Assembly: x86-64

CS 2130: Computer Systems and Organization 1

March 15, 2023

Announcements

- Homework 5 due Monday 3/20 at 11pm
- Prof Hott office hours canceled today and Thursday

Group Example: Loops

Write the following in assembly:

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

example.s

Functions

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



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

Function Calls

`callq myfun`

- Push return address to stack, then jump to myfun

`retq`

- Pop return address from stack and jump back

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

Calling Conventions: Parameters

Calling conventions - recommendations for making function calls

- Where to put arguments/parameters for the function call?
  - First 6 arguments (in order): *`rdi`*, *`rsi`*, *`rdx`*, *`rcx`*, *`r8`*, *`r9`*
  - If more arguments, push onto stack (last to first)
- Where to put return value? in *`rax`* before calling *`retq`*

Calling Conventions: Parameters

What happens to values in the registers since we share 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

The Stack

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

example.s

```
  .globl main
main:
  pushq  %rbp
  movq   $0, %rbp
condition:
  cmpq   $42, %rbp
  jg     after
  movq   %rbp, %rsi
  leaq   fmtstring(%rip), %rdi
  callq  printf
  addq   $1, %rbp
  jmp    condition
after:
  xorl   %eax, %eax
  popq   %rbp
  retq
fmtstring:
  .asciz "i = %ld\n"
```

Compilation Pipeline

Turning our code into something that runs

- **Pipeline** - a sequence of steps in which each builds off the last

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