Day 6
Today
Basics of Processing
x86 assembly
registers
  basic instructions
  => numbers and expressions
  => variables / memory
  => decisions / loops
Review

little memory: [register]

CPU

mem bus

ignore

I/O bus

net

disk

"Processor" executes instructions

Program counter/instruction pointer

Processes:

1. fetch instruction (from addr in IP)
2. decode
3. execute
4. advance IP
5. next instruction
Problem:

large memory

> fast, but not fast enough

CPU: billions inst/sec

Memory: latency ~ 100 ns

Fetch inst "very often"

Solution: general purpose registers

some small # of registers

that instructions can use to put data in,
+ get data from
x86:

- registers: general purpose
- little pieces of memory
- each have a name
- can read / write

- 32-bit x86 -> 32 bits (4 bytes)

x86 register names:

\[
\begin{align*}
&\%eax \\
&\%ebx \\
&\%ecx \\
&\%edx \\
&\%esi \\
&\%edi
\end{align*}
\]

- 32 bits
- program counter: \%eip

%eax -> 32 bits

%eax (16 bits)

%ah %al

8 bits 8 bits
Instruction Sets

instructions themselves

- arithmetic
- memory
- control flow (if, while, call/return)

basic data types
- 32-bit numbers, etc.

other stuff

- Input/Output, support OS

Compiler Toolchain

C \rightarrow \text{assembly (.s)} \rightarrow \text{binary code}

add instruction \rightarrow 1101001 (add)
Have registers, want instructions

**Number => register**

**mov** instruction

- **MOV**
  - **source**
  - **destination**

- **name**
- **operand(s)** [1 or more]

**variants**: different sizes of data

- **movb** byte
- **movl** "long" (int) 4 bytes
- **movq** 8 bytes

**indicate** **numbers in registers**

**movl** $15$, %eax
it's called move
it's a copy

mov %eax, %ebx

(# %eax has same contents after)
operators

addl source, destination
dest = dest + src

subl source, dest
dest = dest - src

imull source, dest
dest = dest * source
(alt: imull aux, source, dest)
dest = aux * source

idivl arg

number to be divided: [%edx:%eax]
result of div = %eax
mod = %edx
Problem #1
Write assembly to:
- move value 1 into %eax
- add 10 to it and put result into %eax

\[
\text{movl } \$1, \%eax \\
\text{addl } \$10, \%eax
\]

Problem #2
(Expression: 3 \* 6 \* 2)
Use one register (%eax), and 3 instructions to compute this piece-by-piece

\[
\text{movl } \$6, \%eax \\
\text{imull } \$2, \%eax \\
\text{addl } \$3, \%eax
\]

Problem #3
\[
\text{movl } \$0, \%edx} \\
\text{movl } \$7, \%eax} \\
\text{movl } \$3, \%ebx} \\
\text{idivl } \%ebx} \\
\text{movl } \%eax, \%ecx} \\
\text{movl } \$0, \%edx} \\
\text{movl } \$9, \%eax} \\
\text{movl } \$2, \%ebx} \\
\text{idivl } \%ebx} \\
\text{movl } \%edx, \%eax} \\
\text{addl } \%ecx, \%eax
\]

Write simple C expression that is equivalent to these instructions

\[
\frac{7}{3} + 9 \% 2
\]

\[
2 + 1 = 3
\]
Memory: how to access memory

:) Many instructions take mem addr (in various forms) as one operand

:) x86 has many different ways of specifying a memory address
Memory Address Forms:

**Absolute address:** (a number)

- 
  ```
  movl 1000, %eax
  load from addr:1000
  value = %eax
  ```
  ```
  addl $1, %eax
  ```
  ```
  movl %eax, 1000
  ```

- 
  ```
  addl $1, %eax
  ```
  ```
  movl %eax, 1000
  ```

**Indirect address:**

- address is in a register
  e.g. %eax

  ```
  movl (%eax), %ebx
  ```

- Base + displacement

  ```
  movl 8(%eax), %ebx
  ```

  addr: contents of eax + 8 = 1008
Indexed:

```
movl 4(%eax, %ebx), %ecx
```

addr: \( 4 + \text{contents} + \text{contents} \)
\( \text{eax} \quad \text{ebx} \)

Most General:

```
movl 8(%eax, %ebx, 4), %ecx
```

addr: \( 8 + \text{contents} + 4 \times \text{contents} \)
\( \text{eax} \quad \text{ebx} \)

---
Problem #4 (from CSAPP 3.1)

Memory

<table>
<thead>
<tr>
<th>Address</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x100</td>
<td>0xFF</td>
</tr>
<tr>
<td>0x104</td>
<td>0xAB</td>
</tr>
<tr>
<td>0x108</td>
<td>0x13</td>
</tr>
<tr>
<td>0x10C</td>
<td>0x11</td>
</tr>
</tbody>
</table>

Registers

<table>
<thead>
<tr>
<th>%eax</th>
<th>0x100</th>
</tr>
</thead>
<tbody>
<tr>
<td>%ecx</td>
<td>0x1</td>
</tr>
<tr>
<td>%edx</td>
<td>0x3</td>
</tr>
</tbody>
</table>

Value of:

- %eax

- 0x104

- $0x108

- (%eax)

- 4(%eax)

- 9(%eax, %edx)

- 260(%ecx, %edx)

- 0x9FC(%ecx, 4)

- (%eax, %edx, 4)

```
movl( ), %reg
```

What ends up in dest reg?

```
264 256
```

```
0x100
```

```
0x108
```

```
0x11
```

```
0xAB
```

```
0x13
```

```
0xFF
```

```
0x9FC
```

```
0x11
```

[memo]
Stack:

CPU

%esp

%eip

Address Space

Top

Valid

Stack:

local variables

Stack pointer:

%esp

(another register)

can read/write
New register to help with stack: esp (extended stack pointer)

Referred to as %esp

| [..........] | eax | 32 bits |
| [..........] | ax  | 16 bits |
| [..........] | ah  |  8 bits |
| [..........] | al  |  8 bits |
| [..........] | ebx | 16 bits |
| [..........] | bx  |  8 bits |
| [..........] | bh  |  8 bits |
| [..........] | bl  |         |
| [..........] | ecx | 32 bits |
| [..........] | cx  |         |
| [..........] | ch  |         |
| [..........] | cl  |         |
| [..........] | edx | 32 bits |
| [..........] | dx  |         |
| [..........] | dh  |         |
| [..........] | dl  |         |
| [..........] | esi |         |
| [..........] | edi |         |
| [..........] | esp | 32 bits |
| [..........] | eip | 32 bits |

Points to "top of stack" when program is running
Changes often (room for local variables, function call/return, etc.)

Can use normal instructions to interact with it, e.g., addl, subl
Can also use special instructions (we'll see this later)

Problem #5

Use instructions to:
- Increase size of stack by 4 bytes
- Store an integer value 10 into the top of the stack
- Retrieve that value and put it into %ecx
- Add 5 to it.
- Put final value into %eax

```
addl $-4, %0esp
movl $10, (%0esp)
movl (%0esp), %0ecx
addl $5, %0ecx
movl %0ecx, %eax
addl $4, %0esp
```
\(\text{statement 1};\)

\(\text{if (expr) \{\)

\(\text{statement 2};\)

\(3\)

\(\text{statement 3};\)

\(2\) possible control flows:

<table>
<thead>
<tr>
<th>s1</th>
<th>s1</th>
</tr>
</thead>
<tbody>
<tr>
<td>s2</td>
<td>s3</td>
</tr>
<tr>
<td>s3</td>
<td>s3</td>
</tr>
</tbody>
</table>

need assembly help

w/ control flow

\(\text{stmt0};\)

\(\text{while (expr) \{\)

\(\text{stmt1};\)

\(\text{stmt2};\)

\(3\)

\(\text{stmt3}.\)

\[\text{Needs:}\]

\(\text{Need to compute expression (perform comparison)}\)

\(\text{if (a > 3) \{\)}

\(\text{Need to change %eip (control flow)}\)
Instructions

⇒ Comparison

\texttt{cmp}\ 
\texttt{cmpl b, a}

internally:
computing \((a-b)\)
(no destination)

what it does:
if \((a-b) = 0\)
\ZF = 1
else
\ZF = 0

if \((a-b) < 0\)
\SF = 1
else
\SF = 0

"state" on CPU:
\%eip
\%esp

gen purpose (\%eax, etc.)

\%eflags

<p>| | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>6</td>
<td>7</td>
<td>11</td>
<td></td>
</tr>
</tbody>
</table>

\begin{tabular}{c c c c c}
Carry & Zero & Sign & Overflow & Flag \\
Flag & Flag & Flag & Flag & Flag \\
\hline
(cf) & (zf)(sf) & (of) & \\
\equiv & \equiv & \equiv & \\
\end{tabular}
Condition codes: new bits in hidden %eflags register.

Some instructions set those bits based on comparisons:
  cmp, test
Other instructions change control flow (%eip) based on results:
  jmp family

**INSTRUCTION:** `cmpl B, A`

computes A-B (but doesn't put result anywhere)

condition codes (incomplete):
  zero flag : ZF=1 if (A-B) == 0 otherwise ZF=0
  signed flag : SF=1 if (A-B) < 0 otherwise SF=0

**INSTRUCTION:** `jmp TARGET`
  always changes %eip to TARGET

**INSTRUCTION:** `je TARGET`
  %eip=TARGET if ZF==1

**INSTRUCTION:** `jne TARGET`
  %eip=TARGET if ZF==0

**INSTRUCTION:** `jg TARGET`
  %eip=TARGET if SF==0

**INSTRUCTION:** `jge TARGET`
  %eip=TARGET if SF==0

**INSTRUCTION:** `jl TARGET`
  %eip=TARGET if SF==1

**INSTRUCTION:** `jle TARGET`
  %eip=TARGET if SF==1 or ZF==1

(Doesn't quite work in all cases)
**Instruction**

Jump instruction:
- Change flow of control

Many forms:
- `jmp target`
  - (unconditional branch)
  - Changes `%eip` to target address

Conditional branch based on values of flags in `%eflags`
- `jne target` (jump not equal)
- `cmp inst1, inst2`
- `jne top` otherwise "fall through"
  - (go to next inst)

Idiom:
- `=> comparison`
- `=> jump/branch`

Name and colon

Top:
- `instruction 1`
- `instruction 2`
- `instruction 3`
- `jmp top`
HW3:

TAs extra office hours

\[
\begin{array}
\text{Fri} \\
\text{Sat} \\
\text{Sun}
\end{array}
\]

\Rightarrow \text{check email}

Test scripts:
\Rightarrow \text{try to run them}

HW2 (regrade)

hw2-regrade/

put slightly mod'd code here

+ get regrade
```c
struct list_t {
    node_t * array,
    int num_elements;
}

void list_init ( list_t * l, ... ) {
    l->num_elements = 0;
    node_t a [chunk_size],
    l->array = 3, a [0],
    return;
}

called:
list_t L;
list_init (3, L, ...);
(OK)
```
void list-init

\[
\text{void } \ast L \Rightarrow \\
\text{not undefined}
\]

\[
L = \text{malloc (sizeof } \text{struct list-t)}
\]

\[
L \rightarrow \text{size} = 0; \alpha
\]

\[
\text{return}
\]

memory leak
Bug #1:

```c
struct list-t *L;
L->size = 0;
```

Dereference

```c
(*L).size
```

Bug #2:

```c
int *P;
int x = *P;
```

List_init (chunk_size)

```c
void list_init(
L, chunk_size);
L->size = 0;
```