

← gloriouscow

🌐 4 h

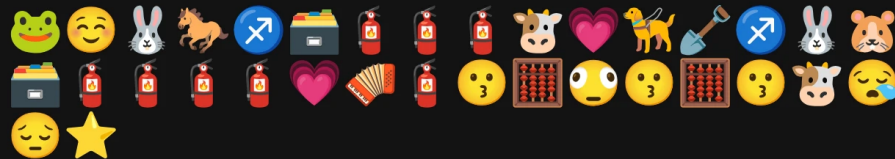
@gloriouscow@oldbytes.space

The following is a valid DOS COM executable that prints "HELLO" in the lower right corner of the screen.

You can copy these emoji into a text editor, and save it as EMOJI.COM. it should be 141 bytes.

It will run in DosBox-X, with the following options:

```
cpu cputype=8086  
machine cga
```



may god have mercy on my soul.

TODO

- Assignment 2 assigned due October 7
- Study for Exam 1 October 9

Overview for today

- Calling functions
- Control flow operations

Lab 1 graded

- No hidden tests on this one
- If you passed all tests, you generally got a perfect grade
- Otherwise I left hints for incomplete solutions
- Group submissions: submitter received grade

Exam 1 – Any questions?

- First third (Exam 1):
 - Data Representation
 - Memory
- Second third (Exam 2):
 - Machine Code
 - Memory exploits
- Final third (Final exam):
 - Memory hierarchy
 - Processes

lea.c

Memory operands

General Form:

$D(Rb, Ri, S) \quad Mem[D + Reg[Rb] + S * Reg[Ri]]$

D Constant "displacement"

Rb Base register

Ri Index register

s Scale constant: 1, 2, 4, or 8

$5(\%rax, \%rbx, 8) \quad Mem[5 + \%rax + 8 * \%rbx]$

Special Cases:

(Rb)	(%rax)	Mem[Reg[Rb]]
D(Rb)	5(%rax)	Mem[D + Reg[Rb]]
(Rb, Ri)	(%rax, %rbx)	Mem[Reg[Rb] + Reg[Ri]]
D(Rb, Ri)	5(%rax, %rbx)	Mem[D + Reg[Rb] + Reg[Ri]]
(Rb, Ri, S)	(%rax, %rbx, 8)	Mem[Reg[Rb] + S * Reg[Ri]]
(, Ri, S)	(, %rbx, 8)	Mem[S * Reg[Ri]]
D(, Ri, S)	5(, %rbx, 8)	Mem[D + S * Reg[Ri]]

Three categories of instructions

1. Data movement – moving data between registers and memory
 - E.g., load data from memory address 0x20 into register `%rax`
2. Arithmetic – perform arithmetic on data in registers or in memory
 - E.g., take the value in `%rax` and add it to the value in `%rbx`
3. **Control flow** – choosing next instruction
 - Calling a function
 - Conditional branches, loops, etc.

Control flow

- Sometimes we don't want to just keep executing instructions sequentially
 - Function calls
 - Loops
 - Branches/conditionals

Calling functions

- `callq label_name`
- `ret`

Calling functions

- `callq label_name`
 - “Jumps” to the address of the function
- `ret`
 - “Jumps” back to the instruction after `callq`
- Jump: change the program counter
- Anything important missing from this picture?

Calling functions

- `callq label_name`
 - “Jumps” to the address of the function
- `ret`
 - “Jumps” back to the instruction after `callq`
- Jump: change the program counter
- Where are the function arguments!?!?
- Where is the return value!?!?

Return	%rax	%eax
	%rbx	%ebx
Arg 4	%rcx	%ecx
Arg 3	%rdx	%edx
Arg 2	%rsi	%esi
Arg 1	%rdi	%edi
Stack ptr	%rsp	%esp
	%rbp	%ebp

%r8	%r8d	Arg 5
%r9	%r9d	Arg 6
%r10	%r10d	
%r11	%r11d	
%r12	%r12d	
%r13	%r13d	
%r14	%r14d	
%r15	%r15d	

Calling *convention*

- Other than `%rsp`, none of this convention is dictated by the hardware
- It is merely a *convention*
- You don't have to follow it, but other people's code might have trouble calling yours

```
long x = add(3, 5);  
doSomething(x);
```

```
long add(long x, long y) {  
    return x + y;  
}
```

```
movq $3, %rdi  
movq $5, %rsi  
callq add  
movq %rax, %rdi  
callq doSomething
```

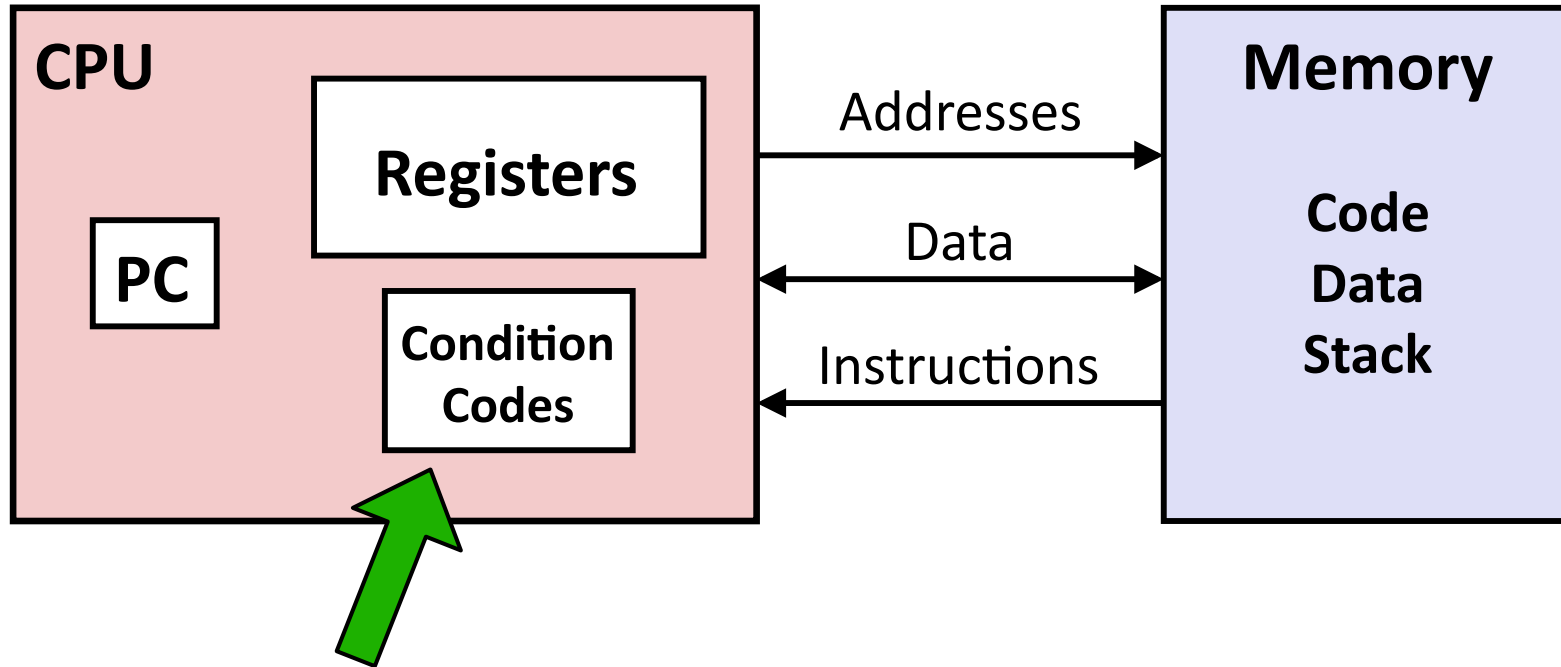
```
add:  
    movq %rdi, %rax  
    addq %rsi, %rax  
    ret
```

lea.c

How do we implement...

- If statements?
- Conditional branching?
- We first need to understand condition codes...

Condition codes



How do we implement branching?

- We first need to understand condition codes
- These are effectively special 1-bit registers
- They get automatically updated after most arithmetic instructions based on the result
- x86 processors have at least four...

Control codes

- What are they exactly?
- How are they assigned?
- How do we read them?
- How do we use them for control flow?

CF

ZF

SF

OF

Condition codes

CF: **Carry flag** (set if carry-out bit = 1)

ZF: **Zero flag** (set if result = 0)

SF: **Sign flag** (set if result high bit = 1)

OF: **Overflow flag** (set if signed overflow)

Setting
CCs

(1) `cmpq a,b` (set based on `b - a`)

(2) `testq a,b` (set based on `a & b`)

(3) arithmetic insts (implicit, all except `leaq`)

How are control codes assigned?

- Based on the value of the **destination register** after an arithmetic operation
- Exception: LEA does not update codes

Can hardware detect overflow?

- Some of these might be easier to understand than others...
 - **ZF**
 - **SF**
- But what about **CF** and **OF**?

Can hardware detect overflow?

- Addition: **CF** = carry out bit

$$\begin{array}{r} 1111 \quad (15) \\ + \quad \underline{0001} \quad (1) \\ \hline \mathbf{1}0000 \quad (0) \end{array}$$

Can hardware detect overflow?

- Subtraction: **CF** = borrow bit

0000 (0)

- 0001 (1)

11111 (15)

Can hardware detect overflow?

- Signed overflow: high bit consistency
- Observation: overflow occurs when adding two positive numbers and the result is negative or two negative numbers and the result is positive

Can hardware detect overflow?

- Signed overflow: high bit consistency

0111 (7)

+ 0001 (1)

1000 (-8)

- A, B, S: high bits of addends and sum
- Addition: $OF = \sim(A \wedge B) \& (A \wedge S)$
- “A is same as B but different than S.”

Can hardware detect overflow?

- Signed overflow: high bit consistency

0111 (7)

- 1111 (-1)

1000 (-8)

- A, B, S: high bits of minuend, subtrahend, and diff
- Subtraction: $OF = (A \wedge B) \& (A \wedge S)$
- “A is different than B and different than S.”

Special instructions

- `cmpq a, b` – like `subq` except does not change `b`
- `testq a, b` – like `andq` but does not change `b`
- These are useful when all you want to change is the control codes

cmpq a, b

- Sets flags based on computation of $b - a$
- E.g., `cmpq %rax, %rbx`
- If `%rax` and `%rbx` are equal, then **ZF** = 1

testq a, b

- Sets flags based on computation of a & b
- A very common use is testing if an operand is zero, e.g., testq %rax, %rax
- If %rax is zero, then **ZF** = 1
- Why might a compiler use testq %rax, %rax instead of cmpq \$0, %rax?
- Also note: testq always sets **CF** and **OF** to 0

Remember

- We don't need `cmpq` and `testq`
- We could use `subq` and `andq`
- But `cmpq` and `testq` are often more convenient

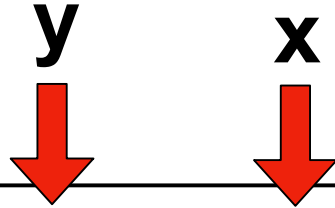
So that's how we set control codes

- How do we read them?
- Using `set*` instructions
- These only accept a single 8-bit operand
- Set the operand to either 0x00 or 0x01 depending on the control codes
- Here are some examples...

SetX	Condition	Description
sete setz	ZF	Equal / Zero
setne setnz	~ZF	Not Equal / Not Zero
sets	SF	Negative
setns	~SF	Nonnegative
setg	~ (SF^OF) & ~ZF	Greater (Signed)
setge	~ (SF^OF)	Greater or Equal (Signed)
setl	(SF^OF)	Less (Signed)
setle	(SF^OF) ZF	Less or Equal (Signed)
seta	~CF & ~ZF	Above (unsigned)
setb	CF	Below (unsigned)

```
int gt(long x, long y) {  
    return x > y;  
}
```

Register	Use(s)
%rdi	Argument x
%rsi	Argument y
%rax	Return value



```
cmpq    %rsi, %rdi    # Compare x:y  
setg    %al           # Set when >  
movzbl  %al, %eax     # Zero rest of %rax  
ret
```

```
int gt(long x, long y) {  
    return x > y;  
}
```

Note: “backwards”
argument order 🙄

	y	x	
			
cmpq	%rsi,	%rdi	# Compare x:y
setg	%al		# Set when >
movzbl	%al,	%eax	# Zero rest of %rax
ret			

```
int gt(long x, long y) {  
    return x > y;  
}
```

Why is movzbl
necessary?

	y	x	
			
cmpq	%rsi,	%rdi	# Compare x:y
setg	%al		# Set when >
movzbl	%al,	%eax	# Zero rest of %rax
ret			

Assembly exercises

Burying the lede

- Like, sure, you can say:
- `int foo = x < y;`
- But where do we typically see expressions like “`x < y`”?

Jumps

- We've seen a special form of them via `callq`
- In C, closest analog is `goto...`
- Syntax: `goto label_name;`

goto.c

```
#include <stdio.h>

int main() {
    int n = 0;

    // If the number is zero, jump to `jump_here`
    if (n == 0)
        goto jump_here;

    // This will be skipped
    printf("You entered: %d\n", n);

jump_here:
    printf("Exiting the program.\n");
    return 0;
}
```

In assembly...

- All we have are gotos...
- No loop constructs
- No if statements
- We must construct all of these using gotos!

TODO

- Assignment 2 assigned due October 7
- Study for Exam 1 October 9