CS 261
Fall 2017

Mike Lam, Professor

Y86-64 Introduction

Projects 3 & 4: Support Utilities

- Run this script: `/cs/students/cs261/y86/install.sh`
  - **`yas`**: Y86-64 assembler (.ys → .yo and .o)
  - **`yref`**: 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 X forwarded enabled; use “ssh -X”)
- 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 `yref`/`ysim` over this one

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						

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	

Operations			Branches				Moves					
addq	6	0	jmp	7	0	jne	7	4	rrmovq	2	0	cmovne	2	4
subq	6	1	jle	7	1	jge	7	5	cmovle	2	1	cmovge	2	5
andq	6	2	jl	7	2	jg	7	6	cmovl	2	2	cmovg	2	6
xorq	6	3	je	7	3				cmove	2	3			

CC: Condition codes

ZF	SF	OF

Stat: Program status

PC

DMEM: Memory

Differences from textbook

- Execution begins at "entry point" from MiniELF, not address zero
  - Question: is there a significance to "address zero"?
  - 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 segments
  - Retrieve address (i.e., create pointer) using labels and `irmovq`
  - 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 0x300 data
vals:
    .quad 1
    .quad 2
    .quad 3
    .quad 4
```

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
- Write a function that adds any two numbers
  - Use standard x86 calling conventions
    - (params in `%rdi` and `%rsi`, return in `%rax`)
  - Include driver code that calls the function
  - Don't forget to set up the stack!

Template

```
.pos 0 code
    jmp _start

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

.pos 0xf00 stack
_stack:
```