

# CS 261 Fall 2021

Mike Lam, Professor

**x86-64**



<https://www.amazon.com/dp/B001DZTJRQ>

**Y86-64**



<https://www.amazon.com/dp/B000004YVB2/>

## Y86-64 Introduction

# Projects 3 & 4: Y86-64 ISA

Byte	0	1	2	3	4	5	6	7	8	9
halt	0	0								
nop	1	0								
rrmovq rA, rB	2	0	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	
OPq rA, rB	6	fn	rA	rB						
jXX Dest	7	fn							Dest	
cmovXX rA, rB	2	fn	rA	rB						
call Dest	8	0							Dest	
ret	9	0								
pushq rA	A	0	rA	F						
popq rA	B	0	rA	F						

Not in CS:APP:  
iotrap id

C	id
---	----

Trap IDs:	
charout	0
charin	1
decout	2
decin	3
strout	4
flush	5

Registers:			
%rax <sup>+</sup>	0	%rbp <sup>*</sup>	5
%rcx <sup>+</sup>	1	%rsi <sup>+</sup>	6
%rdx <sup>+</sup>	2	%rdi <sup>+</sup>	7
%rbx <sup>*</sup>	3	%r8-%r11 <sup>+</sup>	
%rsp	4	%r12-%r14 <sup>*</sup>	
<sup>+</sup> caller-save		<sup>*</sup> callee-save	

Args:
%rdi
%rsi
%rdx
%rcx
%r8
%r9

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	

CC:  
Condition  
codes

ZF	SF	OF
----	----	----

PC

--

Stat: Program status

--

DMEM: Memory

--

Operations

addq	6	0
subq	6	1
andq	6	2
xorq	6	3

Branches

jmp	7	0	jne	7	4
jle	7	1	jge	7	5
jl	7	2	jg	7	6
je	7	3			

Moves

rrmovq	2	0	cmovne	2	4
cmovle	2	1	cmovge	2	5
cmovl	2	2	cmovg	2	6
cmove	2	3			

# 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					
OPq rA, rB	6	fn	rA	rB						

Instruction	Byte offset from PC									
	0	1	2	3	4	5	6	7	8	
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						
iotrap id	c	id								

cmovXX:	OPq:	jXX:	Trap IDs:	Registers:	Args:
rrmovq 20	addq 60	jmp 70	charout 0	%rax <sup>+</sup> 0	%rdi
cmovle 21	subq 61	jle 71	charin 1	%rcx <sup>+</sup> 1	%rsi
cmovl 22	andq 62	jl 72	decout 2	%rdx <sup>+</sup> 2	%rdx
cmove 23	xorq 63	je 73	decin 3	%rbx <sup>*</sup> 3	%rcx
cmovne 24		jne 74	strout 4	%r8-%r11 <sup>+</sup>	%r8
cmovge 25		jge 75	flush 5	%rsp 4	%r12-%r14 <sup>*</sup>
cmovg 26		jg 76		<sup>+</sup> caller-save <sup>*</sup> callee-save	%r9

0x100:		.pos 0x100 code
0x100:		_start:
0x100: 30f107000000000000000000		irmovq \$7, %rcx
0x10a: 30f002000000000000000000		irmovq \$2, %rax
0x114: 6010		addq %rcx, %rax
0x116: 00		halt

# Y86 utilities

- Run this script on stu: `/cs/students/cs261/y86/install.sh`
  - **yas**: Y86-64 assembler
    - `"yas yourfile.y8" to assemble code into object files (.yo and .o)`
  - **y86ref**: compiled reference solution for interpreter
    - `"y86ref -d yourfile.o" to disassemble object files (P3)`
    - `"y86ref -e yourfile.o" to execute object files (P4, use "-E" for trace mode)`
  - **ysim**: CS:APP hardware simulator (runs .yo files)
    - `"ysim yourfile.yo" to run the simulator`
    - Use `"-g"` option for visual mode (must have X11 forwarding enabled; use `"ssh -Y"`)
  - These will help with P3/P4: learn to use them!
- 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

**Caution: neither simulator supports the `iotrap` instruction!**

# 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>
```

# Differences from textbook

- Generally, Y86 uses the **base + displacement** addressing mode for memory operands
  - Offset (D) + base (rA or rB)
    - Example: `rmmovq %rax, 0x400(%rcx)`
    - Encoding: `40 01 00 04 00 00 00 00 00 00`
  - **Absolute** addressing is possible with rB = 15 (NOREG)
    - Example: `rmmovq %rax, 0x400`
    - Encoding: `40 0f 00 04 00 00 00 00 00 00`
  - **Indirect** addressing is also possible with D = 0
    - Example: `rmmovq %rax, (%rcx)`
    - Encoding: `40 01 00 00 00 00 00 00 00 00`

# 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]
    halt
```

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



# 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 my_str, %rsi        # rsi = &my_str
    iotrap 4                   # strout
    iotrap 5                   # flush
    halt
```

```
.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 0x600 rodata  
    # YOUR DATA GOES HERE
```

```
.pos 0xf00 stack  
_stack:
```