CS 33: Introduction to Computer Organization

Week 3

Agenda

- x86-64 Basics + Control in a nutshell
- x86-64 Stack Basics
- x86-64 Advanced Stack
- Practice Midterm Excerpt Review

Admin

- Homework 2 “due” tonight
- Midterm 1: Wednesday, October 21st
- Practice Midterm 1 has been posted

Midterm 1

- Next Wednesday (10/21) in class.
- Covers the reading/lecture material up to/including next Monday's (10/19) lecture.
- Covers any assignments (homeworks, labs)
- If any labs or assignments are posted between now and the midterm, you are also expected to have at least read them.

Midterm 1

- Open book
- Open note
- Open anything-that-might-be-useful-except-for-electronics
  - Printed out Wikipedia articles
  - Printed out homework/lab solutions
  - Excerpts from that digital copy of the book that you totally own.

Midterm 1

- Basic topics (AKA topics that a normal professor would cover on the test):
- Boolean/bitwise arithmetic
  - Bitwise manipulations to solve problems in log n time where n is the number of bits.
- x86-64 general assembly, stack protocol, control
  - Given some assembly, complete the C code or vice versa.

Midterm 1

- Basic topics (AKA topics that a normal professor would cover on the test):
- Control flow in assembly, specifically switch cases and loops.
  - Engineer x86-64 assembly or reverse engineer C code from x86-64 assembly
- Data storage
  - Given some memory configuration and/or assembly, determine something about the data types (sizes, alignments, lengths, etc). Endianness

Midterm 1

- Advanced concepts (AKA topics that Prof. Eggert might cover on the test):
- Tail recursion optimization – Optimization if the last action in a recursive function is to call a function.
- Non-Conventional/Optimized Procedure calls
- Wrap vs. Trap semantics and overflow conditions
- ...and other tricky details (arrays of structs, caller/callee saved registers, multiple precision arithmetic).

Midterm 1

- There are also things that the professor has mentioned offhand that may come up. You probably won't be expected to know these in detail, but you may be expected to be aware of them and answer questions that have them in mind:
  - “asm” declarations in C: non-standardized blocks of code that allow you to specify assembly instructions in high level code.
  - How assembly instructions appear in byte code: ex. call instructions require 5 bytes, many quad-word instructions require an extra byte, etc.
  - How assembly deals with the __int128 data type

Midterm 1

- The existence of other control flags? Stuff like the Trap Flag, Interrupt Flag? Maybe? This is probably a stretch.
- The different types of jump (relative, absolute, indirect)
- Optimizer tradeoffs: What is a higher level of optimization going to do. What's the tradeoff of invoking a higher level of optimization (the aliasing problem)

x86-64 Assembly Basics

- Consider the following assembly snippet:
  - movq %rsi, %rax
- What can we say about the data type stored in %rsi (ie, pointer? value? signed? etc.)

x86-64 Assembly Basics

- Consider the following assembly snippet:
  - movq %rsi, %rax
- What can we say about the data type stored in %rsi?
  - From this, not much.
  - Because each register simply holds a bit vector, it is difficult to make assumptions about what the original value is without some context.
- For example, if the previous line was:
  - movq (%rsi), %rbx
- It's probably safe to assume that %rsi contains a pointer

x86-64 Assembly Basics

- Final note:
- Memory to memory operations are prohibited (operations where both operands are memory accesses):
  - movq (%rax), (%rbx)
  - addq (%rax), (%rbx)
- You'll need to pull at least one value into a register
  - addq (%rax), (%rbx)
  - becomes...
  - movq (%rax), %rax
  - addq %rax, (%rbx)

x86-64 Assembly Basics

- As a quick note:
  - jmp 0x400356
    - Jump to the instruction at that address
  - jmp *%rax
    - Jump to the instruction at the address that is formed by the bit configuration in %rax
  - jmp *(%rax)
    - Using %rax as an address, follow %rax into memory and find the address stored in memory. Jump to *that* address

x86-64 Assembly nuances: Instruction suffixes

- Recall that many common operations specify a suffix to indicate the size of the data type (b for 8-bit, w for 16-bits, l for 32-bits, q for 64-bits)
- It is possible to move from a smaller container to a larger container.
- Assume that %dh = 0xCD, %eax = 0x98765432
- movb %dh, %eax
  - What's the result?

x86-64 Assembly nuances: Instruction suffixes

- It is possible to move from a smaller container to a larger container.
- Assume that %dh = 0xCD, %eax = 0x98765432
- movb %dh, %eax
  - What's the result? It's not allowed (suffix mismatch). The destination has a 32-bit length while the mov**b** expects only to move a byte.
- movb %dh, %al
  - Result: %eax = 0x987654CD
- However, that might not be what you want. Maybe you want %eax = 0xCD.

x86-64 Assembly nuances: Instruction suffixes

- movsXY : move and sign extend from size X to size Y.
  - Ex: movsbl : move a byte from the source and sign extend it to a long (4 bytes)
- movzXY : move and zero extend from size X to size Y.
  - Ex: movzbw : move a byte from the source and zero extend it to a word (2 bytes)

x86-64 Assembly nuances: Instruction suffixes

- %dh = 0xCD, %eax = 0x98765432
- movsbl %dh, %eax
  - Result: %eax = 0xFFFFFFCD
- movzbl %dh, %eax
  - Result: %eax = 0x000000CD
- movzbw %dh, %eax?
  - Result: Not allowed. Sign extending to w (16-bits) but to %eax (32-bit container).

x86-64 Assembly nuances: Instruction suffixes

- As a final note about mov, the size of the prefix must match the operands. You cannot have:
  - movl %ax, (%esp) // Can't move a 32-bit quantity from a 16-bit register
  - movl %eax, %dx // Can't move a 32-bit quantity into a 16-bit register.

x86-64 Assembly nuances: Instruction suffixes

- Additionally memory references match all sizes:
  - movb %al, (%rbx)
  - movw %ax, (%rbx)
  - movq %rax, (%rbx)
  - All allowed. The data will be moved to memory starting at that address based on the data type size
- However:
  - movb %al, (%ebx)
  - Is not meaningful on x86-64 because the %ebx register is only the lower 32-bits while addresses are 48-bits.

x86-64 Assembly nuances: Instruction suffixes

- Say %rax = 0xFFFFFFFFFFFFFFFF
- addb $1, %al
- %rax = ?

x86-64 Assembly nuances: Instruction suffixes

- Say %rax = 0xFFFFFFFFFFFFFFFF
- addb $1, %al
- %rax = 0xFFFFFFFFFFFFFF00
- The addition is performed as an 8-bit addition. The carry out bit is dismissed and the truncated 8-bit result is stored into the least significant byte of %rax.

x86-64 Assembly: Control Flags

- As the datalab has probably demonstrated, it would often be convenient if we could extract some information about certain values (is negative, is equal to zero, etc)
- Control flags do just that.

x86-64 Assembly: Control Flags

- The control flags are single bit values.
- Each of the control flags are located in a distinct RFLAGS register.
- Many arithmetic operations change the control flags as a side effect.
- Some instructions will *only* change the control flags.

x86-64 Assembly: Control Flags

- Carry Flag (CF)
  - CF = 1 if the most recent operation caused the most significant bits to have a carry out. Otherwise, CF = 0.
  - Informal usage: the purpose of this is to check for unsigned overflow.
  - If t = a+b, then CF = 1 if
    - (unsigned) t < (unsigned) a
  - Set by most arithmetic instructions and some bitwise instructions, but not inc or dec.
  - Why not? Let's consult our esteemed book.

x86-64 Assembly: Control Flags

- “For reasons that we will not delve into, the INC and DEC instructions set the overflow and zero flag, but they leave the carry flag unchanged.”
  - Computer Systems: A Programmer's Perspective, 3^{nd} Ed., pg. 201
- Thanks, book.

x86-64 Assembly: Control Flags

- Zero Flag (ZF)
  - ZF = 1 if the result of the most recent operation is zero. Otherwise, ZF = 0.
  - Informal usage: check if two values are equal, check if an operation resulted in a zero.
  - ex. if t = a+b, then ZF = 1 if:
    - t == 0

x86-64 Assembly: Control Flags

- Sign Flag (SF)
  - SF = 1 if the result of the operation has the most significant bit as 1. Otherwise SF = 0.
  - This sets the flag if the number is negative
  - If t = a+b, then SF = 1 if:
    - $t < 0$
  - Set by arithmetic (except for mul and div), boolean/bitwise, cmp, and test.

x86-64 Assembly: Control Flags

- Overflow Flag (OF)
  - If t = a+b, then SF = 1 if:
    - (a < 0) == (b < 0) && (t < 0) != (a < 0)
  - OF is set if the above expression is 1.
  - Informal usage: This effectively checks for signed overflow.

x86-64 Assembly: Control Flags

- The Carry Flag detects unsigned overflow and the Overflow Flag detects signed overflow.
- Which flag should be set in the following operation:
- addl %eax, %ebx

x86-64 Assembly: Control Flags

- From the machine perspective, it does not do anything to distinguish between signed and unsigned add. Remember, a register is a set of bits and the add operation simply manipulates the bits..
- Regardless of what the programmer's original intent was, both the Carry and Overflow flags will be set.
- ...it's just that depending on whether it was supposed to be signed/unsigned, one of the flags isn't going to mean a whole lot.

x86-64 Assembly: Control Flags

- As a clarification, if an instruction is going to change a control flag, after each operation, the control flag will either be set to 0 or 1.
- For example, addl %eax, %ebx will always set the ZF, but it will set ZF = 0 if the resulting sum is non-zero and it will set ZF = 1 if the resulting sum is zero.

x86-64 Assembly: Control Flags

- cmp S2, S1 : Sets the flags based on S1 – S2, but doesn't change S1.
- test S2, S1 : Sets the flags based on S1&S2 but does not change S1.
  - Most often used as test %eax, %eax in order to just get the flag info of a particular value.

x86-64 Assembly: Control Flags

- After setting the flags, you can use them in two ways:
  1. Manually set a register based on the state of the flags.
  2. Use a conditional instruction which does something based on the state of the flags.

x86-64 Assembly: Control Flags

- 1. Manually set a register based on flags with the set_ family of ops.
- The set_ family of operations read from the RFLAGS register and set the destination operand. Ex:
  - sete %al : sets %al = ZF
  - sets %al : sets %al = SF
- ...and etc.
- However, there are also ones that set the register based on combinations of flags that indicate other information.

x86-64 Assembly: Control Flags

- setg D : D = ~(SF^OF)&~ZF
  - Sets if greater than. Used in conjunction with something like: cmp %eax, %ebx. If %ebx is greater than %eax, D will be 1 (why?).
- setge D : D = ~(SF^OF)
  - Sets if greater than or equal to
- ...and so on (pg. 203)

x86-64 Assembly: Control Flags

- 2. Use conditional operations:
- Normal operations will always perform a particular action when it is reached (ex. mov)
- Conditional instructions will either perform or not perform the operation depending on the state of the control flags.
- Conditional move (cmov_)
  - cmove S, D : D = S, but only if ZF is 1.
  - cmovs S, D : D = S, but only is SF is 1.
  - ...and so on (pg. 217). The suffixes are the same as in set_

x86-64 Assembly: Control Flags

- In C, we can call functions or use conditional statements to jump to other parts of code rather than executing the next instruction.
- This is accomplished by setting the %rip/%eip register to the target address.
- With assembly, this is done using the jump instruction:
- jmp Label // Unconditionally jump to the Label

x86-64 Assembly: Control Flags

- There are also conditional jumps:
- je, jne, etc. (pg. 206) which jumps only if the flag configuration is met.
- The suffixes also match that of set_.

x86-64 Assembly: Some Context

- Hopefully at this point, the basics of assembly are clear. That is, if you see a snippet of assembly, you should be able to describe the operations that are happening (if not always understand what the goal is).
- Now, let's establish some context as to how a program is run.
- Namely, its use of memory

x86-64 Assembly: Some Context

- What we know:
  - Data is stored in memory (and in registers when we need to operate on them)
- int i = 100;
- If you do &i, you may get something like 0x7FFF4980, which is it's address in memory.
- What address in memory? I thought this value was stored in a register. That's how we can use assembly instructions on it.

x86-64 Assembly: Some Context

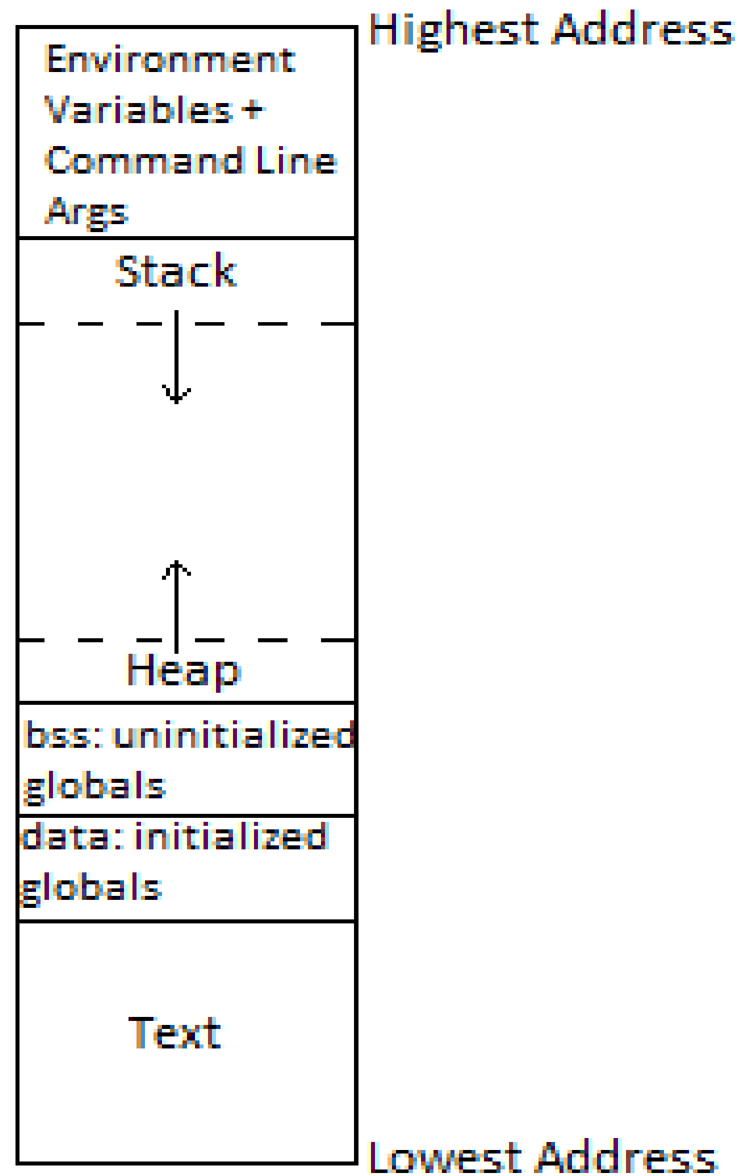
- The fact is, while registers can be thought of as “where all of the work is done”, registers are more accurately considered to be temporary storage for data that is actually stored in memory.
- So when I have the (totally pointless) function:

```
int foo(int arg)
{
  int a = 10;
  a += arg;
  return a;
}
```

- We can think of “int a” as belonging in memory.
- Where?

x86-64 Assembly: Some Context

- This is the memory that an executing program sees:
- We'll talk about “bss” and “data” later.
- Stack: storage for local variables
  - Ex. int a is stored here.
- Heap: storage for dynamically allocated data.
- Text: The executable code of the running program.



x86-64 Assembly: Some Context

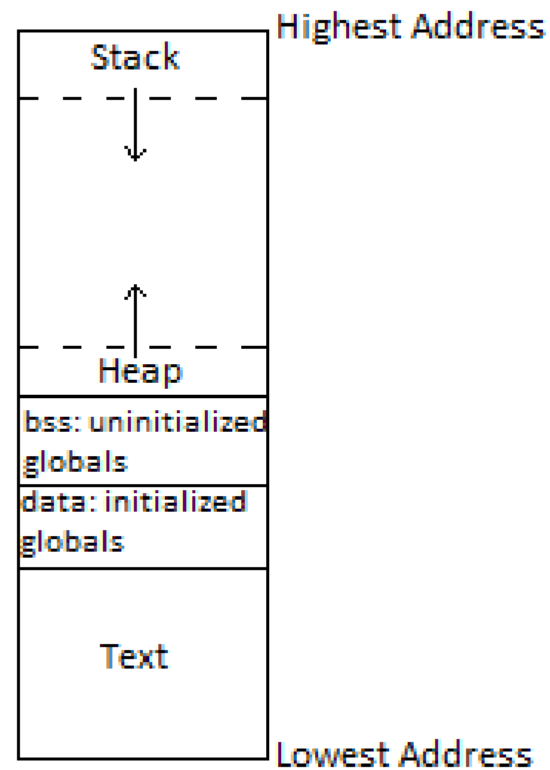
- Notice that the “text” section contains the machine code of the current program.
- The currently running program is stored in memory, just like data.
- As a result, we keep track of what instruction we're executing the same way we keep track of data in memory: with a pointer.

x86-64 Assembly: Some Context

- This pointer is stored in register %eip (in 32-bit) or %rip (in 64-bit).
- “ip” stands for instruction pointer. You will come to know this in CS M151B as the “PC” or “program counter” (which is probably confusing for multiple reasons).
- This register simply contains the *address* of the instruction that is to be executed (note: not the bytes of the instruction itself).

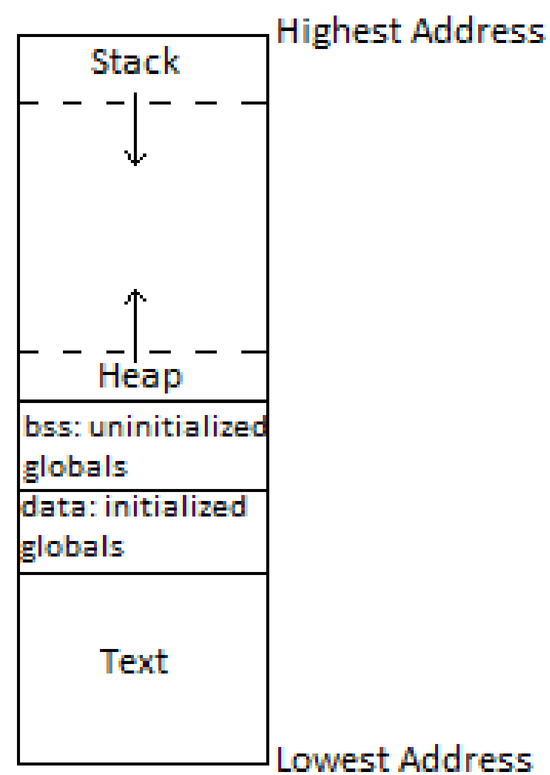
x86-64 Stack Convention

- The stack is a chunk of memory that spans from some address to the highest address in the memory space. (ex. 0x7FFF..F00 to 0x7FFF..FFF)
- Recall that the purpose of the stack is to maintain local variables that are needed during functions.
- As the program progresses and we need more memory to store our local variables, we need to increase the size of the stack.



x86-64 Stack Convention

- This is done by allocating and deallocating memory on the stack as necessary, but something to keep in mind is that the stack “grows downwards”.
- Which is to say when we grow the stack, we'll need to decrease the lower boundary. But more on that later
- However, we only want to store the variables that we need to use at the time.



x86-64 Stack Convention

- Consider this incredibly nonsensical code.
- At line 3 (in foo), we need to keep track of a, b, and c.
- At line 10 (in bar), we need to keep track of d, e, and f. However, because we need to return to foo, we still need to keep track of a, b, and c.
- At line 5 (in foo), after we're done with bar, we can completely forget about d, e, and f.

```
1. int foo(int a, int b)
2. {
3.   int c = a + b;
4.   bar(a, c);
5.   return c;
6. }

7.  int bar(int e, int f)
8.  {
9.    int d = e + f;
10.   return d;
11. }
```

x86-64 Stack Convention

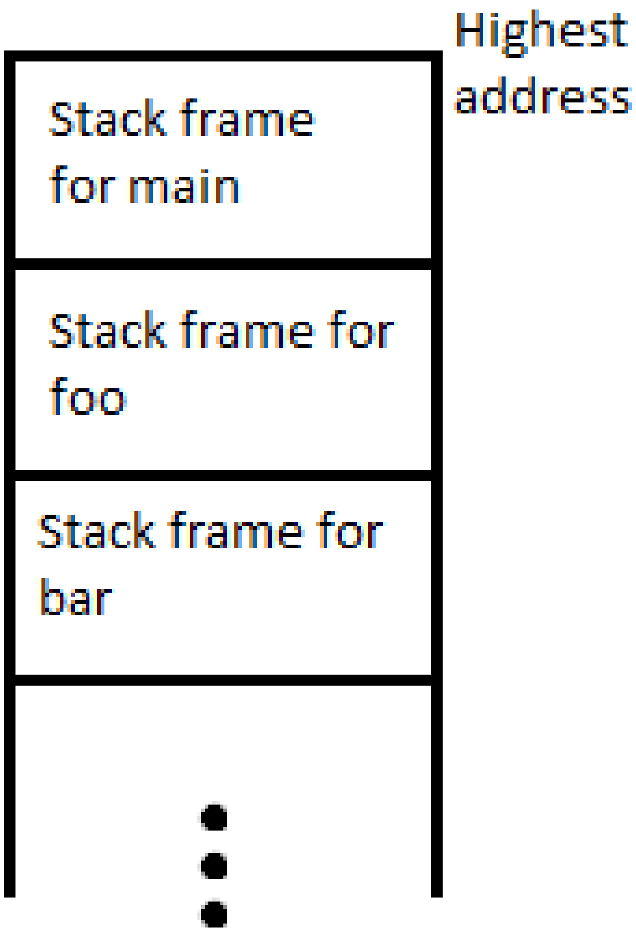
- When we want to call a new function, we need to store a new set of variables, while maintaining the old.
- By the time we access the old variables again, we will be completely finished with the newer ones.
- As a result, this suggests a last-in first-out (LIFO) policy for keeping track of everything.
- Hence, the stack.

```
int foo(int a, int b)
{
  int c = a + b;
  bar(a, c);
  return c;
}

int bar(int e, int f)
{
  int d = e + f;
  return d;
}
```

x86-64 Stack Convention

- This means, at a high level, we will treat the stack as a set of stack frames, each of which correspond to a function.
- If 'main' calls 'foo' and 'foo' calls 'bar', this is the organization of the stack while execution is in bar.
- Once 'bar' is completed, the stack frame for bar can be popped off the stack.

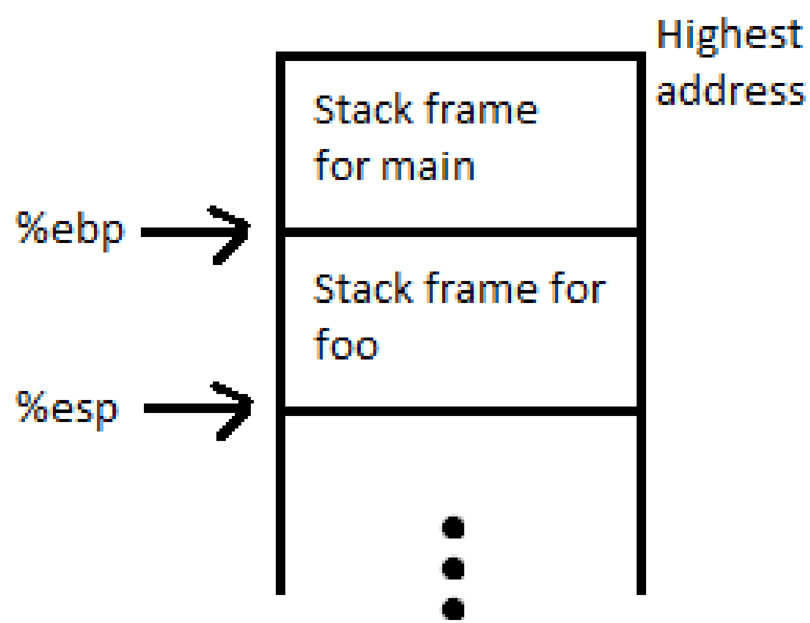


x86-64 Stack Convention

- But stack frames can be of arbitrary size since functions can have an arbitrary amount of variables.
- As a result, we need to keep track of the range of the stack frame.
- That sounds like a job for registers.

x86-64 Stack Convention

- Two of them in fact:
  - %rbp: Base pointer, points to the base address of the frame (highest address), sort of
  - %rsp: Stack pointer, points to the top of the stack frame (lowest address).



x86-64 Stack Convention

- How are %rbp and %rsp maintained? First consider two core instructions:
- push op1
  - Pushes value of op1 on to the top of the stack
  - Essentially equivalent to:
    - subq 8, %rsp
    - movq op1, (%rsp)
  - The subq “allocates a quad-word of space” on the stack. The mov stores op1 on the new top. As a result, (%rsp) = op1 after this operation.
  - Note: increasing stack size means decreasing %rsp value

x86-64 Stack Convention

- pop op1
  - Pops the top of the stack and stores it in op1.
  - Essentially equivalent to:
    - movq (%rsp), op1
    - addq 8, %rsp
  - The mov takes the value at the top and stores it in op1. The add increases the value of %rsp and decreases the stack size.
- But wait, there's more (ret, call, leave). Let's explain those via an example.

x86-64 Stack Convention

- Disclaimer: The following is primarily applicable for x86-64. x86 is a bit different when it comes to this stuff.
- Disclaimer 2: The illustrations here have the stack drawn with the high addresses at the top and low addresses at the bottom. This follows the book, but not the professor's illustrations. KEEP THIS IN MIND. The professor draws the memory with the high address at the bottom.
- Let's say we have 4 functions, 'main', 'foo', 'bar', and 'useless'.
- main calls foo, foo calls bar, bar calls useless.
- main → foo → bar → useless
- Disclaimer 3: I suck at names.

x86-64 Stack Convention

```
long foo()
{
  long a = 0xfeed;
  long b = 0xface;
  long c = bar(a, b) + 1;
  return c;
}

int main()
{
  foo();
}
```

```
void useless()
{
  int a = 0;
}

long bar(long a, long b)
{
  unsigned long ret =
((unsigned long) (a << 16)) |
((unsigned long) b);
  useless();
  return ret;
}
```

x86-64 Stack Convention

Dump of assembler code for function foo (gcc invoked with no arguments):

```
   0x000000000040050c <+0>:     push   %rbp
   0x000000000040050d <+1>:     mov    %rsp,%rbp
   0x0000000000400510 <+4>:     sub    $0x20,%rsp
   0x0000000000400514 <+8>:     movq   $0xfeed,-0x8(%rbp)
   0x000000000040051c <+16>:    movq   $0xface,-0x10(%rbp)
   0x0000000000400524 <+24>:    mov    -0x10(%rbp),%rdx
   0x0000000000400528 <+28>:    mov    -0x8(%rbp),%rax
   0x000000000040052c <+32>:    mov    %rdx,%rsi
   0x000000000040052f <+35>:    mov    %rax,%rdi
   0x0000000000400532 <+38>:    callq  0x4004d6 <bar>
   0x0000000000400537 <+43>:    add    $0x1,%rax
   0x000000000040053b <+47>:    mov    %rax,-0x18(%rbp)
   0x000000000040053f <+51>:    mov    -0x18(%rbp),%rax
   0x0000000000400543 <+55>:    leaveq
   0x0000000000400544 <+56>:    retq
```

x86-64 Stack Convention

Let's just focus on the black text at the moment:

```
0x000000000040050c <+0>:     push   %rbp
0x000000000040050d <+1>:     mov    %rsp,%rbp
0x0000000000400510 <+4>:     sub    $0x20,%rsp
0x0000000000400514 <+8>:     movq   $0xfeed,-0x8(%rbp)
0x000000000040051c <+16>:    movq   $0xface,-0x10(%rbp)
0x0000000000400524 <+24>:    mov    -0x10(%rbp),%rdx
0x0000000000400528 <+28>:    mov    -0x8(%rbp),%rax
0x000000000040052c <+32>:    mov    %rdx,%rsi
0x000000000040052f <+35>:    mov    %rax,%rdi
0x0000000000400532 <+38>:    callq  0x4004d6 <bar>
0x0000000000400537 <+43>:    add    $0x1,%rax
0x000000000040053b <+47>:    mov    %rax,-0x18(%rbp)
0x000000000040053f <+51>:    mov    -0x18(%rbp),%rax
0x0000000000400543 <+55>:    leaveq
0x0000000000400544 <+56>:    retq
```

x86-64 Stack Convention

```
0x0000000000400514 <+8>:     movq   $0xfeed,-0x8(%rbp)
0x000000000040051c <+16>:    movq   $0xface,-0x10(%rbp)
```

These two lines are establishing long a and long b's location in memory (on the stack). Makes sense. But what's this nonsense?

```
0x0000000000400524 <+24>:    mov    -0x10(%rbp),%rdx
0x0000000000400528 <+28>:    mov    -0x8(%rbp),%rax
0x000000000040052c <+32>:    mov    %rdx,%rsi
0x000000000040052f <+35>:    mov    %rax,%rdi
```

We're pulling a and b from memory into registers %rdx and %rax. Then immediately copying them into %rsi and %rdi?

x86-64 Stack Convention

- %rdi and %rsi are used, by convention, to pass arguments into functions, in particular, the first and second arguments respectively.
- Argument: register
  1: %rdi
  2: %rsi
  3: %rdx
  4: %rcx
  5: %r8 (that's not confusing at all)
  6: %r9
  7 and onwards: stored on the stack. Example later.

x86-64 Stack Convention

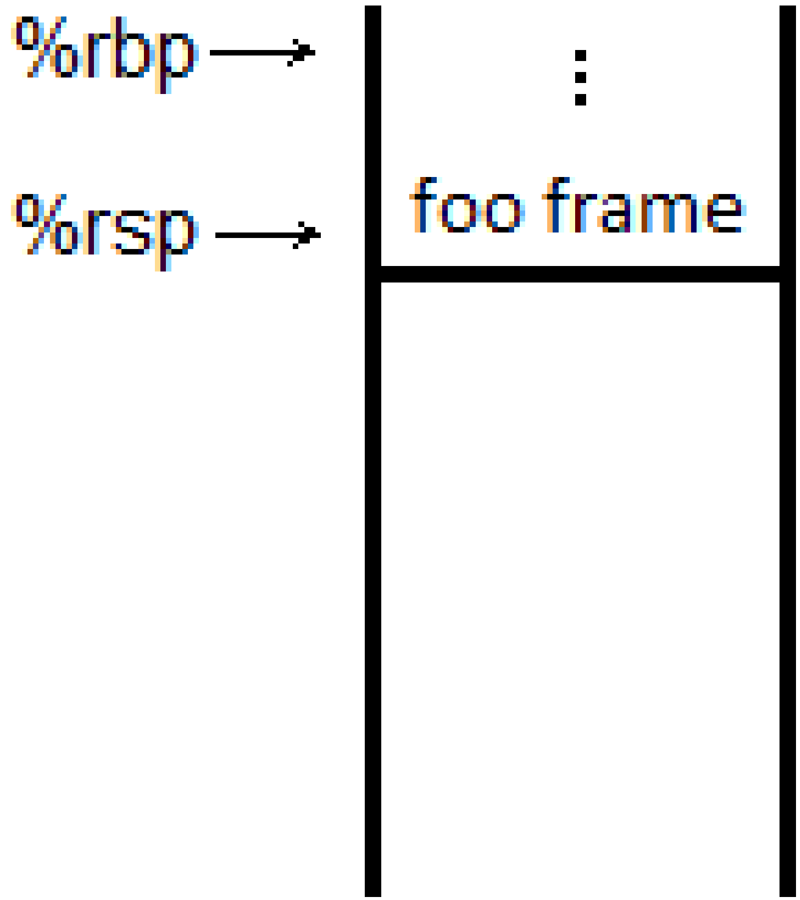
Now we're at the call, arguments a and b are in the appropriate registers. We're ready to call bar.

```
...
0x0000000000400514 <+8>:     movq   $0xfeed,-0x8(%rbp)
0x000000000040051c <+16>:    movq   $0xfacd,-0x10(%rbp)
0x0000000000400524 <+24>:    mov    -0x10(%rbp),%rdx
0x0000000000400528 <+28>:    mov    -0x8(%rbp),%rax
0x000000000040052c <+32>:    mov    %rdx,%rsi
0x000000000040052f <+35>:    mov    %rax,%rdi
0x0000000000400532 <+38>:    callq  0x4004d6 <bar>
0x0000000000400537 <+43>:    add    $0x1,%rax
...
```

What does call do?

x86-64 Stack Convention

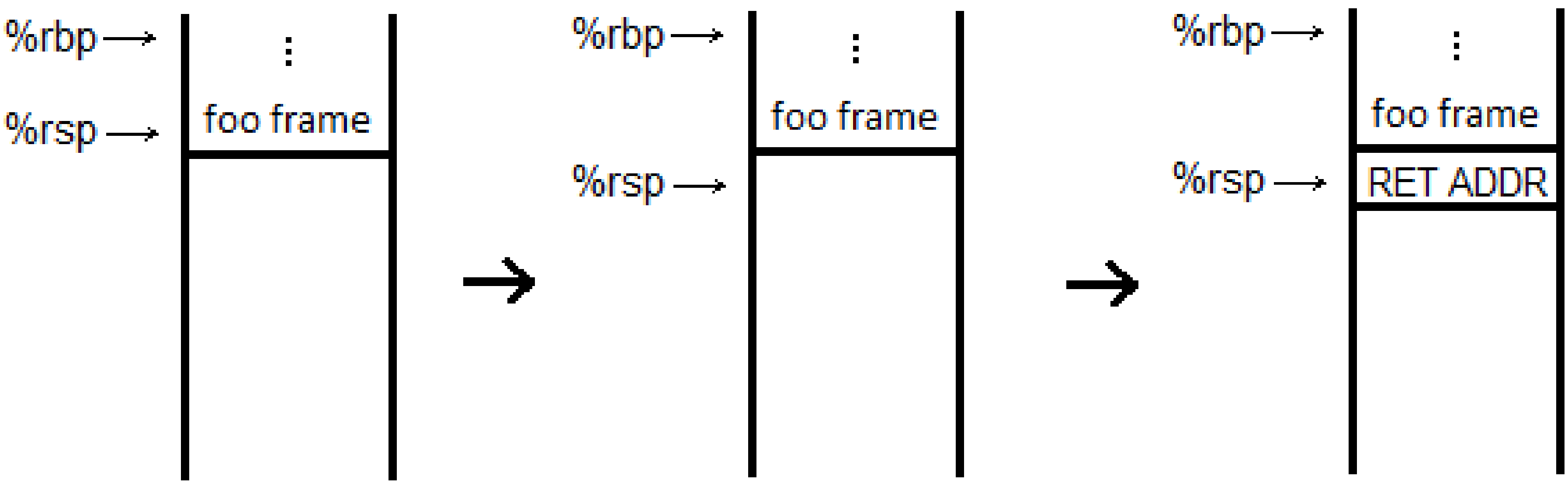
- First, this is approximately what the stack looks like at this point.
- The %rsp is a pointer is at the top of the stack (lowest address).
- The %rbp is a pointer to the "bottom" of the foo stack frame (high address).



x86-64 Stack Convention

- call LABEL
- “Calls” the function that begins at LABEL
- Equivalent to:
  - %rsp = %rsp – 8
  - (%rsp) = 5 + %rip
    - %rip contains the current instruction being executed (call LABEL). The call function takes 5 bytes to represent in memory. %rip + 5 is equal to the next instruction, which is the **return address**. Note: this is the return *address*, not the return value.
  - %rip = LABEL
    - Set the new instruction pointer to the beginning of the new function.

x86-64 Stack Convention



- The stack as “call” is executed. First, %rsp is decremented by 8, allocating space on the stack. Then (%rsp) is set to the return address.
- The return address is 0x400537 and %rip is set to the address of bar. Henceforth, we are executing the bar function.

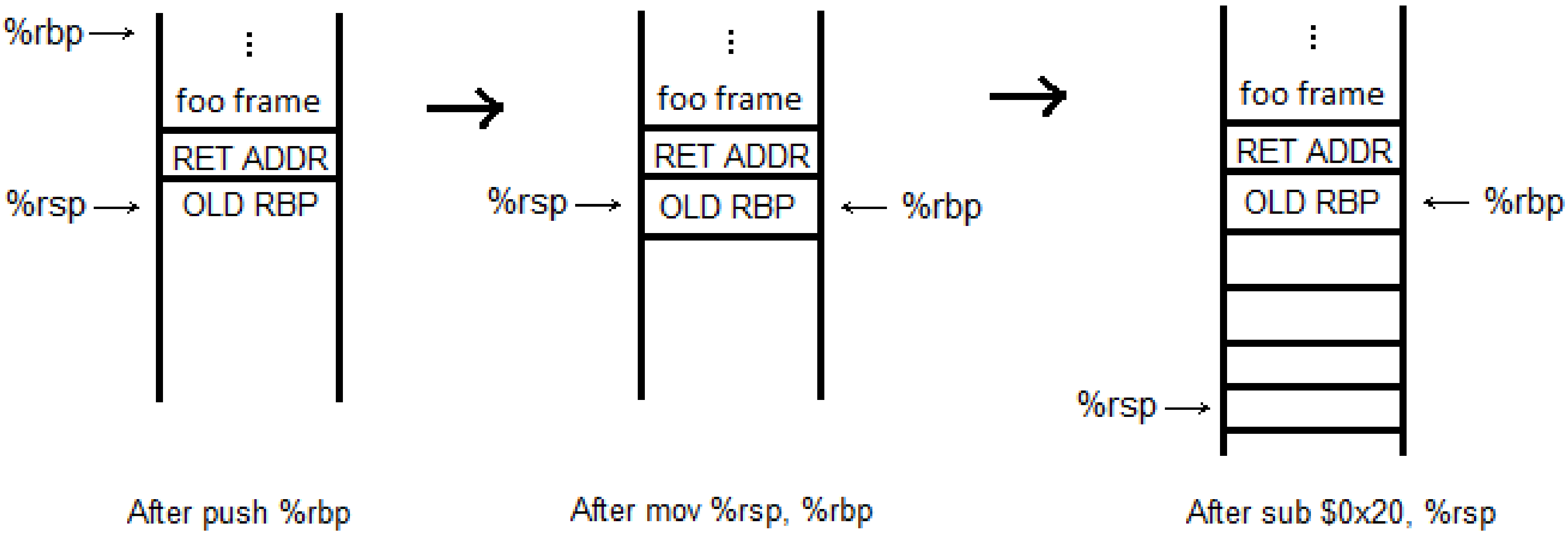
x86-64 Stack Convention

```
Dump of assembler code for function bar:
   0x00000000004004d6 <+0>:     push   %rbp
   0x00000000004004d7 <+1>:     mov    %rsp,%rbp
   0x00000000004004da <+4>:     sub    $0x20,%rsp
   0x00000000004004de <+8>:     mov    %rdi,-0x18(%rbp)
   0x00000000004004e2 <+12>:    mov    %rsi,-0x20(%rbp)
   0x00000000004004e6 <+16>:    mov    -0x18(%rbp),%rax
   0x00000000004004ea <+20>:    shl    $0x10,%rax
   0x00000000004004ee <+24>:    mov    %rax,%rdx
   0x00000000004004f1 <+27>:    mov    -0x20(%rbp),%rax
   0x00000000004004f5 <+31>:    or     %rdx,%rax
   0x00000000004004f8 <+34>:    mov    %rax,-0x8(%rbp)
   0x00000000004004fc <+38>:    mov    $0x0,%eax
   0x0000000000400501 <+43>:    callq  0x4004c8 <useless>
   0x0000000000400506 <+48>:    mov    -0x8(%rbp),%rax
   0x000000000040050a <+52>:    leaveq
   0x000000000040050b <+53>:    retq
```

x86-64 Stack Convention

- The new frame allocates stack space in the beginning of the function.
- push   %rbp
  - Save previous stack frame on top of stack.
- mov   %rsp,%rbp
  - Set %rbp to the top of the stack.
- sub $0x20,%rsp
  - Allocate space on the stack. The “top” of the stack is now lower while %rbp remains at the base of the frame.

x86-64 Stack Convention



- The next three instructions. The stack frame is now prepared.
- Note: The OLD RBP is the %rbp as of the leftmost image. It is the %rbp of foo's frame. Now we're in bar and we have a different frame and thus a different %rbp.

x86-64 Stack Convention

- Important conventions:
  - (%rbp) contains the %rbp of the previous stack frame (so that you can restore the stack frame of the previous function)
  - 8(%rbp) contains the return address of the previous function.
  - %rax contains the return value when the function completes.
  - What next? Let's look at the instructions that modify the stack

x86-64 Stack Convention

```
Dump of assembler code for function bar:
   0x00000000004004d6 <+0>:     push   %rbp
   0x00000000004004d7 <+1>:     mov    %rsp,%rbp
   0x00000000004004da <+4>:     sub    $0x20,%rsp
   0x00000000004004de <+8>:     mov    %rdi,-0x18(%rbp)
   0x00000000004004e2 <+12>:    mov    %rsi,-0x20(%rbp)
   0x00000000004004e6 <+16>:    mov    -0x18(%rbp),%rax
   0x00000000004004ea <+20>:    shl    $0x10,%rax
   0x00000000004004ee <+24>:    mov    %rax,%rdx
   0x00000000004004f1 <+27>:    mov    -0x20(%rbp),%rax
   0x00000000004004f5 <+31>:    or     %rdx,%rax
   0x00000000004004f8 <+34>:    mov    %rax,-0x8(%rbp)
   0x00000000004004fc <+38>:    mov    $0x0,%eax
   0x0000000000400501 <+43>:    callq  0x4004c8 <useless>
   0x0000000000400506 <+48>:    mov    -0x8(%rbp),%rax
   0x000000000040050a <+52>:    leaveq
   0x000000000040050b <+53>:    retq
```

x86-64 Stack Convention

```
0x00000000004004de <+8>:     mov    %rdi,-0x18(%rbp)
0x00000000004004e2 <+12>:    mov    %rsi,-0x20(%rbp)
```

From before, %rdi contains argument 1 (0xFEED) and %rsi contains argument 2 (0xFACE).

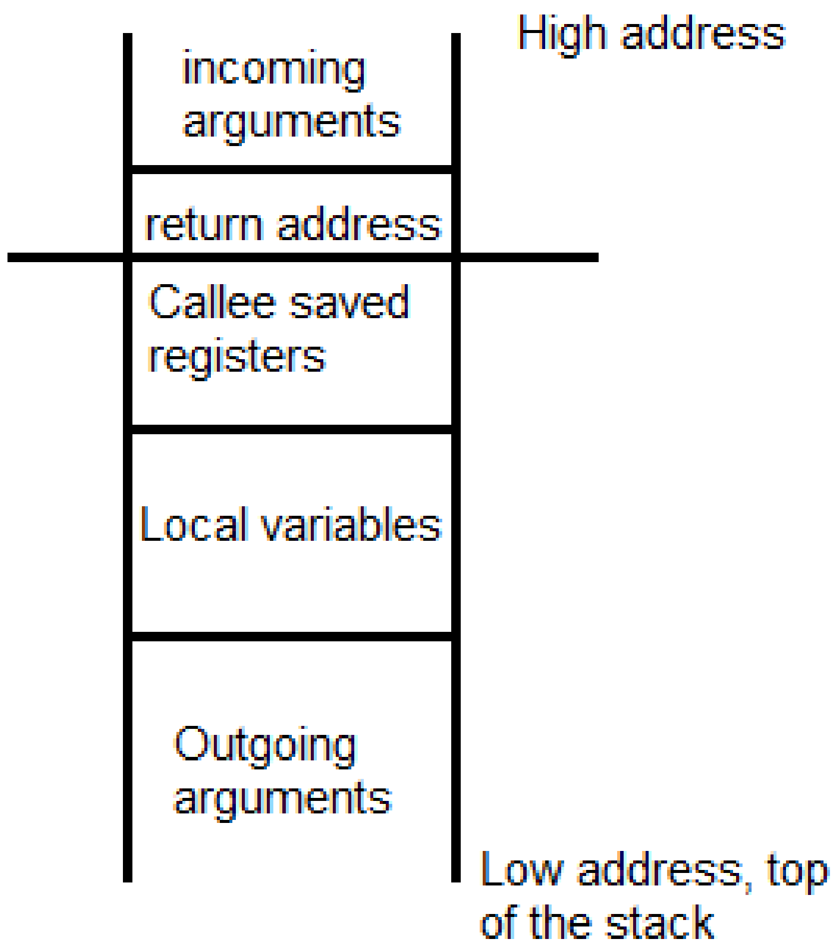
```
0x00000000004004e6 <+16>:    mov    -0x18(%rbp),%rax
0x00000000004004ea <+20>:    shl    $0x10,%rax
0x00000000004004ee <+24>:    mov    %rax,%rdx
0x00000000004004f1 <+27>:    mov    -0x20(%rbp),%rax
0x00000000004004f5 <+31>:    or     %rdx,%rax
```

The above section computes the main operation of the function, producing the value 0xFEEDFACE, which is placed in %rax. Then, %rax is placed on the stack. This effectively completes the stack frame fro this function call.

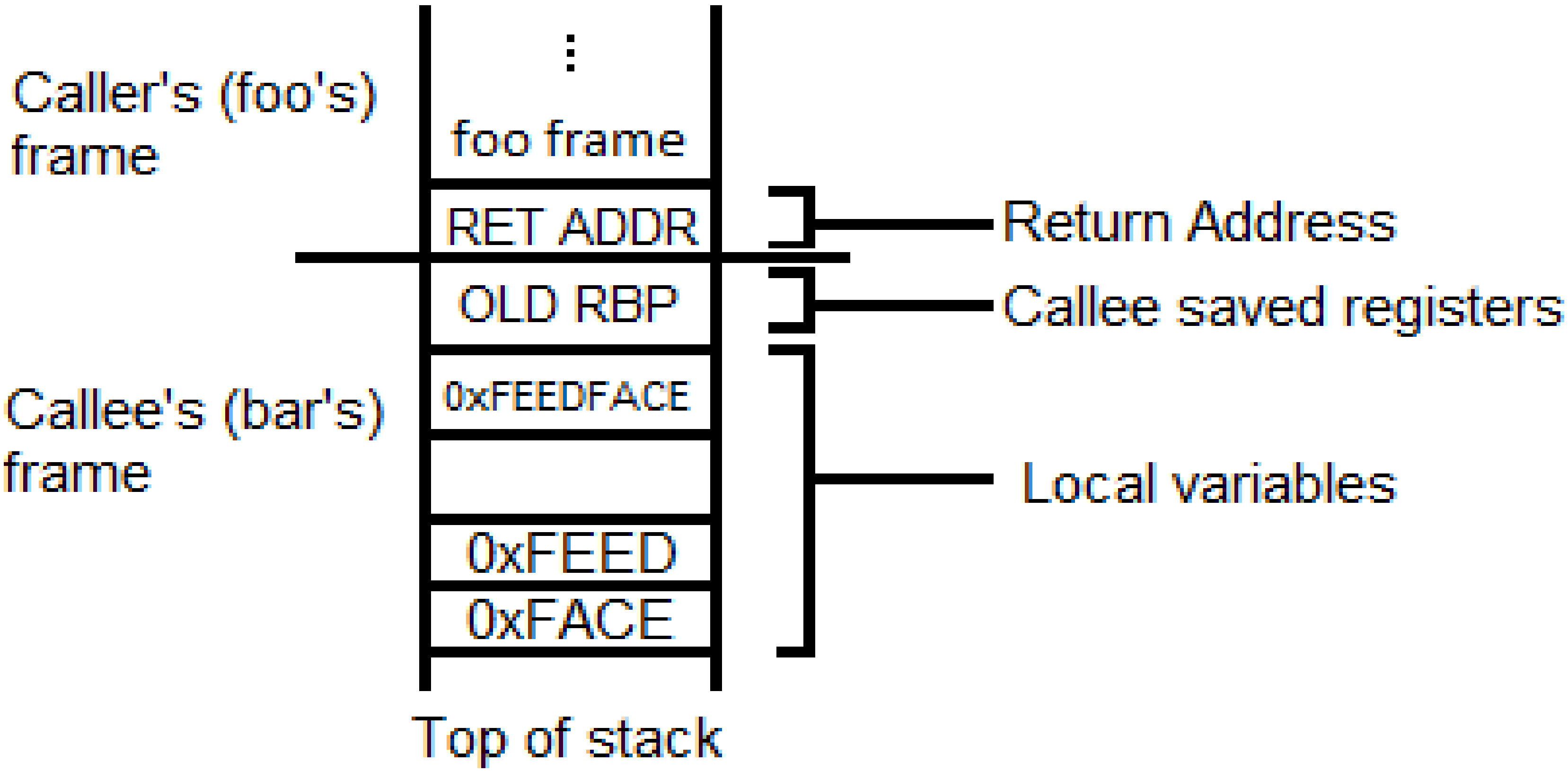
```
0x00000000004004f8 <+34>:    mov    %rax,-0x8(%rbp)
```

x86-64 Stack Convention

- Now that the stack frame in this example is complete, how does this compare against the general stack frame presented in class and in the book?
- To the right is a sloppy recreation of the stack diagram in the book (pg. 240, 3rd ed.)



x86-64 Stack Convention



- Looks pretty good.
- Note: We don't have any incoming arguments because we had only 2 arguments into bar. We don't have any outgoing arguments since bar doesn't call a function that takes arguments.
- How do we finish the function and restore foo's frame?

x86-64 Stack Convention

```
Dump of assembler code for function bar:
...
   0x0000000000400506 <+48>:    mov    -0x8(%rbp),%rax
   0x000000000040050a <+52>:    leaveq
   0x000000000040050b <+53>:    retq
```

First, the return value (0xFEEDFACE) is placed into %rax. Then leave and ret are called.

x86-64 Stack Convention

- leave
- “Calls” the function that begins at LABEL
- Equivalent to:
  - mov %rbp, %rsp
    - Point %esp to top of previous stack frame
  - mov (%rbp), %rbp
    - Point %ebp to previous stack frame base.
  - add $8, %rsp
    - Increment the %esp to have it point to the return address.

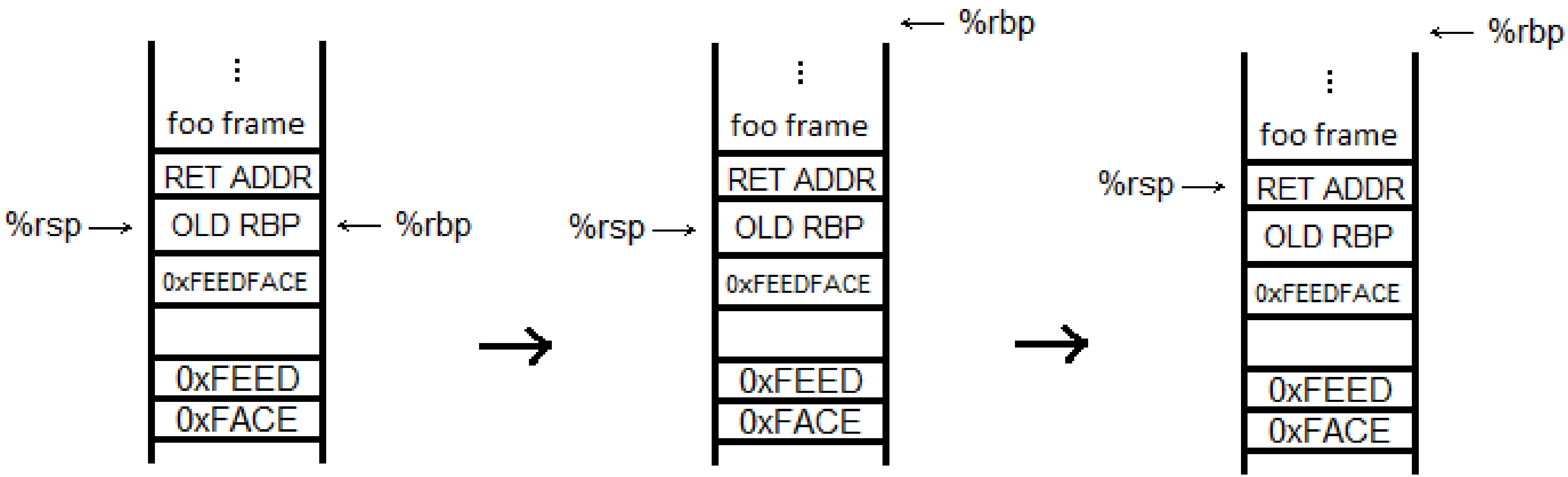
x86-64 Stack Convention

- 'leave' essentially prepares the stack so that the current function can return.
- It essentially does the reverse of the first three instructions.
- If the first three instructions allocate the necessary space for the new stack frame, 'leave' deallocates that space and sets %rsp and %rbp back to the previous frame.
- 'leave' is usually followed by a 'ret'

x86-64 Stack Convention

- Note: The purpose of leave is to deallocate the the current stack frame, leave the stack pointer pointing at the return address, and leave the frame pointer at the previous frame's base.
- This can be done with:
  - add __, %esp
  - pop %ebp
- Sometimes a function will have these instructions instead of a 'leave'.

x86-64 Stack Convention



- The intermediate steps and the final result of leave.
  - %rsp is moved back to the same location as %rbp
  - %rbp is set to (%rbp), which is the previous frame's %rbp.
  - %rsp is iterated by 8, so that %rsp points to the return address

x86-64 Stack Convention

- ret
- Reverts the instruction pointer back to the calling function.
- Equivalent to:
  - mov (%rsp), %rip
  - add $8, %rsp
- When the function returns, the return value is in %rax.
- “ret” is ready to be called when %rsp is pointing to the return address stored on the stack.

x86-64 Stack Convention

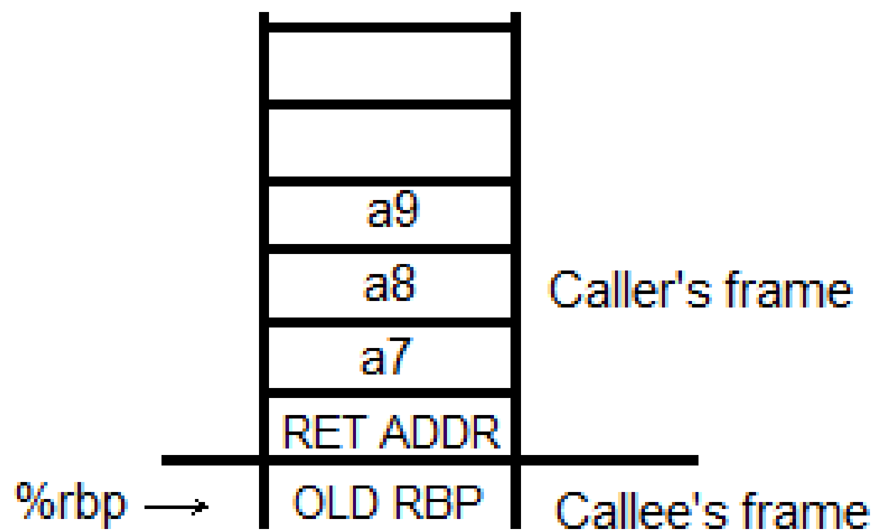
- When we allocated and deallocated the stack frames, what did we really do?
- We simply moved the %rbp and %rsp pointers.
- We didn't clear out or modify the memory as part of the allocation/deallocation process.

x86-64 Stack Convention

- What would the stack look like if we passed more than 6 arguments into the function?
- Ex:
- int func(long a1, long a2, long a3, long a4, long a5, long a6, long a7, long a8, long a9)

x86-64 Stack Convention

- int func (long a1, long a2, long a3,
  long a4, long a5, long a6,
  long a7, long a8, long a9)
- The first six arguments will be stored in registers.
- The following arguments will be placed the the stack by the caller.
- The earlier the argument in the list, the closer to %rbp.
- In this example, a7 is accessible to the callee via 0x10(%rbp), a8 is accessible via 0x18(%rbp), etc.



x86-64 Stack (not following) Convention

- The general representation of the stack includes many different segments (local variables, callee saved registers, etc.)
- However, as demonstrated in the previous example, if a section is not necessary, a compiler will optimize it out.
- For example, consider the foo function. What does foo look like when compiled with -O1?
- Please. What can a measly level 1 optimization do?

x86-64 Stack (not following) Convention

```
Dump of assembler code for function foo:
   0x00000000004004d5 <+0>:     mov    $0xfeedfacf,%eax
   0x00000000004004da <+5>:     retq
```

x86-64 Stack (not following) Convention

- Because the compiler could tell exactly what the string of functions was going to do (bar returns 0xFEEDFACE and foo returns 0xFEEDFACE + 1), it decided it didn't need to bother with all of the stack stuff. This means:
  - No saving %rbp
  - No allocating space on the stack
  - No push/leave
- The only thing it maintained was the behavior that %rax contains the return value and that (%rsp) contains the return address.
- This is the minimum behavior necessary for creating a function that is compatible with other functions following the convention.

x86-64: Caller/Callee saved registers

- In procedure calls, the function that calls another function is the “caller”.
- The function that *is* called is the “callee”.
- The general idea:
  - Consider the case where function “a” uses the register %rax for local calculations.
  - function “a” calls function “b”
  - function “b” uses %rax for local calculations
  - There is only one register %rax.

x86-64: Caller/Callee saved registers

- When the callee returns and execution goes back to the caller, the caller expects the %rax value it was working with.
- However, without any protection, the callee will have changed %rax, losing the caller's data.
- The solution is to back-up the registers on the stack as necessary.
  - An example of this is “push %rbp”, even though that is a special case.

x86-64: Caller/Callee saved registers

- In x86-64:
- Caller saved: %rax, %rcx, %rdx, %rsi, %rdi, %r8, %r9, %r10, %r11
  - Say P calls Q.
  - Q should be allowed to use these registers freely.
  - This means, P must save these register values onto the stack (or other register) before calling Q.

x86-64: Caller/Callee saved registers

- In x86-64:
- Callee saved: %rbx, %rbp, %r12-%r15
  - Say P calls Q.
  - P should be allowed to call a function without backing up these registers.
  - This means, Q must save these register values onto the stack before using them.
  - Q must also restore the values back into the registers before exiting.

x86-64: Optimizing away "call"

- Apparently it's a thing.
- The compiler recognizes optimizations in which using the "call" instruction is unnecessary, even in cases where we need to call a function.
- How does it work?
- Two ways:
  - Tail recursion optimization
  - Turning a call into a loop (consult the preorder example for more)

x86-64: Tail recursion optimization

- Consider the postorder function from class:

```
struct tree {
  struct tree *left
  struct tree *right;
  int val;
};

void postorder(struct tree *t, void (*visit)(int))
{
  if(t)
  {
    postorder(t->left, visit);
    postorder(t->right, visit);
    visit(t->val);
  }
}
```

x86-64: Tail recursion optimization

```
Dump of assembler code for function postorder:
   0x00000000004004d0 <+0>:     test   %rdi,%rdi
   0x00000000004004d3 <+3>:     je     0x400508 <postorder+56>
   0x00000000004004d5 <+5>:     push   %rbp
   0x00000000004004d6 <+6>:     push   %rbx
   0x00000000004004d7 <+7>:     mov    %rdi,%rbx
   0x00000000004004da <+10>:    mov    %rsi,%rbp
   0x00000000004004dd <+13>:    sub    $0x8,%rsp
   0x00000000004004e1 <+17>:    mov    (%rdi),%rdi
   0x00000000004004e4 <+20>:    callq  0x4004d0 <postorder>
   0x00000000004004e9 <+25>:    mov    0x8(%rbx),%rdi
   0x00000000004004ed <+29>:    mov    %rbp,%rsi
   0x00000000004004f0 <+32>:    callq  0x4004d0 <postorder>
   0x00000000004004f5 <+37>:    mov    0x10(%rbx),%edi
   0x00000000004004f8 <+40>:    add    $0x8,%rsp
   0x00000000004004fc <+44>:    mov    %rbp,%rax
   0x00000000004004ff <+47>:    pop    %rbx
   0x0000000000400500 <+48>:    pop    %rbp
   0x0000000000400501 <+49>:    jmpq   *%rax
   0x0000000000400503 <+51>:    nopl   0x0(%rax,%rax,1)
   0x0000000000400508 <+56>:    repz retq
```

x86-64: Tail recursion optimization

- We have 3 function calls but only two “call” instructions.
- Additionally our trusty %rbp frame pointer isn't used as a frame pointer at all. It's used as a common callee saved register.
- Why does this work?
- First, where do we expect the call to occur?

x86-64: Tail recursion optimization

```
Dump of assembler code for function postorder:
   0x00000000004004d0 <+0>:     test   %rdi,%rdi
   0x00000000004004d3 <+3>:     je     0x400508 <postorder+56>
   0x00000000004004d5 <+5>:     push   %rbp
   0x00000000004004d6 <+6>:     push   %rbx
   0x00000000004004d7 <+7>:     mov    %rdi,%rbx
   0x00000000004004da <+10>:    mov    %rsi,%rbp
   0x00000000004004dd <+13>:    sub    $0x8,%rsp
   0x00000000004004e1 <+17>:    mov    (%rdi),%rdi
   0x00000000004004e4 <+20>:    callq  0x4004d0 <postorder>
   0x00000000004004e9 <+25>:    mov    0x8(%rbx),%rdi
   0x00000000004004ed <+29>:    mov    %rbp,%rsi
   0x00000000004004f0 <+32>:    callq  0x4004d0 <postorder>
   0x00000000004004f5 <+37>:    mov    0x10(%rbx),%edi
   0x00000000004004f8 <+40>:    add    $0x8,%rsp
   0x00000000004004fc <+44>:    mov    %rbp,%rax
   0x00000000004004ff <+47>:    pop    %rbx      ← Here, but instead
   0x0000000000400500 <+48>:    pop    %rbp      there's popping and
   0x0000000000400501 <+49>:    jmpq   *%rax     a jumping
   0x0000000000400503 <+51>:    nopl   0x0(%rax,%rax,1)
   0x0000000000400508 <+56>:    repz retq
```

x86-64: Tail recursion optimization

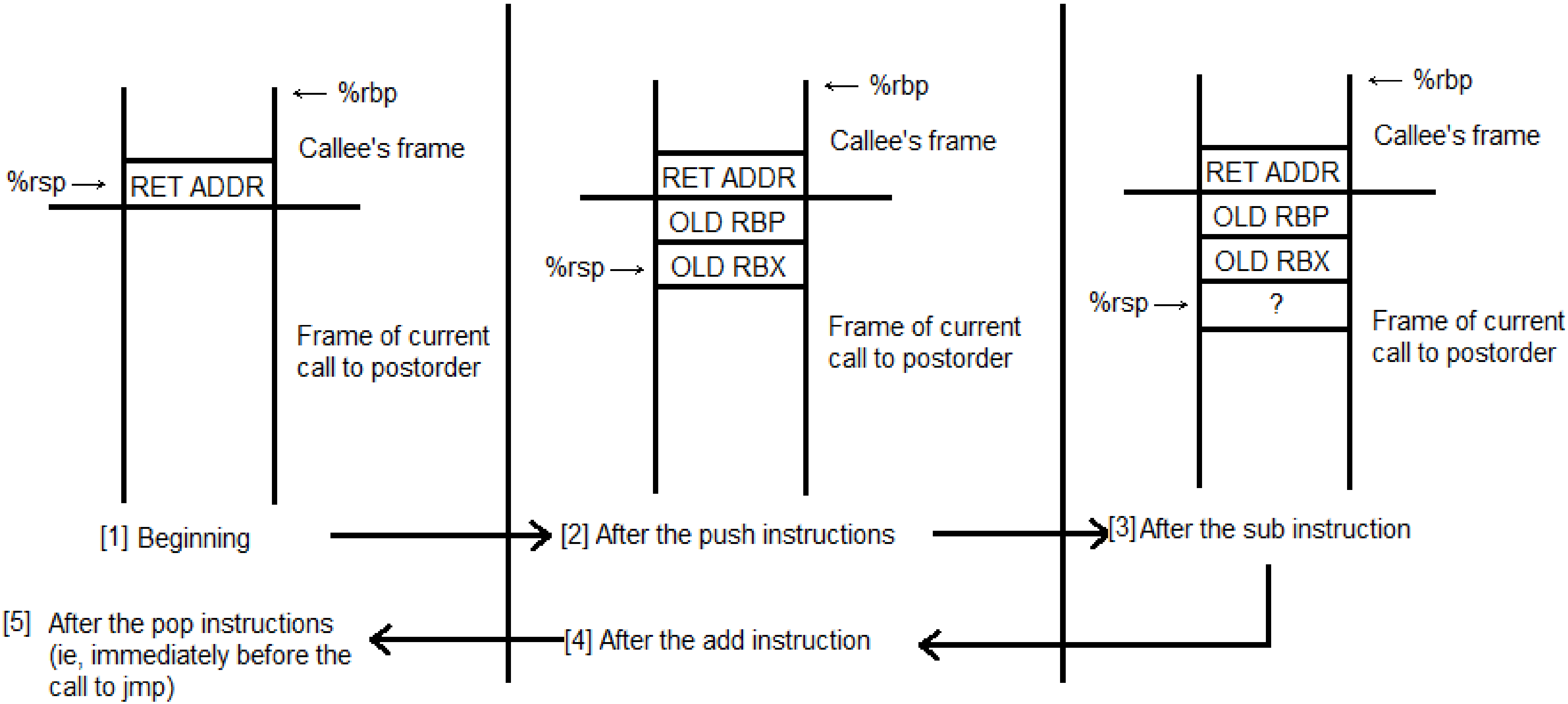
- Instead of using 'call' on the 'visit' function (which is located in %rax), we jump to it. We haven't pushed a return value!
- Since all of the calls to postorder fall within our expectations of how procedure calls work, let's ignore them.
- In fact, let's only consider the operations that modify the stack and are needed to execute the call to visit.

x86-64: Tail recursion optimization

```
Dump of assembler code for function postorder:
...
   0x00000000004004d5 <+5>:     push   %rbp
   0x00000000004004d6 <+6>:     push   %rbx
   0x00000000004004d7 <+7>:     mov    %rdi,%rbx
   0x00000000004004da <+10>:    mov    %rsi,%rbp
   0x00000000004004dd <+13>:    sub    $0x8,%rsp
...
   0x00000000004004f5 <+37>:    mov    0x10(%rbx),%edi
   0x00000000004004f8 <+40>:    add    $0x8,%rsp
   0x00000000004004fc <+44>:    mov    %rbp,%rax
   0x00000000004004ff <+47>:    pop    %rbx
   0x0000000000400500 <+48>:    pop    %rbp
   0x0000000000400501 <+49>:    jmpq   *%rax
   0x0000000000400503 <+51>:    nopl   0x0(%rax,%rax,1)
   0x0000000000400508 <+56>:    repz retq
```

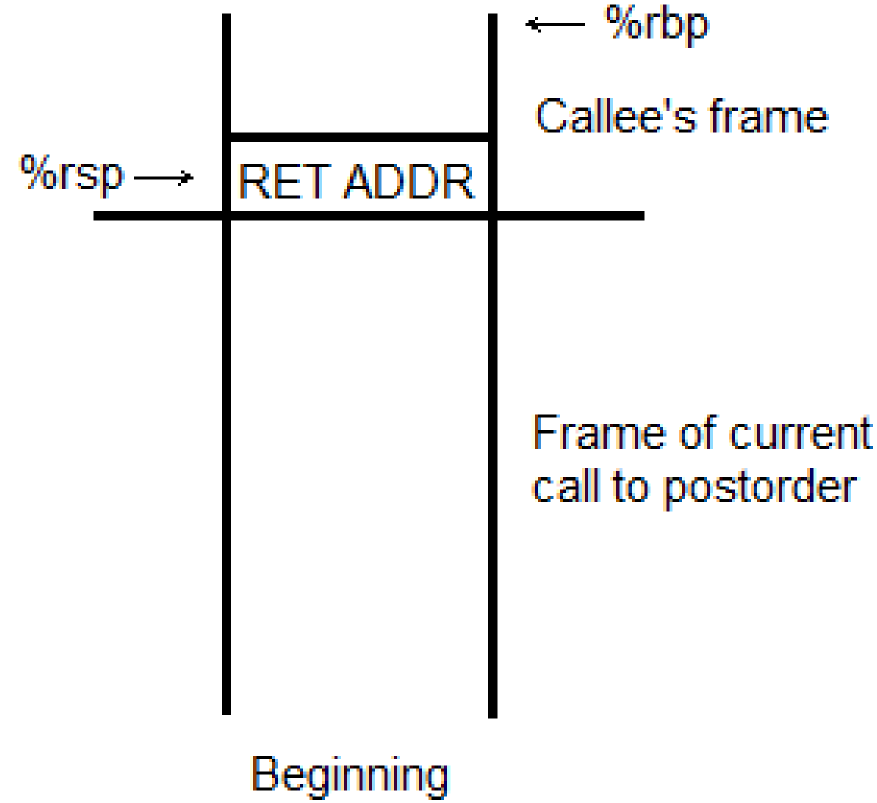
- Operations in red: instructions that modify the stack.
- What does the stack look like throughout the course of the operations.

x86-64: Tail recursion optimization



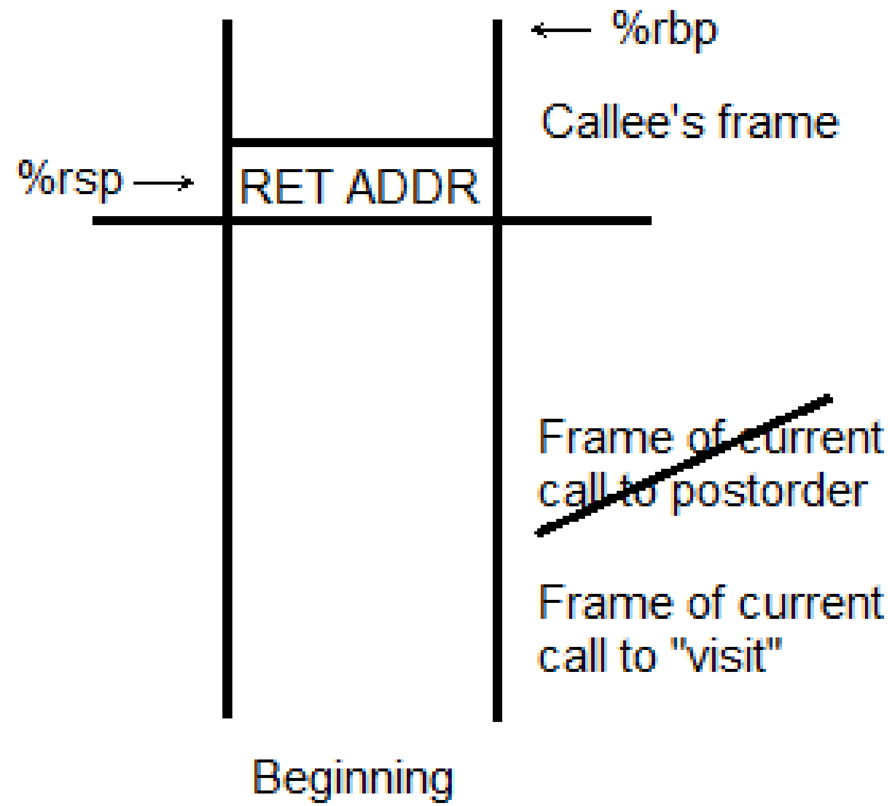
x86-64: Tail recursion optimization

- The final state looks like the beginning state where the stack has been prepared for the new postorder stack frame.
- Note that before this, “t->val” was placed into %rdi (mov 0x10(%rbx),%edi). This means the argument to the visit function is prepared.
- As a result, when jmp *%rax is called, you jmp to the “visit” function and this view of the stack...



x86-64: Tail recursion optimization

- ...becomes this:
- Via tail recursion optimization, the visit function essentially takes over the stack frame of the calling function.
- Now, the call to visit shares the same %rsp the previous call to postorder had.
- But that means, when “visit” returns, it will jump to the return value that was meant for postorder. Is that right?



x86-64: Tail recursion optimization

- Recall:

```
void postorder(struct tree *t, void (*visit)(int))
{
  if(t)
  {
    postorder(t->left, visit);
    postorder(t->right, visit);
    visit(t->val);
  }
}
```

- The condition that permits for tail recursion optimization is: “the very last thing that a function does is call another function”.
- When visit returns, the only thing that would happen next is that postorder would return. Therefore visit can just return to postorder's return without messing anything up.

Alignment

- In order to simplify design, primitive data types should be placed in memory according to certain restrictions based on the side of the data type.
- 'X aligned' means that the starting address of a particular data type must be a multiple of X.
- In x86-64:
  - char (1 byte) : 1 aligned
  - short (2 bytes) : 2 aligned
  - int/float (4 bytes) : 4 aligned
  - long/double/void * (8 bytes) : 8 aligned

Alignment

- Although x86-64 will work if the alignment rules are violated, it is heavily advised that they are followed.
- Chances are, any code you see generated by any compiler will follow these rules unless you explicitly compile it with the flags telling it not to (those probably exist right?).
- Data types are not the only things that should follow the alignment.

Alignment

- The stack has an alignment rule as well in x86-64. In particular, the rule is the following:
  - The stack (ie %rsp) should be 16 byte aligned when "call" is executed.
  - Because that call will push the return address and %rbp is likely to be pushed on the stack in the called function, this will likely maintain alignment.
- However, in cases where either %rbp isn't pushed or when other registers are pushed, the resulting code can be a little wonky. Consider postorder...

Alignment

```
Dump of assembler code for function postorder:
...
   0x00000000004004d5 <+5>:     push   %rbp
   0x00000000004004d6 <+6>:     push   %rbx
   0x00000000004004d7 <+7>:     mov    %rdi,%rbx
   0x00000000004004da <+10>:    mov    %rsi,%rbp
   0x00000000004004dd <+13>:    sub    $0x8,%rsp
...
   0x00000000004004f5 <+37>:    mov    0x10(%rbx),%edi
   0x00000000004004f8 <+40>:    add    $0x8,%rsp
   0x00000000004004fc <+44>:    mov    %rbp,%rax
   0x00000000004004ff <+47>:    pop    %rbx
   0x0000000000400500 <+48>:    pop    %rbp
   0x0000000000400501 <+49>:    jmpq   *%rax
   0x0000000000400503 <+51>:    nopl   0x0(%rax,%rax,1)
   0x0000000000400508 <+56>:    repz retq
```

- `What was that point of the that nonsense?`

Alignment

- Assume the stack was aligned before the call to postorder.
  - The call to post order pushes an 8-byte quantity on the stack. %rsp is no longer 16-byte aligned.
  - %rbp is pushed. %rsp is aligned.
  - %rbx is pushed. %rsp is no longer aligned.
- As a result, the %rsp was manually decremented to maintain the alignment rule.

structs review

- struct: heterogeneous data structure or a collection of different (or the same) data types.
- Although similar to C++ classes in that they package different data types into one, the following struct rules do not apply to classes.

structs review

```
struct s {
  char c1;
  int i;
  char c2;
  int j;
};
```

- The C standard demands that the elements within a struct appear in memory the same order that they appear in the declaration. No reordering compiler optimization!
- Each element within a struct must follow the normal alignment rules of that data type.

structs review

```
struct s {
  char c1;
  int i;
  char c2;
  int j;
};
```

- What's the problem with this struct?

structs review

```
struct s {
  char c1;
  int i;
  char c2;
  int j;
};
```

- Say an instance of the struct begins at 0x10. Then c1 is at address 0x10. However, 'i' cannot be at address 0x11 (it needs to be 4-aligned). As a result, we need 3 bytes of padding.
- 'i' will be positioned at 0x14. c2 will be positioned at 0x15. j will be positioned at 0x18

Structs review

- This is a waste of space! There will be 3 bytes of padding after c1 and 3 bytes of padding after c2, meaning that this struct will take up 16 bytes when really it only needs 10.

structs review

- Two common struct ordering guidelines (which could be at odds):
    1. Place the most commonly used data type first.
    2. Place the elements in descending order of size (ie largest first)
- Why?

structs review

- 1.
- Memory references are expensive (ex. (%eax))... but memory references with an offset are more expensive (ex. 8(%eax))
- Chances are, you'll be referring to the struct by a pointer to the beginning of the struct, which means that dereferencing the pointer without an offset will point to the first element.

structs review

- 2.
- If the elements with larger sizes are first, that means there will be less of a need for padding.
- For example, consider struct s, except with the first two elements swapped:

```
struct s {
  int i;
  char c1;
  char c2;
  int j;
};
```

structs review

- 2.
- ```
  struct s {
    int i;
    char c1;
    char c2;
    int j;
  };
  ```
- Now, we need 2 bytes of padding between c2 and j for a total of 12 bytes.
  ```

structs review

- Because each internal element must follow their own alignment rules, the alignment of the struct must be equal to the strictest of the elements within a struct.
- But wait...

structs review

- Consider:

```
struct s {
  char c;
  int i;
};
```

- Because int i is aligned by 4, instances of struct s must be aligned by 4.
- There must also be 3 bytes of padding between c and I, meaning a total size of 8.

structs review

- Thus, a possible placement of (struct s s1) where s1.c = 0xFF and s1.i = 0x33221100 is the following:

Address:	**0x10**	**0x11**	**0x12**	**0x13**	**0x14**	**0x15**	**0x16**	**0x17**
Value:	**0XFF**	**0xXX**	**0xXX**	**0xXX**	**0x00**	**0x11**	**0x22**	**0x33**

- Where s begins at 0x10.
- This is how we meet the alignment requirements of each individual item

structs review

- Now consider:

```
struct s {
  int i;
  char c;
};
```

- i has an offset of 0, c has an offset of 4.
- As before, if we had an array of these, because i must be aligned properly, there would be 3 bytes of padding between elements.
- Thus, sizeof(s) is still 8

structs review

- But wait... what if you had the following code:

```
struct T
{
  struct s foo;
  char c;
} t;
```

- If t began at 0x10:
  - t.foo.i : 0x10 → 0x13
  - t.foo.c : 0x14
- But wait, t.c doesn't have to start at 0x18. It can be at 0x15 and all of the rules will be followed, right?

structs review

- But does it? If you try this:

```
int main()
{
  struct T test;
  printf("%p\n", &test.foo.i);
  printf("%p\n", &test.foo.c);
  printf("%p\n", &test.c);
}
```

- Output:
- 0x7ffdca140b40
- 0x7ffdca140b44
- 0x7ffdca140b48

- Nope, looks like sizeof(s) is really 8 bytes.

unions review

- Like structs except all of the values begin at the same address.
- union s {
-   short s;
-   char c;
- };
- This means that in a union that contains several values, only one of them is likely to be meaningful and assigning one term a value will trample other terms.

unions review

- union s {
- short s;
- char c;
- };
- union s foo;
- Say foo begins at 0x10.
- foo.s will be located in addresses 0x10 and 0x11
- foo.c will be located in address 0x10.

unions review

- union s {
-   short s;
-   char c;
- };
- union s foo;
- foo.s = 0xFFFF;
- foo.c = 0;
- printf(“%hx\n”, foo.s) => FF00

unions of structs review

- Unions are generally used in conjunction with operations that deal with types at runtime.
- For example, type casting:

```
union s {
  int i;
  int* p;
}
```

- C prohibits you from certain pointer arithmetic.
- union s foo;
- foo.p = (int *) malloc(4);
- foo.s = foo.s << 1;

unions of structs review

- A far more common use of structs is to do discriminated union.

```
union {
  struct s { char c , int p } s;
  struct t { char c , double p } t;
  struct v { char c , char* p } v;
} x;
```

- s, t, and v all begin at the same location and each struct begins with a c. This means that at any time, given union x foo;, foo.s.c == foo.t.c == foo.v.c.
- 'c' then becomes an identifier for the actual data.

End of

The Third Week

-Seven Weeks Remain-