University of Kansas | Drew Davidson

Function Codegen

Announcements

Administrivia

In depth 3ac -> x64 statement translation

References

Tutorials

Screencasts and references

- Windows Setup on WSL
- FIRST/FOLLOW
- X64 Arithmetic
- Stmt 3AC->X64

Readings

Primary reference for the course

- Table of contents
- All readings, single-page

Last Time

Review: Statement Codegen

From Quads to Assembly

- Approach Overview
- Planning out memory
- Writing out x64

Code generation

Generating Code for Quads

Review – Statement 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>

setret <int> <operand>

getret <int> <operand>

Last Time

Statement Codegen

From Quads to Assembly

- Approach Overview
- Planning out memory
- Writing out x64

You Should Know

- How to set up/break down an activation record
- The basic formula for turning most quads into x64

Code generation

This Time

Function Codegen

Handling Calls and Returns

- Respecting binary code conventions
- Translating interprocedural quads

Code generation

Functions are an Illusion!

Function Codegen – Respecting Conventions

Program state is just:

- Bytes in registers
- Bytes in memory

The compiler must ensure caller-callee interoperability

- Make caller places values where callee expects (and vice-versa)

Interoperability Conventions

Function Codegen

The Callee needs to trust that the caller put data where it needs to be

- **Memory layout**
  - Stack grows down
  - Denoted by %rsp
- **syscall args**
  - Which syscall: %rax
  - First param: %rdi
  - Second param: %rsi

Programs are basically a series of trust falls

Application Binary Interfaces

Function Codegen

Ensure interoperability between modules

- Maybe even between compilers!

Calling conventions

- One part of an ABI
- Indicate where arguments are passed
- Which registers can be changed
- Where the AR is restored

Modules all work together to support programmer "intent"

Application Binary Interfaces

Function Codegen

Ensure interoperability between modules

- Maybe even between compilers!

Calling conventions

- One part of an ABI
- Indicate where arguments are passed
- Which registers can be changed
- Where the AR is restored

System V AMD 64 Calling convention

1st argument: %rdi
2nd argument: %rsi
3rd argument: %rdx
4th argument: %rcx
5th argument: %r08
6th argument: %r09
7th+ argument: on stack R-to-L

This Time

Function Codegen

Handling Calls and Returns

- Respecting binary code conventions
- Translating interprocedural quads

```
setin <int> <operand>
getin <int> <operand>
setret <int> <operand>
getret <int> <operand>
call <name>
```

Code generation

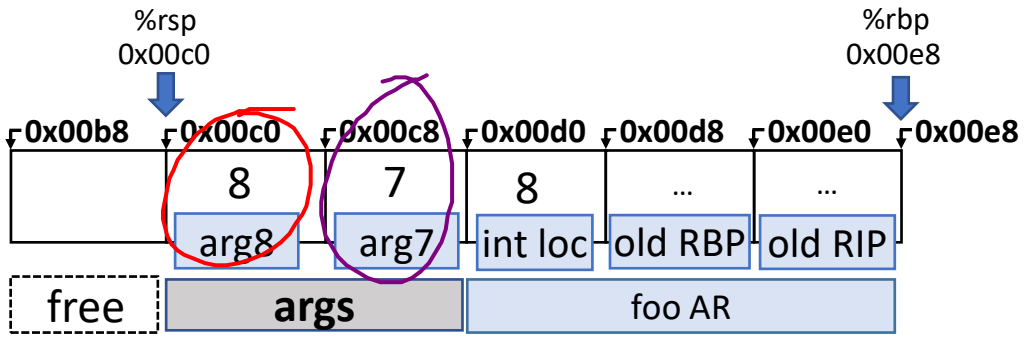
Setting Arguments in Caller

Function Codegen

```
void bar(int f1, int f2, int f3, int f4, int f5, int f6, int f7, int f8){
  int b;
  b = f8;
}
void foo(){
  int loc;
  loc = 8;
  bar(1,2,3,4,5,6,7,loc);
}
```

X64 for call to bar

```
movq $1, %rdi
movq $2, %rsi
movq $3, %rdx
movq $4, %rcx
movq $5, %r8
movq  $6, %r9
pushq  $7
movq -24(%rbp), %r12
pushq %r12
callq bar
```



This Time

Function Codegen

Handling Calls and Returns

- Respecting binary code conventions
- Translating interprocedural quads

```
setin <int> <operand>
getin <int> <operand>
setret <int> <operand>
getret <int> <operand>
call <name>
```

Code generation

Using Arguments in Callee

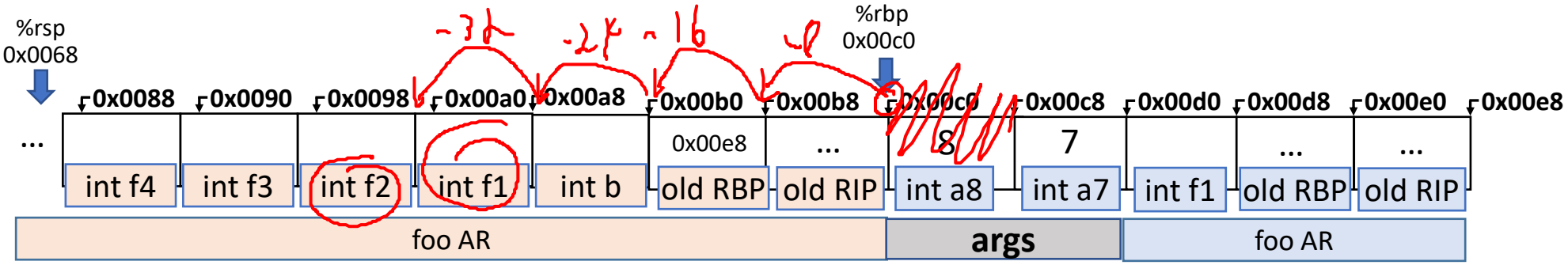
Function Codegen

Args 1 – 6

- Were passed in register
- Should be allocated saved/in current AR

getarg 1, [f1]	`movq %rdi, -32(%rbp)`
getarg 2, [f2]	`movq %rsi, -40(%rbp)`
getarg 3, [f3]	`movq %rdx, -48(%rbp)`
getarg 4, [f4]	`movq %r08, -56(%rbp)`
getarg 5, [f5]	`movq %r09, -64(%rbp)`
getarg 6, [f6]	`movq %r09, -64(%rbp)`

(keeps them from getting clobbered if the callee calls something else)



Using Arguments in Callee

Function Codegen

Args 7+

- Were pushed on stack
- Offset can be calculated statically - we just need to math it out

Formal position (in callee) = %rbp + 8 * (#args – argIdx)

Arg index 7 (of 8 total) @ %rbp + 8 * (8 – 7)

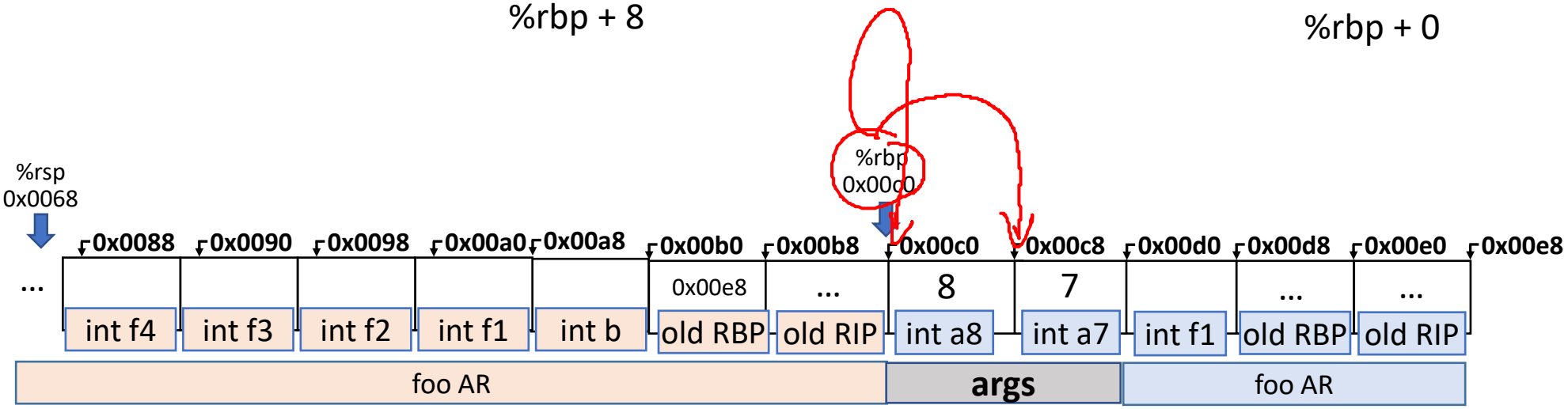
=

%rbp + 8

Arg index 8 (of 8 total) @ %rbp + 8 * (8 – 8)

=

%rbp + 0



Using Arguments

Function Codegen

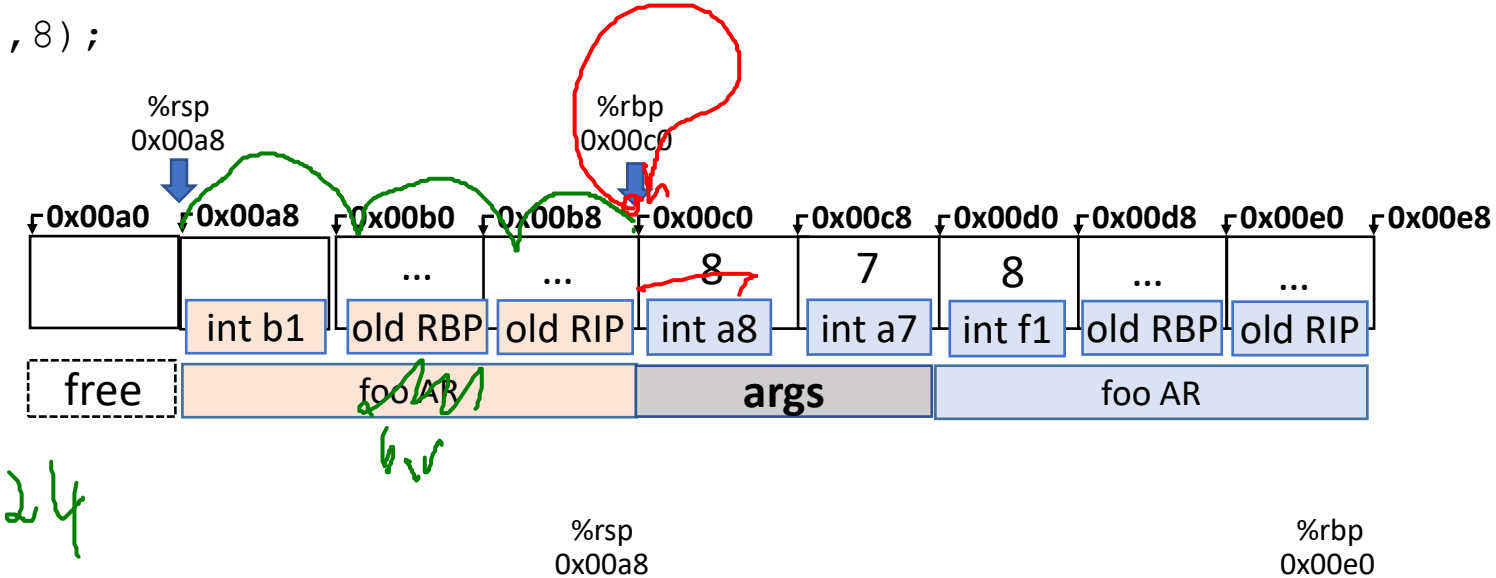
```
void bar(int a1, int a2, int a3, int a4, int a5, int a6, int a7, int a8){
  int b1;
  b1 = a8;
}
void foo(){
  int f1;
  f1 = 8;
  bar(1,2,3,4,5,6,7,8);
}
```

put a8 into register
put the register into b1

X64 for b1 := a8

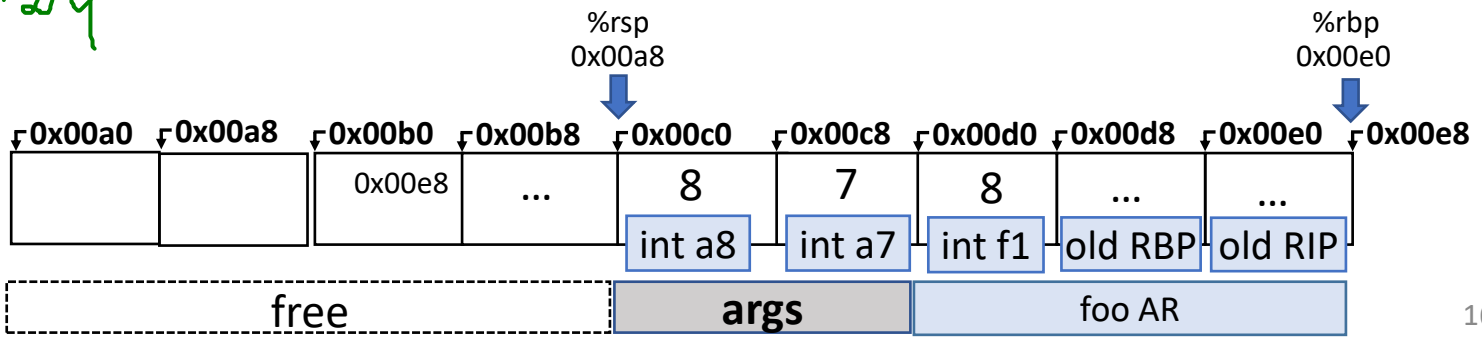
```
movq (%rbp), %r11
movq %r11, -12(%rbp)
addq $16, %rbp
subq $8, %rsp
```

-24



X64 for leave bar

```
addq $8, %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>
- nop
- goto <lbl>
- ifz <opr> goto <lbl>
- setarg <idx> <opd>
- getarg <idx> <opd>
- setret <opd>
- getret <opd>
- call <name>

Two things to do with a call

1. Transfer into the callee
   callq <LBL_FN>
2. Cleanup the argument stack
   addq <X> where <X> is the size of the actuals pushed on the stack

Argument Cleanup

Parameters

Note that arguments weren't removed by the caller

- That leaves it up to the caller to clean them off the stack
- As part of translating a call quad to assembly, add cleanup AFTER the callq instruction

Argument Cleanup

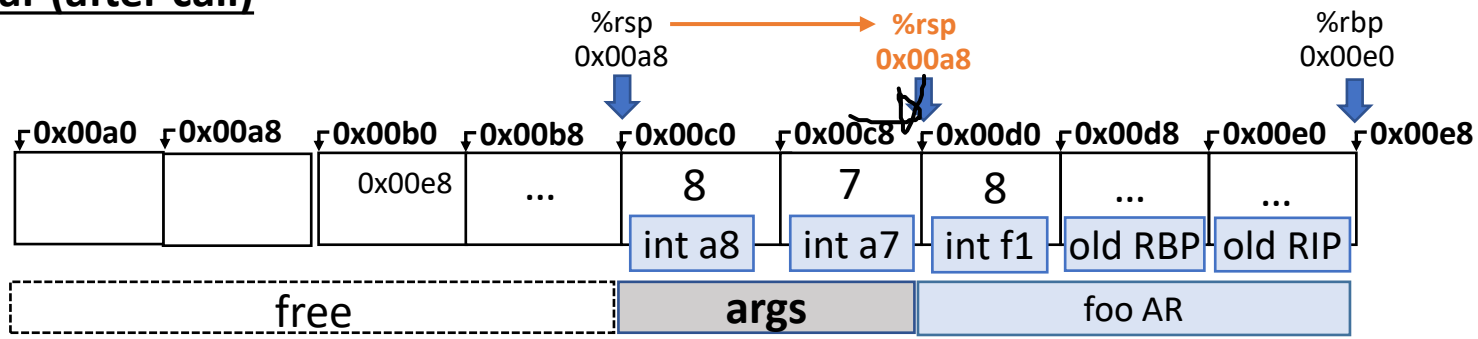
Finishing off ARs

```
void bar(int a1, int a2, int a3, int a4, int a5, int a6, int a7, int a8){
  int b1;
  b1 = a8;
}
void foo(){
  int f1;
  f1 = 8;
  bar(1,2,3,4,5,6,7,8);
}
```

X64 for call to bar (after call)

setins
- movq $1, %rdi
- movq $2, %rsi
- movq $3, %rdx
- movq $4, %rcx
- movq $5, %r8
- movq $6, %r9
- pushq $7
- movq -12(%ebp), %r12
- pushq %r12

call
- callq bar
- **addq $16, %rsp**



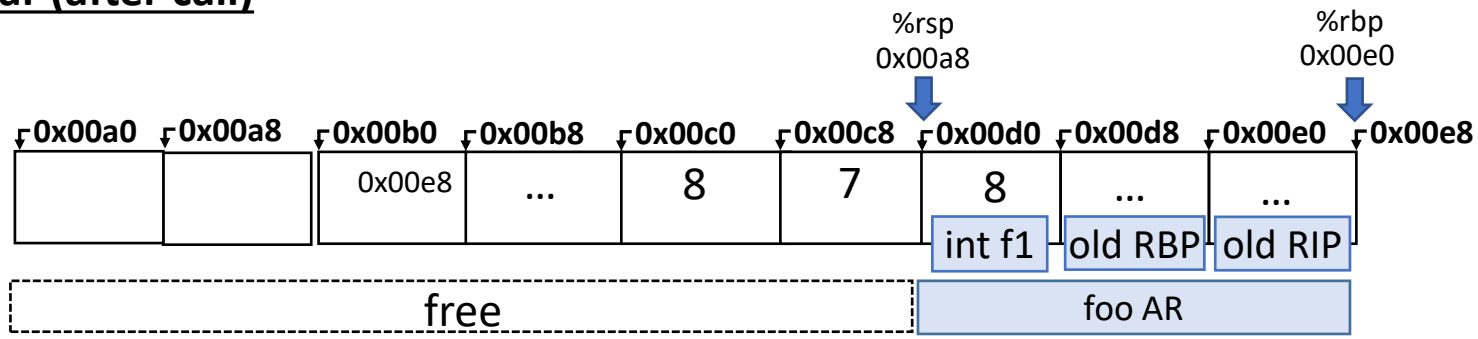
Argument Cleanup

Finishing off ARs

```
void bar(int a1, int a2, int a3, int a4, int a5, int a6, int a7, int a8){
  int b1;
  b1 = a8;
}
void foo(){
  int f1;
  f1 = 8;
  bar(1,2,3,4,5,6,7,8);
}
```

X64 for call to bar (after call)

```
movq $1, %rdi
movq $2, %rsi
movq $3, %rdx
movq $4, %rcx
movq $5, %r8
movq  $6, %r9
pushq  $7
movq -12(%ebp), %r12
pushq %r12
callq bar
addq $16, %rsp
```



Done For Today!

Code Generation

- We've basically got the required quads done!
  - Next, we'll look at "advanced" features (some of which we won't need for the projects)
    - Classes/structs
    - Pointers
    - Arrays
    - etc