x86 Review | Intro

(choose your own adventure)

Martin Kellogg

Course Announcements

- the PA3c1 (codegen testing) deadline has already passed
  - if you forgot about it, we will still accept submissions (with a penalty)

Course Announcements

- the PA3c1 (codegen testing) deadline has already passed
  - if you forgot about it, we will still accept submissions (with a penalty)
- PA3c2 (TAC) is due later this week (“before spring break”)
  - you should already have started, or you are behind
  - if there is demand from the class, I will consider a short extension on this assignment to e.g., Monday

Course Announcements

- the PA3c1 (codegen testing) deadline has already passed
  - if you forgot about it, we will still accept submissions (with a penalty)
- PA3c2 (TAC) is due later this week (“before spring break”)
  - you should already have started, or you are behind
  - if there is demand from the class, I will consider a short extension on this assignment to e.g., Monday
- I have become aware of a **bug in the reference compiler**’s x86-64 module; a fix will be forthcoming. For now, don’t trust it.

Course Announcements

- the PA3c1 (codegen testing) deadline has already passed
  - if you forgot about it, we will still accept submissions (with a penalty)
- PA3c2 (TAC) is due later this week (“before spring break”)
  - you should already have started, or you are behind
  - if there is demand from the class, I will consider a short extension on this assignment to e.g., Monday
- I have become aware of a **bug in the reference compiler**’s x86-64 module; a fix will be forthcoming. For now, don’t trust it.
- Don’t forget there is a **midterm** in this class the week after spring break!

Agenda

- Overview of x86-64 architecture

Agenda

- Overview of x86-64 architecture
  - this might be a review of some things you learned in 350

Agenda

- Overview of x86-64 architecture
  - this might be a review of some things you learned in 350
  - it might also be new
    - either is fine! (but we’re going to go quickly…)

Agenda

- Overview of x86-64 architecture
  - this might be a review of some things you learned in 350
  - it might also be new
    - either is fine! (but we're going to go quickly…)
  - my goal today: make sure you're aware of the "usual traps" in the x86-64 standard, and make sure you know where to go to learn more
    - I am not trying to give you a detailed understanding of each construct today (you'll get that from doing PA3…)

Resources

- When you’re doing PA3, you’re going to want to refer to some resources
  - there are many on the web
    - and I encourage you to explore

Resources

- When you’re doing PA3, you’re going to want to refer to some resources
  - there are many on the web
    - and I encourage you to explore
    - I don’t suggest trusting ChatGPT or similar tools on this - assembly programming requires getting all the details right, and LLMs are very bad at being detail-oriented

Resources

- When you're doing PA3, you're going to want to refer to some resources
  - there are many on the web
    - and I encourage you to explore
    - I don't suggest trusting ChatGPT or similar tools on this - assembly programming requires getting all the details right, and LLMs are very bad at being detail-oriented
  - I have curated a few resources here: https://kelloggm.github.io/martinjkellogg.com/teaching/cs485-sp25/languages/#x86-64

Resources

Suggestions welcome!

- When you're doing PA3, you're going to want to refer to some resources
  - there are many on the web
    - and I encourage you to explore
    - I don't suggest trusting ChatGPT or similar tools on this - assembly programming requires getting all the details right, and LLMs are very bad at being detail-oriented
  - I have curated a few resources here: https://kelloggm.github.io/martinjkellogg.com/teaching/cs485-sp25/languages/#x86-64

x86 Story Time

x86 Story Time

- x86 is a **very old** assembly language
  - 8086 processor for which it was originally designed was released in 1976…

x86 Story Time

- x86 is a **very old** assembly language
  - 8086 processor for which it was originally designed was released in 1976…
  - microarchitecture has **changed a lot** since then: pipelining, super-scalar, out-of-order, caching, multicore, …

x86 Story Time

- x86 is a **very old** assembly language
  - 8086 processor for which it was originally designed was released in 1976…
  - microarchitecture has **changed a lot** since then: pipelining, super-scalar, out-of-order, caching, multicore, …
- Modern x86 is **still backward-compatible** with 8086 code
  - You can get VisiCalc 1.0 on the web & run it!

x86 Story Time

- x86 is a **very old** assembly language
  - 8086 processor for which it was originally designed was released in 1976…
  - microarchitecture has **changed a lot** since then: pipelining, super-scalar, out-of-order, caching, multicore, …
- Modern x86 is **still backward-compatible** with 8086 code
  - You can get VisiCalc 1.0 on the web & run it!
- Intel's descriptions of the architecture are engulfed with modes and flags; the **modern processor is fairly straightforward**
  - Load/Store from memory
  - Register-register operations

x86: RISC or CISC?

x86: RISC or CISC?

- x86 is technically a ***CISC*** (***complex instruction set computer***) architecture

x86: RISC or CISC?

- x86 is technically a ***CISC*** (***complex instruction set computer***) architecture
  - key definitional feature of a CISC: there are instructions that take **more than one clock cycle** to execute

x86: RISC or CISC?

- x86 is technically a ***CISC*** (***complex instruction set computer***) architecture
  - key definitional feature of a CISC: there are instructions that take **more than one clock cycle** to execute
- However, the **parts of x86 that you should be using** in your compiler are actually closer to a traditional RISC architecture

x86: RISC or CISC?

- x86 is technically a ***CISC*** (***complex instruction set computer***) architecture
  - key definitional feature of a CISC: there are instructions that take **more than one clock cycle** to execute
- However, the **parts of x86 that you should be using** in your compiler are actually closer to a traditional RISC architecture
  - ***RISC*** = "***reduced instruction set computer***"

x86: RISC or CISC?

- x86 is technically a ***CISC*** (***complex instruction set computer***) architecture
  - key definitional feature of a CISC: there are instructions that take **more than one clock cycle** to execute
- However, the **parts of x86 that you should be using** in your compiler are actually closer to a traditional RISC architecture
  - ***RISC*** = "***reduced instruction set computer***"
  - most complex instructions exist for backward-compatibility and can be slow

x86: RISC or CISC?

- x86 is technically a ***CISC*** (***complex instruction set computer***) architecture
  - key definitional feature of a CISC: there are instructions that take **more than one clock cycle** to execute
- However, the **parts of x86 that you should be using** in your compiler are actually closer to a traditional RISC architecture
  - ***RISC*** = "***reduced instruction set computer***"
  - most complex instructions exist for backward-compatibility and can be slow
  - other complex instructions exist to take advantage of peculiar hardware

x86-64 Main Features

x86-64 Main Features

- 16 64-bit general registers; 64-bit integers
  - but int is 32 bits usually; long is 64 bits

x86-64 Main Features

- 16 64-bit general registers; 64-bit integers
  - but int is 32 bits usually; long is 64 bits
- 64-bit address space; pointers are 8 bytes

x86-64 Main Features

- 16 64-bit general registers; 64-bit integers
  - but int is 32 bits usually; long is 64 bits
- 64-bit address space; pointers are 8 bytes
- 16 SSE registers for floating point, SIMD

x86-64 Main Features

- 16 64-bit general registers; 64-bit integers
  - but int is 32 bits usually; long is 64 bits
- 64-bit address space; pointers are 8 bytes
- 16 SSE registers for floating point, SIMD
- Register-based function call conventions

x86-64 Main Features

- 16 64-bit general registers; 64-bit integers
  - but int is 32 bits usually; long is 64 bits
- 64-bit address space; pointers are 8 bytes
- 16 SSE registers for floating point, SIMD
- Register-based function call conventions
- Additional addressing modes (pc relative)

x86-64 Main Features

- 16 64-bit general registers; 64-bit integers
  - but int is 32 bits usually; long is 64 bits
- 64-bit address space; pointers are 8 bytes
- 16 SSE registers for floating point, SIMD
- Register-based function call conventions
- Additional addressing modes (pc relative)
- 32-bit legacy mode

x86-64 Main Features

- 16 64-bit general registers; 64-bit integers
  - but int is 32 bits usually; long is 64 bits
- 64-bit address space; pointers are 8 bytes
- 16 SSE registers for floating point, SIMD
- Register-based function call conventions
- Additional addressing modes (pc relative)
- 32-bit legacy mode
- Some pruning of old features

x86-64 Syntax

- Two main assembler languages for x86-64:

x86-64 Syntax

- Two main assembler languages for x86-64:
  - **Intel/Microsoft syntax**: what's in the Intel docs

x86-64 Syntax

- Two main assembler languages for x86-64:
  - **Intel/Microsoft syntax**: what's in the Intel docs
  - **AT&T/GNU syntax**: what we're generating and what's in the linked handouts, course webpage, etc.

x86-64 Syntax

- Two main assembler languages for x86-64:
  - **Intel/Microsoft syntax**: what's in the Intel docs
  - **AT&T/GNU syntax**: what we're generating and what's in the linked handouts, course webpage, etc.
    - You can use `gcc -S` to generate AT&T-style assembly code from C/C++ code for more examples

x86-64 Syntax

- Two main assembler languages for x86-64:
  - **Intel/Microsoft syntax**: what's in the Intel docs
  - **AT&T/GNU syntax**: what we're generating and what's in the linked handouts, course webpage, etc.
    - You can use `gcc -S` to generate AT&T-style assembly code from C/C++ code for more examples
  - I will always use AT&T/GNU syntax

Intel vs AT&T Syntax

	Intel/Microsoft	**AT&T/GNU**

Intel vs AT&T Syntax

	Intel/Microsoft	**AT&T/GNU**
Operand order	a = a op b (dst first)	b = a op b (dst last)

Intel vs AT&T Syntax

	Intel/Microsoft	**AT&T/GNU**
Operand order	a = a op b (dst first)	b = a op b (dst last)
Memory addresses	[register+offset]	offset(register)

Intel vs AT&T Syntax

	Intel/Microsoft	**AT&T/GNU**
Operand order	a = a op b (dst first)	b = a op b (dst last)
Memory addresses	[register+offset]	offset(register)
Instruction mnemonics	mov, add, push, …	movq, addq, pushq (explicit operand size after op)

Intel vs AT&T Syntax

	Intel/Microsoft	**AT&T/GNU**
Operand order	a = a op b (dst first)	b = a op b (dst last)
Memory addresses	[register+offset]	offset(register)
Instruction mnemonics	mov, add, push, …	movq, addq, pushq (explicit operand size after op)
Register names	rax, rbx, rbp, rsp, …	%rax, %rbx, %rbp, %rsp, …

Intel vs AT&T Syntax

	Intel/Microsoft	**AT&T/GNU**
Operand order	a = a op b (dst first)	b = a op b (dst last)
Memory addresses	[register+offset]	offset(register)
Instruction mnemonics	mov, add, push, …	movq, addq, pushq (explicit operand size after op)
Register names	rax, rbx, rbp, rsp, …	%rax, %rbx, %rbp, %rsp, …
Constants	17, 42	$17, $42

Intel vs AT&T Syntax

	Intel/Microsoft	**AT&T/GNU**
Operand order	a = a op b (dst first)	b = a op b (dst last)
Memory addresses	[register+offset]	offset(register)
Instruction mnemonics	mov, add, push, …	movq, addq, pushq (explicit operand size after op)
Register names	rax, rbx, rbp, rsp, …	%rax, %rbx, %rbp, %rsp, …
Constants	17, 42	$17, $42
Comments	; to end of line	# to end of line or /* … */

Intel vs AT&T Syntax

Intel docs include many complex, historical instructions and artifacts that aren't commonly used by modern compilers

	Intel/Microsoft	
Operand order	a = a op b (dst firs	
Memory addresses	[register+offset]	offset(register)
Instruction mnemonics	mov, add, push, …	movq, addq, pushq (explicit operand size after op)
Register names	rax, rbx, rbp, rsp, …	%rax, %rbx, %rbp, %rsp, …
Constants	17, 42	$17, $42
Comments	; to end of line	# to end of line or /* … */

x86-64 Memory Model

x86-64 Memory Model

- 8-bit bytes, byte-addressable

x86-64 Memory Model

- 8-bit bytes, byte-addressable
- 16-, 32-, 64-bit ***words***, **double words** and **quad words** (Intel terminology)

x86-64 Memory Model

- 8-bit bytes, byte-addressable
- 16-, 32-, 64-bit ***words***, **double words** and **quad words** (Intel terminology)
  - That's why the 'q' in 64-bit instructions like movq, addq, etc.

x86-64 Memory Model

- 8-bit bytes, byte-addressable
- 16-, 32-, 64-bit ***words***, **double words** and **quad words** (Intel terminology)
  - That's why the 'q' in 64-bit instructions like movq, addq, etc.
- Data should normally be ***aligned*** on "natural" boundaries

x86-64 Memory Model

- 8-bit bytes, byte-addressable
- 16-, 32-, 64-bit ***words***, **double words** and **quad words** (Intel terminology)
  - That's why the 'q' in 64-bit instructions like movq, addq, etc.
- Data should normally be ***aligned*** on "natural" boundaries
  - unaligned accesses are generally supported, but with a big performance penalty on modern machines

x86-64 Memory Model

- 8-bit bytes, byte-addressable
- 16-, 32-, 64-bit ***words***, **double words** and **quad words** (Intel terminology)
  - That's why the 'q' in 64-bit instructions like movq, addq, etc.
- Data should normally be ***aligned*** on "natural" boundaries
  - unaligned accesses are generally supported, but with a big performance penalty on modern machines
  - though function calls into e.g., glibc need to have the stack 16-bit aligned (ask me how I know)

x86-64 Memory Model

- 8-bit bytes, byte-addressable
- 16-, 32-, 64-bit ***words***, **double words** and **quad words** (Intel terminology)
  - That's why the 'q' in 64-bit instructions like movq, addq, etc.
- Data should normally be ***aligned*** on "natural" boundaries
  - unaligned accesses are generally supported, but with a big performance penalty on modern machines
  - though function calls into e.g., glibc need to have the stack 16-bit aligned (ask me how I know)
- ***Little-endian***: address of a multi-byte integer is address of low-order byte

x86-64 Registers

x86-64 Registers

- 16 64-bit general registers
  - %rax, %rbx, %rcx, %rdx, %rsi, %rdi, %rbp, %rsp, %r8-%r15

x86-64 Registers

- 16 64-bit general registers
  - %rax, %rbx, %rcx, %rdx, %rsi, %rdi, %rbp, %rsp, %r8-%r15
- Registers can be used as 64-bit integers or pointers, or as 32-bit ints

x86-64 Registers

- 16 64-bit general registers
  - %rax, %rbx, %rcx, %rdx, %rsi, %rdi, %rbp, %rsp, %r8-%r15
- Registers can be used as 64-bit integers or pointers, or as 32-bit ints
  - Also possible to reference low-order 16- and 8-bit chunks
    - (for the most part you shouldn’t need to)

x86-64 Registers

- 16 64-bit general registers
  - %rax, %rbx, %rcx, %rdx, %rsi, %rdi, %rbp, %rsp, %r8-%r15
- Registers can be used as 64-bit integers or pointers, or as 32-bit ints
  - Also possible to reference low-order 16- and 8-bit chunks
    - (for the most part you shouldn't need to)
- To simplify your project, **all Cool types have the same size** (ints, pointers, even booleans!)

x86-64 Registers

- 16 64-bit general registers
  - %rax, %rbx, %rcx, %rdx, %rsi, %rdi, %rbp, %rsp, %r8-%r15
- Registers can be used as 64-bit integers or pointers, or as 32-bit ints
  - Also possible to reference low-order 16- and 8-bit chunks
    - (for the most part you shouldn't need to)
- To simplify your project, **all Cool types have the same size** (ints, pointers, even booleans!)
  - I suggest you use 64 bits to store them to make your life easy
    - 32 bits is okay too, but it's your funeral

Processor Fetch-Execute Cycle

- Basic cycle (same as every processor you've ever seen):

Processor Fetch-Execute Cycle

- Basic cycle (same as every processor you've ever seen):

```
while (running) {
```

Processor Fetch-Execute Cycle

- Basic cycle (same as every processor you've ever seen):

```
while (running) {
   fetch instruction beginning at rip address
```

Processor Fetch-Execute Cycle

- Basic cycle (same as every processor you've ever seen):

```
while (running) {
   fetch instruction beginning at rip address
   rip <- rip + instruction length
```

Processor Fetch-Execute Cycle

- Basic cycle (same as every processor you've ever seen):

```
while (running) {
   fetch instruction beginning at rip address
   rip <- rip + instruction length
   execute instruction
}
```

Processor Fetch-Execute Cycle

- Basic cycle (same as every processor you've ever seen):

```
while (running) {
   fetch instruction beginning at rip address
   rip <- rip + instruction length
   execute instruction
}
```

- Sequential execution unless a jump stores a new "next instruction" address in %rip

Processor Fetch-Execute Cycle

- Basic cycle (same as every processor you've ever seen):

```
while (running) {
   fetch instruction beginning at rip address
   rip <- rip + instruction length
   execute instruction
}
```

- Sequential execution unless a jump stores a new "next instruction" address in %rip
  - %rip is a hidden register; cannot access directly as a register from assembly code, change by sequential instruction execution and jumps (including call and return)

Instruction Format

Instruction Format

- Typical **data manipulation** instruction:

  opcode src, dst # comment

Instruction Format

- Typical **data manipulation** instruction:

  opcode src, dst # comment

- Meaning is:

  dst <- dst op src

Instruction Format

- Typical **data manipulation** instruction:

  opcode src, dst # comment

- Meaning is:

  dst <- dst op src

- Normally, one operand is a register; the other is a register, memory location, or integer constant

Instruction Format

- Typical **data manipulation** instruction:

  opcode src, dst # comment

- Meaning is:

  dst <- dst op src

- Normally, one operand is a register; the other is a register, memory location, or integer constant
  - **Can’t have both operands in memory** – can’t encode two memory addresses in a single instruction (e.g., cmp, mov)

Instruction Format

- Typical **data manipulation** instruction:

  opcode src, dst # comment

- Meaning is:

  dst <- dst op src

- Normally, one operand is a register; the other is a register, memory location, or integer constant
  - **Can't have both operands in memory** – can't encode two memory addresses in a single instruction (e.g., cmp, mov)
- Language is free-form, comments and labels may appear on lines by themselves (and can have multiple labels per line of code)

x86-64 Memory Stack

x86-64 Memory Stack

- Register %rsp points to the “top” of stack

x86-64 Memory Stack

- Register %rsp points to the “top” of stack
  - Dedicated for this use; **don’t use for anything else**!

x86-64 Memory Stack

- Register %rsp points to the “top” of stack
  - Dedicated for this use; **don’t use for anything else**!
- Points to the last 64-bit quadword pushed onto the stack (not next “free” quadword)

x86-64 Memory Stack

- Register %rsp points to the “top” of stack
  - Dedicated for this use; **don’t use for anything else**!
- Points to the last 64-bit quadword pushed onto the stack (not next “free” quadword)
  - Should always be quadword (8-byte) aligned
    - It will start out this way, and will stay aligned unless your code does something bad

x86-64 Memory Stack

- Register %rsp points to the “top” of stack
  - Dedicated for this use; **don’t use for anything else**!
- Points to the last 64-bit quadword pushed onto the stack (not next “free” quadword)
  - Should always be quadword (8-byte) aligned
    - It will start out this way, and will stay aligned unless your code does something bad
- Should be **16-byte aligned on function calls**

x86-64 Memory Stack

- Register %rsp points to the “top” of stack
  - Dedicated for this use; **don’t use for anything else**!
- Points to the last 64-bit quadword pushed onto the stack (not next “free” quadword)
  - Should always be quadword (8-byte) aligned
    - It will start out this way, and will stay aligned unless your code does something bad
- Should be **16-byte aligned on function calls**
- Stack **grows down** (towards lower addresses)

Stack Instructions

Stack Instructions

- `pushq src`
  - %rsp <- %rsp – 8; memory[%rsp] <- src (e.g., push src onto the stack)

Stack Instructions

- `pushq src`
  - %rsp <- %rsp – 8; memory[%rsp] <- src (e.g., push src onto the stack)
- `popq dst`
  - dst <- memory[%rsp]; %rsp <- %rsp + 8
    - (e.g., pop top of stack into dst and logically remove it from the stack)

Stack Frames

Stack Frames

- When a method is called, a stack frame is normally allocated on the logical “top” of the stack to hold its **local variables**

Stack Frames

- When a method is called, a stack frame is normally allocated on the logical “top” of the stack to hold its **local variables**
  - Stack actually grows down towards lower memory addresses when a new stack frame is pushed (allocated)

Stack Frames

- When a method is called, a stack frame is normally allocated on the logical “top” of the stack to hold its **local variables**
  - Stack actually grows down towards lower memory addresses when a new stack frame is pushed (allocated)
- Frame is popped on method return

Stack Frames

- When a method is called, a stack frame is normally allocated on the logical “top” of the stack to hold its **local variables**
  - Stack actually grows down towards lower memory addresses when a new stack frame is pushed (allocated)
- Frame is popped on method return
- By convention, %rbp (base pointer) points to a **known offset** into the current active stack frame

Stack Frames

- When a method is called, a stack frame is normally allocated on the logical “top” of the stack to hold its **local variables**
  - Stack actually grows down towards lower memory addresses when a new stack frame is pushed (allocated)
- Frame is popped on method return
- By convention, %rbp (base pointer) points to a **known offset** into the current active stack frame
  - Local variables referenced relative to %rbp

Stack Frames

- When a method is called, a stack frame is normally allocated on the logical “top” of the stack to hold its **local variables**
  - Stack actually grows down towards lower memory addresses when a new stack frame is pushed (allocated)
- Frame is popped on method return
- By convention, %rbp (base pointer) points to a **known offset** into the current active stack frame
  - Local variables referenced relative to %rbp
  - Base pointer common in 32-bit x86 code; less so in x86-64 code where push/pop used less & stack frame normally has fixed size so locals can be referenced from %rsp easily

Operand Address Modes (1)

- These should cover most of what you'll need:

Operand Address Modes (1)

- These should cover most of what you'll need:

```
movq $17,%rax                    # store 17 in %rax
```

Operand Address Modes (1)

- These should cover most of what you'll need:

```
movq $17,%rax              # store 17 in %rax
movq %rcx,%rax          # copy %rcx to %rax
```

Operand Address Modes (1)

- These should cover most of what you'll need:

```
movq $17,%rax              # store 17 in %rax
movq %rcx,%rax          # copy %rcx to %rax
movq 16(%rbp),%rax      # copy memory to %rax
```

Operand Address Modes (1)

- These should cover most of what you'll need:

```
movq $17,%rax              # store 17 in %rax
movq %rcx,%rax           # copy %rcx to %rax
movq 16(%rbp),%rax         # copy memory to %rax
movq %rax,-24(%rbp)        # copy %rax to memory
```

Operand Address Modes (1)

- These should cover most of what you'll need:

```
movq $17,%rax              # store 17 in %rax
movq %rcx,%rax           # copy %rcx to %rax
movq 16(%rbp),%rax         # copy memory to %rax
movq %rax,-24(%rbp)        # copy %rax to memory
```

- References to object fields work similarly – put the object's memory address in a register and use that address plus an offset

Operand Address Modes (1)

- These should cover most of what you'll need:

```
movq $17,%rax              # store 17 in %rax
movq %rcx,%rax           # copy %rcx to %rax
movq 16(%rbp),%rax         # copy memory to %rax
movq %rax,-24(%rbp)        # copy %rax to memory
```

- References to object fields work similarly – put the object's memory address in a register and use that address plus an offset
- Remember: can't have two memory addresses in a single instruction

Operand Address Modes (2)

- A memory address can combine the contents of two registers (with one optionally multiplied by 2, 4, or 8) plus a constant:

  base address + (index register * scale) + constant

Operand Address Modes (2)

- A memory address can combine the contents of two registers (with one optionally multiplied by 2, 4, or 8) plus a constant:
  base address + (index register * scale) + constant
- Main use of general form is for array subscripting or small computations - if the compiler is clever

Operand Address Modes (2)

- A memory address can combine the contents of two registers (with one optionally multiplied by 2, 4, or 8) plus a constant:
  base address + (index register * scale) + constant
- Main use of general form is for array subscripting or small computations - if the compiler is clever
- Example: suppose we have an array A of 8-byte ints with the address of the array in %rcx and the index i in %rax.

Operand Address Modes (2)

- A memory address can combine the contents of two registers (with one optionally multiplied by 2, 4, or 8) plus a constant:
  base address + (index register * scale) + constant
- Main use of general form is for array subscripting or small computations - if the compiler is clever
- Example: suppose we have an array A of 8-byte ints with the address of the array in %rcx and the index i in %rax.
- Code to store %rbx in A[i]:

```
movq %rbx, 0(%rcx, %rax, 8)
```

Operand Address Modes (2)

- A memory address can combine the contents of two registers (with one optionally multiplied by 2, 4, or 8) plus a constant:

  base address + (index register * scale) + constant
- Main use of general form is for array subscripting or small computations - if the compiler is clever
- Example: suppose we have an array A of 8-byte ints with the address of the array in %rcx and the index i in %rax.
- Code to store %rbx in A[i]:

```
movq %rbx, 0(%rcx, %rax, 8)
```

base address

Operand Address Modes (2)

- A memory address can combine the contents of two registers (with one optionally multiplied by 2, 4, or 8) plus a constant:
  base address + (**index register** * scale) + constant
- Main use of general form is for array subscripting or small computations - if the compiler is clever
- Example: suppose we have an array A of 8-byte ints with the address of the array in %rcx and the index i in %rax.
- Code to store %rbx in A[i]:

```
movq %rbx, 0(%rcx, %rax, 8)
```

index register

Operand Address Modes (2)

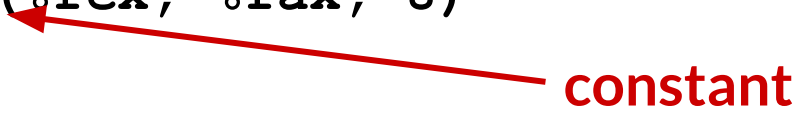
- A memory address can combine the contents of two registers (with one optionally multiplied by 2, 4, or 8) plus a constant:
  base address + (index register * **scale**) + constant
- Main use of general form is for array subscripting or small computations - if the compiler is clever
- Example: suppose we have an array A of 8-byte ints with the address of the array in %rcx and the index i in %rax.
- Code to store %rbx in A[i]:

```
movq %rbx, 0(%rcx, %rax, 8)
```

Operand Address Modes (2)

- A memory address can combine the contents of two registers (with one optionally multiplied by 2, 4, or 8) plus a constant:
  base address + (index register * scale) + **constant**
- Main use of general form is for array subscripting or small computations - if the compiler is clever
- Example: suppose we have an array A of 8-byte ints with the address of the array in %rcx and the index i in %rax.
- Code to store %rbx in A[i]:

```
movq %rbx, 0(%rcx, %rax, 8)
```



Basic Data Movement/Arithmetic Instructions

Basic Data Movement/Arithmetic Instructions

`movq src,dst`

dst <- src

Basic Data Movement/Arithmetic Instructions

`movq src,dst`

dst <- src

`addq src,dst`

dst <- dst + src

Basic Data Movement/Arithmetic Instructions

`movq src,dst`
    dst <- src

`addq src,dst`
    dst <- dst + src

`subq src,dst`
    dst <- dst−src

Basic Data Movement/Arithmetic Instructions

`movq src,dst`
    dst <- src

`addq src,dst`
    dst <- dst + src

`subq src,dst`
    dst <- dst−src

`incq dst`
    dst <- dst + 1

`decq dst`
    dst <- dst - 1

Basic Data Movement/Arithmetic Instructions

`movq src,dst`
dst <- src

`addq src,dst`
dst <- dst + src

`subq src,dst`
dst <- dst−src

`incq dst`
dst <- dst + 1

`decq dst`
dst <- dst - 1

`negq dst`
dst <- -dst
(2’s complement arithmetic negation)

Integer Multiply and Divide

Integer Multiply and Divide

`imulq src,dst`

dst ⟵ dst * src

dst must be a register

Integer Multiply and Divide

`imulq src,dst`

dst ⟵ dst * src

dst must be a register

`idivq src`

Divide %rdx:%rax by src

(%rdx:%rax holds sign-extended 128-bit value; cannot use other registers for division!)

%rax <- quotient

%rdx <- remainder

Integer Multiply and Divide

`imulq src,dst`

dst ⟵ dst * src

dst must be a register

`cqto`

%rdx:%rax <- 128-bit sign extended copy of %rax

(why? To prep numerator for idivq!)

`idivq src`

Divide %rdx:%rax by src

(%rdx:%rax holds sign-extended 128-bit value; cannot use other registers for division!)

%rax <- quotient

%rdx <- remainder

Bitwise Operators

`andq src,dst`

dst <- src & dst

Bitwise Operators

`andq src,dst`
    dst <- src & dst

`orq src,dst`
    dst <- dst | src

`xorq src,dst`
    dst <- dst ^ src

Bitwise Operators

`andq src,dst`
dst <- src & dst

`orq src,dst`
dst <- dst | src

`xorq src,dst`
dst <- dst ^ src

`notq dst`
dst <- ~dst
(1's complement logical negation)

Bitwise Operators

`andq src,dst`
    dst <- src & dst

`orq src,dst`
    dst <- dst | src

`xorq src,dst`
    dst <- dst ^ src

`notq dst`
    dst <- ~dst
    (1's complement logical negation)

Note similarity between **`notq`** and **`negq`** (a few slides back). Difference is 1's vs 2's complement negation.

Shifts and Rotates

Shifts and Rotates

`shlq dst,count`

dst <- dst shifted left count bits

Shifts and Rotates

`shlq dst,count`

dst <- dst shifted left count bits

`shrq dst,count`

dst <- dst shifted right count bits (0 fill)

Shifts and Rotates

`shlq dst,count`
dst <- dst shifted left count bits

`shrq dst,count`
dst <- dst shifted right count
bits (0 fill)

`sarq dst,count`
dst <- dst shifted right count
bits (sign bit fill)

Shifts and Rotates

`shlq dst,count`
dst <- dst shifted left count bits

`shrq dst,count`
dst <- dst shifted right count bits (0 fill)

`sarq dst,count`
dst <- dst shifted right count bits (sign bit fill)

`rolq dst,count`
dst <- dst rotated left count bits

`rorq dst,count`
dst <- dst rotated right count bits

Uses for Shifts and Rotates

Uses for Shifts and Rotates

- **Very fast** and can often be used to **optimize multiplication and division** by small constants
  - If you're interested, look at "Hacker's Delight" by Henry Warren

Uses for Shifts and Rotates

- **Very fast** and can often be used to **optimize multiplication and division** by small constants
  - If you're interested, look at "Hacker's Delight" by Henry Warren
- Lots of very cool bit fiddling and other algorithms
  - But **be careful** in the project: be sure semantics are OK

Uses for Shifts and Rotates

- **Very fast** and can often be used to **optimize multiplication and division** by small constants
  - If you're interested, look at "Hacker's Delight" by Henry Warren
- Lots of very cool bit fiddling and other algorithms
  - But **be careful** in the project: be sure semantics are OK
- Example: right shift is not the same as Java/C/C++/etc. integer divide for negative numbers (why?)

Uses for Shifts and Rotates

- **Very fast** and can often be used to **optimize multiplication and division** by small constants
  - If you're interested, look at "Hacker's Delight" by Henry Warren
- Lots of very cool bit fiddling and other algorithms
  - But **be careful** in the project: be sure semantics are OK
- Example: right shift is not the same as Java/C/C++/etc. integer divide for negative numbers (why?)
- There are additional instructions that shift and rotate double words, use a calculated shift amount instead of a constant, etc.

Uses for Shifts and Rotates

- **Very fast** and can often be used to **optimize multiplication and division** by small constants
  - If you're interested, look at "Hacker's Delight" by Henry Warren
- Lots of very cool bit fiddling and other algorithms
  - But **be careful** in the project: be sure semantics are OK
- Example: right shift is not the same as divide for negative numbers (why?)
- There are additional instructions for double words, use a calculated shift amount instead of a constant, etc.

Should you use any of these tricks in PA3?

Uses for Shifts and Rotates

- **Very fast** and can often be used to **optimize multiplication and division** by small constants
  - If you're interested, look at "Hacker's Delight" by Henry Warren
- Lots of very cool bit fiddling and other algorithms
  - But **be careful** in the project: be sure semantics are OK
- Example: right shift is not the same as divide for negative numbers (why?)
- There are additional instructions for double-words, use a calculated shift amount instead of a constant, etc.

Should you use any of these tricks in PA3? **NO!** (encouraged in PA4!)

Load Effective Address (`lea`)

Load Effective Address (`lea`)

- The unary `&` operator in C/C++

Load Effective Address (`lea`)

- The unary & operator in C/C++

```
leaq src,dst # dst <- address of src
```

Load Effective Address (`lea`)

- The unary & operator in C/C++

```
leaq src,dst # dst <- address of src
```

- Things to note:

Load Effective Address (`lea`)

- The unary & operator in C/C++

```
leaq src,dst # dst <- address of src
```

- Things to note:
  - **`dst`** must be a register

Load Effective Address (`lea`)

- The unary & operator in C/C++

```
leaq src,dst # dst <- address of src
```

- Things to note:
  - **`dst`** must be a register
  - Address of **`src`** includes any address arithmetic or indexing

Load Effective Address (`lea`)

- The unary & operator in C/C++

```
leaq src,dst # dst <- address of src
```

- Things to note:
  - **`dst`** must be a register
  - Address of **`src`** includes any address arithmetic or indexing
  - Useful to capture addresses for pointers, reference parameters, etc.

Load Effective Address (`lea`)

- The unary & operator in C/C++

```
leaq src,dst # dst <- address of src
```

- Things to note:
  - **`dst`** must be a register
  - Address of **`src`** includes any address arithmetic or indexing
  - Useful to capture addresses for pointers, reference parameters, etc.
  - Also useful for computing arithmetic expressions of the form *const + r1 + scale * r2*

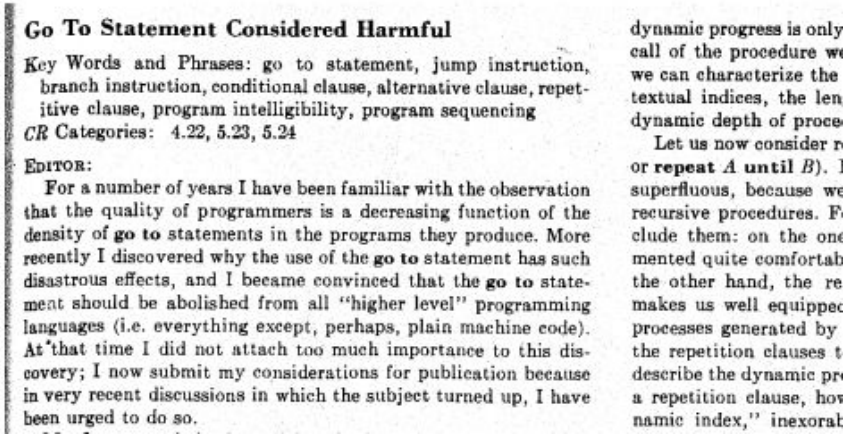
Trivia Break: Computer Science

This Dutch computer scientist won the 1972 Turing Award for fundamental contributions to developing structured programming languages. Some of his other important contributions include formulating and solving the shortest-path problem in graph theory and co-developing the first Algol 60 compiler. He also famously authored a long, eponymous series of technical reports on various topics both within and without computer science. If you took 490 with me, you might remember his famous aphorism: “tests can only show the presence of bugs, never their absence”.

Control Flow: GOTO

Control Flow: GOTO

Edgar Dijkstra: Go To Statement Considered Harmful



Go To Statement Considered Harmful

Key Words and Phrases: go to statement, jump instruction, branch instruction, conditional clause, alternative clause, repetitive clause, program intelligibility, program sequencing
CR Categories: 4.22, 5.23, 5.24

EDITOR:

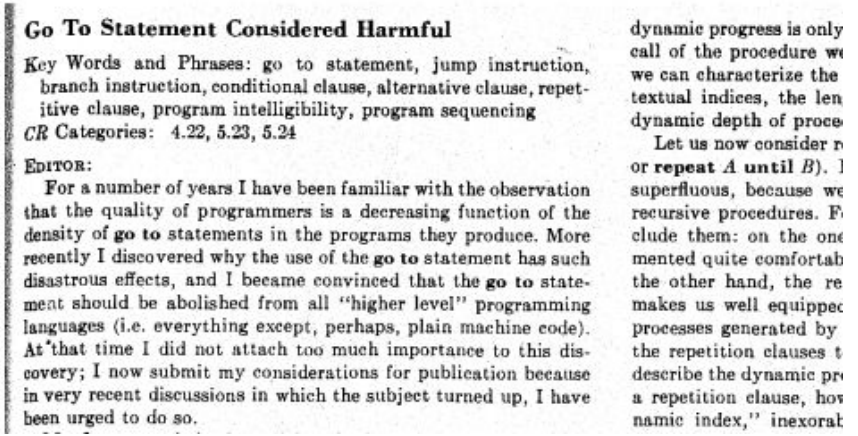
For a number of years I have been familiar with the observation that the quality of programmers is a decreasing function of the density of **go to** statements in the programs they produce. More recently I discovered why the use of the **go to** statement has such disastrous effects, and I became convinced that the **go to** statement should be abolished from all "higher level" programming languages (i.e. everything except, perhaps, plain machine code). At that time I did not attach too much importance to this discovery; I now submit my considerations for publication because in very recent discussions in which the subject turned up, I have been urged to do so.

dynamic progress is only
call of the procedure we
we can characterize the
textual indices, the len
dynamic depth of proce
Let us now consider r
or **repeat** *A* **until** *B*).
superfluous, because we
recursive procedures. F
clude them: on the one
mented quite comfortab
the other hand, the re
makes us well equippe
processes generated by
the repetition clauses t
describe the dynamic pr
a repetition clause, hov
namic index," inexorab

Control Flow: GOTO

- At the assembly level, all we have is goto and conditional goto

Edgar Dijkstra: Go To Statement Considered Harmful



Go To Statement Considered Harmful

Key Words and Phrases: go to statement, jump instruction, branch instruction, conditional clause, alternative clause, repetitive clause, program intelligibility, program sequencing
CR Categories: 4.22, 5.23, 5.24

EDITOR:

For a number of years I have been familiar with the observation that the quality of programmers is a decreasing function of the density of **go to** statements in the programs they produce. More recently I discovered why the use of the **go to** statement has such disastrous effects, and I became convinced that the **go to** statement should be abolished from all "higher level" programming languages (i.e. everything except, perhaps, plain machine code). At that time I did not attach too much importance to this discovery; I now submit my considerations for publication because in very recent discussions in which the subject turned up, I have been urged to do so.

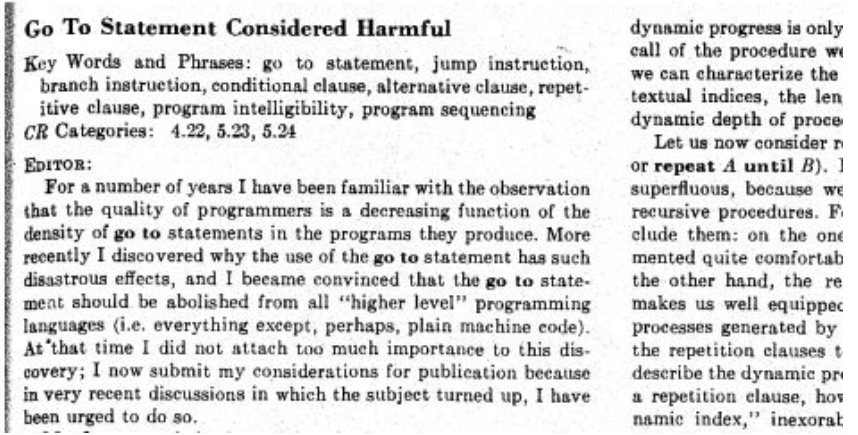
dynamic progress is only
call of the procedure we
we can characterize the
textual indices, the len
dynamic depth of proce

Let us now consider r
or **repeat** *A* **until** *B*). I
superfluous, because we
recursive procedures. F
clude them: on the one
mented quite comfortab
the other hand, the re
makes us well equippe
processes generated by
the repetition clauses t
describe the dynamic pr
a repetition clause, hov
namic index," inexorab

Control Flow: GOTO

- At the assembly level, all we have is goto and conditional goto
- Loops and conditional statements are built from these

Edgar Dijkstra: Go To Statement Considered Harmful



Go To Statement Considered Harmful

Key Words and Phrases: go to statement, jump instruction, branch instruction, conditional clause, alternative clause, repetitive clause, program intelligibility, program sequencing
CR Categories: 4.22, 5.23, 5.24

EDITOR:

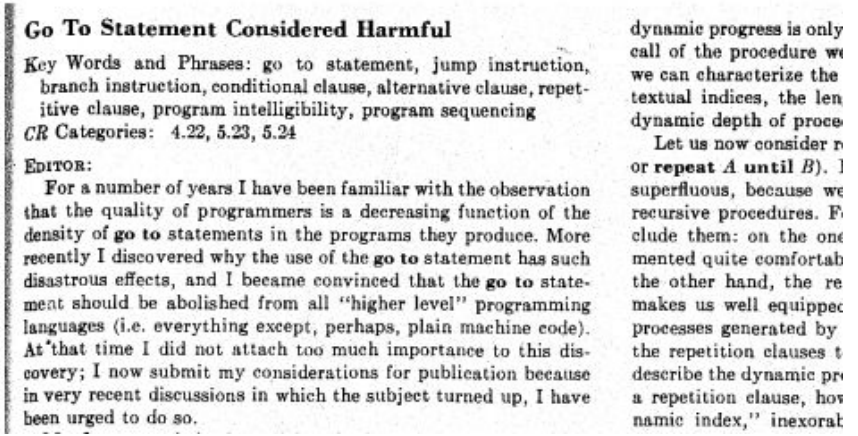
For a number of years I have been familiar with the observation that the quality of programmers is a decreasing function of the density of **go to** statements in the programs they produce. More recently I discovered why the use of the **go to** statement has such disastrous effects, and I became convinced that the **go to** statement should be abolished from all "higher level" programming languages (i.e. everything except, perhaps, plain machine code). At that time I did not attach too much importance to this discovery; I now submit my considerations for publication because in very recent discussions in which the subject turned up, I have been urged to do so.

dynamic progress is only
call of the procedure we
we can characterize the
textual indices, the len
dynamic depth of proce
Let us now consider r
or **repeat** *A* **until** *B*).
superfluous, because we
recursive procedures. F
clude them: on the one
mented quite comfortab
the other hand, the re
makes us well equippe
processes generated by
the repetition clauses t
describe the dynamic pr
a repetition clause, hov
namic index," inexorab

Control Flow: GOTO

- At the assembly level, all we have is goto and conditional goto
- Loops and conditional statements are built from these
- Note: random jumps play havoc with pipeline efficiency; much work is done in modern compilers and processors to minimize this impact

Edgar Dijkstra: Go To Statement Considered Harmful



Go To Statement Considered Harmful

Key Words and Phrases: go to statement, jump instruction, branch instruction, conditional clause, alternative clause, repetitive clause, program intelligibility, program sequencing
CR Categories: 4.22, 5.23, 5.24

EDITOR:

For a number of years I have been familiar with the observation that the quality of programmers is a decreasing function of the density of **go to** statements in the programs they produce. More recently I discovered why the use of the **go to** statement has such disastrous effects, and I became convinced that the **go to** statement should be abolished from all "higher level" programming languages (i.e. everything except, perhaps, plain machine code). At that time I did not attach too much importance to this discovery; I now submit my considerations for publication because in very recent discussions in which the subject turned up, I have been urged to do so.

dynamic progress is only
call of the procedure we
we can characterize the
textual indices, the len
dynamic depth of proce
Let us now consider r
or **repeat** *A* **until** *B*).
superfluous, because we
recursive procedures. F
clude them: on the one
mented quite comfortab
the other hand, the re
makes us well equippe
processes generated by
the repetition clauses t
describe the dynamic pr
a repetition clause, hov
namic index," inexorab

Unconditional Jumps

Unconditional Jumps

`jmp dst`

%rip <- address of dst

Unconditional Jumps

`jmp dst`

%rip <- address of dst

- dst is usually a ***label*** in the code (which can be on a line by itself)

Unconditional Jumps

`jmp dst`

%rip <- address of dst

- dst is usually a ***label*** in the code (which can be on a line by itself)
  - “labels” are just arbitrary named instructions chosen by the compiler (i.e., you)

Unconditional Jumps

`jmp dst`

%rip <- address of dst

- dst is usually a ***label*** in the code (which can be on a line by itself)
  - “labels” are just arbitrary named instructions chosen by the compiler (i.e., you)
- dst address can also be indirect using the address in a register or memory location (*reg or *(reg))
  - useful for method calls, case, etc.

Conditional Jumps

Conditional Jumps

- Most arithmetic instructions set "***condition code***" bits to record information about the result (zero, non-zero, >0, etc. - read the docs for an instruction before you rely on this behavior)

Conditional Jumps

- Most arithmetic instructions set "***condition code***" bits to record information about the result (zero, non-zero, >0, etc. - read the docs for an instruction before you rely on this behavior)
  - true of: **`addq, subq, andq, orq`**

Conditional Jumps

- Most arithmetic instructions set "***condition code***" bits to record information about the result (zero, non-zero, >0, etc. - read the docs for an instruction before you rely on this behavior)
  - true of: **`addq, subq, andq, orq`**
  - but not: **`imulq, idivq, leaq`**

Conditional Jumps

- Most arithmetic instructions set "***condition code***" bits to record information about the result (zero, non-zero, >0, etc. - read the docs for an instruction before you rely on this behavior)
  - true of: **`addq, subq, andq, orq`**
  - but not: **`imulq, idivq, leaq`**
- Other instructions can set condition codes. E.g.,:

Conditional Jumps

- Most arithmetic instructions set "***condition code***" bits to record information about the result (zero, non-zero, >0, etc. - read the docs for an instruction before you rely on this behavior)
  - true of: **`addq, subq, andq, orq`**
  - but not: **`imulq, idivq, leaq`**
- Other instructions can set condition codes. E.g.,:
  - **`cmpq src,dst`** # compare dst to src (e.g., dst-src)
  - **`testq src,dst`** # calculate dst & src (logical and)

Conditional Jumps

- Most arithmetic instructions set "***condition code***" bits to record information about the result (zero, non-zero, >0, etc. - read the docs for an instruction before you rely on this behavior)
  - true of: **`addq, subq, andq, orq`**
  - but not: **`imulq, idivq, leaq`**
- Other instructions can set condition codes. E.g.,:
  - **`cmpq src,dst`** # compare dst to src (e.g., dst-src)
  - **`testq src,dst`** # calculate dst & src (logical and)
    - These do not alter src or dst, but then *do* impact jump targets of conditional jumps

Conditional Jumps: Examples

```
jz label          # jump if result  == 0
```

Conditional Jumps: Examples

```
jz label          # jump if result  == 0
jnz    label      # jump if result != 0
```

Conditional Jumps: Examples

```
jz label          # jump if result  == 0
jnz    label      # jump if result != 0
jg     label      # jump if result > 0
```

Conditional Jumps: Examples

```
jz label        # jump if result  == 0
jnz   label     # jump if result != 0
jg    label     # jump if result > 0
jnglabel        # jump if result  <= 0
jgelabel        # jump if result >= 0
jnge  label     # jump if result < 0
jl    label     # jump if result < 0
jnllabel        # jump if result >= 0
jlelabel        # jump if result <= 0
jnle  label     # jump if result > 0
```

Conditional Jumps: Examples

```
jz label        # jump if result  == 0
jnz   label     # jump if result != 0
jg    label     # jump if result > 0
jnglabel        # jump if result  <= 0
jgelabel        # jump if result >= 0
jnge  label     # jump if result < 0
jl    label     # jump if result < 0
jnllabel        # jump if result >= 0
jlelabel        # jump if result <= 0
jnle  label     # jump if result > 0
```

note: the assembler is mapping multiple mnemonics to a single actual instruction

Compare and Jump

- Common desire: compare two operands and **jump if a relationship holds** between them

Compare and Jump

- Common desire: compare two operands and **jump if a relationship holds** between them
  - in other words, we *want* an instruction like this:

    $\texttt{jump}_{\texttt{cond}}$ `op1, op2, label`

Compare and Jump

- Common desire: compare two operands and **jump if a relationship holds** between them
  - in other words, we *want* an instruction like this:

    $\texttt{jump}_{\texttt{cond}}$ `op1, op2, label`

- However, we can't have this instruction: x86-64 **cannot support 3-operand instructions**
  - (also true of most other practical machines)

Compare and Jump

- Common desire: compare two operands and **jump if a relationship holds** between them
  - in other words, we *want* an instruction like this:

    `jump`$_{\texttt{cond}}$ `op1, op2, label`

- However, we can't have this instruction: x86-64 **cannot support 3-operand instructions**
  - (also true of most other practical machines)
- What should we do instead?

Compare and Jump

- Common desire: compare two operands and **jump if a relationship holds** between them
  - in other words, we *want* an instruction like this:

    `jump`$_{\texttt{cond}}$ `op1, op2, label`

- However, we can't have this instruction: x86-64 **cannot support 3-operand instructions**
  - (also true of most other practical machines)
- What should we do instead?

```
cmpq   op1, op2  # compute op2 - op1, set condition code
j_cc   label
```

Compare and Jump

- Common desire: compare two operands and **jump if a relationship holds** between them
  - in other words, we *want* an instruction like this:

```
jump_cond op1, op2, label
```

- However, we can't have this instruction: x86-64 **cannot support 3-operand instructions**
  - (also true of most other practical machines)
- What should we do instead?

```
cmpq   op1, op2  # co
j_cc   label
```

j_{cc} is a conditional jump that's taken if comparison result matches cc

Compare and Jump

Special conditional jump instructions like **`je`**, **`jne`** are useful mnemonics here; you can also use the ones on the last slide

- Common desire: compare tw
  relationship holds between
  - in other words, we *want* an instruction like this:

```
jump_cond op1, op2, label
```

- However, we can't have this instruction: x86-64 **cannot support 3-operand instructions**
  - (also true of most other practical machines)
- What should we do instead

```
cmpq   op1, op2  # co
j_cc   label
```

j_{cc} is a conditional jump that's taken if comparison result matches cc

Function Calls and Returns

Function Calls and Returns

- The x86-64 instruction set itself only provides for transfer of control (via **jump**) and return

Function Calls and Returns

- The x86-64 instruction set itself only provides for transfer of control (via **jump**) and return
- Stack is used to capture return address and recover it

Function Calls and Returns

- The x86-64 instruction set itself only provides for transfer of control (via **jump**) and return
- Stack is used to capture return address and recover it
- Everything else–parameter passing, stack frame organization, register usage – is a matter of convention

Function Calls and Returns

- The x86-64 instruction set itself only provides for transfer of control (via **jump**) and return
- Stack is used to capture return address and recover it
- Everything else–parameter passing, stack frame organization, register usage – is a matter of convention
  - Follow the conventions even if you write all the code!

Function Calls and Returns

- The x86-64 instruction set itself only provides for transfer of control (via **jump**) and return
- Stack is used to capture return address and recover it
- Everything else–parameter passing, stack frame organization, register usage – is a matter of convention
  - Follow the conventions even if you write all the code!
    - Helps anyone reading your code figure out what's happening
    - Lets standard tools like gdb work successfully with your code (in the *unlikely* event that you have to debug something...)

`call` and `ret` Instructions

`call` and `ret` Instructions

`call label`

- Push address of next instruction and jump

`call` and `ret` Instructions

`call label`

- Push address of next instruction and jump
- %rsp <- %rsp – 8; memory[%rsp] <- %rip; %rip <- address of label

`call` and `ret` Instructions

`call label`

- Push address of next instruction and jump
- %rsp <- %rsp – 8; memory[%rsp] <- %rip; %rip <- address of label
- Address can also be in a register or memory as with jmp – we'll use these for dynamic dispatch of method calls (more later)

`call` and `ret` Instructions

`call label`

- Push address of next instruction and jump
- %rsp <- %rsp – 8; memory[%rsp] <- %rip; %rip <- address of label
- Address can also be in a register or memory as with jmp – we'll use these for dynamic dispatch of method calls (more later)

`ret`

- Pop address from top of stack and jump

`call` and `ret` Instructions

`call label`

- Push address of next instruction and jump
- %rsp <- %rsp – 8; memory[%rsp] <- %rip; %rip <- address of label
- Address can also be in a register or memory as with jmp – we'll use these for dynamic dispatch of method calls (more later)

`ret`

- Pop address from top of stack and jump
- %rip <- memory[%rsp]; %rsp <- %rsp + 8

`call` and `ret` Instructions

`call label`

- Push address of next instruction and jump
- %rsp <- %rsp – 8; memory[%rsp] <- %rip; %rip <- address of label
- Address can also be in a register or memory as with jmp – we'll use these for dynamic dispatch of method calls (more later)

`ret`

- Pop address from top of stack and jump
- %rip <- memory[%rsp]; %rsp <- %rsp + 8
- **WARNING!** The word on the top of the stack had better be the address we want and not some leftover data

Register Usage

Register Usage

- **%rax** – function result

Register Usage

- **%rax** – function result
- First six arguments passed in these registers, in this order:
  - **%rdi**, **%rsi**, **%rdx**, **%rcx**, **%r8**, **%r9**

Register Usage

- **%rax** – function result
- First six arguments passed in these registers, in this order:
  - **%rdi**, **%rsi**, **%rdx**, **%rcx**, **%r8**, **%r9**
  - For Java/C++ “this” pointer is first argument, in %rdi
    - you may want to do the same for your Cool programs

Register Usage

- **%rax** – function result
- First six arguments passed in these registers, in this order:
  - **%rdi**, **%rsi**, **%rdx**, **%rcx**, **%r8**, **%r9**
  - For Java/C++ “this” pointer is first argument, in %rdi
    - you may want to do the same for your Cool programs
- **%rsp** – stack pointer;value must be 8-byte aligned always and 16-byte aligned when calling a function

Register Usage

- **%rax** – function result
- First six arguments passed in these registers, in this order:
  - **%rdi**, **%rsi**, **%rdx**, **%rcx**, **%r8**, **%r9**
  - For Java/C++ “this” pointer is first argument, in %rdi
    - you may want to do the same for your Cool programs
- **%rsp** – stack pointer;value must be 8-byte aligned always and 16-byte aligned when calling a function
- **%rbp** – frame pointer (optional use)

Register Saving Conventions

Register Saving Conventions

- A called function **must preserve** these registers(or save/restore them if it wants to use them):
  - %rbx, %rbp, %r12-%r15

Register Saving Conventions

- A called function **must preserve** these registers(or save/restore them if it wants to use them):
  - %rbx, %rbp, %r12-%r15
  - %rsp isn't officially on this “callee save list”, but needs to be properly restored for return

Register Saving Conventions

- A called function **must preserve** these registers(or save/restore them if it wants to use them):
  - %rbx, %rbp, %r12-%r15
  - %rsp isn't officially on this “callee save list”, but needs to be properly restored for return
- All other registers **can change** across a function call

Register Saving Conventions

- A called function **must preserve** these registers(or save/restore them if it wants to use them):
  - %rbx, %rbp, %r12-%r15
  - %rsp isn't officially on this “callee save list”, but needs to be properly restored for return
- All other registers **can change** across a function call
  - Debugging/correctness note: always assume every called function will change **all** registers it is allowed to

Register Saving Conventions

- A called function **must preserve** these registers(or save/restore them if it wants to use them):
  - %rbx, %rbp, %r12-%r15
  - %rsp isn't officially on this “callee save list”, but needs to be properly restored for return
- All other registers **can change** across a function call
  - Debugging/correctness note: always assume every called function will change **all** registers it is allowed to
    - including registers containing function parameters!

Register Saving Conventions

- A called function **must preserve** these registers(or save/restore them if it wants to use them):
  - %rbx, %rbp, %r12-%r15
  - %rsp isn't officially on this “callee save list”, but needs to be properly restored for return
- All other registers **can change** across a function call
  - Debugging/correctness note: always assume every called function will change **all** registers it is allowed to
    - including registers containing function parameters!
      - for debugging, you may want to **deliberately** clobber them!

x86-64 Function Call

x86-64 Function Call

- Caller places up to 6 arguments in registers, rest on stack, then executes `call` instruction (which pushes 8- byte return address)

x86-64 Function Call

- Caller places up to 6 arguments in registers, rest on stack, then executes `call` instruction (which pushes 8- byte return address)
- On entry, called function prologue sets up the **stack frame**:

x86-64 Function Call

- Caller places up to 6 arguments in registers, rest on stack, then executes `call` instruction (which pushes 8- byte return address)
- On entry, called function prologue sets up the **stack frame**:

```
pushq %rbp                    # save old frame ptr
```

x86-64 Function Call

- Caller places up to 6 arguments in registers, rest on stack, then executes `call` instruction (which pushes 8- byte return address)
- On entry, called function prologue sets up the **stack frame**:

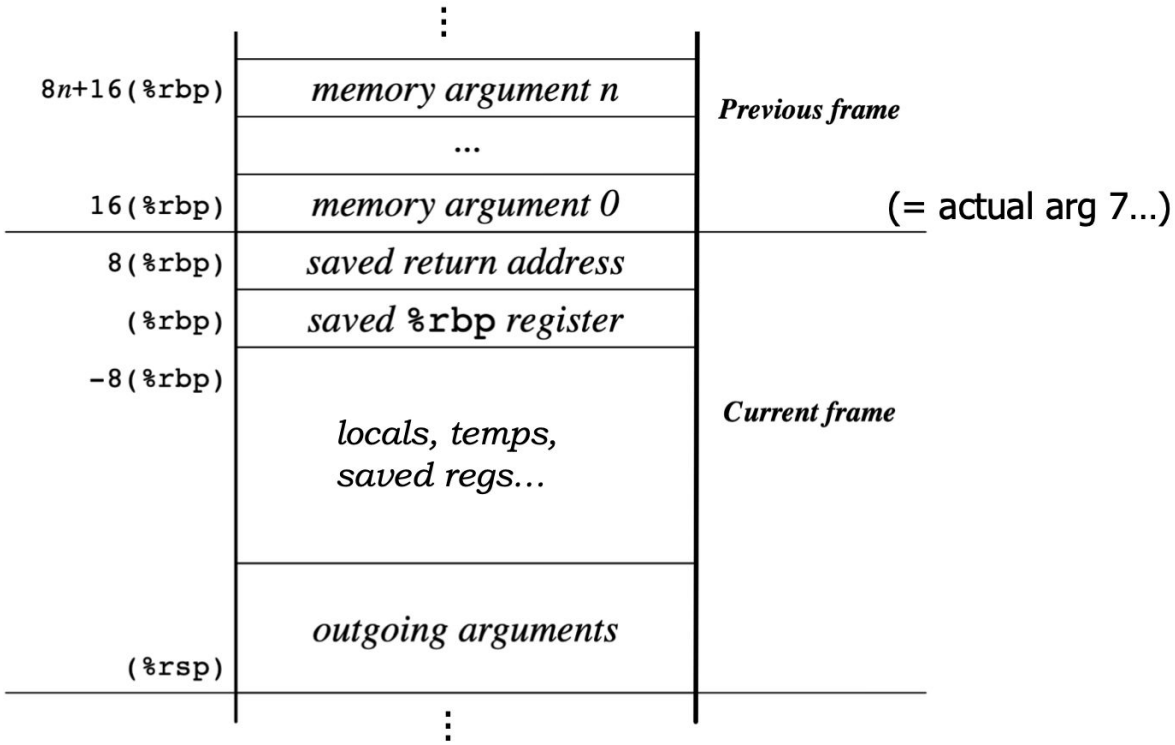
```
pushq %rbp                  # save old frame ptr
movq %rsp,%rbp              # new frame ptr is top of
                            # stack after ret addr and
                            # old rbp pushed
```

x86-64 Function Call

- Caller places up to 6 arguments in registers, rest on stack, then executes `call` instruction (which pushes 8- byte return address)
- On entry, called function prologue sets up the **stack frame**:

```
pushq %rbp                  # save old frame ptr
movq %rsp,%rbp              # new frame ptr is top of
                            # stack after ret addr and
                            # old rbp pushed
subq $framesize,%rsp        # allocate stack frame
                            # (size should be multiple
                            # of 16 normally)
```

Stack Frame Layout



x86-64 Function Return

x86-64 Function Return

- Called function puts result (if any) in %rax and restores any callee-save registers if needed

x86-64 Function Return

- Called function puts result (if any) in %rax and restores any callee-save registers if needed
- Called function returns with:

```
movq %rbp,%rsp     # or can use “leave”
popq %rbp             # instead of movq/popq
```

x86-64 Function Return

- Called function puts result (if any) in %rax and restores any callee-save registers if needed
- Called function returns with:

```
movq %rbp,%rsp     # or can use "leave"
popq %rbp             # instead of movq/popq
```

- If caller allocated space for arguments (beyond the 6 in regs) it deallocates as needed

Caller Example

```
n = sumOf(17,42)
```

Caller Example

```
n = sumOf(17,42)
```

```
movq $42,%rsi                    # load arguments in
```

Caller Example

```
n = sumOf(17,42)
```

```
movq $42,%rsi                # load arguments in
movq $17,%rdi                # either order, but use
                             # correct registers
```

Caller Example

```
n = sumOf(17,42)
```

```
movq $42,%rsi                  # load arguments in
movq $17,%rdi                  # either order, but use
                               # correct registers
call sumOf                     # jump & push ret addr
```

Caller Example

```
n = sumOf(17,42)
```

```
movq $42,%rsi                    # load arguments in
movq $17,%rdi                    # either order, but use
                                 # correct registers
call sumOf                       # jump & push ret addr
movq %rax,offset_n(%rbp)         # store result
```

Example Function Body

```
int sumOf(int x, int y) {
   int a, int b;
   a = x;
   b = a + y;
   return b;
}
```

Example Function Body

```
int sumOf(int x, int y) {
    int a, int b;
    a = x;
    b = a + y;
    return b;
}
```

```
sumOf:
    pushq %rbp    # prologue
    movq %rsp,%rbp
```

Example Function Body

```
int sumOf(int x, int y) {
    int a, int b;
    a = x;
    b = a + y;
    return b;
}
```

```
sumOf:
    pushq %rbp    # prologue
    movq %rsp,%rbp
    subq $16,%rsp
```

Example Function Body

```
int sumOf(int x, int y) {
    int a, int b;
    a = x;
    b = a + y;
    return b;
}
```

```
sumOf:
    pushq %rbp    # prologue
    movq %rsp,%rbp
    subq $16,%rsp
    movq %rdi,-8(%rbp)
```

Example Function Body

```
int sumOf(int x, int y) {
    int a, int b;
    a = x;
    b = a + y;
    return b;
}
```

```
sumOf:
    pushq %rbp    # prologue
    movq %rsp,%rbp
    subq $16,%rsp
    movq %rdi,-8(%rbp)
    movq -8(%rbp),%rax
    addq %rsi,%rax
    movq %rax,-16(%rbp)
```

Example Function Body

```
int sumOf(int x, int y) {
    int a, int b;
    a = x;
    b = a + y;
    return b;
}
```

```
sumOf:
    pushq %rbp    # prologue
    movq %rsp,%rbp
    subq $16,%rsp
    movq %rdi,-8(%rbp)
    movq -8(%rbp),%rax
    addq %rsi,%rax
    movq %rax,-16(%rbp)
    movq -16(%rbp),%rax
    movq %rbp,%rsp
    popq %rbp
```

Example Stack Frame for `sumOf`

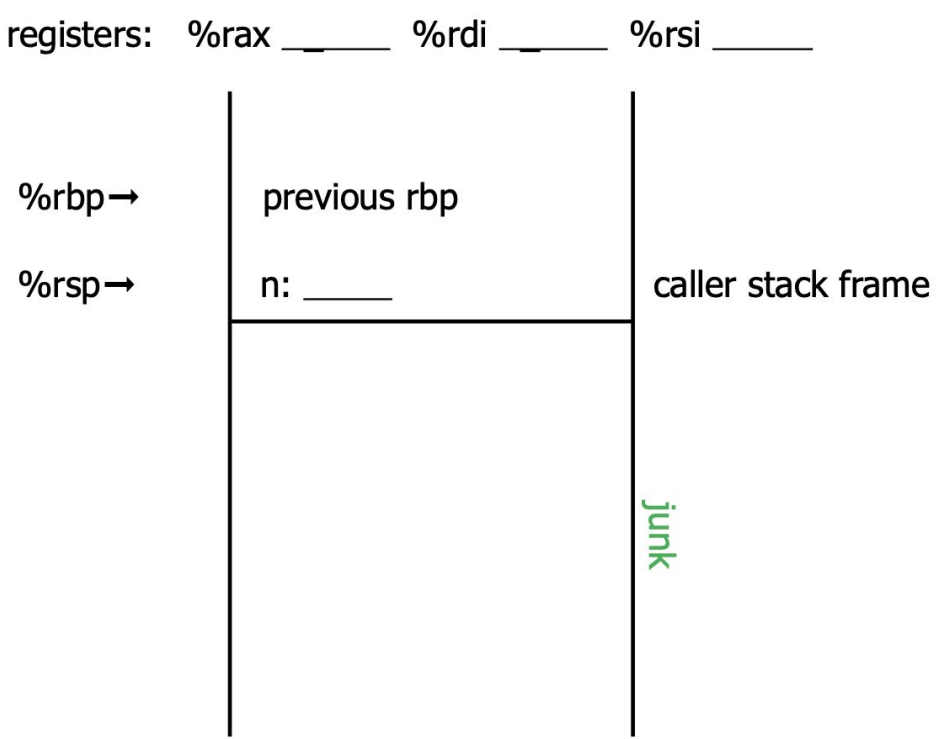
```
int sumOf(int x, int y) {
    int a, int b;
    a = x;
    b = a + y;
    return b;
}
```

Example Stack Frame for `sumOf`

```
int sumOf(int x, int y) {
    int a, int b;
    a = x;
    b = a + y;
    return b;
}

sumOf:
# prologue
  pushq  %rbp
  movq   %rsp,%rbp
  subq   $16,%rsp
#  a = x;
  movq   %rdi,-8(%rbp)
#  b = a + y;
  movq   -8(%rbp),%rax
  addq   %rsi,%rax
  movq   %rax,-16(%rbp)
#  return b;
  movq   -16(%rbp),%rax
  movq   %rbp,%rsp
  popq   %rbp
  ret
# }
```

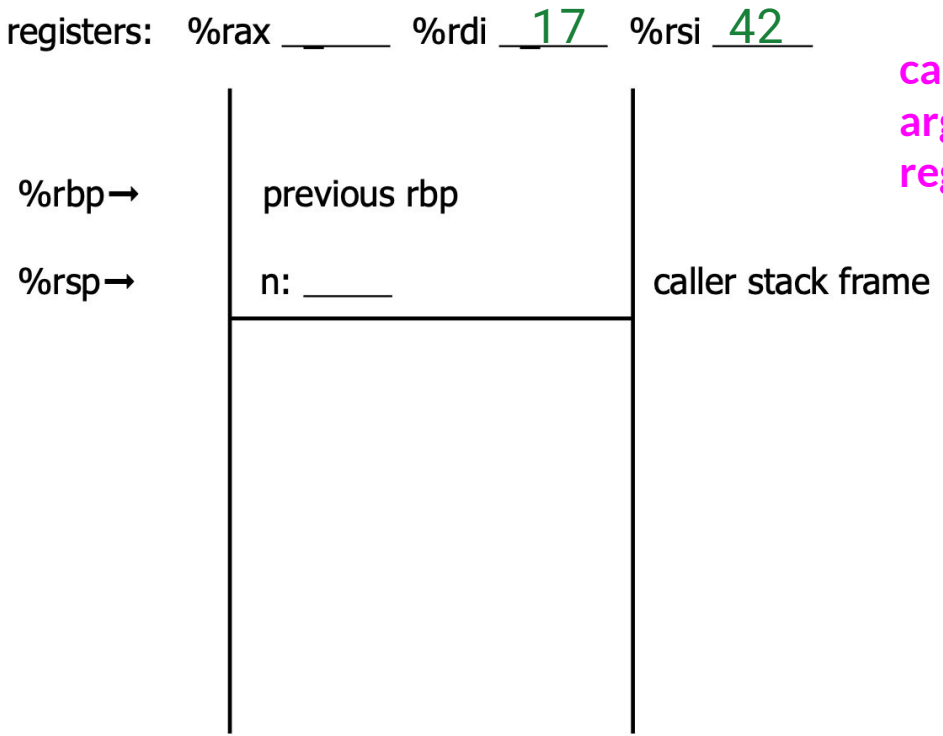
Example Stack Frame for `sumOf`




```
int sumOf(int x, int y) {
    int a, int b;
    a = x;
    b = a + y;
    return b;
}

sumOf:
# prologue
  pushq  %rbp
  movq   %rsp,%rbp
  subq   $16,%rsp
#  a = x;
  movq   %rdi,-8(%rbp)
#  b = a + y;
  movq   -8(%rbp),%rax
  addq   %rsi,%rax
  movq   %rax,-16(%rbp)
#  return b;
  movq   -16(%rbp),%rax
  movq   %rbp,%rsp
  popq   %rbp
  ret
# }
```

Example Stack Frame for `sumOf`



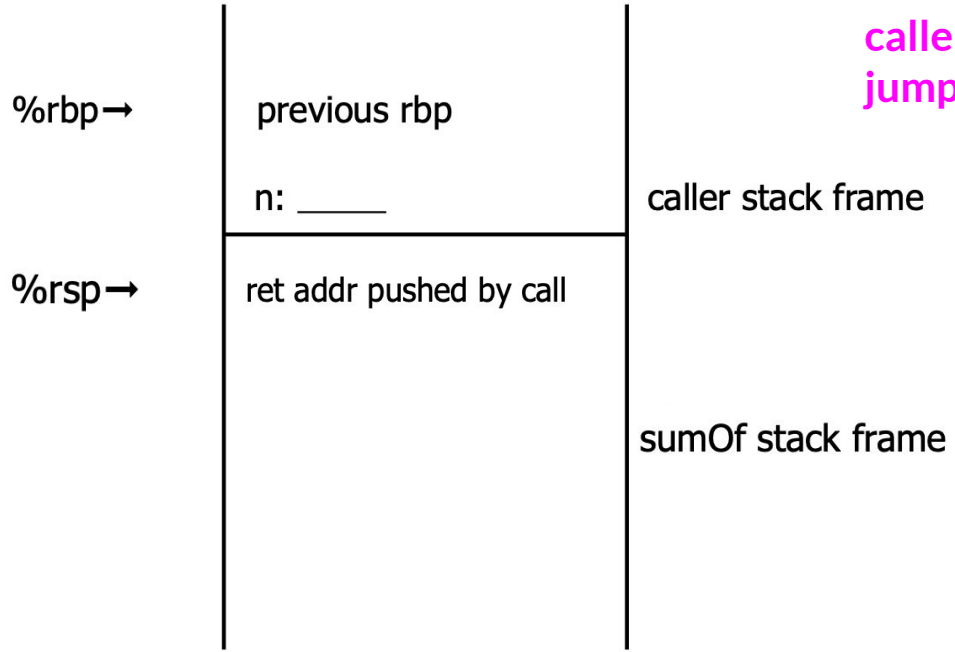
caller loads argument registers →

```
int sumOf(int x, int y) {
    int a, int b;
    a = x;
    b = a + y;
    return b;
}

sumOf:
# prologue
  pushq  %rbp
  movq   %rsp,%rbp
  subq   $16,%rsp
#  a = x;
  movq   %rdi,-8(%rbp)
#  b = a + y;
  movq   -8(%rbp),%rax
  addq   %rsi,%rax
  movq   %rax,-16(%rbp)
#  return b;
  movq   -16(%rbp),%rax
  movq   %rbp,%rsp
  popq   %rbp
  ret
# }
```

Example Stack Frame for `sumOf`

registers: %rax _____ %rdi 17 %rsi 42

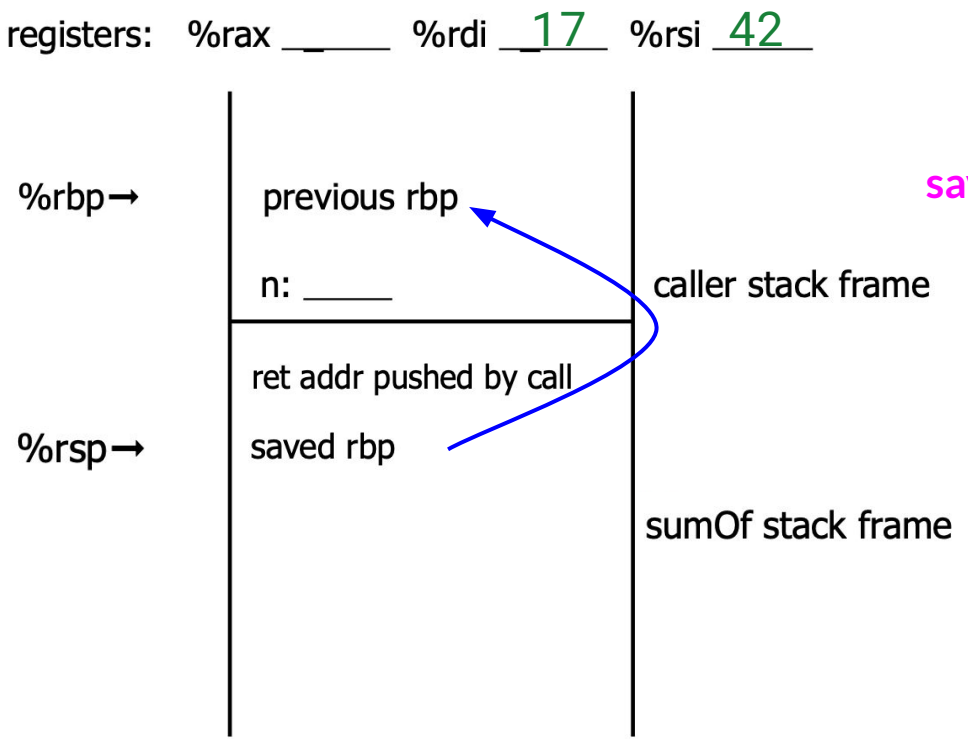


caller calls, jump here

```
int sumOf(int x, int y) {
    int a, int b;
    a = x;
    b = a + y;
    return b;
}

sumOf:
# prologue
  pushq  %rbp
  movq   %rsp,%rbp
  subq   $16,%rsp
#  a = x;
  movq   %rdi,-8(%rbp)
#  b = a + y;
  movq   -8(%rbp),%rax
  addq   %rsi,%rax
  movq   %rax,-16(%rbp)
#  return b;
  movq   -16(%rbp),%rax
  movq   %rbp,%rsp
  popq   %rbp
  ret
# }
```

Example Stack Frame for `sumOf`



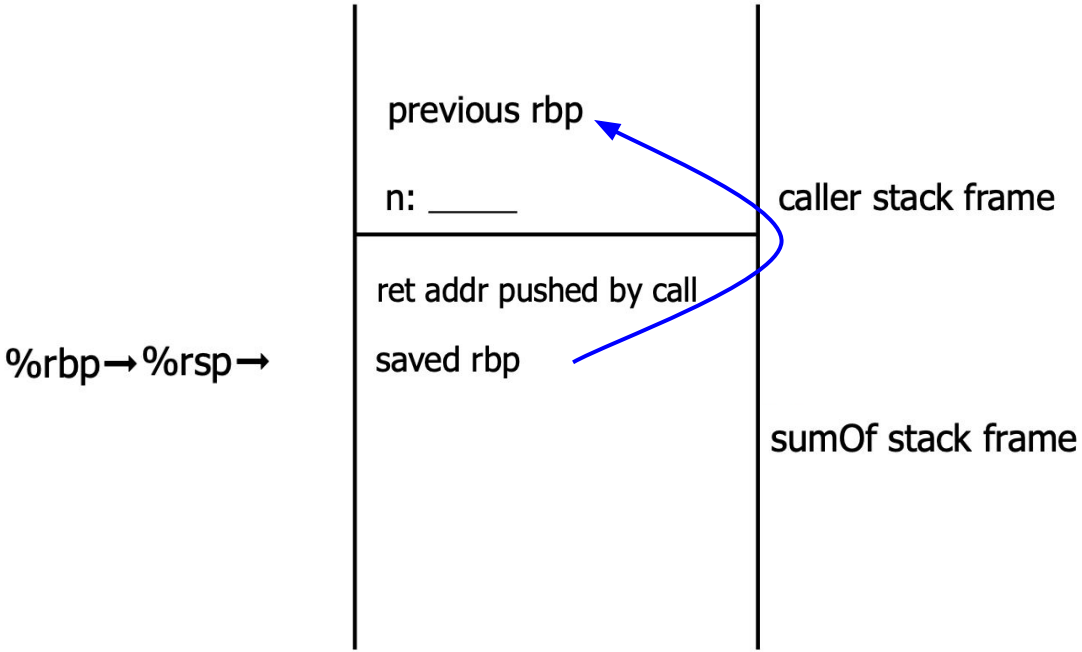
save %rbp →

```
int sumOf(int x, int y) {
    int a, int b;
    a = x;
    b = a + y;
    return b;
}

sumOf:
# prologue
  pushq  %rbp
  movq   %rsp,%rbp
  subq   $16,%rsp
#  a = x;
  movq   %rdi,-8(%rbp)
#  b = a + y;
  movq   -8(%rbp),%rax
  addq   %rsi,%rax
  movq   %rax,-16(%rbp)
#  return b;
  movq   -16(%rbp),%rax
  movq   %rbp,%rsp
  popq   %rbp
  ret
# }
```

Example Stack Frame for `sumOf`

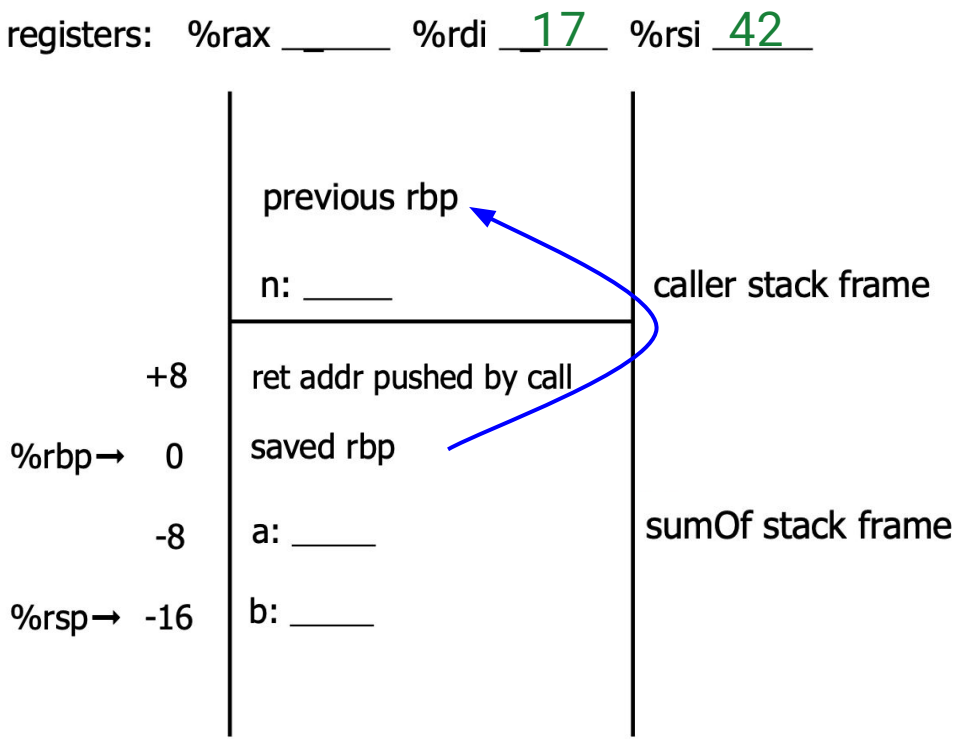
registers: %rax _____ %rdi 17 %rsi 42




```
int sumOf(int x, int y) {
    int a, int b;
    a = x;
    b = a + y;
    return b;
}

sumOf:
# prologue
  pushq  %rbp
  movq   %rsp,%rbp
  subq   $16,%rsp
#  a = x;
  movq   %rdi,-8(%rbp)
#  b = a + y;
  movq   -8(%rbp),%rax
  addq   %rsi,%rax
  movq   %rax,-16(%rbp)
#  return b;
  movq   -16(%rbp),%rax
  movq   %rbp,%rsp
  popq   %rbp
  ret
# }
```

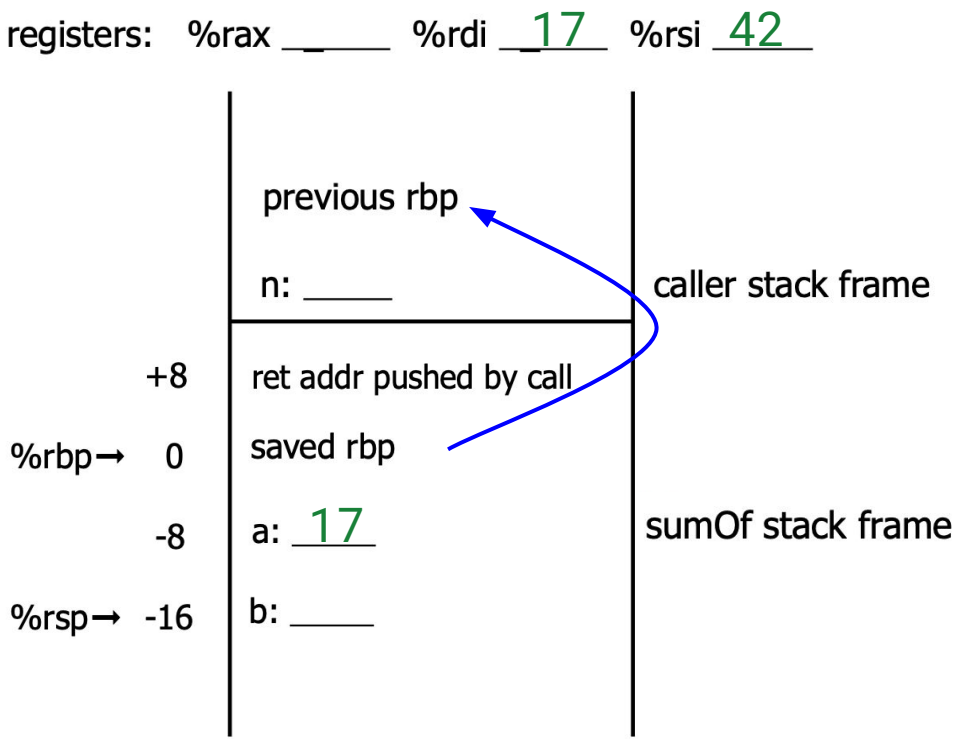
Example Stack Frame for `sumOf`




```
int sumOf(int x, int y) {
    int a, int b;
    a = x;
    b = a + y;
    return b;
}

sumOf:
# prologue
  pushq  %rbp
  movq   %rsp,%rbp
  subq   $16,%rsp
#  a = x;
  movq   %rdi,-8(%rbp)
#  b = a + y;
  movq   -8(%rbp),%rax
  addq   %rsi,%rax
  movq   %rax,-16(%rbp)
#  return b;
  movq   -16(%rbp),%rax
  movq   %rbp,%rsp
  popq   %rbp
  ret
# }
```

Example Stack Frame for `sumOf`

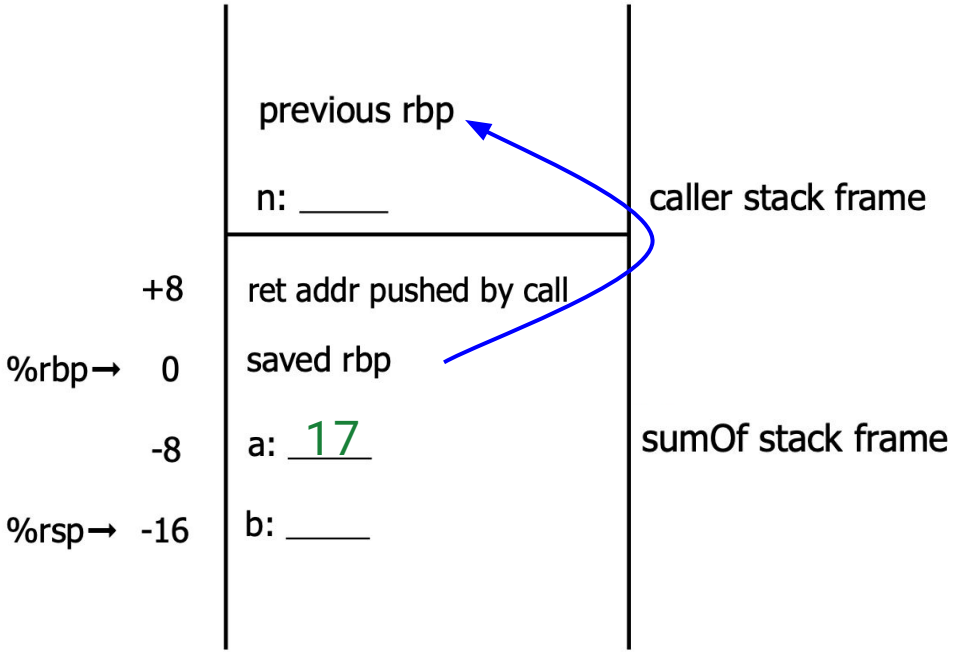



```
int sumOf(int x, int y) {
    int a, int b;
    a = x;
    b = a + y;
    return b;
}

sumOf:
# prologue
  pushq  %rbp
  movq   %rsp,%rbp
  subq   $16,%rsp
#  a = x;
  movq   %rdi,-8(%rbp)
#  b = a + y;
  movq   -8(%rbp),%rax
  addq   %rsi,%rax
  movq   %rax,-16(%rbp)
#  return b;
  movq   -16(%rbp),%rax
  movq   %rbp,%rsp
  popq   %rbp
  ret
# }
```

Example Stack Frame for `sumOf`

registers: %rax 17 %rdi 17 %rsi 42

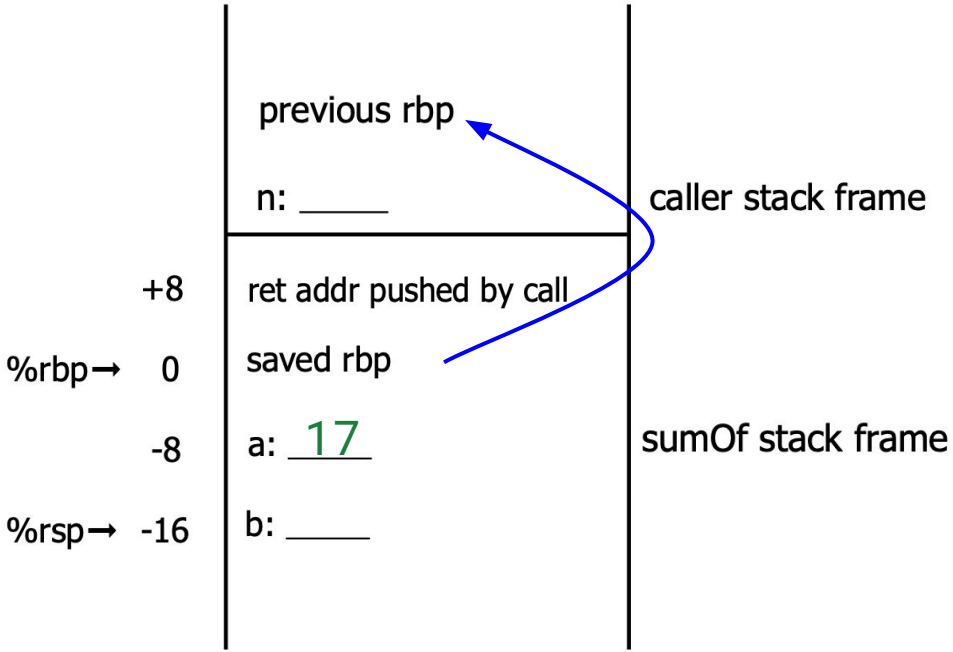



```
int sumOf(int x, int y) {
    int a, int b;
    a = x;
    b = a + y;
    return b;
}

sumOf:
# prologue
  pushq  %rbp
  movq   %rsp,%rbp
  subq   $16,%rsp
#  a = x;
  movq   %rdi,-8(%rbp)
#  b = a + y;
  movq   -8(%rbp),%rax
  addq   %rsi,%rax
  movq   %rax,-16(%rbp)
#  return b;
  movq   -16(%rbp),%rax
  movq   %rbp,%rsp
  popq   %rbp
  ret
# }
```

Example Stack Frame for `sumOf`

registers: %rax ~~17~~ 59 %rdi 17 %rsi 42

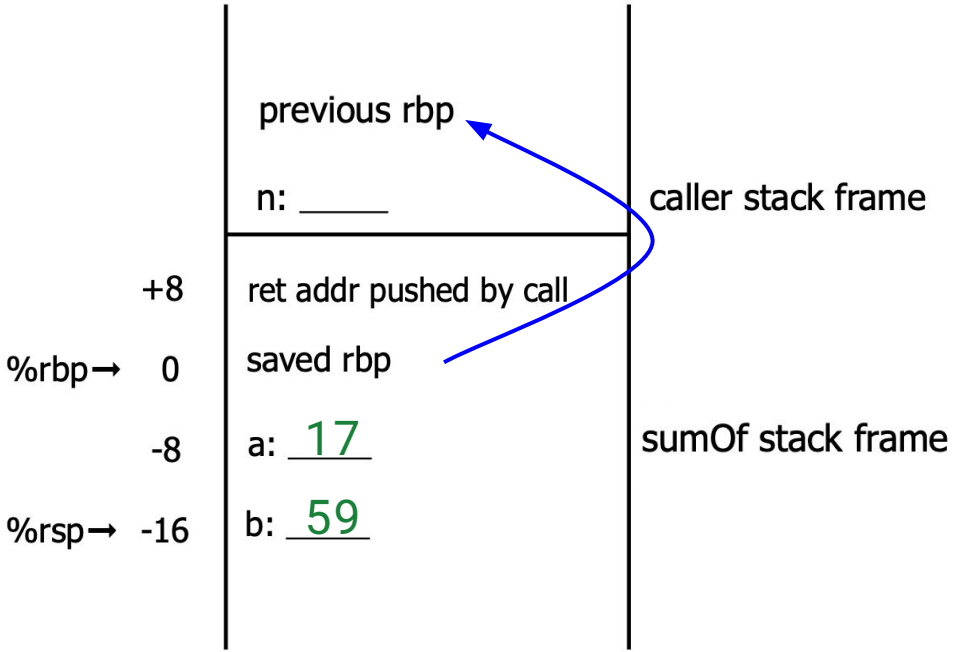



```
int sumOf(int x, int y) {
    int a, int b;
    a = x;
    b = a + y;
    return b;
}

sumOf:
# prologue
  pushq  %rbp
  movq   %rsp,%rbp
  subq   $16,%rsp
#  a = x;
  movq   %rdi,-8(%rbp)
#  b = a + y;
  movq   -8(%rbp),%rax
  addq   %rsi,%rax
  movq   %rax,-16(%rbp)
#  return b;
  movq   -16(%rbp),%rax
  movq   %rbp,%rsp
  popq   %rbp
  ret
# }
```

Example Stack Frame for `sumOf`

registers: %rax ~~17~~ 59 %rdi 17 %rsi 42

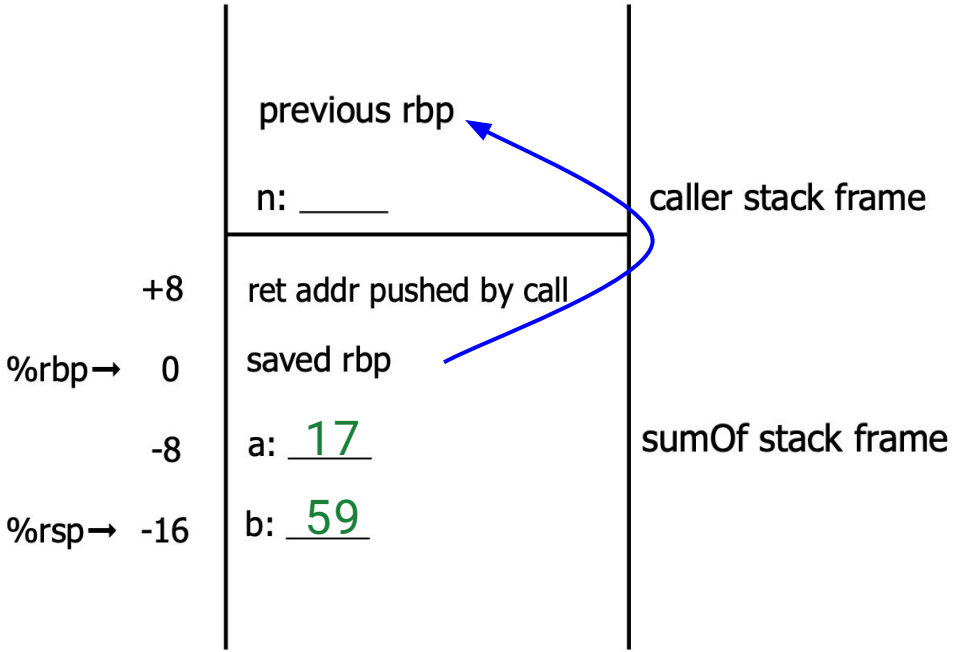



```
int sumOf(int x, int y) {
    int a, int b;
    a = x;
    b = a + y;
    return b;
}

sumOf:
# prologue
  pushq  %rbp
  movq   %rsp,%rbp
  subq   $16,%rsp
#  a = x;
  movq   %rdi,-8(%rbp)
#  b = a + y;
  movq   -8(%rbp),%rax
  addq   %rsi,%rax
  movq   %rax,-16(%rbp)
#  return b;
  movq   -16(%rbp),%rax
  movq   %rbp,%rsp
  popq   %rbp
  ret
# }
```

Example Stack Frame for `sumOf`

registers: %rax 59 %rdi 17 %rsi 42

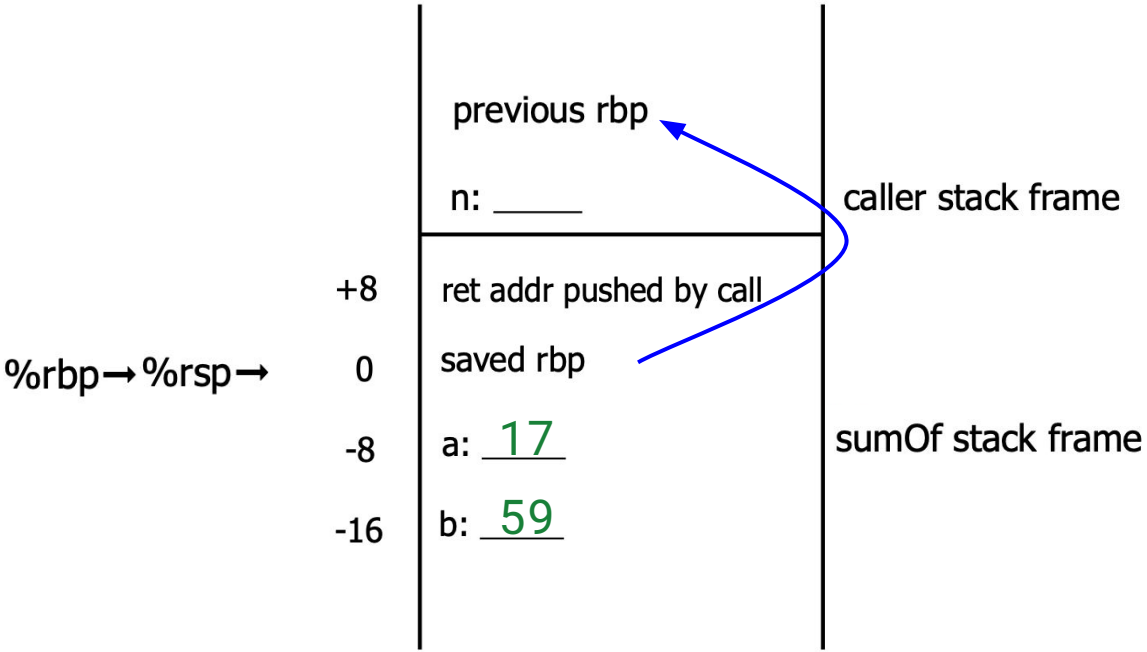



```
int sumOf(int x, int y) {
    int a, int b;
    a = x;
    b = a + y;
    return b;
}

sumOf:
# prologue
  pushq  %rbp
  movq   %rsp,%rbp
  subq   $16,%rsp
#  a = x;
  movq   %rdi,-8(%rbp)
#  b = a + y;
  movq   -8(%rbp),%rax
  addq   %rsi,%rax
  movq   %rax,-16(%rbp)
#  return b;
  movq   -16(%rbp),%rax
  movq   %rbp,%rsp
  popq   %rbp
  ret
# }
```

Example Stack Frame for `sumOf`

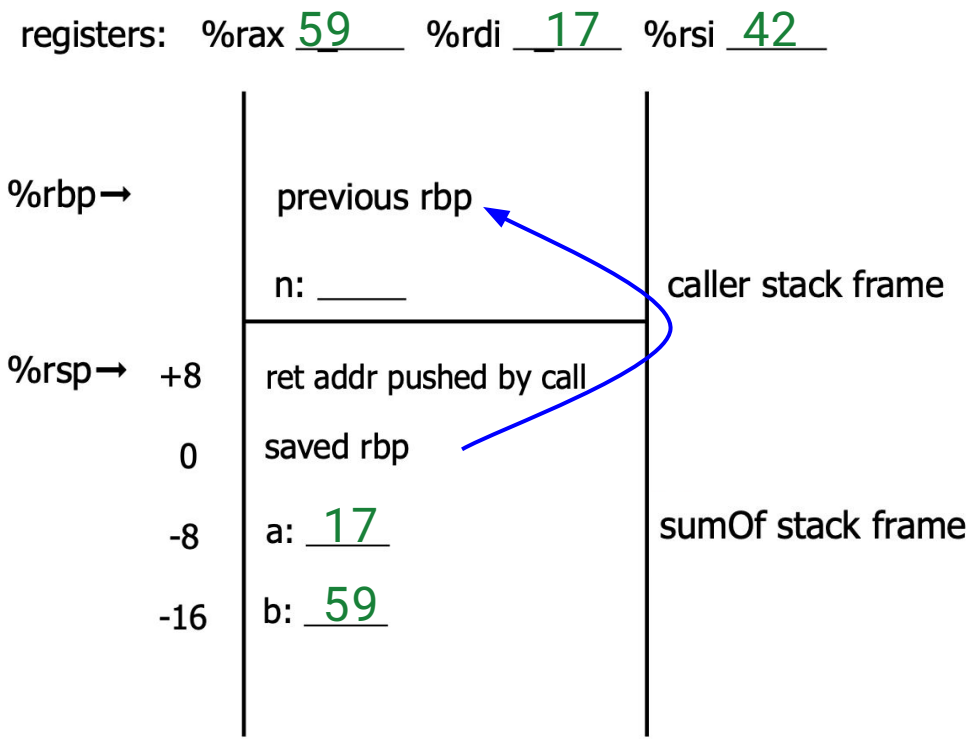
registers: %rax 59 %rdi 17 %rsi 42




```
int sumOf(int x, int y) {
    int a, int b;
    a = x;
    b = a + y;
    return b;
}

sumOf:
# prologue
  pushq  %rbp
  movq   %rsp,%rbp
  subq   $16,%rsp
#  a = x;
  movq   %rdi,-8(%rbp)
#  b = a + y;
  movq   -8(%rbp),%rax
  addq   %rsi,%rax
  movq   %rax,-16(%rbp)
#  return b;
  movq   -16(%rbp),%rax
  movq   %rbp,%rsp
  popq   %rbp
  ret
# }
```

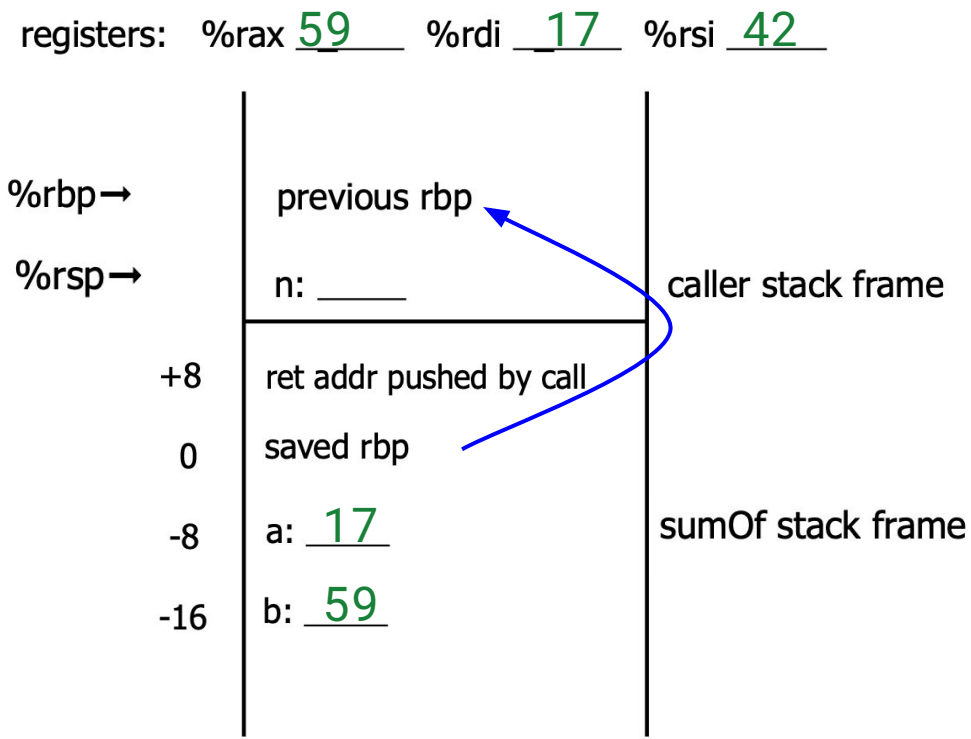
Example Stack Frame for `sumOf`




```
int sumOf(int x, int y) {
    int a, int b;
    a = x;
    b = a + y;
    return b;
}

sumOf:
# prologue
  pushq  %rbp
  movq   %rsp,%rbp
  subq   $16,%rsp
#  a = x;
  movq   %rdi,-8(%rbp)
#  b = a + y;
  movq   -8(%rbp),%rax
  addq   %rsi,%rax
  movq   %rax,-16(%rbp)
#  return b;
  movq   -16(%rbp),%rax
  movq   %rbp,%rsp
  popq   %rbp
  ret
# }
```

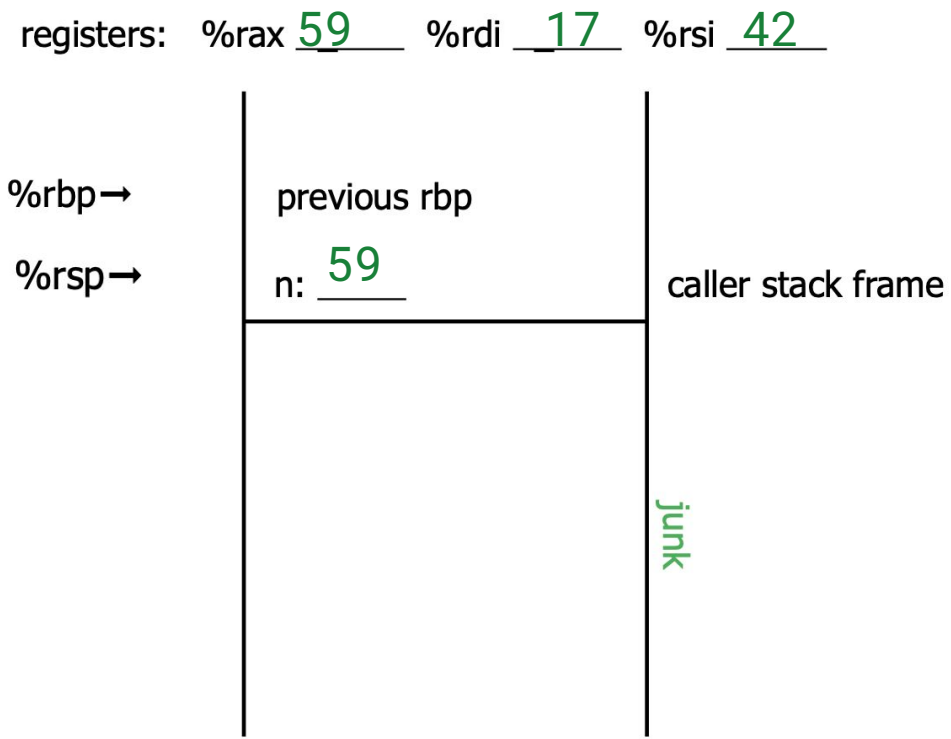
Example Stack Frame for `sumOf`




```
int sumOf(int x, int y) {
    int a, int b;
    a = x;
    b = a + y;
    return b;
}

sumOf:
# prologue
  pushq  %rbp
  movq   %rsp,%rbp
  subq   $16,%rsp
#  a = x;
  movq   %rdi,-8(%rbp)
#  b = a + y;
  movq   -8(%rbp),%rax
  addq   %rsi,%rax
  movq   %rax,-16(%rbp)
#  return b;
  movq   -16(%rbp),%rax
  movq   %rbp,%rsp
  popq   %rbp
  ret
# }
```

Example Stack Frame for `sumOf`



(caller sets n to %rax)

```
int sumOf(int x, int y) {
    int a, int b;
    a = x;
    b = a + y;
    return b;
}
```

```
sumOf:
# prologue
  pushq  %rbp
  movq   %rsp,%rbp
  subq   $16,%rsp
#  a = x;
  movq   %rdi,-8(%rbp)
#  b = a + y;
  movq   -8(%rbp),%rax
  addq   %rsi,%rax
  movq   %rax,-16(%rbp)
#  return b;
  movq   -16(%rbp),%rax
  movq   %rbp,%rsp
  popq   %rbp
  ret
# }
```

The Nice Thing About Standards…

The Nice Thing About Standards…

- The convention I’ve just shown is the System V/AMD64 ABI convention (used by Linux, MacOS X)

The Nice Thing About Standards…

- The convention I've just shown is the System V/AMD64 ABI convention (used by Linux, MacOS X)
- Microsoft's calling conventions are slightly different (of course)

The Nice Thing About Standards…

- The convention I've just shown is the System V/AMD64 ABI convention (used by Linux, MacOS X)
- Microsoft's calling conventions are slightly different (of course)
  - First four parameters in registers %rcx, %rdx, %r8, %r9; rest on the stack

The Nice Thing About Standards…

- The convention I've just shown is the System V/AMD64 ABI convention (used by Linux, MacOS X)
- Microsoft's calling conventions are slightly different (of course)
  - First four parameters in registers %rcx, %rdx, %r8, %r9; rest on the stack
  - Called function stack frame must include empty space for called function to save values passed in parameter registers if desired

The Nice Thing About Standards…

- The convention I've just shown is the System V/AMD64 ABI convention (used by Linux, MacOS X)
- Microsoft's calling conventions are slightly different (of course)
  - First four parameters in registers %rcx, %rdx, %r8, %r9; rest on the stack
  - Called function stack frame must include empty space for called function to save values passed in parameter registers if desired
- Not relevant for us, but worth being aware of it
  - (except that providing space in each stack frame to save parameter registers will be handy for our simple code gen)

Course Announcements

- the PA3c1 (codegen testing) deadline has already passed
  - if you forgot about it, we will still accept submissions (with a penalty)
- PA3c2 (TAC) is due later this week (“before spring break”)
  - you should already have started, or you are behind
  - if there is demand from the class, I will consider a short extension on this assignment to e.g., Monday
- I have become aware of a **bug in the reference compiler**’s x86-64 module; a fix will be forthcoming. For now, don’t trust it.
- Don’t forget there is a **midterm** in this class the week after spring break!