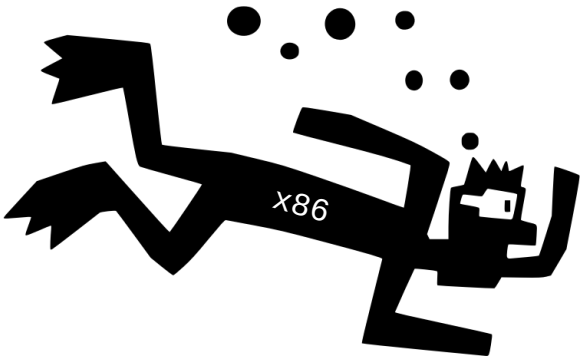
CS 261
Fall 2018

Mike Lam, Professor

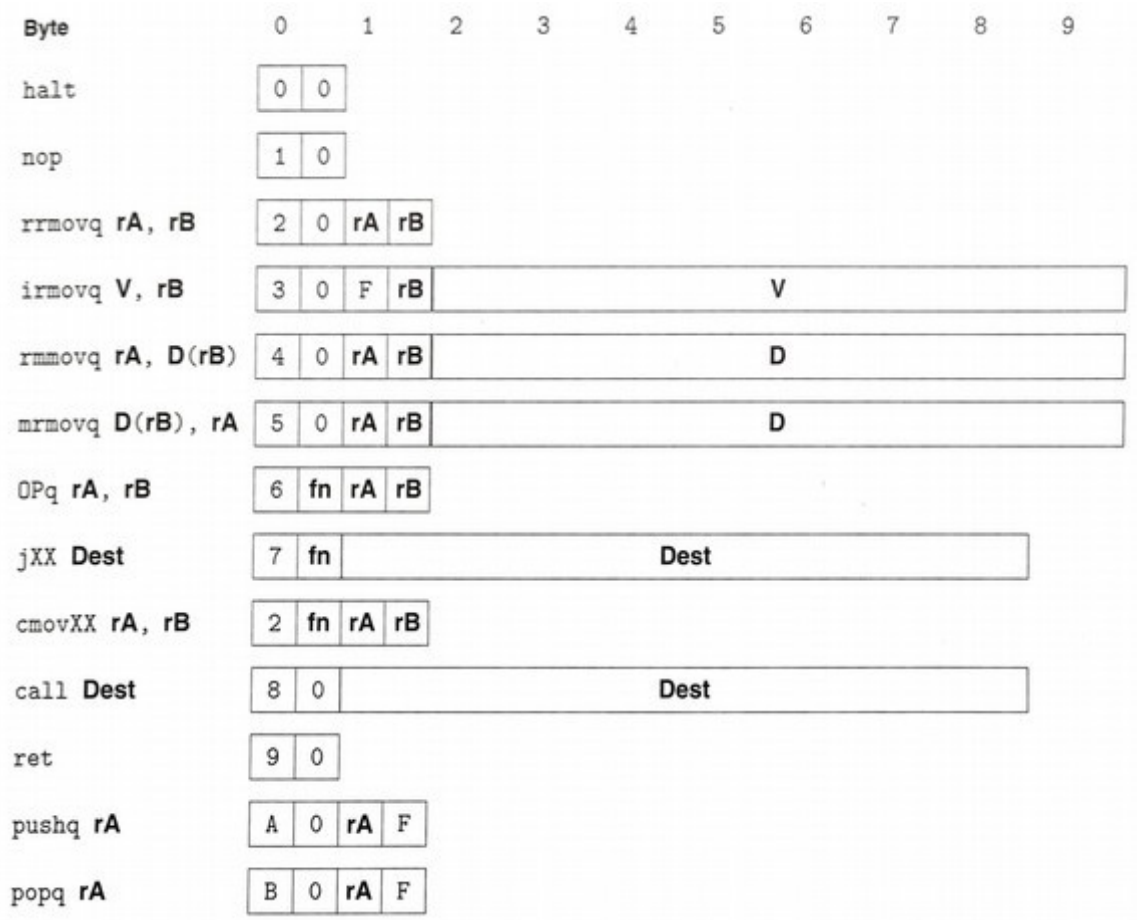
Q. Why do assembly programmers need to know how to swim?

A. Because they work below C level!



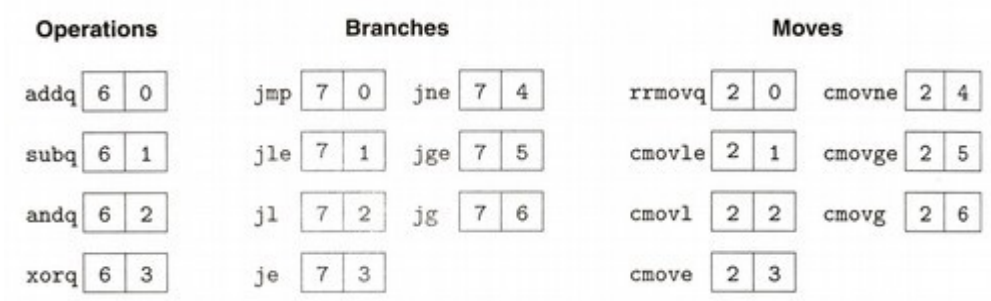
Y86-64 Introduction

Projects 3 & 4: Y86-64 ISA



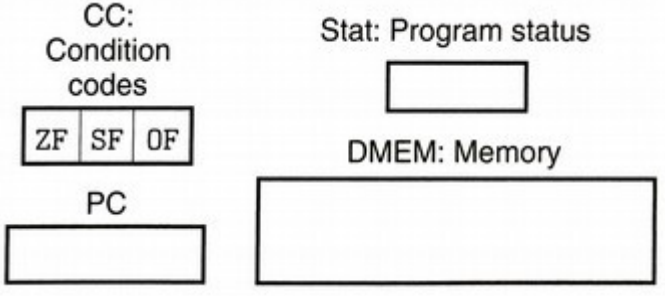
Number	Register name
0	%rax
1	%rcx
2	%rdx
3	%rbx
4	%rsp
5	%rbp
6	%rsi
7	%rdi

Value	Name	Meaning
1	AOK	Normal operation
2	HLT	halt instruction encountered
3	ADR	Invalid address encountered
4	INS	Invalid instruction encountered



RF: Program registers

%rax	%rsp	%r8	%r12
%rcx	%rbp	%r9	%r13
%rdx	%rsi	%r10	%r14
%rbx	%rdi	%r11	



Y86 quick reference

Y86 Instruction Set Reference

Instruction	Byte offset from PC									
	0	1	2	3	4	5	6	7	8	9
halt	0 0									
nop	1 0									
cmovXX rA, rB	2 fn	rA rB								
irmovq V, rB	3 0	f rB	V							
rmmovq rA, D(rB)	4 0	rA rB	D							
mrmovq D(rB), rA	5 0	rA rB	D							

Instruction	Byte offset from PC								
	0	1	2	3	4	5	6	7	8
OPq rA, rB	6 fn	rA rB							
jXX Dest	7 fn	Dest							
call Dest	8 0	Dest							
ret	9 0								
pushq rA	a 0	rA f							
popq rA	b 0	rA f							

cmovXX:			
rrmovq	20	cmovne	24
cmovle	21	cmovge	25
cmovl	22	cmovg	26
cmove	23		

OPq:	
addq	60
subq	61
andq	62
xorq	63

jXX:			
jmp	70	jne	74
jle	71	jge	75
jl	72	jg	76
je	73		

Registers:				Args:
%rax+	0	%rsp	4	%rdi
%rcx+	1	%rbp*	5	%rsi
%rdx+	2	%rsi+	6	%rdx
%rbx*	3	%rdi+	7	%rcx

+caller-save *callee-save

```
0x100:                                | .pos 0x100 code
0x100:                                | _start:
0x100: 30f10700000000000000           |     irmovq $7, %rcx
0x10a: 30f00200000000000000           |     irmovq $2, %rax
0x114: 6010                           |     addq %rcx, %rax
0x116: 00                             |     halt
                                      |
```

Projects 3 & 4: Support Utilities

- Run this script: `/cs/students/cs261/y86/install.sh`
  - **`yas`**: Y86-64 assembler (.ys → .yo and .o)
  - **`y86ref`**: compiled reference solution to P3/P4
    - Use “-d” to disassemble (P3) or “-e” to execute (P4)
  - **`ysim`**: Y86-64 simulator (runs .yo files)
    - Use “-g” option for visual mode (must have X11 forwarding enabled; use “ssh -Y”)
- These will help with P3/P4: learn to use them!
  - "`yas <yourfile.ys>`" to assemble code into object files
- Web-based simulator: `https://lam2mo.github.io/js-y86-64/`
  - Non-authoritative; use with caution
  - If there is a discrepancy, trust `y86ref`/`ysim` over this one

Exercises

- Write Y86-64 code to add 3 and 5 (store result in `%rbx`)
- Write Y86-64 code to multiply 3 and 5 (store result in `%rcx`)
  - HINT: add 3 to itself 5 times, or vice versa

Differences from textbook

- Execution begins at "entry point" from MiniELF, not address zero
  - This avoids the situation of having program code at “NULL”
  - Use "`_start`" label to indicate entry point in assembly
  - Use a jump if you want to run the simulator
  - Example:

```
.pos 0 code
    jmp _start

.pos 0x100 code
_start:
    <code goes here>
```

Using the stack

- The stack must be initialized manually
  - Example:

```
.pos 0 code
   jmp _start

.pos 0x100 code
_start:
   irmovq _stack, %rsp
   <code goes here>

.pos 0xf00 stack
_stack:
```

Data segments

- Data should be stored in `data` or `rodata` segments
  - Retrieve address (i.e., create pointer) using labels and `irmovq`
    - `.quad` for 64-bit signed integers and `.string` for character strings
  - No indexed addressing mode--must do pointer arithmetic yourself!
  - Example:

```
.pos 0x100 code
_start:
    irmovq vals, %rbx          # rbx = &vals
    mrmovq (%rbx), %rax        # rax = *rbx

    irmovq $16, %rdi           # 16 = 8 * 2
    addq %rbx, %rdi
    mrmovq (%rdi), %rcx        # rcx = vals[2]

.pos 0x400 data
vals:
    .quad 1
    .quad 2
    .quad 3
    .quad 4

.pos 0x600 rodata
my_str:
    .string “Hello”
```

Template

```
.pos 0 code
    jmp _start

.pos 0x100 code
_start:
    irmovq _stack, %rsp
    # YOUR CODE GOES HERE
    halt

.pos 0x400 data
    # YOUR DATA GOES HERE

.pos 0xf00 stack
_stack:
```