Lecture 7
(in-class)

Review
Decisions
Function Call
Review:

\[ x86 \] 32-bit

Registers
- memory on CPU
- 32-bit registers
- eip: hold address of next instruction to fetch
- stack pointer: esp => top of stack

C language
human \downarrow compiler
S \downarrow assembler
\circ (binary code) \rightarrow executable
linker

CPU

Memory

code
**Last Time:**

- `mov` e.g. `movl $10, %eax`
- **FAST** `movl %eax, %ebx`
  (copy: eax still has value after)
- `movl (%eax), %ebx`
  - compute address:
  - contents of eax
  - fetch 4 bytes @ addr
  - put result = %ebx

- `movl offset(reg1, reg2, mul)`
  - compute addr: 1, 2, 4, 8
  - offset + reg1 +
  - reg2 * mul
  - useful in array access
Decisions:

- if
- while

=> 2 types of instruction

1) perform comparison:
\[
\text{cmp} b, a
\]

2) change control flow of program based on result of comparison
\[
\text{jmp} \quad \text{family of instructions}
\]
```plaintext
`cmp b/a

result of comparison
= condition codes
bits in
that we can use to make decisions
(e.g. zero flag, sign flag, overflow flag)

jmp target => label in .S
change eip to target
unconditionally
jmp target
conditionally

if a > b
else
fall through to next transfer control inst.
```
cmp     b, a
je      a == b
jne     a != b
jg      a > b
jge     a >= b
jl      a < b
jle     a <= b
Example 7:

\[
\begin{align*}
\text{if } (a > b) \\
(a++);
\end{align*}
\]

\[
\Rightarrow \text{ cmp } %ebx, %eax
\]

\[
\Rightarrow \text{ jg } \text{ do } \text{ if } a \leq b
\]

\[
\text{jmp } \text{ dont+}
\]

\[
\text{do: addl } \text{ $1, } %eax
\]

\[
\text{dont: inst}
\]

\[
\text{cmp } %ebx, %eax
\]

\[
\text{jle dont}
\]

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

\[
\text{ dont:}
\]

\[
\text{target:}
\]

\[
\text{inst+1}
\]

\[
\text{inst+2}
\]
if (a > b)
    a++;
else
    a = b;

    cmp %ebx, %eax
    jg doadd
    jmp domov
doadd:
    add $7, %eax
    jmp end

domov:
    movl %ebx, %eax
end:
    (inst)
```
while (b > 0) {
    a++;
    b--;
}
```

```
jmp looptest
```

```
addl $1, %eax
subl $2, %ebx
```

```
looptest:
    cmp $0, %ebx
    (jg top)
```

```
branches or conditional branches
```
other instructions that set condition codes

thus far:
\[ \text{cmpl } b, a \Rightarrow a - b \Rightarrow \]

arithmetic instructions do

\[ \text{e.g. addl} \]

also:
\[ \text{test } b, a \Rightarrow a \& b \]

bitwise and

see condition codes

```
Set instruction
```

```
  eax
  1111
  cmpl b, a
  sete %al

\[ \Rightarrow 0 \]
```
function call / return

in C: easy
to call:

```
void foo (int a, int b)
{
    int c; // stack

    return;
}
```

Goal: same @ assembly level

Questions:

- How to call a function?
- How to return from function to inst after call?
- How to obtain return value?
- How to pass parameters?
- How to alloc space for local vars?
- How to access parameters/local variables?
- How to use registers w/in function?
Calling Convention

set of rules to follow when call/return

x86 convention: cdecl

gcc on x86
new instructions:

```
call target

⇒ transfers control to target (jmp)
⇒ push return address onto stack

ret
⇒ pops ret addr off stack
⇒ transfers control (jmp)
```

example

```
inst1:
call foo:
inst2 = addr of => A012
```

Return value:
how to return a value to caller?

Define a Convention

goal: use stack

foo:
    movl $3, 4(%esp)
    ret

int foo () {
    return 3;
}

main:

[subl $4, %esp]
Call foo
    movl 8(%esp), %eax

int rc = foo ();
printf("%d\n", rc);
    printf 3

esp &AA
ret add 1000
ESP @3
use this for ret value

AA
movl $3, (%esp)

movl $996, %esp

call foo

inst2;

ret

=> eip
return value convention #2:

put return value in register %eax (not memory)

⇒ use this convention

func:

```
movl $3, %eax
```

ret

main:

```c
    call foo
```

⇒ return value in %eax

implication

⇒ cannot assume eax

will be overwritten
Passing parameters:

```c
int func (int a, int b) {
    return a + b;
}
```

Use stack:

before call:
- Push params onto stack in "reverse" order
  - Param N-1
  - Param 0

`call func (1, 2)`
int func (int a, int b) {
    return a + b;
}

func(1, 2);

main:
    100: [subl $8, %esp]
    104: movl $2, 4(%esp)
    108: movl $1, 0(%esp)
    112: call func
    116: addl $8, %esp

ret

push $2
push $1
Space for local variables:

before call
params
call : ret addr

how to make space for local vars on stack?
```c
int func (int a, int b) {
  int c = 1;
  local allocate room for c
  return a + b + c;
}
```

inside func:
need space for c
subl $4, %esp
movl $1, 0(%esp)
before: (accessing 'a')

```
[movl 4(%esp), %eax]
```

goal:
to refer to params/local
in same way
even though esp may be changing during function

```
stack
```

```
esp add ret addr
```

```
a
b
```

```
1
```

allocate room on entry, deallocate on return (automatic)
base pointer (frame pointer)
new register: [%ebp]
stable (non-changing) pointer into stack during function call

\[ a : 4 \text{ (%ebp)} \]
\[ b : 8 \text{ (%ebp)} \]
\[ c : -4 \text{ (%ebp)} \]