Check-In #27

Review: Activation Records

Show the layout of an activation record with two 64-bit locals. Write the function prologue and epilogue corresponding to that function

Check-In #27 Solution

Review: Activation Records

Announcements
Administrivia

P5 outdated instructions

University of Kansas | Drew Davidson

EECS 665

COMPILER CONSTRUCTION

Statement Code Generation

Last Lecture
Activation Records

Managing the Stack

- Managing data
- Managing control

Architecture

Big Picture: Architecture Aims and Means

Review: Activation Records

Big Picture: Architecture Aims and Means

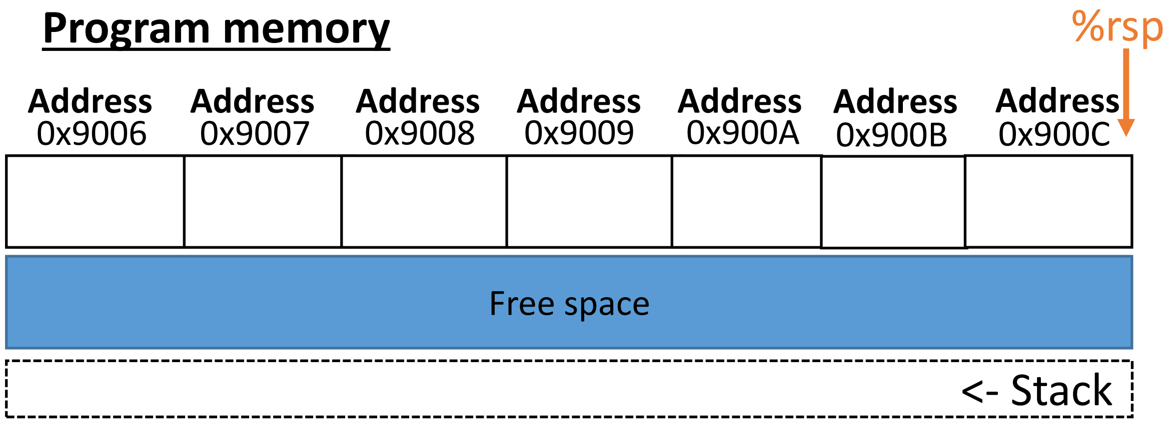
Review: Activation Records

Aim: *Simulate source code concepts*

Functions with local variables

Call chains

Program memory



Means: *Leverage x64 capabilities*

A region of available memory bounded by rsp

subq $X, %rsp: claim X bytes on the stack

addq $X, %rsp: free X bytes on the stack

movq(%rsp): access top of stack

Divide used memory into frames, one frame per function invocation

Track the base of the current frame with %rbp

Store return address and previous frame base in the activation record

Big Picture: Architecture Aims and Means

Review: Activation Records

Two useful instructions for manipulating stack memory

pushq <opd> - decrement %rsp by 8, place opd in memory at %rsp location

popq <opd> - read value at %rsp location, incremenent %rsp by 8

Two useful instructions for simulating call chains

callq <lbl> - effectively does pushq *the return point instruction's address*, then sets %rip to <lbl>

retq – effectively does popq %rip

Maintaining Activation Records

Review: Activation Records

Each activation record can store…

- Local data for a function invocation
- Enough bookkeeping to restore the caller's frame

AR setup / break-down

- Claim AR memory with the function prologue at entry to each function

```
pushq %rbp
movq %rsp, %rbp
addq $16, %rbp
subq $X, %rsp
```

- Release AR memory with the function epilogue at exit point of the function

```
addq $X, %rsp
popq $rbp
retq
```

Addressing modes

Toward Local Variables

Some Nice "Shortcuts"

- Often want to read memory at a fixed offset from some register

  "the memory at 8 bytes before %rbp"

- Good news! X64 can do that:

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

- This is a very handy addressing mode

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

"Move the value AT %rbp – 8 into %rax" = movq %rbp, %rdx / subq $8, %rdx / movq (%rdx), %rax

"Move the value OF %rbp – 8 into %rax" = movq %rbp, %rdx / subq $8, %rdx / movq %rdx, %rax

Last Lecture
Activation Records

Managing the Stack

- Managing data
- Managing control

<u>You Should Know</u>

How to code up stack frames
The function prologue
The function epilogue

Architecture

Where We're At

Progress Pics

Assembled quite a few x64 concepts

- Basic data manipulation (movq)
- Basic math (addq, idivq, etc)
- Global data (.data, .quad, .byte)
- Local data
- Function calls

This is really all we need for a basic language!

A Less-Trivial x64 Program

Working with Activation Records

```
g : int;
v : () -> void {
  local : int;
  k :int;
  local = g - 1;
}
main : () -> int {
  loc1 : int;
  loc2 : int;
  g = 2;
  v();
};
```

```
.data
.quad g 0
.globl main
.text
fn_v:  pushq %rbp
       movq %rsp, %rbp
       addq $16, %rbp
       subq $16, %rsp
       movq (g), %rax
       subq $1, %rax
       movq %rax, -24(%rbp)
       addq $16, %rsp
       retq
main:  pushq %rbp
       movq %rsp, %rbp
       addq $16, %rbp
       subq $16, $rsp
       movq $2, (g)
       callq fn_v
       addq $16
       retq
```


COMPILER
LAND
Code Generation
Optimization
Architecture
Runtime Environments
Semantic Analysis
Syntactic Definition
Parsing
Syntax-Dir Translation
Regular Languages
Lexical Analysis
MOLASSES SWAMP
GLOPPY
CANDY CASTLE
KING KANDY
QUEEN FROSTINE
GRAMMA NUTT
PRINCESS LOLLY
LORD LICORICE
LICORICE CASTLE
JOLLY
PLUMPY

Lecture Outline

Statement Code Generation

From Quads to Assembly

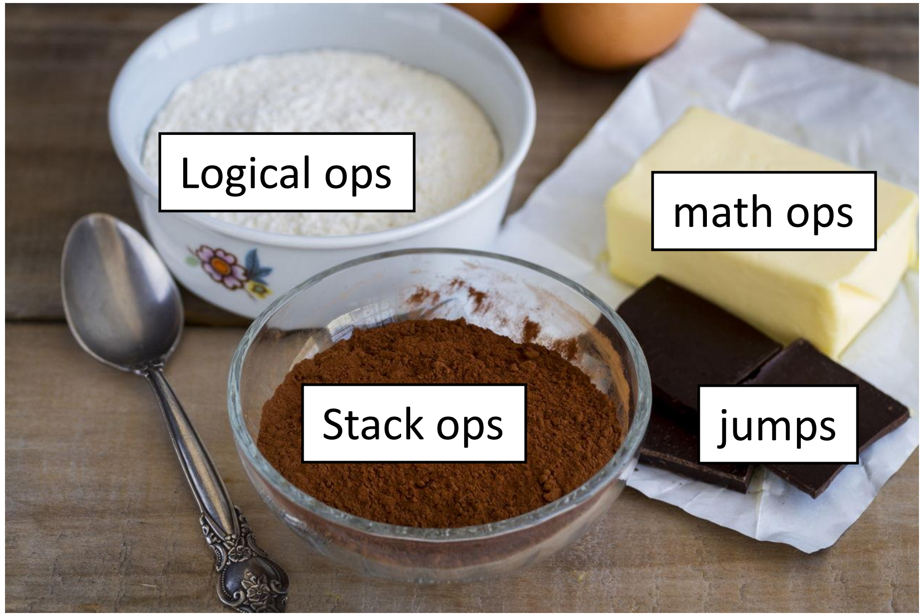
- Approach Overview
- Planning out memory
- Writing out x64

Code generation

Representing Abstract Constructs

Statement Code Generation

Combine (simple) target language constructs…



…to build (complex) source language constructs

Our Approach: Small Steps

Code Generation

2 passes over IRProgram (like passes over AST)

1. Allocate memory for opds (data pass)
2. Generate code for quads (code pass)

Code Generation Objectives

Designing Code Generators

- Two high level goals:
  - Generate correct code ← **Top priority**
  - Generate efficient code

- It can be difficult to achieve both at once
  - Efficient code can be harder to understand
  - Efficient code may have unanticipated side effects

Our Approach: Small Steps

Code Generation

2 passes over IRProgram (like passes over AST)

1. Allocate memory for opds (data pass)
2. Generate code for quads (code pass)

Preparing the 3AC memory layout

Variable Allocation

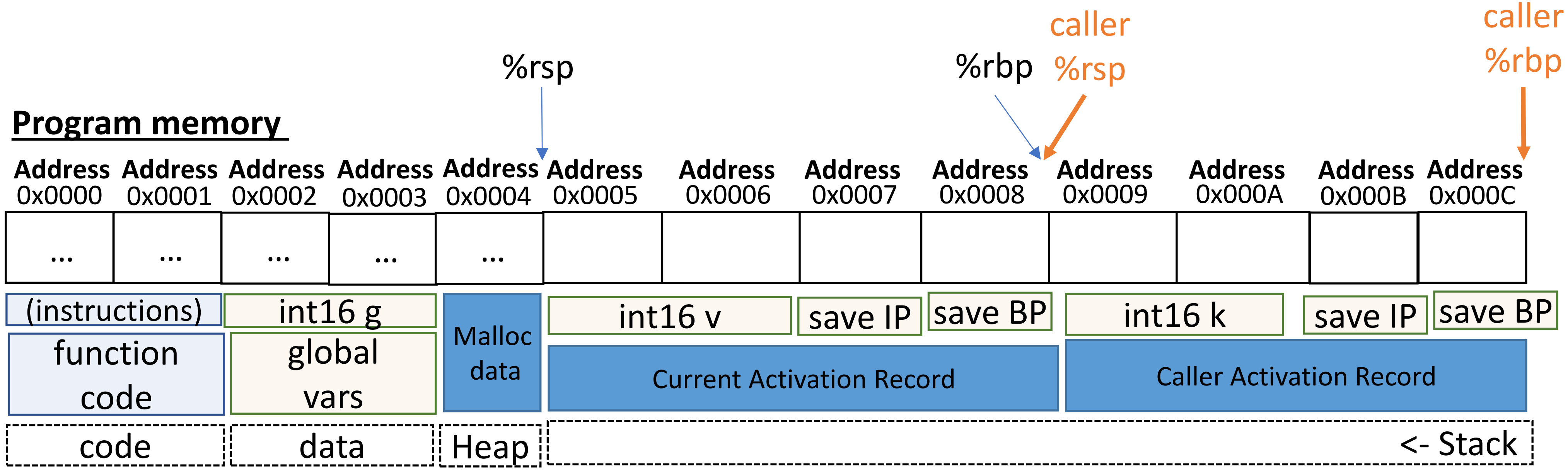
Code Generation

Big picture:

- Every variable needs space in enough space in memory for its type
- Every quad using that variable needs to access the same location

Need a mix of static/dynamic allocation

- Put globals/strings at fixed addresses in memory (absolute locations)
- Put locals/formals at stack offsets in memory (relative locations)



Allocation: In Code (suggestion)

Code Generation

Add a location field (std::string) to semantic symbols

- All related SymOpds have pointers to the same symbol

Location can be a string

- For globals, the label that you'll write
- For locals, the stack offset you'll arrange

Variable Allocation: Globals

Code Generation

3AC Code

```
[g] := 4
```

Where g is a global int

location: At label gbl_g

X64 Code

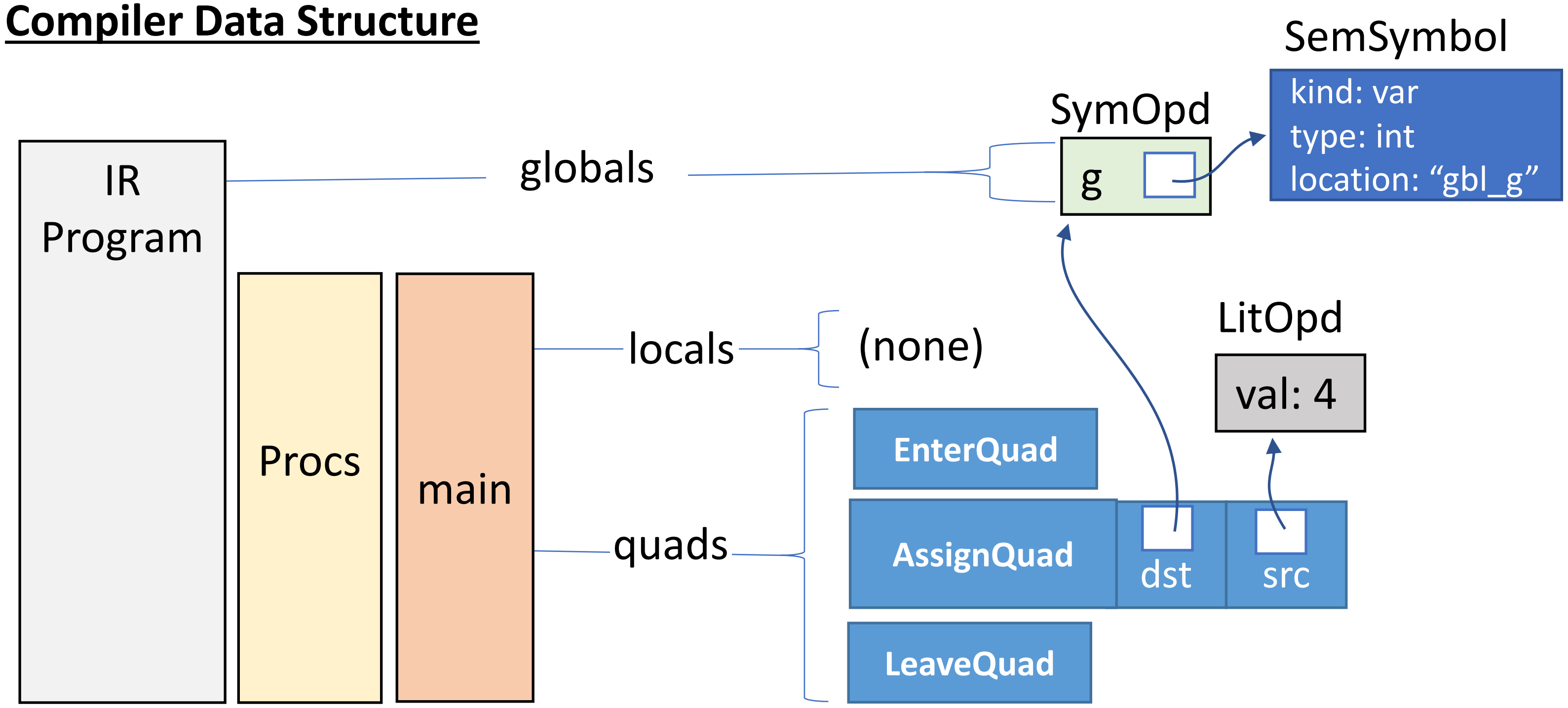
… in .data section …

```
gbl_g:     .quad 0
```

… somewhere in .text section …

```
movq $4, (gbl_g)
```

Compiler Data Structure



Variable Allocation: Locals

Code Generation

3AC Code

```
[v] := 7
```

Where v is a local int

location:
At offset
-24(%rbp)

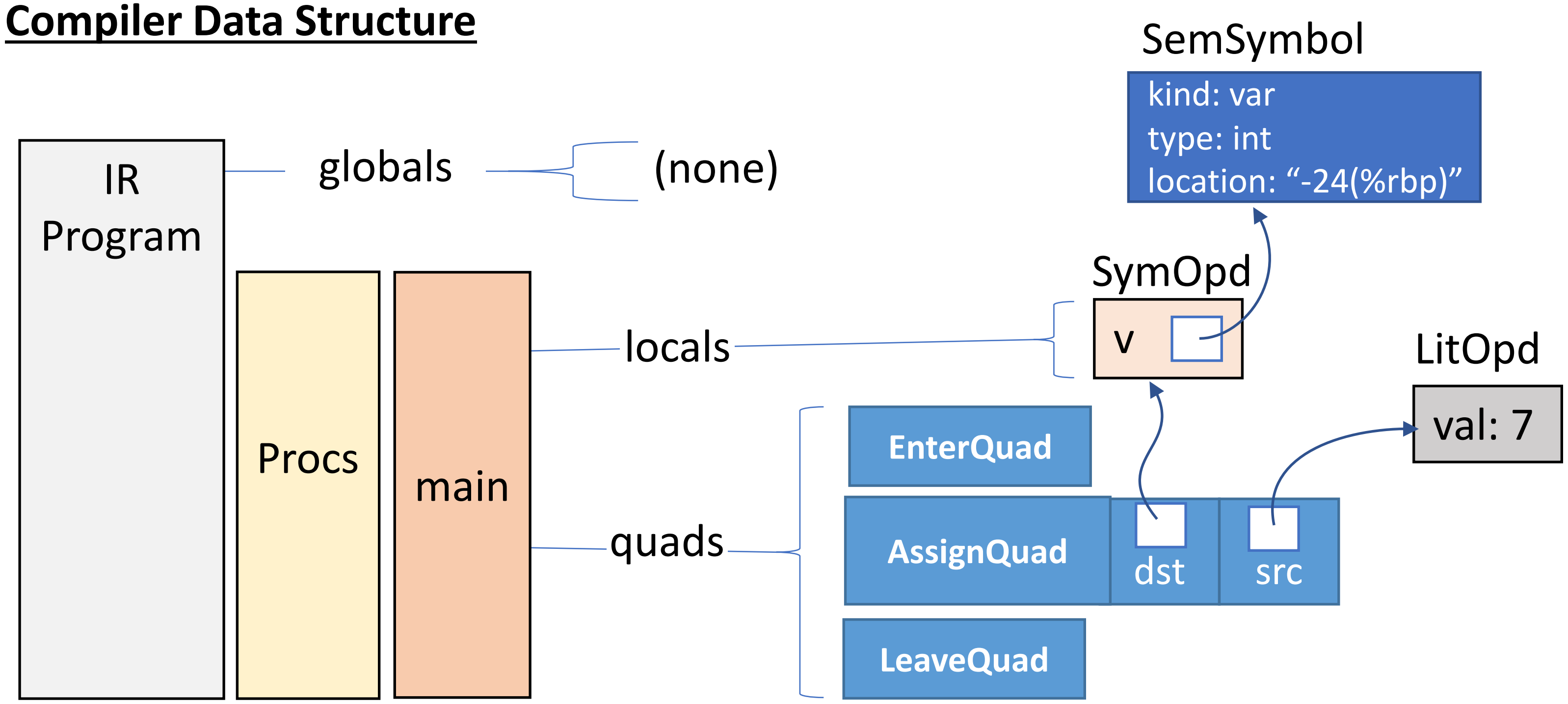
X64 Code

… assume stack frame setup …

… somewhere in main's asm …

```
movq $7, -24(%rbp)
```

Compiler Data Structure



Our Approach: Small Steps

Code Generation

2 passes over IRProgram (like passes over AST)

1. Allocate memory for opds (data pass)
2. Generate code for quads (code pass)

Write the assembly file

Assembly Directives/Initialization

Code Generation

We're gonna write the whole file in one shot

- Aided greatly by our preparatory layout pass
- Also aided by the assembler

Write out .data section:

```
.data
.globl: main
<global1_label> : <global1_type> <global1_val>
…
<global1_label> : <global1_type> <global1_val>
```

Walk each 3AC Procedure, output each quad

```
enter main
```

Generating Code for Quads

Code Generation

Generating Code for Quads

Code Generation

enter <proc>

leave <proc>

<opd> := <opd>

<opd> := <opr> <opd>

<opd> := <opd> <opr> <opd>

<lbl>: <INSTR>

ifz <opd> goto <lbl>

goto Li

nop

call <name>

setin <int> <operand>

getin <int> <operand>

setout <int> <operand>

getout <int> <operand>

Generating Code for Quads: enter/leave

Code Generation

On entry to the function:

- Set up the activation record

On exit from the function

- Break down the activation record

Make function prologue/epilogue

enter <proc>

leave <proc>

Prologue

```
pushq %rbp
movq %rsp, %rbp
addq $16, %rbp
subq $X, %rsp
```

Epilogue

```
addq $X, %rsp
popq %rbp
retq
```

Generating Code for Quads: enter/leave

Code Generation

enter <proc>

leave <proc>

src code	**3ac code**	**asm code**

int main(){
}
	enter main
	leave main
		lbl_main: pushq %rbp
		movq %rsp, %rbp
		addq $16, %rbp
		subq $0, %rsp
		addq $0, %rsp
		pushq %rbp
		retq

Prologue

```
pushq %rbp
movq %rsp, %rbp
addq $16, %rbp
subq $X, %rsp
```

Epilogue

```
addq $X, %rsp
popq %rbp
retq
```

Generating Code for Quads

Code Generation

- enter <proc>
- leave <proc>
- <opd> := <opd>
- <opd> := <opr> <opd>
- <opd> := <opd> <opr> <opd>
- <lbl>: <INSTR>
- ifz <opd> goto <lbl>
- goto Li
- nop
- call <name>
- setin <int> <operand>
- getin <int> <operand>
- setout <int> <operand>
- getout <int> <operand>

For assignment-style quads…

1) Load operand src locations into registers
2) Compute a value to register
3) Store result at dst location

Assignment-Style Quads

Code Generation

3AC

[a] := [b] + 4

SymOpd
Symbol location: “gbl_a”

SymOpd
Symbol location: “-24(%rbp)

ASM

1) `movq -24(%rbp), %rax`
1) `movq $4, %rbx`
2) `addq %rbx %rax`
3) `movq %rax (gbl_a)`

For assignment-style quads…

1) Load operand src locations into registers
2) Compute a value to register
3) Store result at dst location

Questions?

Code Generation