X86_64

CS 2130: Computer Systems and Organization 1

Xinyao Yi Ph.D.
Assistant Professor

UNIVERSITY of VIRGINIA | ENGINEERING

Announcements

- Homework 5 **due Monday at 11:59pm** on Gradescope
- Quiz 5 this weekend

Addressing Memory

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

- Address can have up to 4 parts: 2 numbers, 2 registers
- Combines as: `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
- Don't need all parts: `(%rax)` or ~~`(%rax, 4)`~~ or `4(%rax)`
- This is all one operand (one memory address)

invalid.

4(%rax) ⇒ %rax + 4

(%rax, %rbx) ⇒ %rax + %rbx

(%rax, %rbx, 4) ⇒ %rax + %rbx * 4

Load Effective Address

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

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

Jumps

`jmp foo`

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

Conditional jumps

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

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

Jumps

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
```

  - 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!

Example: Loops

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

Function Calls: Calling Conventions

`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

Calling Conventions: 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
    - `rbx`, `rsp`, `rbp`, `r12`, `r13`, `r14`, `r15`
  - **Caller-save** - Before making a function call, save the value, since the function may change it

example for callee-save:

```
pushq %rbx
movq $10, %rbx
popq %rbx
retq
```


example for caller-save:

```
pushq %rdi
callq f
popq %rdi
```

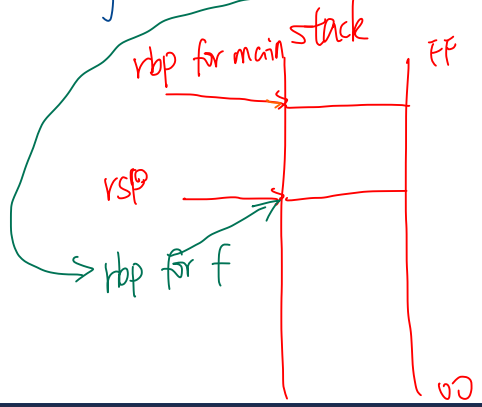
Example: Functions

```
f(x,y):
    ...
    ...
    return 4

...
z = f(2,5)
```

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

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



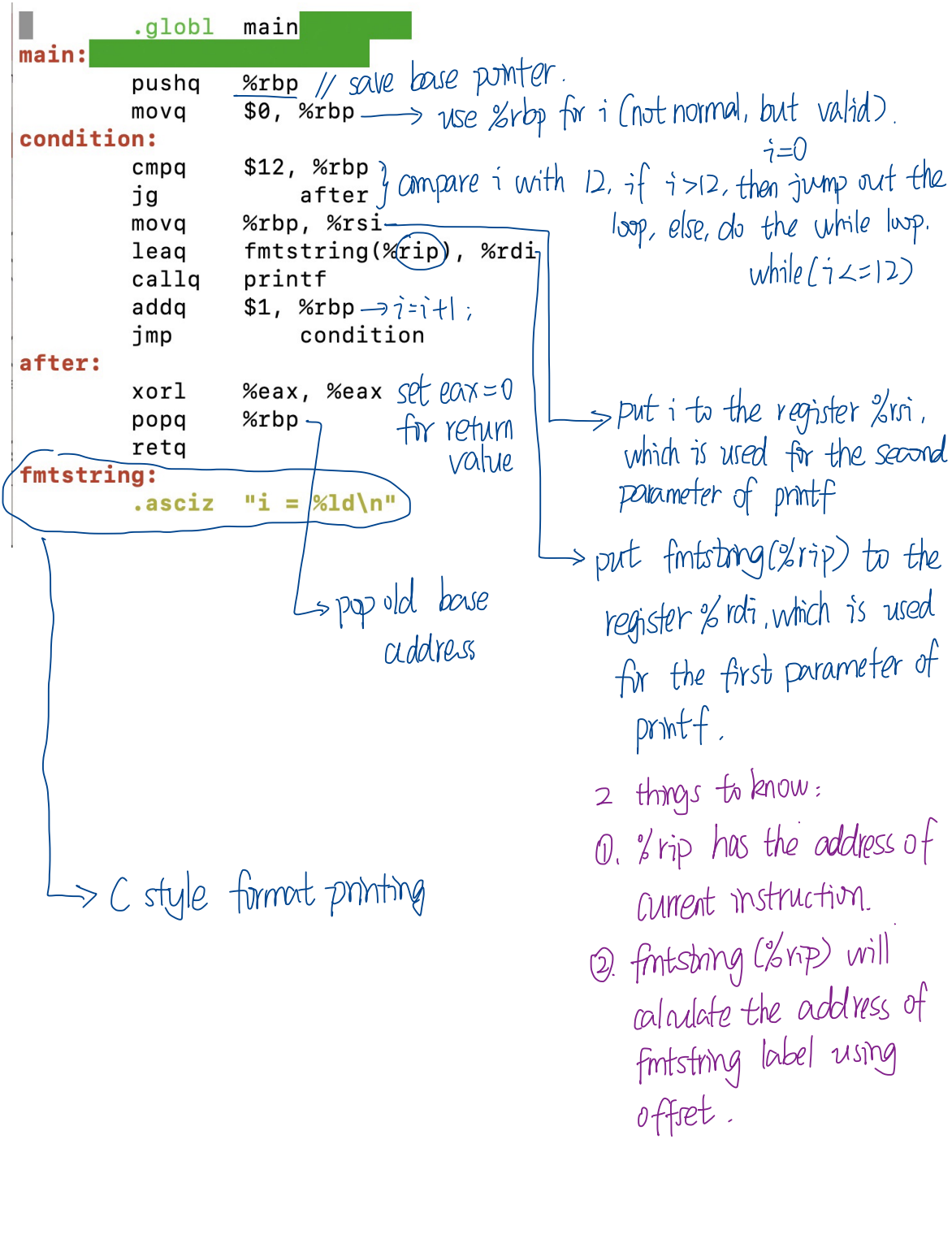

```
.global f
f:
    pushq %rbp          //store old base pointer
    movq %rsp, %rbp     //create a new stack for f
    movl $4, %eax       //put return value 4 to %eax
    popq %rbp           //pop old base pointer (for main)
    retq                //return.
.global main
main:
    pushq %rbp
    movq %rsp, %rbp
    movl $2, %edi       //put parameter 2 to %edi
    movl $5, %esi       //put 5 to %esi
    callq f
    movl %eax, -4(%rbp) //store return value to local variable z.
```

The Stack

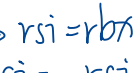
```
pushq %rax
popq %rdx
```

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


.globl main
main:
pushq %rbp // save base pointer.
movq $0, %rbp → use %rbp for i (not normal, but valid).
i=0
condition:
cmpq $12, %rbp } compare i with 12, if i>12, then jump out the
jg after loop, else, do the while loop.
while(i<=12)
movq %rbp, %rsi
leaq fmtstring(%rip), %rdi
callq printf
addq $1, %rbp → i=i+1;
jmp condition
after:
xorl %eax, %eax set eax=0
popq %rbp for return
retq value
fmtstring:
.asciz "i = %ld\n"
put i to the register %rsi, which is used for the second parameter of printf
put fmtstring(%rip) to the register %rdi, which is used for the first parameter of printf.
pop old base address
C style format printing
2 things to know:
① %rip has the address of current instruction.
② fmtstring(%rip) will calculate the address of fmtstring label using offset.

```
        .globl  main
main:
        pushq   %rbp
    movq    $0x42, %rax
    movq    $0x15, %rbx
    movq    %rbx, %rsi
    negq    %rsi
    addq    %rax, %rsi
        leaq    fmtstring(%rip), %rdi
        callq   printf
    xorq    %rax, %rax
        popq    %rbp
        retq
fmtstring:
        .asciz  "%X\n"
```



Debugger

objdump -D substract | less → show by pages.
(tool name → disassemble all sections.)

// disassemble all sections from binary to assembly

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!**

lldb substrat // run the executable file "substract" in debugging mode.

b main // add a breakpoint at the beginning of main function

r / run // run the code

n // run the next instruction

disassemble // disassemble the binary code to assembly code (work for the current frame-main).

re read // read the registers.