# An introduction to x86-64 assembly language programming from a compiler writer's perspective

## What is x86-64?

- We'll be using the "x86-64" architecture as our main target:
  - A "64-bit" instruction set
  - First introduced by AMD in around 2000 as an extension of Intel's 32-bit "IA-32" or "x86" instruction set
  - Also known as "AMD64", "Intel64", "x86_64", ...
  - Broadly adopted by:
    - processors from Intel, AMD, Via, ...
    - laptops, desktops, servers, gaming consoles, ...
    - Linux, Mac OS X, Windows, ...

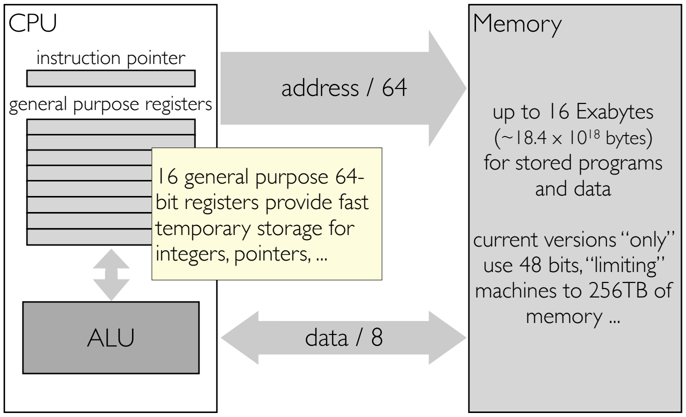
## Other architectures:

- Not to be confused with:
  - IA64: a completely different 64-bit Intel architecture (Itanium)
  - ARM: widely used in phones, tablets, and more
  - IBM Power: used in Xbox 360, PS3, Wii, servers, and more
  - SPARC: used by some of the college's Unix servers
- You won't be able to run x86-64 code directly on a computer that uses one of these alternative instruction sets!

## Notes

- No prior or in-depth knowledge of x86-64 programming will be assumed
- We will only use a small subset of the full instruction set
- If you're looking to become an expert on x86-64 programming, you'll want to look for another class!
- We'll be using the *AT&T syntax* for x86-64 assembly language rather than the *Intel syntax*. This is the default syntax used by the free GNU tools in Linux, MacOS, and DJGPP or Cygwin on Windows, and others
- For simplicity, I recommend: ssh yourid@linuxlab.cs.pdx.edu (or, on Windows, the equivalent using PuTTY)

## A (greatly) simplified view of the x86-64



CPU

instruction pointer

general purpose registers

16 general purpose 64-bit registers provide fast temporary storage for integers, pointers, ...

ALU

address / 64

data / 8

Memory

up to 16 Exabytes (~18.4 × $10^{18}$ bytes) for stored programs and data

current versions "only" use 48 bits, "limiting" machines to 256TB of memory ...

## Programming for x86-64

- In concrete terms, an x86-64 program is just a collection of byte values (*machine code*)
- Once it has been loaded in to memory, the processor can *execute* a program by interpreting the byte values as *instructions* for the processor to act on
- For practical purposes, we will usually write x86-64 programs in a textual format called *assembly language* that is easier to read than the raw byte values
- A compiler that translates assembly language programs in to machine code is called an *assembler*

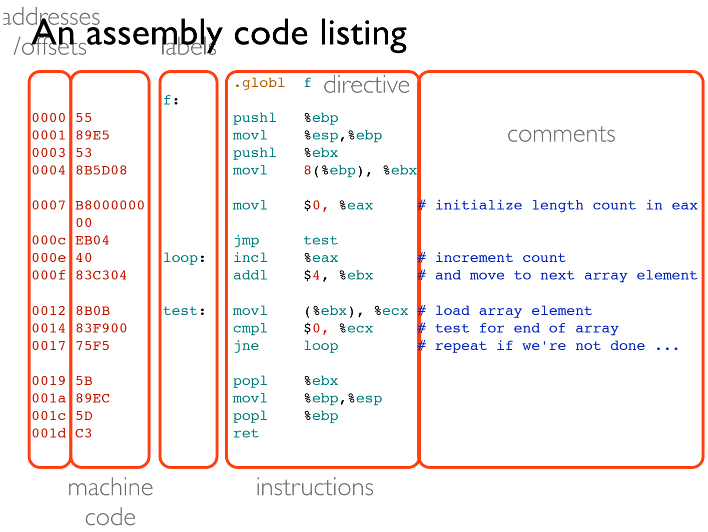
# An assembly code listing

```
                  .globl  f
          f:
0000 55           pushl   %ebp
0001 89E5         movl    %esp,%ebp
0003 53           pushl   %ebx
0004 8B5D08       movl    8(%ebp), %ebx

0007 B8000000     movl    $0, %eax      # initialize length count in eax
     00
000c EB04         jmp     test
000e 40   loop:   incl    %eax          # increment count
000f 83C304       addl    $4, %ebx      # and move to next array element

0012 8B0B test:   movl    (%ebx), %ecx  # load array element
0014 83F900       cmpl    $0, %ecx      # test for end of array
0017 75F5         jne     loop          # repeat if we're not done ...

0019 5B           popl    %ebx
001a 89EC         movl    %ebp,%esp
001c 5D           popl    %ebp
001d C3           ret
```

Machine code

Assembly code

# An assembly code listing



```
                  .globl  f
          f:
0000 55           pushl   %ebp
0001 89E5         movl    %esp,%ebp
0003 53           pushl   %ebx
0004 8B5D08       movl    8(%ebp), %ebx

0007 B8000000     movl    $0, %eax      # initialize length count in eax
     00
000c EB04         jmp     test
000e 40   loop:   incl    %eax          # increment count
000f 83C304       addl    $4, %ebx      # and move to next array element

0012 8B0B test:   movl    (%ebx), %ecx  # load array element
0014 83F900       cmpl    $0, %ecx      # test for end of array
0017 75F5         jne     loop          # repeat if we're not done ...

0019 5B           popl    %ebx
001a 89EC         movl    %ebp,%esp
001c 5D           popl    %ebp
001d C3           ret
```

# x86-64 registers

# 8-bit registers (holding a single byte, 0-255)

| | high | low |
|---|---|---|
| accumulator | ah | al |
| base | bh | bl |
| count | ch | cl |
| data | dh | dl |

Introduced in 1978 as part of the 8086 architecture

# 16-bit registers ("word")

| | | | |
|---|---|---|---|
| accumulator | ax | ah | al |
| base | bx | bh | bl |
| count | cx | ch | cl |
| data | dx | dh | dl |
| source index | si | | |
| destination index | di | | |
| base pointer | bp | | |
| stack pointer | sp | | |

Introduced in 1978 as part of the 8086 architecture

# 32-bit registers ("double word")

| | | | | |
|---|---|---|---|---|
| accumulator | eax | ax | ah | al |
| base | ebx | bx | bh | bl |
| count | ecx | cx | ch | cl |
| data | edx | dx | dh | dl |
| source index | esi | si | | |
| destination index | edi | di | | |
| base pointer | ebp | bp | | |
| stack pointer | esp | sp | | |

"e" for extended

sometimes referred to as "long word"s

Introduced in 1985 as part of the 80386 architecture

## 64-bit registers (“quad word”)

| | | | | |
|---|---|---|---|---|
| rax | eax | ax | ah | al |
| rbx | ebx | bx | bh | bl |
| rcx | ecx | cx | ch | cl |
| rdx | edx | dx | dh | dl |
| rsi | esi | si | | |
| rdi | edi | di | | |
| rbp | ebp | bp | | |
| rsp | esp | sp | | |
| r8 | | | | |
| r9 | | | | |
| r10 | | | | |
| r11 | | | | |
| r12 | | | | |
| r13 | | | | |
| r14 | | | | |
| r15 | | | | |

“r” for register

Introduced in 2000 as part of the x86-64 architecture

## 64-bit registers (“quad word”)

| | | | | |
|---|---|---|---|---|
| rax | eax | ax | ah | al |
| rbx | ebx | bx | bh | bl |
| rcx | ecx | cx | ch | cl |
| rdx | edx | dx | dh | dl |
| rsi | esi | si | | sil |
| rdi | edi | di | | dil |
| rbp | ebp | bp | | bpl |
| rsp | esp | sp | | spl |
| r8 | r8d | r8w | | r8b |
| r9 | r9d | r9w | | r9b |
| r10 | r10d | r10w | | r10b |
| r11 | r11d | r11w | | r11b |
| r12 | r12d | r12w | | r12b |
| r13 | r13d | r13w | | r13b |
| r14 | r14d | r14w | | r14b |
| r15 | r15d | r15w | | r15b |

## The 32-bit and 64-bit registers

| | |
|---|---|
| rax | eax |
| rbx | ebx |
| rcx | ecx |
| rdx | edx |
| rsi | esi |
| rdi | edi |
| rbp | ebp |
| rsp | esp |
| r8 | |
| r9 | |
| r10 | |
| r11 | |
| r12 | |
| r13 | |
| r14 | |
| r15 | |

We'll mostly just be using 64-bit and 32-bit registers

- 32-bit registers for C-style `int` values
- 64-bit registers for addresses and C-style `long` values

## Special vs. general purpose registers

- `rip`: the instruction pointer register
- `rsp`: the stack pointer register
- `eflags`: the flags register, stores information about the results of the most recent arithmetic or logic instruction
- Other registers can typically be used for any purpose (although some instructions—division, for example—work only with specific registers)

# x86-64 instructions

## Instruction format

- A typical x86-64 instruction has the form:

```
opcode src, dst
```

what to do — input source — result destination

- A suffix on the opcode indicates the size of the data that is being operated on:
  - 64-bit values use the suffix **q**(uad word)
  - 32-bit values use the suffix **l**(ong)
  - 16-bit values use the suffix **w**(ord)
  - 8-bit values use the suffix **b**(yte)

## Addressing modes

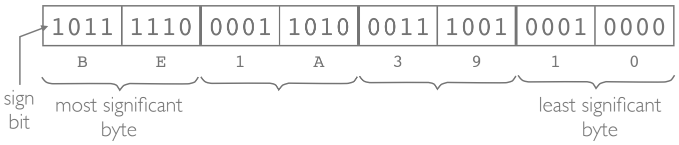
- **Register access**, reg:
  - `%rax`: the value in register `rax`
  - Can typically use any registers except `rip` and `eflags`
- **Memory access**, mem:
  - `var`: the value in the memory location at address var
  - `(%rax)`: the value in memory at the address in `rax`
  - `8(%rax)`: the value in memory at the address given by adding 8 to the value in `rax`
- **Immediate**, immed:
  - `$42`: the constant value 42 (decimal; use `$0x2A` for hex)
  - `$var`: the address of memory location var

## Directives for "declaring" variables

```
        .data          # put variables in the "data" section
                       # (code usually goes in .text)

        .align  4      # make sure address is multiple of 4
myvar:  .long   42     # Simple variable, initialized to 42


        .global days   # A globally accessible array of ints
days:   .long   31, 28, 31, 30, 30, 30
        .long   31, 31, 30, 31, 30, 31


scratch:.space  4*100  # reserve uninitialized space

big:    .quad   123    # a 64-bit integer (takes 8 bytes)
medium: .long   123    # a 32-bit integer (takes 4 bytes)
regular:.short  123    # a 16-bit integer (takes 2 bytes)
small:  .byte   123    # an 8-bit integer (takes 1 byte)
```

## How values are stored in memory

- A double word holds 32 binary digits ("bits") (i.e., 4 bytes)



- 0xBE1A3910 can be interpreted as -1,105,577,712 (signed) or 3,189,389,584 (unsigned)
- Stored in memory with the least significant byte at the lowest address ("little endian"):

| stored byte | 0x10 | 0x39 | 0x1A | 0xBE |
|---|---|---|---|---|
| address | 400 | 401 | 402 | 403 |

## x86-64 instructions: data movement

## Move instructions

- Copy data from a source to a destination (where `X` is one of the size suffixes: b,w,`l`,q):

  `movX src, dst`

- Any of the following combinations of arguments is allowed:

  `movX` reg, (reg | mem)

  `movX` mem, reg

  `movX` immed, (reg | mem)

- Note that you can't move mem to mem in one instruction

## Examples

Suppose that the memory (starting at address 0) contains the following (eight byte) values:

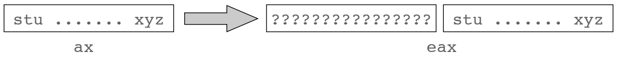
| 8 | 6 | 2 | 8 | 0 | 2 | 4 | 1 | 7 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 8 | 16 | 24 | 32 | 40 | 48 | 56 | 64 | 72 | 80 | 88 | 96 |

Then

| instruction | contents of `rax` |
|---|---|
| `movq $24, %rax` | 24 |
| `movq (%rax), %rax` | 8 |
| `movq 24(%rax), %rax` | 0 |

# Zero and sign-extension

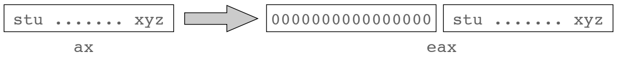
- Suppose we want to copy a value from a 16-bit register in to a 32-bit register



stu ....... xyz → ???????????????? stu ....... xyz

ax eax

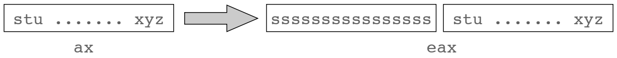
- Two common strategies:
  - Zero extension: for unsigned values



stu ....... xyz → 0000000000000000 stu ....... xyz

ax eax

  - Sign extension: for signed values



stu ....... xyz → ssssssssssssssss stu ....... xyz

ax eax

# Move with sign extension

- Copy from source to larger destination with sign extension:

```
movsFT src, dst
```

- `F` and `T` are the "from" and "to" sizes (either `b`, `w`, `l`, or `q`)
- Valid combinations: `bw`, `bl`, `bq`, `wl`, `wq`, or `lq`
- Examples:

```
movsbq %al, %rdi    # byte to quad
movswl %ax, %eax    # word to long
movslq %eax, %rdx   # long to quad
```

# Move with zero extension

- Copy from source to larger destination with zero extension:

```
movzFT src, dst
```

- Here, `FT` is one of the combinations: `bw`, `bl`, `bq`, `wl`, or `wq`
- There is no case for `lq`:
  - In x86-64, every instruction with a 32-bit register destination automatically zero extends the result to fill the associated 64-bit register
  - For example: `movl %eax,%ebx` sets
    - the lower 32 bits of `rbx` to the value from `eax`
    - the upper 32 bits of `rbx` to zero

# Scaled indexed addressing

- [base]`(`[reg$_1$]`,`reg$_2$ [`,`index]`)`

  a memory operand whose address is the value in reg$_1$, *plus* the specified base constant, *plus* the value of reg$_2$ *times* the index (which must be 1, 2, 4, or 8)

- Any of the parts in [...] can be omitted
- Examples:

| | |
|---|---|
| `(rax,rbx,4)` | the **`rbx`**th element in the array of 32-bit words starting at the address in **`rax`** |
| `days(,rbx,4)` | the **`rbx`**th element in the array of 32-bit words starting at the address **`days`** |

# More examples

Suppose that the memory (starting at address 0) contains the following (eight byte) values:

| | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 8 | 6 | 2 | 8 | 0 | 2 | 4 | 1 | 7 | 3 | 4 | 5 | 6 |
| 0 | 8 | 16 | 24 | 32 | 40 | 48 | 56 | 64 | 72 | 80 | 88 | 96 |

Then

| instruction | rax | rbx |
|---|---|---|
| `movq $24, %rax` ◀ | 24 | |
| `movq 16(%rax), %rbx` | 24 | 2 |
| `movq 32(%rax,%rbx,4), %rax` | 7 | 2 |

# The `lea` (load effective address) instruction

- Load the address of the source operand (must be memory) to a destination (where `X` is one of the size suffixes: `b`,`w`,`l`,`q`):

```
leaX src, dst
```

- Can also be used to co-opt the addressing mode circuitry into performing arithmetic operations:

```
leaq 4(%rax),%rax         # rax += 4
leal 1(%eax,%eax,2),%eax  # eax = 3*eax + 1
leaq 1(%rax,%rax), %rax   # rax = 2*rax + 1
leal 4(,%eax,8), %eax     # eax = 8*eax + 4
```

- These instructions just do an address calculation and do not attempt to read the data at that address.

## The exchange instruction

- Exchange data between two locations

```
xchgX  (reg | mem), reg
```

- Consider the following instructions in a high-level language:

```
int tmp = x;
x       = y;
y       = tmp;
```

- If x and y are held in registers, then a “clever enough” compiler can translate this code into a single xchgl instruction

## The instruction pointer, rip

- The rip register holds the address of the next instruction to be executed
- As the processor reads each instruction, it increments the value in rip by the appropriate number of bytes to point to the following instruction
- This mechanism allows the processor to execute a sequence of instructions stored in contiguous locations in memory
- What would happen if we “move” a different value in to rip?

## Jumping and labels

- We can transfer control and start executing instructions at address addr by using a jump instruction

```
jmp  addr
```

- Labels can be attached to instructions in an assembly language program:

```
►         jmp b
    a:    jmp c
    b:    jmp a
    c:    ...
```

- Modern, pipelined machines work well with sequences of instructions that appear in consecutive locations. Jumps can be expensive: one of the goals of an optimizing compiler is to avoid unnecessary jumps.

## x86-64 instructions: arithmetic and logic operations

## Arithmetic instructions

- Combine a given src with a given dst value and leave the result in dst:

```
► addX   src, dst  }
  subX   src, dst  } integer arithmetic
  imulX  src, dst  } (signed)
  andX   src, dst  }
  orX    src, dst  } bitwise arithmetic
  xorX   src, dst  }
```

- Similar to dst += src, dst -= src, etc.. in C/C++

## Examples

- To compute $x^2 + y^2$ and store the result in z:

```
► movq  x, %rax
  imulq %rax, %rax
  movq  y, %rbx
  imulq %rbx, %rbx
  addq  %rbx, %rax
  movq  %rax, z
```

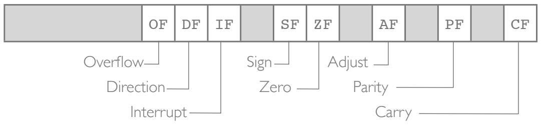
| register | contents |
|---|---|
| rax | $x^2+y^2$ |
| rbx | $y^2$ |

```
    .data
x:  .quad  4
y:  .quad  3
```

# x86-64 instructions: conditional execution

## Flags

- In addition to performing the required operation, arithmetic instructions also change bits in the `eflags` register



(Not to scale. Shaded areas indicate reserved or system fields.)

- The flags record details about the last operation, such as:
  - Was the result zero?
  - Was the result positive?
  - Did a carry occur?
  - etc...

## Conditional jumps, `jCC`

We can test these flags in *conditional jump* instructions

| | | |
|---|---|---|
| `jz addr` | (jump to `addr` if the zero flag is set) | |
| `jnz addr` | (jump to `addr` if the zero flag is not set) | |
| `je addr` | (jump to `addr` if equal; same as `jz`) | |
| `jne addr` | (jump to `addr` if not equal; same as `jnz`) | |
| `jl addr` | (jump to `addr` if less than) | (signed) |
| `jnl addr` | (jump to `addr` if not less than) | |
| `jg addr` | (jump to `addr` if greater than) | |
| `jng addr` | (jump to `addr` if not greater than) | |
| ... | | |

## Examples

```
subl   %eax,%ebx
jz     addr
```

jump to `addr` if `ebx` = `eax`

```
subq   %rax,%rbx
jnz    addr
```

jump to `addr` if `rbx` ≠ `rax`

```
subl   %eax,%ebx
jl     addr
```

jump to `addr` if `ebx` < `eax`

```
subq   %rax,%rbx
jnl    addr
```

jump to `addr` if `rbx` >= `rax`

If the specified condition does not apply, then execution just continues with the next instruction ...

## The compare instruction

- The `cmpX` instruction behaves like `subX` except that the result is not saved; only the flags are changed
- For example:

```
cmpl  %eax,%ebx
jl    addr
```

will jump to `addr` if the value in `ebx` is less than the value in `eax`, but it will not change the values in either register

## Other conditional instructions

- There are some other instructions that perform an action based on the conditional flags without the cost of a jump
- `setCC` reg8 sets the value in a specified 8-bit register to 0 or 1, based on the condition specified by `CC`:

```
cmpl     %ecx,%ebx    # set eax to 1 if
setl     %al          # ebx < ecx, or
movzbl   %al,%eax     # else to 0
```

- `cmovCC` src, dst copies data from the specified src to dst, but only if the condition specified by `CC` holds:

```
cmpl     %ebx,%eax    # set eax to the max of
cmovl    %ebx,%eax    # eax and ebx
```

condition code; no size suffix here!

# x86-64 instructions: more arithmetic

## Unary operations

- The following arithmetic operations have only one argument (which serves as both source and destination)

| | | |
|---|---|---|
| `negX` | (reg \| mem) | negate |
| `notX` | (reg \| mem) | complement |
| `incX` | (reg \| mem) | increment |
| `decX` | (reg \| mem) | decrement |

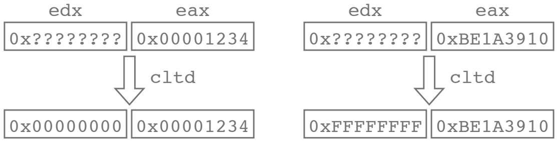
- Like the binary operators, these instructions also set the flags for subsequent testing

## Division

- Divide implicit destination (`edx:eax`) (a 64-bit quantity) by a specified argument with result in `eax` and remainder in `edx`

`idivl` (reg | mem)

- Often used in conjunction with the `cltd` instruction ("convert long to double", a.k.a. `cdq`), which converts a signed 32-bit value in `eax` into the corresponding signed 64-bit value in `edx:eax`.



## Example 1

Divide 4,660 (i.e., 0x1234) by 25:

```
movl   $0x1234, %eax
cltd
movl   $25, %ecx
idivl  %ecx
```

Results: `eax` = `0xBA` (186)
`edx` = `0xA` (10)

Sure enough: 186*25 + 10 = 4,660

## Example 2

Divide -1,105,577,712 (i.e., 0xBE1A3910) by 256

```
movl   $0xBE1A3910, %eax
cltd
movl   $256, %ecx
idivl  %ecx
```

Results: `eax` = `0xFFBE1A3A` (-4,318,662)
`edx` = `0xffffff10` (-240)

Sure enough: -4,318,662 * 256 - 240 = -1,105,577,712

## Complications of division

- Division produces multiple results: a quotient and a remainder
- Division uses special registers: we'd better not store any other values in `eax` or `edx` if there's a chance that a division instruction might be executed
- Division can raise an exception if the src is zero (or -1)
- (We can do division on 64-bit registers in the analogous way using `rax` and `rdx`. I described the 32-bit case here because the numbers are smaller, and because that is what we'll use for `int` arithmetic.)

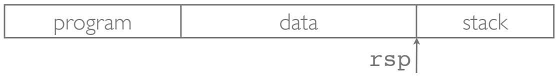
# x86-64 instructions: using the stack

## Stack

- The x86-64 includes features that allow the programmer to use a region of memory as a simple stack:
  - the `rsp` (stack pointer) register
  - special instructions like `push`, `pop`, `call`, `ret`, ...
- There is no obligation for the programmer to use these features, but it is often convenient to do so:
  - for temporary/scratch storage when a calculation needs more storage than the CPU registers can provide
  - to support calling and returning from functions

## A typical memory layout

- A typical operating system reserves an area of scratch memory for each program, and sets the `rsp` register to point to the end of this region when the program begins



- The stack pointer moves
  - down (decreases) as values are pushed on to the stack
  - up (increases) as values are popped off of the stack
- So long as they never overlap, the data and stack areas can grow or shrink as necessary as the program runs

## Stack operations

- Push a value onto the stack

  `pushq` (reg | mem | immed)

- Pop a value of the stack

  `popq` (reg | mem)

- Roughly speaking:

```
• pushq src   =   subq $8, %rsp;   movq src, (%rsp)
• popq dst    =   movq (%rsp), dst;   addq $8, %rsp
```

## Spilling temporaries on the stack

- The stack is often used for saving the contents of a register on the stack ("spilling") so that the register can be used, temporarily, for some other reason
- For example:

```
pushq  %rax
pushq  %rdx
... code that changes rax and/or rdx ...
popq   %rdx
popq   %rax
```

pop values in reverse order that was used to push them!

- Note that values on the stack can still be accessed, from memory, using `(%rsp)`, `8(%rsp)`, `16(%rsp)`, `24(%rsp)`, ...

## Call and return

- There is a special instruction for calling a function

```
call addr      ≃         pushq  $lab
                         jmp    addr
                     lab: ...
```

- And a special instruction for returning from a function

```
ret            ≃         popq   %rax
                         jmp    *%rax
```

assuming `rax` isn't being used for something else ...

special syntax: jump to the address given by the contents of `rax`

- In practice, additional instructions are often needed to deal with parameter passing, etc. ...
- (to be continued!)

# Closing thoughts

## CISC: a complex instruction set computer?

- The x86-64 is often referred to as a CISC because a single instruction can execute multiple low-level operations
- For example:<br>does the same as:

```
For example:       movq   8(%rbx, %rax, 4), %rax
does the same as:  imulq  $4, %rax
                   addq   %rbx, %rax
                   addq   $8, %rax
                   movq   (%rax), %rax
```

- Read the technical documentation to determine which sequence is faster/takes fewer bytes ... but the single instruction is usually the winner
- Using CISC instructions effectively is a major challenge for compiler writers!

## RISC vs CISC

- RISC machines are "reduced instruction set computers" that require multiple instructions to simulate one CISC instruction ... but each RISC instruction can potentially run faster
- Easier compiler targets? RISC machines are typically simpler and more regular than CISC machines. They often have more registers that the programmer/compiler can use.
- Harder compiler targets? A general philosophy of some RISC machine designs is to let compilers rather than CPUs handle the complex tasks.

## The GNU assembler, as

- Assembly code goes in files with a .s suffix
- We will typically use `gcc` to invoke the assembler

```
gcc -o output assemblyCode.s runtime.c
```

- You can also invoke the assembler directly: detailed documentation is available from:

  http://sourceware.org/binutils/docs/as/

  For x86 programming, look in particular at the section on "80386 Dependent Features"
- This information is provided only for the curious; for the purposes of this class, all the information that you need should be on these slides, or in the distributed source code.

## Summary

- An x86-64 machine has
  - A fixed set of registers
  - Instructions for moving and operating on data
  - Instructions for testing and control transfer
- To generate good code for an x86-64 machine, we need:
  - To select appropriate instructions
  - To make good use of registers and avoid unnecessary memory accesses
  - To avoid using registers that are already in use
  - To keep the number of jumps as low as possible