Check-in 28

Review: Statement Code Generation

Write the code that a compiler might output for the following 3AC Code:

```
enter f
[a] := 6 ADD64 [g]
[g] := 6 ADD64 [b]
leave f
```

Assume:

- a is the first local in f and occupies 8 bytes
- b is the second local in f and occupies 8 bytes
- g is a global at label var_g and occupies 8 bytes
- There are 2 locals in f

Check-in 28 Solution

Review: Statement Code Generation

Write the code that a compiler might output for the following 3AC Code:

```
enter f
[a] := 6 ADD64 [g]
[g] := 6 ADD64 [b]
leave f
```

Assume:

- a is the first local in f and occupies 8 bytes
- b is the second local in f and occupies 8 bytes
- g is a global at label var_g and occupies 8 bytes
- There are 2 locals in f

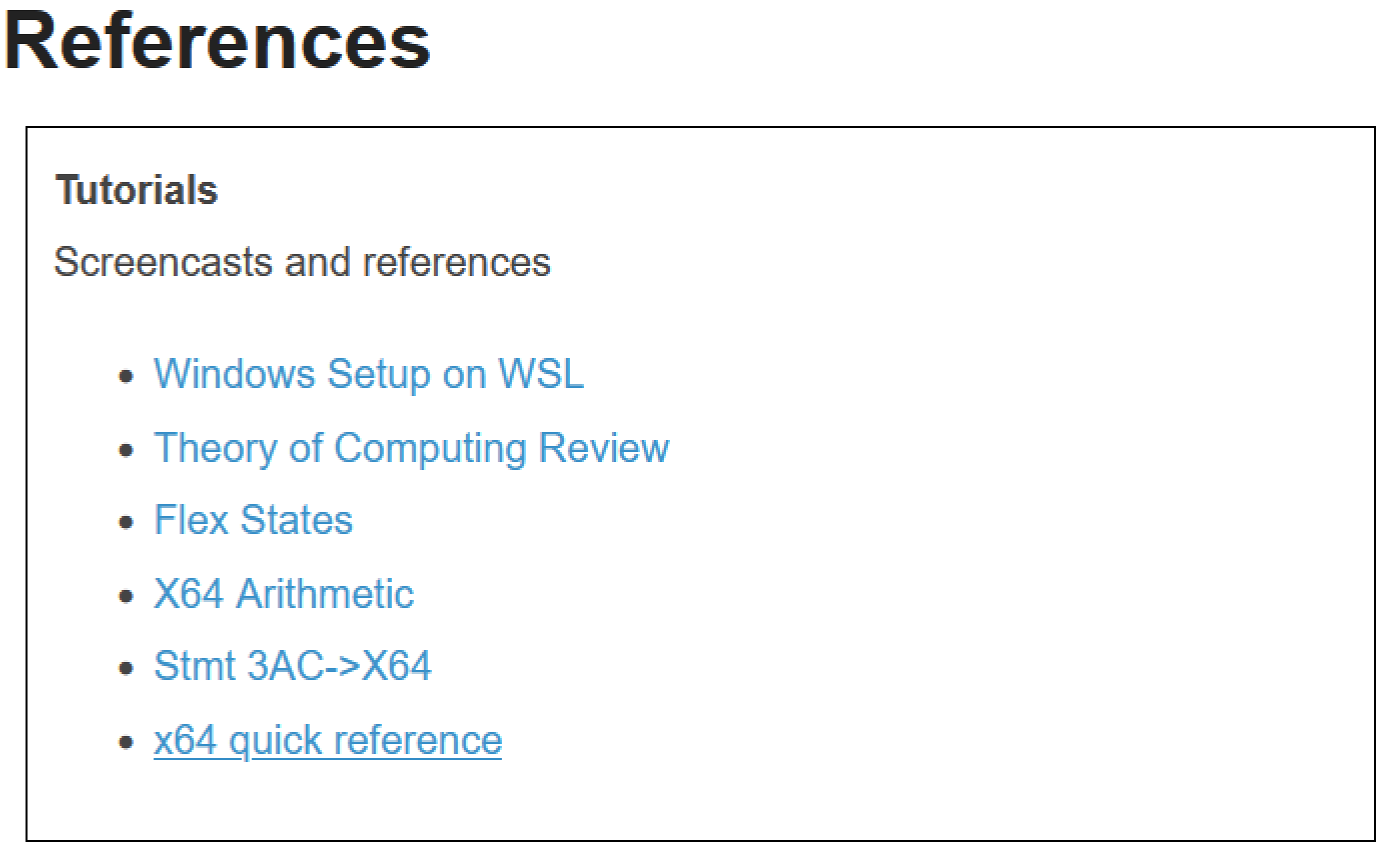
University of Kansas | Drew Davidson

Function Codegen

Announcements

Administrivia

In depth 3ac -> x64 statement translation



References

Tutorials

Screencasts and references

- Windows Setup on WSL
- Theory of Computing Review
- Flex States
- X64 Arithmetic
- Stmt 3AC->X64
- x64 quick reference

Last Time

Review: Statement Codegen

From Quads to Assembly

- Approach Overview
- Planning out memory
- Writing out x64

Handled Some Basic Quads

- Assignments
- Binary ops

Code generation

Generating Code for Quads

Review – Statement Code Generation

- ✓ enter <proc>
- ✓ leave <proc>
- ✓ call <name>
- ✓ <opd> := <opd>
- ✓ <opd> := <opr> <opd>
- ✓ <opd> := <opd> <opr> <opd>
- <lbl>: <INSTR>
- ifz <opd> goto <lbl>
- goto Li
- nop
- 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

Handled Some Basic Quads

- Assignments
- Binary ops

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 jumps

- Conditionals
- Unconditionals

Handling Calls and Returns

- Respecting binary code conventions
- Translating interprocedural quads

Code generation

Generating Code for Quads

Handling Jumps (Unconditional)

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

3AC Code

```
LBL_1: nop
LBL_2: goto LBL_1
```

X64 Code

```
LBL_1: nop
LBL_2: jmp LBL_1
```

Generating Code for Quads

Handling Jumps (Conditional)

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

-24(%rbp)

3AC Code

```
LBL_1: ifz [tmp1] goto LBL_1
```

X64 Code

```
LBL_1: movq -24(%rbp), %rdi
       cmpq $0, %rdi
       je LBL_1
```

Generating Code for Quads

Handling Jumps (Conditional)

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

Source Code

```
while(a < b){
  a = 1 + a;
}
...
```

3AC Code

```
L_1: [tmp1] = [a] LT64 [b]
          ifz [tmp1] goto L_2
          [a] = [a] ADD64 1
          goto L_1
L_2: nop
```

X64 Code

```
L_1: movq -24(%rbp), %rax
     movq -32(%rbp), %rbx
     movq $0, %rdi
     cmpq %rbx, %rax
     setlt %dil
     movq %rdi, -40(%rbp)
     movq -40(%rbp), %r11
     cmpq $0, %r11
     je L_2
     movq $1, %rax
     movq -24(%rbp), %rbx
     addq %rbx, %rax
     movq %rax, -24(%rbp)
     jmp L_1
L_2: nop
```

This Time

Function Codegen

Handling jumps

- Conditionals
- Unconditionals

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)

OK KO!
YOUR WORLD IS AN ILLUSION
Written & Storyboarded by
Danny Ducker & Parker Simmons

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

Return value: %rax

This Time

Function Codegen

Handling jumps

- Conditionals
- Unconditionals

Handling Calls and Returns

- Respecting binary code conventions
- Translating interprocedural quads

Code generation

This Time

Function Codegen

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

Handling jumps

- Conditionals
- Unconditionals

Handling Calls and Returns

- Respecting binary code conventions
- Translating interprocedural quads

Code generation

Returning Values and Accessing Return Values

Function Codegen: setret / getret

System V ABI: Return values through %rax

- Set %rax in the callee
- Get %rax in the caller

Source code

```
int foo(){
  v = bar();
  return 4;
}
```

3AC code

```
fn_foo: enter foo
        call bar
        getret [v]
        setret 4
        goto lv_foo
lv_foo  leave
```

movq %rax, (glb_v)

movq $4, %rax

This Time

Function Codegen

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

Handling jumps

- Conditionals
- Unconditionals

Handling Calls and Returns

- Respecting binary code conventions
- Translating interprocedural quads

Code generation

Setting Arguments in Caller

Function Codegen: setin

```
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 v;
  v = 8;
  bar(1,2,3,4,5,6,7,v);
}
```

3AC code

```
setin 1, 1
setin 2, 2
setin 3, 3
setin 4, 4
setin 5, 5
setin 6, 6
setin 7, 7
setin 8, [v]
call fn_bar
```

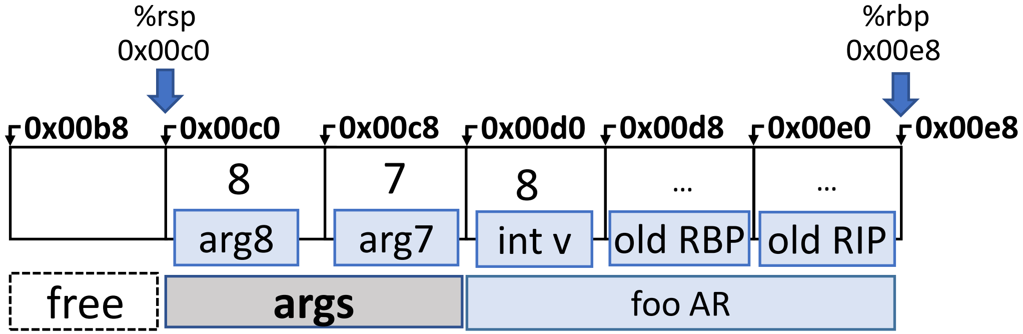
X64 for call to bar

```
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
```

System V 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

Return value: %rax



This Time

Function Codegen

- ✓ enter <proc>
- ✓ leave <proc>
- ✓ call <name>
- ✓ <opd> := <opd>
- ✓ <opd> := <opr> <opd>
- ✓ <opd> := <opd> <opr> <opd>
- ✓ <lbl>: <INSTR>
- ✓ goto Li
- ✓ nop
- ✓ ifz <opd> goto <lbl>
- ✓ setret <int> <operand>
- ✓ getret <int> <operand>
- ✓ setin <int> <operand>
- getin <int> <operand> ←

Handling jumps

- Conditionals
- Unconditionals

Handling Calls and Returns

- Respecting binary code conventions
- Translating interprocedural quads

Code generation

Using Arguments in Callee

Function Codegen: getin

```
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 v;
  v = 8;
  bar(1,2,3,4,5,6,7,v);
}
```

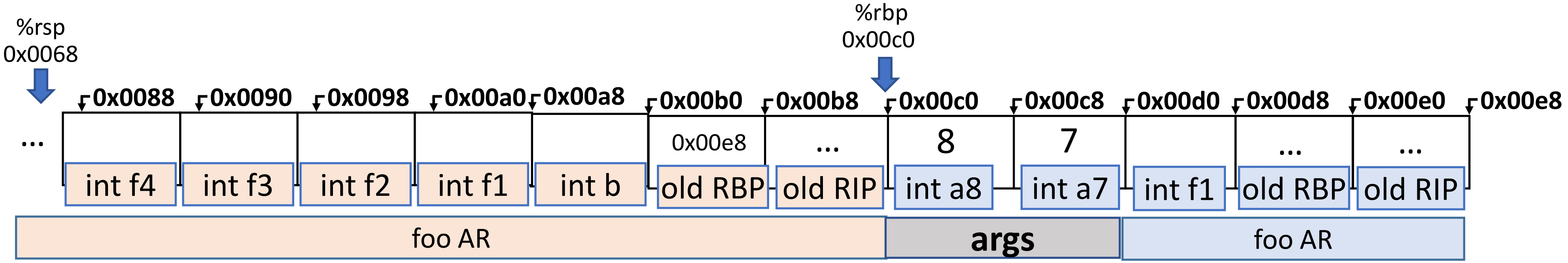
The getin quads appear right after the enter quad

Assume current AR is already set up

System V 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

Return value: %rax



Using Arguments in Callee

Function Codegen: getin

Args 1 – 6

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

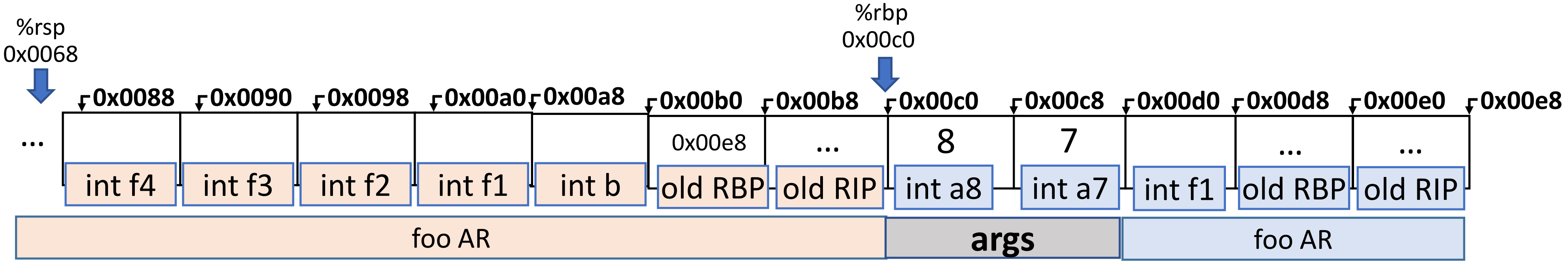
getarg 1, [f1]	`movq %rdi, -32(%rbp)`
getarg 2, [f2]	`movq %rsi, -40(%rbp)`
getarg 3, [f3]	`movq %rdx, -48(%rbp)`
getarg 4, [f4]	`movq %r08, -56(%rbp)`
getarg 5, [f5]	`movq %rdx, -48(%rbp)`
getarg 6, [f6]	`movq %r09, -64(%rbp)`

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

System V 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

Return value: %rax



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)

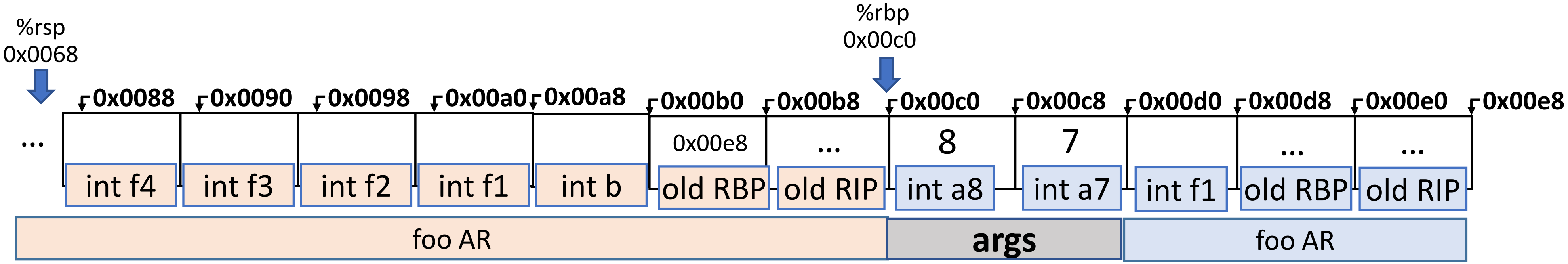
=

%rbp + 8

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

=

%rbp + 0



This Time

Function Codegen

- ✓ enter <proc>
- ✓ leave <proc>
- ✓ call <name> ← REVISIT THIS!
- ✓ <opd> := <opd>
- ✓ <opd> := <opr> <opd>
- ✓ <opd> := <opd> <opr> <opd>
- ✓ <lbl>: <INSTR>
- ✓ goto Li
- ✓ nop
- ✓ ifz <opd> goto <lbl>
- ✓ setret <int> <operand>
- ✓ getret <int> <operand>
- ✓ setin <int> <operand>
- ✓ getin <int> <operand>

Handling jumps

- Conditionals
- Unconditionals

Handling Calls and Returns

- Respecting binary code conventions
- Translating interprocedural quads

Code generation

This Time

Function Codegen

- ✓ enter <proc>
- ✓ leave <proc>
- ✓ call <name> ← **REVISIT THIS!**
- ✓ <opd> := <opd>
- ✓ <opd> := <opr> <opd>
- ✓ <opd> := <opd> <opr> <opd>
- ✓ <lbl>: <INSTR>
- ✓ goto Li
- ✓ nop
- ✓ ifz <opd> goto <lbl>
- ✓ setret <int> <operand>
- ✓ getret <int> <operand>
- ✓ setin <int> <operand>
- ✓ getin <int> <operand>

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

Code generation

Argument Cleanup

Parameters

We pushed arguments 7+ on the Stack

- We never popped them back off!
- System V ABI: Delegates stack cleanup to the caller

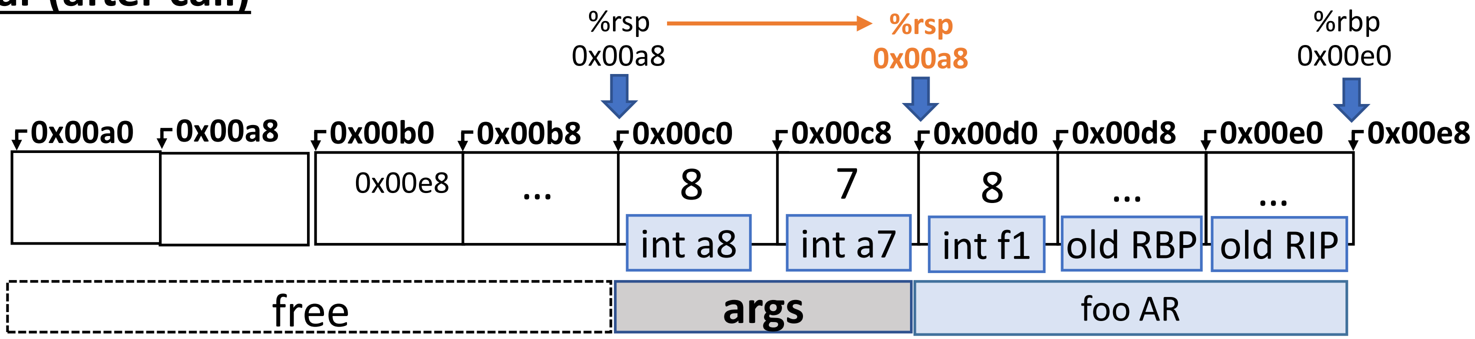
Argument Cleanup

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
```



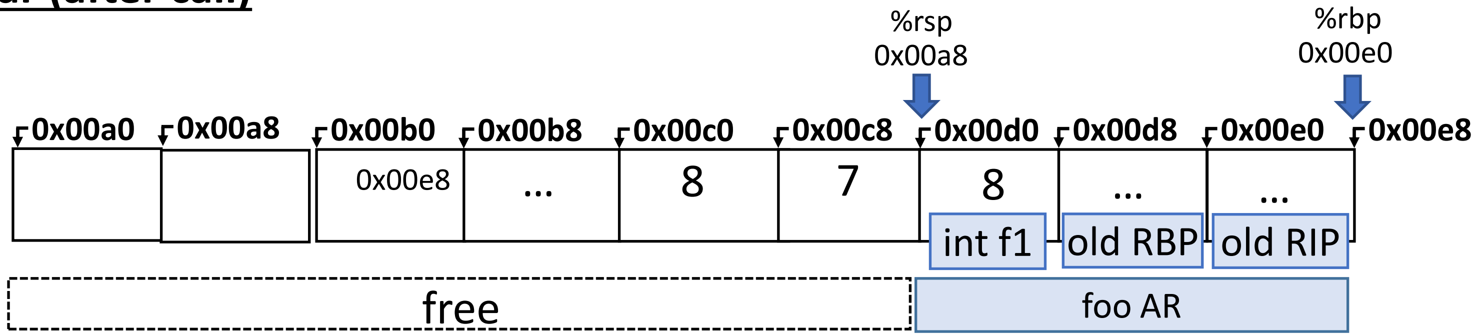
Argument Cleanup

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
```



%rsp 0x00a8

%rbp 0x00e0

0x00a0	0x00a8	0x00b0	0x00b8	0x00c0	0x00c8	0x00d0	0x00d8	0x00e0	0x00e8
		0x00e8	…	8	7	8	…	…	
						int f1	old RBP	old RIP	
free						foo AR			

Done For Today!

Function 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