operation formats

*SRToPR[ ] and LU[ ] need an entry for each **SR** (keep track during parsing; it must be bounded by the block length)*

Back to the Example

Initial State of the Algorithm

INDEX	OPCODE	OP1				OP2				OP3				NEXT OP
		SR	VR	PR	NU	SR	VR	PR	NU	SR	VR	PR	NU	
0	loadI	128	–	–	∞	–	–	–	∞	r0	–	–	∞	1
1	load	r0	–	–	∞	–	–	–	∞	r1	–	–	∞	2
2	loadI	132	–	–	∞	–	–	–	∞	r2	–	–	∞	3
3	load	r2	–	–	∞	–	–	–	∞	r3	–	–	∞	4
4	loadI	136	–	–	∞	–	–	–	∞	r4	–	–	∞	5
5	load	r4	–	–	∞	–	–	–	∞	r5	–	–	∞	6
6	mult	r3	–	–	∞	r5	–	–	∞	r3	–	–	∞	7
7	add	r1	–	–	∞	r3	–	–	∞	r1	–	–	∞	8
8	store	r1	–	–	∞	–	–	–	∞	r0	–	–	∞	∞

index →

	0	1	2	3	4	5
SRToVR	–	–	–	–	–	–
LU	∞	∞	∞	∞	∞	∞

VRName: 0

Not a hash map in sight!

Back to the Example

After Processing Operation 8

INDEX	OPCODE	OP1				OP2				OP3				NEXT OP
		SR	VR	PR	NU	SR	VR	PR	NU	SR	VR	PR	NU	
0	loadI	128	–	–	∞	–	–	–	∞	r0	–	–	∞	1
1	load	r0	–	–	∞	–	–	–	∞	r1	–	–	∞	2
2	loadI	132	–	–	∞	–	–	–	∞	r2	–	–	∞	3
3	load	r2	–	–	∞	–	–	–	∞	r3	–	–	∞	4
4	loadI	136	–	–	∞	–	–	–	∞	r4	–	–	∞	5
5	load	r4	–	–	∞	–	–	–	∞	r5	–	–	∞	6
6	mult	r3	–	–	∞	r5	–	–	∞	r3	–	–	∞	7
7	add	r1	–	–	∞	r3	–	–	∞	r1	–	–	∞	8
8	store	r1	vr0	–	∞	–	–	–	∞	r0	vr1	–	∞	∞

index →

	0	1	2	3	4	5
SRToVR	1	0	–	–	–	–
LU	8	8	∞	∞	∞	∞

VRName: 2

store has no def

Update r1 as a use

Update r0 as a use

Back to the Example

After Processing Operation 7

INDEX	OPCODE	OP1				OP2				OP3				NEXT OP
		SR	VR	PR	NU	SR	VR	PR	NU	SR	VR	PR	NU	
0	loadI	128	–	–	∞	–	–	–	∞	r0	–	–	∞	1
1	load	r0	–	–	∞	–	–	–	∞	r1	–	–	∞	2
2	loadI	132	–	–	∞	–	–	–	∞	r2	–	–	∞	3
3	load	r2	–	–	∞	–	–	–	∞	r3	–	–	∞	4
4	loadI	136	–	–	∞	–	–	–	∞	r4	–	–	∞	5
5	load	r4	–	–	∞	–	–	–	∞	r5	–	–	∞	6
6	mult	r3	–	–	∞	r5	–	–	∞	r3	–	–	∞	7
7	add	r1	vr2	–	∞	r3	vr3	–	∞	r1	vr0	–	8	8
8	store	r1	vr0	–	∞	–	–	–	∞	r0	vr1	–	∞	∞

index →

	0	1	2	3	4	5
SRToVR	1	–	–	–	–	–
LU	8	∞	∞	∞	∞	∞

VRName: 2

Update r1 as a def
Kill r1
Update NU of OP3

Back to the Example

After Processing Operation 7

INDEX	OPCODE	OP1				OP2				OP3				NEXT OP
		SR	VR	PR	NU	SR	VR	PR	NU	SR	VR	PR	NU	
0	loadI	128	–	–	∞	–	–	–	∞	r0	–	–	∞	1
1	load	r0	–	–	∞	–	–	–	∞	r1	–	–	∞	2
2	loadI	132	–	–	∞	–	–	–	∞	r2	–	–	∞	3
3	load	r2	–	–	∞	–	–	–	∞	r3	–	–	∞	4
4	loadI	136	–	–	∞	–	–	–	∞	r4	–	–	∞	5
5	load	r4	–	–	∞	–	–	–	∞	r5	–	–	∞	6
6	mult	r3	–	–	∞	r5	–	–	∞	r3	–	–	∞	7
7	add	r1	vr2	–	∞	r3	vr3	–	∞	r1	vr0	–	8	8
8	store	r1	vr0	–	∞	–	–	–	∞	r0	vr1	–	∞	∞

index →

	0	1	2	3	4	5
SRToVR	1	2	–	3	–	–
LU	8	7	∞	7	∞	∞

VRName: 4

Update r1 as a use
Update r3 as a use

Back to the Example

After Processing Operation 6

INDEX	OPCODE	OP1				OP2				OP3				NEXT OP
		SR	VR	PR	NU	SR	VR	PR	NU	SR	VR	PR	NU	
0	loadI	128	–	–	∞	–	–	–	∞	r0	–	–	∞	1
1	load	r0	–	–	∞	–	–	–	∞	r1	–	–	∞	2
2	loadI	132	–	–	∞	–	–	–	∞	r2	–	–	∞	3
3	load	r2	–	–	∞	–	–	–	∞	r3	–	–	∞	4
4	loadI	136	–	–	∞	–	–	–	∞	r4	–	–	∞	5
5	load	r4	–	–	∞	–	–	–	∞	r5	–	–	∞	6
6	mult	r3	vr4	–	∞	r5	vr5	–	∞	r3	vr3	–	7	7
7	add	r1	vr2	–	∞	r3	vr3	–	∞	r1	vr0	–	8	8
8	store	r1	vr0	–	∞	–	–	–	∞	r0	vr1	–	∞	∞

index →

	0	1	2	3	4	5
SRToVR	1	2	–	–	–	–
LU	8	7	∞	∞	∞	∞

VRName: 4

Update r3 as a def

Kill r3

Update NU for OP3

Back to the Example

After Processing Operation 6

INDEX	OPCODE	OP1				OP2				OP3				NEXT OP
		SR	VR	PR	NU	SR	VR	PR	NU	SR	VR	PR	NU	
0	loadI	128	–	–	∞	–	–	–	∞	r0	–	–	∞	1
1	load	r0	–	–	∞	–	–	–	∞	r1	–	–	∞	2
2	loadI	132	–	–	∞	–	–	–	∞	r2	–	–	∞	3
3	load	r2	–	–	∞	–	–	–	∞	r3	–	–	∞	4
4	loadI	136	–	–	∞	–	–	–	∞	r4	–	–	∞	5
5	load	r4	–	–	∞	–	–	–	∞	r5	–	–	∞	6
6	mult	r3	vr4	–	∞	r5	vr5	–	∞	r3	vr3	–	7	7
7	add	r1	vr2	–	∞	r3	vr3	–	∞	r1	vr0	–	8	8
8	store	r1	vr0	–	∞	–	–	–	∞	r0	vr1	–	∞	∞

index →

	0	1	2	3	4	5
SRToVR	1	2	–	4	–	5
LU	8	7	∞	6	∞	6

VRName: 6

Update r3 as a use
Update r5 as a use

Back to the Example

After Processing Operation 5

INDEX	OPCODE	OP1				OP2				OP3				NEXT OP
		SR	VR	PR	NU	SR	VR	PR	NU	SR	VR	PR	NU	
0	loadI	128	–	–	∞	–	–	–	∞	r0	–	–	∞	1
1	load	r0	–	–	∞	–	–	–	∞	r1	–	–	∞	2
2	loadI	132	–	–	∞	–	–	–	∞	r2	–	–	∞	3
3	load	r2	–	–	∞	–	–	–	∞	r3	–	–	∞	4
4	loadI	136	–	–	∞	–	–	–	∞	r4	–	–	∞	5
5	load	r4	vr6	–	∞	–	–	–	∞	r5	vr5	–	6	6
6	mult	r3	vr4	–	∞	r5	vr5	–	∞	r3	vr3	–	7	7
7	add	r1	vr2	–	∞	r3	vr3	–	∞	r1	vr0	–	8	8
8	store	r1	vr0	–	∞	–	–	–	∞	r0	vr1	–	∞	∞

index →

	0	1	2	3	4	5
SRToVR	1	2	–	4	6	–
LU	8	7	∞	6	5	∞

VRName: 7

Update r5 as a def

Update r4 as a use

Back to the Example

After Processing Operation 4

INDEX	OPCODE	OP1				OP2				OP3				NEXT OP
		SR	VR	PR	NU	SR	VR	PR	NU	SR	VR	PR	NU	
0	loadI	128	–	–	∞	–	–	–	∞	r0	–	–	∞	1
1	load	r0	–	–	∞	–	–	–	∞	r1	–	–	∞	2
2	loadI	132	–	–	∞	–	–	–	∞	r2	–	–	∞	3
3	load	r2	–	–	∞	–	–	–	∞	r3	–	–	∞	4
4	loadI	136	–	–	∞	–	–	–	∞	r4	vr6	–	5	5
5	load	r4	vr6	–	∞	–	–	–	∞	r5	vr5	–	6	6
6	mult	r3	vr4	–	∞	r5	vr5	–	∞	r3	vr3	–	7	7
7	add	r1	vr2	–	∞	r3	vr3	–	∞	r1	vr0	–	8	8
8	store	r1	vr0	–	∞	–	–	–	∞	r0	vr1	–	∞	∞

index →

	0	1	2	3	4	5
SRToVR	1	2	–	4	–	–
LU	8	7	∞	6	∞	∞

VRName: 7 ***Update r4 as a def***

Back to the Example

After Processing Operation 3

INDEX	OPCODE	OP1				OP2				OP3				NEXT OP
		SR	VR	PR	NU	SR	VR	PR	NU	SR	VR	PR	NU	
0	loadI	128	–	–	∞	–	–	–	∞	r0	–	–	∞	1
1	load	r0	–	–	∞	–	–	–	∞	r1	–	–	∞	2
2	loadI	132	–	–	∞	–	–	–	∞	r2	–	–	∞	3
3	load	r2	vr7	–	∞	–	–	–	∞	r3	vr4	–	6	4
4	loadI	136	–	–	∞	–	–	–	∞	r4	vr6	–	5	5
5	load	r4	vr6	–	∞	–	–	–	∞	r5	vr5	–	6	6
6	mult	r3	vr4	–	∞	r5	vr5	–	∞	r3	vr3	–	7	7
7	add	r1	vr2	–	∞	r3	vr3	–	∞	r1	vr0	–	8	8
8	store	r1	vr0	–	∞	–	–	–	∞	r0	vr1	–	∞	∞

index →

	0	1	2	3	4	5
SRToVR	1	2	7	–	–	–
LU	8	7	3	∞	∞	∞

VRName: 8

Update r3 as a def

Update r2 as a use

Back to the Example

After Processing Operation 2

index →

INDEX	OPCODE	OP1				OP2				OP3				NEXT OP
		SR	VR	PR	NU	SR	VR	PR	NU	SR	VR	PR	NU	
0	loadI	128	–	–	∞	–	–	–	∞	r0	–	–	∞	1
1	load	r0	–	–	∞	–	–	–	∞	r1	–	–	∞	2
2	loadI	132	–	–	∞	–	–	–	∞	r2	vr7	–	3	3
3	load	r2	vr7	–	∞	–	–	–	∞	r3	vr4	–	6	4
4	loadI	136	–	–	∞	–	–	–	∞	r4	vr6	–	5	5
5	load	r4	vr6	–	∞	–	–	–	∞	r5	vr5	–	6	6
6	mult	r3	vr4	–	∞	r5	vr5	–	∞	r3	vr3	–	7	7
7	add	r1	vr2	–	∞	r3	vr3	–	∞	r1	vr0	–	8	8
8	store	r1	vr0	–	∞	–	–	–	∞	r0	vr1	–	∞	∞

	0	1	2	3	4	5
SRToVR	1	2	–	–	–	–
LU	8	7	∞	∞	∞	∞

VRName: 8 ***Update r2 as a def***

Back to the Example

After Processing Operation 1

index →

INDEX	OPCODE	OP1				OP2				OP3				NEXT OP
		SR	VR	PR	NU	SR	VR	PR	NU	SR	VR	PR	NU	
0	loadI	128	–	–	∞	–	–	–	∞	r0	–	–	∞	1
1	load	r0	vr1	–	8	–	–	–	∞	r1	vr2	–	7	2
2	loadI	132	–	–	∞	–	–	–	∞	r2	vr7	–	3	3
3	load	r2	vr7	–	∞	–	–	–	∞	r3	vr4	–	6	4
4	loadI	136	–	–	∞	–	–	–	∞	r4	vr6	–	5	5
5	load	r4	vr6	–	∞	–	–	–	∞	r5	vr5	–	6	6
6	mult	r3	vr4	–	∞	r5	vr5	–	∞	r3	vr3	–	7	7
7	add	r1	vr2	–	∞	r3	vr3	–	∞	r1	vr0	–	8	8
8	store	r1	vr0	–	∞	–	–	–	∞	r0	vr1	–	∞	∞

	0	1	2	3	4	5
SRToVR	1	–	–	–	–	–
LU	1	∞	∞	∞	∞	∞

VRName: 8

Update r1 as a def

Update r0 as a use

Update NU of OP1

Back to the Example

After Processing Operation 0

index →

INDEX	OPCODE	OP1				OP2				OP3				NEXT OP
		SR	VR	PR	NU	SR	VR	PR	NU	SR	VR	PR	NU	
0	loadI	128	–	–	∞	–	–	–	∞	r0	vr1	–	1	1
1	load	r0	vr1	–	8	–	–	–	∞	r1	vr2	–	7	2
2	loadI	132	–	–	∞	–	–	–	∞	r2	vr7	–	3	3
3	load	r2	vr7	–	∞	–	–	–	∞	r3	vr4	–	6	4
4	loadI	136	–	–	∞	–	–	–	∞	r4	vr6	–	5	5
5	load	r4	vr6	–	∞	–	–	–	∞	r5	vr5	–	6	6
6	mult	r3	vr4	–	∞	r5	vr5	–	∞	r3	vr3	–	7	7
7	add	r1	vr2	–	∞	r3	vr3	–	∞	r1	vr0	–	8	8
8	store	r1	vr0	–	∞	–	–	–	∞	r0	vr1	–	∞	∞

	0	1	2	3	4	5
SRToVR	–	–	–	–	–	–
LU	∞	∞	∞	∞	∞	∞

VRName: 8 ***Update r0 as a def***

Back to the Example

Code renamed so that virtual registers correspond to live ranges

INDEX	OPCODE	OP1				OP2				OP3				NEXT OP
		SR	VR	PR	NU	SR	VR	PR	NU	SR	VR	PR	NU	
0	loadI	128	–	–	∞	–	–	–	∞	r0	vr1	–	1	1
1	load	r0	vr1	–	8	–	–	–	∞	r1	vr2	–	7	2
2	loadI	132	–	–	∞	–	–	–	∞	r2	vr7	–	3	3
3	load	r2	vr7	–	∞	–	–	–	∞	r3	vr4	–	6	4
4	loadI	136	–	–	∞	–	–	–	∞	r4	vr6	–	5	5
5	load	r4	vr6	–	∞	–	–	–	∞	r5	vr5	–	6	6
6	mult	r3	vr4	–	∞	r5	vr5	–	∞	r3	vr3	–	7	7
7	add	r1	vr2	–	∞	r3	vr3	–	∞	r1	vr0	–	8	8
8	store	r1	vr0	–	∞	–	–	–	∞	r0	vr1	–	∞	∞

What about **MAXLIVE**?

- **MAXLIVE** is the largest number of live entries in the *SRToVR* table during the algorithm's execution — or 4 in this example (*at ops 4 & 5*)

Back to the Example

Code renamed so that virtual registers correspond to live ranges

Operation			Live Ranges								MAXLIVE
			VR0	VR1	VR2	VR3	VR4	VR5	VR6	VR7	@ end of op
loadI	128	=> vr1		•							1
load	vr1	=> vr2			•						2
loadI	132	=> vr7								•	3
load	vr7	=> vr4					•				3
loadI	136	=> vr6							•		4
load	vr6	=> vr5					•	•			4
mult	vr4,vr5	=> vr3			•	•					3
add	vr2,vr3	=> vr0	•	•							2
store	vr0	=> vr1	*Live at the end of the operation*								0

For the code check 1, your lab must rename registers & print out the renamed **ILOC.**

Allocation

Allocation

Assume that the allocator has already renamed source registers into virtual registers and computed next use information

	OPCODE	OP1			OP2			OP3		
		VR	PR	NU	VR	PR	NU	VR	PR	NU
0	loadI	0		∞			∞	vr0		2
1	loadI	4		∞			∞	vr1		2
2	add	vr0		3	vr1		7	vr2		3
3	add	vr0		4	vr2		5	vr3		4
4	store	vr3		∞			∞	vr0		5
5	store	vr2		∞			∞	vr0		7
6	output	0		∞			∞			∞
7	store	vr1		∞			∞	vr0		∞
8	output	0		∞			∞			∞

Bottom-up Allocator

A bottom-up allocator synthesizes the allocation based on low-level knowledge of the code & the partial solution thus far computed.

```
for i ← 0 to n
    if (OP[i].OP1.PR is invalid)
        get a PR, say x, load OP[i].OP1.VR into x, and set OP[i].OP1.PR ← x
    if (OP[i].OP2.PR is invalid)
        get a PR, say y, load OP[i].OP2.VR into y, and set OP[i].OP2.PR ← y
    if either OP1 or OP2 is a last use
        free the corresponding PR
    Get a PR, say z, and set OP[i].OP3.PR ← z
```

The action "get a **PR**" is the heart of the algorithm.

- If a **PR** is free, "get a PR" is easy
- If no **PR** is free, code must choose an occupied **PR** and spill its contents to memory

The Block

Assume that we have pr0 & pr1 for values, and pr2 reserved for spilling

index →

	OPCODE	OP1			OP2			OP3		
		VR	PR	NU	VR	PR	NU	VR	PR	NU
0	loadI	0		∞			∞	vr0		2
1	loadI	4		∞			∞	vr1		2
2	add	vr0		3	vr1		7	vr2		3
3	add	vr0		4	vr2		5	vr3		4
4	store	vr3		∞			∞	vr0		5
5	store	vr2		∞			∞	vr0		7
6	output	0		∞			∞			∞
7	store	vr1		∞			∞	vr0		∞
8	output	0		∞			∞			∞

- Op1 needs no register
- Op2 needs no register
- Op3 needs a register; allocate pr0 to Op3 & update the maps

	vr0	vr1	vr2	vr3
VRToPR	—	—	—	—

	pr0	pr1	pr2
PRToVR	—	—	—
Next Use	∞	∞	∞

The Block

Assume that we have pr0 & pr1 for values, and pr2 reserved for spilling

	OPCODE	OP1			OP2			OP3		
		VR	PR	NU	VR	PR	NU	VR	PR	NU
0	loadI	0		∞			∞	vr0	pr0	2
1	loadI	4		∞			∞	vr1		2
2	add	vr0		3	vr1		7	vr2		3
3	add	vr0		4	vr2		5	vr3		4
4	store	vr3		∞			∞	vr0		5
5	store	vr2		∞			∞	vr0		7
6	output	0		∞			∞			∞
7	store	vr1		∞			∞	vr0		∞
8	output	0		∞			∞			∞

index →

After processing 0

Advance the index

	vr0	vr1	vr2	vr3
VRToPR	pr0	—	—	—

	pr0	pr1	pr2
PRToVR	vr0	—	—
Next Use	2	∞	∞

The Block

Assume that we have pr0 & pr1 for values, and pr2 reserved for spilling

	OPCODE	OP1			OP2			OP3		
		VR	PR	NU	VR	PR	NU	VR	PR	NU
0	loadI	0		∞			∞	vr0	pr0	2
1	loadI	4		∞			∞	vr1		2
2	add	vr0		3	vr1		7	vr2		3
3	add	vr0		4	vr2		5	vr3		4
4	store	vr3		∞			∞	vr0		5
5	store	vr2		∞			∞	vr0		7
6	output	0		∞			∞			∞
7	store	vr1		∞			∞	vr0		∞
8	output	0		∞			∞			∞

index →

Process operation 1

- Op1 needs no register
- Op2 needs no register
- Op3 needs a register; allocate pr1 to vr1 and update the maps

	vr0	vr1	vr2	vr3
VRToPR	pr0	—	—	—

	pr0	pr1	pr2
PRToVR	vr0	—	—
Next Use	2	∞	∞

The Block

Assume that we have pr0 & pr1 for values, and pr2 reserved for spilling

	OPCODE	OP1			OP2			OP3		
		VR	PR	NU	VR	PR	NU	VR	PR	NU
0	loadI	0		∞			∞	vr0	pr0	2
1	loadI	4		∞			∞	vr1	pr1	2
2	add	vr0		3	vr1		7	vr2		3
3	add	vr0		4	vr2		5	vr3		4
4	store	vr3		∞			∞	vr0		5
5	store	vr2		∞			∞	vr0		7
6	output	0		∞			∞			∞
7	store	vr1		∞			∞	vr0		∞
8	output	0		∞			∞			∞

index →

After processing 1

Advance the index

	vr0	vr1	vr2	vr3
VRToPR	pr0	pr1	—	—

	pr0	pr1	pr2
PRToVR	vr0	vr1	—
Next Use	2	2	∞

The Block

Assume that we have pr0 & pr1 for values, and pr2 reserved for spilling

index →

	OPCODE	OP1			OP2			OP3		
		VR	PR	NU	VR	PR	NU	VR	PR	NU
0	loadI	0		∞			∞	vr0	pr0	2
1	loadI	4		∞			∞	vr1	pr1	2
2	add	vr0	pr0	3	vr1	pr1	7	vr2		3
3	add	vr0		4	vr2		5	vr3		4
4	store	vr3		∞			∞	vr0		5
5	store	vr2		∞			∞	vr0		7
6	output	0		∞			∞			∞
7	store	vr1		∞			∞	vr0		∞
8	output	0		∞			∞			∞

	vr0	vr1	vr2	vr3
VRToPR	pr0	pr1	—	—

	pr0	pr1	pr2
PRToVR	vr0	vr1	—
Next Use	3	7	∞

Process operation 2

- Op1 is already in pr0
- Op2 is already in pr1
- Op3 needs a register but none is free; must spill pr0 or pr1.
  - → Next use info says that pr1's use is farther away than pr0's use
  - → Spill pr1 (before operation 2) & allocate pr1 to vr2

The Block

Assume that we have pr0 & pr1 for values, and pr2 reserved for spilling

Spill vr1

index

	OPCODE	OP1			OP2			OP3		
		VR	PR	NU	VR	PR	NU	VR	PR	NU
0	loadI	0		∞			∞	vr0	pr0	2
1	loadI	4		∞			∞	vr1	pr1	2
2	add	vr0	pr0	3	vr1	pr1	7	vr2		3
3	add	vr0		4	vr2		5	vr3		4
4	store	vr3		∞			∞	vr0		5
5	store	vr2		∞			∞	vr0		7
6	output	0		∞			∞			∞
7	store	vr1		∞			∞	vr0		∞
8	output	0		∞			∞			∞

	vr0	vr1	vr2	vr3
VRToPR	pr0	pr1	—	—

	pr0	pr1	pr2
PRToVR	vr0	vr1	—
Next Use	3	7	∞

Process operation 2

- Op1 is already in pr0
- Op2 is already in pr1
- Op3 needs a register but none is free; must spill pr0 or pr1.
  - → Next use info says that pr1's use is farther away than pr0's use
  - → Spill pr1 (before operation 2) & allocate pr1 to vr2

The Block

Assume that we have pr0 & pr1 for values, and pr2 reserved for spilling

Spill vr1

index

	OPCODE	OP1			OP2			OP3		
		VR	PR	NU	VR	PR	NU	VR	PR	NU
0	loadI	0		∞			∞	vr0	pr0	2
1	loadI	4		∞			∞	vr1	pr1	2
2	add	vr0	pr0	3	vr1	pr1	7	vr2	pr1	3
3	add	vr0		4	vr2		5	vr3		4
4	store	vr3		∞			∞	vr0		5
5	store	vr2		∞			∞	vr0		7
6	output	0		∞			∞			∞
7	store	vr1		∞			∞	vr0		∞
8	output	0		∞			∞			∞

After Processing 2

Advance the index

	vr0	vr1	vr2	vr3
VRToPR	pr0	—	pr1	—

	pr0	pr1	pr2
PRToVR	vr0	vr2	—
Next Use	3	3	∞

The Block

Assume that we have pr0 & pr1 for values, and pr2 reserved for spilling

Spill vr1

index

	OPCODE	OP1			OP2			OP3		
		VR	PR	NU	VR	PR	NU	VR	PR	NU
0	loadI	0		∞			∞	vr0	pr0	2
1	loadI	4		∞			∞	vr1	pr1	2
2	add	vr0	pr0	3	vr1	pr1	7	vr2	pr1	3
3	add	vr0	pr0	4	vr2	pr1	5	vr3		4
4	store	vr3		∞			∞	vr0		5
5	store	vr2		∞			∞	vr0		7
6	output	0		∞			∞			∞
7	store	vr1		∞			∞	vr0		∞
8	output	0		∞			∞			∞

	vr0	vr1	vr2	vr3
VRToPR	pr0	—	pr1	—

	pr0	pr1	pr2
PRToVR	vr0	vr2	—
Next Use	4	5	∞

Process operation 3

- Op1 is already in pr0
- Op2 is already in pr1
- Op3 needs a register; must spill pr0 or pr1
  - → Next use info says that pr1's use is farther away than pr0's
  - → Spill pr1 (before operation 3) & allocate pr1 to vr3

The Block

Assume that we have pr0 & pr1 for values, and pr2 reserved for spilling

Spill vr1

Spill vr2

index →

	OPCODE	OP1			OP2			OP3		
		VR	PR	NU	VR	PR	NU	VR	PR	NU
0	loadI	0		∞			∞	vr0	pr0	2
1	loadI	4		∞			∞	vr1	pr1	2
2	add	vr0	pr0	3	vr1	pr1	7	vr2	pr1	3
3	add	vr0	pr0	4	vr2	pr1	5	vr3		4
4	store	vr3		∞			∞	vr0		5
5	store	vr2		∞			∞	vr0		7
6	output	0		∞			∞			∞
7	store	vr1		∞			∞	vr0		∞
8	output	0		∞			∞			∞

	vr0	vr1	vr2	vr3
VRToPR	pr0	—	pr1	—

	pr0	pr1	pr2
PRToVR	vr0	vr2	—
Next Use	4	5	∞

Process operation 3

- Op1 is already in pr0
- Op2 is already in pr1
- Op3 needs a register; must spill pr0 or pr1
  - → Next use info says that pr1's use is farther away than pr0's
  - → Spill pr1 (before operation 3) & allocate pr1 to vr3

The Block

Assume that we have pr0 & pr1 for values, and pr2 reserved for spilling

Spill vr1

Spill vr2

index

	OPCODE	OP1			OP2			OP3		
		VR	PR	NU	VR	PR	NU	VR	PR	NU
0	loadI	0		∞			∞	vr0	pr0	2
1	loadI	4		∞			∞	vr1	pr1	2
2	add	vr0	pr0	3	vr1	pr1	7	vr2	pr1	3
3	add	vr0	pr0	4	vr2	pr1	5	vr3	pr1	4
4	store	vr3		∞			∞	vr0		5
5	store	vr2		∞			∞	vr0		7
6	output	0		∞			∞			∞
7	store	vr1		∞			∞	vr0		∞
8	output	0		∞			∞			∞

After Processing 3

Advance the index

	vr0	vr1	vr2	vr3
VRToPR	pr0	—	—	pr1

	pr0	pr1	pr2
PRToVR	vr0	vr3	—
Next Use	4	4	∞

The Block

Assume that we have pr0 & pr1 for values, and pr2 reserved for spilling

Spill vr1

Spill vr2

index

	OPCODE	OP1			OP2			OP3		
		VR	PR	NU	VR	PR	NU	VR	PR	NU
0	loadI	0		∞			∞	vr0	pr0	2
1	loadI	4		∞			∞	vr1	pr1	2
2	add	vr0	pr0	3	vr1	pr1	7	vr2	pr1	3
3	add	vr0	pr0	4	vr2	pr1	5	vr3	pr1	4
4	store	vr3	pr1	∞			∞	vr0	pr0	5
5	store	vr2		∞			∞	vr0		7
6	output	0		∞			∞			∞
7	store	vr1		∞			∞	vr0		∞
8	output	0		∞			∞			∞

Process operation 4

- Op1 is in pr1
- Op3 is in pr0
  - Op3 is a **use**

	vr0	vr1	vr2	vr3
VRToPR	pr0	—	—	pr1

	pr0	pr1	pr2
PRToVR	vr0	vr3	—
Next Use	5	∞	∞

The Block

Assume that we have pr0 & pr1 for values, and pr2 reserved for spilling

Spill vr1

Spill vr2

index

	OPCODE	OP1			OP2			OP3		
		VR	PR	NU	VR	PR	NU	VR	PR	NU
0	loadI	0		∞			∞	vr0	pr0	2
1	loadI	4		∞			∞	vr1	pr1	2
2	add	vr0	pr0	3	vr1	pr1	7	vr2	pr1	3
3	add	vr0	pr0	4	vr2	pr1	5	vr3	pr1	4
4	store	vr3	pr1	∞			∞	vr0	pr0	5
5	store	vr2		∞			∞	vr0		7
6	output	0		∞			∞			∞
7	store	vr1		∞			∞	vr0		∞
8	output	0		∞			∞			∞

After processing 4

Advance the index

	vr0	vr1	vr2	vr3
VRToPR	pr0	—	—	—

	pr0	pr1	pr2
PRToVR	vr0	—	—
Next Use	5	∞	∞

The Block

Assume that we have pr0 & pr1 for values, and pr2 reserved for spilling

Spill vr1

Spill vr2

index

	OPCODE	OP1			OP2			OP3		
		VR	PR	NU	VR	PR	NU	VR	PR	NU
0	loadI	0		∞			∞	vr0	pr0	2
1	loadI	4		∞			∞	vr1	pr1	2
2	add	vr0	pr0	3	vr1	pr1	7	vr2	pr1	3
3	add	vr0	pr0	4	vr2	pr1	5	vr3	pr1	4
4	store	vr3	pr1	∞			∞	vr0	pr0	5
5	store	vr2		∞			∞	vr0		7
6	output	0		∞			∞			∞
7	store	vr1		∞			∞	vr0		∞
8	output	0		∞			∞			∞

	vr0	vr1	vr2	vr3
VRToPR	pr0	—	—	—

	pr0	pr1	pr2
PRToVR	vr0	—	—
Next Use	7	∞	∞

Process operation 5

- Op1 needs a register; pr1 is free (NU is ∞), so allocate vr2 to pr1 and restore the value from memory (before operation 5)
- Op3 is already in pr0
  - Op3 is a **use**

The Block

Assume that we have pr0 & pr1 for values, and pr2 reserved for spilling

Spill vr1

Spill vr2

Load vr2

index

	OPCODE	OP1			OP2			OP3		
		VR	PR	NU	VR	PR	NU	VR	PR	NU
0	loadI	0		∞			∞	vr0	pr0	2
1	loadI	4		∞			∞	vr1	pr1	2
2	add	vr0	pr0	3	vr1	pr1	7	vr2	pr1	3
3	add	vr0	pr0	4	vr2	pr1	5	vr3	pr1	4
4	store	vr3	pr1	∞			∞	vr0	pr0	5
5	store	vr2	pr1	∞			∞	vr0	pr0	7
6	output	0		∞			∞			∞
7	store	vr1		∞			∞	vr0		∞
8	output	0		∞			∞			∞

	vr0	vr1	vr2	vr3
VRToPR	pr0	—	pr1	—

	pr0	pr1	pr2
PRToVR	vr0	vr2	—
Next Use	7	∞	∞

Process operation 5

- Op1 needs a register; pr1 is free (NU is ∞), so allocate vr2 to pr1 and restore the value from memory (before operation 5)
- Op3 is already in pr0
  - Op3 is a **use**

The Block

Assume that we have pr0 & pr1 for values, and pr2 reserved for spilling

Spill vr1

Spill vr2

Load vr2

index

	OPCODE	OP1			OP2			OP3		
		VR	PR	NU	VR	PR	NU	VR	PR	NU
0	loadI	0		∞			∞	vr0	pr0	2
1	loadI	4		∞			∞	vr1	pr1	2
2	add	vr0	pr0	3	vr1	pr1	7	vr2	pr1	3
3	add	vr0	pr0	4	vr2	pr1	5	vr3	pr1	4
4	store	vr3	pr1	∞			∞	vr0	pr0	5
5	store	vr2	pr1	∞			∞	vr0	pr0	7
6	output	0		∞			∞			∞
7	store	vr1		∞			∞	vr0		∞
8	output	0		∞			∞			∞

After processing 5

Advance the index

	vr0	vr1	vr2	vr3
VRToPR	pr0	—	—	—

	pr0	pr1	pr2
PRToVR	vr0	—	—
Next Use	7	∞	∞

The Block

Assume that we have pr0 & pr1 for values, and pr2 reserved for spilling

Spill vr1

Spill vr2

Load vr2

index

	OPCODE	OP1			OP2			OP3		
		VR	PR	NU	VR	PR	NU	VR	PR	NU
0	loadI	0		∞			∞	vr0	pr0	2
1	loadI	4		∞			∞	vr1	pr1	2
2	add	vr0	pr0	3	vr1	pr1	7	vr2	pr1	3
3	add	vr0	pr0	4	vr2	pr1	5	vr3	pr1	4
4	store	vr3	pr1	∞			∞	vr0	pr0	5
5	store	vr2	pr1	∞			∞	vr0	pr0	7
6	output	0		∞			∞			∞
7	store	vr1		∞			∞	vr0		∞
8	output	0		∞			∞			∞

	vr0	vr1	vr2	vr3
VRToPR	pr0	—	—	—

	pr0	pr1	pr2
PRToVR	vr0	—	—
Next Use	7	∞	∞

Process operation 6

- "output" does not touch any registers
- Nothing to do here

The Block

Assume that we have pr0 & pr1 for values, and pr2 reserved for spilling

Spill vr1

Spill vr2

Load vr2

index

	OPCODE	OP1			OP2			OP3		
		VR	PR	NU	VR	PR	NU	VR	PR	NU
0	loadI	0		∞			∞	vr0	pr0	2
1	loadI	4		∞			∞	vr1	pr1	2
2	add	vr0	pr0	3	vr1	pr1	7	vr2	pr1	3
3	add	vr0	pr0	4	vr2	pr1	5	vr3	pr1	4
4	store	vr3	pr1	∞			∞	vr0	pr0	5
5	store	vr2	pr1	∞			∞	vr0	pr0	7
6	output	0		∞			∞			∞
7	store	vr1		∞			∞	vr0		∞
8	output	0		∞			∞			∞

After operation 6

Advance the index

	vr0	vr1	vr2	vr3
VRToPR	pr0	—	—	—

	pr0	pr1	pr2
PRToVR	vr0	—	—
Next Use	7	∞	∞

The Block

Assume that we have pr0 & pr1 for values, and pr2 reserved for spilling

Spill vr1

Spill vr2

Load vr2

index

	OPCODE	OP1			OP2			OP3		
		VR	PR	NU	VR	PR	NU	VR	PR	NU
0	loadI	0		∞			∞	vr0	pr0	2
1	loadI	4		∞			∞	vr1	pr1	2
2	add	vr0	pr0	3	vr1	pr1	7	vr2	pr1	3
3	add	vr0	pr0	4	vr2	pr1	5	vr3	pr1	4
4	store	vr3	pr1	∞			∞	vr0	pr0	5
5	store	vr2	pr1	∞			∞	vr0	pr0	7
6	output	0		∞			∞			∞
7	store	vr1		∞			∞	vr0		∞
8	output	0		∞			∞			∞

	vr0	vr1	vr2	vr3
VRToPR	pr0	—	—	—

	pr0	pr1	pr2
PRToVR	vr0	—	—
Next Use	7	∞	∞

Processing operation 7

- Op1 needs a register; allocate vr1 to pr1 (its NU is ∞) and load its value from the spill location

The Block

Assume that we have pr0 & pr1 for values, and pr2 reserved for spilling

Spill vr1

Spill vr2

Load vr2

Load vr1

index →

	OPCODE	OP1			OP2			OP3		
		VR	PR	NU	VR	PR	NU	VR	PR	NU
0	loadI	0		∞			∞	vr0	pr0	2
1	loadI	4		∞			∞	vr1	pr1	2
2	add	vr0	pr0	3	vr1	pr1	7	vr2	pr1	3
3	add	vr0	pr0	4	vr2	pr1	5	vr3	pr1	4
4	store	vr3	pr1	∞			∞	vr0	pr0	5
5	store	vr2	pr1	∞			∞	vr0	pr0	7
6	output	0		∞			∞			∞
7	store	vr1	pr1	∞			∞	vr0	pr0	∞
8	output	0		∞			∞			∞

	vr0	vr1	vr2	vr3
VRToPR	pr0	pr1	—	—

	pr0	pr1	pr2
PRToVR	vr0	vr1	—
Next Use	∞	∞	∞

Processing operation 7

- Op1 needs a register; allocate vr1 to pr1 (its NU is ∞) and load its value from the spill location
- Op3 is already in pr0

The Block

Assume that we have pr0 & pr1 for values, and pr2 reserved for spilling

Spill vr1

Spill vr2

Load vr2

Load vr1

	OPCODE	OP1			OP2			OP3		
		VR	PR	NU	VR	PR	NU	VR	PR	NU
0	loadI	0		∞			∞	vr0	pr0	2
1	loadI	4		∞			∞	vr1	pr1	2
2	add	vr0	pr0	3	vr1	pr1	7	vr2	pr1	3
3	add	vr0	pr0	4	vr2	pr1	5	vr3	pr1	4
4	store	vr3	pr1	∞			∞	vr0	pr0	5
5	store	vr2	pr1	∞			∞	vr0	pr0	7
6	output	0		∞			∞			∞
7	store	vr1	pr1	∞			∞	vr0	pr0	∞
8	output	0		∞			∞			∞

index →

	vr0	vr1	vr2	vr3
VRToPR	—	—	—	—

	pr0	pr1	pr2
PRToVR	—	—	—
Next Use	∞	∞	∞

Process operation 8

- "output" does not touch any registers
- Nothing to do here

Implementing the Spills

Let's examine each of the spills and restores

Spill vr1

Spill vr2

Load vr2

Load vr1

index →

	OPCODE	OP1			OP2			OP3		
		VR	PR	NU	VR	PR	NU	VR	PR	NU
0	loadI	0		∞			∞	vr0	pr0	2
1	loadI	4		∞			∞	vr1	pr1	2
2	add	vr0	pr0	3	vr1	pr1	7	vr2	pr1	3
3	add	vr0	pr0	4	vr2	pr1	5	vr3	pr1	4
4	store	vr3	pr1	∞			∞	vr0	pr0	5
5	store	vr2	pr1	∞			∞	vr0	pr0	7
6	output	0		∞			∞			∞
7	store	vr1	pr1	∞			∞	vr0	pr0	∞
8	output	0		∞			∞			∞

Consider the 1st spill

- Spill must preserve the value of vr1 for the later restore.
- Since vr1 is the result of a loadI, the restore can be a loadI; thus, the spill does not need any code.
- pr1 is reallocated in the middle of op 2, so the spill goes before op 2. The value of vr1 is still in pr1 for use in op 2, but when pr1 is redefined at the end of op 2, the value is safely in memory.

Implementing the Spills

Examine each of the spills and restores

Spill vr1

Spill vr2

Load vr2

Load vr1

index

	OPCODE	OP1			OP2			OP3		
		VR	PR	NU	VR	PR	NU	VR	PR	NU
0	loadI	0		∞			∞	vr0	pr0	2
1	loadI	4		∞			∞	vr1	pr1	2
2	add	vr0	pr0	3	vr1	pr1	7	vr2	pr1	3
3	add	vr0	pr0	4	vr2	pr1	5	vr3	pr1	4
4	store	vr3	pr1	∞			∞	vr0	pr0	5
5	store	vr2	pr1	∞			∞	vr0	pr0	7
6	output	0		∞			∞			∞
7	store	vr1	pr1	∞			∞	vr0	pr0	∞
8	output	0		∞			∞			∞

Consider the 2nd spill

- Spill must preserve the value of vr2 for the later restore.
- Since vr2 is the result of a computation, its value must be stored to memory
- Pick a spill location, say 32,768, and generate the store after op 2:

  loadI 32768 => pr2
  store pr1 => pr2

- After the store, pr1 is free for its new value

Implementing the Spills

Examine each of the spills and restores

Spill vr1

Spill vr2

Load vr2

Load vr1

	OPCODE	OP1			OP2			OP3		
		VR	PR	NU	VR	PR	NU	VR	PR	NU
0	loadI	0		∞			∞	vr0	pr0	2
1	loadI	4		∞			∞	vr1	pr1	2
2	add	vr0	pr0	3	vr1	pr1	7	vr2	pr1	3
3	add	vr0	pr0	4	vr2	pr1	5	vr3	pr1	4
4	store	vr3	pr1	∞			∞	vr0	pr0	5
5	store	vr2	pr1	∞			∞	vr0	pr0	7
6	output	0		∞			∞			∞
7	store	vr1	pr1	∞			∞	vr0	pr0	∞
8	output	0		∞			∞			∞

index →

Consider the 1st restore

- To restore vr2, load it from the location where the allocator stored it: 32768. Insert the load after op 4

  loadI 32768 => pr2
  load pr2 => pr1

- After the load, pr1 holds vr2 and is ready for use in operation 5

Implementing the Spills

Examine each of the spills and restores

Spill vr1

Spill vr2

Load vr2

Load vr1

index

	OPCODE	OP1			OP2			OP3		
		VR	PR	NU	VR	PR	NU	VR	PR	NU
0	loadI	0		∞			∞	vr0	pr0	2
1	loadI	4		∞			∞	vr1	pr1	2
2	add	vr0	pr0	3	vr1	pr1	7	vr2	pr1	3
3	add	vr0	pr0	4	vr2	pr1	5	vr3	pr1	4
4	store	vr3	pr1	∞			∞	vr0	pr0	5
5	store	vr2	pr1	∞			∞	vr0	pr0	7
6	output	0		∞			∞			∞
7	store	vr1	pr1	∞			∞	vr0	pr0	∞
8	output	0		∞			∞			∞

Consider the 2nd restore

- To restore vr1, the allocator can use a copy of loadI that created it — op 1

  loadI 4 => pr1

- After the load, pr1 holds vr1 and is ready for use in operation 7

The Final Code

	OPCODE	OP1			OP2			OP3		
		VR	PR	NU	VR	PR	NU	VR	PR	NU
0	loadI	0		∞			∞	vr0	pr0	2
1	loadI	4		∞			∞	vr1	pr1	2
2	add	vr0	pr0	3	vr1	pr1	7	vr2	pr1	3
–	loadI	128		∞			∞		pr2	∞
–	store		pr1	∞			∞		pr2	∞
3	add	vr0	pr0	4	vr2	pr1	5	vr3	pr1	4
4	store	vr3	pr1	∞			∞	vr0	pr0	5
–	loadI	128		∞			∞		pr2	∞
–	load	pr2		∞			∞		pr1	∞
5	store	vr2	pr1	∞			∞	vr0	pr0	7
6	output	0		∞			∞			∞
–	loadI	4		∞			∞		pr1	7
7	store	vr1	pr1	∞			∞	vr0	pr0	∞
8	output	0		∞			∞			∞

← Spill vr1; no code

← Spill vr2

← Restore vr2

← Restore vr1

Pro Tips for Lab 2

Manage your development process for self-protection

- Test your heuristic widely
  - Report blocks, plus one of the timing blocks (all have the same pattern)
  - Libraries of contributed blocks range from trivial to hard
  - Invent new test blocks & share them with your classmates

Pro Tips for Lab 2

Manage your development process for self-protection

- Test your heuristic widely
  - Report blocks, plus one of the timing blocks (all have the same pattern)
  - Libraries of contributed blocks range from trivial to hard
  - Invent new test blocks & share them with your classmates
- Checkpoint your allocator regularly as you change it
  - Checkpoint each working version
  - Keep them until after the code due date

Pro Tips for Lab 2

Manage your development process for self-protection

- Test your heuristic widely
  - Report blocks, plus one of the timing blocks (all have the same pattern)
  - Libraries of contributed blocks range from trivial to hard
  - Invent new test blocks & share them with your classmates
- Checkpoint your allocator regularly as you change the heuristics
  - Checkpoint each working version
  - Keep them until after the code due date
- Be careful when copying your data structures that hold instructions
  - Safer to just store a flag indicating a "rematerializable" value and the constant representing the value, then generate a loadI instruction

Pro Tips for Lab 2

Manage your development process for self-protection

- Test your heuristic widely
  - Report blocks, plus one of the timing blocks (all have the same pattern)
  - Libraries of contributed blocks range from trivial to hard
  - Invent new test blocks & share them with your classmates
- Checkpoint your allocator regularly as you change the heuristics
  - Checkpoint each working version
  - Keep them until after the code due date
- Be careful when copying your data structures that hold instructions
  - Safer to just store a flag indicating a "rematerializable" value and the constant representing the value, then generate a loadI instruction
- Post your questions to Piazza
  - The "community effect" is powerful