

# 18-600: Recitation #5

## Arch Lab (Y86-64 Basics)

27th September 2016

# Agenda

- Go over important points for Arch Lab
- Y86-64
  - Introduction
  - Registers
  - Instructions
  - Instruction Encoding
  - An example

# Why learn processor design?

- Aid the learning of how the overall computer system works
- Many real-world systems like automobiles and appliances have embedded processors

# Y86-64: Introduction

- A simple instruction set inspired by x86-64
- Y86-64 processor based on sequential operation  
i.e. executes an instruction on every clock cycle
- Has fewer data types, and instructions in  
comparison to x86-64

# Registers (or visible states)

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

- 15 program registers (16 in x86-64) with 4-bit ID
- Each store a 64-bit word
- %rsp is used as a stack pointer by push, pop, call, and return instructions; others don't have a fixed purpose

# Registers

- Condition Codes (CC): Set by most recent arithmetic or logical instruction

|               |               |                   |
|---------------|---------------|-------------------|
| ZF: Zero Flag | SF: Sign Flag | OF: Overflow Flag |
|---------------|---------------|-------------------|

- PC: Program Counter holds address of the instruction currently being executed
- Stat: Program Status; Indicates the overall state of the program; either normal or some sort of exception occurred
- Memory: Large array of bytes; stored in little-endian byte order

# mov Instructions

- `irmovq`: immediate to register (ex: `irmovq $18600, %r8`)
- `rrmovq`: register to register (ex: `rrmovq %r8, %r9`)
- `mrmovq`: memory to register (ex: `mrmovq (%rbp), %rdx` )
- `rmmovq`: register to memory (ex: `rmmovq %rdx, (%rbp)`)

# Integer & Jump Instructions

## ■ Integer

- addq
- subq
  - subq rA, rB: subtract rA from rB
- andq
- xorq

Integer operation instructions set the three condition codes (ZF, SF, and OF)

## ■ Jump

- jmp
- jle
- jl
- je
- jne
- jge
- jg



# Conditional Move Instructions

- `cmovle`
- `cmovl`
- `cmove`
- `cmovne`
- `cmovge`
- `cmovg`

Destination register updated only when condition code satisfied

# Other Instructions

- `call`: pushes the return address on stack, and jumps to the destination address
- `pushq` and `popq`: similar to x86-64
- `halt`: stops instruction execution

# Instruction Encoding

Initial byte of any instruction is used for identification

Byte split into two 4-bit parts:

1. High-order or instruction code
2. Low-order or function code

|      |   |   |     |   |   |     |   |   |        |   |   |        |   |   |
|------|---|---|-----|---|---|-----|---|---|--------|---|---|--------|---|---|
| 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 |        |   |   |

# Determine the byte code

```
.pos 0x100
    irmovq $15,%rbx
    rrmovq %rbx,%rcx
loop:
    rmmovq %rcx,-3(%rbx)
    addq   %rbx,%rcx
    jmp   loop
```

Let's try it out on shark...

# Exceptions

| Name | Meaning             |
|------|---------------------|
| AOK  | Normal operation    |
| HLT  | halt instruction    |
| ADR  | Invalid address     |
| INS  | Invalid instruction |

# Program Structure

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

    .align 8                          # Program data
array:
    . . .

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

len:                                  # Length function
    . . .

    .pos 0x100                        # Placement of stack
Stack:
```

- Program starts at address 0
- Must set up stack
- Must Initialize data

# Example program: asum.ys

```

# Execution begins at address 0
.pos 0
irmovq stack, %rsp      # Set up stack pointer
call main               # Execute main program
halt                   # Terminate program

# Array of 4 elements
.align 8
array: .quad 0x000d000d000d
       .quad 0x00c000c000c0
       .quad 0x0b000b000b00
       .quad 0xa000a000a000

main:  irmovq array,%rdi
       irmovq $4,%rsi
       call sum          # sum(array, 4)
       ret

# long sum(long *start, long count)
# start in %rdi, count in %rsi
sum:   irmovq $8,%r8      # Constant 8
       irmovq $1,%r9      # Constant 1
       xorq %rax,%rax     # sum = 0
       andq %rsi,%rsi     # Set CC
       jmp test          # Goto test
loop:  mrmovq (%rdi),%r10  # Get *start
       addq %r10,%rax     # Add to sum
       addq %r8,%rdi      # start++
       subq %r9,%rsi      # count--. Set CC
test:  jne loop          # Stop when 0
       ret               # Return

# Stack starts here and grows to lower addresses
.pos 0x200
stack:

```

# Running the Simulator

1. `./yas asum.js` - to generate the object code file `asum.yo`
2. `./yis asum.yo` - to run the simulator



# Archlab Part B

- **Modify `ncopy.hs` and `pipe-full.hcl` with the goal of making `ncopy` run as fast as possible**
- **Optimize y86 assembly as well as pipeline code for efficiency**
- **`ncopy` must work for arbitrary array sizes**
- **More on this next week....**

# Questions?