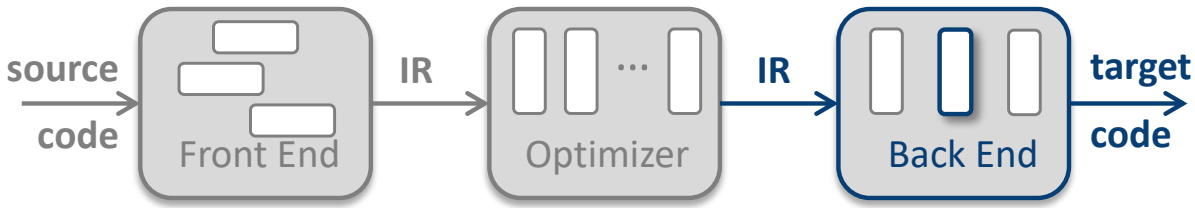
RICE

COMP 412

Local Register Allocation

Local Allocation Algorithm

Comp 412, Lab 2



Chapter 13, EaC3e

The Local Allocator

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

```
for each OP in the block, top to bottom
  for each use O in OP
    if (O.VR has no PR) then
      get a PR, say x
      load O.VR into x
      set O.PR to x
  for each use O in OP
    if O is last use of O.VR, free its PR
  for each def O in OP
    get a PR, say z
    set O.PR to z
```

Conceptual Sketch of the Algorithm (See Figure 13.5 in EaC3e excerpt)

Conceptually simple algorithm

- Process one operation at a time
- Allocate uses first
- Free last uses
- Allocate definition(s) last

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, allocator must choose an occupied **PR** and spill its contents to memory

Algorithm due to Sheldon Best, ~1957, Fortran I compiler @ IBM.

The Local Allocator

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

```
for each OP in the block, top to bottom
    for each use O in OP
        if (O.VR has no PR) then
            get a PR, say x
            load O.VR into x
            set O.PR to x
    for each use O in OP
        if O is last use of O.VR, free its PR
    for each def O in OP
        get a PR, say z
        set O.PR to z

    check the maps for consistency
```

Need some data structures to support allocation

- A map, VRToPR
- A map PRToVR
- A map VRToSpillLoc
- A map PRNU

The code to update & maintain these various maps is a critical part of the allocator.

The most common bugs involve messing up one or more of the maps.

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*)

Can maintain a stack of free **PR**s or iterate over PRToVR.

- 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 value currently in **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

Spill locations map to **VR**s, not **PR**s

Complications will arise

- Breaking a tie: what happens if two **PR**s both have the farthest next use?

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.

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 each OP in the block, top to bottom
    for each use O in OP
        if (O.VR has no PR) then
            get a PR, say x
            load O.VR into x
            set O.PR to x
    for each use O in OP
        if O is a last use, free its PR
    for each def O in OP
        get a PR, say z
        set O.PR to z

    check the maps for consistency
```

One More Nit (*common bug*)

- In a single op, a PR cannot hold two different use values
  - Cannot be both OP1 and OP2 unless they have the same **VR**
  - Mark **PR**s as they are allocated
  - *get a PR* checks the mark
  - Unmark **PR**s when checking for last uses.
- Multiple people will hit this bug & bring it to office hours

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

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 →

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
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		∞

index →

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

k = 4

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**

	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

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 with a Reserved Spill Register

Local Allocator, Same Example, 1 Register Reserved for Spills

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		∞

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

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

	pr0	pr1	pr2
PRToVR	—	—	—

Example

k = 4, with 1 register reserved for spilling

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
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
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		∞

index →

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	**vr5**	**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”

How do Spill and Restore actually work?

- A general-purpose spill, in **ILOC**, uses two operations:
  - A **loadI** to put the spill location’s address into the reserved register
  - A **store** to move the spilled value from its **PR** into its spill location
- A general-purpose restore, in **ILOC**, uses two operations:
  - A **loadI** to put the spill location’s address into the reserved register
  - A **load** to retrieve the spilled value from its spill location into its new **PR**
- The reserved register is critical for the spill
  - If the allocator spills, all registers are in use
  - On a restore, it could actually use the newly allocated **PR** for the address
- We don’t want to reserve the register, unless it will be used
  - If **MAXLIVE** > ***k***, the allocator reserves the register
  - If **MAXLIVE** ≤ ***k***, the allocator will not spill & does not reserve the register

Final Code, with the Spill & Restore

	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
Spill	loadI	32,768						—	pr3	5
	store	vr1	pr0					—	pr3	∞
	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
Restore	loadI	32,768		∞				—	pr3	8
	load	—	pr3	∞				vr1	pr1	8
	store	vr0	pr0	∞				vr1	pr1	∞

Spill Code

- Allocator assigned 32,768 for **VR1**'s spill location
- Inserted **loadI** / **store** pair before operation where it spilled **VR1**
- Inserted **loadI** / **load** pair before operation where it restored **VR1**

“Spill” and “Restore”

There is an important special case for spill and restore.

- If the allocator spills a **VR** that was created by a **loadI**:
  - The value does not need to go into memory
  - We can re-create the value with another **loadI**
  - Such a value can be **rematerialized** rather than **restored**
- A rematerializable value:
  - Does not need a store to spill the value.
    The allocator can just free the register.
  - Does not need a **loadI** / **load** pair to restore the value.
    The allocator can use a **loadI**, identical to the one that created the value.

- A general spill / restore pair takes two **loadI**s, a **store**, & a **load**
- A rematerializable value takes just one **loadI** — a much lower cost

In the example, we spilled a rematerializable value.

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

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	∞

Because the allocator rematerialized **VR1**, it did not use the reserved register.

- If all the spills can be rematerialized, it does not need a reserved register.

You cannot tell in advance if all the spills will rematerialize.

Remember, the problem is **NP-Complete.**

— You are not going to find an algorithm that always gets the optimal solution

Some Final Words

You now know enough to get started

- Build your renamer and pass Code Check 1
- Build your initial allocator and pass Code Check 2
  - Get **SPILL** and **RESTORE** correct
  - Test extensively for correctness
    - → Use the scripts and the student contributed test files
  - Test for scalability (*growth in allocator running time*)
    - → Use the timing harness
- Add advanced spilling
- Tinker with speed

And, above all,

- Do regression testing and save the working versions of your code

Remember, 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.

COMP 412, Fall 2020
Lab 2, Lecture on the Register Allocator

Slide 0

This lecture is the third lecture for **COMP 412 Lab 2** — the **local register allocator**.

You should have already watched the **overview** lecture and the **renaming** lecture.

This lecture focuses on the actual algorithm that performs **allocation** and **assignment**. It also rewrites the code to reflect the decisions that it makes.

Slide 1

The algorithm that we recommend you build is a **bottom-up local allocator**.

It is a **bottom-up** algorithm because it synthesizes the allocation based on low-level knowledge of the code and the partial solution that it has computed so far.

The word **local** refers to the scope of the allocator. It operates on **straight-line** or **branch-free** code, often called a **block**. Algorithms that optimize a single block are called **local** algorithms.

The algorithm is a **simple greedy heuristic** that makes **incremental decisions** as it processes the operations in a block from **top to bottom**. The box on the right side of the slide shows a conceptual sketch of the algorithm. You will want to look closely at Figure 13.5 in the excerpt from the 3rd edition of the textbook. It provides a more detailed version of the algorithm.

The algorithm is conceptually simple. It processes one operation at a time.

Within an operation, it first "allocates" the uses — that is, it makes sure that each of the values used at the start of the operation is available in a **physical register** at the start of the algorithm.

It looks at each **use**. If the value is **not** in a physical register, it allocates one, moves the value into that register, and updates the **IR** to reflect the allocation.

Next, it checks to see if any of the uses are **last uses** for their value. If they are, then it can free the **physical registers** that those **last uses** occupy. The allocator can reuse those freed registers for the definition in this operation or for some subsequent operation.

Finally, if the operation defines a value, the algorithm allocates a **physical register** to hold that new value. (The new value cannot already occupy a register, since it did not exist before this point in the code.)

Clearly, the action **get a physical register** is the key to this algorithm. If some **physical register** is free, then **get a physical register** can simply return that register's name.

If no **physical register** is free, then the allocator must choose some **physical register**, **insert code to move its contents to memory**, **record where it stored those contents,** and return that freed register's name.

We call this process of evicting a value from a register and storing its value **a spill**. (When we later reverse this action, we will call it a **restore**.)

This algorithm was invented in 1957 by Sheldon Best at IBM. It has been reinvented many times — most famously as the offline page replacement algorithm **MAX** by Belady (also at IBM).

Slide 2

To make this algorithm work efficiently, the implementation needs to build and maintain a set of **maps** between **virtual registers**, **physical registers**, **memory locations**, and **operations**.

Specifically, it needs:

- a map from **virtual register** to **physical register** that we will call **VRToPR**
- a map from **physical register** to **virtual register** that we will call **PRToVR**
- a map from a **virtual register** to its **spill location in memory** that we will call **VRToSpillLoc**, and
- a map from a **physical register** to the operation that next uses the value that is currently in that **physical register**. We will call this map **PRNU**.

The code to build and update these maps is a critical part of the allocator.

In my experience building multiple versions of the reference allocator and helping to debug literally hundreds if not thousands of student labs over the years, the most common bugs involve messing up one or more of these maps.

To make your experience easier, you should simply add a check at the end of the main loop that checks the maps for consistency. If **VRToPR** says that **VR17** is in **PR2**, then **PRToVR** should say that **PR2** holds **VR17.**

This will be expensive. It will cost time proportional to the # VRs. On T128k, # VRs is about 128,000. Don't worry about the cost. You can comment out the check once your code is working correctly.

Slide 3

How does **get a physical register** work?

If there is a free physical register, **get a physical register** can return its name. To make this efficient, the reference implementation maintains a stack of free registers.

If there are no free physical registers — that is, they are all in use — then the allocator must pick one. Our greedy heuristic says to pick the physical register whose next use is farthest in the future (hence the need for **next use** information from the renamer).

Once it has that physical register name, say PRX, it must

1. Clear out **PRX**, saving its value to memory — into its "spill location". It allocates a place for the value in memory — its **spill location**. It records that location in **VRToSpillLoc** under the virtual register name of the spilled value.

   (It can use **PRToVR** to find the **virtual register** that corresponds to the value, then use that name to index **VRToSpillLoc**.)
2. Next, **get a physical register** inserts a store operation into the code that moves the value in **PRX** into the spill location in memory. The store operation goes before the current operation.
3. Finally, it returns the **physical register name** to the allocator.

The allocator uses that physical register name in the operation.

1. If the allocator is processing a **use,** the allocator inserts code after any spill and before the operation to restore the value from memory (move it from its location in memory into the named register).
2. If the allocator is processing a **definition**, the allocator just uses that name as the result register in the operation.

Of course, since this is a compilers course, there are complications. What happens if two physical registers have the same distance to their next use? Does the choice matter? Can this make a difference?

If the allocator needs to spill, it will need a register to hold the memory address. Thus, most people will check **MAXLIVE**. If **MAXLIVE** > *k*, then the allocator will reserve one physical register to hold spill addresses.

This particular problem is a property of **ILOC**. Other **instruction sets** may not have it.

Slide 4

One more thing:

This point may seem obvious, but multiple people hit this bug every year.

In a single operation, a given **PR** can only hold one value — that is one **VR**.

If you have a three-register add operation and the inputs are different **VR**s, you **cannot** use the same **PR** to hold both of them. Thus, your code needs to keep track of whether or not you have used a given **PR** in the current operation.

Conceptually, the allocator should mark **PR**s as being used in the current operation.
Initially, it marks all **PR**s as unused.
As it allocates a **PR** to one of the input operands, it marks that **PR** as used.
Then, **get a physical register** can check the marks and return an unmarked (and, therefore, unused) register.
When the allocator checks for last uses, it clears the marks.

This problem does not happen with definitions.
An operation can use the same **PR** for a use and a definition.

If you think about this problem, you will realize that you can get away with a single mark — recording the register assigned to the first operation.

How does this bug manifest itself? The allocator starts to work on three-register operation. Neither of the input operands is in a register. It starts to work on the first use, so it asks **get a physical register** to find a register, which it does — say **PR1**.

Now, the allocator looks at the second use. Again, it needs a register. It asks **get a physical register** to find a register. Unless we have marked **PR1** in a way that tells **get a physical register** to ignore it, **get a physical register** will return the same physical register — **PR1**. Since nothing has changed, we should expect **get a physical register** to give us the same answer.

Slide 5

To see how the algorithm works, let's go back to the example from the renaming lecture.

Assume that *k* is **four** — that is, we have four physical registers.
When we renamed this code, we discovered that **MAXLIVE** was four, so the code should fit into four registers.

Since we should not need to spill, we don't need to reserve a register.

Each slide shows the state of the **IR** and **maps** after the allocator is done processing the operation indicated by the arrow.

Slide 6

Here is the initial state of the code and the maps.
All of the entries in **VRToPR** and **PRToVR** are set to "**invalid**".

We will start with the first operation, a **load immediate**.

Slide 7

The **load immediate** has only one use—a constant that does not need a register.
Since the use has no **PR**, the check for a **last use** — a dead value in a **PR**— finds nothing.

The definition of **VR1** needs a **PR**. **Get a physical register** finds that all four physical registers are free. It returns **PR0**.

The allocator records **PR0** in the **IR**. It updates the maps.

Slide 8

The second operation is a **load**, which has only one use.

This **load** uses **VR1**, which already resides in **PR0**.
The allocator records **PR0** for it in the **IR**.

Next, it checks to see if the **use** of **VR1** is a last use. The **next use field** for **VR1** in this operation shows the value 8, so **VR1** is still live (that is, it has a next use), and the allocator does **not** free it.

The definition of **VR2** needs a register. **Get a physical register** finds that **PR1** is free and returns it. The allocator records **PR1** in the intermediate representation and updates the maps.

Slide 9

The third operation is another **load immediate**. It's use is a constant that does not need a register.

The definition of **VR1** needs a **physical register**. **Get a physical register** finds that all **PR2** is free. It returns **PR2**.

The allocator records **PR2** in the intermediate representation. It updates the maps.

Slide 10

The fourth operation is another **load**, which has just one use.
It uses **VR7**, which already resides in **PR2. T**he allocator records **PR2** for it in the **IR**.
The allocator checks to see if **VR7** is dead. Since its **next use field** is set to infinity, **VR7** is dead and the allocator frees **PR2.**

(*This is the first time we have seen the allocator **free** a physical register.)*

The definition of **VR4** needs a register. **Get a physical register** finds that **PR2** is free and returns it. The allocator records **PR2** in the **IR** and updates the maps.

We are adding information to the **VRToPR** map. The assignment of **VRs** at the start of the operation is shown in gray and the assignment at the end is shown in black.

Slide 11

Notice how the allocator assigned **PR2** to both the use and the definition in this **load**.

Because the operation reads its uses when it starts to execute and writes its definition at the end of its execution, this pattern of register use works fine. In fact, it is something that an assembly language programmer would write.

The reference implementation exhibits this pattern of reuse regularly, because it keeps the free physical registers in a stack. Using a stack in this way drops the cost of **get a physical register** to a small constant when one or more of the **physical registers** are free.

Slide 12

The fifth operation is another **load immediate**, which has no uses that need registers.

The definition of **VR6** needs a **physical register**. **Get a physical register** finds that **PR3** is free.

The allocator records **PR3** in the **IR** and updates the maps.

Slide 13

The sixth operation is a **load**, which has one **use**, **VR6**.

The allocator finds that **VR6** is already in **PR3**, so it records that in the **IR**.

When it checks for **last uses**, it finds that **VR6** is dead after this use, so it frees **PR3**.

The definition of **VR5** needs a register. **Get a physical register** finds that **PR3** is free and returns it. The allocator records **PR3** in the **IR** and updates the maps.

Slide 14

The seventh operation is a **multiply**, which has two uses and a definition.

Looking at the uses, the allocator first finds that both **VR4** and **VR5** have **physical registers**. It records them in the **IR**.

Next, it finds that both **VR4** and **VR5** are dead after this operation, so it frees both of their **physical registers** (**PR2** and **PR3**).

The operation defines **VR3**, so **VR3** needs a physical register. **Get a physical register** returns **PR2**. The allocator records **PR2** in the **IR** and updates the maps.

Slide 15

The eighth operation is an **add**. It has two uses and one definition.

Looking at the uses, both **VR2** and **VR3** already have **physical registers**. It records those registers in the **IR**.

It discovers that both registers are dead after this operation, so it **frees** both of them.

Finally, it looks for a **physical register** for the definition of **VR0**; **get a PR** returns **PR1**. The allocator records **PR1** in the **IR** and updates the maps.

Slide 16

The last operation is **store**, which has **two uses** and no **definitions**.

The first use is **VR0**, which already resides in **PR1**.

The second use is **VR1**, which already resides in **PR0**.

The allocator records those physical registers in the **IR**.

Both of the uses are last uses, so it frees them.

At this point, the allocator is done with the code. Notice that all of the physical registers are free. That suggests, that we updated the maps correctly.

As expected, the code allocates easily into **four** physical registers.

Slide 17

Now, let's consider what happens if the allocator reserves a register for the address calculation for a spill. That reduces the number of registers available to hold values by one.

The allocation behaves exactly as it did before, until it reaches the point where it needs a fourth physical register.

Slide 18

For the **load immediate**, the allocator again assigns **PR0** to the definition of **VR1**.

(set timing to five or ten seconds)

Slide 19

For the **load**, **VR1** is already in **PR1** and is live after the operation because of its use in the last operation.

The allocator assigns **VR2** to **PR1**.

Slide 20

The **load immediate** in slot three has no uses. It needs a **physical register** to hold the definition of **VR7**. The allocator assigns **PR2** to **VR7**.

Slide 21

The fourth operation is a **load**. It has one use, **VR7**, that resides in **PR2**. This use is a last use, so the allocator frees **PR2**.

The allocator then turns around and assigns **PR2** to hold the new value created by the definition of **VR4**.

Up to this point, the allocator has produced the same decisions and results as it did when it had four registers available to hold computed values.

Slide 22

The fifth operation is another **load immediate**. Recall that **MAXLIVE** rose to four at the end of operation **five**.

The **load immediate** has no uses. It needs a **physical register** to hold its result, **VR6**.

Get a physical register looks for a **PR**; all of them are in use. Thus, it needs to pick one of the three **physical registers** to spill.

Slide 23

The algorithm says to pick the **VR** that has the **farthest next use**. As you can see from the **NextUse** table, that is **VR1.** So, the allocator marks it to spill before operation four.

At this point in the lecture, we will just mark the spill on the slide.

Slide 24

Spilling **VR1** frees **PR0**. **Get a PR** returns **PR0**. The allocator records **PR0** for **VR6** in the **IR** and updates the various maps.

Slide 25

Operation six is a load.

Its use, **VR6**, resides in **PR0**. The allocator records that fact in the **IR**.
Because the use is a **last use**, the allocator frees **PR0**, which makes it available to hold the **virtual register** defined by the load, **VR5**.

The allocator updates the IR and the maps and moves on to the next operation.

Slide 26

Operation seven is a **multiply**. It has two uses, **VR4** and **VR5**. Both are in physical registers, so the allocator records them in the **IR** — **PR2** and **PR0**, respectively.

Both of the uses are last uses, so the allocator frees the two **physical registers**.

Finally, the allocator assigns **PR0** to the definition of **VR3** and records it in the **IR**.

Slide 27

The eighth operation in the block as an **add**. Again, both **VR**s used in the operation **already** reside in **physical registers**. Both of the uses are last uses. The allocator records the respective **physical registers** in the **IR** and then frees them.

When it looks at the definition, it assigns **PR0** to **VR0**, records the assignment in the **IR**, and updates the various maps.

Slide 28

The last operation is a **store**. It has two uses, **VR0 and VR1**.

VR0 is already in **PR0**, so the allocator can just record that fact.

VR1 needs a **physical register**. Recall that the allocator spilled **VR1** just before operation **five**.

The allocator finds a **physical register**, **PR1**, and assigns it to **VR1**.

Next, it needs to restore the spilled value of **VR1** into **PR1**. For the moment, we will just mark that on the slide.

Finally, the algorithm frees both **PR0** and **PR1**, because they both represent last uses.

The allocator has **fit the computation** into **three physical registers** plus one used for address calculations in **spills**.

Slide 29

In the example, we simply marked the **spill** and **restore** on the slide. That does not provide us with any actual insight as to what your allocator should do.

First, let's consider the general case.

In a **spill**, the allocator inserts two operations **before** the operation that provoked the spill.

— a **load immediate** to move the address of the spill location into the reserved register, and

— a **store** from the **spilled physical register** into the spill location

A **restore** is similar. The allocator inserts, **before** the operation that provoked the restore:

— a **load immediate** of the spill location's address into the reserved register, and

— a **load** from the **spill location** into the newly allocated **physical register**

If the allocator needs to spill and restore before an operation, the **spills** precede the **restores.**

Both these sequences used the reserved register, but the register is crucial in the spill. When the allocator decides to spill, it has already run out of registers. Thus, it **cannot find** a register to hold the address. It needs the reserved register In a restore, it could actually put the address into the newly allocated **PR**, and then use that **PR** on both sides of the **load** operation.

Slide 30

Here is the example with the spill and restore code inserted.

The allocator assigned the memory location at 32,768 to **VR1** for use as its spill location. It recorded that value in **VRToSpillLoc** so that it could find the location when it needed to restore **VR1.**

To spill **VR1**, it used a **loadI** followed by a **store**.

To restore **VR1**, it used a **loadI** followed that with a **load**. Of course, it found the address for the **loadI** by looking in **VRToSpillLoc** indexed by **VR1**.

Slide 31

There is one important special case.

If the allocator decides to spill a value that was created by a **load immediate**, it does not need to store the value and then re-load it. Instead, it can re-create the value using a **load immediate** operation.

We say that such a value is **rematerializable**.

To **spill** a rematerializable value, the allocator just frees the register. It does not need to insert code for the spill.

To **restore** a rematerializable value, the allocator can just insert a copy of the original **loadI** operation. It does not need the two operation sequence.

In fact, if the allocator spills a rematerializable value, the restore will make the original **loadI** operation **dead** — that is, its result is not **live**. The allocator can delete that original **loadI** operation. This is an allowed optimization.

The key point is that a rematerializable value takes one **loadI** to spill, where a general spill is much more expensive — two **loadI** operations plus a **store** and a **load**.

Slide 31

If the allocator recognizes that **VR1** is rematerializable, it can produce this code.
It accomplishes the spill with a single **loadI** just before the **store** operation.

The total cost of spilling is the one-cycle **loadI** operation.
It is significantly less expensive than the code shown two slides ago.

Slide 32

Notice that we ran the algorithm with ***k*** *= 4* and a reserved register, but the allocator did not need the reserved register because it spilled a rematerializable value.

This is a conundrum. If we don't reserve the register, we don't spill. If we do reserve the register, we have an extra register.

The allocator cannot easily make this decision. The problem is, after all, **NP Complete**.

If you really wanted to fine tune this point, you could run the allocator. If every spilled value was rematerializable, you could run it again without reserving the register and restricting the allocator to only spill rematerializable values. Changing the number of registers will probably change the spill choices.

In code generated from real programs, the rematerializable values are a smaller fraction of the code than they will be in small **ILOC** blocks built to test your allocators. This problem is not, in the larger scheme of things, a burning issue.

Slide 33

Now you know enough to start writing your allocator.

First, add code to your parser to collect the **maximum source register number**. The renamer needs that information.

Next, build the renamer and pass Code Check 1.
Test your renamer on codes other than the ones in the code_check_1 directory. There are literally hundreds of **ILOC** blocks in the **ILOC** contributed directory.

Once you have a renamer, start on the allocator.
Focus on getting **SPILL** and **RESTORE** to work correctly.
Test extensively — for correctness and also for scalabiliry

If you have a working allocator and have time left, add some of the advanced spilling ideas from the **SPILLING video.**

Finally:

(1) Do regression testing. If you are like me, you will focus on one or two examples where you are not happy with the results. Because **allocation** is **NP COMPLETE**, improving those cases may produce worse results, or incorrect results, on other cases. You need to test against an extensive set of blocks — larger than the code check blocks or the report blocks.

(2) At each stage, when you have a working version, save it. Every year, I talk to a student who made “one more change” and could not remember all of the details to undo it.