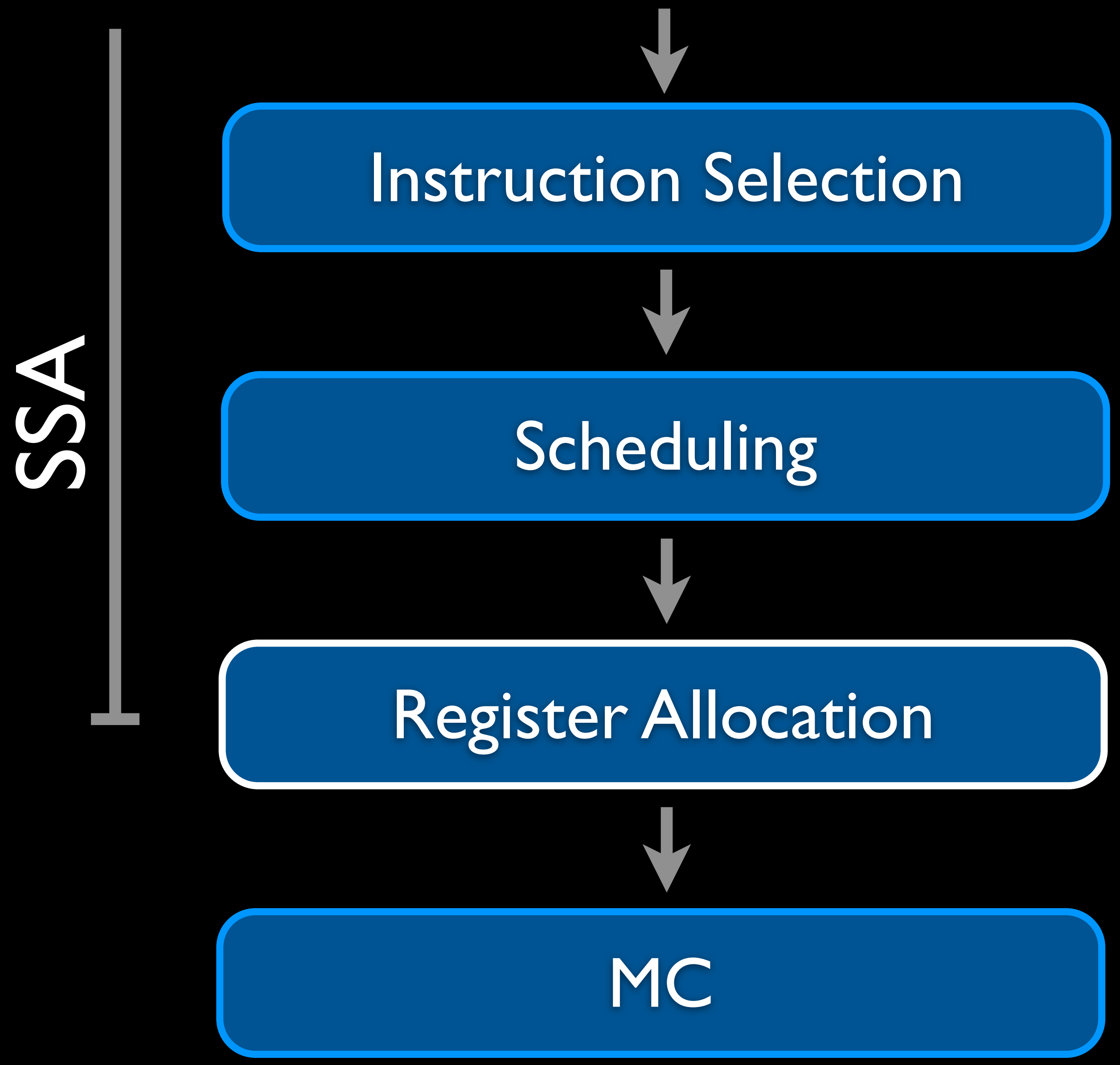
Register Allocation in LLVM 3.0

Jakob Stoklund Olesen

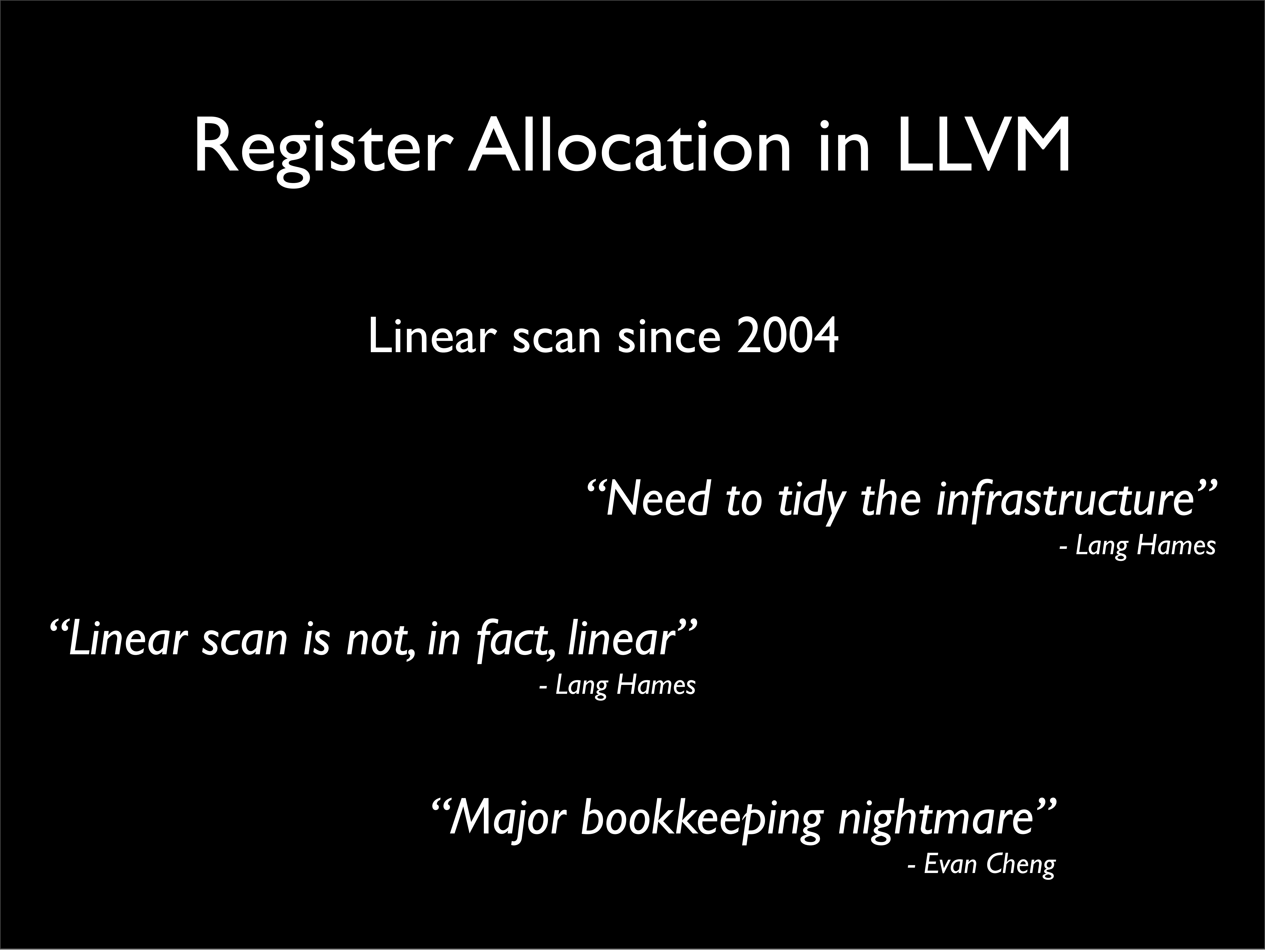
Talk Overview

- Register allocation in LLVM
- What’s wrong with linear scan?
- The new LLVM 3.0 register allocator
- Global live range splitting
- Future work

Register Allocation in LLVM



This is a simplified view of the LLVM code generator.
Instruction selection and scheduling works on SSA form.
The register allocator takes the code out of SSA form and replaces virtual registers with physical registers.



Register Allocation in LLVM

Linear scan since 2004

"Need to tidy the infrastructure"
- Lang Hames

"Linear scan is not, in fact, linear"
- Lang Hames

"Major bookkeeping nightmare"
- Evan Cheng

LLVM has used a linear scan register allocator since 2004.
It has worked very well for users, but judging from previous register allocator talks, LLVM developers have not been happy.
The code needs to be cleaned up, but I want to talk about more fundamental problems.

Register Allocation in LLVM

Eliminate virtual registers and SSA

```
  %v1 = movs #10
loop:
  %v2 = phi %v1, %v3
  %v3 = sub %v2, %v0
  str %v50, [%v51, %v3]
  cbnz %v3, loop
```

→

```
  movs r0, #10
loop:
  sub r0, r1
  str r7, [r8, r0]
  cbnz r0, loop
```

Optimize code speed and size

Let’s take a look at what register allocation does in LLVM.
What problem are we trying to solve?

How?

> *"Register allocation can then be reduced to the problem of K-coloring the resulting [interference] graph, where K is the number of registers available on the target architecture."*
>
> *- Wikipedia*

How does the register allocator do this?
Wikipedia says the problem is isomorphic to graph coloring.
That's only a small part of the problem. There is a lot more to it than that.

Graph Coloring

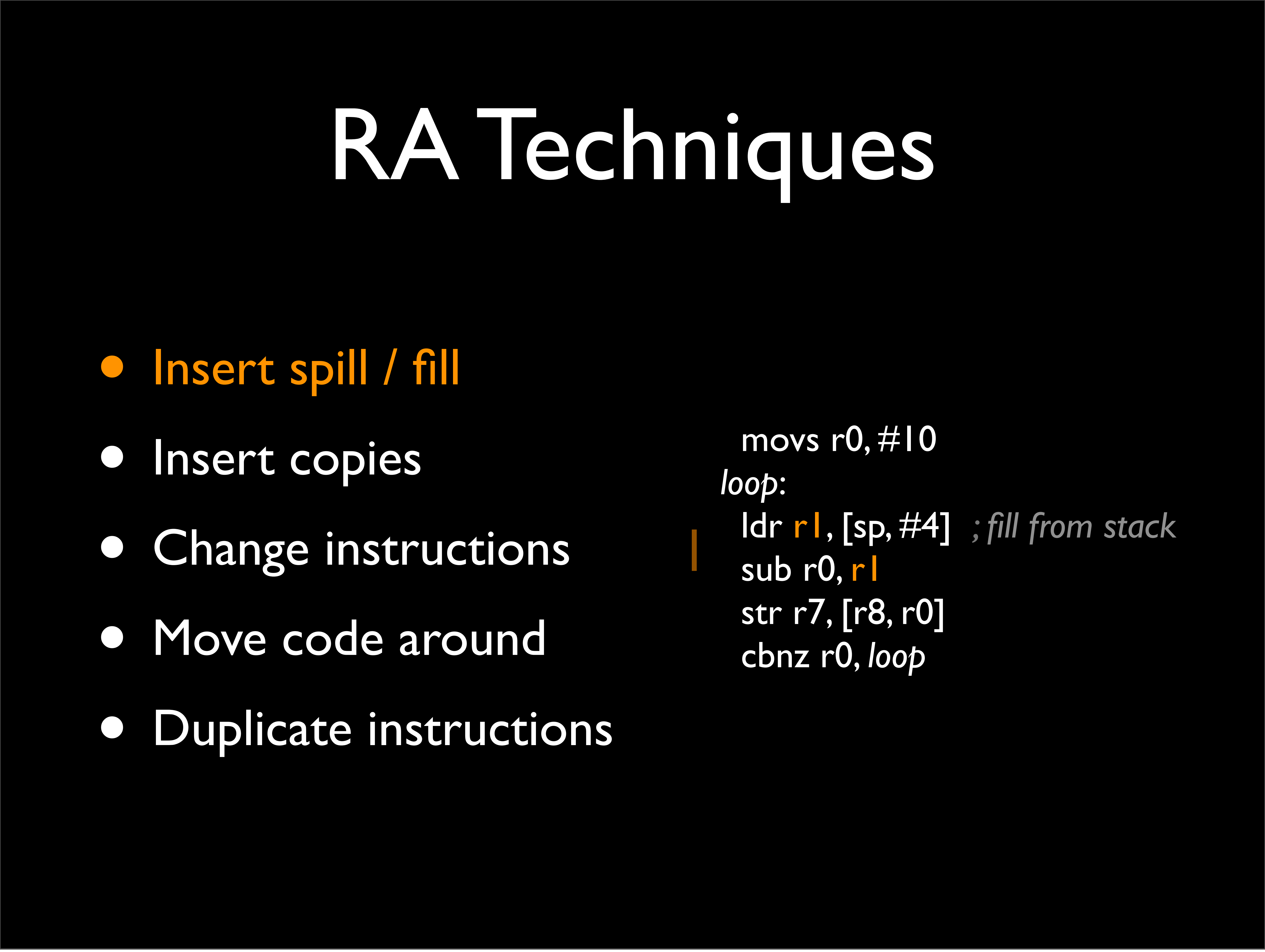
- Interference graph is expensive to build
- Spill code placement is more important than coloring
- Need to model aliases and overlapping register classes
- Flexibility is more important than the coloring algorithm

RA Techniques

- Insert spill / fill
- Insert copies
- Change instructions
- Move code around
- Duplicate instructions

```
  %v1 = movs #10
loop:
  %v2 = phi %v1, %v3
  %v3 = sub %v2, %v0
  str %v50, [%v51, %v3]
  cbnz %v3, loop
```

The most common register allocator operation is to assign a virtual register to a stack slot and insert spill and fill instructions.
In the example, we are unable to find a register for %v0 in the whole function, so it is spilled.



RA Techniques

- Insert spill / fill
- Insert copies
- Change instructions
- Move code around
- Duplicate instructions

```
  movs r0, #10
loop:
  ldr r1, [sp, #4]   ; fill from stack
  sub r0, r1
  str r7, [r8, r0]
  cbnz r0, loop
```

We could insert a fill instruction to load the value right before it is used.
However, it is very important to place spill and fill instructions carefully to optimize code speed.
- In this case, the fill can be placed outside the loop, so it will only execute once.

RA Techniques

- Insert spill / fill
- Insert copies
- Change instructions
- Move code around
- Duplicate instructions

```
  movs r0, #10
  ldr r1, [sp, #4]  ; fill from stack
loop:
  sub r0, r1
  str r7, [r8, r0]
  cbnz r0, loop
```

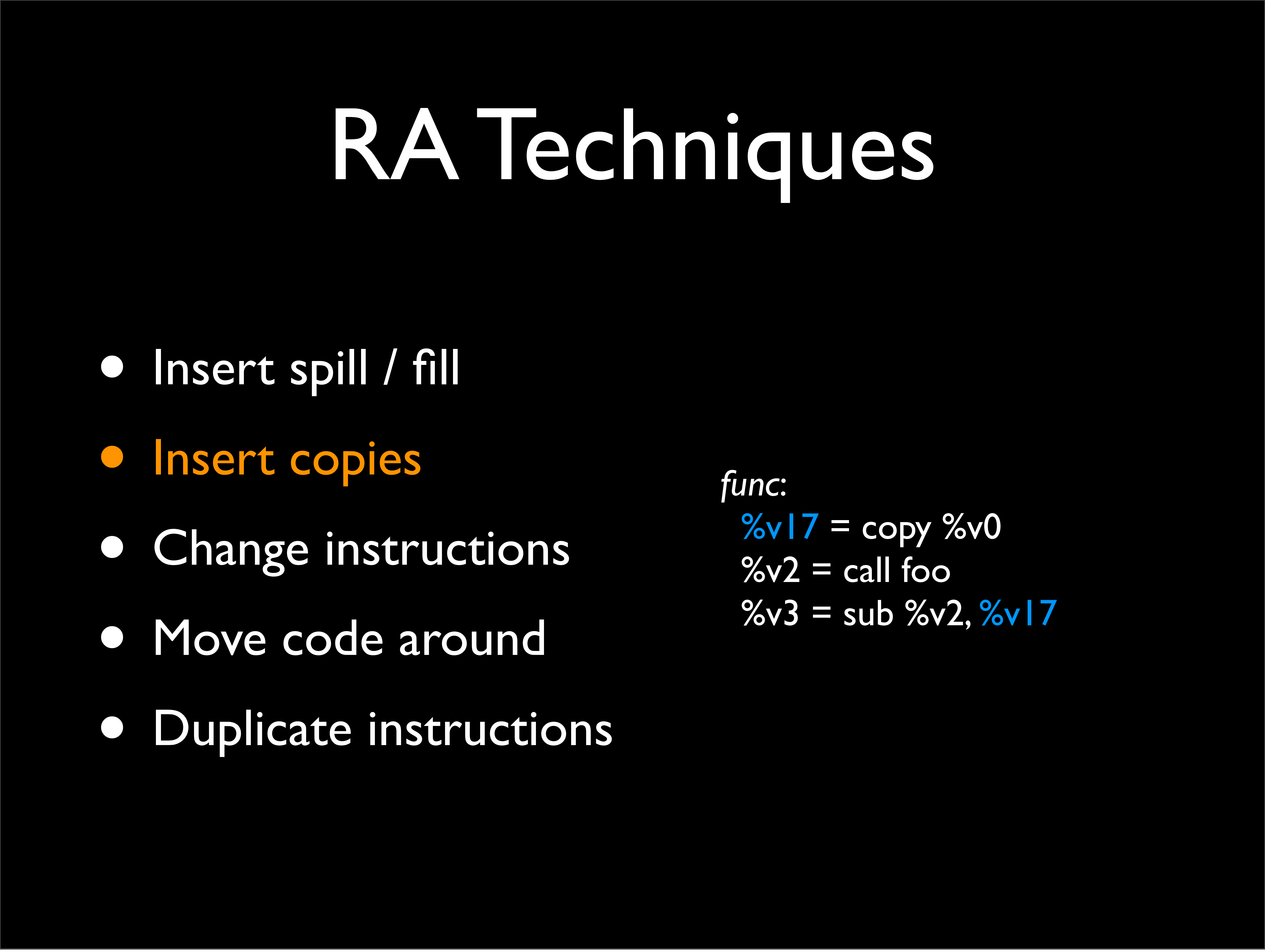
We are effectively splitting the live range of %v0 into two parts.
The part inside the loop is assigned to r1. The part outside the loop is assigned to a stack slot.
Spill code placement and live range splitting is a very important optimization.

RA Techniques

- Insert spill / fill
- Insert copies
- Change instructions
- Move code around
- Duplicate instructions

```
func:
  ; %v0 is 1st func argument
  %v2 = call foo
  %v3 = sub %v2, %v0
```

Another register allocator trick is to insert copy instructions.
Suppose %v0 is a function argument in r0, and we want to use it after a function call.
Copy the value to a callee saved register before the function call.

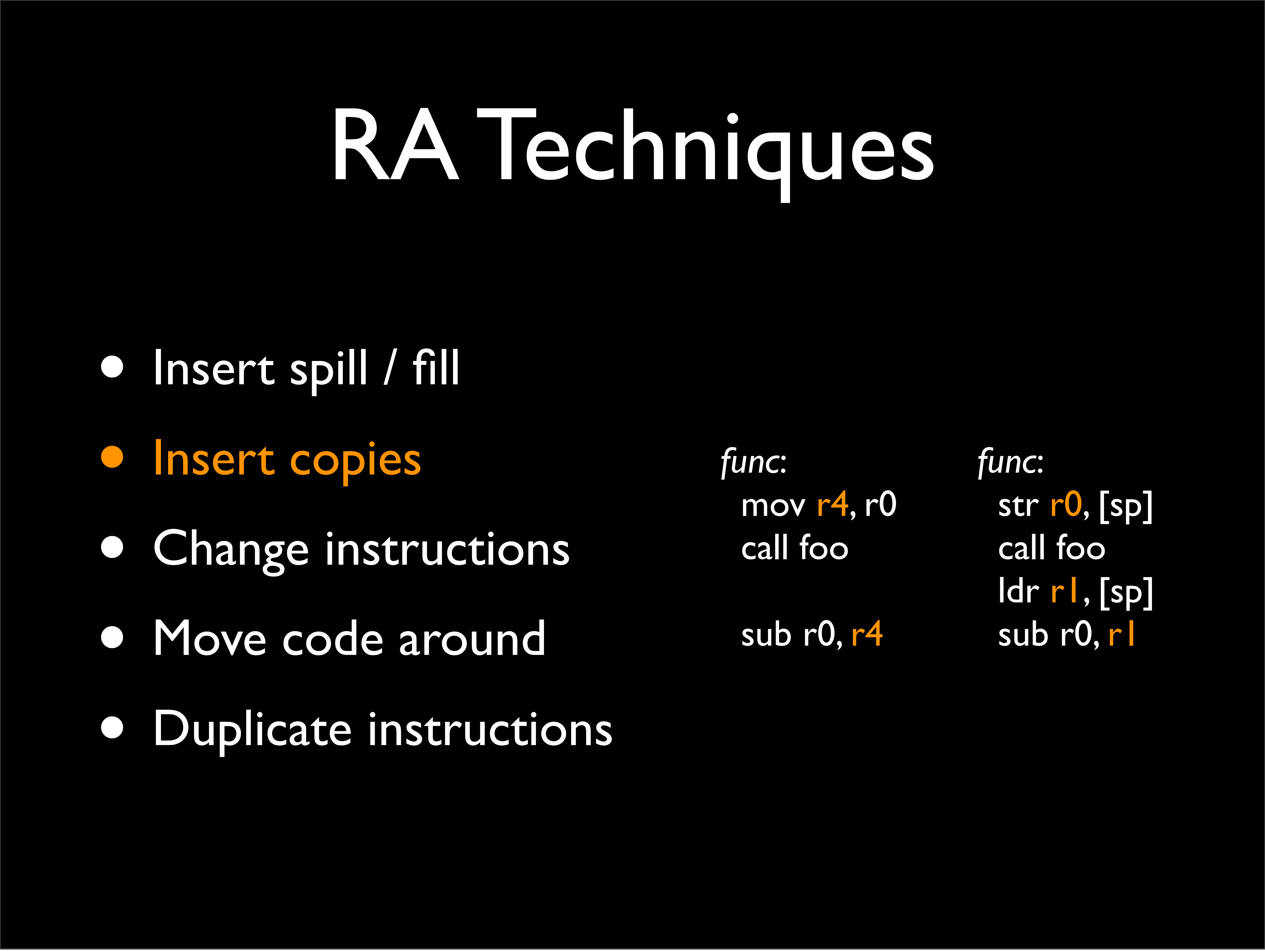


RA Techniques

- Insert spill / fill
- Insert copies
- Change instructions
- Move code around
- Duplicate instructions

```
func:
  %v17 = copy %v0
  %v2 = call foo
  %v3 = sub %v2, %v17
```

Now, %v17 can be assigned to r4, a callee-saved register.

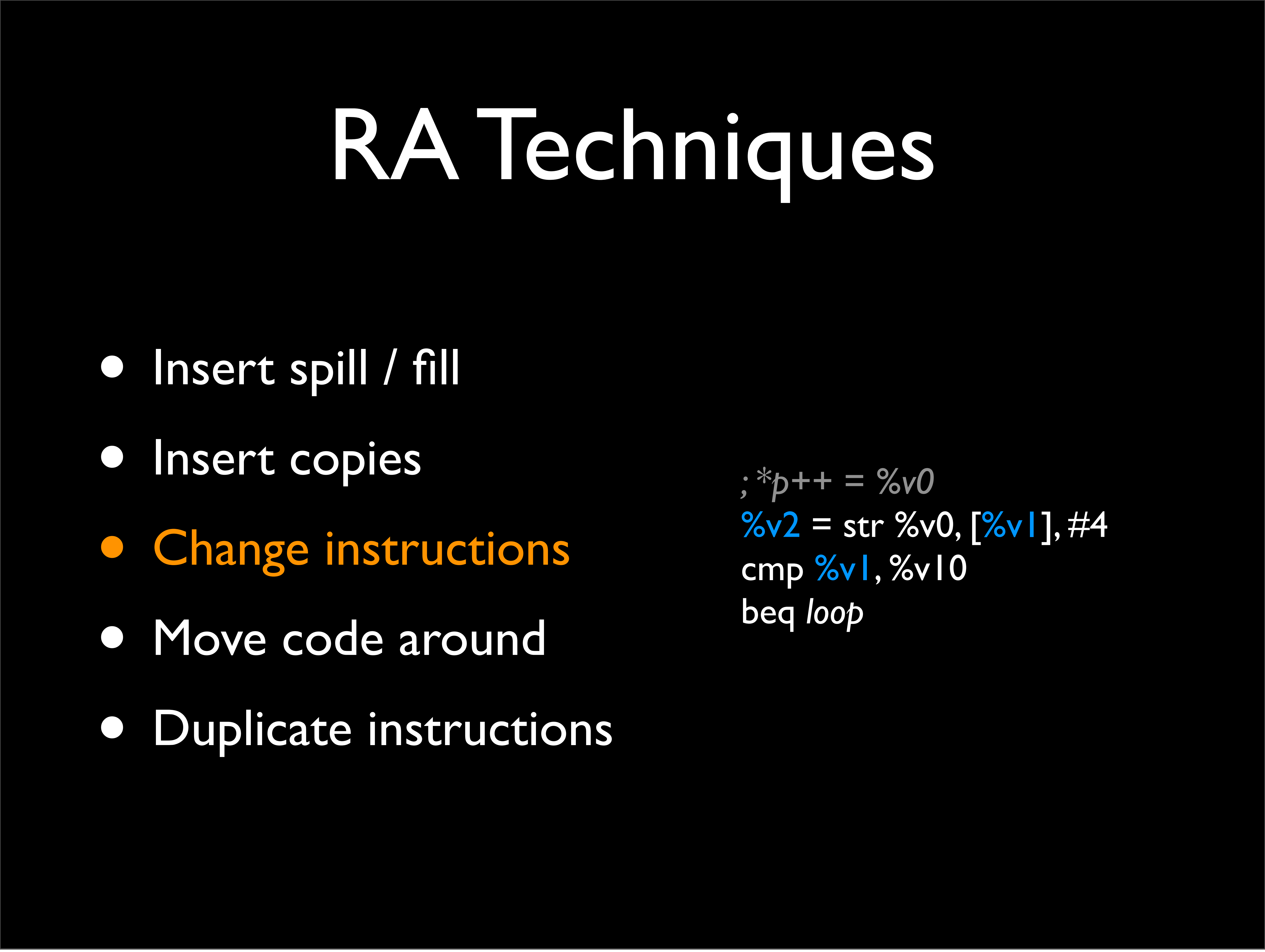


RA Techniques

- Insert spill / fill
- Insert copies
- Change instructions
- Move code around
- Duplicate instructions

```
func:                 func:
  mov r4, r0            str r0, [sp]
  call foo              call foo
                        ldr r1, [sp]
  sub r0, r4            sub r0, r1
```

The alternative would be to spill the register.



RA Techniques

- Insert spill / fill
- Insert copies
- Change instructions
- Move code around
- Duplicate instructions

```
; *p++ = %v0
%v2 = str %v0, [%v1], #4
cmp %v1, %v10
beq loop
```

The register allocator can also change the machine code instructions.
Here we have a store instruction with address write-back.
The input address in %v1 and the updated address in %v2 must be assigned the same physical register.
That means we won’t be able to use %v1 after the store unless we copy it first.
We can turn the store with write-back into a store and an add instead.

RA Techniques

- Insert spill / fill
- Insert copies
- Change instructions
- Move code around
- Duplicate instructions

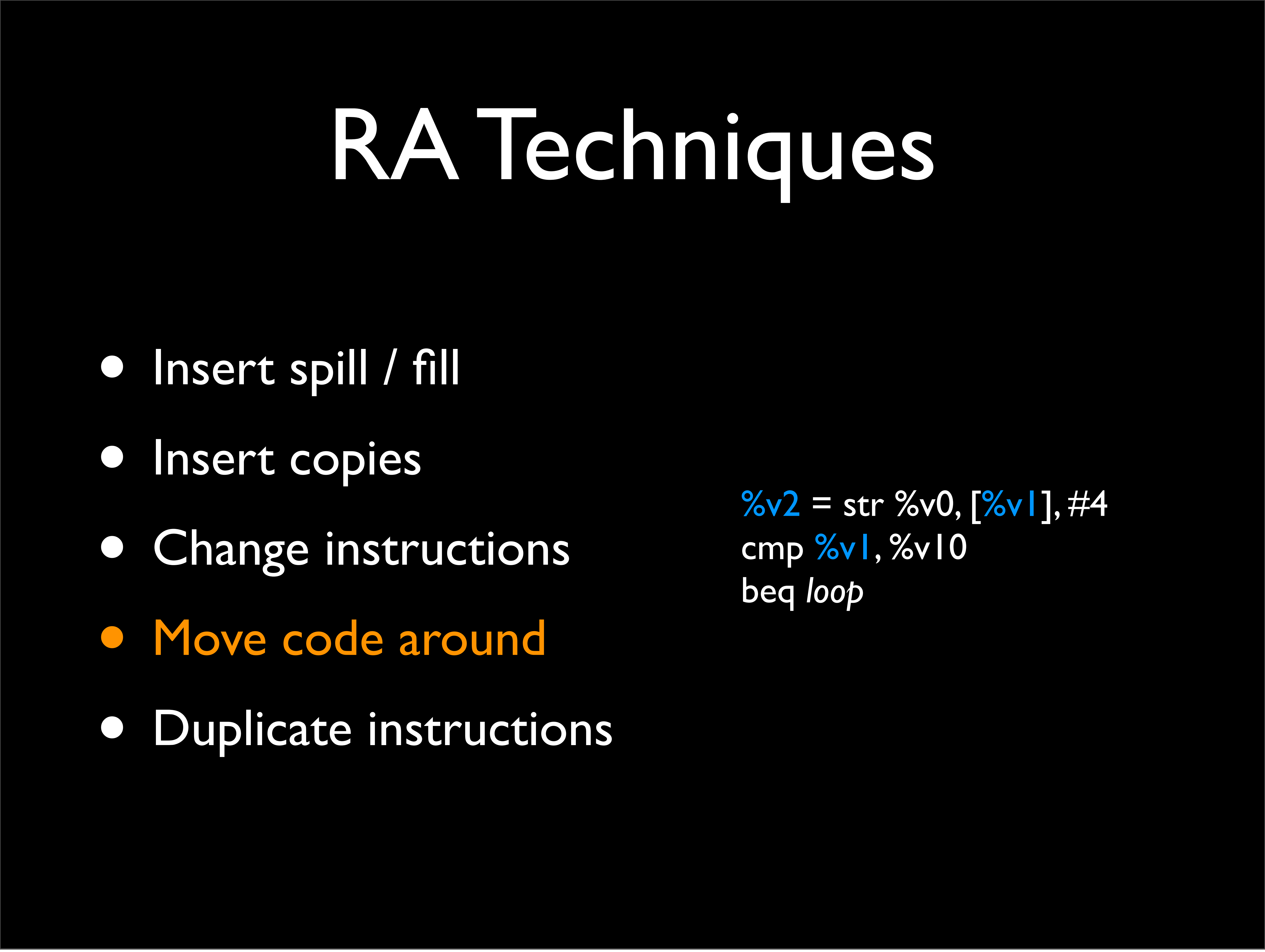
```
str %v0, [%v1]  ; *p = %v0
%v2 = add %v1, #4
cmp %v1, %v10
beq loop
```

Now, %v1 and %v2 can be assigned to different physical registers, r1 and r2.

RA Techniques

- Insert spill / fill
- Insert copies
- Change instructions
- Move code around
- Duplicate instructions

```
str r0, [r1]         mov r2, r1
add r2, r1, #4       str r0, [r2], #4
cmp r1, r7           cmp r1, r7
beq loop             beq loop
```

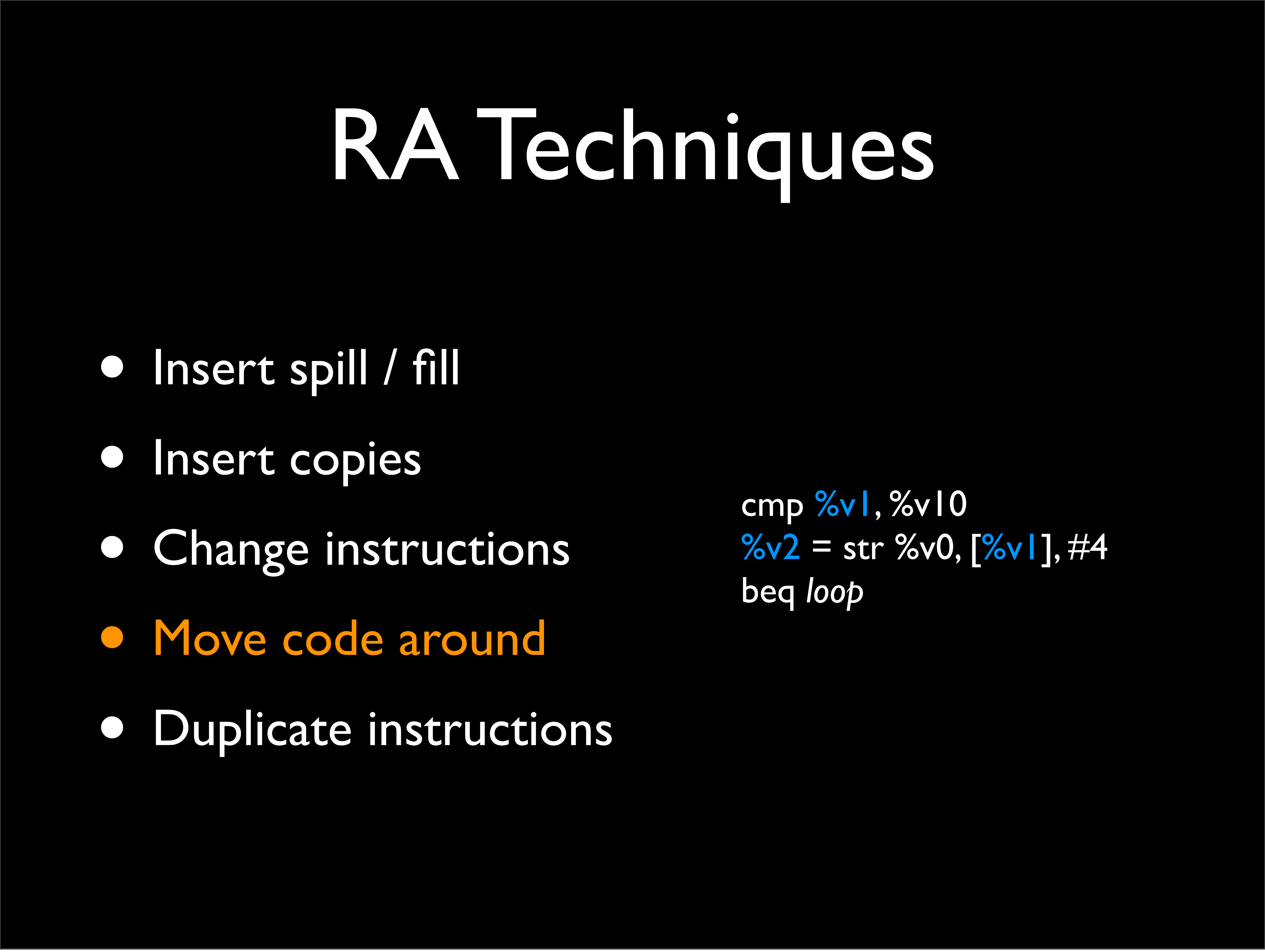


RA Techniques

- Insert spill / fill
- Insert copies
- Change instructions
- Move code around
- Duplicate instructions

```
%v2 = str %v0, [%v1], #4
cmp %v1, %v10
beq loop
```

Another way to solve the same problem is to rearrange the instructions.
By moving the compare in front of the store, the interference is resolved.



RA Techniques

- Insert spill / fill
- Insert copies
- Change instructions
- **Move code around**
- Duplicate instructions

```
cmp %v1, %v10
%v2 = str %v0, [%v1], #4
beq loop
```

Now the store is the last use of %v1, and we can allocate the same register to %v1 and %v2.

RA Techniques

- Insert spill / fill
- Insert copies
- Change instructions
- Move code around
- Duplicate instructions

```
cmp r1, r7
str r0, [r1], #4
beq loop
```

```
mov r2, r1
str r0, [r2], #4
cmp r1, r7
beq loop
```

Now the store is the last use of %v1, and we can allocate the same register to %v1 and %v2.

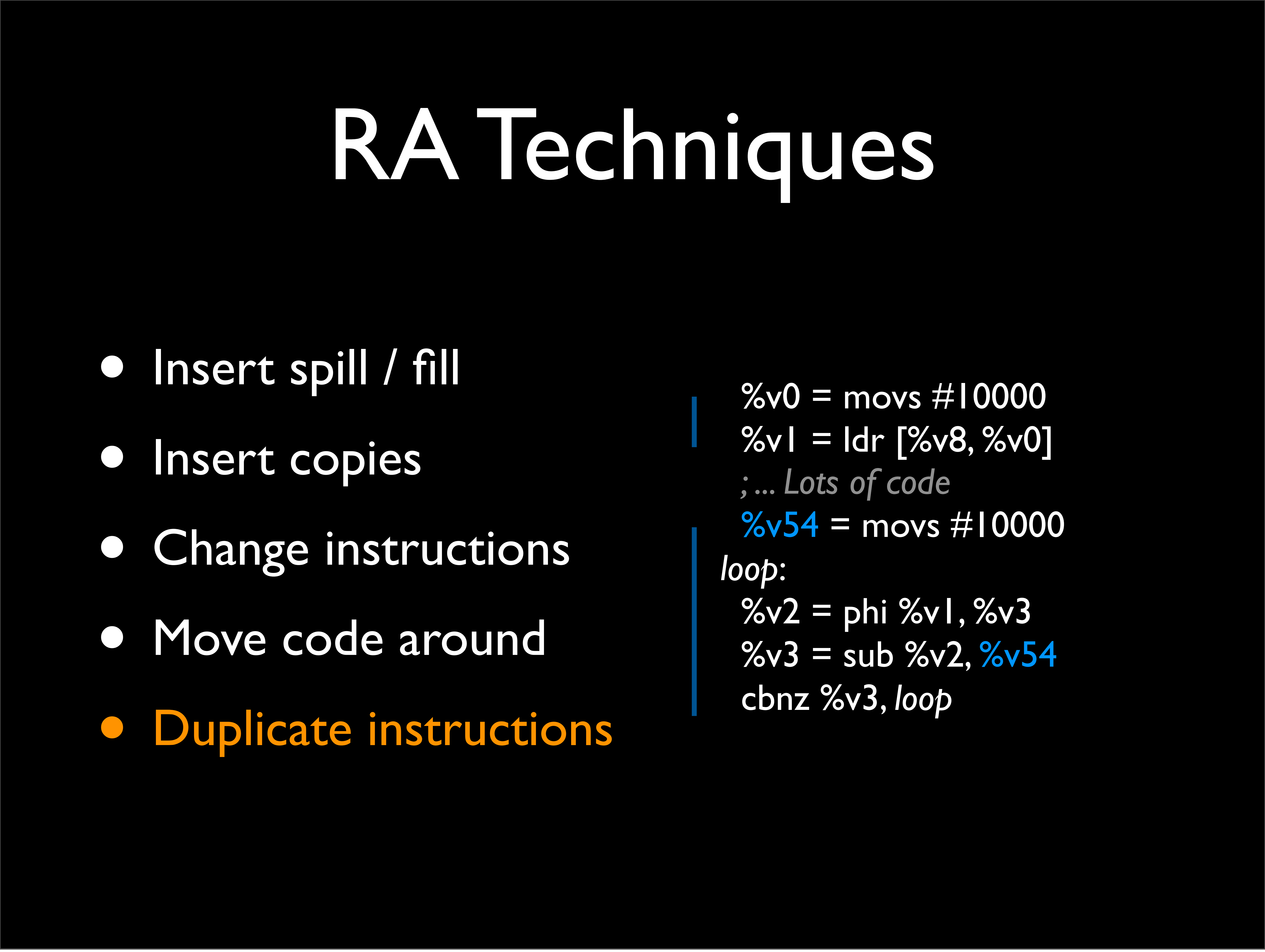
RA Techniques

- Insert spill / fill
- Insert copies
- Change instructions
- Move code around
- Duplicate instructions

```
  %v0 = movs #10000
  %v1 = ldr [%v8, %v0]
  ; ... Lots of code

loop:
  %v2 = phi %v1, %v3
  %v3 = sub %v2, %v0
  cbnz %v3, loop
```

Finally, the register allocator can duplicate instructions.
Constants are often put in registers early in the function.
Duplicating these instructions is cheap, and it reduces register pressure.



RA Techniques

- Insert spill / fill
- Insert copies
- Change instructions
- Move code around
- Duplicate instructions

```
  %v0 = movs #10000
  %v1 = ldr [%v8, %v0]
  ; ... Lots of code
  %v54 = movs #10000
loop:
  %v2 = phi %v1, %v3
  %v3 = sub %v2, %v54
  cbnz %v3, loop
```

Just like spill / fill instructions, it is important to place the duplicated instructions carefully. Here we materialize the constant outside the loop.

Linear Scan

- Insert spill / fill
- Move code around
- Duplicate instructions

```
  %v1 = movs #10
loop:
  %v2 = phi %v1, %v3
  %v3 = sub %v2, %v0
  str %v50, [%v51, %v3]
  cbnz %v3, loop
```

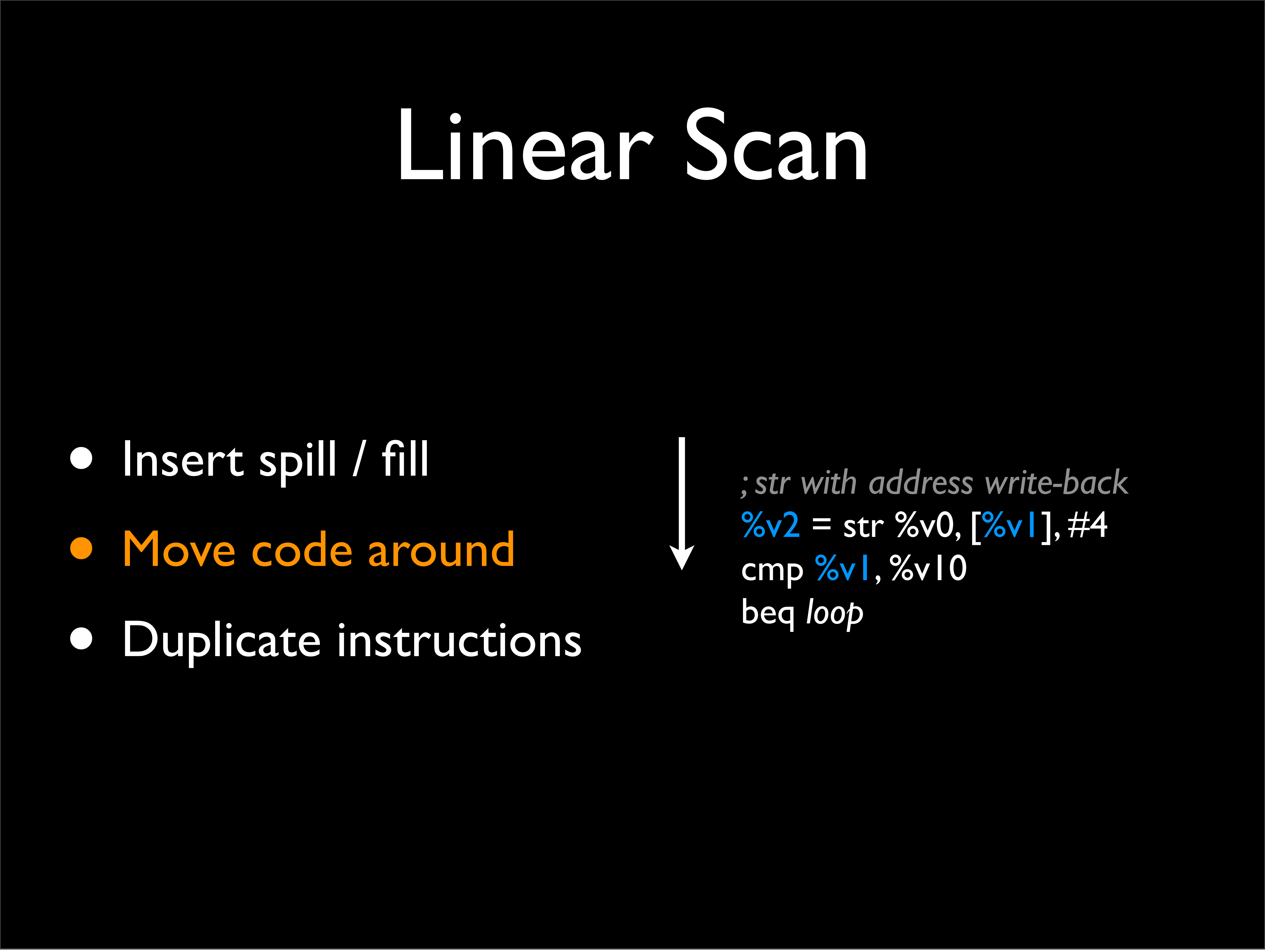
The linear scan allocator visits instructions in top-down order.
How do these techniques work with linear scan?
Obviously, linear scan can insert spill/fill instructions.

Linear Scan

- Insert spill / fill
- Move code around
- Duplicate instructions

```
  movs r0, #10
loop:
  ldr r1, [sp, #4]  ; fill from stack
  sub r0, r1
  str r7, [r8, r0]
  cbnz r0, loop
```

But the fill is inserted right before the instruction that uses the value.
There is no global live range splitting.



Linear Scan

- Insert spill / fill
- Move code around
- Duplicate instructions

```
; str with address write-back
%v2 = str %v0, [%v1], #4
cmp %v1, %v10
beq loop
```

How about rearranging instructions?
This doesn’t work either since the str is already done when we see the cmp.

Linear Scan

- Insert spill / fill
- Move code around
- Duplicate instructions

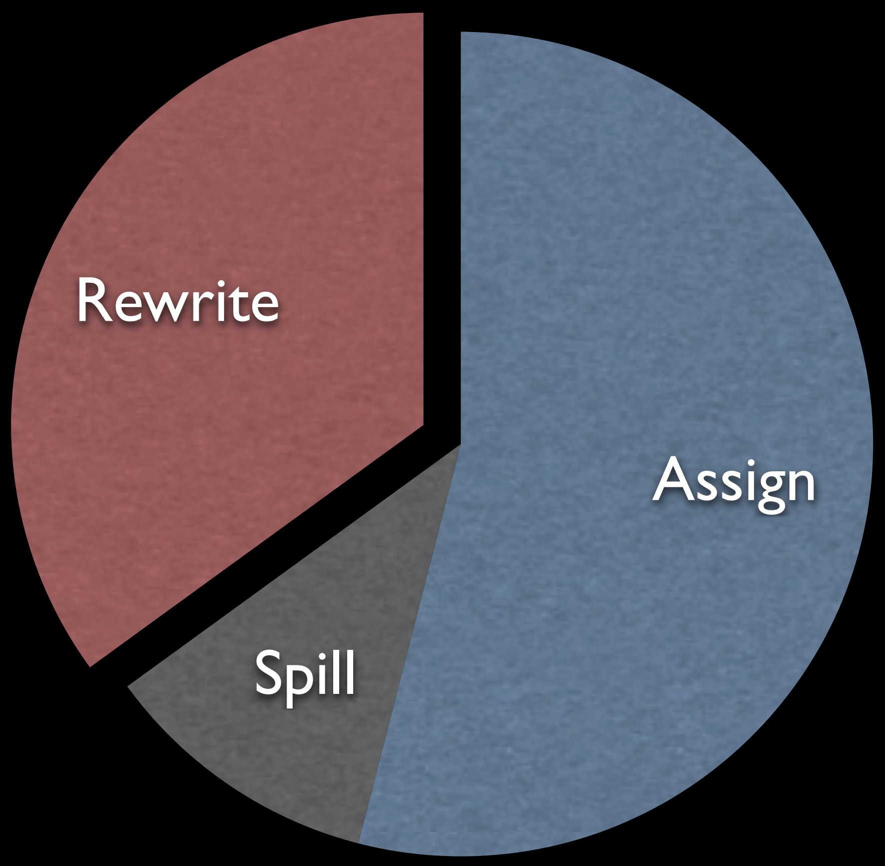
```
  %v0 = movs #10000
  %v1 = ldr [%v8, %v0]
  ; ... Lots of code
loop:
  %v2 = phi %v1, %v3
  %v54 = movs #10000
  %v3 = sub %v2, %v54
  cbnz %v3, loop
```

Linear scan can duplicate instructions.
Only immediately before the value is used.
Cannot take advantage of freed register in visited code.

Linear Scan

- Rewriter is very complicated
- Maintenance liability

RA Compile Time



The rewriter runs after linear scan as a peephole pass.
It cleans up a lot of the bad code.
It is quite expensive, a full third of the compile time.
It is very tricky source code.

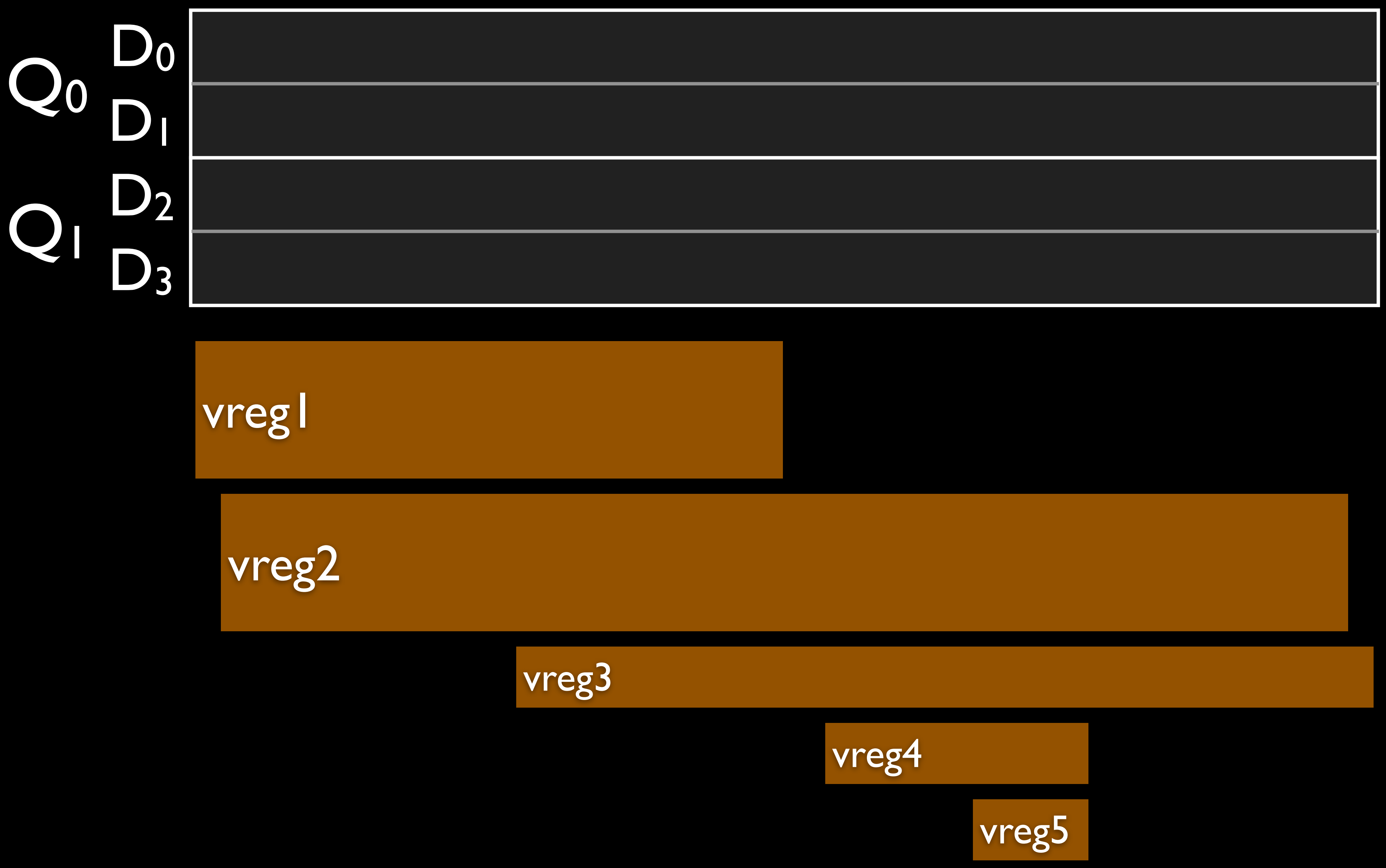
LLVM 3.0 Register Allocator

The LLVM 3.0 register allocator was designed to address these problems.

Design Goals

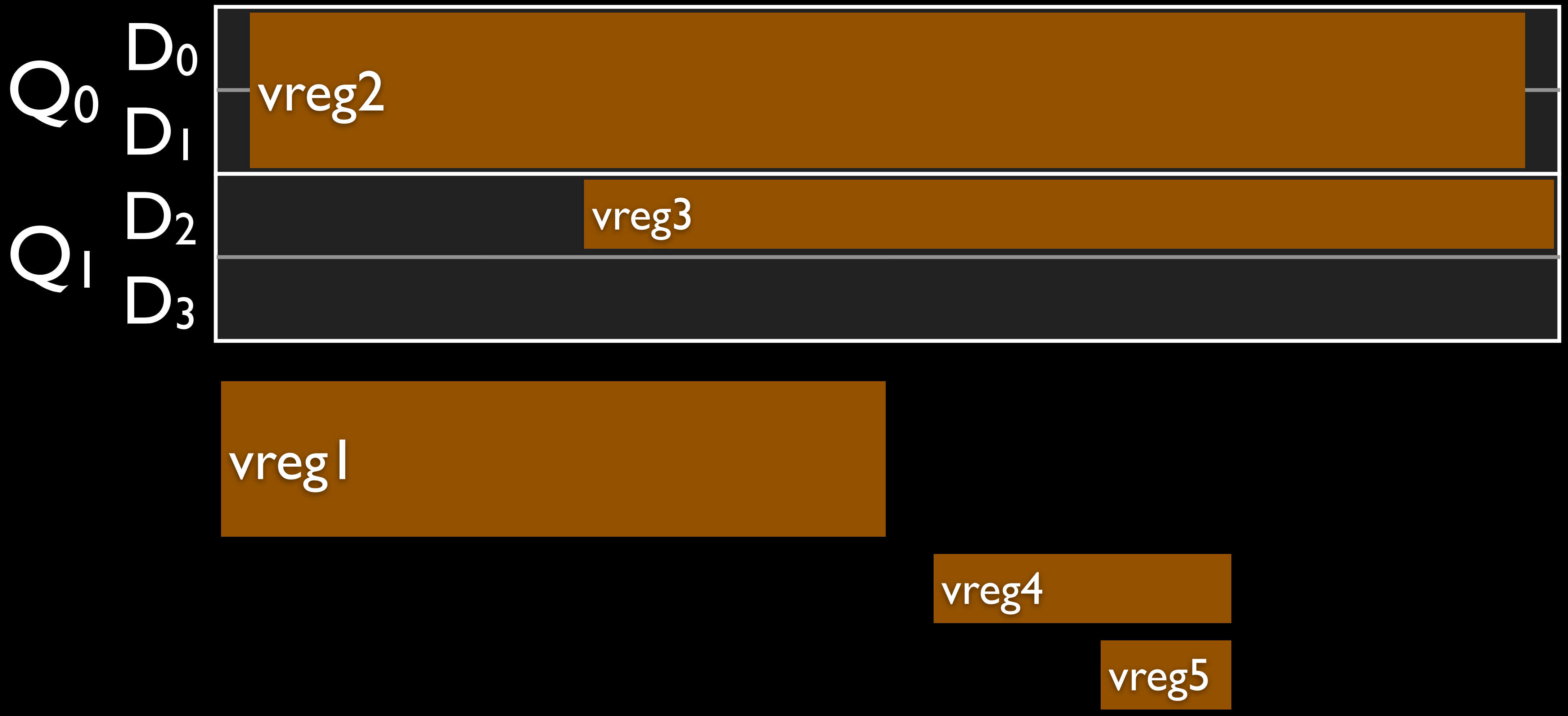
- Support existing constraints
- Don’t regress compile time
- Full live range splitting
- Edit machine code in flight
- Enable future improvements
- Eliminate complicated rewriter

Support overlapping register classes, sub-registers just like linear scan.

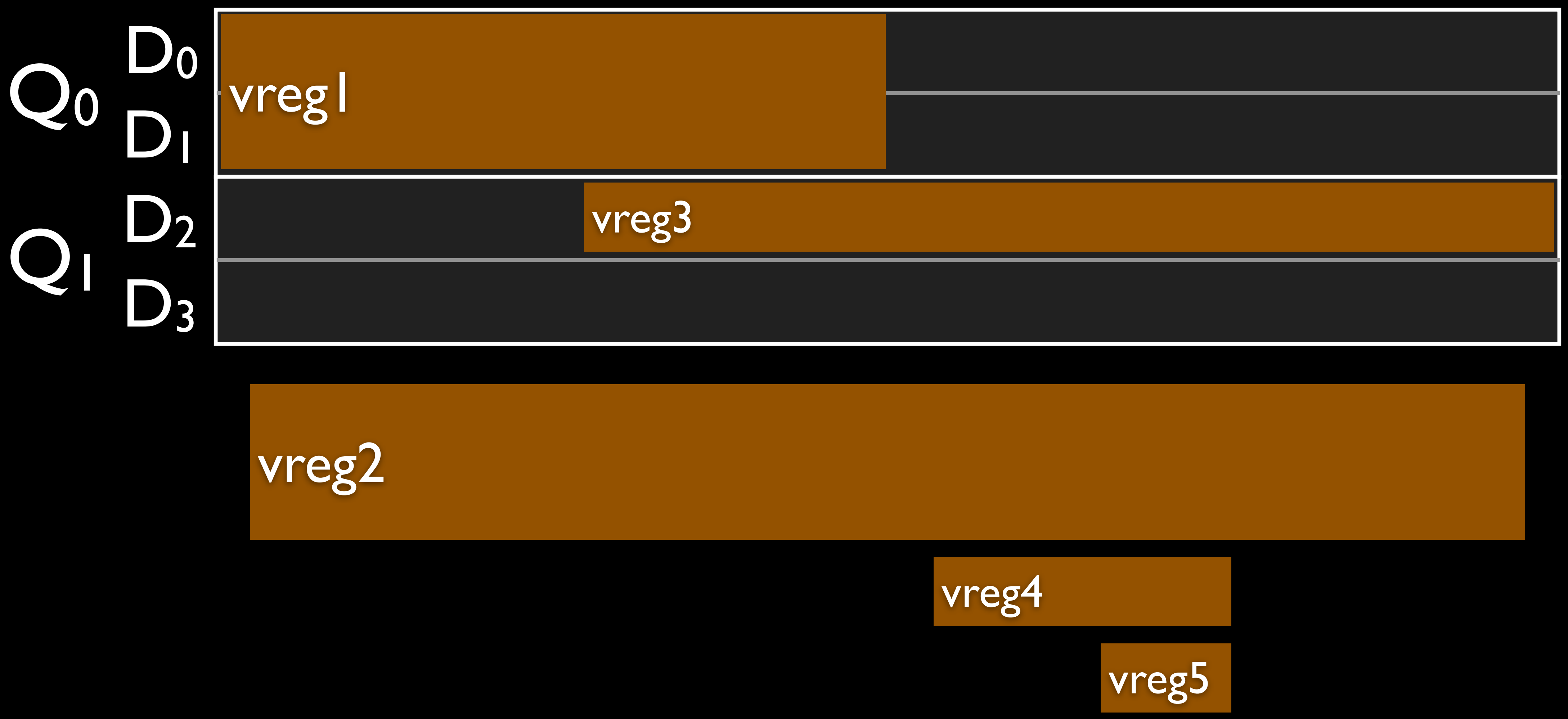


- Order live ranges by size
- Allocate the longest ranges first

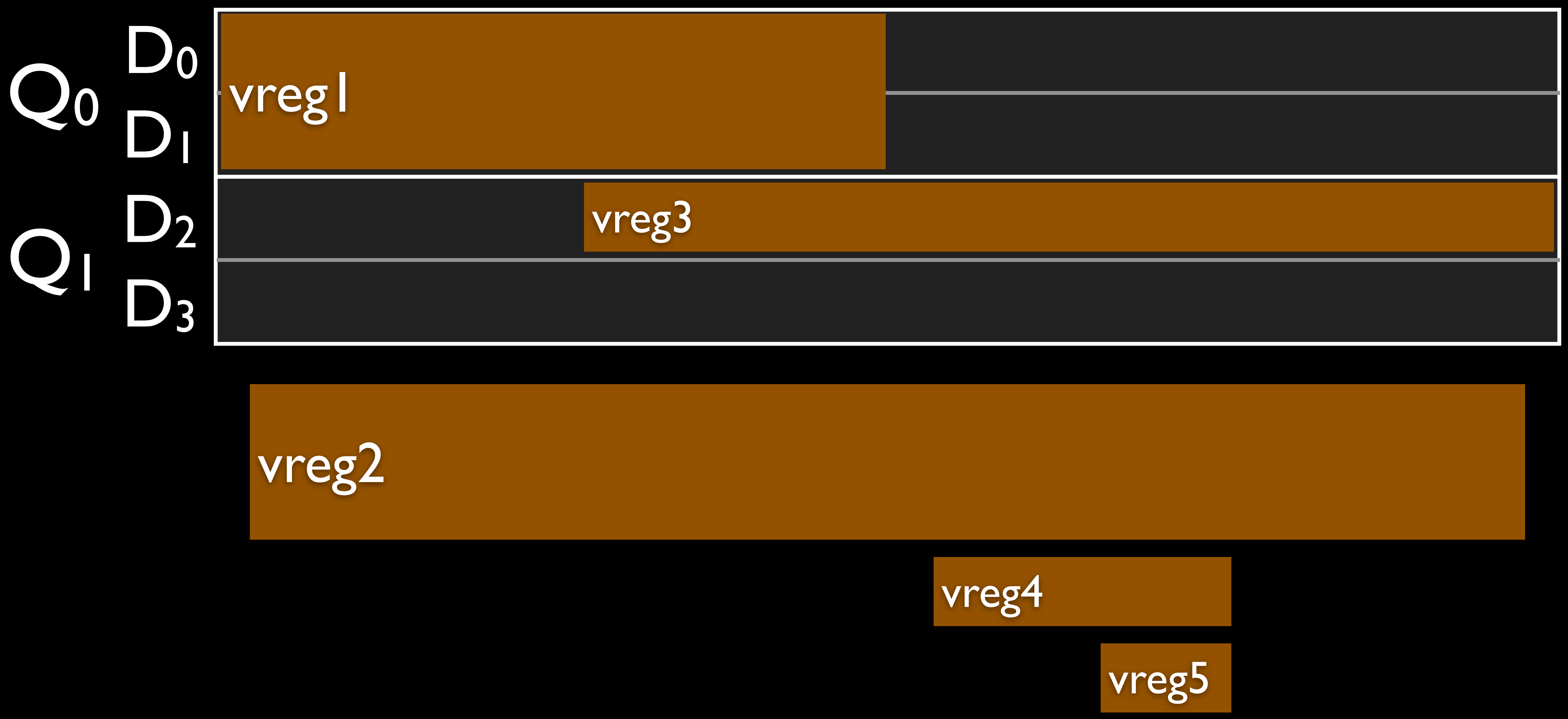
This toy NEON machine has two 128-bit vector registers, each divided into two 64-bit registers.
We are compiling a function with 5 virtual registers.
- First, order the live ranges by size.
- The two longest live ranges allocate right away.



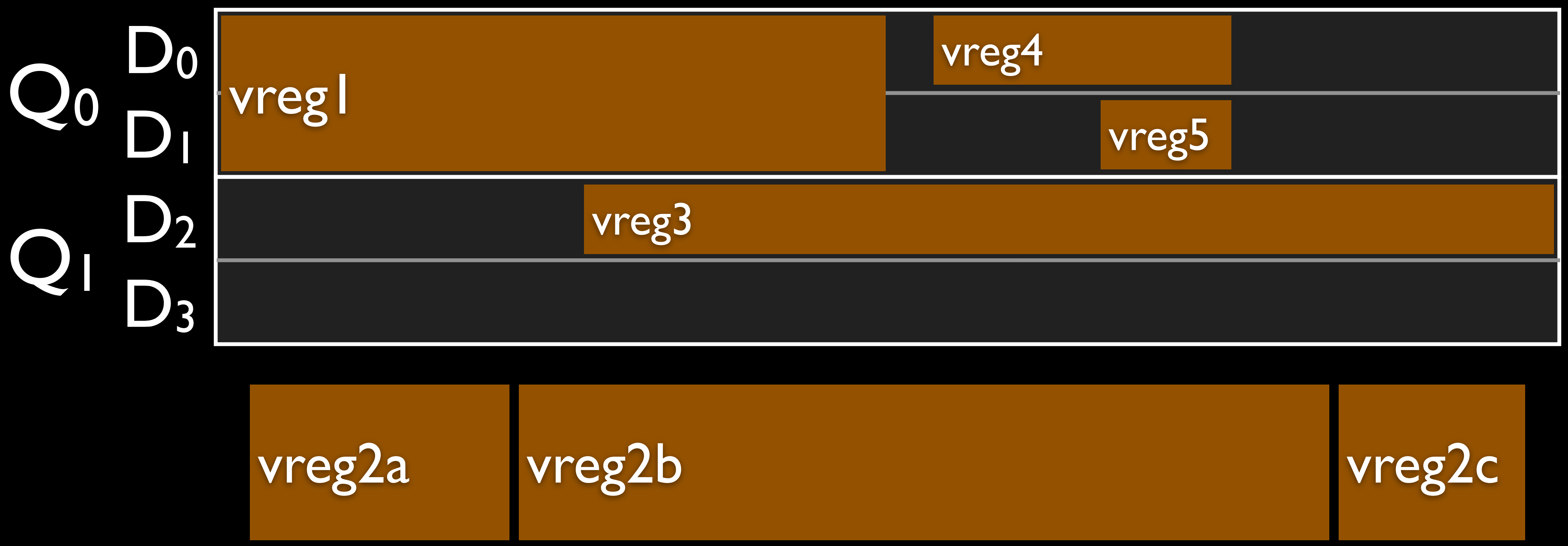
- No room for vreg1
- Evict the smallest spill weight



- At this point linear scan spills
- Much too aggressive

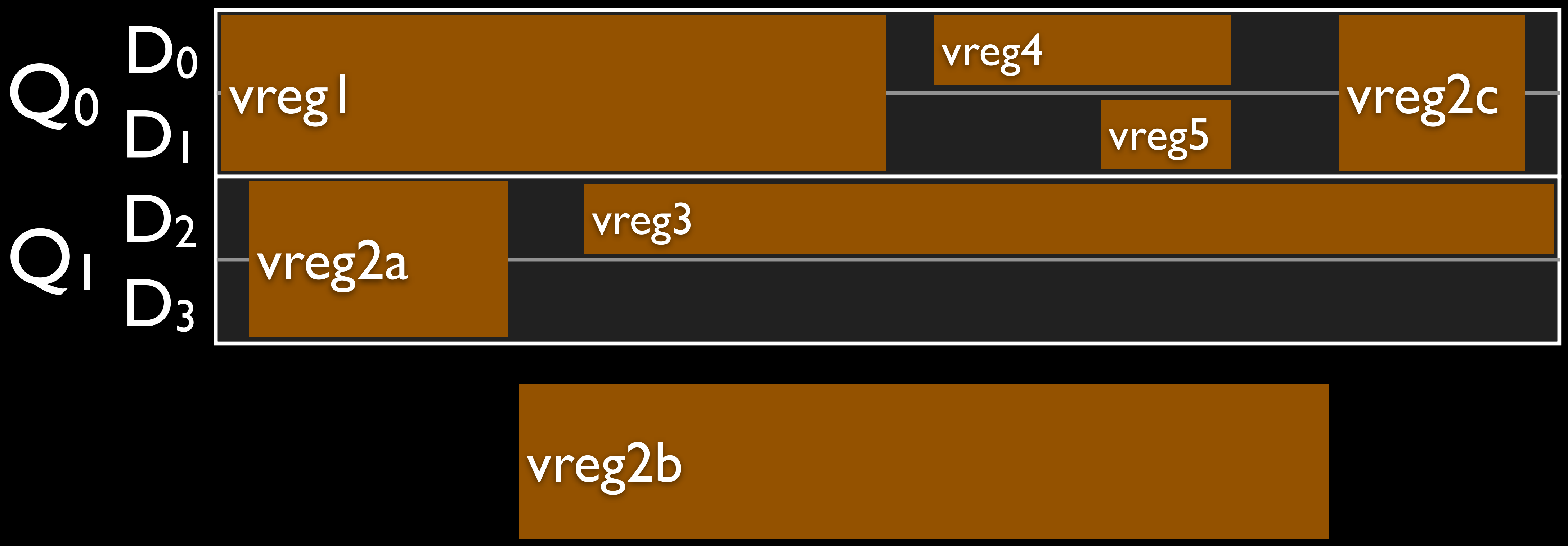


- Save vreg2 for later
- Look for splitting opportunities

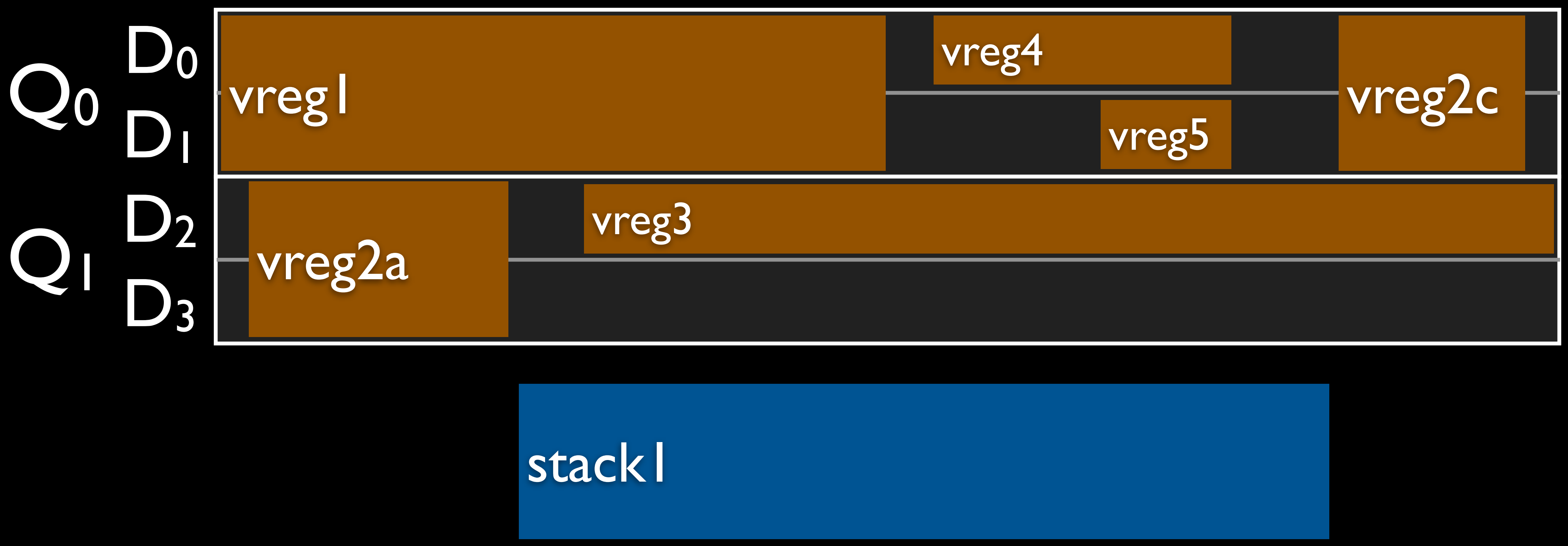


- Split to match available registers
- Allocate known good fragments

Note that the matrix is full, we know the interference when splitting.
- The two ends allocate immediately.



- No lesser spill weight to evict
- Splitting won't help

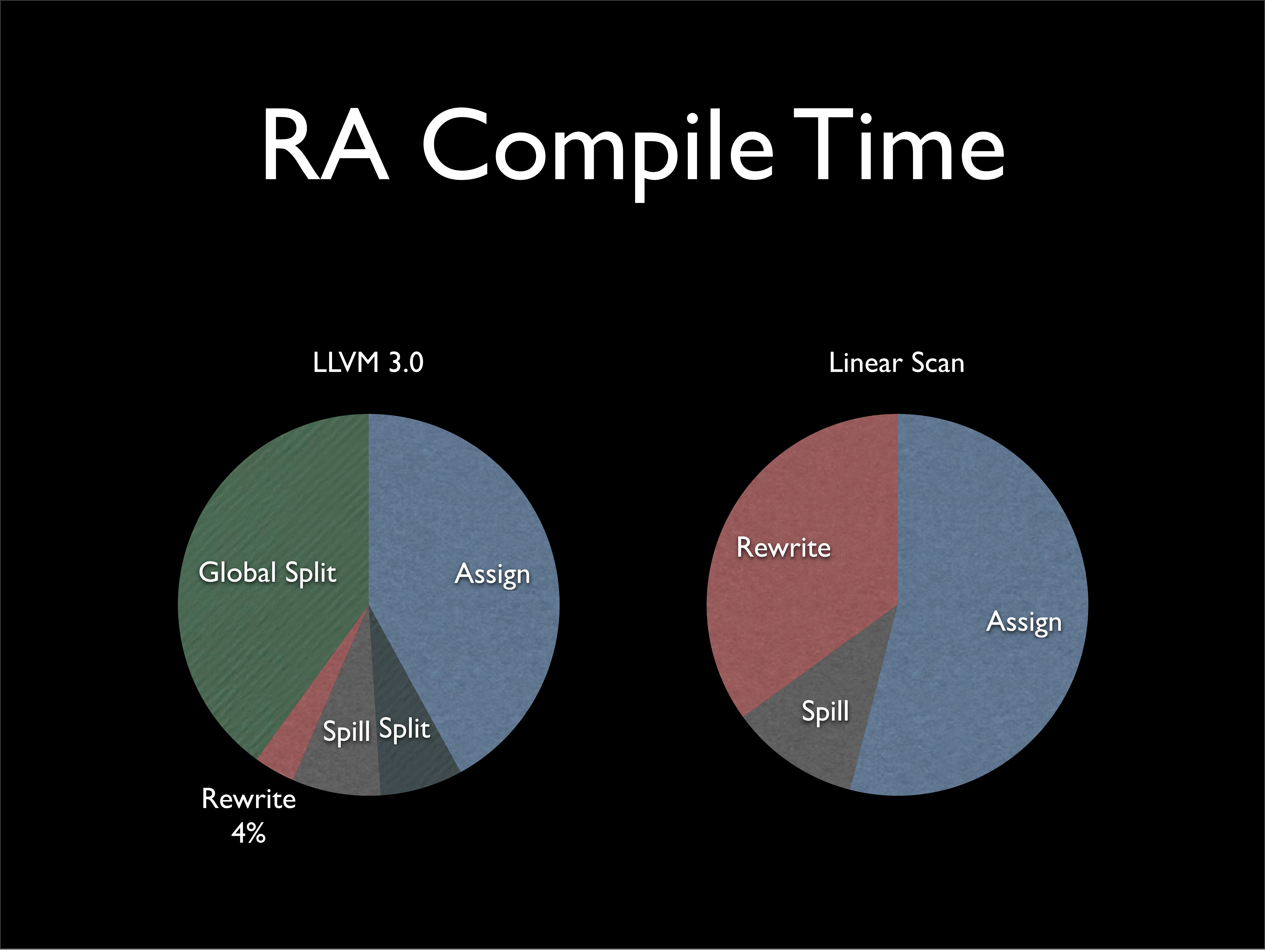


- Spill as a last resort
- All live ranges allocated

LLVM 3.0 Register Allocator

- Any iteration order
- Undo any assignment without backtracking
- Allow arbitrary code changes
- Interference guided live range splitting
- Trivial rewriter

The new algorithm can process live ranges in any order.
Handling long live ranges first gives the best results.

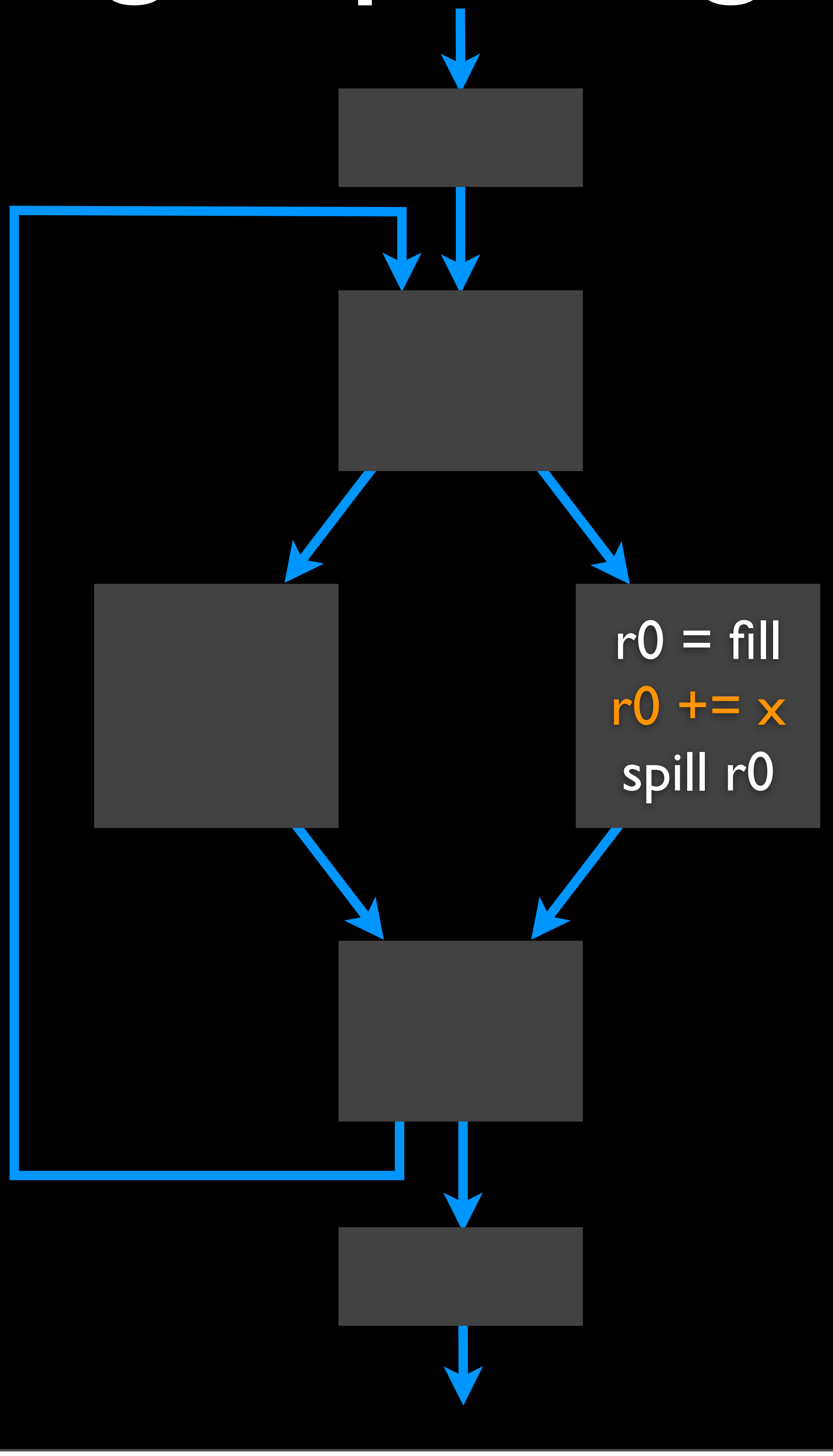


RA Compile Time

Assignment is faster than linear scan, local splitting means less spilling.
- Trivial rewriter is 10x faster.
- We have time for better global live range splitting.

Global Live Range Splitting

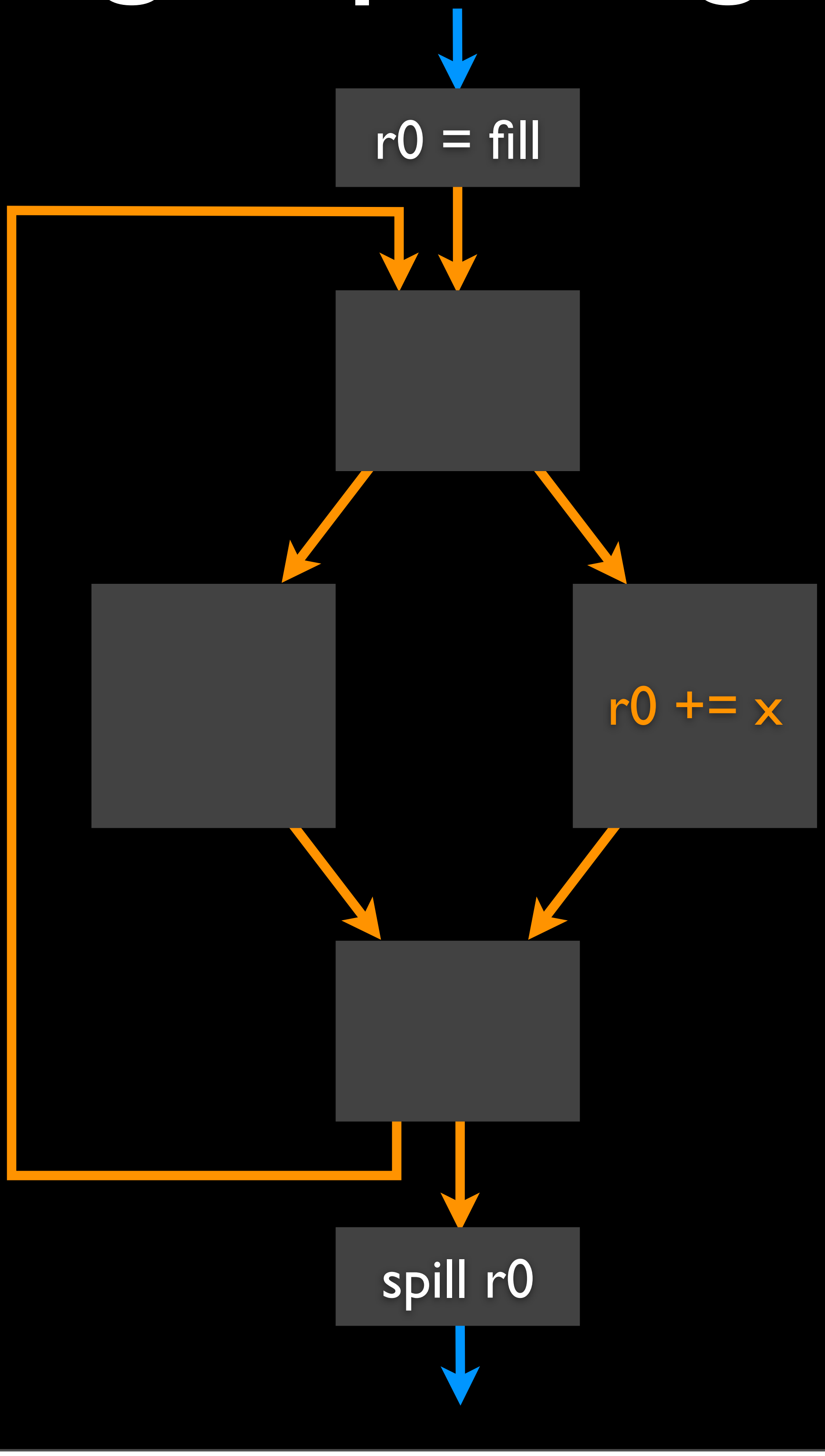
- Loops
- Calls
- Arbitrary regions



Splitting around loops is very important.
- We want to move the spill/fill instructions outside the loop.

Global Live Range Splitting

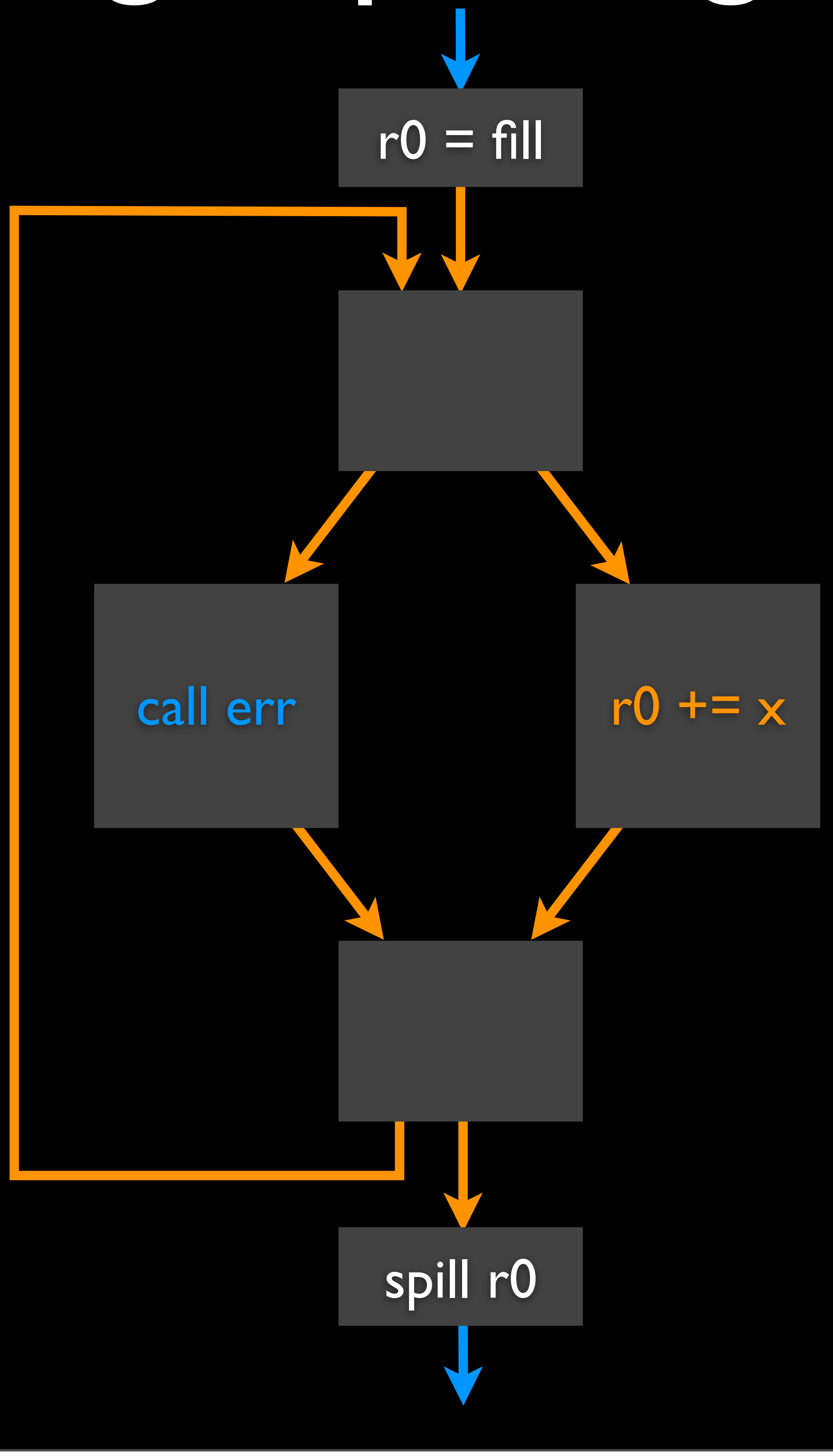
- Loops
- Calls
- Arbitrary regions



It reduces the number of executed spill/fill instructions.

Global Live Range Splitting

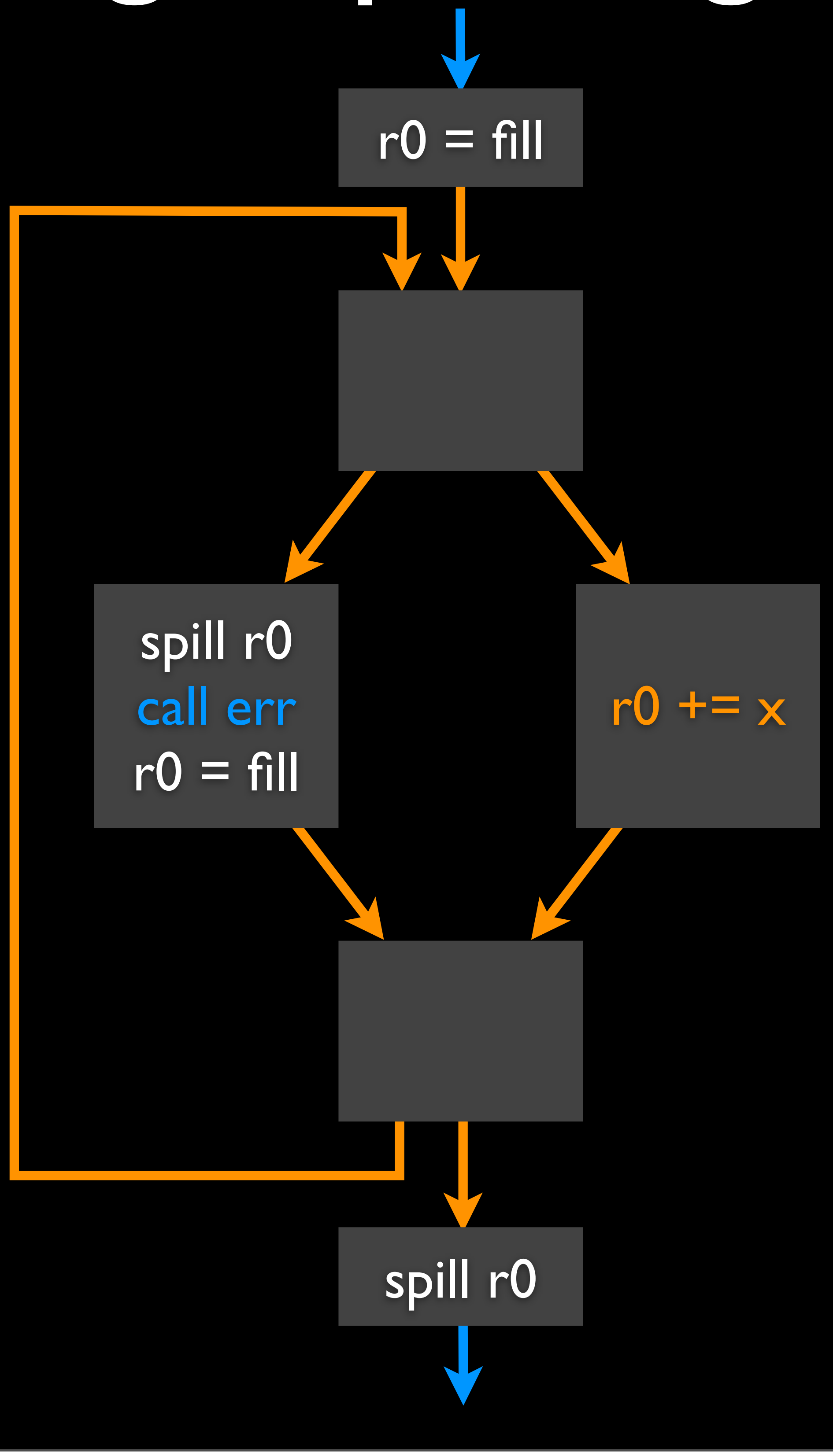
- Loops
- Calls
- Arbitrary regions



If a live range crosses a cold call, we don’t want to affect other code.
– Just spill around the call.

Global Live Range Splitting

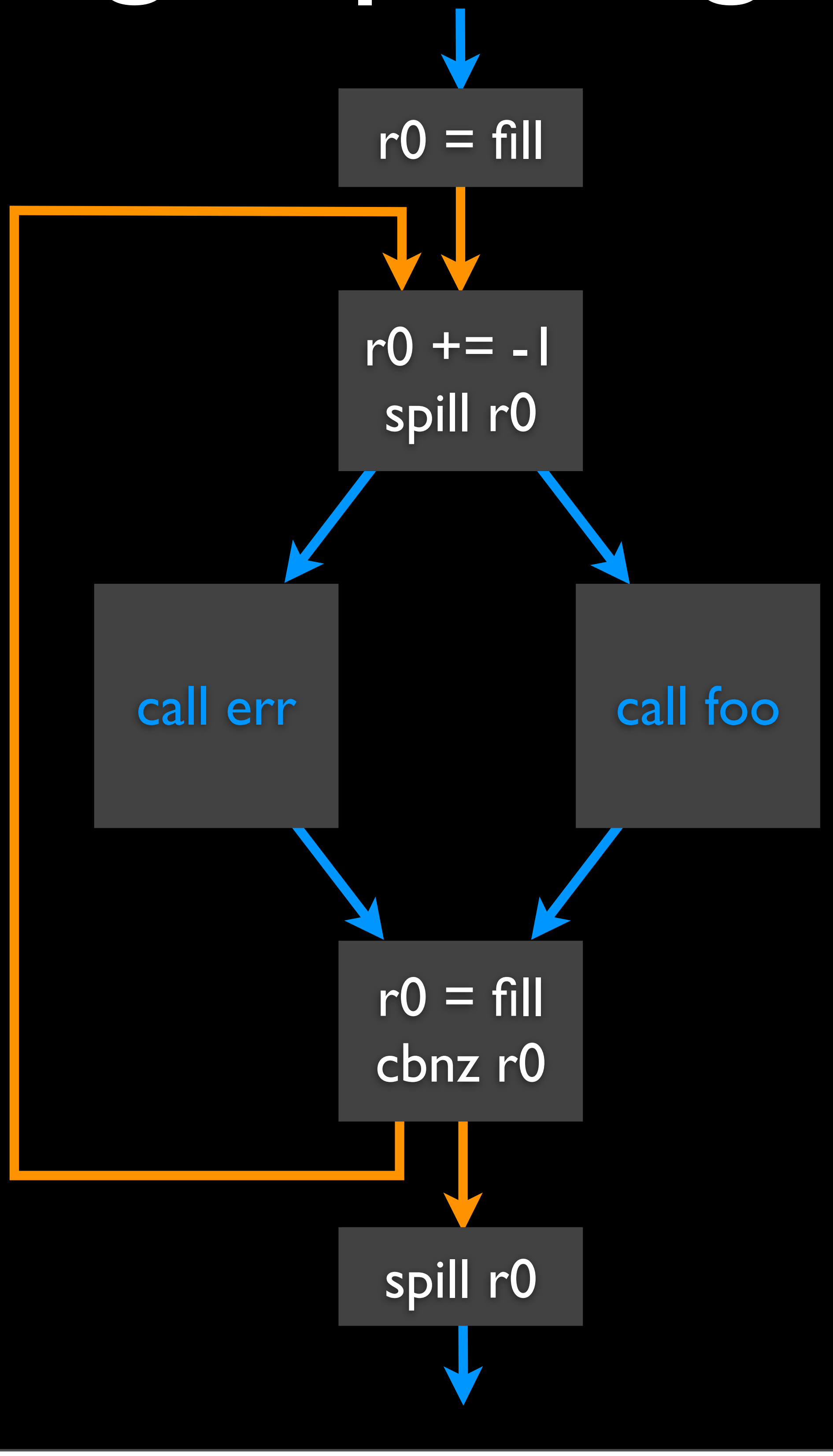
- Loops
- Calls
- Arbitrary regions



If a live range crosses a call, we don’t want to affect other code.
Just spill around the call.

Global Live Range Splitting

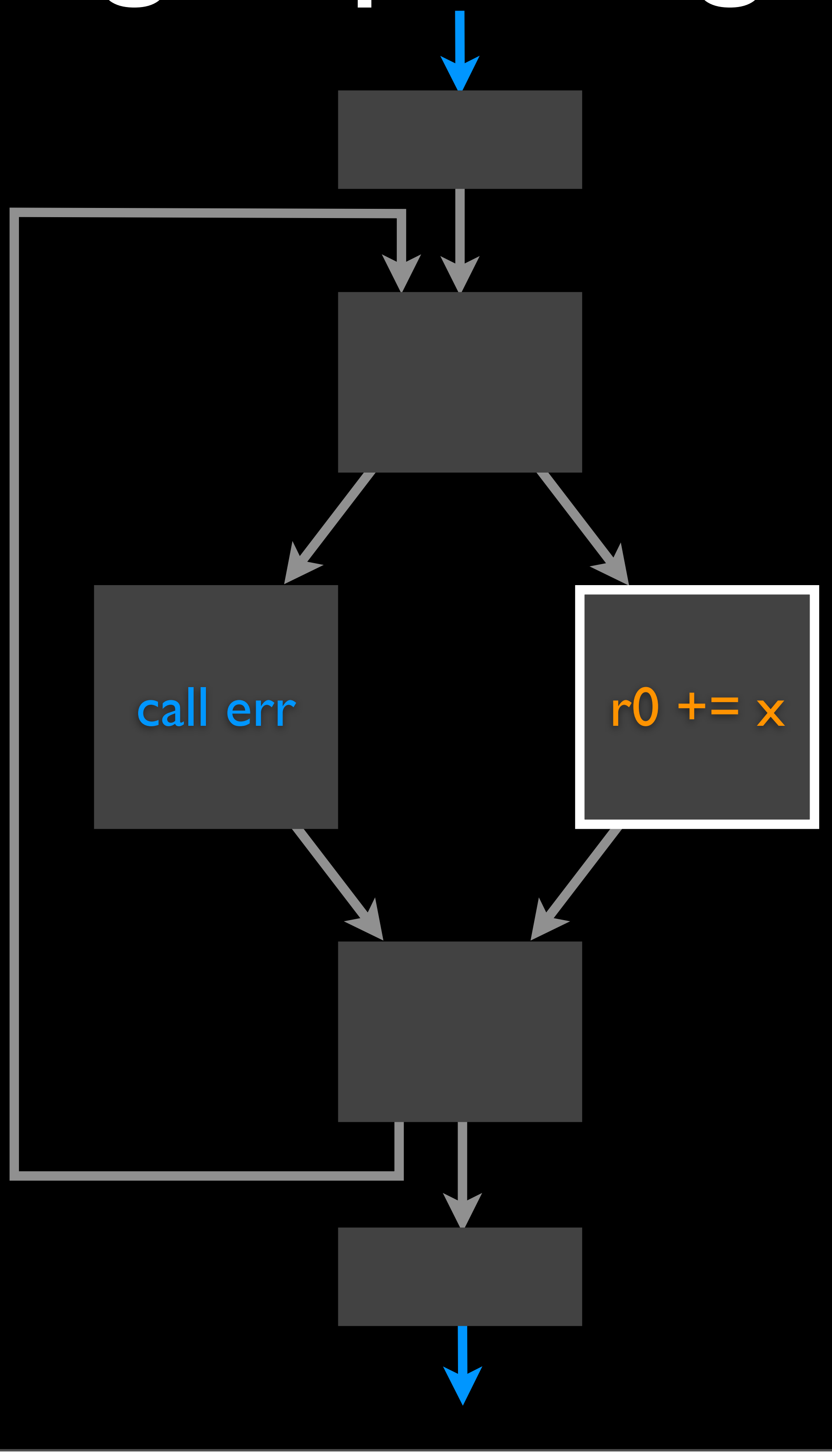
- Loops
- Calls
- Arbitrary regions



In a more complicated loop, we may need more complicated regions.
We don’ want to be limited to just splitting around loops.

Global Live Range Splitting

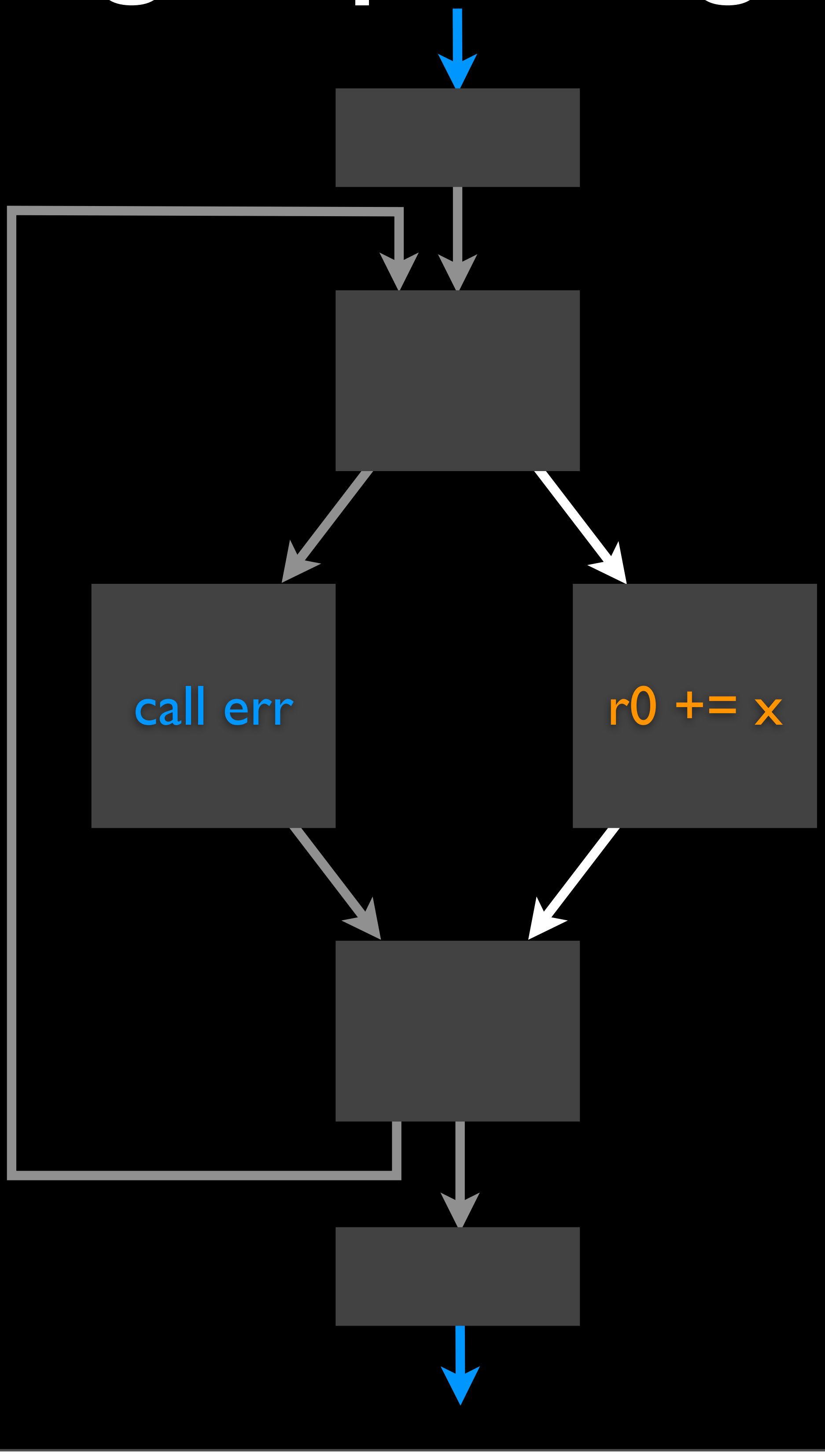
- Start from uses
- Grow region
- Insert spill code



How are regions formed?
Start from the instructions using the live range.
We want to grow the region until we find a cheap place to spill.

Global Live Range Splitting

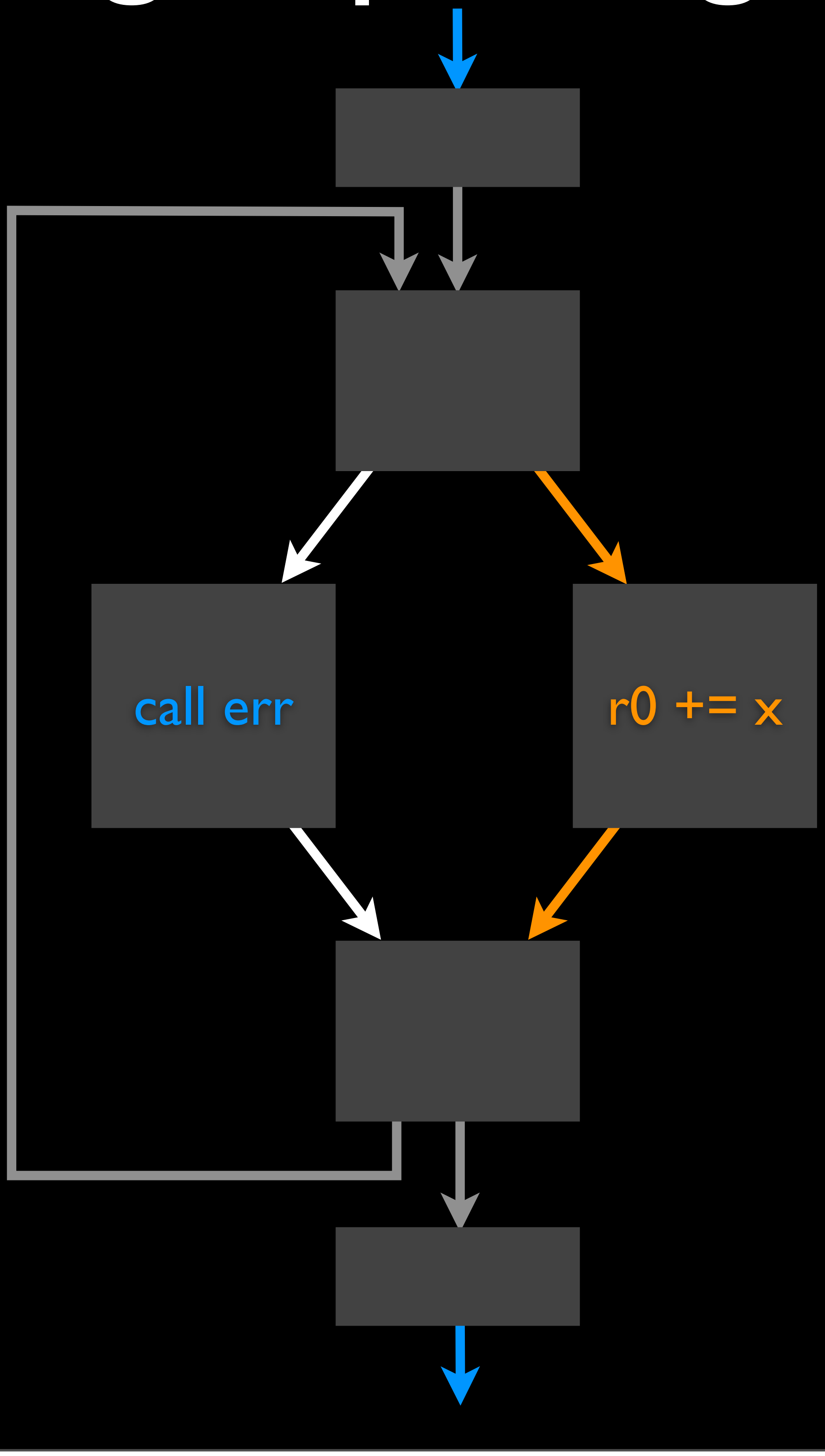
- Start from uses
- Grow region
- Insert spill code



We want the value to be live-in and live-out in a register on these edges.

Global Live Range Splitting

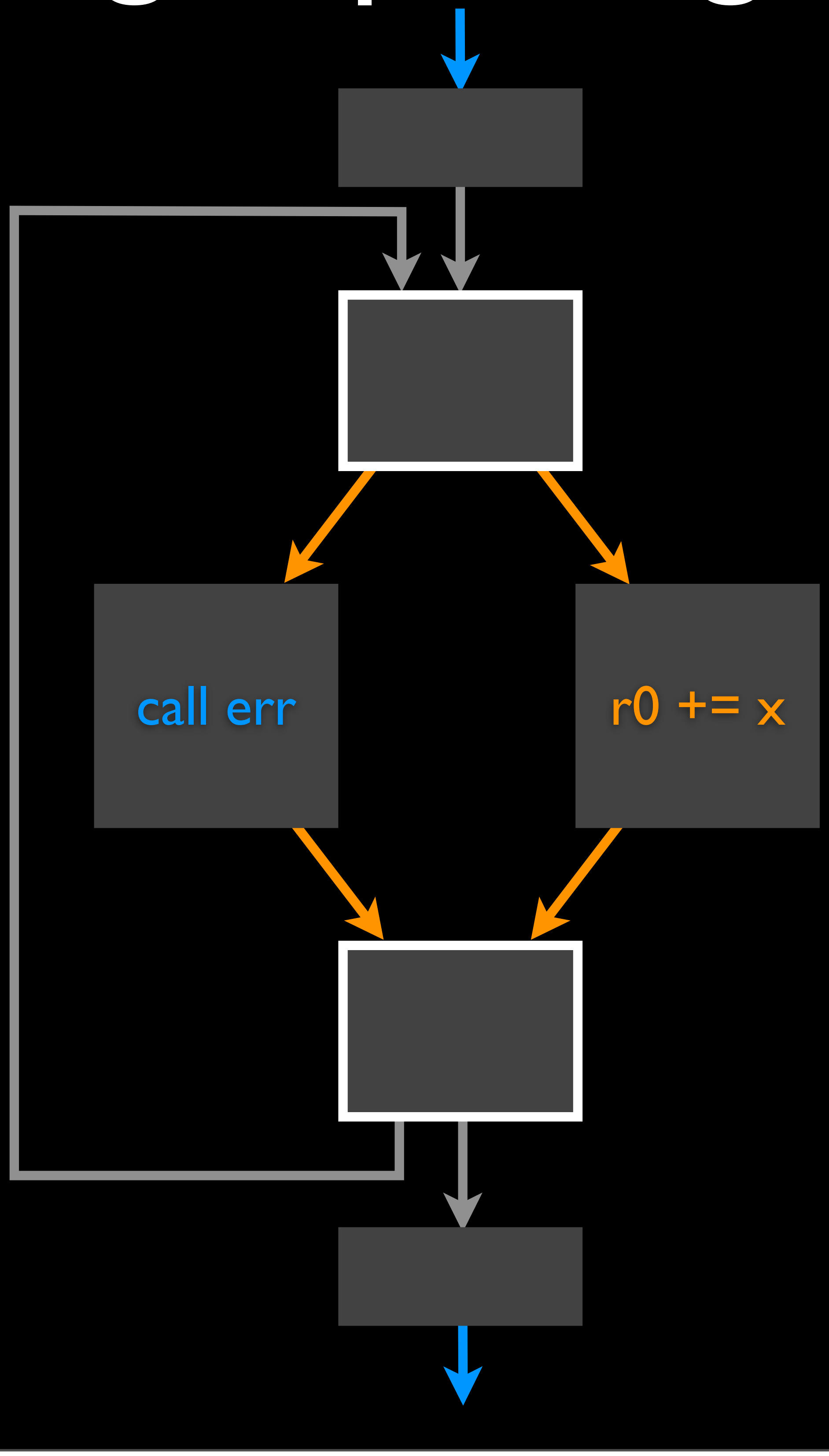
- Start from uses
- Grow region
- Insert spill code



But then these edges must also be live.

Global Live Range Splitting

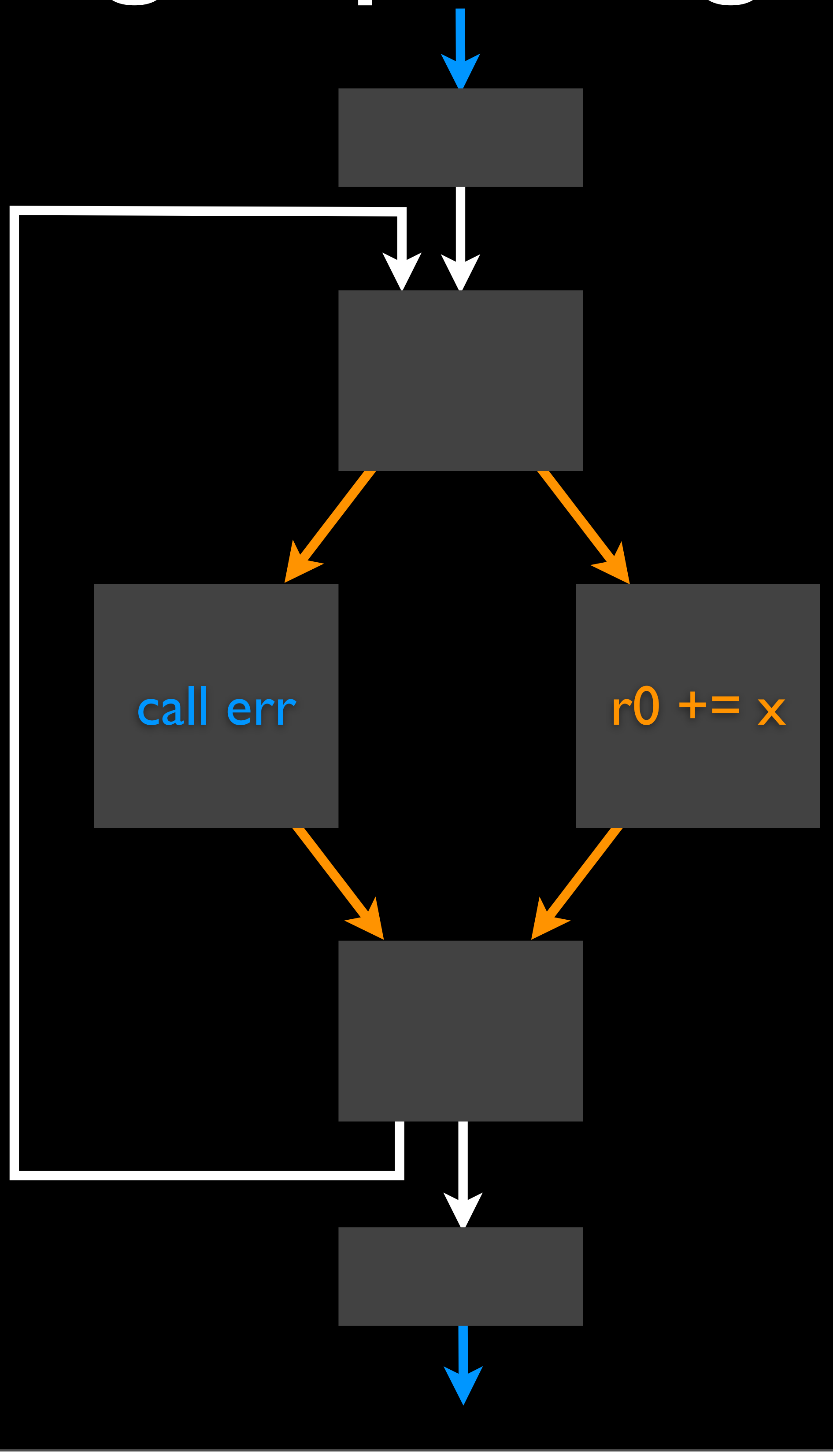
- Start from uses
- Grow region
- Insert spill code



This activates new blocks.

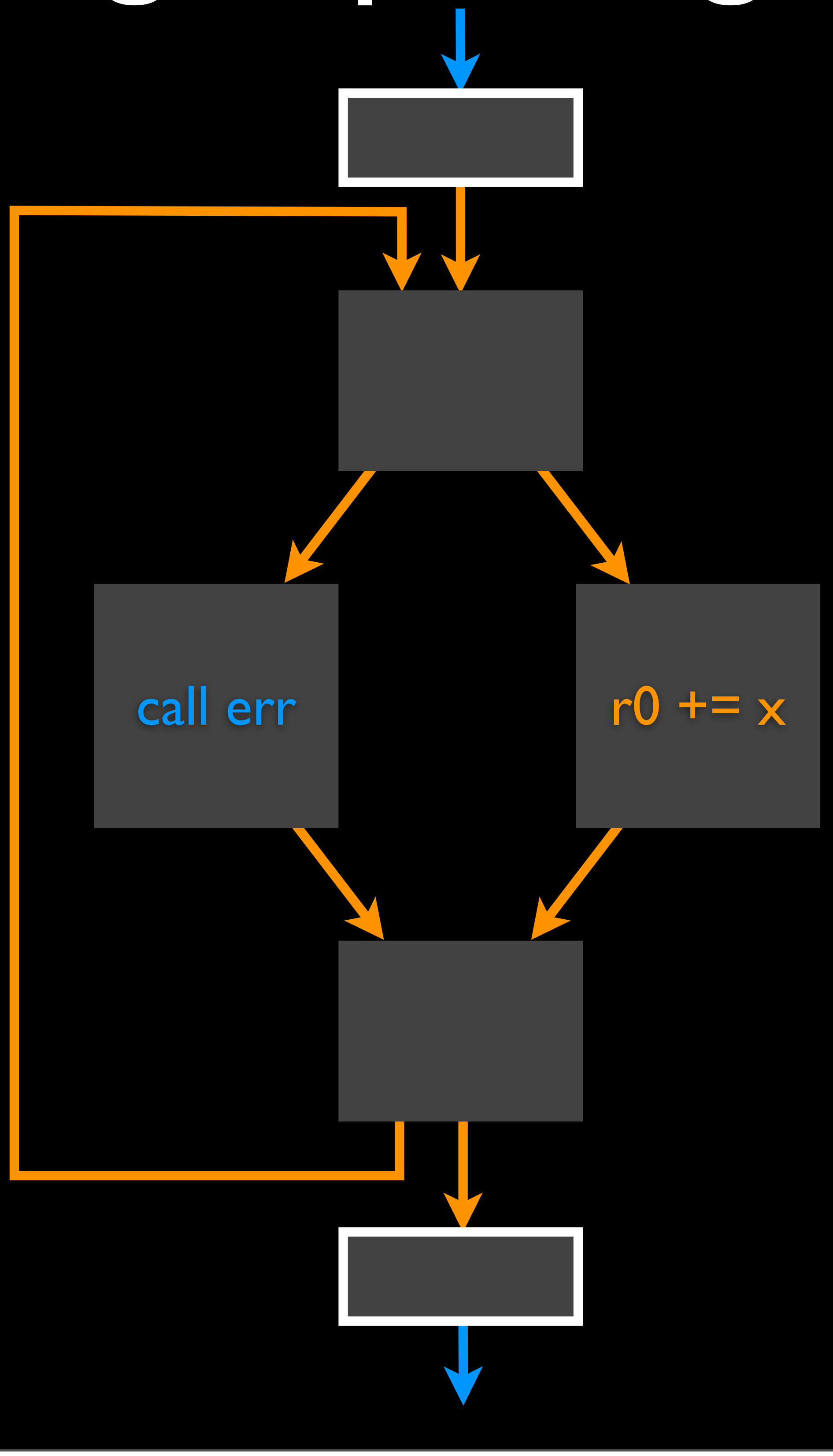
Global Live Range Splitting

- Start from uses
- Grow region
- Insert spill code



Global Live Range Splitting

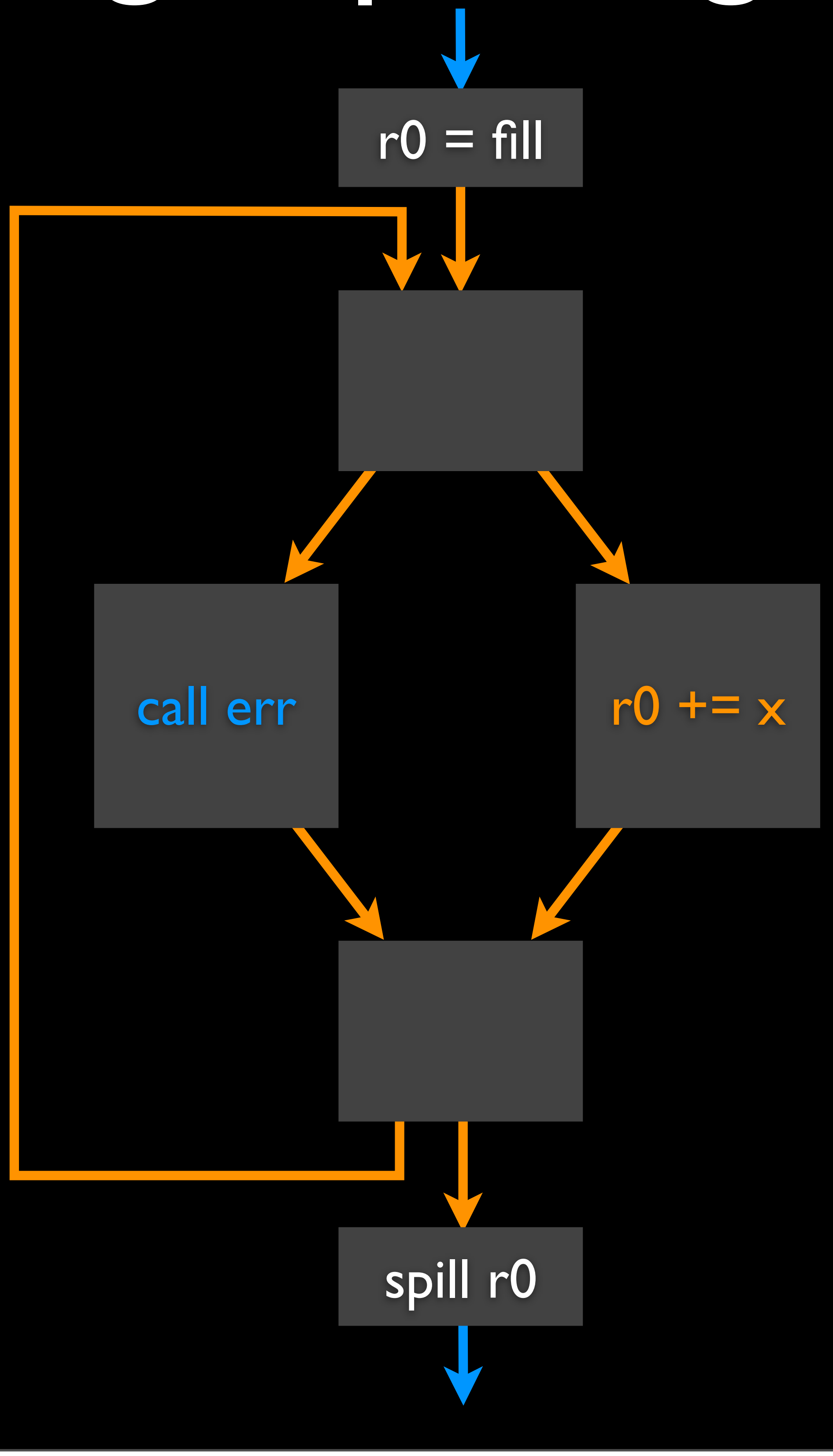
- Start from uses
- Grow region
- Insert spill code



This activates new live-through blocks.

Global Live Range Splitting

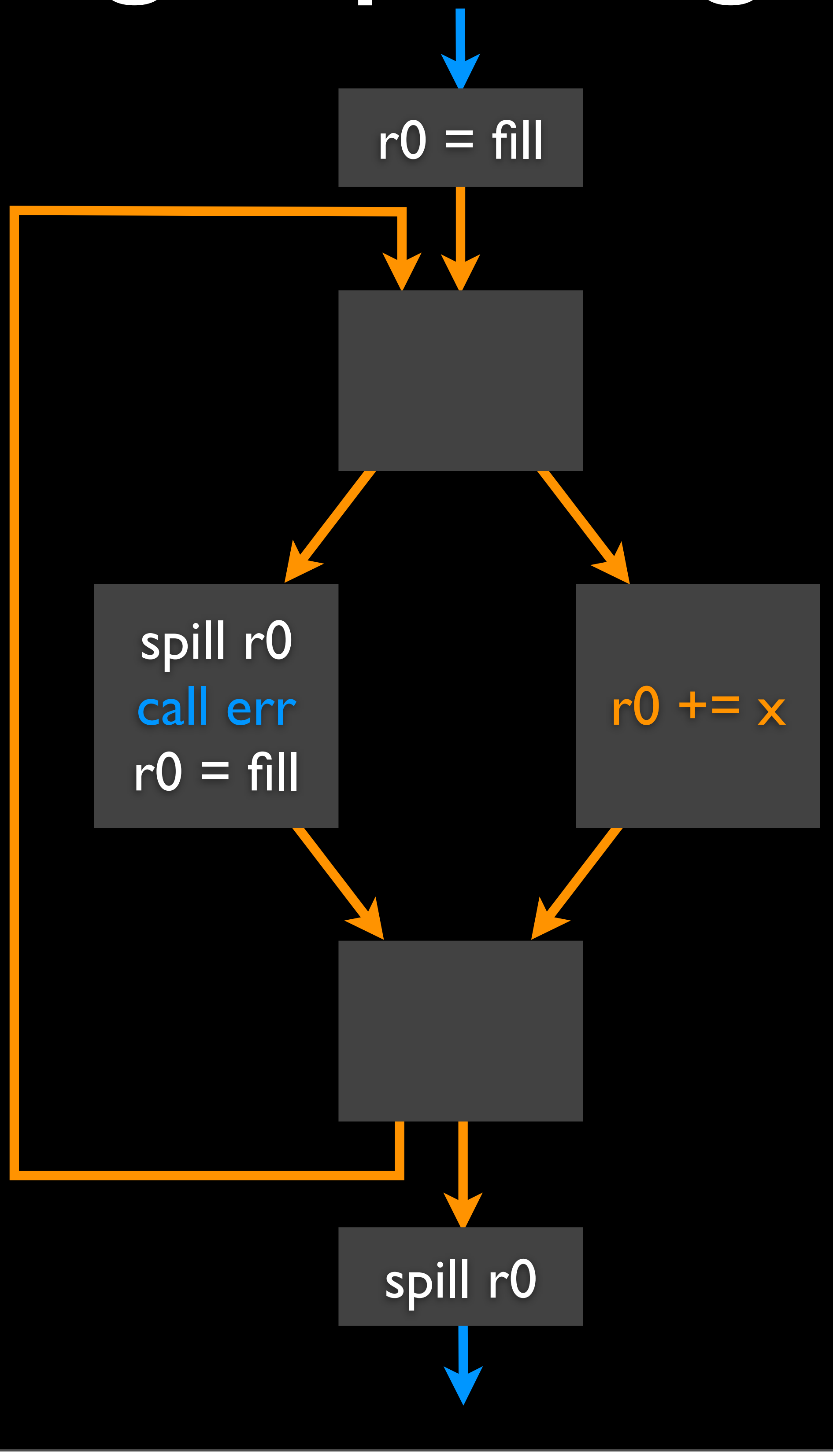
- Start from uses
- Grow region
- Insert spill code



Insert spills and fills in the live-through blocks.

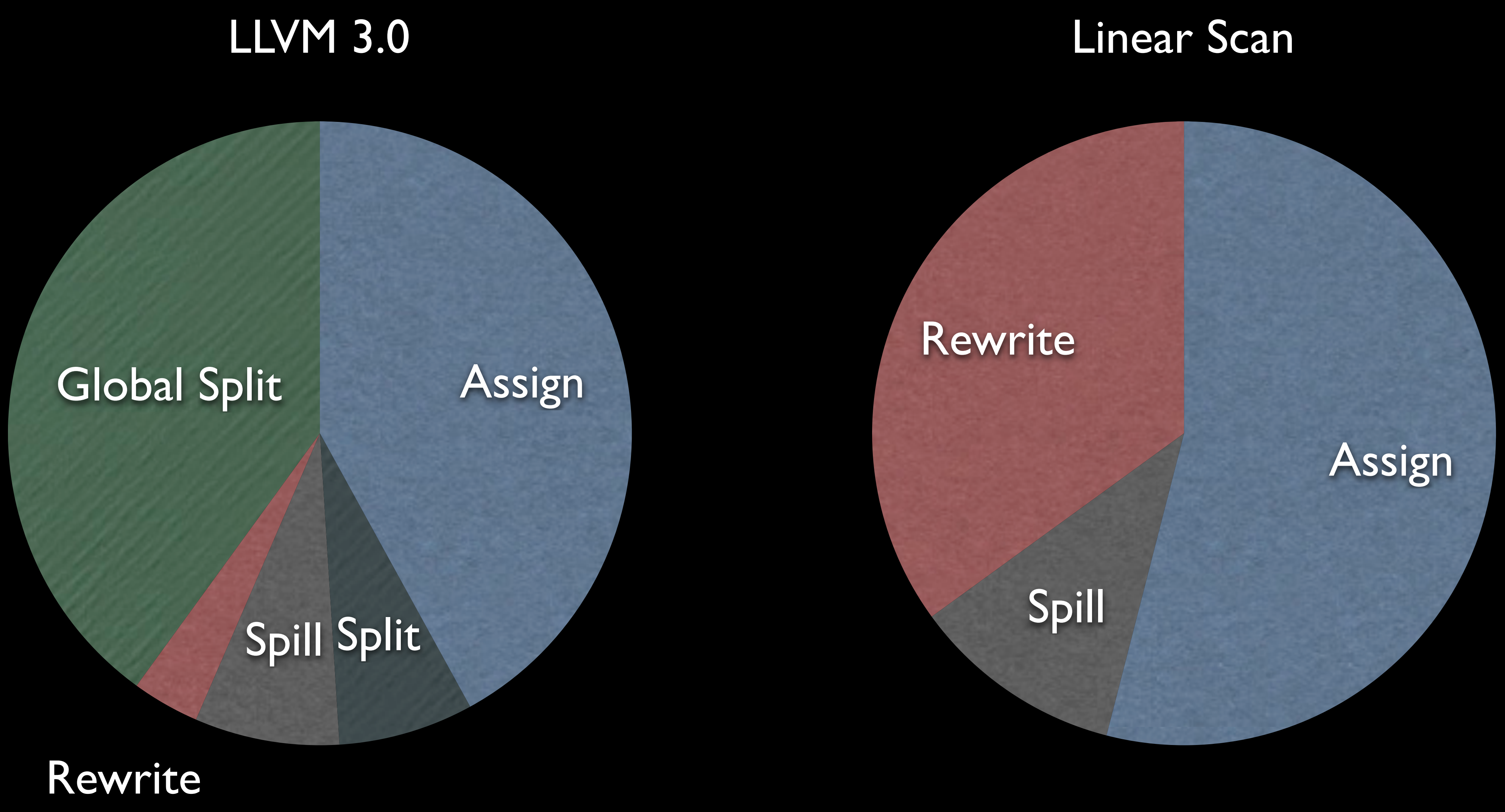
Global Live Range Splitting

- Start from uses
- Grow region
- Insert spill code

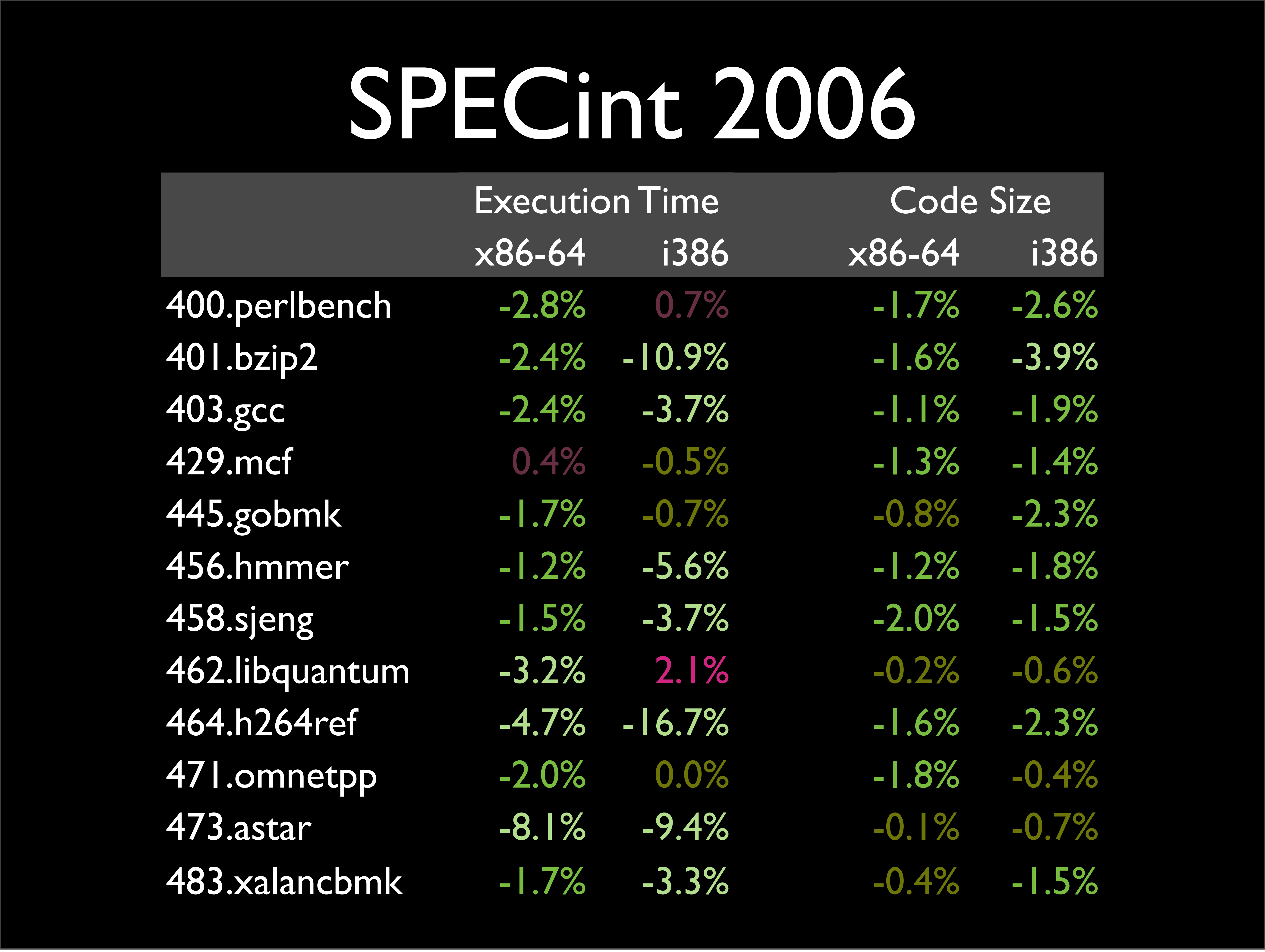


Insert internal spills and fills for blocks with interference.

Same Compile Time



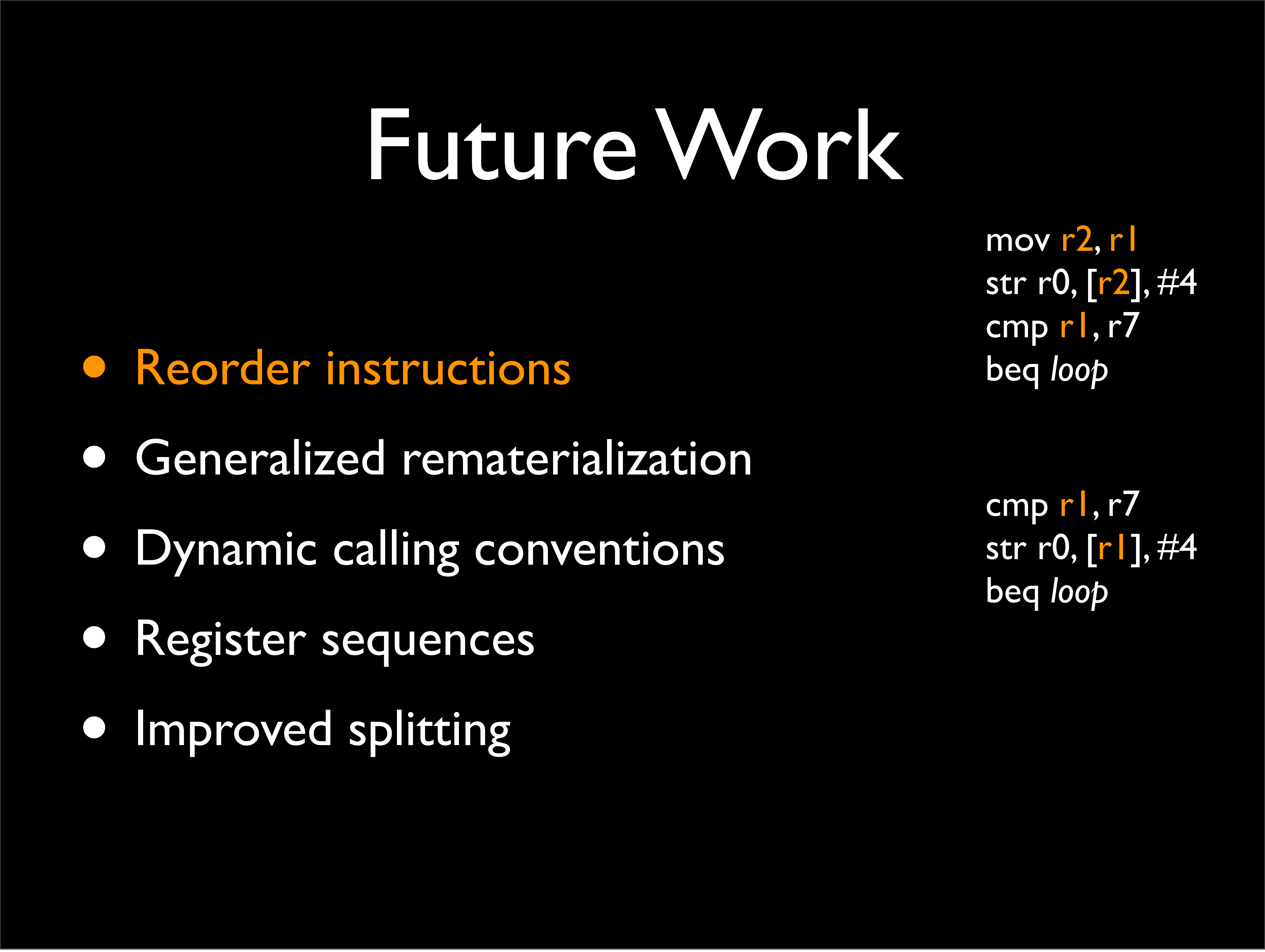
Global splitting is around 40% of the total compile time.
The LLVM 3.0 register allocator runs in the same compile time as linear scan.



SPECint 2006

	Execution Time		Code Size	
	x86-64	i386	x86-64	i386
400.perlbench	-2.8%	0.7%	-1.7%	-2.6%
401.bzip2	-2.4%	-10.9%	-1.6%	-3.9%
403.gcc	-2.4%	-3.7%	-1.1%	-1.9%
429.mcf	0.4%	-0.5%	-1.3%	-1.4%
445.gobmk	-1.7%	-0.7%	-0.8%	-2.3%
456.hmmer	-1.2%	-5.6%	-1.2%	-1.8%
458.sjeng	-1.5%	-3.7%	-2.0%	-1.5%
462.libquantum	-3.2%	2.1%	-0.2%	-0.6%
464.h264ref	-4.7%	-16.7%	-1.6%	-2.3%
471.omnetpp	-2.0%	0.0%	-1.8%	-0.4%
473.astar	-8.1%	-9.4%	-0.1%	-0.7%
483.xalancbmk	-1.7%	-3.3%	-0.4%	-1.5%

Comparing linear scan to the LLVM 3.0 register allocator.
Global live range splitting gives nearly universal improvements.
Both execution time and code size is improved.
The i386 architecture is more affected because it only has 8 registers.



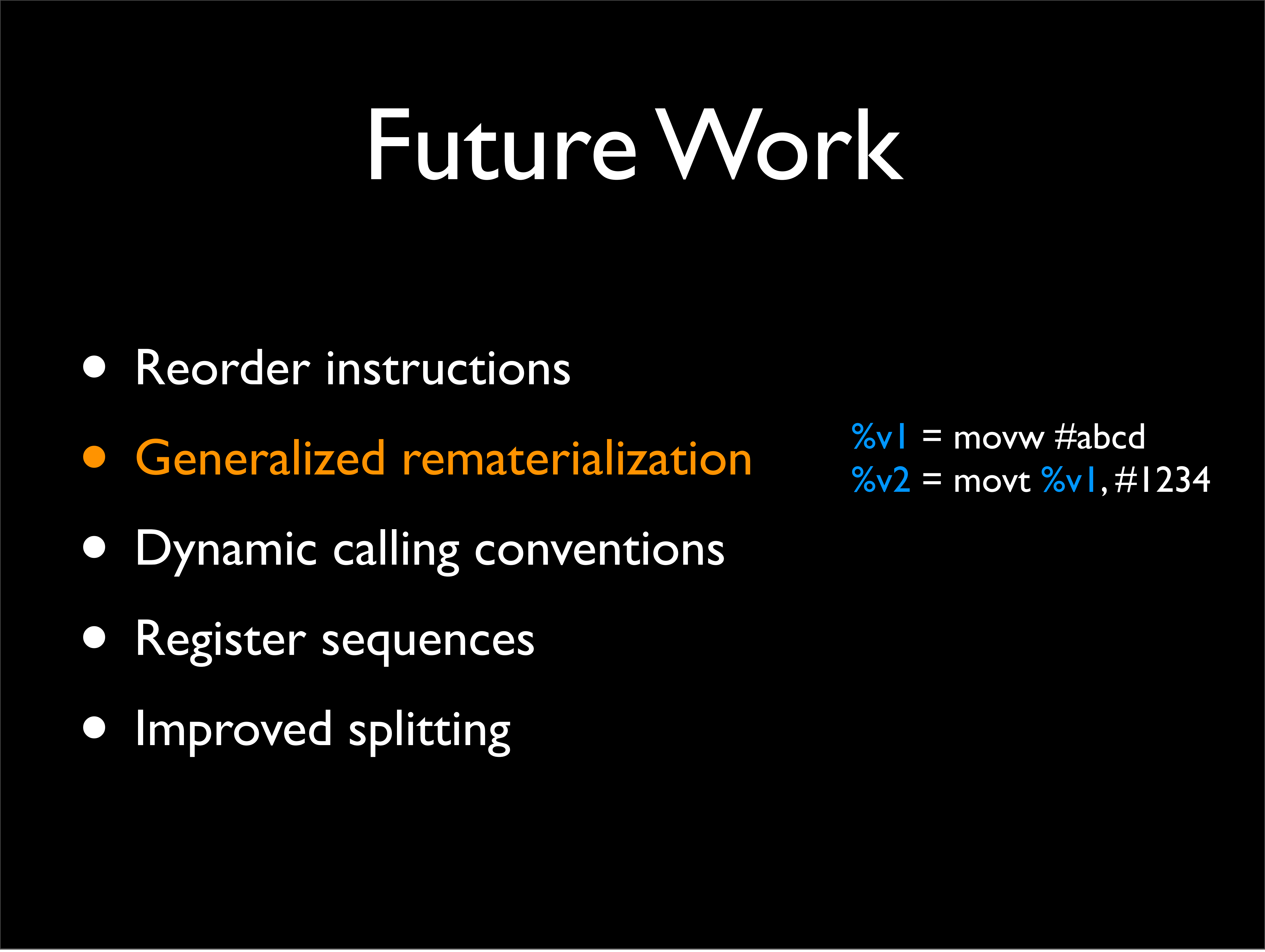
Future Work

- Reorder instructions
- Generalized rematerialization
- Dynamic calling conventions
- Register sequences
- Improved splitting

```
mov r2, r1
str r0, [r2], #4
cmp r1, r7
beq loop
```

```
cmp r1, r7
str r0, [r1], #4
beq loop
```

Reorder and perhaps reassociate instructions.



Future Work

- Reorder instructions
- Generalized rematerialization
- Dynamic calling conventions
- Register sequences
- Improved splitting

```
%v1 = movw #abcd
%v2 = movt %v1, #1234
```

Rematerialize multiple chained instructions.

Future Work

- Reorder instructions
- Generalized rematerialization
- Dynamic calling conventions
- Register sequences
- Improved splitting

coldcall
GHC

Handle multiple calling conventions at once.

Future Work

- Reorder instructions
- Generalized rematerialization
- Dynamic calling conventions
- Register sequences
- Improved splitting

```
vld1 {d1,d2,d3}, [r1]
```

Future Work

- Reorder instructions
- Generalized rematerialization
- Dynamic calling conventions
- Register sequences
- Improved splitting

LLVM 3.0 Register Allocator

Region-based

Priority-based?

Global

Greedy?

LLVM 3.0 Register Allocator

- More flexible than linear scan
- Enables future improvements
- Global live range splitting
- Generates faster, smaller code
- Same compile time

http://blog.llvm.org/2011/09/greedy-register-allocation-in-llvm-30.html

Questions?