X86_64

CS 2130: Computer Systems and Organization 1

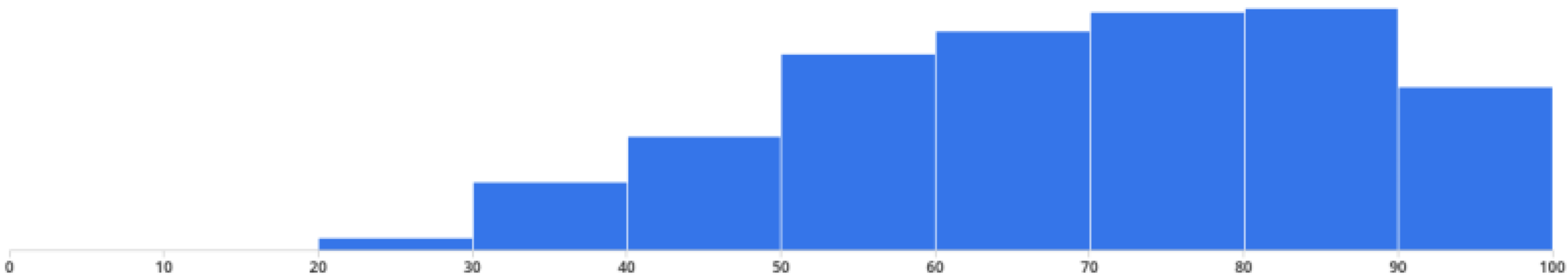
Xinyao Yi Ph.D.
Assistant Professor

UNIVERSITY of VIRGINIA | ENGINEERING

Announcements

- Homework 5 available today, **due Monday at 11:59pm** on Gradescope
- Exam 1 scores released

Exam 1 Scores



	Mean	Median
Overall	68.92	70.0
Suspected Gen AI use on HW3	64.55	63.0
No suspected Gen AI use on HW3	70.18	73.0

AT&T x86-84 Assembly

```
instruction source, destination
```

- Instruction followed by 0 or more operands (arguments)

  (typically we will not see more than 2)

- 4 types of operands:
  - Number (immediate value): `$0x123`
  - Register: `%rax`
  - Address of memory: `(%rax)` or 24 or `labelname`
  - Value at an address in memory: `(%rax)` or 24 or `labelname`

lea

loading the addresses

In most of the cases, we are doing something using the value. Except for

AT&T x86-84 Assembly

`mylabelname:` end with a colon

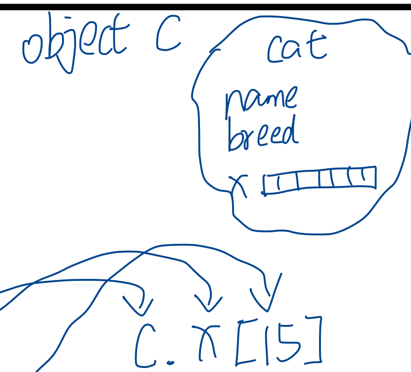
- Label - remember the address of next thing to use later

`.something something` start with a dot

- Metadirective - extra information that is not code
- How the code works with other things (i.e., talk to OS)
- Ex: `.globl main`

`// we can have comments!`

Addressing Memory

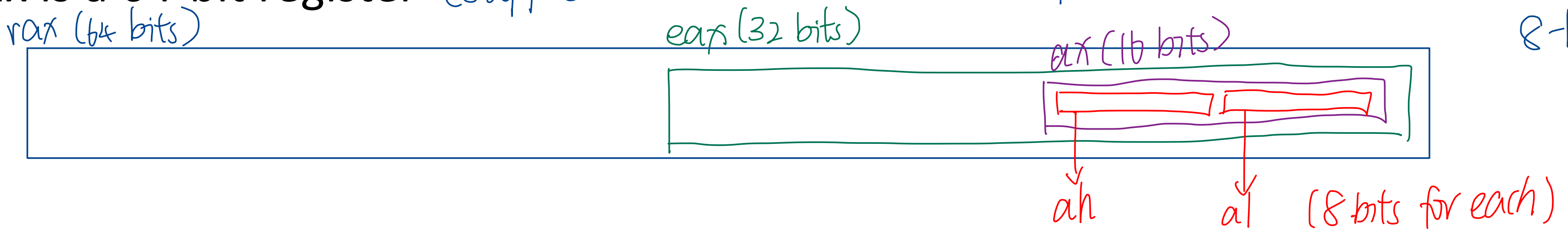


`2130(%rax, %rsp, 8)`

- Address can have up to 4 parts: 2 numbers, 2 registers
- Combines as: $\boxed{\texttt{2130 + \%rax + (\%rsp * 8)}}$
- Common usage from this example:
  - `rax` - address of an object in memory
  - `2130` - offset of an array into the object
  - `rsp` - index into the array
  - `8` - size of the values in the array (used to calculate the offset)
- Don't need all parts: `(%rax)` or `(%rax, 4)` or `4(%rax)` If I don't have all the pieces, it will calculate what it can.
- This is all one operand (one memory address)

Registers

rax is a 64-bit register (supposed to be backwards compatible with x86 (32-bit), 16-bit, 8-bit)



If I look at 32-bit version, it will just zero out the top 32 bits.

We'll see this with all our registers, in slightly different way.

(Check the reading)

Instructions (short acronyms for what we want to do, like mov, add, and, or, xor, neg)

Instructions have different versions depending on number of bits to use

- `movq` - 64-bit move (similar for addq, subq)
  - q = quad word → The instruction followed by how wide of the thing we want to do.
- `movl` - 32-bit move
  - l = long
- There are encodings for shorter things, but we will mostly see 32- and 64-bit

UNIVERSITY of VIRGINIA

More powerful than our ISA

Instructions can move/operate between memory and register

- `movq %rax, %rcx` - register to register
  - Remember our icode 0
- `movq (%rax), %rcx` - memory to register
  - Remember our icode 3
- `movq %rax, (%rcx)` - register to memory
  - Remember our icode 4
- `movq $21, %rax` - Immediate to register
  - Remember our icode 6 (b=0)

Note: at most one memory address per instruction

We cannot do memory to memory calculations.

Other Instructions

Other instructions work the same way

- src dest
  `addq %rax, %rcx` — rcx += rax
- `subq (%rbx), %rax` — rax -= M[rbx]
  going to memory and get the value
- xor, and, and others work the same way!
- Assembly has virtually no 3-argument instructions
  - All will be modifying something (i.e., +=, &=, ...)
    modify one of the inputs directly. doesn't have a seperate output.

Load Effective Address

I'm not going to the memory; "lea" is a special instruction that calculates the memory address and store the memory address itself in a register.

Load effective address: `leaq 4(%rcx), %rax`

- Performs memory address calculation
- Stores address, not value at the address in memory

%rax = %rcx + 4.

Jumps

`jmp foo`

- Unconditional jump to `foo`
- `foo` is a label or memory address
- Need `jmp*` to use register value (jump to a value in a register)

Conditional jumps

- `jl,` `jle,` `je,` `jne,` `jg,` `jge,` `ja,` `jb,` `js,` `jo`

  <   <=   ==   !=   >   >=   above   below

  unsigned

  If there's a overflow

  If the assigned bit is set.

Unlike our Toy ISA, these do not compare given register to 0

Jumps

We jump based on the result of some special registers called condition codes.

Condition codes - 4 1-bit registers set by every math operation, `cmp`, and `test`

- Result for the operation compared to 0 (if no overflow)
- Example:

```
addq $-5, %rax
// ...code that doesn't set condition codes...
je foo
```

They don't have to be back to back.

→ You can do something like move things around.

jump will be based on the most recent thing that set the condition code.

  - Sets condition codes from doing math (subtract 5 from rax)
  - Tells whether result was positive, negative, 0, if there was overflow, ...
  - Then jump if the result of operation should have been = 0

Jumps: compare…

```
cmpq %rax, %rdx
```

- Compare checks result of -= and sets condition codes
- How `rdx - rax` compares with 0
- Be aware of ordering!
  - if `rax` is bigger, sets < flag
  - if `rdx` is bigger, sets > flag

Jumps: … and test

`testq %rax, %rdx`

- Sets the condition codes based on `rdx & rax`
- Less common

Neither save their result, just set condition codes!

test could be used to check if a register has 0 in it.

```
testq %rax, %rax
je   zero_case      // if rax==0
jne  nonzero_case   // if rax!=0
```

Example: Loops

```
while (i < 10)
    i += 1
```

```
top: ← lable
    // check !condition , jump out
    if(i>=10) goto end
        i+=1 ;
    // jump back to condition
    go to top ;
end : ← lable
```

```
main:  → lable
    movq $0, %rax    // we set rax=0 for int i=0;
loop:
    cmp $10, %rax    // rax-10 = ?   { if rax<10, we got negtive, then do loop body.
                                       if rax >=10, we got positive or 0, then jump out the loop.
    jge after
    addq $1, %rax
    jmp loop
after:
    retq    // return with a "q" because we're working with a 64 bit thing. (pop a 8-byte address and jump back to caller)
```

Example: Functions

f(x,y):
    …
    …
    return 4

…
z = f(2,5)

```
int f(int x, int y) {
    return 4;
}

int main() {
    int z = f(2,5);
}
```

UNIVERSITY of VIRGINIA

Function Calls: Calling Conventions

The function I'm running currently and the function that I call are both sharing the same registers.

`callq myfun`

- Push return address, then jump to myfun
- Convention: Store arguments in registers and stack before call
  - First 6 arguments (in order): `rdi`, `rsi`, `rdx`, `rcx`, `r8`, `r9`
  - If more arguments, pushed onto stack (last to first)

`retq`

- Pop return address from stack and jump back
- Convention: store return value in `rax` before calling `retq`

This is similar to our Toy ISA's function calls in homework 4

More conventions, check readings.

Calling Conventions: Registers

Why? Caller and callee share the same registers.

Calling conventions - recommendations for making function calls

- Where to put arguments/parameters for the function call?
- Where to put return value? in `rax` before calling `retq`.
- What happens to values in the registers?
  - **Callee-save** - The function should ensure the values in these registers are unchanged when the function returns

    ① push the old values before calling ②. pop the values before returning.
    - `rbx, rsp, rbp, r12, r13, r14, r15`
  - **Caller-save** - Before making a function call, save the value, since the function may change it

Most Common Instructions

- `mov` - =
- `lea` - load effective address
- `call` - push PC and jump to address
- `add` - +=
- `cmp` - set flags as if performing subtract
- `jmp` - unconditional jump
- `test` - set flags as if performing &
- `je` - jump iff flags indicate == 0
- `pop` - pop value from stack
- `push` - push value onto stack
- `ret` - pop PC from the stack

Debugger

Debugger - step through code!

- You will be using this for lab 7
- Experience seeing results of these instructions step-by-step
- **Please read the x86-64 summary reading before lab!**