Machine-Level Programming: Y86-64

CSCI 237: Computer Organization
20th Lecture, Friday, October 25

Kelly Shaw

1

Administrative Details

- Read CSAPP Ch. 4.2
- Colloquium Friday at 2:35pm in Wege
  - Info about winter study and spring classes
- Fill out TA Feedback Forms by today
- Apply to be a TA by today

2

Last Time: Structs and Y86-64 ISA

- Structures (Ch 3.9)
  - Allocation
  - Access
  - Alignment
- Y86-64 Instruction Set Architecture
  - Similar state and instructions as x86-64
  - Simpler encodings

3

Today: Y86-64 ISA and RISC vs. CISC

- Y86-64 Instruction Set Architecture
  - Similar state and instructions as x86-64
  - Simpler encodings
- Compiling and simulating Y86-64 code

4

Ch 4 Processor Architecture Overview

- Background
  - Instruction sets
  - Logic design
- Sequential Implementation
  - A simple, but not very fast processor design
- Pipelining
  - Several overlapping tasks running simultaneously
- Pipelined Implementation
  - Making it work in the face of "hazards"

5

5

Coverage

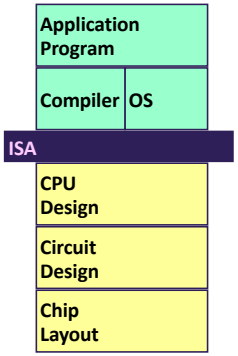
- Our Approach
  - Work through designs for particular instruction set
    - Y86-64 – a simplified (gentler) version of the Intel x86-64 ISA
  - Work at "micro-architectural" level
    - Assemble basic hardware blocks into overall processor structure
      - Memories, functional units, etc.
    - Surround by control logic to make sure each instruction flows through properly
  - Use simple hardware description language to describe control logic
    - Can extend and modify
    - Test via simulation

6

6

Ch 4.1 - Instruction Set Architecture

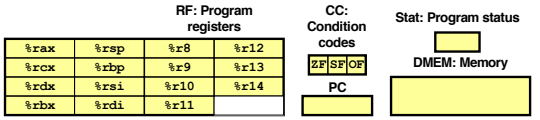
- Assembly Language View
  - Processor state
    - Registers, memory, …
  - Instructions
    - `addq`, `pushq`, `ret`, …
    - How instructions are encoded as bytes
- Layer of Abstraction
  - Above: how to program machine
    - Processor executes instructions in a sequence
  - Below: what needs to be built
    - Use variety of tricks to make it run fast
    - E.g., execute multiple instructions simultaneously



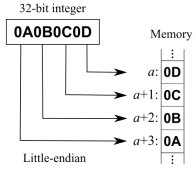
7

7

Y86-64 Processor State



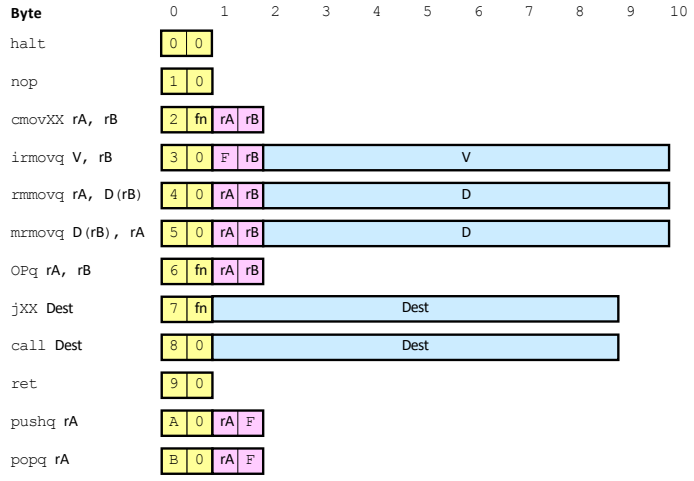
- Program Registers
  - 15 registers (omit `%r15`). Each 64 bits.
- Condition Codes
  - Single-bit flags set by arithmetic or logical instructions
    - ZF: Zero SF: Negative OF: Overflow
- Program Counter
  - Indicates address of next instruction
- Program Status
  - Indicates either normal operation or some error condition
- Memory
  - Byte-addressable storage array
  - Words stored in little-endian byte order



8

8

Y86-64 Instruction Set



9

9

Y86-64 Instructions

- Format
  - 1–10 bytes of information read from memory
    - Can determine instruction length from first byte
    - Not as many instruction types, and simpler encoding than with x86-64
  - Each accesses and modifies some part(s) of the program state

10

10

Encoding Registers

- Each register has 4-bit ID

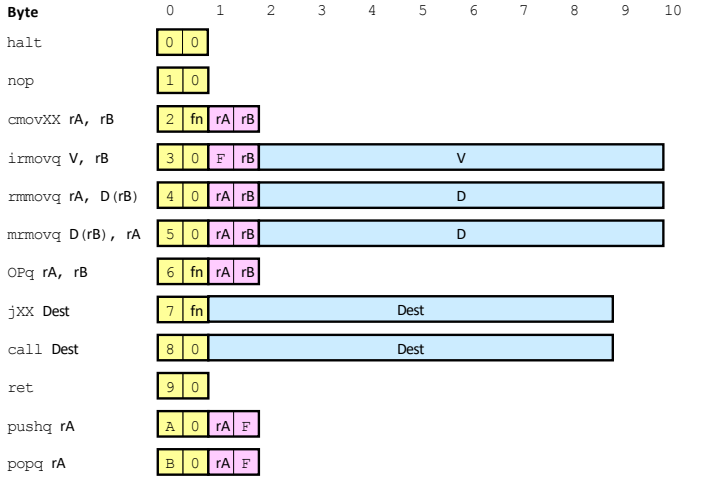
%rax	0	%r8	8
%rcx	1	%r9	9
%rdx	2	%r10	A
%rbx	3	%r11	B
%rsp	4	%r12	C
%rbp	5	%r13	D
%rsi	6	%r14	E
%rdi	7	No Register	F

  - Same encoding as in x86-64
- Register ID 15 (0xF) indicates "no register"
  - Will use this in our hardware design in multiple places

11

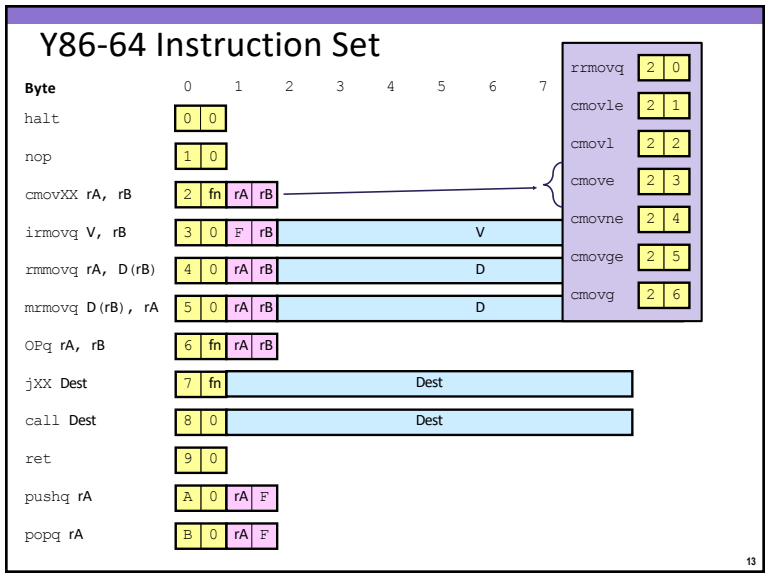
11

Y86-64 Instruction Set

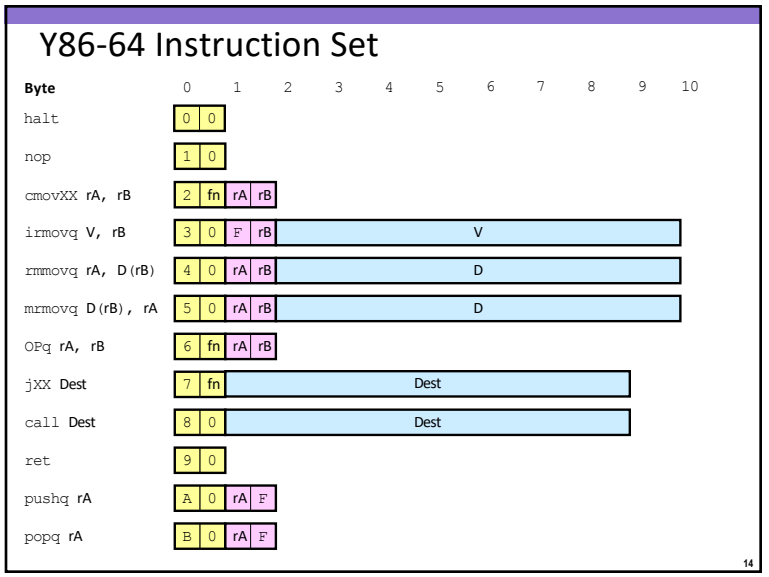


12

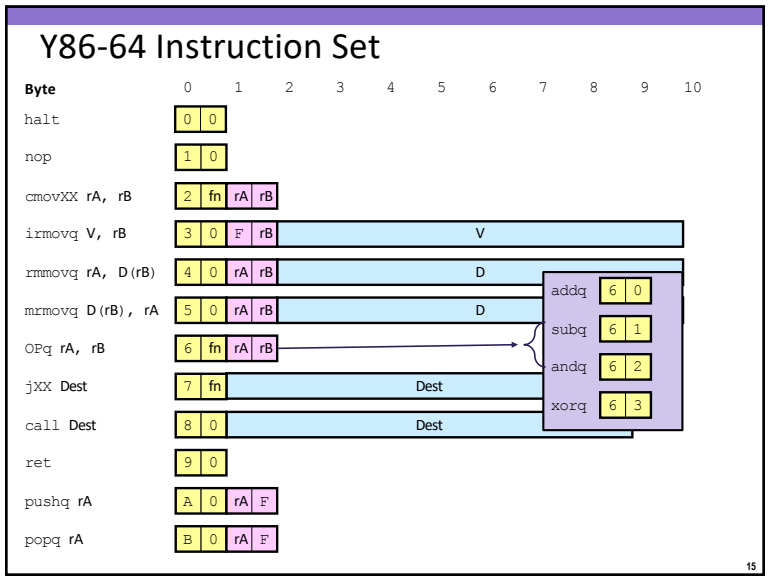
12


Y86-64 Instruction Set
Byte
0 1 2 3 4 5 6 7
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
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
rrmovq 2 0
cmovle 2 1
cmovl 2 2
cmove 2 3
cmovne 2 4
cmovge 2 5
cmovg 2 6
13

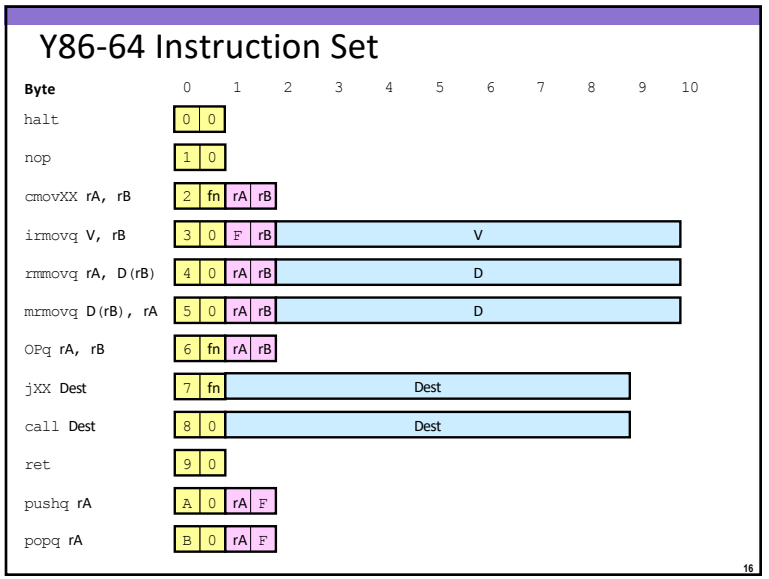
13


Y86-64 Instruction Set
Byte
0 1 2 3 4 5 6 7 8 9 10
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
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
14

14

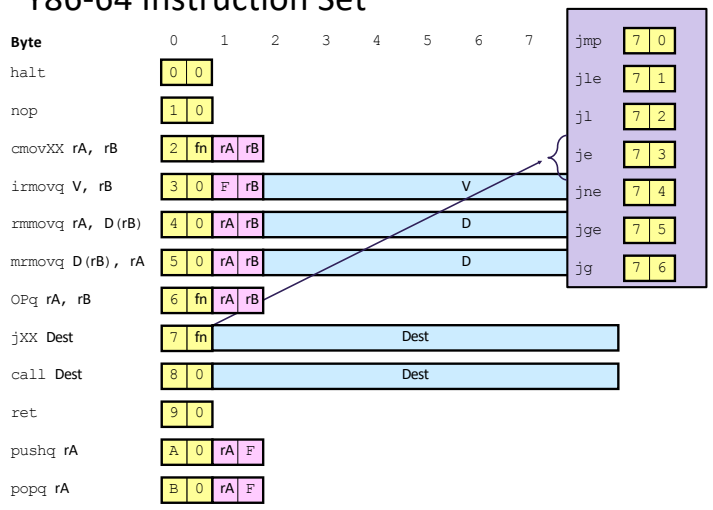

Y86-64 Instruction Set
Byte
0 1 2 3 4 5 6 7 8 9 10
halt 0 0
nop 1 0
cmovXX rA, rB 2 fn rA rB
irmovq V, rB 3 0 F rB V
rmmovq rA, D(rB) 4 0 rA rB D
mrmovq D(rB), rA 5 0 rA rB D
OPq rA, rB 6 fn rA rB
jXX Dest 7 fn Dest
call Dest 8 0 Dest
ret 9 0
pushq rA A 0 rA F
popq rA B 0 rA F
addq 6 0
subq 6 1
andq 6 2
xorq 6 3
15

15


Y86-64 Instruction Set
Byte
0 1 2 3 4 5 6 7 8 9 10
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
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
16

16

Y86-64 Instruction Set

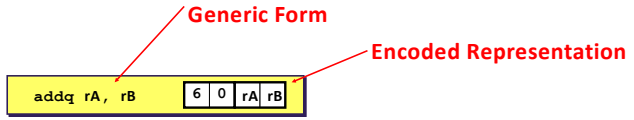


17

17

Instruction Example

- Addition Instruction

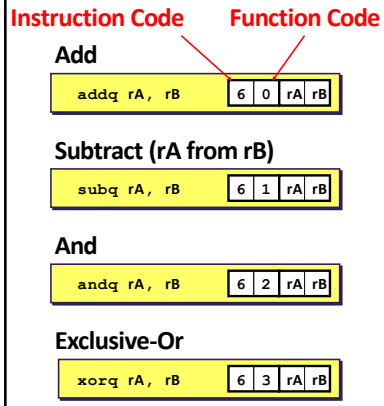


- Add value in register rA to that in register rB
  - Store result in register rB
  - Note that Y86-64 **only allows addition to be applied to register data**
- Set condition codes based on result
- e.g., `addq %rax,%rsi` Encoding: `60 06`
- Two-byte encoding
  - First indicates instruction type
  - Second gives source and destination registers

18

18

Arithmetic and Logical Operations

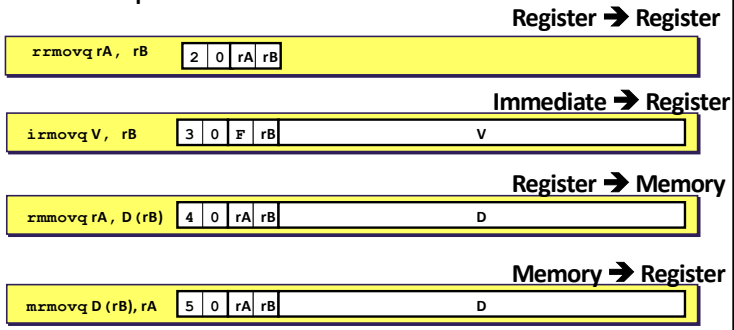


- Refer to generically as "`OPq`"
- Encodings differ only by "function code"
  - Low-order 4 bits in first instruction word
- Set condition codes as side effect

19

19

Move Operations



- Like the x86-64 `movq` instruction
- Simpler format for memory addresses
- Give different names to keep them distinct

20

20

Move Instruction Examples

%rax	0	%r8	8
%rcx	1	%r9	9
%rdx	2	%r10	A
%rbx	3	%r11	B
%rsp	4	%r12	C
%rbp	5	%r13	D
%rsi	6	%r14	E
%rdi	7	No Register	F

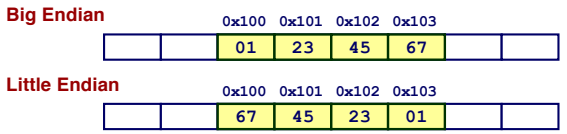
X86-64	Y86-64
`movq $0xabcd, %rdx`	`irmovq $0xabcd, %rdx`
Encoding: `30 F2 cd ab 00 00 00 00 00 00`	
`movq %rsp, %rbx`	`rrmovq %rsp, %rbx`
Encoding: `20 43`	
`movq -12(%rbp),%rcx`	`mrmovq -12(%rbp),%rcx`
Encoding: `50 15 f4 ff ff ff ff ff ff ff`	
`movq %rsi,0x41c(%rsp)`	`rmmovq %rsi,0x41c(%rsp)`
Encoding: `40 64 1c 04 00 00 00 00 00 00`	

21

21

Review: Byte Ordering Example

- Example
  - Let variable x have 4-byte value of 0x01234567
  - The address given by `&x` is 0x100



Big Endian: Least significant byte has highest address
Little Endian: Least significant byte has lowest address

22

22

Conditional Move Instructions

Move Unconditionally	`rrmovq rA, rB`	2	0	rA rB
Move When Less or Equal	`cmovle rA, rB`	2	1	rA rB
Move When Less	`cmovl rA, rB`	2	2	rA rB
Move When Equal	`cmove rA, rB`	2	3	rA rB
Move When Not Equal	`cmovne rA, rB`	2	4	rA rB
Move When Greater or Equal	`cmovge rA, rB`	2	5	rA rB
Move When Greater	`cmovg rA, rB`	2	6	rA rB

- Refer to generically as "`cmovXX`"
- Encodings differ only by "function code"
- Based on values of condition codes
- Variants of `rrmovq` instruction
  - (Conditionally) copy value from source to destination register

23

23

Jump Instructions

Jump (Conditionally)

`jXX Dest`	7	fn	Dest

- Refer to generically as "`jXX`"
- Encodings differ only by "function code" fn
- Based on values of condition codes
- Same as x86-64 counterparts
- Encode full destination address
  - Unlike PC-relative addressing seen in x86-64

24

24

Jump Instructions

Jump Unconditionally

jmp Dest	7	0	Dest

Jump When Less or Equal

jle Dest	7	1	Dest

Jump When Less

jl Dest	7	2	Dest

Jump When Equal

je Dest	7	3	Dest

Jump When Not Equal

jne Dest	7	4	Dest

Jump When Greater or Equal

jge Dest	7	5	Dest

Jump When Greater

jg Dest	7	6	Dest

25

25

Stack Operations

pushq rA	A	0	rA	F

- Decrement `%rsp` by 8
- Store word from rA to memory at `%rsp`
- Like x86-64

popq rA	B	0	rA	F

- Read word from memory at `%rsp`
- Save in rA
- Increment `%rsp` by 8
- Like x86-64

26

26

Procedure Call and Return

call Dest	8	0	Dest

- Push address of next instruction onto stack
- Start executing instructions at Dest
- Like x86-64

ret	9	0

- Pop value from stack
- Use as address for next instruction
- Like x86-64

27

27

Miscellaneous Instructions

nop	1	0

- Don't do anything

halt	0	0

- Stop executing instructions
- x86-64 has comparable instruction, but can't execute it in user mode
- We will use it to stop the simulator
- Encoding ensures that program hitting memory initialized to zero will halt

28

28

Program Status Conditions (Stat)

Mnemonic	Code
AOK	1

- Normal operation

Mnemonic	Code
HLT	2

- Halt instruction encountered

Mnemonic	Code
ADR	3

- Bad address (either instruction or data) encountered
  - Exception

Mnemonic	Code
INS	4

- Invalid instruction encountered
  - Exception

- Desired Behavior
  - If AOK, keep going
  - Otherwise, stop program execution

29

29

Y86-64 Sample Program Structure #1

```
init:           # Initialization
   . . .
   call Main
   halt

   .align 8     # Program data
array:
   . . .

Main:           # Main function
   . . .
   call len
   . . .

len:            # len function

   .pos 0x100   # Placement of stack

Stack:
   . . .
```

- Must set up stack
  - Where located
  - Pointer values
  - Make sure don't overwrite code!
- Must initialize data array

32

32

Y86-64 Program Structure #2

```
init:
     .pos 0                 # Start at address 0
     irmovq Stack, %rsp     # Set up stack pointer
     call Main              # Execute main program
     halt                   # Terminate

# Array of 4 elements + terminating 0
     .align 8
array:
     .quad 0x000d000d000d000d
     .quad 0x00c000c000c000c0
     .quad 0x0b000b000b000b00
     .quad 0xa000a000a000a000
     .quad 0
```

- Program starts at address 0 (.pos 0)
- Must set up stack and stack pointer (.pos 0x100)
- Must initialize data array
- Can use symbolic names

33

33

Y86-64 Program Structure #3

```
Main:
     irmovq array,%rdi
     call len           #len(array)
     ret
```

- Set up call to len
  - Follow x86-64 procedure conventions
  - Put array address as argument

34

34

Assembling Y86-64 Program

```
unix> yas len.ys
```

- Generates "object code" file `len.yo`
  - Actually looks like disassembler output

```
0x054:                      | len:
0x054: 30f80100000000000000 |   irmovq $1, %r8          # Constant 1
0x05e: 30f90800000000000000 |   irmovq $8, %r9          # Constant 8
0x068: 30f00000000000000000 |   irmovq $0, %rax         # len = 0
0x072: 5027000000000000000  |   mrmovq (%rdi), %rdx     # val = *a
0x07c: 6222                 |   andq %rdx, %rdx         # Test val
0x07e: 73a000000000000000   |   je Done                 # If zero, goto Done
0x087:                      | Loop:
0x087: 6080                 |   addq %r8, %rax          # len++
0x089: 6097                 |   addq %r9, %rdi          # a++
0x08b: 5027000000000000000  |   mrmovq (%rdi), %rdx     # val = *a
0x095: 6222                 |   andq %rdx, %rdx         # Test val
0x097: 748700000000000000   |   jne Loop                # If !0, goto Loop
0x0a0:                      | Done:
0x0a0: 90                   |   ret
```

35

35

Simulating Y86-64 Program

```
unix> yis len.yo
```

- Instruction set simulator
  - Computes effect of each instruction on processor state
  - Prints changes in state from original

```
Stopped in 33 steps at PC = 0x13.  Status 'HLT', CC Z=1 S=0 O=0
Changes to registers:
%rax:   0x0000000000000000      0x0000000000000004
%rsp:   0x0000000000000000      0x0000000000000100
%rdi:   0x0000000000000000      0x0000000000000038
%r8:    0x0000000000000000      0x0000000000000001
%r9:    0x0000000000000000      0x0000000000000008

Changes to memory:
0x00f0: 0x0000000000000000      0x0000000000000053
0x00f8: 0x0000000000000000      0x0000000000000013
```

36

36

Debugging Tips

- Insert halt instructions to see the values in registers and memory at certain points in your code
  - Harder to do in loops, but you can.
    - Load the loop index you want to examine into a register
    - If that value you loaded is equal to the current loop counter register, jump to a label that just has a halt instruction
- Don't use labels more than once in a file.
- For problem 4, you probably need to put the position of the stack to be at a value larger than 0x100 (e.g., 0x400) because you'll have a lot of instructions

37

37