Computer Systems

CS107

Cynthia Lee
Today’s Topics

Lecture:
- Pop quiz on floating point!
  - Just kidding.
- New: Assembly code

Two friendly reminders:
Midterm

- 2 hours
- 1 page of notes
- Exam also comes with a reference sheet for C library functions

Topics:
  - Integer representation and math
    - Unsigned
    - Two’s complement
  - Bitwise operators
    - “puzzle” problems like HW
  - Pointers and memory
    - Arrays
    - Strings
    - Pointer arithmetic
    - * and & operators, levels of indirection
    - Drawing a memory diagram
  - Generics
    - void*
    - Function pointers
REMINDER:
Everything is bits!
Everything is bits!

- We’ve seen many data types so far:
  - Integers:
    - char/int/long (encoding as unsigned or two’s complement signed)
  - Letters/punctuation:
    - Char (ASCII encoding)
  - Real numbers:
    - float/double (IEEE floating point encoding)
  - Memory addresses:
    - pointer types (unsigned long encoding)
  - **Now a new one.....the code itself!**
    - Instructions (AMD64 encoding)
What happens when we compile our code?

ANATOMY OF AN EXECUTABLE FILE
What happens when we compile our code?

```c
int sum_array(int arr[], int nelems) {
    int sum = 0;
    for (int i = 0; i < nelems; i++) {
        sum += arr[i];
    }
    return sum;
}
```

> make
> ls
Makefile  sum  sum.c
> objdump -d sum
000000000040052d <sum_array>:

40052d:      55                      push %rbp
40052e:      48 89 e5               mov %rsp,%rbp
400531:      48 89 7d e8            mov %rdi,-0x18(%rbp)
400535:      89 75 e4               mov %esi,-0x1c(%rbp)
400538:      c7 45 fc 00 00 00 00    movl $0x0,-0x4(%rbp)
40053f:      c7 45 f8 00 00 00 00    movl $0x0,-0x8(%rbp)
400546:      eb 1d                   jmp 400565 <sum_array+0x38>
400548:      8b 45 f8                mov -0x8(%rbp),%eax
40054b:      48 98                   cltq
40054d:      48 8d 14 85 00 00 00   lea 0x0(%rax,4),%rdx
400554:      00                      
400555:      48 8b 45 e8            mov -0x18(%rbp),%rax
400559:      48 01 d0               add %rdx,%rax
40055c:      8b 00                   mov (%rax),%eax
40055e:      01 45 fc               add %eax,-0x4(%rbp)
400561:      83 45 f8 01            addl $0x1,-0x8(%rbp)
400565:      8b 45 f8               mov -0x8(%rbp),%eax
400568:      3b 45 e4               cmp -0x1c(%rbp),%eax
40056b:      7c db                   jl 400548 <sum_array+0x1b>
40056d:      8b 45 fc               mov -0x4(%rbp),%eax
400570:      5d                      pop %rbp
400571:      c3                      retq

Stanford University
Name of the function (same as in the C code) and the memory address where the code for this function starts
Memory address where
each of line of
instruction is found—
sequential instructions
are found sequentially
in memory
000000000040052d <sum_array>:

Assembly code: “human-readable” version of each instruction

40052d:      55                      push  %rbp
40052e:      48 89 e5                mov  %rsp,%rbp
400531:      48 89 7d e8             mov  %rdi,-0x18(%rbp)
400535:      89 75 e4                mov  %esi,-0x1c(%rbp)
400538:      c7 45 fc 00 00 00 00    movl  $0x0,-0x4(%rbp)
40053f:      c7 45 f8 00 00 00 00    movl  $0x0,-0x8(%rbp)
400546:      eb 1d                    jmp  400565 <sum_array+0x38>
400548:      8b 45 f8                mov  -0x8(%rbp),%eax
40054b:      48 98                   cltq
40054d:      48 8d 14 85 00 00 00    lea    0x0(,%rax,4),%rdx
400554:      00
400555:      48 8b 45 e8             mov  -0x18(%rbp),%rax
400559:      3b 45 e4                add  %rdx,%rax
40055c:      48 8d 14 85 00 00 00    lea    0x0(,%rax,4),%rdx
40055f:      5d                      pop  %rbp
400561:      c3                      retq
Machine code: raw hexadecimal version of each instruction, representing the binary as it would be read by the computer.
Anatomy of an individual instruction
Anatomy of an individual instruction

<table>
<thead>
<tr>
<th>Address</th>
<th>Assembly Instruction</th>
<th>Operation Name</th>
<th>Operands</th>
</tr>
</thead>
<tbody>
<tr>
<td>40052e:</td>
<td>48 89 e5</td>
<td>mov</td>
<td>%rsp,%rbp</td>
</tr>
<tr>
<td>400531:</td>
<td>48 89 7d e8</td>
<td>mov</td>
<td>%rdi,-0x18(%rbp)</td>
</tr>
<tr>
<td>400548:</td>
<td>8b 45 f8</td>
<td>mov</td>
<td>-0x8(%rbp),%eax</td>
</tr>
<tr>
<td>400559:</td>
<td>48 01 d0</td>
<td>add</td>
<td>%rdx,%rax</td>
</tr>
<tr>
<td>400561:</td>
<td>83 45 f8 01</td>
<td>addl</td>
<td>$0x1,-0x8(%rbp)</td>
</tr>
</tbody>
</table>

Operation name (sometimes called “opcode”)  
Operands (like arguments)
Anatomy of an individual instruction

40052e: 48 89 e5
400531: 48 89 7d e8
400548: 8b 45 f8
400559: 48 01 d0
400561: 83 45 f8 01

mov %rsp,%rbp
mov %rdi,-0x18(%rbp)
mov -0x8(%rbp),%eax
add %rdx,%rax
addl $0x1,-0x8(%rbp)

[%[name]] names a register—these are a small collection of memory slots right on the CPU that can hold variables’ values
Anatomy of an individual instruction

```
40052e:  48 89 e5          mov   %rsp,%rbp
400531:  48 89 7d e8       mov   %rdi,-0x18(%rbp)
400548:  8b 45 f8          mov   -0x8(%rbp),%eax
400559:  48 01 d0          add   %rdx,%rax
400561:  83 45 f8 01       addl  $0x1,-0x8(%rbp)
```

[number] means a constant value (this is the number 1)

```c
int sum_array(int arr[], int nelems) {
    int sum = 0;
    for (int i = 0; i < nelems; i++) {
        sum += arr[i];
    }
    return sum;
}
```
Registers and memory

ANATOMY OF THE COMPUTER
An architecture view of computer hardware
The almighty mov

Our first instruction
Dude, where’s my data?

- **A main job of assembly language is to manage data:**
  - Data can be on the CPU (in registers) or in memory (at an address)
    - Turns out this distinction REALLY MATTERS for performance
    - [https://people.eecs.berkeley.edu/~rcs/research/interactive_latency.html](https://people.eecs.berkeley.edu/~rcs/research/interactive_latency.html)
  - Instructions often want to move data:
    - Move from one place in memory to another
    - Move from one register to another
    - Move from memory to register
    - Move from register to memory
  - Instructions often want to operate on data:
    - Add contents of register X to contents of register Y

- **Hence “mov” (move) instruction is paramount!**
mov

- **mov** **src,dst**
  - Optional suffix (b,w,l,q): `movb`, `movw`, `movl`, `movq`
  - One confusing thing about “move” it makes it sound like it leaves the src “empty”—no!
    - Does a **copy**, like the assignment operator you are familiar with
  - **src,dst** options:
    - Immediate (AKA constant value)
    - Register
    - Memory
### Addressing modes

<table>
<thead>
<tr>
<th>Address</th>
<th>Opcode</th>
<th>Assembly Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>40052e:</td>
<td>48 89 e5</td>
<td>mov %rsp,%rbp</td>
</tr>
<tr>
<td>400531:</td>
<td>48 89 7d e8</td>
<td>mov %rdi,-0x18(%rbp)</td>
</tr>
<tr>
<td>400548:</td>
<td>8b 45 f8</td>
<td>mov -0x8(%rbp),%eax</td>
</tr>
<tr>
<td>400559:</td>
<td>48 01 d0</td>
<td>add %rdx,%rax</td>
</tr>
<tr>
<td>400561:</td>
<td>83 45 f8 01</td>
<td>addl $0x1,-0x8(%rbp)</td>
</tr>
</tbody>
</table>

```c
int sum_array(int arr[], int nelems) {
    int sum = 0;
    for (int i = 0; i < nelems; i++) {
        sum += arr[i];
    }
    return sum;
}
```
Addressing modes

40052e:       48 89 e5                    mov     %rsp,%rbp
400531:       48 89 7d e8                mov     %rdi,-0x18(%rbp)
400548:       8b 45 f8                    mov     -0x8(%rbp),%eax
400559:       48 01 d0                    add     %rdx,%rax
400561:       83 45 f8 01                addl    $0x1,-0x8(%rbp)

int sum_array(int arr[], int nelems) {
    int sum = 0;
    for (int i = 0; i < nelems; i++) {
        sum += arr[i];
    }
    return sum;
}
Instruction Set Architectures

Some context and terminology
Instruction Set Architecture

- **The ISA defines:**
  - Operations that the processor can execute
  - Data transfer operations + how to access data
  - Control mechanisms like branch, jump (think loops and if-else)
  - Contract between programmer/compiler and hardware

- **Layer of abstraction:**
  - Above:
    - Programmer/compiler can write code for the ISA
    - New programming languages can be built on top of the ISA as long as the compiler will do the translation
  - Below:
    - New hardware can implement the ISA
    - Can have even potentially radical changes in hardware implementation
    - Have to “do” the same thing from programmer point of view

- **ISAs have incredible inertia!**
  - Legacy support is a huge issue for x86-64
Two major categories of Instruction Set Architectures

- **CISC:**
  - **Complex** instruction set computers
    - e.g., x86
  - Have special instructions for each thing you might want to do
  - Can write code with fewer instructions, because each instruction is very expressive

- **RISC:**
  - **Reduced** instruction set computers
    - e.g., MIPS
  - Have only a very tiny number of instructions, optimize the heck out of them in the hardware
  - Code may need to be longer because you have to go roundabout ways of achieving what you wanted