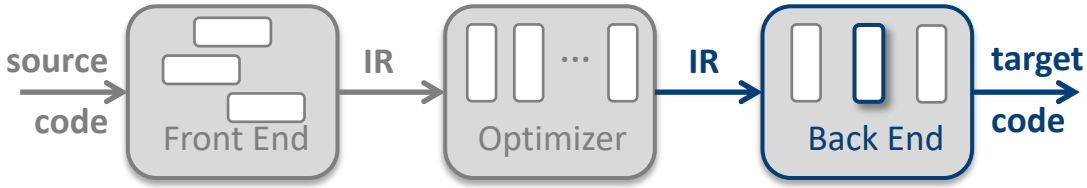
Updated After Tutorial

COMP 412
FALL 2018

Local Register Allocation

(critical content for Lab 2)

Comp 412



§ 13.1 & 13.2 in EaC2e

Lab 2 Schedule

N.B.: Code checks 1 & 2 are milestones, not exhaustive tests. Many students still find bugs after code check 2, to say nothing of tuning for effectiveness & efficiency.

Sunday	Monday	Tuesday	Wednesday	Thursday	Friday	Saturday
9	10	11	12	13	14 **Lab 2 Specs available**	15
16	17	18 **Tutorial 5 PM TBA**	19	20	21 **Deadline: Code Check 1**	22
23	24 **Tutorial 5 PM TBA**	25 **Dan Grove Talk (Dart Group)**	26	27	28	29
30	1 **Deadline: Code Check 2**	2	3	4	5	6
7	8 **Deadline: Lab 2 Code**	9	10	12 **Deadline: Lab Report**	12	13

Advice: Pay attention to the timing / scaling tests

Focus on connecting to Lab 1 IR, and Rename

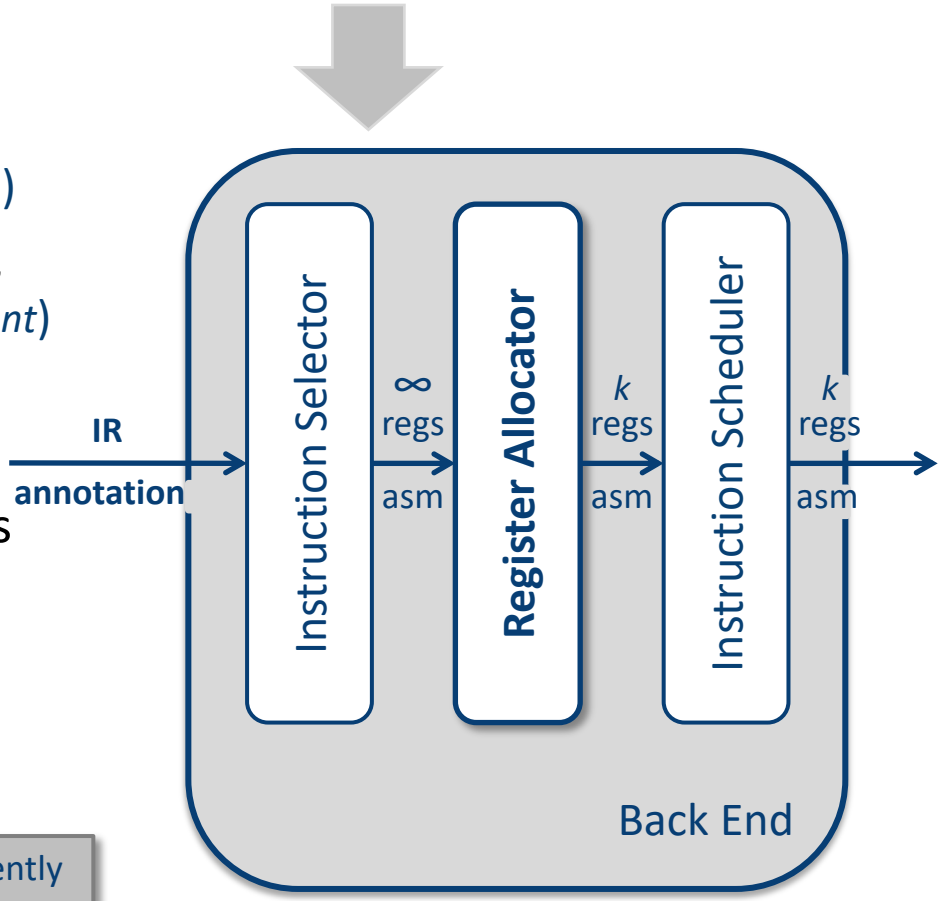
Allocate correctly

Improve performance & allocation

The Back End

Register Allocator:

- Decides, at each operation, which values reside in registers (*allocation*)
- Decides, for each **en**registered value, which register it occupies (*assignment*)
- Inserts code to "spill" **un**registered values to memory
- Tries to minimize the cost of spills
- Operates efficiently
  - **O**(n), **O**(n $\log_2$ n), maybe **O**(n^2)
  - Not **O**(2^n)



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

Local Register Allocation & Lab 2

In Lab 2, you will build a register allocator for straight-line ILOC code

- A sequence of branch-free (or straight-line) code is called a block
  - A **basic** block is a maximal length sequence of straight-line code
  - Algorithms that operate on one basic block are called **local** techniques

§ 8.3

- Your code will perform both *allocation* & *assignment* on a basic block
  - **Allocation** determines which values reside in registers at each operation
  - **Assignment** maps each value allocated a register into a specific register
  - In **local** allocation, we solve these problems together

§ 13

- This problem is *conceptually simple* and *combinatorially hard*
  - Realistic versions of the problem, including Lab 2, are **NP**-Complete
  - Your lab must find an approximate solution
  - Your lab should run in linear time (*even though the problem is **NPC***)
  - Welcome to the world of writing compilers!

The lab is designed to have a high ratio of thought to programming

You will reuse your ILOC front end from Lab 1 in both Lab 2 and Lab 3.

Terminology:

Values, Virtual Registers, & Physical Registers

A virtual register (VR) is a register name that the compiler uses in its internal representation of the code

- *Virtual* is used as in *virtual memory* or *virtual address*, not *virtual reality*
- Using virtual registers in the **IR**, rather than physical registers, lets the compiler defer allocation decisions & simplifies the compiler
  - *Correctly separates concerns over optimization and allocation* [*Backus*]
- A *physical register* (**PR**) is a name for an actual target machine register
  - *Limited supply of PRs, cannot add more*
  - *Valid assembly code cannot name more PRs than machine supports*
- Register allocators (& *many compilers*) use virtual registers to create a name space where each name corresponds to a *value* or *live range*
  - *The first thing your allocator should do is find live ranges and rename the code so each live range is a unique **VR**. The allocator then replaces **VR**s with **PR**s.*
  - *Using **VR**s that correspond to live ranges simplifies the implementation of the allocator and sidesteps some potential problems in the allocator.*

We use "**SR**" as an abbreviation for "source register"

Local Register Allocation

In $x \Leftarrow y$, x is a *definition* and y is a *use*. (standard terminology)

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 Lab 2 **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

† **If you find ILOC confusing, see lecture notes on the ILOC virtual machine.**

Local Register Allocation

In $x \Leftarrow y$, x is a *definition* and y is a *use*. (standard terminology)

A Critical Point

```
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
```

One clever strategy is to have the parser keep **op3** in the **op2** slot for **stores**, and teach the pretty printer to get it right on output.

Note Well:

- **store** is different than all other opcodes
- In a **store**, the second operand is a use, even though it appears to the right of **=>**
- The second operand is an address; the operation modifies **Mem(op2)**, not **op2.**
- Your lab needs to treat both operands as **uses**, not as **defs.**
- This will catch someone.

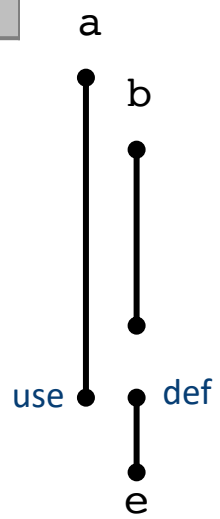
Finding Live Ranges

"Renaming" for Lab 2 Code Check

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)

#	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. An **ILOC** operation reads its operands before it writes its result

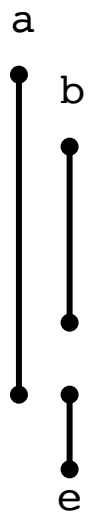
Finding Live Ranges

"Renaming" for Lab 2 Code Check

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

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



Observation

Let MAXLIVE be the largest number, over all instructions, of live values

- If MAXLIVE ≤ *k,* allocation is easy
- If MAXLIVE > k, some values must be spilled to memory
- To find live ranges & to compute MAXLIVE, we will compute a set LIVE(*i*) at each operation *i*

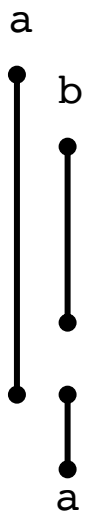
Finding Live Ranges

"Renaming" for Lab 2 Code Check

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

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



A Subtly Different Case

A single name can be part of multiple distinct live ranges.

- Changed **e** to **a**
- Operation 4:
  - → uses the value defined in op 0
  - → creates a new value used in op 5
  - → *kills* the value from operation 0
- Overwriting a value kills it

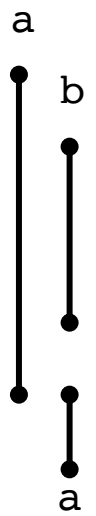
Finding Live Ranges

"Renaming" for Lab 2 Code Check

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

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



Why Use Live Ranges?

Consider an **SR** that comprises > 1 **VR**s

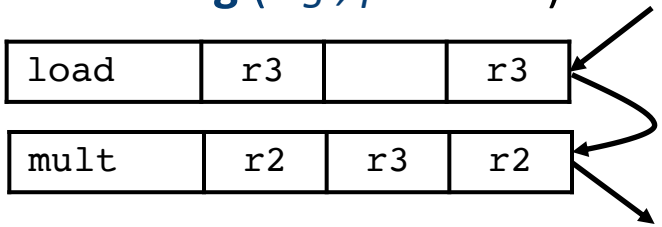
- The distinct **VR**s can be allocated to different **PR**s
  - Makes allocation easier
- Each **VR** now corresponds to a unique definition & value
  - Makes it easier to reason about spilled and restored values
- Simplifies writing and debugging a register allocator (*local or global*)

Representing **ILOC**

Why are the SRs in the abstract & concrete versions different?

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*)



When I implemented Lab 2, the most important design decision I made was to 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.

```
VRName ← 0              // start VR names at 0
for i ← 0 to max SR number
    SRToVR[i] ← invalid
    LU[i] ← ∞
for i ← n to 0 by -1
    if OPS[i].OP has a def then  // handle the def
      OPS[i].OP3.VR ← SRToVR[OPS[i].OP[3].SR]
      OPS[i].OP3.NU ← LU[OPS[i].OP3.SR]
      SRToVR[OPS[i].OP3.SR] ← invalid  // kill OP3
      LU[OPS[i].OP3.SR] ← ∞
    if OPS[i].OP1 is a register then  // handle uses
       Update(OPS[i].OP1,i)
    if OPS[i].OP2 is a register then
      Update(OPS[i].OP2,i)
    if  OPS[i].OP is STORE then  // STORE is special
      Update(OPS[i].OP3,i)

Update(OP, index) {
    if SRToVR[OP.SR] = invalid   // SR has no VR
        then SRToVR[OP.SR] ← VRName++
    OP.VR ← SRToVR[OP.SR]
    OP.NU ← LU[OP.SR]
    LU[OP.SR] ← index
}
```

(*Lab 2 Code Check 1*)

Finding live ranges, next uses, and renaming to VRs

- Algorithm walks 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

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

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

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 <u>has no def</u>

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	2	–	3	–	–
LU	8	7	∞	7	∞	∞

VRName: 4

Update r1 as a def & kill it

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	–	4	–	5
LU	8	7	∞	6	∞	6

VRName: 6

Update r3 as a def & kill it

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 & kill it

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	6	–
LU	8	7	∞	6	5	∞

VRName: 7

Update r4 as a def & kill it

No uses

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 & kill it

Update r2 as a use

Back to the Example

After Processing Operation 2

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	–	∞	∞

index →

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

VRName: 8 ***Update r2 as a def & kill it***

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 & kill it

Update r0 as a use

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 & kill it***

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 5*)

Back to the Example

Code renamed so that virtual registers correspond to live ranges

Operation			Live Ranges VR0	VR1	VR2	VR3	VR4	VR5	VR6	VR7	# LIVE @ 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 code check 1, your lab must rename registers & print out the renamed ILOC. The code must be executable by the ILOC simulator (e.g., register names "r1")

Back to the Example

Gray dashed lines show points where demand for registers is greatest.

Code renamed so that virtual registers correspond to live ranges

Operation			Live Ranges VR0	VR1	VR2	VR3	VR4	VR5	VR6	VR7	# LIVE @ 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									0

Live at the end of the operation

MaxLive

Lab 2 Code Check 1

Code Check 1 tests to see if your allocator does renaming properly

Original Code		
loadI	128	=> r0
load	r0	=> r1
loadI	132	=> r2
load	r2	=> r3
loadI	136	=> r4
load	r4	=> r5
mult	r3,r5	=> r3
add	r1,r3	=> r1
store	r1	=> r0

Code Check Output		
loadI	128	=> r1
load	r1	=> r2
loadI	132	=> r7
load	r7	=> r4
loadI	136	=> r6
load	r6	=> r5
mult	r4,r5	=> r3
add	r2,r3	=> r0
store	r0	=> r1

- The output code defines each register exactly once.
- It needs to label registers as “r0”, “r1”, …, so that the simulator can execute the code.

The Value of Renaming

Local Register Allocation

→ An example that came to my attention in 2009 …

$a_0 \leftarrow a_1 + a_2$
$a_1 \leftarrow a_2 + a_0$
$a_2 \leftarrow a_0 + a_1$
$a_3 \leftarrow a_1 + a_2$
$a_4 \leftarrow a_2 + a_3$
$a_5 \leftarrow a_3 + a_4$
•••

This block does a series of additions, in a specific pattern that creates a known and controlled demand for registers.

The example needs three registers. By increasing the distance between the definition of a value & its last reuse, we can arbitrarily increase the demand for registers.

Why would we build this code?

- Constructing a microbenchmark to measure the number of registers available to the compiled code — in essence, the compiler's effectiveness at register allocation

The Value of Renaming

gcc 4.1, -O3, PowerPC G4 target

Local Register Allocation

→ An example that came to my attention in 2009 …

$a_0 \leftarrow a_1 + a_2$
$a_1 \leftarrow a_2 + a_0$
$a_2 \leftarrow a_0 + a_1$
$a_3 \leftarrow a_1 + a_2$
$a_4 \leftarrow a_2 + a_3$
$a_5 \leftarrow a_3 + a_4$
…

Without reuse of names, gcc handles 30 values with no spills.

$a_0 \leftarrow a_1 + a_2$
$a_1 \leftarrow a_2 + a_0$
$a_2 \leftarrow a_0 + a_1$
$a_0 \leftarrow a_1 + a_2$
$a_1 \leftarrow a_2 + a_0$
$a_2 \leftarrow a_0 + a_1$
…

3-value pattern

If names are reused, gcc spills on patterns with > 20 values.

The use of names, a code shape issue, makes a fundamental difference in the quality of code that gcc generates for this pattern.

llvm/clang handles 30 values either way.

The point of this example is that the allocator confused the distinct live ranges with the same name (e.g., a_0) with the result that it saw the problem as more constrained that it should have been. As a result it began spilling when it had at least 10 spare registers.

Renaming the "virtual registers" so that each distinct definition targets a unique name would avoid this problem.

In lab one, most 412 students discover that they should immediately rename each "live range" to a unique name because it simplifies the rest of the code. (**HINT**)

$a_3 \leftarrow a_1 + a_2$
$a_4 \leftarrow a_2 + a_3$
$a_5 \leftarrow a_3 + a_4$
...

Without reuse of names, gcc handles 30 values with no spills.

$a_0 \leftarrow a_1 + a_2$
$a_1 \leftarrow a_2 + a_0$
$a_2 \leftarrow a_0 + a_1$
...

If names are reused, gcc spills on patterns with > 20 values.

The use of names, a code shape issue, makes a fundamental difference in the quality of code that gcc generates for this pattern.

Bottom-up Allocator

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

```
for i ← 0 to n
    if (OP[i].OP1.VR has no PR)
       get a PR, say x, load OP[i].OP1.VR into x, and set OP[i].OP1.PR ← x
    if (OP[i].OP2.VR has no PR)
       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
```

Consult the *VRToPR* map

y ≠ x unless OP1.VR = OP2.VR

if x = y, do not free the PR twice

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

OP_1 & OP_2 can have the same PR, if & only if they have the same SR

Bottom-up Allocator

"Get a PR"

- If some **PR**, say **prx**, is available, can use **prx** to hold the **VR** and either
  - **Restore** the value from memory (for a *use*), or
  - Use **prx** as the target (result) register (for a *def*)
- If all **PR**'s are in use, the bottom-up allocator must free one
  - Select the **PR** whose next use is farthest in the future, say **pry**
  - **Spill** the current contents of **pry** to memory (into its "spill location")
  - Use **pry** to hold the **VR**, as discussed above
    - → For a *use*, **restore** the value into **pry**; for a *def*, **pry** becomes the target register

Complications will arise

- Breaking a tie: what happens if two **VR**s both have the farthest next use?
- Dirty versus clean values: what happens if farthest **VR** needs a store and the second farthest one does not?

If the allocator needs to spill, it will need a register for the spill address. The simple way to handle that need is to reserve one register to hold the spill address, if **MAXLIVE** > k.

This issue is a property of the **ILOC** subset, not a general truth.

Back to the Example

k = 4

Bottom-up Allocator on the Example

OPCODE	OP1			OP2			OP3		
	VR	PR	NU	VR	PR	NU	VR	PR	NU
loadI	128		∞				vr1		1
load	vr1		8				vr2		7
loadI	132		∞				vr7		3
load	vr7		∞				vr4		6
loadI	136		∞				vr6		5
load	vr6		∞				vr5		6
mult	vr4		∞	vr5		∞	vr3		7
add	vr2		∞	vr3		∞	vr0		8
store	vr0		∞				vr1		∞

Assume that ***k*** = 4.

Since MAXLIVE = 4 and ***k*** = 4, we would expect the allocator to keep all values in registers and avoid spilling.

The allocator does not need to reserve a register for address computations on spilled values.

Example

k = 4

Initial State

index →

OPCODE	OP1			OP2			OP3		
	VR	PR	NU	VR	PR	NU	VR	PR	NU
loadI	128		∞				vr1		1
load	vr1		8				vr2		7
loadI	132		∞				vr7		3
load	vr7		∞				vr4		6
loadI	136		∞				vr6		5
load	vr6		∞				vr5		6
mult	vr4		∞	vr5		∞	vr3		7
add	vr2		∞	vr3		∞	vr0		8
store	vr0		∞				vr1		∞

index tracks "i"

	vr0	vr1	vr2	vr3	vr4	vr5	vr6	vr7
VRToPR	—	—	—	—	—	—	—	—

	pr0	pr1	pr2	pr3
PRToVR	—	—	—	—

Example

k = 4

Operation 0

index →

OPCODE	OP1			OP2			OP3		
	VR	PR	NU	VR	PR	NU	VR	PR	NU
loadI	128		∞				vr1	**pr0**	1
load	vr1		8				vr2		7
loadI	132		∞				vr7		3
load	vr7		∞				vr4		6
loadI	136		∞				vr6		5
load	vr6		∞				vr5		6
mult	vr4		∞	vr5		∞	vr3		7
add	vr2		∞	vr3		∞	vr0		8
store	vr0		∞				vr1		∞

OP1 needs no **PR**
OP2 needs no **PR**

OP3 needs a **PR**
— allocate **pr0** to **vr1**

	vr0	vr1	vr2	vr3	vr4	vr5	vr6	vr7
VRToPR	—	**pr0**	—	—	—	—	—	—

	pr0	pr1	pr2	pr3
PRToVR	**vr1**	—	—	—

Example

k = 4

Operation 1

	OPCODE	OP1			OP2			OP3		
		VR	PR	NU	VR	PR	NU	VR	PR	NU
	loadI	128		∞				vr1	pr0	1
index →	load	vr1	**pr0**	8				vr2	**pr1**	7
	loadI	132		∞				vr7		3
	load	vr7		∞				vr4		6
	loadI	136		∞				vr6		5
	load	vr6		∞				vr5		6
	mult	vr4		∞	vr5		∞	vr3		7
	add	vr2		∞	vr3		∞	vr0		8
	store	vr0		∞				vr1		∞

OP1 (**vr1**) is in **pr0**
OP2 needs no **PR**

OP3 needs a **PR**
— allocate **pr1** to **vr2**

	vr0	vr1	vr2	vr3	vr4	vr5	vr6	vr7
VRToPR	—	**pr0**	**pr1**	—	—	—	—	—

	pr0	pr1	pr2	pr3
PRToVR	**vr1**	**vr2**	—	—

Example

k = 4

Operation 2

OPCODE	OP1			OP2			OP3		
	VR	PR	NU	VR	PR	NU	VR	PR	NU
loadI	128		∞				vr1	pr0	1
load	vr1	pr0	8				vr2	pr1	7
loadI	132		∞				vr7	**pr2**	3
load	vr7		∞				vr4		6
loadI	136		∞				vr6		5
load	vr6		∞				vr5		6
mult	vr4		∞	vr5		∞	vr3		7
add	vr2		∞	vr3		∞	vr0		8
store	vr0		∞				vr1		∞

index →

OP1 needs no **PR**
OP2 needs no **PR**

OP3 needs a **PR**
— allocate **pr2** to **vr7**

	vr0	vr1	vr2	vr3	vr4	vr5	vr6	vr7
VRToPR	—	**pr0**	**pr1**	—	—	—	—	**pr2**

	pr0	pr1	pr2	pr3
PRToVR	**vr1**	**vr2**	**vr7**	—

Example

If the allocator uses a **LIFO** discipline with names, it will produce this pattern of reuse within the operation.

Operation 3

OPCODE	OP1			OP2			OP3		
	VR	PR	NU	VR	PR	NU	VR	PR	NU
loadI	128		∞				vr1	pr0	1
load	vr1	pr0	8				vr2	pr1	7
loadI	132		∞				vr7	pr2	3
load	vr7	**pr2**					vr4	**pr2**	1
loadI	136		∞				vr6		5
load	vr6		∞				vr5		6
mult	vr4		∞	vr5		∞	vr3		7
add	vr2		∞	vr3		∞	vr0		8
store	vr0		∞				vr1		∞

index →

OP1 (**vr7**) is in **pr2**
OP2 needs no **PR**
vr7 is dead, so free **pr2**

OP3 needs a **PR**
— allocate **pr2** to **vr4**

	vr0	vr1	vr2	vr3	vr4	vr5	vr6	vr7
VRToPR	—	**pr0**	**pr1**	—	**pr2**	—	—	pr2
					@ end of op			@ start of op

	pr0	pr1	pr2	pr3
PRToVR	**vr1**	**vr2**	**vr4**	—

Example

k = 4

Operation 4

OPCODE	OP1			OP2			OP3		
	VR	PR	NU	VR	PR	NU	VR	PR	NU
loadI	128		∞				vr1	pr0	1
load	vr1	pr0	8				vr2	pr1	7
loadI	132		∞				vr7	pr2	3
load	vr7	pr2	∞				vr4	pr2	6
loadI	136		∞				vr6	**pr3**	5
load	vr6		∞				vr5		6
mult	vr4		∞	vr5		∞	vr3		7
add	vr2		∞	vr3		∞	vr0		8
store	vr0		∞				vr1		∞

index →

OP1 needs no **PR**
OP2 needs no **PR**

OP3 needs a **PR**
— allocate **pr3** to **vr6**

	vr0	vr1	vr2	vr3	vr4	vr5	vr6	vr7
VRToPR	—	pr0	pr1	—	pr2	—	pr3	—

	pr0	pr1	pr2	pr3
PRToVR	vr1	vr2	vr4	vr6

Example

k = 4

Operation 5

OPCODE	OP1			OP2			OP3		
	VR	PR	NU	VR	PR	NU	VR	PR	NU
loadI	128		∞				vr1	pr0	1
load	vr1	pr0	8				vr2	pr1	7
loadI	132		∞				vr7	pr2	3
load	vr7	pr2	∞				vr4	pr2	6
loadI	136		∞				vr6	pr3	5
load	vr6	**pr3**	∞				vr5	**pr3**	6
mult	vr4		∞	vr5		∞	vr3		7
add	vr2		∞	vr3		∞	vr0		8
store	vr0		∞				vr1		∞

index →

OP1 (**vr6**) in in **pr3**
OP2 needs no **PR**

vr6 is dead, so free **pr3**

OP3 needs a **PR**
— allocate **pr3** to **vr5**

	vr0	vr1	vr2	vr3	vr4	vr5	vr6	vr7
VRToPR	—	pr0	pr1	—	pr2	pr3	pr3	—

	pr0	pr1	pr2	pr3
PRToVR	vr1	vr2	vr4	vr5

Example

k = 4

Operation 6

OPCODE	OP1			OP2			OP3		
	VR	PR	NU	VR	PR	NU	VR	PR	NU
loadI	128		∞				vr1	pr0	1
load	vr1	pr0	8				vr2	pr1	7
loadI	132		∞				vr7	pr2	3
load	vr7	pr2	∞				vr4	pr2	6
loadI	136		∞				vr6	pr3	5
load	vr6	pr3	∞				vr5	pr3	6
mult	vr4	**pr2**	∞	vr5	**pr3**	∞	vr3	**pr2**	7
add	vr2		∞	vr3		∞	vr0		8
store	vr0		∞				vr1		∞

index →

OP1 (**vr4**) in in **pr2**
OP2 (**vr5**) is in **pr3**

Both **vr4** & **vr5** are dead, so free **pr2** & **pr3**

OP3 needs a **PR**
— allocate **pr2** to **vr5**

	vr0	vr1	vr2	vr3	vr4	vr5	vr6	vr7
VRToPR	—	pr0	pr1	pr2	pr2	pr3	—	—

	pr0	pr1	pr2	pr3
PRToVR	vr1	vr2	vr3	—

Example

k = 4

Operation 7

OPCODE	OP1			OP2			OP3		
	VR	PR	NU	VR	PR	NU	VR	PR	NU
loadI	128		∞				vr1	pr0	1
load	vr1	pr0	8				vr2	pr1	7
loadI	132		∞				vr7	pr2	3
load	vr7	pr2	∞				vr4	pr2	6
loadI	136		∞				vr6	pr3	5
load	vr6	pr3	∞				vr5	pr3	6
mult	vr4	pr2	∞	vr5	pr3	∞	vr3	pr2	7
add	vr2	**pr1**	∞	vr3	**pr2**	∞	vr0	**pr1**	8
store	vr0		∞				vr1		∞

index →

OP1 (**vr2**) in in **pr1**
OP2 (**vr3**) is in **pr2**

Both **vr2** & **vr3** are dead, so free **pr1** & **pr2**

OP3 needs a **PR**
— allocate **pr1** to **vr0**

	vr0	vr1	vr2	vr3	vr4	vr5	vr6	vr7
VRToPR	**pr1**	**pr0**	pr1	pr2	—	—	—	—

	pr0	pr1	pr2	pr3
PRToVR	**vr1**	**vr0**	—	—

Example

k = 4

Operation 8

OPCODE	OP1			OP2			OP3		
	VR	PR	NU	VR	PR	NU	VR	PR	NU
loadI	128		∞				vr1	pr0	1
load	vr1	pr0	8				vr2	pr1	7
loadI	132		∞				vr7	pr2	3
load	vr7	pr2	∞				vr4	pr2	6
loadI	136		∞				vr6	pr3	5
load	vr6	pr3	∞				vr5	pr3	6
mult	vr4	pr2	∞	vr5	pr3	∞	vr3	pr2	7
add	vr2	pr1	∞	vr3	pr2	∞	vr0	pr1	8
store	vr0	**pr1**	∞				vr1	**pr0**	∞

index →

OP1 (**vr0**) in in **pr1**
OP2 needs no **PR**
OP3 is a use (**vr1**) that is in **pr0**

Both **vr0** & **vr1** are dead, so free **pr0** & **pr1**

	vr0	vr1	vr2	vr3	vr4	vr5	vr6	vr7
VRToPR	pr1	pr0	—	—	—	—	—	—

	pr0	pr1	pr2	pr3
PRToVR	—	—	—	—

Example

k = 4

Final Code

OPCODE	OP1			OP2			OP3		
	VR	PR	NU	VR	PR	NU	VR	PR	NU
loadI	128	128	∞				vr1	pr0	1
load	vr1	pr0	8				vr2	pr1	7
loadI	132	132	∞				vr7	pr2	3
load	vr7	pr2	∞				vr4	pr2	6
loadI	136	136	∞				vr6	pr3	5
load	vr6	pr3	∞				vr5	pr3	6
mult	vr4	pr2	∞	vr5	pr3	∞	vr3	pr2	7
add	vr2	pr1	∞	vr3	pr2	∞	vr0	pr1	8
store	vr0	pr1	∞				vr1	pr0	∞

As expected, the code allocates easily into 4 physical registers.

Example with a Reserved Spill Register

Local Allocator, Same Example, 1 Register Reserved for Spills

OPCODE	OP1			OP2			OP3		
	VR	PR	NU	VR	PR	NU	VR	PR	NU
loadI	128		∞				vr1		1
load	vr1		8				vr2		7
loadI	132		∞				vr7		3
load	vr7		∞				vr4		6
loadI	136		∞				vr6		5
load	vr6		∞				vr5		6
mult	vr4		∞	vr5		∞	vr3		7
add	vr2		∞	vr3		∞	vr0		8
store	vr0		∞				vr1		∞

Now, lets look at what happens if we reserve a register for spilling, so the allocator has only **3** registers to hold **VR**s.

Example

k = 4, with 1 register reserved for spilling

Initial State

index →

OPCODE	OP1			OP2			OP3		
	VR	PR	NU	VR	PR	NU	VR	PR	NU
loadI	128		∞				vr1		1
load	vr1		8				vr2		7
loadI	132		∞				vr7		3
load	vr7		∞				vr4		6
loadI	136		∞				vr6		5
load	vr6		∞				vr5		6
mult	vr4		∞	vr5		∞	vr3		7
add	vr2		∞	vr3		∞	vr0		8
store	vr0		∞				vr1		∞

index tracks "i"

	vr0	vr1	vr2	vr3	vr4	vr5	vr6	vr7
VRToPR	—	—	—	—	—	—	—	—

	pr0	pr1	pr2
PRToVR	—	—	—

Example

k = 4, with 1 register reserved for spilling

Operation 0

	OPCODE	OP1			OP2			OP3		
		VR	PR	NU	VR	PR	NU	VR	PR	NU
index →	loadI	128		∞				vr1	**pr0**	1
	load	vr1		8				vr2		7
	loadI	132		∞				vr7		3
	load	vr7		∞				vr4		6
	loadI	136		∞				vr6		5
	load	vr6		∞				vr5		6
	mult	vr4		∞	vr5		∞	vr3		7
	add	vr2		∞	vr3		∞	vr0		8
	store	vr0		∞				vr1		∞

OP1 needs no **PR**
OP2 needs no **PR**

OP3 needs a **PR**
— allocate **pr0** to **vr1**

	vr0	vr1	vr2	vr3	vr4	vr5	vr6	vr7
VRToPR	—	**pr0**	—	—	—	—	—	—

	pr0	pr1	pr2
PRToVR	**vr1**	—	—

Example

k = 4, with 1 register reserved for spilling

Operation 1

	OPCODE	OP1			OP2			OP3		
		VR	PR	NU	VR	PR	NU	VR	PR	NU
	loadI	128		∞				vr1	pr0	1
index →	load	vr1	**pr0**	8				vr2	**pr1**	7
	loadI	132		∞				vr7		3
	load	vr7		∞				vr4		6
	loadI	136		∞				vr6		5
	load	vr6		∞				vr5		6
	mult	vr4		∞	vr5		∞	vr3		7
	add	vr2		∞	vr3		∞	vr0		8
	store	vr0		∞				vr1		∞

OP1 (**vr1**) is in **pr0**
OP2 needs no **PR**
OP3 needs a **PR**
— allocate **pr1** to **vr2**

	vr0	vr1	vr2	vr3	vr4	vr5	vr6	vr7
VRToPR	—	**pr0**	**pr1**	—	—	—	—	—

	pr0	pr1	pr2
PRToVR	vr1	vr2	—

Example

k = 4, with 1 register reserved for spilling

Operation 2

OPCODE	OP1			OP2			OP3		
	VR	PR	NU	VR	PR	NU	VR	PR	NU
loadI	128		∞				vr1	pr0	1
load	vr1	pr0	8				vr2	pr1	7
loadI	132		∞				vr7	**pr2**	3
load	vr7		∞				vr4		6
loadI	136		∞				vr6		5
load	vr6		∞				vr5		6
mult	vr4		∞	vr5		∞	vr3		7
add	vr2		∞	vr3		∞	vr0		8
store	vr0		∞				vr1		∞

index →

OP1 needs no **PR**
OP2 needs no **PR**

OP3 needs a **PR**
— allocate **pr2** to **vr7**

So far, no difference

	vr0	vr1	vr2	vr3	vr4	vr5	vr6	vr7
VRToPR	—	**pr0**	**pr1**	—	—	—	—	**pr2**

	pr0	pr1	pr2
PRToVR	**vr1**	**vr2**	**vr7**

Example

k = 4, with 1 register reserved for spilling

Operation 3

OPCODE	OP1			OP2			OP3		
	VR	PR	NU	VR	PR	NU	VR	PR	NU
loadI	128		∞				vr1	pr0	1
load	vr1	pr0	8				vr2	pr1	7
loadI	132		∞				vr7	pr2	3
load	vr7	**pr2**	∞				vr4	**pr2**	6
loadI	136		∞				vr6		5
load	vr6		∞				vr5		6
mult	vr4		∞	vr5		∞	vr3		7
add	vr2		∞	vr3		∞	vr0		8
store	vr0		∞				vr1		∞

index →

OP1 (**vr7**) is in **pr2**
OP2 needs no **PR**
vr7 is dead, so free **pr2**

OP3 needs a **PR**
— allocate **pr2** to **vr4**

Works just as before

	vr0	vr1	vr2	vr3	vr4	vr5	vr6	vr7
VRToPR	—	**pr0**	**pr1**	—	**pr2**	—	—	pr2

	pr0	pr1	pr2
PRToVR	**vr1**	**vr2**	**vr4**

Example

k = 4, with 1 register reserved for spilling

Operation 4

OPCODE	OP1			OP2			OP3		
	VR	PR	NU	VR	PR	NU	VR	PR	NU
loadI	128		∞				vr1	pr0	1
load	vr1	pr0	8				vr2	pr1	7
loadI	132		∞				vr7	pr2	3
load	vr7	pr2	∞				vr4	pr2	6
loadI	136		∞				vr6		5
load	vr6		∞				vr5		6
mult	vr4		∞	vr5		∞	vr3		7
add	vr2		∞	vr3		∞	vr0		8
store	vr0		∞				vr1		∞

index →

OP1 needs no **PR**
OP2 needs no **PR**

OP3 needs a **PR**
— none is available
— choose one to *spill* *

	vr0	vr1	vr2	vr3	vr4	vr5	vr6	vr7
VRToPR	—	**pr0**	**pr1**	—	**pr2**	—	—	—

	pr0	pr1	pr2
PRToVR	**vr1**	**vr2**	**vr4**

Example

k = 4, with 1 register reserved for spilling

Operation 4

OPCODE	OP1			OP2			OP3		
	VR	PR	NU	VR	PR	NU	VR	PR	NU
loadI	128		∞				vr1	pr0	1
load	vr1	pr0	8				vr2	pr1	7
loadI	132		∞				vr7	pr2	3
load	vr7	pr2	∞				vr4	pr2	6
loadI	136		∞				vr6		5
load	vr6		∞				vr5		6
mult	vr4		∞	vr5		∞	vr3		7
add	vr2		∞	vr3		∞	vr0		8
store	vr0		∞				vr1		∞

spill vr1

OP1 needs no **PR**
OP2 needs no **PR**

OP3 needs a **PR**
- none is available
- choose one to *spill*
- Algorithm says to pick the **VR** with the farthest NextUse
- Spill **vr1** to free **pr0** before operation 4

	vr0	vr1	vr2	vr3	vr4	vr5	vr6	vr7
VRToPR	—	**pr0**	**pr1**	—	**pr2**	—	—	—

	pr0	pr1	pr2
PRToVR	**vr1**	**vr2**	**vr4**
NextUse	8	7	6

Example

k = 4, with 1 register reserved for spilling

Operation 4

OPCODE	OP1			OP2			OP3		
	VR	PR	NU	VR	PR	NU	VR	PR	NU
loadI	128		∞				vr1	pr0	1
load	vr1	pr0	8				vr2	pr1	7
loadI	132		∞				vr7	pr2	3
load	vr7	pr2	∞				vr4	pr2	6
loadI	136		∞				vr6	**pr0**	5
load	vr6		∞				vr5		6
mult	vr4		∞	vr5		∞	vr3		7
add	vr2		∞	vr3		∞	vr0		8
store	vr0		∞				vr1		∞

spill vr1

OP1 needs no **PR**
OP2 needs no **PR**

OP3 needs a **PR**
- none is available
- choose one to *spill*
- Algorithm says to pick the **PR** with the farthest NextUse
- Spill **vr1** to free **pr0** before operation 4
- Allocate **pr0** to **vr6**

	vr0	vr1	vr2	vr3	vr4	vr5	vr6	vr7
VRToPR	—	pr0	**pr1**	—	**pr2**	—	**pr0**	—

	pr0	pr1	pr2
PRToVR	**vr6**	**vr2**	**vr4**
NextUse	5	7	6

Example

k = 4, with 1 register reserved for spilling

Operation 5

OPCODE	OP1			OP2			OP3		
	VR	PR	NU	VR	PR	NU	VR	PR	NU
loadI	128		∞				vr1	pr0	1
load	vr1	pr0	8				vr2	pr1	7
loadI	132		∞				vr7	pr2	3
load	vr7	pr2	∞				vr4	pr2	6
loadI	136		∞				vr6	pr0	5
load	vr6	**pr0**	∞				vr5	**pr0**	6
mult	vr4		∞	vr5		∞	vr3		7
add	vr2		∞	vr3		∞	vr0		8
store	vr0		∞				vr1		∞

spill vr1

index

OP1 (**vr6**) is in **pr0**
OP2 needs no **PR**

vr6 is dead, so free **pr0**

OP3 needs a **PR**
— Allocate **pr0** to **vr5**

	vr0	vr1	vr2	vr3	vr4	vr5	vr6	vr7
VRToPR	—	—	**pr1**	—	**pr2**	**pr0**	pr0	—

	pr0	pr1	pr2
PRToVR	**vr6**	**vr2**	**vr4**
NextUse	6	7	6

Example

k = 4, with 1 register reserved for spilling

Operation 6

OPCODE	OP1			OP2			OP3		
	VR	PR	NU	VR	PR	NU	VR	PR	NU
loadI	128		∞				vr1	pr0	1
load	vr1	pr0	8				vr2	pr1	7
loadI	132		∞				vr7	pr2	3
load	vr7	pr2	∞				vr4	pr2	6
loadI	136		∞				vr6	pr0	5
load	vr6	pr0	∞				vr5	pr0	6
mult	vr4	**pr2**	∞	vr5	**pr0**	∞	vr3	**pr0**	7
add	vr2		∞	vr3		∞	vr0		8
store	vr0		∞				vr1		∞

spill vr1

index

OP1 (**vr4**) is in **pr2**
OP2 (**vr5**) is in **pr0**

Both **vr4** & **vr5** are dead, so free **pr2** & **pr0**

OP3 needs a **PR**
— Allocate **pr0** to **vr3**

	vr0	vr1	vr2	vr3	vr4	vr5	vr6	vr7
VRToPR	—	—	**pr1**	**pr0**	pr2	pr0	—	—

	pr0	pr1	pr2
PRToVR	**vr3**	**vr2**	—
NextUse	7	7	∞

Example

k = 4, with 1 register reserved for spilling

Operation 7

OPCODE	OP1			OP2			OP3		
	VR	PR	NU	VR	PR	NU	VR	PR	NU
loadI	128		∞				vr1	pr0	1
load	vr1	pr0	8				vr2	pr1	7
loadI	132		∞				vr7	pr2	3
load	vr7	pr2	∞				vr4	pr2	6
loadI	136		∞				vr6	pr0	5
load	vr6	pr0	∞				vr5	pr0	6
mult	vr4	pr2	∞	vr5	pr0	∞	vr3	pr0	7
add	vr2	**pr1**	∞	vr3	**pr0**	∞	vr0	**pr0**	8
store	vr0		∞				vr1		∞

spill vr1

index

OP1 (**vr2**) is in **pr1**
OP2 (**vr3**) is in **pr0**

Both **vr2** & **vr3** are dead, so free **pr1** & **pr0**

OP3 needs a **PR**
— Allocate **pr0** to **vr0**

	vr0	vr1	vr2	vr3	vr4	vr5	vr6	vr7
VRToPR	**pr0**	—	pr1	pr0	—	—	—	—

	pr0	pr1	pr2
PRToVR	**vr0**	—	—
NextUse	8	∞	∞

Example

k = 4, with 1 register reserved for spilling

Operation 8

OPCODE	OP1			OP2			OP3		
	VR	PR	NU	VR	PR	NU	VR	PR	NU
loadI	128		∞				vr1	pr0	1
load	vr1	pr0	8				vr2	pr1	7
loadI	132		∞				vr7	pr2	3
load	vr7	pr2	∞				vr4	pr2	6
loadI	136		∞				vr6	pr0	5
load	vr6	pr0	∞				vr5	pr0	6
mult	vr4	pr2	∞	vr5	pr0	∞	vr3	pr0	7
add	vr2	pr1	∞	vr3	pr0	∞	vr0	pr0	8
store	vr0	**pr0**	∞				vr1	**pr1**	∞

spill vr1

restore vr1

index

OP1 (**vr0**) is in **pr0**
OP2 needs no **PR**
OP3 (**vr1**) is a use, since the operation is a `store`
- **vr1** was spilled, so we need to allocate a register & restore it
- allocate **pr1** to **vr1**
- Insert a "restore"

Both **vr0** & **vr1** are dead, so free **pr1** & **pr0**

OP3 needs a **PR**

	vr0	vr1	vr2	vr3	vr4	vr5	vr6	vr7
VRToPR	pr0	pr1	—	—	—	—	—	—

	pr0	pr1	pr2
PRToVR	vr0	vr1	—
NextUse	∞	∞	∞

"Spill" and "Restore"

In general, if the allocator needs to spill and restore values, it will need to reserve a register for the address of the spill location

- Reduces the number of available registers by one
  - In example, would have only had **pr1** and **pr2**, not **pr0**
  - Can check this case by keeping track of MAXLIVE during renaming and comparing it to ***k***
- Spill has two real cases:
  - If the value was computed or is the result of a **load**, it must be stored into a "spill location"
    → Allocate a memory location to hold it and generate the store to spill it
  - If the value is the result of a **loadI,** the allocator can restore it without a spill
    → In the example, **vr1** is the result of a **loadI**
    → It can be restored with an identical **loadI**, so the spill needs no code
    → The restore will consist of a copy of the original **loadI**

"Spill" and "Restore"

In general, if the allocator needs to spill and restore values, it will need to reserve a register for the address of the spill location

- Reduces the number of available registers by one
  - In example, would have only had **pr1** and **pr2**, not **pr0**
  - Can check this case by keeping track of MAXLIVE during renaming and comparing it to ***k***
- Similarly, restore has two real cases:
  - If the value was spilled to memory
    → Compute the address of the spill location (a **loadI**) and load from it
  - If the value did not need a spill
    → Restore the value using the **loadI**
    ("*rematerialization*")
- Of course, if all spills are rematerialized, the allocator did not need to reserve a register — it gets complex
  - Remember, the problem *is* NP-Complete

Example with the Spill Code

k = 4, with 1 register reserved for spilling

Final Code

OPCODE	OP1			OP2			OP3		
	VR	PR	NU	VR	PR	NU	VR	PR	NU
loadI	128	128	∞				vr1	pr0	1
load	vr1	pr0	8				vr2	pr1	7
loadI	132	132	∞				vr7	pr2	3
load	vr7	pr2	∞				vr4	pr2	6
loadI	136	136	∞				vr6	pr0	5
load	vr6	pr0	∞				vr5	pr0	6
mult	vr4	pr2	∞	vr5	pr0	∞	vr3	pr0	7
add	vr2	pr1	∞	vr3	pr0	∞	vr0	pr0	8
loadI	**128**	**128**	**∞**				**vr1**	**pr1**	**8**
store	vr0	pr0	∞				vr1	pr1	∞

vr1 is rematerializable

"spill" requires no code

"restore" becomes a duplicate of operation 0

Accuracy doesn't matter on the *NextUse*; relative magnitude does.

We can treat the "restore" operation as if it were 7.5

“Spill” and “Restore”

In the example, the allocator spilled vr1 before operation 4. Should it have spilled vr1 earlier?

- It would have been legal to spill it after its use in operation 1
  - No further use until operation 8
- In this case, it would not have gained us anything, in this case
  - The algorithm spills a value only when it runs out of registers
  - Spilling pre-emptively does not help in straight-line code
  - With loops and conditionals, your mileage may vary

Should we work the example with ***k*** = 3 and a reserved register?

- That case is left as an exercise for you …
- Or, code it up and try it with the reference allocator on CLEAR …

Spill and Restore

One more spill case that you might consider
(once everything else is working…)

- Two values with equal next-use distance
  - r12 and r13 have same next use
  - Assume neither is rematerializable
  - No obvious basis to choose
- If r13 is the result of a **load**, with a known address (result of a **loadI**) and r12 is not
  - Might be able to **restore** r13 with a **loadI** followed by a **load**, avoiding the **spill**
  - A value that can be reloaded from its original location is a **CLEAN** value
    - → Cannot be a store between the original load and the restore
    - → The store might change the value in memory
  - No telling how often this happens

Need to spill here →

```
add     r2,r8       =>  r12
loadI   132         =>  r10
load    r10         =>  13
  …
mult    r12, r13    =>  r25
sub     r17, r25    =>  r26
  …
add     r12, r13    =>  r32
  …
```