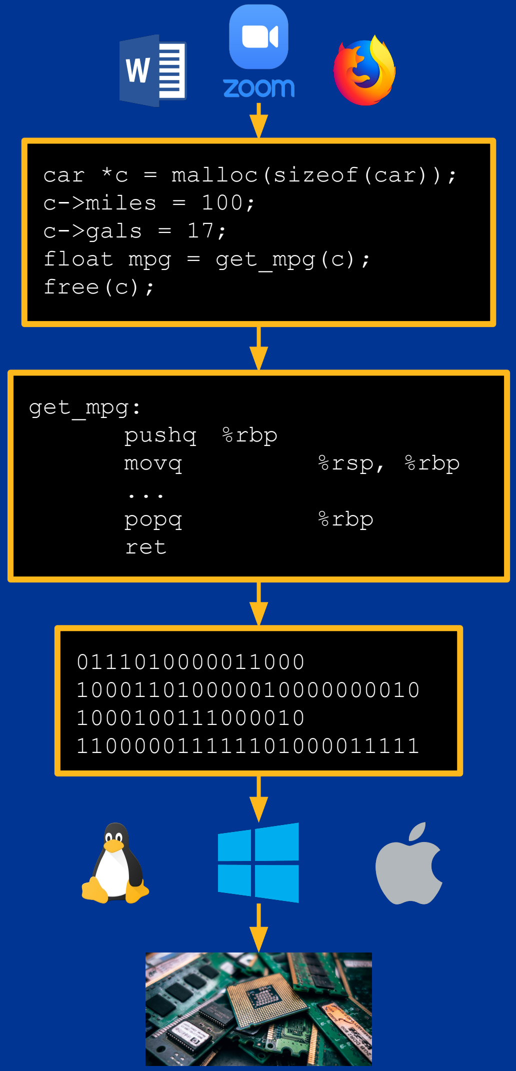
University of Pittsburgh | School of Computing and Information

Week 7
Assembly

CS 449 Spring 2020

View of a System



```
car *c = malloc(sizeof(car));
c->miles = 100;
c->gals = 17;
float mpg = get_mpg(c);
free(c);
```

```
get_mpg:
       pushq  %rbp
       movq          %rsp, %rbp
       ...
       popq          %rbp
       ret
```

```
0111010000011000
100011010000010000000010
1000100111000010
110000011111101000011111
```

Very abstract

Level of abstraction

Not abstract

Application level (Word, Zoom, Firefox)

High-level language level (C, Java)

Assembly language level (x86)

Machine language level

Operating system level (Linux, Windows, MacOS)

Hardware level

What is Assembly Code?

High-level language (C, Java)

```
car *c = malloc(sizeof(car));
c->miles = 100;
c->gals = 17;
float mpg = get_mpg(c);
free(c);
```

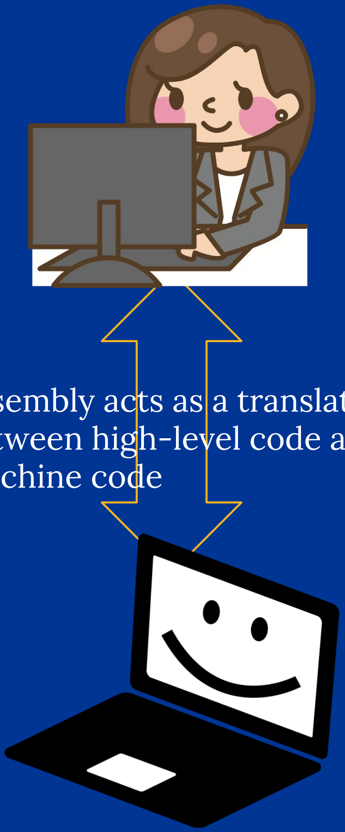
Easy for us to understand

```
get_mpg:
        pushq  %rbp
        movq          %rsp, %rbp
        ...
        popq          %rbp
        ret
```

```
0111010000011000
100011010000010000000010
1000100111000010
110000011111101000011111
```

Machine language

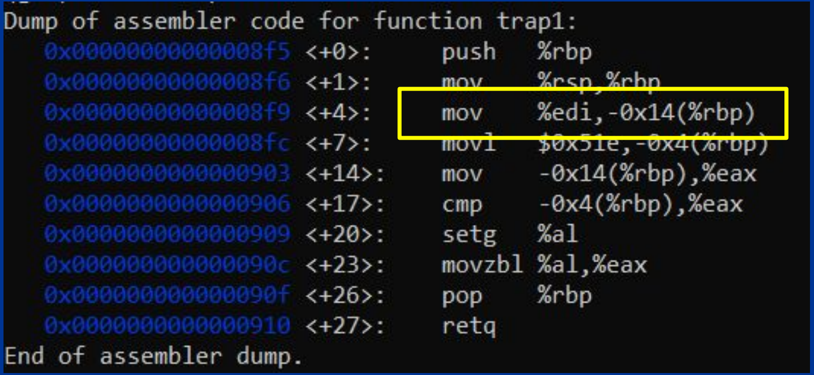
Easy for computer to understand



Assembly acts as a translator between high-level code and machine code

AT&T vs. Intel

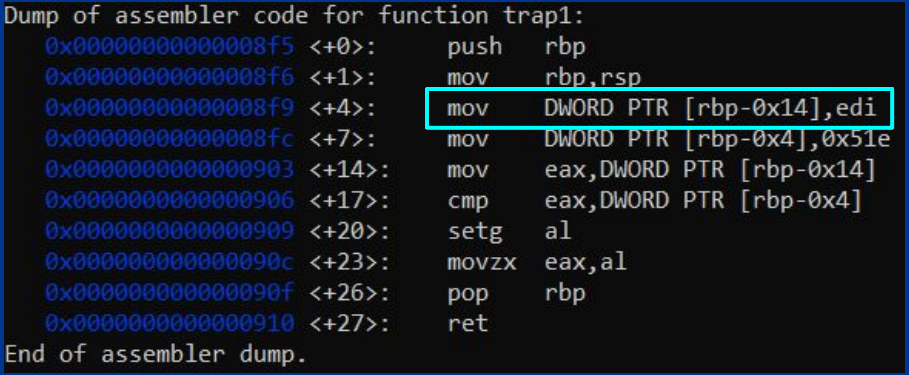
at&t



```
Dump of assembler code for function trap1:
   0x00000000000008f5 <+0>:     push   %rbp
   0x00000000000008f6 <+1>:     mov    %rsp,%rbp
   0x00000000000008f9 <+4>:     mov    %edi,-0x14(%rbp)
   0x00000000000008fc <+7>:     movl   $0x51e,-0x4(%rbp)
   0x0000000000000903 <+14>:    mov    -0x14(%rbp),%eax
   0x0000000000000906 <+17>:    cmp    -0x4(%rbp),%eax
   0x0000000000000909 <+20>:    setg   %al
   0x000000000000090c <+23>:    movzbl %al,%eax
   0x000000000000090f <+26>:    pop    %rbp
   0x0000000000000910 <+27>:    retq
End of assembler dump.
```

`%<register>`

intel



```
Dump of assembler code for function trap1:
   0x00000000000008f5 <+0>:     push   rbp
   0x00000000000008f6 <+1>:     mov    rbp,rsp
   0x00000000000008f9 <+4>:     mov    DWORD PTR [rbp-0x14],edi
   0x00000000000008fc <+7>:     mov    DWORD PTR [rbp-0x4],0x51e
   0x0000000000000903 <+14>:    mov    eax,DWORD PTR [rbp-0x14]
   0x0000000000000906 <+17>:    cmp    eax,DWORD PTR [rbp-0x4]
   0x0000000000000909 <+20>:    setg   al
   0x000000000000090c <+23>:    movzx  eax,al
   0x000000000000090f <+26>:    pop    rbp
   0x0000000000000910 <+27>:    ret
End of assembler dump.
```

`<register>`

```
(gdb) set disassembly-flavor att
(gdb) set disassembly-flavor intel
(gdb) show disassembly-flavor
```

How to change syntax (aka flavor)

Operand Types

Immediate

`$0x400`

Encoded by 1, 2, 4, or 8 bytes, depending on instruction

Registers

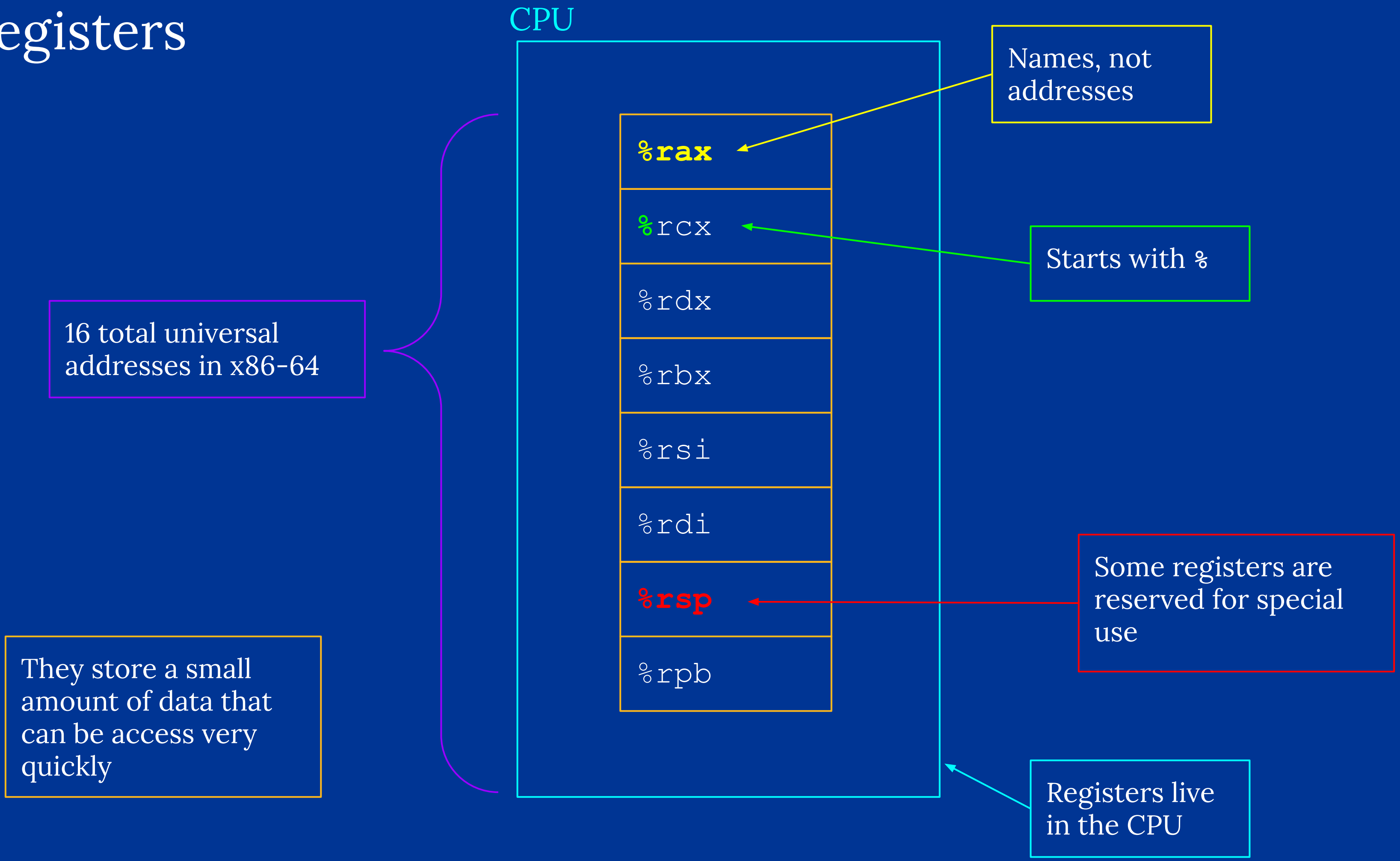
`$rax`

Memory

`($rax)`

Consecutive bytes of memory at a computed address

Registers



CPU

%rax

%rcx

%rdx

%rbx

%rsi

%rdi

%rsp

%rpb

Names, not addresses

Starts with %

Some registers are reserved for special use

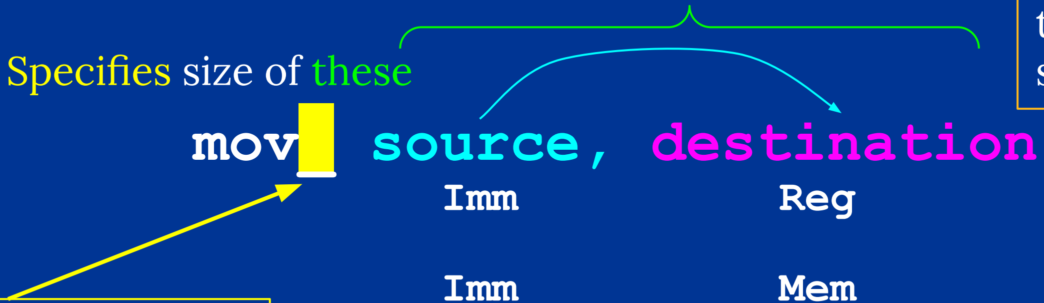
Registers live in the CPU

16 total universal addresses in x86-64

They store a small amount of data that can be access very quickly

Data Transfer - `mov`

Assigning a value from the destination into the source



Specifies size of these

```
mov  source, destination
```

Imm	Reg
Imm	Mem
Reg	Reg
Reg	Mem
Mem	Reg

mov**b**: 1 byte (**byte**)
mov**w**: 2 byte (**word**)
mov**l**: 4 byte (**long word**)
mov**q**: 8 byte (**quad word**)

word → 16 bits → 2 bytes

Note: Can't do `mov mem mem` directly

Have to move memory address to register, then move register to memory address

Arithmetic Operations

Format	Computation	
addq src, dest	dest = dest + src	
subq src, dest	dest = dest - src	
imulq src, dest	dest = dest * src	Signed mult
sarq src, dest	dest = dest >> src	Arithmetic
shrq src, dest	dest = dest >> src	Logical
shlq src, dest	dest = dest << src	
xorq src, dest	dest = dest ^ src	
andq src, dest	dest = dest & src	
orq src, dest	dest = dest | src	

Binary (2 operands)

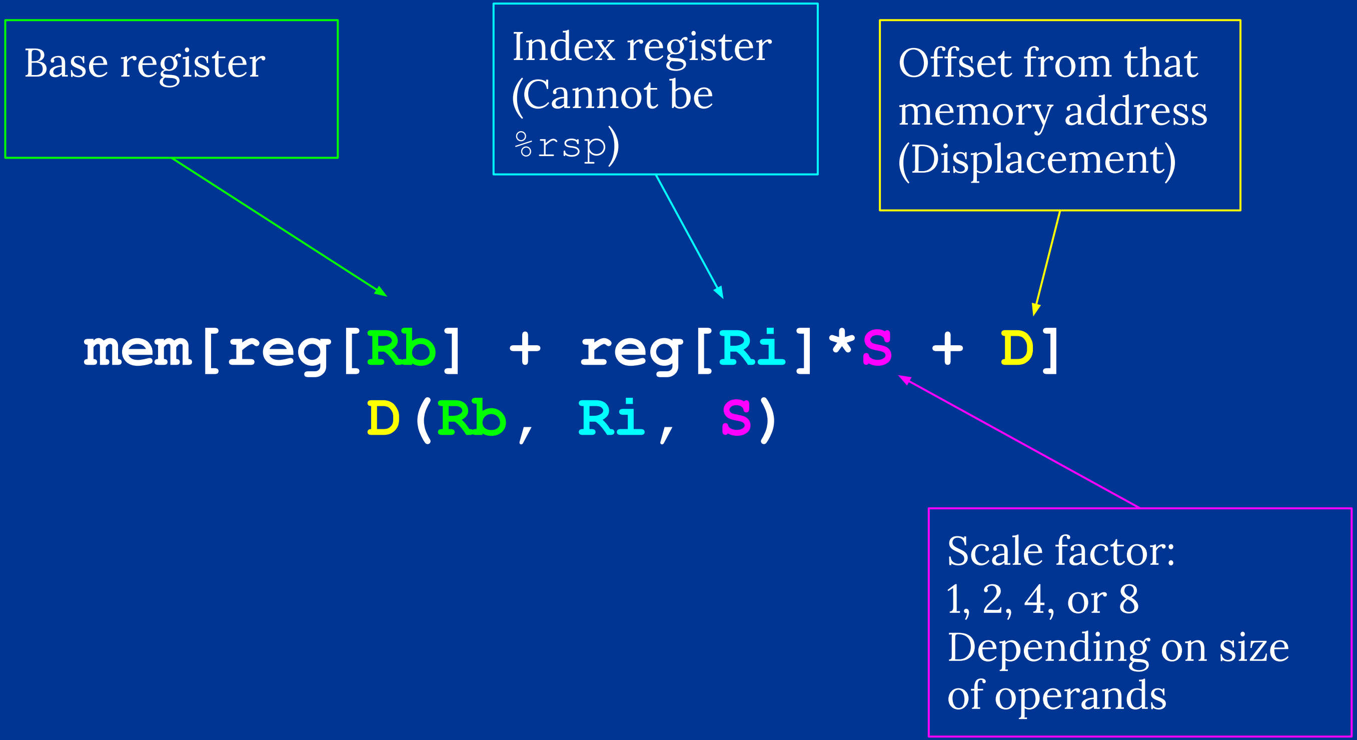
Format	Computation
incq dest	dest = dest + 1
decq dest	dest = dest - 1
negq dest	dest = -dest
notq dest	dest = ~dest

Unary (1 operand)

Remember the operand size specifier

Binary operations can only have **one memory operand** at most!

Memory Addressing - in General



Base register

Index register (Cannot be `%rsp`)

Offset from that memory address (Displacement)

```
mem[reg[Rb] + reg[Ri]*S + D]
        D(Rb, Ri, S)
```

Scale factor:
1, 2, 4, or 8
Depending on size
of operands

Computing Addresses

%rdx	0xf000
%rcx	0x0100

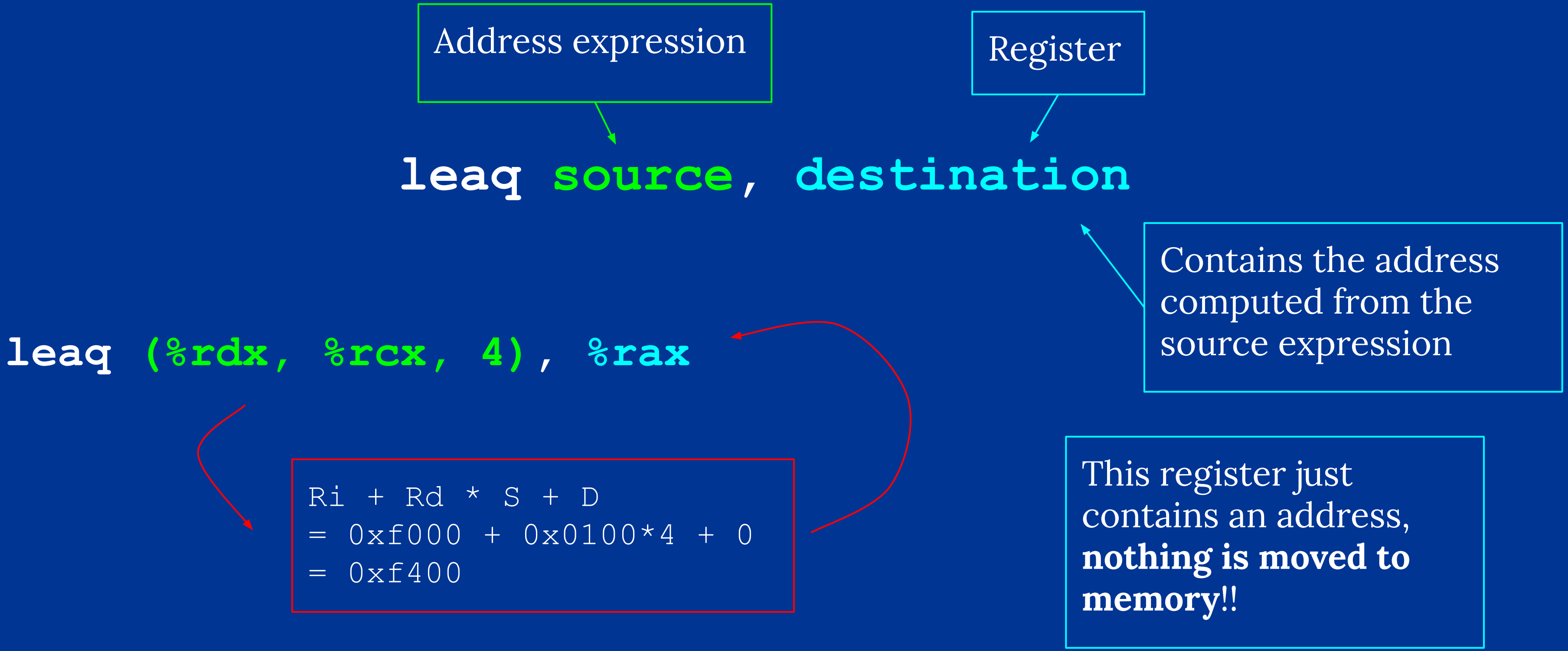
```
D(Rb, Ri, S) = mem[reg[Rb] + reg[Ri]*S + D]
```

Expression	Rb	Ri	D	S	Computation	Address
`0x8(%rdx)`	0xf000	0	0x80	1	0xf000 + 0*1 + 0x8	0xf008
`(%rdx, %rcx)`	0xf000	0x0100	0	1	0xf000 + 0x0100*1 + 0	0xf100
`(%rdx, %rcx, 4)`	0xf000	0x0100	0	4	0xf000 + 0x0100*4 + 0	0xf400
`0x80(, %rdx, 2)`	0	0xf000	0x80	2	0 + 0xf000*2 + 0x80	0x1e080

Rb	Ri	D	S
0	0	0	1

Default values

`lea` - Load Effective Address



Address expression

Register

```
leaq source, destination
```

Contains the address computed from the source expression

```
leaq (%rdx, %rcx, 4), %rax
```

```
Ri + Rd * S + D
= 0xf000 + 0x0100*4 + 0
= 0xf400
```

This register just contains an address, **nothing is moved to memory**!!

mov vs. lea

Registers

%rax	0x110
%rbx	
%rcx	0x4
%rdx	0x100

Memory

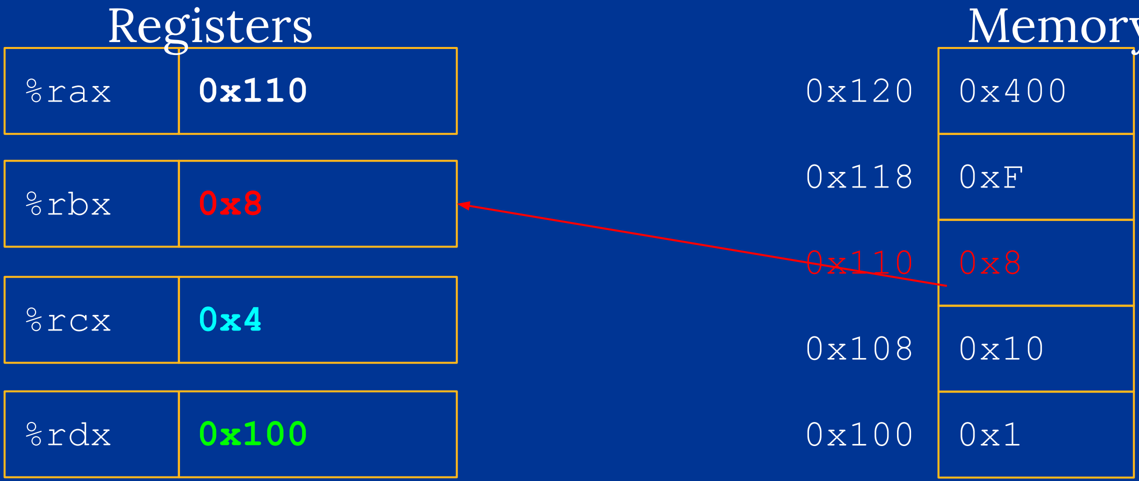
0x120	0x400
0x118	0xF
0x110	0x8
0x108	0x10
0x100	0x1

lea: store the **address** in the register

```
leaq (%rdx, %rcx, 4), %rax
```

```
%rax = %rcx * 0x4 + %rdx
%rax = 0x4 * 0x4 + 0x100
%rax = 0x110
```

mov vs. lea



Registers

%rax	0x110
%rbx	0x8
%rcx	0x4
%rdx	0x100

Memory

0x120	0x400
0x118	0xF
0x110	0x8
0x108	0x10
0x100	0x1

mov: store the **value at the address** in the register

```
movq (%rdx, %rcx, 4), %rbx
```

```
%rbx = MEM[%rcx * 0x4 + %rdx]
%rbx = MEM[0x4 * 0x4 + 0x100]
%rbx = MEM[0x110]
```

mov vs. lea

mov	**lea**
Calculates address and then stores the value at that address, then assigns to the second argument.	Calculates address and assigns it to the second argument. Can use to perform calculations

Comparison

```
cmp operand1, operand2
```

Performs operand1 - operand2 and sets flags based on result

The values of operand1 and operand2 are NOT changed

Comparison is usually followed by a jump or set operation. The flags set by the comparison are used to determine what should happen with a jump/set instruction.

Set after a comparison

Instruction		If comparison is true
`sete` *`dest`*	Set byte if equal	dest = 1
`setg` *`dest`*	Set byte if greater	dest = 1

After a comparison (`cmp`), `set` will set the destination based on whether or not the comparison was true

`setg` example

Instruction	
`setg` *`dest`*	Set byte if greater

Compare **B** with **A**

If **B** **greater than** **A**,

- set $al to 1
- Otherwise, $a1 set to 0

```
cmpl $0x9, -0x4(%rbp)
setg %al
```

($rbp - 0x4) = 10?

Compare 9 and 10

Is 10 greater than 9?

- Yes, $al = 1

($rbp - 0x4) = 0?

Compare 9 and 0

Is 0 greater than 9?

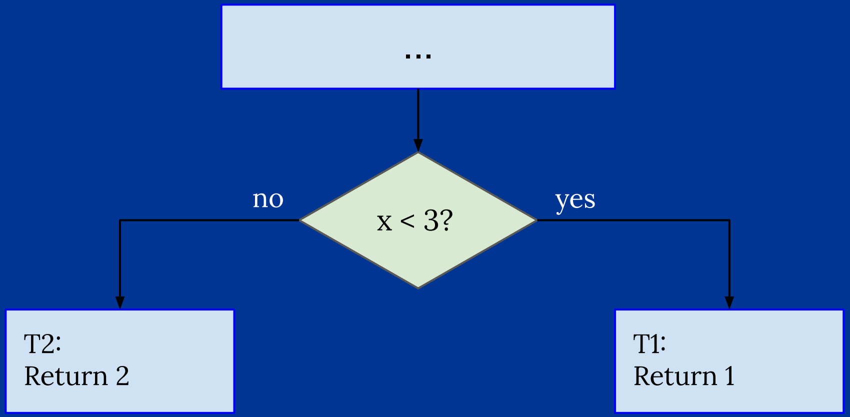
- No, $al = 0

Jump after a comparison

```
if (x < 3) {
    return 1;
}
return 2;
```

```
cmpq $3, %rdi
jge T2
T1:
    movq    $1, %rax
    ret
T2:
    movq    $2, %rax
    ret
```

After a comparison (`cmp`), jump instructions tell the program which line to jump to based on the outcome of the comparison.



1. Compare 3, x
2. Jump to T2 if x >= 3

```
if (x < 3) {
    return 1;
}
return 2;
```

```
cmpq $3, %rdi
jge T2
T1:
    movq    $1, %rax
    ret
T2:
    movq    $2, %rax
    ret
```

Instruction		`cmp a, b`	`test a, b`
`je`	“Equal”	b == a	b&a == 0
`jne`	“Not equal”	b != a	b&a != 0
`js`	“Sign” (negative)	b - a < 0	b&a < 0
`jns`	(non-negative)	b - a >= 0	b&a >= 0
`jg`	“Greater”	b > a	b&a > 0
`jge`	“Greater or equal”	b >= a	b&a >= 0
`jl`	“Less”	b < a	b&a < 0
`jle`	“Less or equal”	b <= a	b&a <= 0
`ja`	“Above” (unsigned >)	b > a	b&a > 0U
`jb`	“Below” (unsigned <)	b < a	b&a < 0U