Check-In

Review: The post-compilation toolchain

Virtual memory simplifies the task of the loader. What extra steps does the loader need to take without virtual memory?

University of Kansas| Drew Davidson

EECS 665 COMPILER CONSTRUCTION

Machine Code Optimization

Previously

Post-compilation toolchain

Compiler Toolchains

- Overview
- What GCC Does

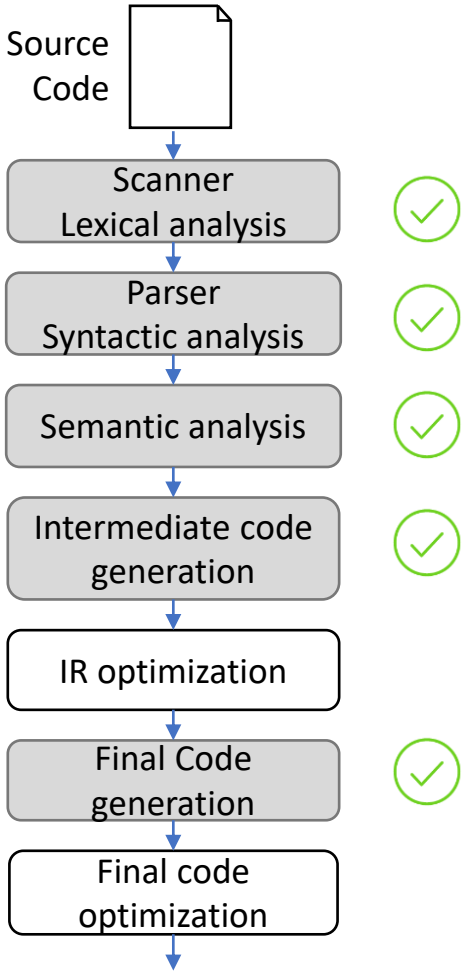
Component Walkthrough

- Assembler
- Linker
- Loader


COMPILER
Postcompilation Pit-stop
Machine Codegen
Optimization
Code Generation
Intermediate Representation
Semantic Analysis
Syntactic Definition
Parsing
Syntax-Dir Translation
Regular Languages
Lexical Analysis
LAND
GLOPPY
CANDY CASTLE
KING KANDY
QUEEN FROSTINE
GRAMMA NUTT
PRINCESS LOLLY
LORD LICORICE
LICORICE CASTLE
JOLLY
PLUMPY

Compiler Construction

Progress Pics



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

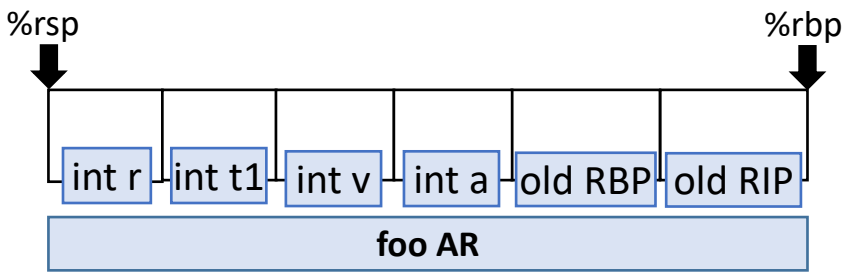
Source code

```
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



Surely one could use fewer memory slots!

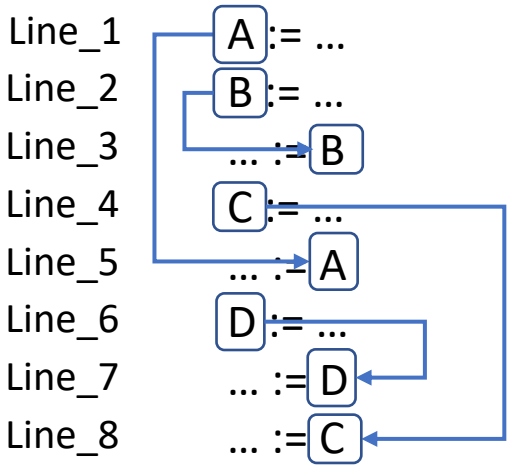
Maybe we could share some slots

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



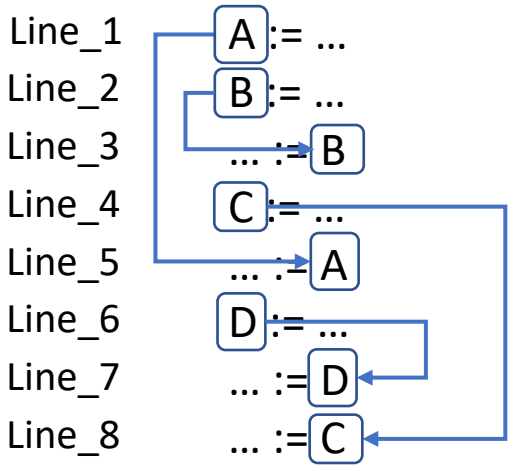
Liveness

Machine Code Optimization: Data Allocation

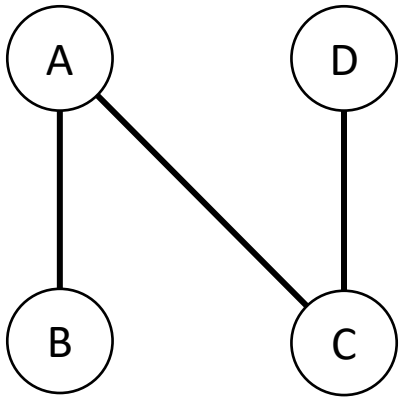
The interference graph:

- Nodes are variables
- Edges show simultaneously live variables

Use/Definition Sequence



Interference Graph



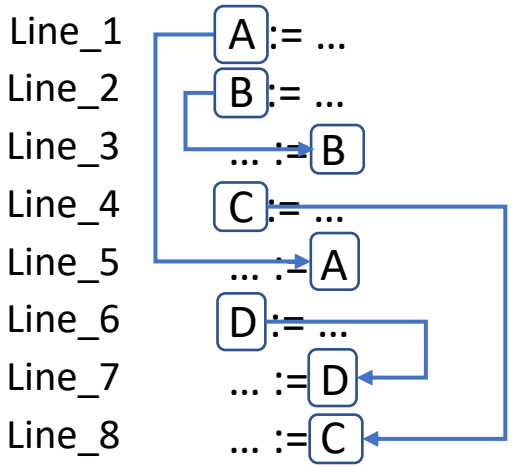
Liveness

Machine Code Optimization: Data Allocation

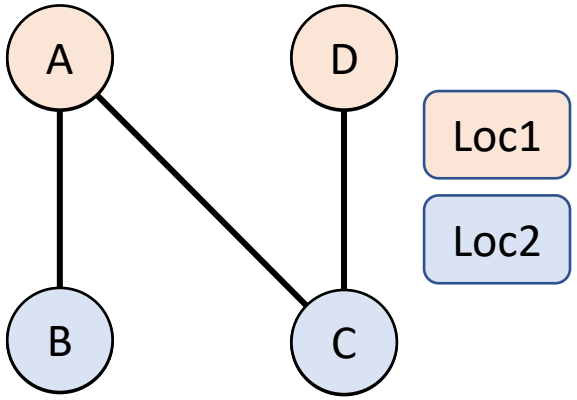
Coloring:

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

Use/Definition Sequence



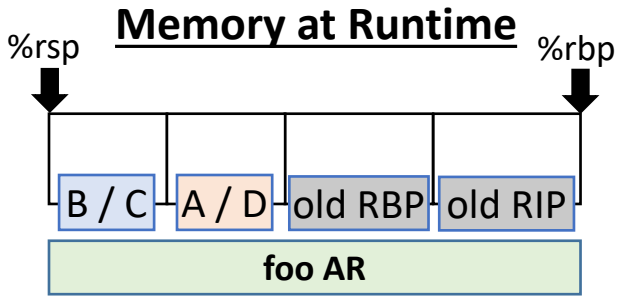
Interference Graph



Liveness

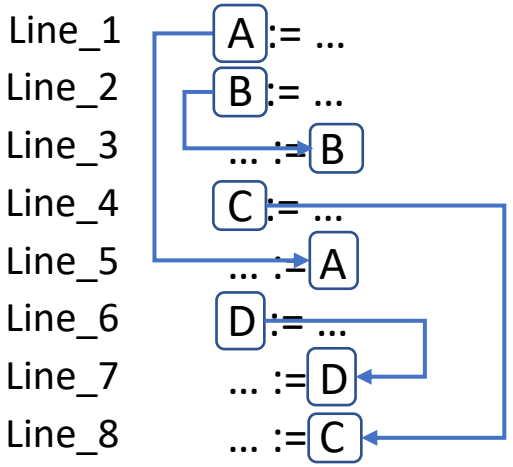
Machine Code Optimization: Data Allocation

Allocation: Map all variables to their color's location

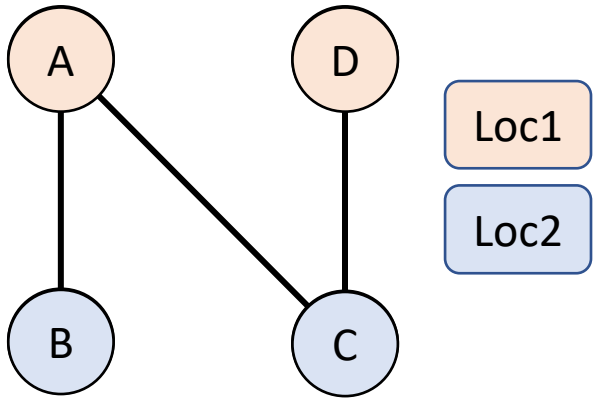


Unfortunately, coloring is NP-Compete ☹

Use/Definition Sequence



Interference Graph



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!

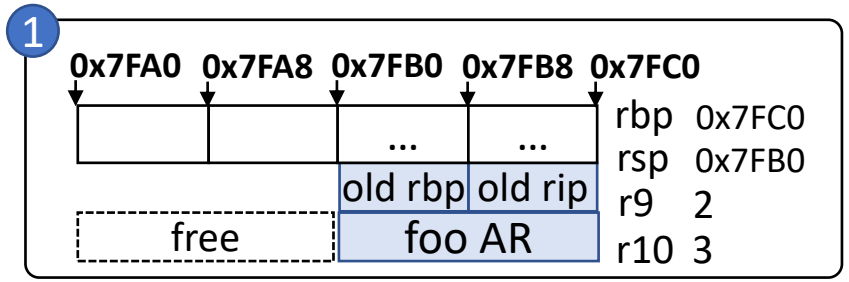
Machine Code Optimization: Register Allocation

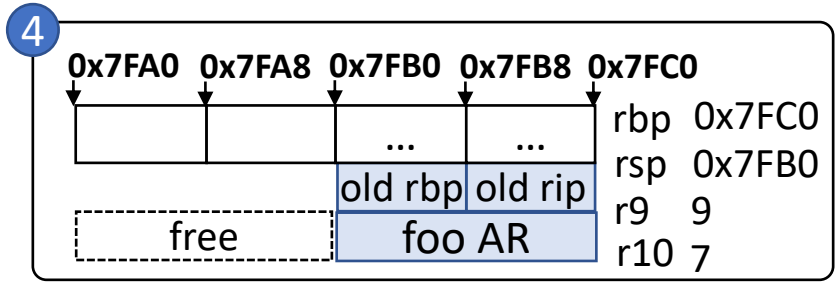
```
%r9   enter foo
      [f1] := 2
%r10  [f2] := 3
      call bar
      [f3] := [f2] + [f1]
      leave foo
```

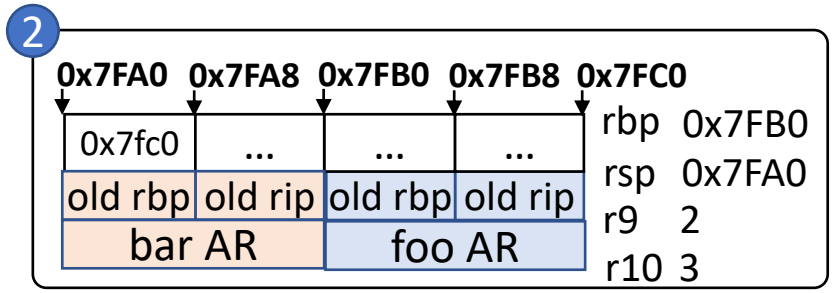
```
%r9   enter bar
      [b1] := 9
%r10  [b2] := 7
      [glb_g] := [b1] - [b2]
      leave foo
```

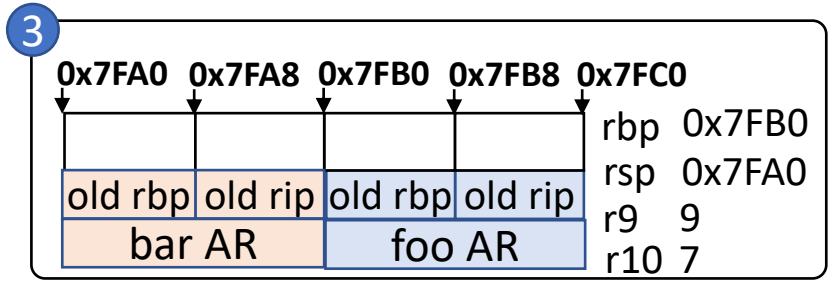
```
foo: pushq %rbp
     movq %rsp, %rbp
     addq $16, %rbp
     movq $2, %r9
     movq $3, %r10
1 →  callq bar
4 →  addq %r9, %r10
     popq %rbp
     retq
```

```
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
```









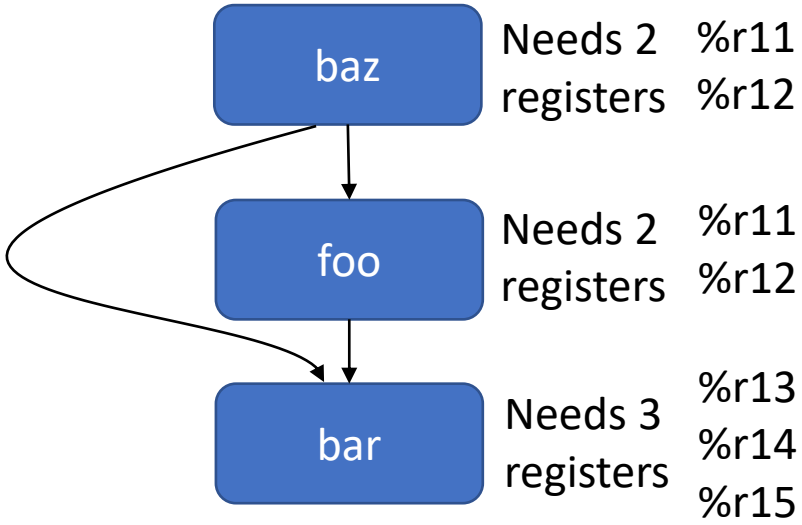
Which Registers To Use?

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



Register Conventions

Machine Code Optimization: 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

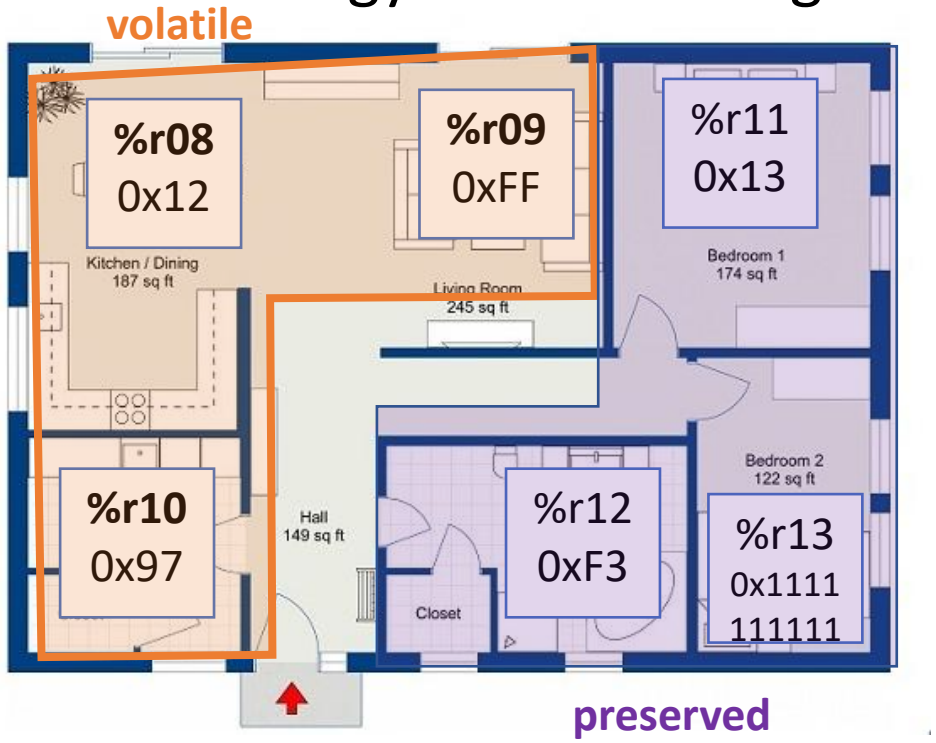
- Me (homeowner): caller
- You (guest): callee

Function call: you stay at my house.

Preserving Register Values

Machine Code Optimization: Register Conventions

Analogy: housesharing



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

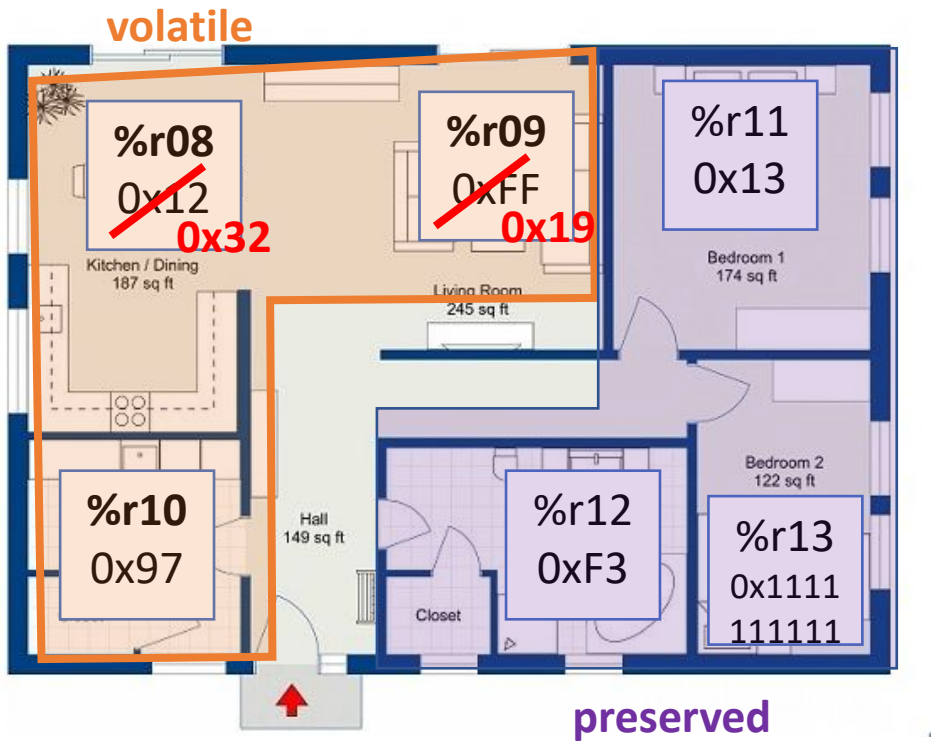
- Me (homeowner): caller
- You (guest): callee

Function call: you stay at my house.
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



Respectful housesharing

In the call

- You only touch the volatile registers

After the caller

- I don't care

Allowed

<u>Caller code</u>

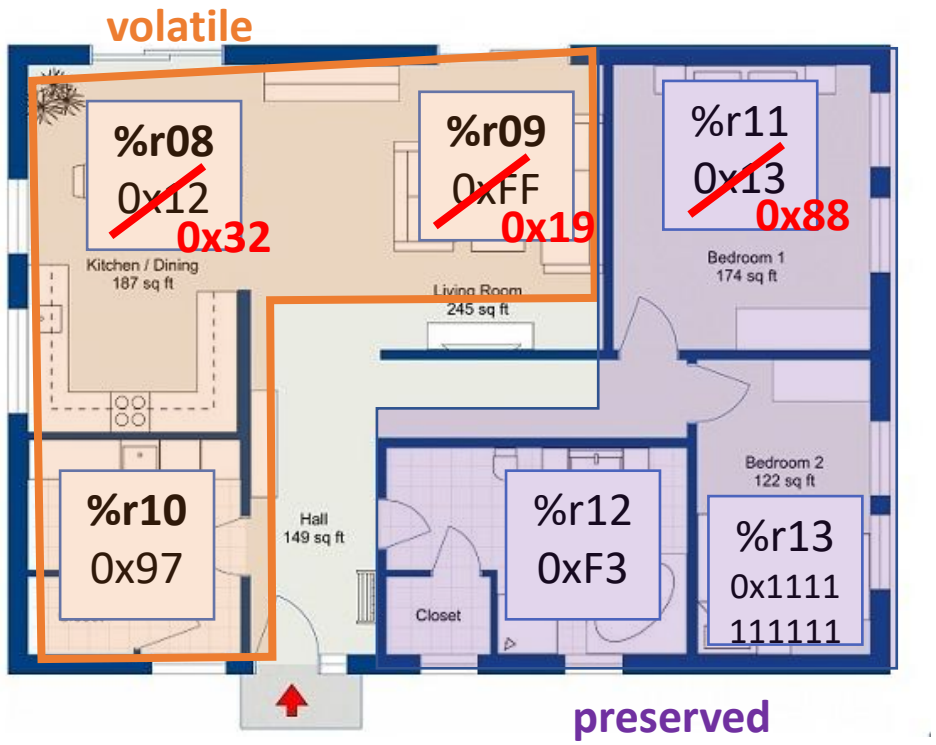
```
movq $0x13, %r11
movq $0x12, %r08
callq
addq $1, %r11
movq $0, %r08
```

I can count on %r11's being same

I <u>cannot</u> count on %r08's value

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!!

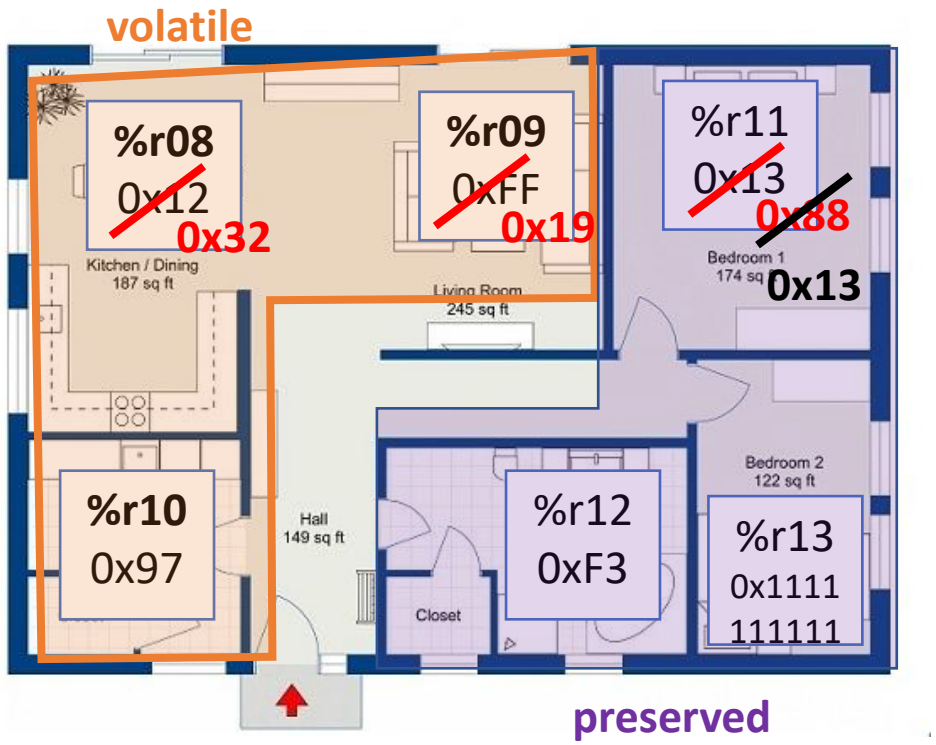
Illegal
(violated System V ABI)

<u>Caller code</u>

```
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)

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

Machine Code Optimization: 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
```

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

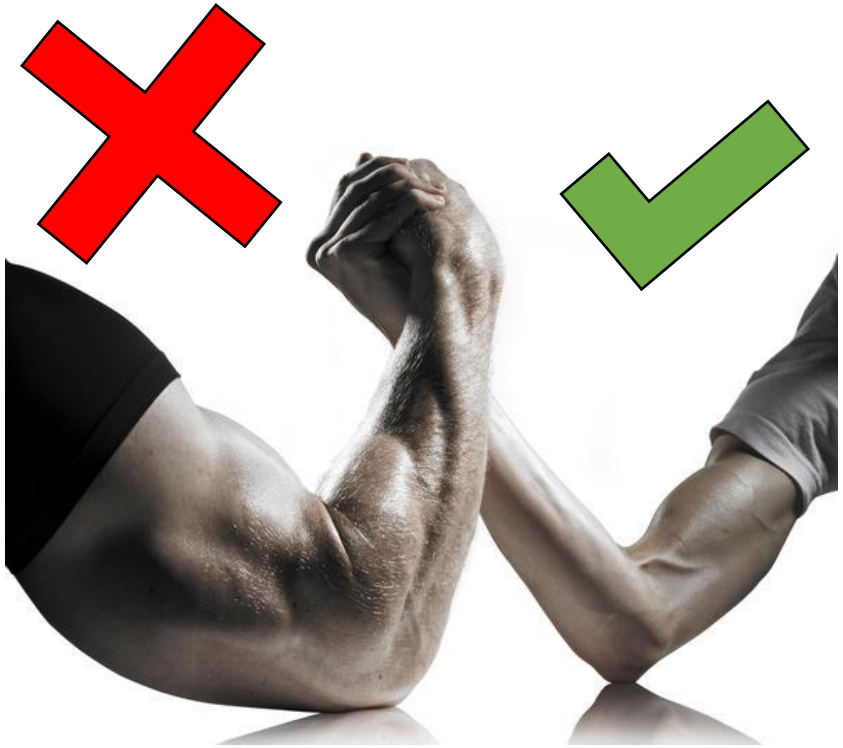
```
imulq   $2 %rax
```

shift-left %rax

```
addq   $1 %rax
```

inc %rax

Requires knowledge of the fast and slow instructions



“Weaker” is better

Peephole Optimization: Summary

Machine Code Optimization: Peephole Optimizations

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

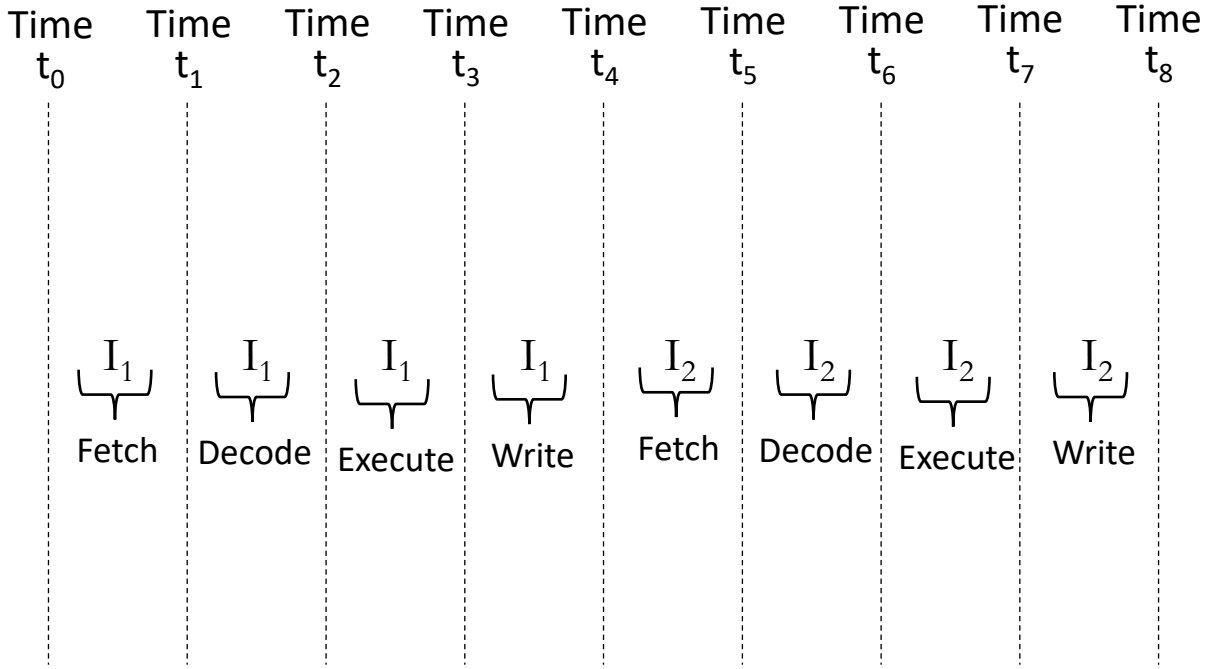
If we did all of this sequentially, we'd waste time & resources

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

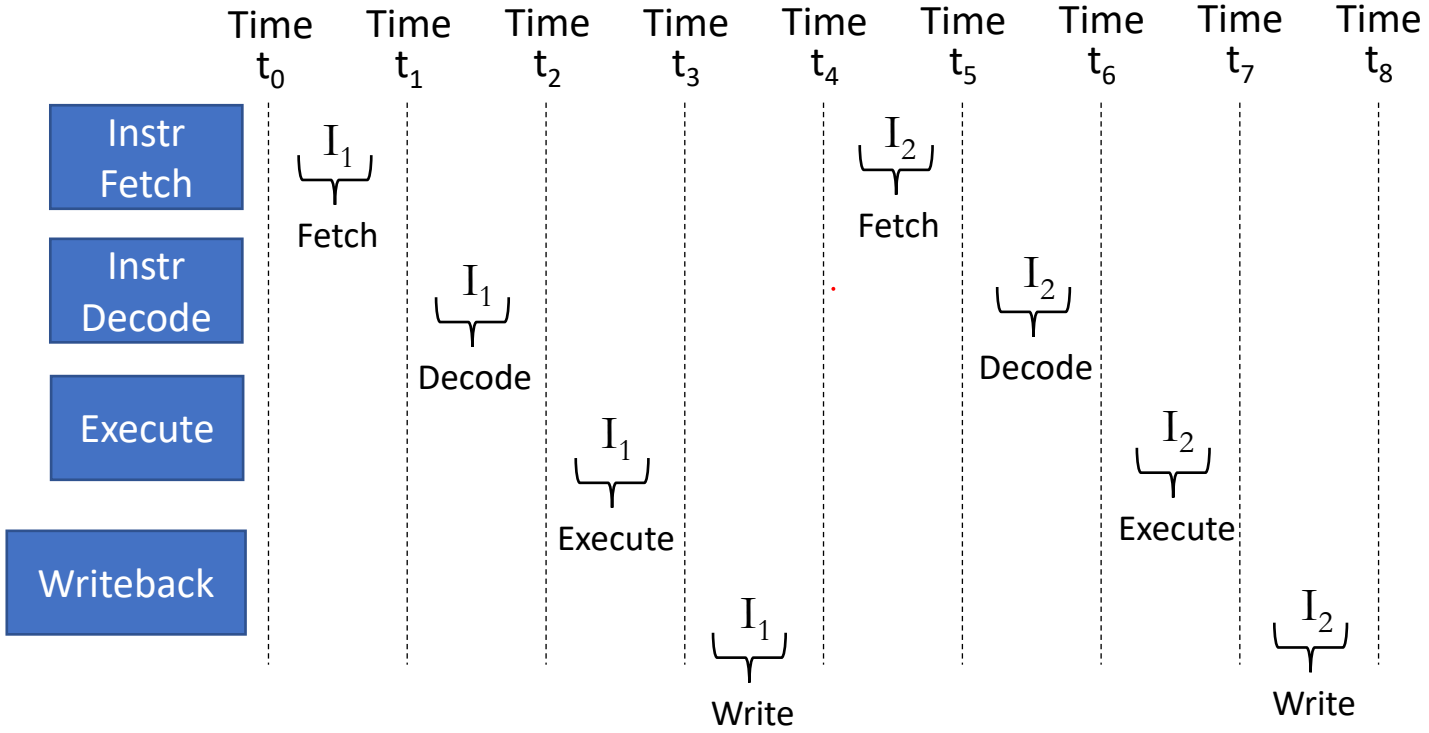


Background: Instruction Pipelines

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

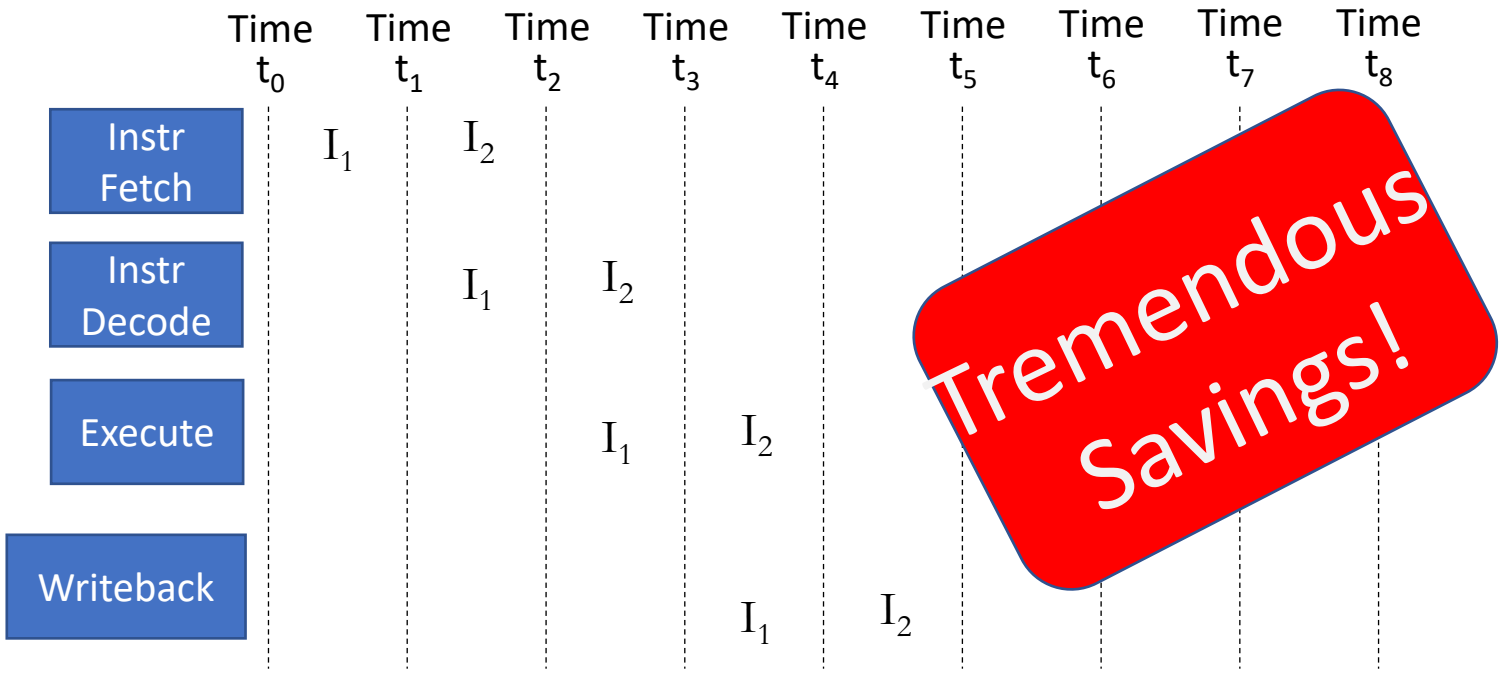


Background: Instruction Pipelines

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




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

```
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