EECS 665

COMPILER CONSTRUCTION

Machine Code Optimization

Previously
Post-compilation toolchain

Compiler Toolchains

- Overview

- What GCC Does
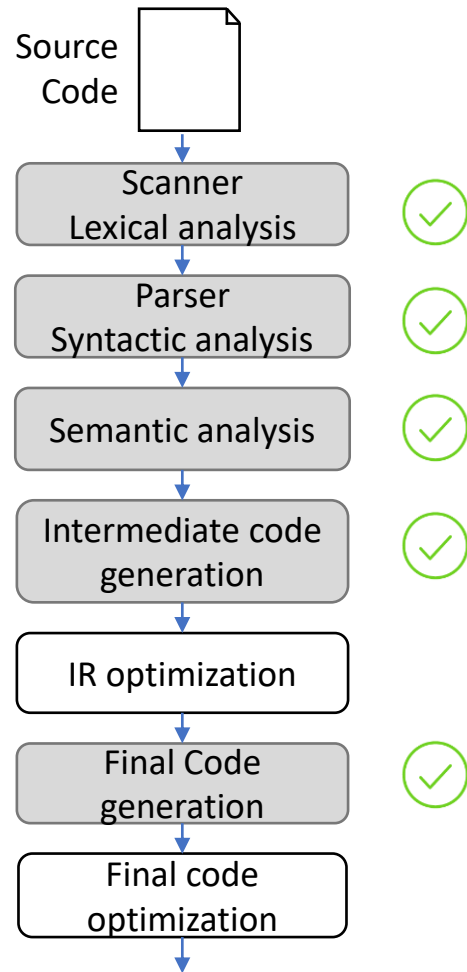
Component Walkthrough

- Assembler

- Linker

- Loader

Compiler Construction
Progress Pics

Source Code

Scanner Lexical analysis	✓
Parser Syntactic analysis	✓
Semantic analysis	✓
Intermediate code generation	✓
IR optimization	
Final Code generation	✓
Final code optimization	

Finished

- A naïve workflow from source code to target code

To-Do

- Clean up some of the corners we cut

We've focused on correctness over efficiency, let's try to win back some efficiency

Today's Outline
Machine Code Optimization

Overview

Improving data allocation

- Register allocation

Improving Final Code

- Peephole optimization

- Instruction Pipelines

Optimization

Working With the Architecture
Machine Code Optimization

Good machine code should:

• Play to the strengths of the hardware

• Compensate for weaknesses of the hardware

Such operations depend on specifics of the architecture

Disclaimer: This is a Deep Area

Machine Code Optimization

We hardly scratch the surface of compiler optimizations

- There are more categories of machine-code optimization than we'll cover

- There are more optimizations within the categories we do cover

The tip of the iceberg

Today's Outline
Machine Code Optimization

Overview

Improving data allocation

- Register allocation

Improving Final Code

- Peephole optimization

- Instruction Pipelines

Optimization

Sizing Activation Records

Machine Code Optimization: Data Allocation

Easy mode: one AR slot for every temp, local, and arg
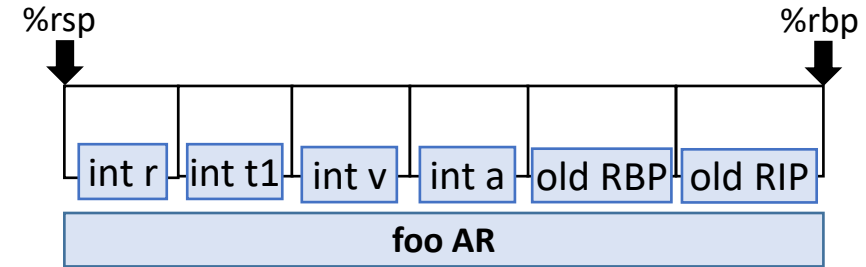
Source code
```
int foo(int a){
    int v;
    int r;
    v = a-a*a;
    r = v - 2;
}
```
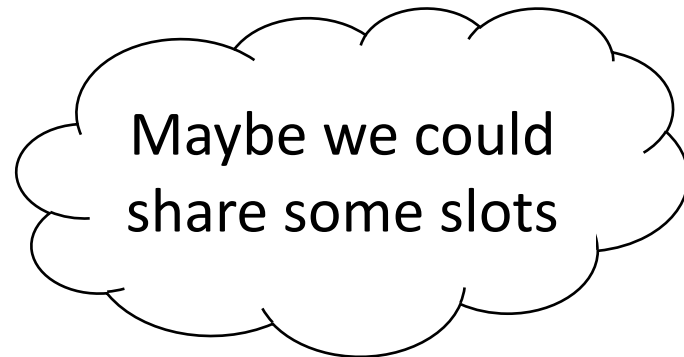
3AC Code
```
enter foo
getarg 1, [a]
[t1] := [a] * [a]
[v] := [a] - [t1]
[r] := [v] - 2
```

Memory at Runtime

%rsp   %rbp

| int r | int t1 | int v | int a | old RBP | old RIP |

foo AR

Surely one could use fewer memory slots!
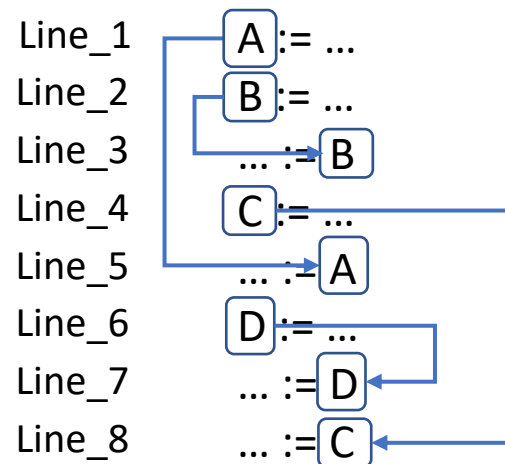
Maybe we could
share some slots

Liveness
Machine Code Optimization: Data Allocation

- A definition is **live** if it's value is subsequently used

- **Insight:** Variables can share space when they don't *interfere* (i.e. aren't simultaneously live)

- We'll capture the constraints via an abstraction called the **interference graph**
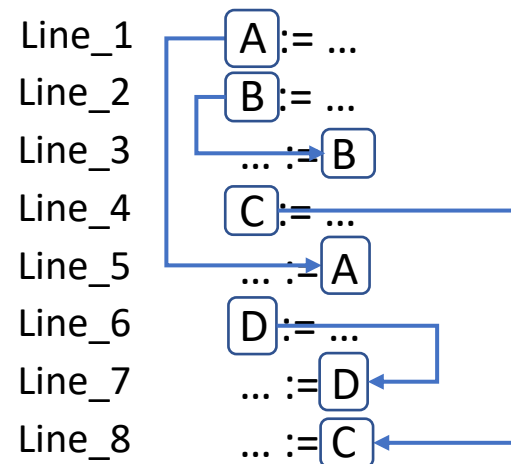
Use/Definition Sequence

Line_1   A := ...
Line_2   B := ...
Line_3   ... := B
Line_4   C := ...
Line_5   ... := A
Line_6   D := ...
Line_7   ... := D
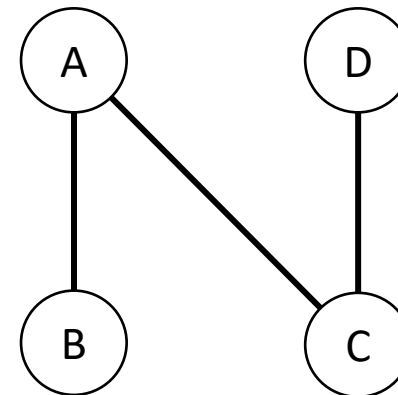Line_8   ... := C

Machine Code Optimization: Data Allocation

The interference graph:

- Nodes are variables

- Edges show simultaneously live variables
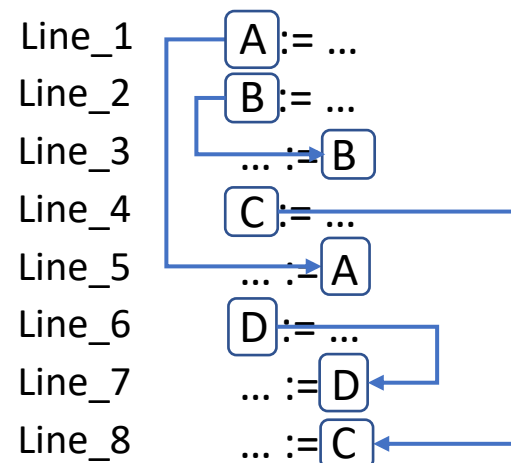
Use/Definition Sequence

Line_1   A := ...
Line_2   B := ...
Line_3   ... := B
Line_4   C := ...
Line_5   ... := A
Line_6   D := ...
Line_7   ... := D
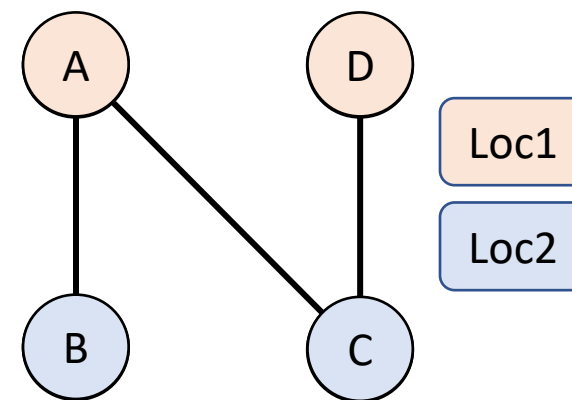Line_8   ... := C

Interference Graph

Liveness

Coloring:

- Assign each storage location to a color
- Color the interference graph so no nodes of the same color are adjacent
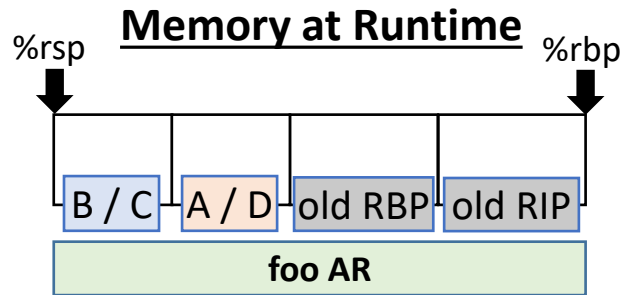
Use/Definition Sequence

Line_1   A := ...
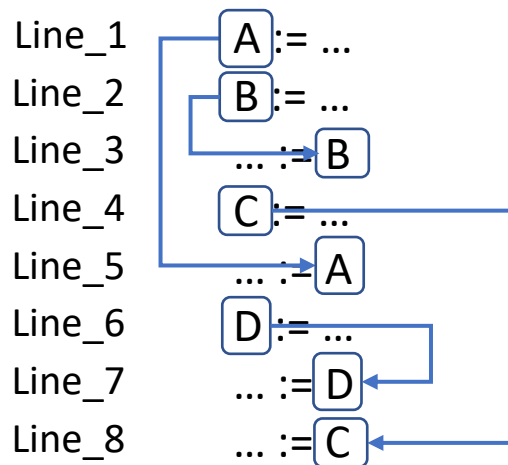Line_2   B := ...
Line_3   ... := B
Line_4   C := ...
Line_5   ... := A
Line_6   D := ...
Line_7   ... := D
Line_8   ... := C

Interference Graph

A   D

Loc1

Loc2

B   C

Liveness

Machine Code Optimization: Data Allocation

Allocation: Map all variables to their color's location

Memory at Runtime

%rsp   %rbp

| B / C | A / D | old RBP | old RIP |

foo AR

Unfortunately, coloring is NP-Compete ☹

Use/Definition Sequence
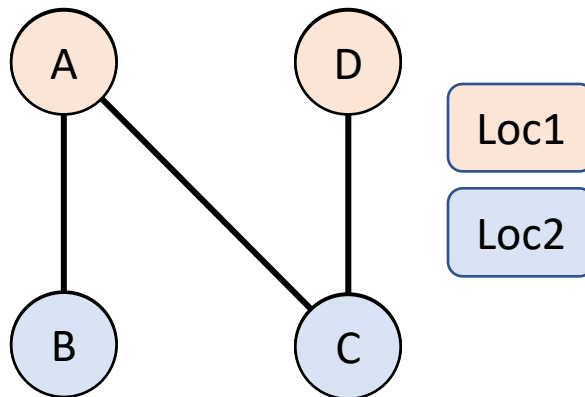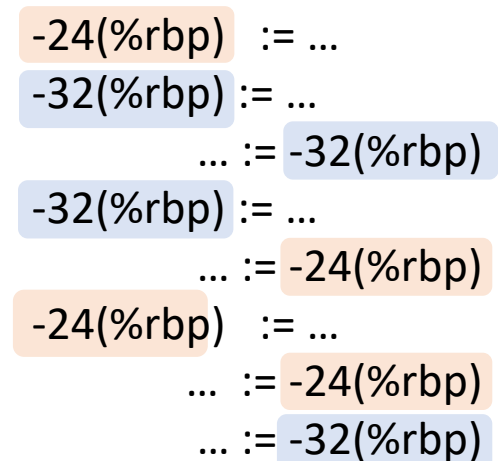
Line_1   A := …
Line_2   B := …
Line_3   … := B
Line_4   C := …
Line_5   … := A
Line_6   D := …
Line_7   … := D
Line_8   … := C

Interference Graph

A   D

Loc1

Loc2

B   C

Allocation

-24(%rbp) := …
-32(%rbp) := …
         … := -32(%rbp)
-32(%rbp) := …
         … := -24(%rbp)
-24(%rbp) := …
         … := -24(%rbp)
         … := -32(%rbp)

Today's Outline
Machine Code Optimization

Improving data allocation

- Register allocation

Improving Final Code

- Peephole optimization

- Instruction Pipelines

Optimization

Register Allocation
Machine Code Optimization: Register Allocation

When possible, keep variables *entirely* in registers

Why?

- Some computation requires register operands

- Register operands are intrinsically faster

Register coloring

- Assign a color to each available register

- Optimal assignment is NP-Complete ☹

Problem: Callee clobbers registers!

Machine Code Optimization: Register Allocation

%r9 enter foo
    [f1] := 2
    [f2] := 3
%r10 call bar
    [f3] := [f2] + [f1]
    leave foo

%r9 enter bar
    [b1] := 9
    [b2] := 7
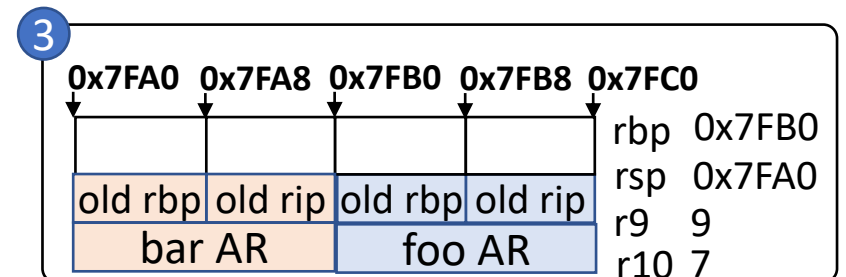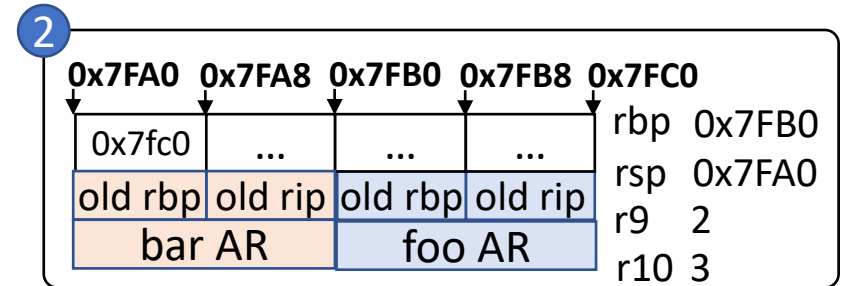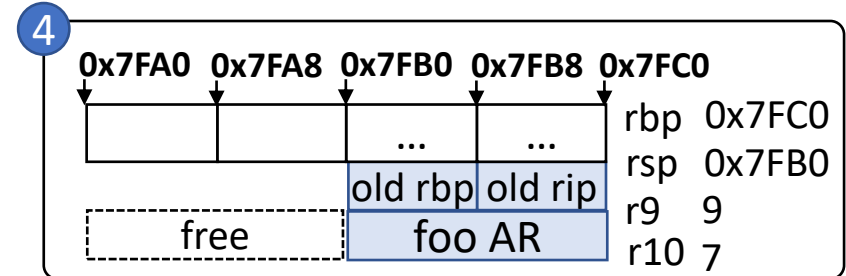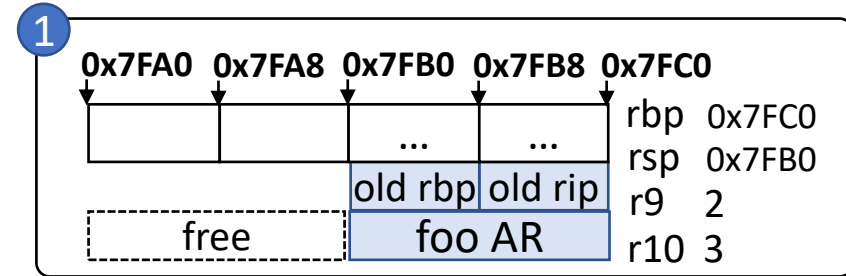%r10 [glb_g] := [b1] - [b2]
    leave foo

```
foo: pushq %rbp
     movq %rsp, %rbp
     addq $16, %rbp
     movq $2, %r9
     movq $3, %r10
  1  callq bar
  4  addq %r9, %r10
     popq %rbp
     retq

bar: pushq %rbp
     movq %rsp, %rbp
     addq $16, %rbp
  2  movq $9, %r9
     movq $7, %r10
     subq %r10, %r9
     movq %r9, (glb_g)
  3  popq %rbp
     retq
```


16

Which Registers To Use?

Machine Code Optimization: Register Allocation

Register Allocation Complication:

- Callees overwriting registers

- Callees can't statically learn which registers the caller is using

Assume these general purpose registers %r11, %r12, %r13, %r14, %r15

baz — Needs 2 registers — %r11 %r12

foo — Needs 2 registers — %r11 %r12

bar — Needs 3 registers — %r13 %r14 %r15
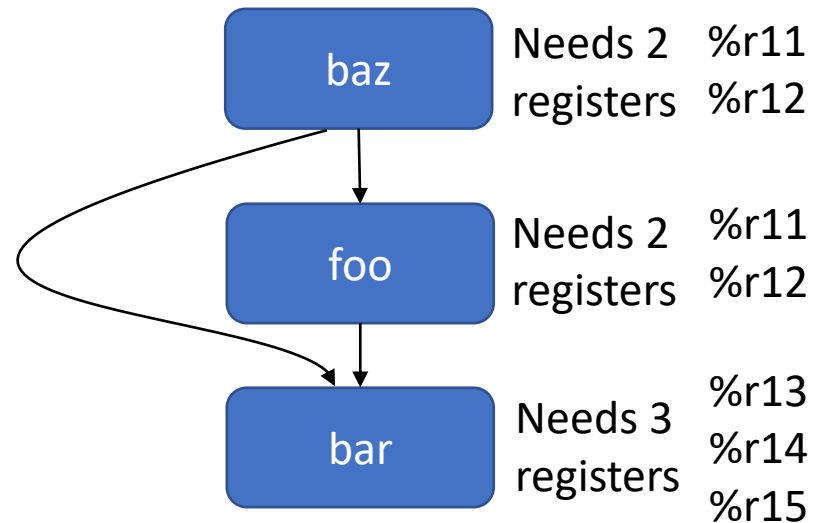
Register Conventions

Calling convention indicates which registers should be preserved across calls

- Preserved (callee-saved): rbx, rsp, rbp, r12, r13, r14, r15

- Volatile (caller-saved): rax, rdi, rsi, rdx, rcx, r8, r9, r10, r11

Preserving Register Values

Machine Code Optimization: Register Conventions

Analogy: housesharing
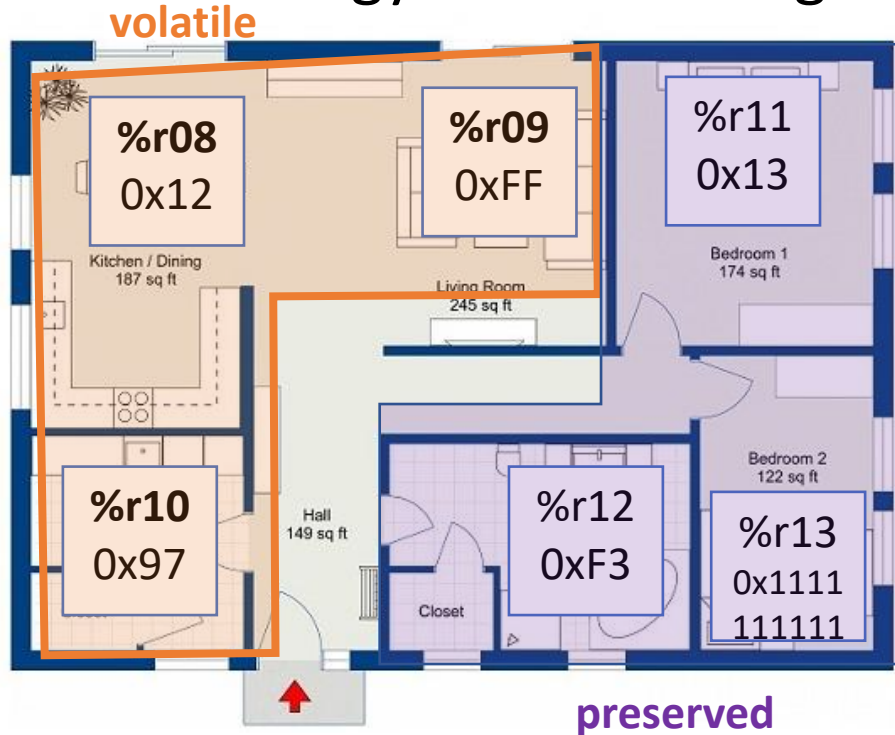
Imagine a function call. There's a caller and a callee. Let's use an analogy

- Me (homeowner): caller
- You (guest): callee

Function call: you stay at my house.

Preserving Register Values

Analogy: housesharing

volatile

%r08
0x12

%r09
0xFF

%r11
0x13

Kitchen / Dining
187 sq ft

Living Room
245 sq ft

Bedroom 1
174 sq ft

%r10
0x97

Hall
149 sq ft

%r12
0xF3

%r13
0x1111
111111

Bedroom 2
122 sq ft

Closet

preserved

Imagine a function call. There's a caller and a callee. Let's use an analogy

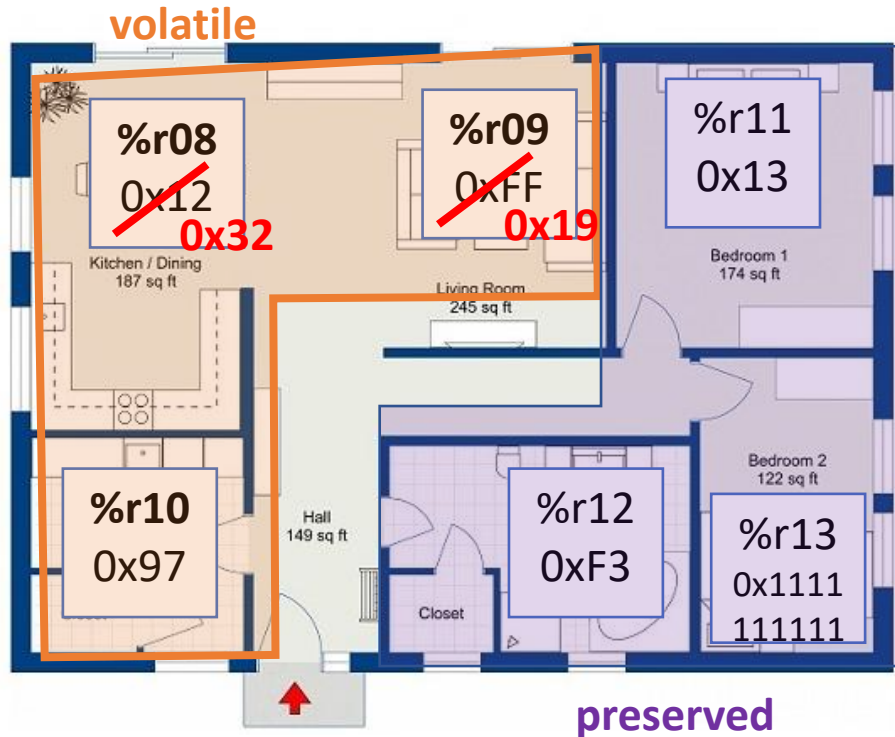- Me (homeowner): caller

- You (guest): callee

Function call: you stay at my house. **Rooms:** registers

- Common rooms: you can goof around in there (volatile)

- Restricted rooms: don't touch anything (preserved)

Preserving Register Values
Machine Code Optimization: Register Conventions

volatile

%r08
0x12
0x32

%r09
0xFF
0x19

%r11
0x13

%r10
0x97

%r12
0xF3

%r13
0x1111
111111

preserved

Respectful housesharing

In the call
- You only touch the volatile registers
After the caller
- I don't care

Allowed

I can count on
%r11's being same

I cannot count on
%r08's value

Caller code
movq $0x13, %r11
movq $0x12, %r08
callq
addq $1, %r11
movq $0, %r08

Preserving Register Values
Machine Code Optimization: Register Conventions

Respectful housesharing
In the call
- You only touch the volatile registers
After the caller
- I don't care
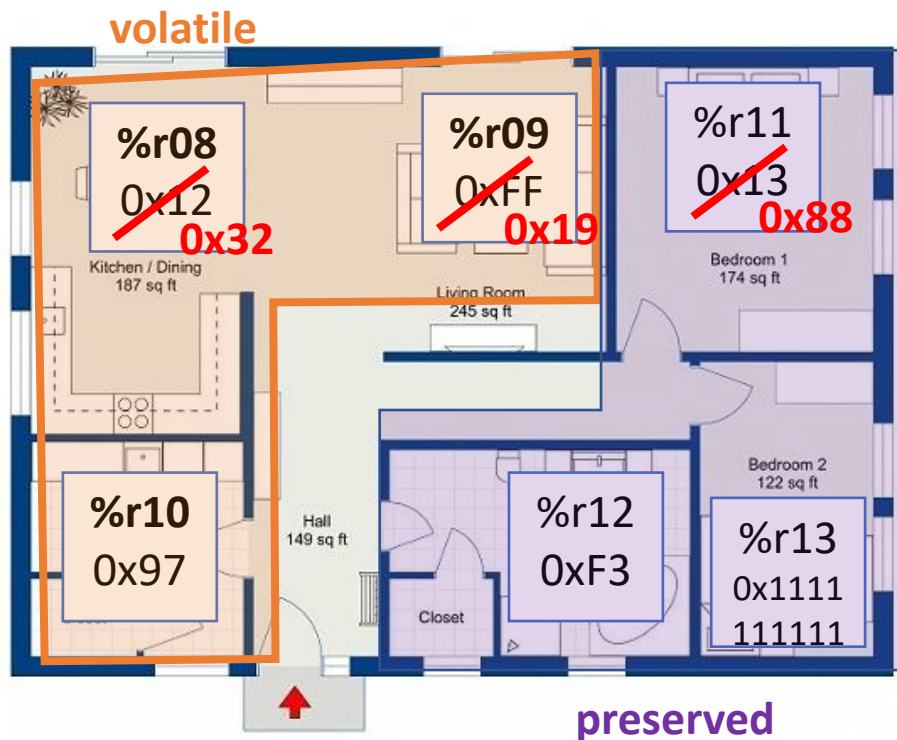
Allowed

Disrespectful housesharing
In the call
- You (also) touch the preserved registers
After the caller
- Caller's expectation violated!!

Illegal
(violated System V ABI)

Caller code
movq $0x13, %r11
movq $0x12, %r08
callq
addq $1, %r11
movq $0, %r08

Preserving Register Values
Machine Code Optimization: Register Conventions

volatile

%r08
0x12 ~~0x12~~ **0x32**

%r09
0x~~FF~~ **0x19**

%r11
0x~~13~~ **0x88**
0x13

Kitchen / Dining
187 sq ft

Living Room
245 sq ft

Bedroom 1
174 sq

%r10
0x97

Hall
149 sq ft

%r12
0xF3

%r13
0x1111
111111

Bedroom 2
122 sq ft

Closet
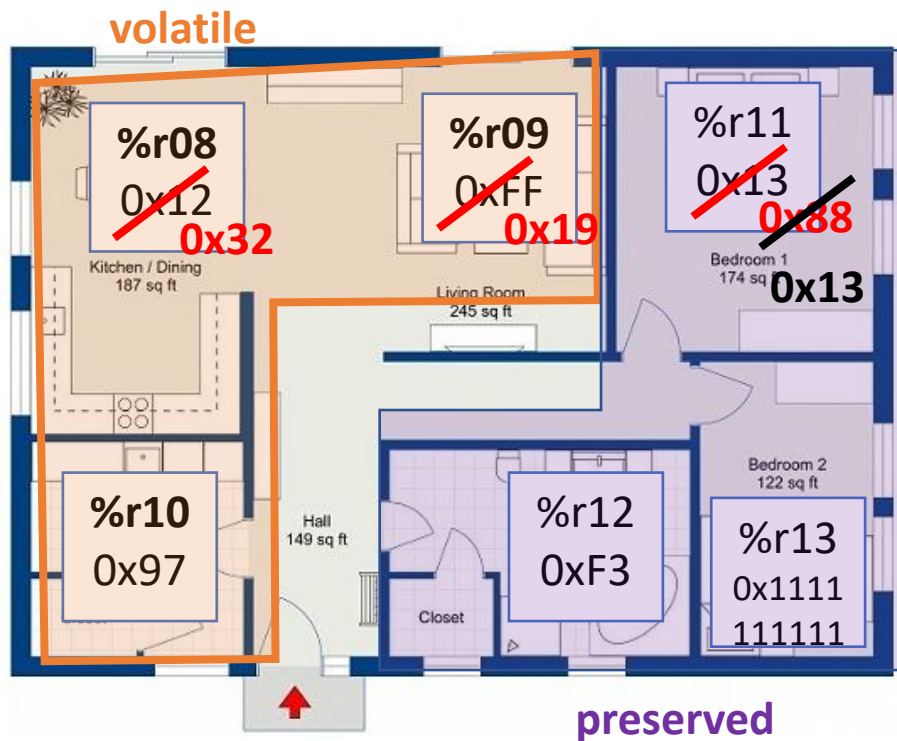
preserved

Respectful housesharing
In the call
- You only touch the volatile registers
After the caller
- I don't care

Allowed

Disrespectful housesharing
In the call
- You (also) touch the preserved registers
After the caller
- Caller's expectation violated!!

Illegal
(violated System V ABI)

Sneaky housesharing
In the call
- You (also) touch the preserved registers
- You restore the preserved values before return
After the caller
- The caller never knows of your deviance

Allowed

Implementing Register Conventions

Using callee-saved registers

Being a "sneaky guest"

- Push the preserved register values before you use them

- Pop the stacked values before you return

Prologue
pushq %rbp
movq %rsp, %rbp
addq $16, %rbp
pushq %r08
pushq %r09
subq $32, %rsp

Epilogue
addq $32, %rsp
popq %r09
popq %r08
popq %rbp
retq

Using caller-saved registers

Being a "sneaky owner"

- Save a volatile register to the stack

- Pop the stacked values before you return

Call site
pushq %r11
pushq %r12
callq fn_callee
popq %r12
popq %r11

24

Today's Outline
Machine Code Optimization

Improving data allocation

- Register allocation

Improving Final Code

- Peephole optimization

- Instruction Pipelines

Optimization

Fixing "Obviously Sub-Optimal" Code

Machine Code Optimization: Peephole Optimizations

A code generator may output obviously "weak" code

- why?
  - Ignoring global context
  - Correctness-first design

Solution: pattern-match the most obvious problems

An obvious flaw

The Idea of the Peephole
Machine Code Optimization: Peephole Optimizations

- Called "peephole" optimization because we are conceptually sliding a small window over the code, looking for small patterns
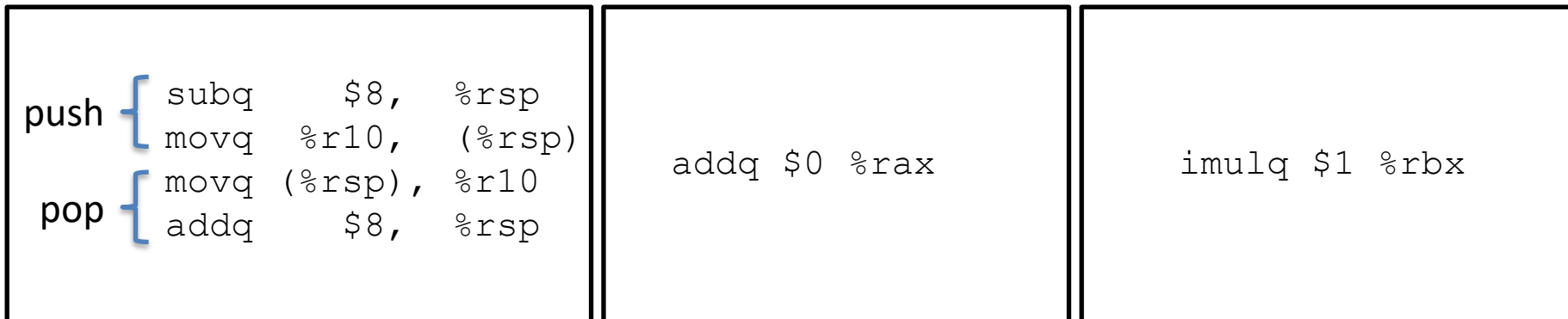
Remove Semantic No-ops
Machine Code Optimization: Peephole Optimizations

Remove *semantic no-op* sequences

- Push followed by pop

- Add/sub 0

- Mul/div 1

```
push  { subq      $8,   %rsp
       { movq  %r10,   (%rsp)
pop   { movq (%rsp),  %r10
       { addq      $8,   %rsp
```

```
addq $0 %rax
```
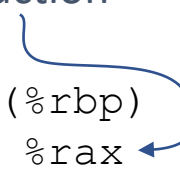
```
imulq $1 %rbx
```

Sequence Simplification
Machine Code Optimization: Peephole Optimizations

- Store then load

Useless instruction

```
movq %rax, -8(%rbp)
movq -8(%rbp) %rax
```

- Arithmetic equivalence

Just add 3

```
addq  $1 %r11
addq  $2 %r11
```

- Jump to next

Useless instruction

```
       jmp  Label1
Label1: addq $2, %rax
```

Instruction Strength Reduction

Machine Code Optimization: Peephole Optimizations

Instruction Strength Reduction

- Prefer "weak" (narrow/specialized instruction) instead

- Avoid "strong" (general-purpose) instruction

```
imulq  $2 %rax
        shift-left %rax
```

```
addq  $1 %rax
        inc %rax
```

Requires knowledge of the fast and slow instructions

"Weaker" is better

Peephole Optimization: Summary

Concept

- Final code "postprocessing"
- Slide a window over the program that pattern-matches suboptimal cases

Benefits

- Remove some consequences of naïve machine code generation
- Leverage hardware features
  - Faster instructions

Today's Outline
Machine Code Optimization

Improving data allocation

- Register allocation

Improving Final Code

- Peephole optimization
- Instruction Pipelines

Optimization

Background: Multi-stage Cycles

Machine Code Optimization: Delay Slots – Branch Hazards

The classic cycle of a processor:

Fetch - read value at the program counter

Decode – figure out what the instruction is

Execute – do what the instruction

Write-back – commit the results to register/memory

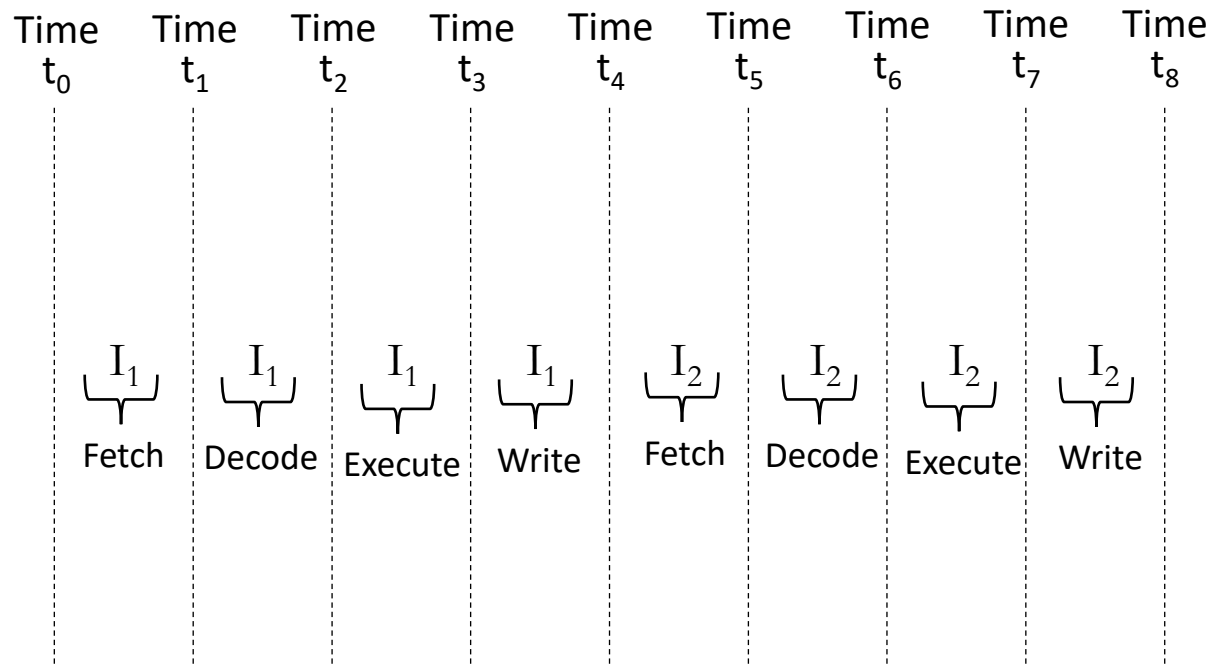**If we did all of this sequentially, we'd waste time & resources**

Background: Instruction Pipelines

Machine Code Optimization: Delay Slots – Branch Hazards

Idea: Start on next instruction before current done

I_1: addq %rax %rbx
I_2: subq %rcx %rdx

Time t_0	Time t_1	Time t_2	Time t_3	Time t_4	Time t_5	Time t_6	Time t_7	Time t_8
I_1	I_1	I_1	I_1	I_2	I_2	I_2	I_2	
Fetch	Decode	Execute	Write	Fetch	Decode	Execute	Write	

Background: Instruction Pipelines

Machine Code Optimization: Delay Slots – Branch Hazards

Idea: Start on next instruction before current done

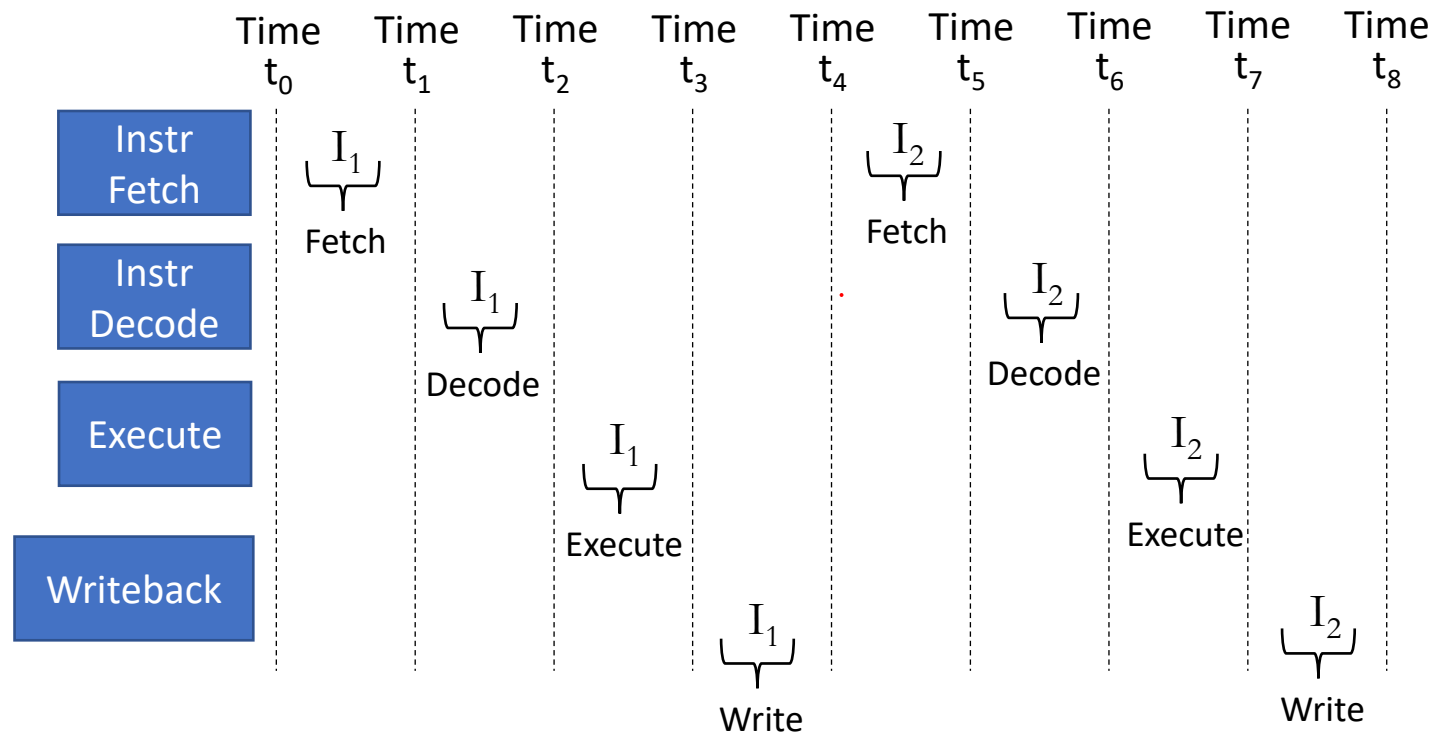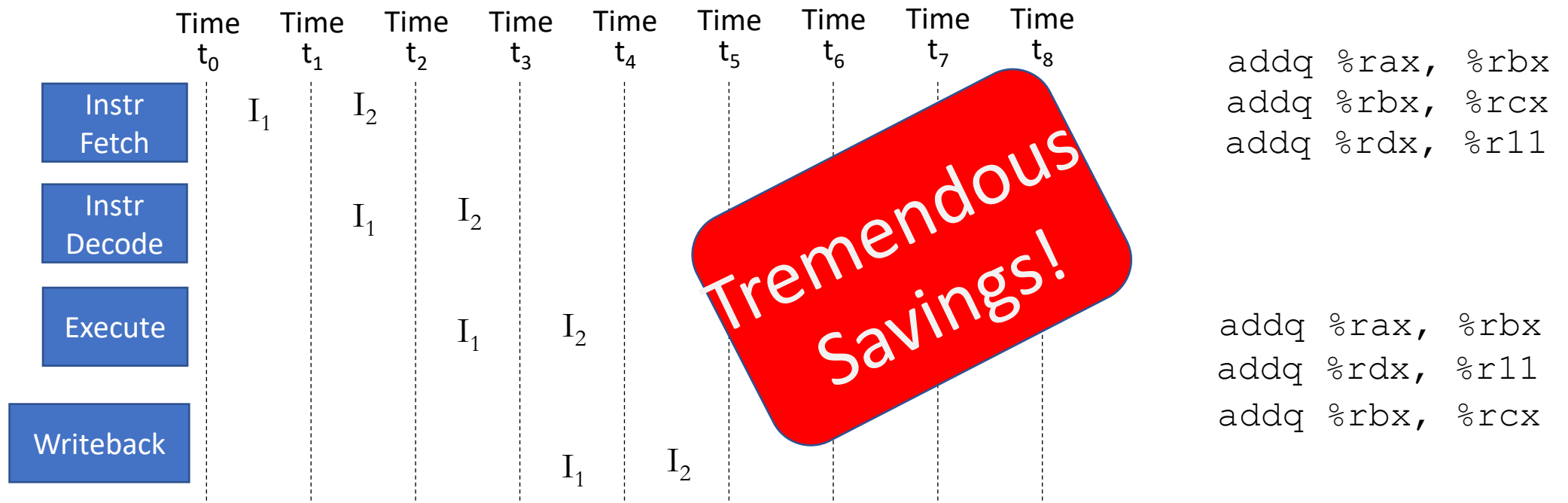I_1: addq %rax %rbx
I_2 : subq %rcx %rdx

Background: Instruction Pipelines

Machine Code Optimization: Delay Slots – Branch Hazards

Idea: Start on next instruction before current done

I_1: addq %rax %rbx
I_2 : subq %rcx %rdx

	Time t_0	Time t_1	Time t_2	Time t_3	Time t_4	Time t_5	Time t_6	Time t_7	Time t_8
Instr Fetch	I_1	I_2							
Instr Decode		I_1	I_2						
Execute			I_1	I_2					
Writeback				I_1	I_2				

```
addq %rax, %rbx
addq %rbx, %rcx
addq %rdx, %r11
```

Tremendous Savings!

```
addq %rax, %rbx
addq %rdx, %r11
addq %rbx, %rcx
```

Lecture End!

Machine Code Optimization: Wrap-Up

Summary:

Be careful about which instructions you use

• Selection: the choice of instructions in output

• Scheduling: the order of instructions in output

Next Time:

• Optimizing program structure