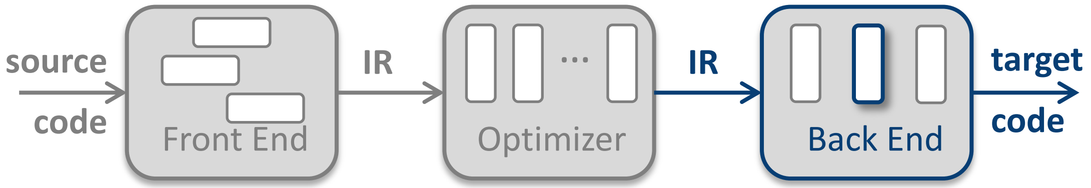
COMP 412

Local Register Allocation

The Renaming Pass

Comp 412, Lab 2



Read excerpt from § 13, EaC3e

Critical Terminology:

Virtual Registers, Physical Registers, & Live Ranges

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 **PR**s, cannot add more*
- *Valid assembly code cannot name more PRs than machine supports*

A live range is a distinct value

- A live range starts with the creation of a value, or its *definition*
- A live range ends with the last use of a value

Your register allocator will find live ranges and assign them unique **VR**s

Local Register Allocation

In *x* ← *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
```

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 source register, **vr** for a virtual register & **pr** for a physical register
- Code reuses registers

[†] **If you find ILOC confusing, see the video on the ILOC virtual machine.**

Local Register Allocation

In *x* ← *y*, *x* is a *definition* and *y* is a *use*. (standard terminology)

A Critical Observation

```
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 strategy that some students use is to have the parser keep the **store**'s second use in the **op2** slot and print **store**s with that knowledge.

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 writes **MEM(op2)**, not **op2**.
- Your lab needs to treat both operands as **uses**, not as **defs.**
- One or more of you will forget this fact.

The Big Picture

Lab 2 Code Check 1 Tests the Renamer

Lab 2 has two major components

- A **renaming** pass
  - Find distinct *live ranges* in the block (*values*)
  - Assigns each live range a unique *virtual register* name
  - Rewrites every source register name with the appropriate virtual register
  - Annotates each *definition* and each *use* with the distance to the next *use*

Builds a map from LR to VR

- An **allocation** pass
  - Assigns **VR**s to **PR**s
  - Discovers points in the code where |*values*| > |*registers*|
  - At those points, chooses one or more registers to move out of registers
  - Inserts code to move values from register to memory (a *spill*) and to move them back from memory into a register (a *restore*)

Builds a map from VR to PR

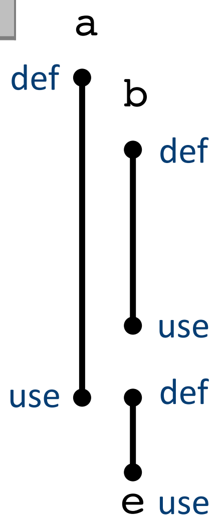
- The renaming pass is tested in Code Check 1 & reused in Lab 3
- The allocator is the harder part of the lab

Find Distinct Live Ranges

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*
  - *An operation reads its operands at the start of its execution & writes its result(s) at the end of its execution*

#	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, because operation 4 reads b before it writes e

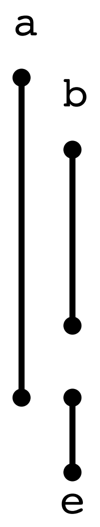
In code with branches & jumps, live ranges are more complex (§ 13)

Find Distinct Live Ranges

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*
  - *An operation reads its operands at the start of its execution & writes its result(s) at the end of its execution*

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



Observation

Let **MAXLIVE** be the largest number of simultaneously live values at any op.

- If **MAXLIVE** ≤ *k*, allocation is easy
- If **MAXLIVE** > k, some values must be spilled to memory

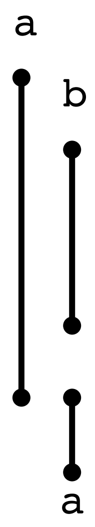
To find live ranges & to compute **MAXLIVE**, we will walk the block from bottom to top and keep track of which values are live.

Find Distinct Live Ranges

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*
  - *An operation reads its operands at the start of its execution & writes its result(s) at the end of its execution*

#	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.

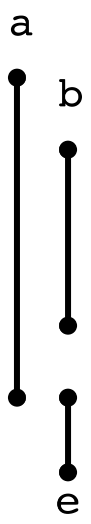
- Changed **e** to **a**
- Operation 4:
  - → Uses the value defined in op 0
  - → Creates a new value used in op 5
  - → *Kills* the value from operation 0
  - → Still two separate live ranges
- Overwriting a value kills it

Find Distinct Live Ranges

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*
  - *An operation reads its operands at the start of its execution & writes its result(s) at the end of its execution*

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



WARNING

The computation of **MAXLIVE** has some subtlety. In particular, think through what happens with:

- *A definition that is never used*
- *A use that has no definition*
- *An operation that uses the same definition twice*

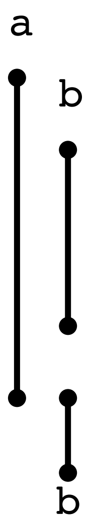
Each of these cases can lead to an incorrect **MAXLIVE** value

Finding Live Ranges

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*
  - *An operation reads its operands at the start of its execution & writes its result(s) at the end of its execution*

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



Why Use Live Ranges?

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

- The distinct **LR**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*)

Renaming Algorithm

```
VRName ← 0                    // start VR names at 0
for i ← 0 to max SR number
    SRToVR[i] ← invalid
    LU[i] ← ∞
index ← block length
for each Op in the block, bottom to top
    for each operand O that OP defines
        if SRToVR[O.SR] = invalid   // Unused DEF
            then SRToVR[O.SR] ← VRName++
        O.VR ← SRToVR[O.SR]
        O.NU ← LU[O.SR]
        SRToVR[O.SR] ← invalid  // kill OP3
        LU[O.SR] ← ∞
    for each operand O that OP uses
        if SRToVR[O.SR] = invalid   // Last USE
            then SRToVR[O.SR] ← VRName++
        O.VR ← SRToVR[O.SR]
        O.NU ← LU[O.SR]
    for each operand O that OP uses
        LU[O.SR] ← index
    index--
```

Fig 13.4 in EaC3e Excerpt

Finding live ranges, next uses, and renaming to VRs

- Walk block from last op to first op
  - Within an *op, defs* before *uses*
  - Builds a map from **SR** to **VR**
  - Tracks last use information
- If **SR** has no **VR**, assign one
  - Indicates a *last use*
- Rename **SR** to **VR** at each mention of the **SR** in the code
- Tag 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)*

Renaming Algorithm

```
VRName ← 0                  // start VR names at 0
for i ← 0 to max SR number
    SRToVR[i] ← invalid
    LU[i] ← ∞
index ← block length
for each Op in the block, bottom to top
    for each operand O that OP defines
        if SRToVR[O.SR] = invalid   // Unused DEF
            then SRToVR[O.SR] ← VRName++
        O.VR ← SRToVR[O.SR]
        O.NU ← LU[O.SR]
        SRToVR[O.SR] ← invalid  // kill OP3
        LU[O.SR] ← ∞
    for each operand O that OP uses
        if SRToVR[O.SR] = invalid   // Last USE
            then SRToVR[O.SR] ← VRName++
        O.VR ← SRToVR[O.SR]
        O.NU ← LU[O.SR]
    for each operand O that OP uses
        LU[O.SR] ← index
    index--
```

Fig 13.4 in EaC3e Excerpt

Finding live ranges, next uses, and renaming to VRs

- Walk block from last op to first op
  - Within an *op, defs* before *uses*
  - Builds a map from **SR** to **VR**
  - Tracks last use information
- If **SR** has no **VR**, assign one
  - Indicates a *last use*
- Rename **SR** to **VR** at each mention of the **SR** in the code
- Tag 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

Traverses block in this direction

INDEX	OPCODE	Argument 1				Argument 2				Argument 3				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 →

Shows tables when alg. is done with op at index arrow

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

VRName: $\boxed{0}$

***store** has no <u>def</u>*

Update r1 as a use

Update r0 as a use

Back to the Example

After Processing Operation 8

INDEX	OPCODE	Argument 1				Argument 2				Argument 3				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 → 8

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

VRName : $\boxed{2}$

Update r1 as a def & kill it

Update r1 as a use
Update r3 as a use

Back to the Example

After Processing Operation 7

INDEX	OPCODE	Argument 1				Argument 2				Argument 3				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 → 7

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

VRName : $\boxed{4}$

Update r3 as a def & kill it

Update r3 as a use
Update r5 as a use

Back to the Example

After Processing Operation 6

INDEX	OPCODE	Argument 1				Argument 2				Argument 3				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 : $\boxed{6}$

Update r5 as a def & kill it

Update r4 as a use

Back to the Example

After Processing Operation 5

INDEX	OPCODE	Argument 1				Argument 2				Argument 3				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
index → 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	–	4	6	–
LU	8	7	∞	6	5	∞

VRName: $\boxed{7}$

Update r4 as a def & kill it

No uses

Back to the Example

After Processing Operation 4

INDEX	OPCODE	Argument 1				Argument 2				Argument 3				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: $\boxed{7}$

Update r3 as a def & kill it

Update r2 as a use

Back to the Example

After Processing Operation 3

INDEX	OPCODE	Argument 1				Argument 2				Argument 3				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: $\boxed{8}$

Update r2 as a def & kill it

Back to the Example

After Processing Operation 2

INDEX	OPCODE	Argument 1				Argument 2				Argument 3				NEXT OP
		SR	VR	PR	NU	SR	VR	PR	NU	SR	VR	PR	NU	
0	loadI	128	–	–	∞	–	–	–	∞	r0	–	–	∞	1
1	load	r0	–	–	∞	–	–	–	∞	r1	–	–	∞	2
index → 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: $\boxed{8}$

Update r1 as a def & kill it

Update r0 as a use

Back to the Example

After Processing Operation 1

index →

INDEX	OPCODE	Argument 1				Argument 2				Argument 3				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 r0 as a def & kill it

Back to the Example

After Processing Operation 0

index →

INDEX	OPCODE	Argument 1				Argument 2				Argument 3				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

Back to the Example

Code renamed so that virtual registers correspond to live ranges

INDEX	OPCODE	Argument 1				Argument 2				Argument 3				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								# LIVE
			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 code check 1, your lab must rename registers & print out the renamed ILOC. The code must execute correctly on 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								# LIVE
			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

MAXLIVE

Placement of the bullets is deliberately ambiguous. Uses occur at the start of an operation's execution; defs occur at the end of its execution. PowerPoint does not provide the precision to draw that well.

Lab 2 Code Check 1

Your lab may assign different VR names than lab2_ref does.

Code Check 1 tests to see if your allocator renames 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

Renamed 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. (*not* ***VR0***)

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

Local Register Allocation

→ An example that came to my attention in 2009 …

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

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

3-value pattern

We built two versions, one that reused names and one that did not

Renaming turns the version with reuse in the version without reuse.

The test block, t128k.i, is a 128,000-line version of the 20-value pattern.

gcc 4.1, -O3, PowerPC G4 target

The Value of Renaming

Local Register Allocation

→ An example that came to my attention in 2009 …

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

Without reuse of names, gcc handled 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 spilled 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.