Assembly: x86-64

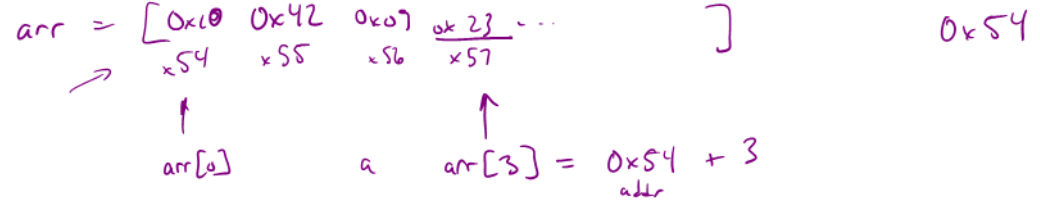
CS 2130: Computer Systems and Organization 1

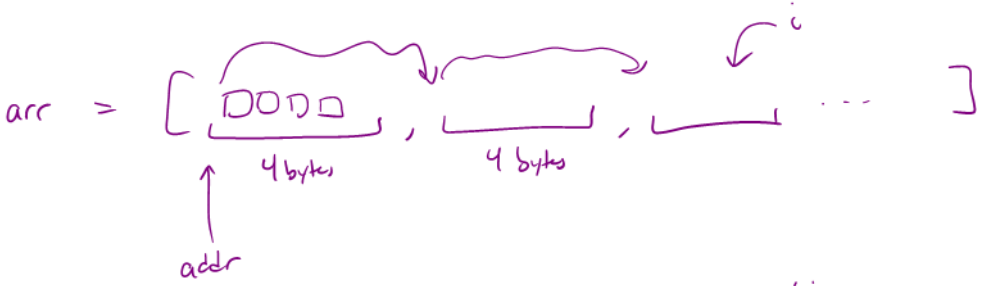
October 12, 2022

Announcements

- Homework 5 due Wednesday 10/19 at 11pm
- Prof Hott office hours tomorrow: 4-6pm

Quiz Review


arr = [0x10 0x42 0x07 0x23 ...]
0x54
0x54
0x55
0x56
0x57
arr[0]
a
arr[3] = 0x54 + 3
addr


arr = [...]
i
4 bytes
4 bytes
addr
arr[i] = addr + 4i

Instructions

Instructions have different versions depending on number of bits to use

- `movq` - 64-bit move
  - `q` = quad word
- `movl` - 32-bit move
  - `l` = long
- There are encodings for shorter things, but we will mostly see 32- and 64-bit

Instruction Operands

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

Other Instructions

Other instructions work the same way

- `addq %rax, %rcx` — rcx += rax
- `subq (%rbx), %rax` — rax -= M[rbx]
- xor, and, and others work the same way!
- Assembly has virtually no 3-argument instructions
  - All will be modifying something (i.e., `+=`, `&=`, …)

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 - 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 that operation should have been = 0

Jumps: compare and test

`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

`testq %rax, %rdx`

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

Neither save their result, just set condition codes!

Example: Loops

i = 0

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

```
top:  <- label
    // check !condition, jump out
    if( i >= 10)  goto end

    i++ = 1
    // jump back to condition
    goto top
end:  <- label
```

i in %rax

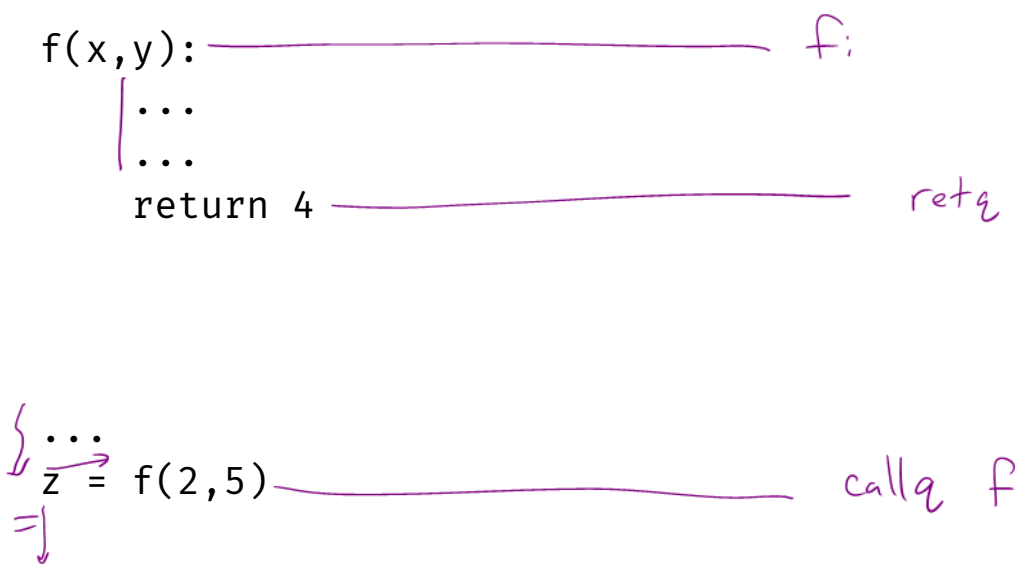
```
mov $0, %rax
```

Assembly

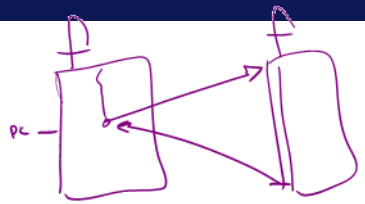
```
top:
    cmpq $10, %rax
    jge end
    addq $1, %rax
    jmp top
end:
```

rax - 10

Functions



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`
- 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

The Stack

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

example.s

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