Sports: HMC CS Professor to coach 2021 U.S. Olympic chocolate-eating team
Weather: 63.79% chance of weather today

News in Brief

CS 5 alien abducted by aliens (p. 42)

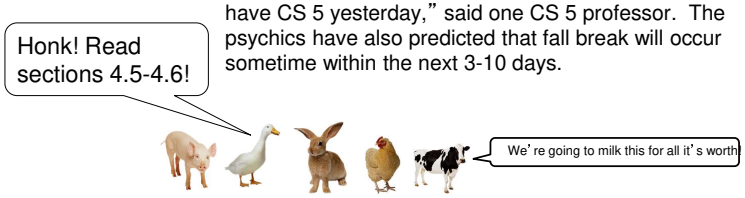
Page 42 will no longer be published (p. 42)

Farm animals displace penguins, invade CS 5 notes (p. 42)

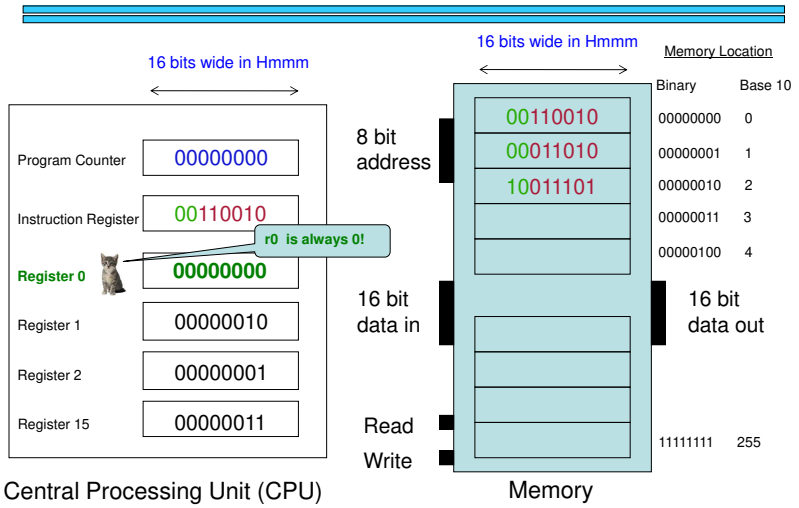
CS 5 Today

Psychics predict that there was no CS 5 lecture yesterday. Definitive proof of paranormal phenomena!

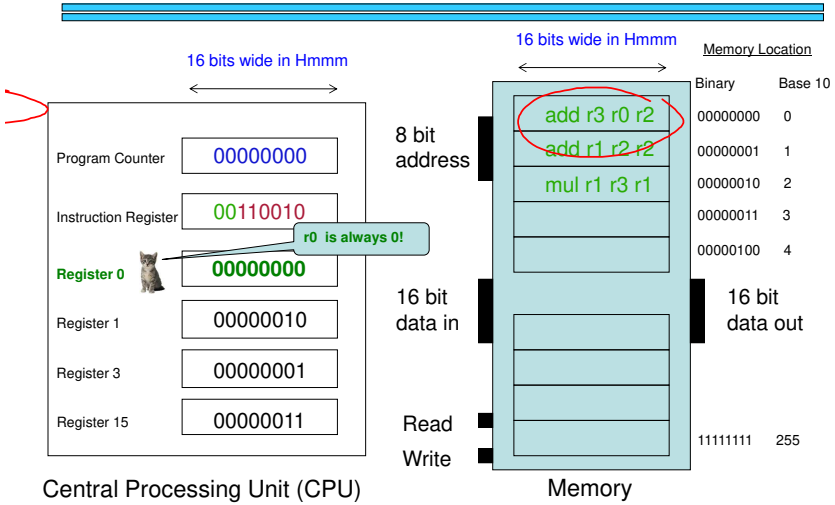
(Claremont AP): A group of psychics has made an extraordinary set of predictions that, one-by-one, are being corroborated by scientists. "It is indeed true that we didn't have CS 5 yesterday," said one CS 5 professor. The psychics have also predicted that fall break will occur sometime within the next 3-10 days.



Machine Language Versus…



…Assembly Language!



Hmmm Assembly Language

```
add r2 r2 r2      reg2 = reg2 + reg2
```
crazy, perhaps, but used ALL the time

```
sub r2 r1 r4      reg2 = reg1 - reg4
```
which is why it is written this way in Python!

```
mul r7 r6 r2      reg7 = reg6 * reg2
div r1 r1 r1      reg1 = reg1 / reg1
```
INTEGER division—no remainders

```
setn r1 42        reg1 = 42
addn r1 -1        reg1 = reg1 - 1
```
you can replace 42 with anything from -128 to 127

a shortcut

```
read r10
write r1
```
read from keyboard and write to screen

Each instruction (and many more) gets implemented for a particular processor and particular machine…

jumps

Unconditional jump

```
jumpn 42
```

Replaces the PC (program counter) with **42**. "Jump to program line **n**umber **42**."

Conditional jumps

```
jeqzn r1 #
```
IF **r1 == 0** THEN jump to line **n**umber #

```
jgtzn r1 #
```
IF **r1 > 0** THEN jump to line **n**umber #

```
jltzn r1 #
```
IF **r1 < 0** THEN jump to line **n**umber #

```
jnezn r1 #
```
IF **r1 != 0** THEN jump to line **n**umber #

Register jump

```
jumpr r1
```
Jump to the line # stored in **reg1**!

Worksheet

Feeling Jumpy?

1 Write an assembly-language program that reads ***one*** integer, X, as keyboard input into register r1. Then the program should compute X^2+3X+4, leaving the result in register r13, and write it out.

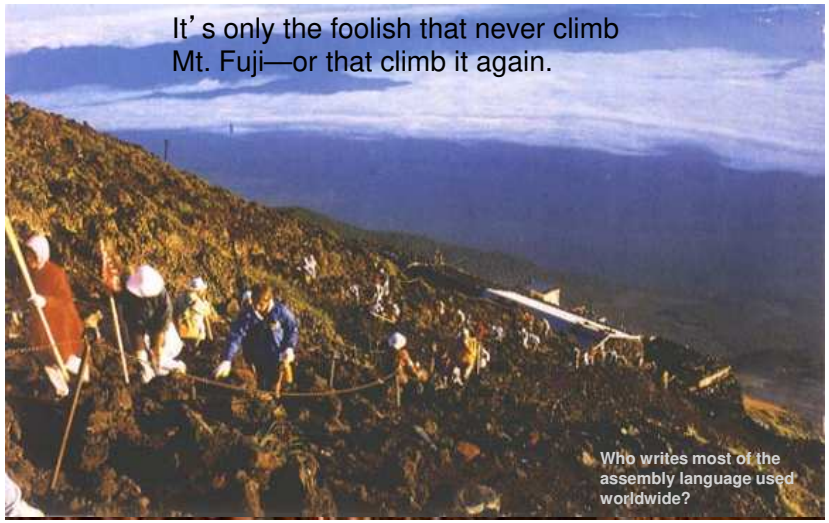
Registers - CPU		Instructions	
r0	0	0	read r1
r1		1	
r2		2	
r3		3	
r4		4	
r5		5	
r13		6	
		7	
		8	
		9	
		10	

2 Write an assembly-language program that reads ***two*** integers r1 and r2 as keyboard input. Then, the program should compute $r1^{r2}$ in register r13 and write it out. You may assume that r2 >= 0.

Registers - CPU		Instructions	
r0	0	0	read r1
r1		1	read r2
r2		2	setn r13 1
r3		3	
r4		4	
r5		5	
r13		6	
		7	
		8	
		9	
		10	

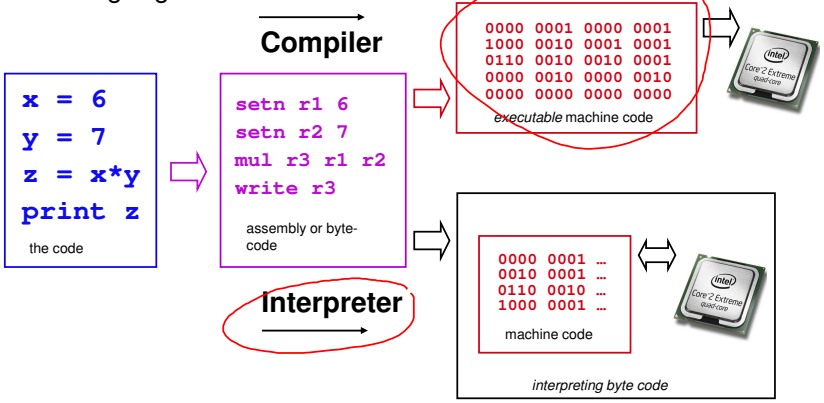
Instruction	Description
	System Instructions
halt	Stop!
read rX	Place user input in register rX
write rX	Print contents of register rX
nop	Do nothing
	Setting register data
setn rX N	Set register rX equal to the integer N (-128 to 127)
addn rX N	Add integer N (-128 to 127) to register rX
copy rX rY	Copy the number in rX into rY
	Arithmetic
add rX rY rZ	Set rX = rY + rZ
sub rX rY rZ	Set rX = rY – rZ
neg rX rY	Set rX = -rY
mul rX rY rZ	Set rX = rY * rZ
div rX rY rZ	Set rX = rY // rZ (integer division; no remainder)
mod rX rY rZ	Set rX = rY % rZ (returns the remainder of integer division)
	Jumps!
jumpn N	Set program counter to address N
jumpr rX	Set program counter to address in rX
jeqzn rX N	If rX == 0, then jump to line N
jnezn rX N	If rX != 0, then jump to line N
jgtzn rX N	If rX > 0, then jump to line N
jltzn rX N	If rX < 0, then jump to line N
calln rX N	Copy the next program address into rX and then jump to memory address N
	Interacting with memory (RAM)
pushr rX rY	Store contents of rX into memory addressed by rY, and then increment rY
popr rX rY	Decrement rY, and then load the contrents of memory addressed by rY into rX

Why Assembly Language?



The Compiler

A program that translates from human-usable language into assembly language and machine language



Examples

Haswell

```
.globl main
    .type   main,function
main:
.LFB2:
    pushq   %rbp
.LCFI:
    movq    %rsp,%rbp
.LCFI1:
    subq    $16,%rsp
.LCFI2:
    movl    $6,-12(%rbp)
    movl    $7,-8(%rbp)
    movl    -12(%rbp),%eax
    imull   -8(%rbp),%eax
    movl    %eax,-4(%rbp)
    movl    -4(%rbp),%esi
    movl    $.LC0,%edi
    movl    $0,%eax
    call    printf
    leave
    ret
```

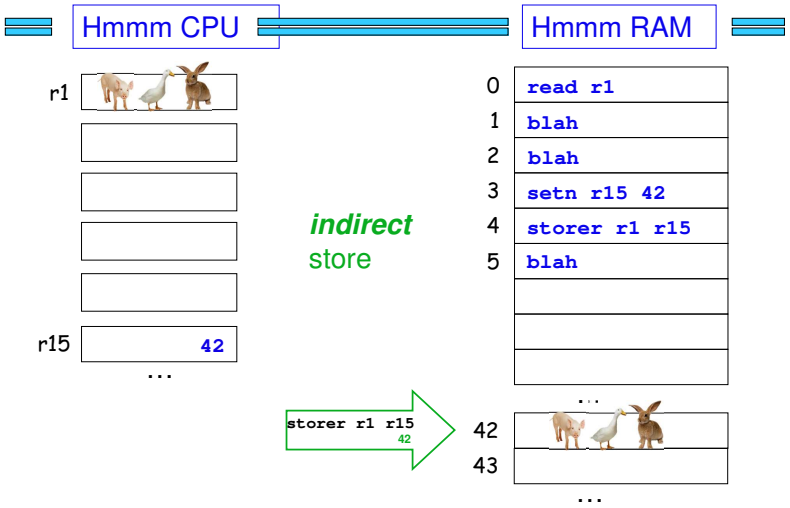
```
x = 6
y = 7
z = x*y
print z
```
The code

Power PC

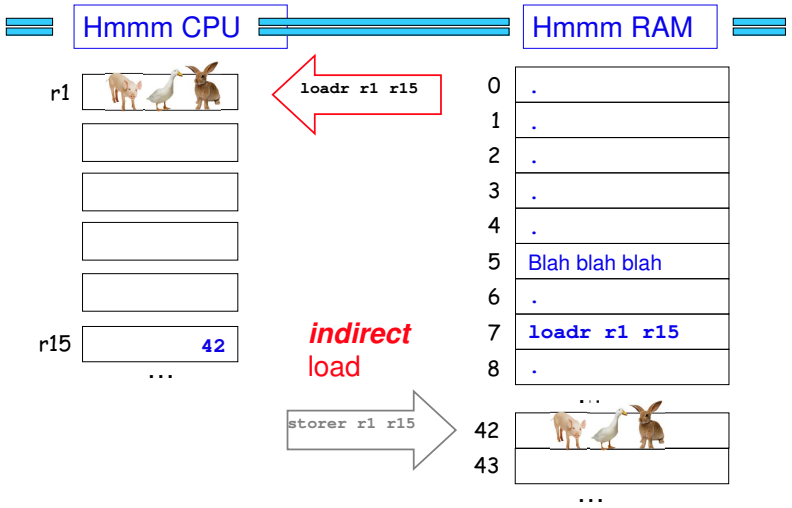
```
LC0:
    .ascii "z is %d\12\0"
    .text
    .align 2
    .globl _main
_main:
    mflr r0
    stmw r30,-8(r1)
    stw r0,8(r1)
    stwu r1,-96(r1)
    mr r30,r1
    bcl 20,31,"L00000000001$pb"
"L00000000001$pb":
    mflr r31
    li r0,6
    stw r0,64(r30)
    li r0,7
    stw r0,60(r30)
    lwz r2,64(r30)
    lwz r0,60(r30)
    mullw r0,r2,r0
    stw r0,56(r30)
    addis r2,r31,ha16(LC0-"L00000000001$pb")
    la r3,lo16(LC0-"L00000000001$pb")(r2)
    lwz r4,56(r30)
    bl L_printf$LDBLStub$stub
    lwz r1,0(r1)
    lwz r0,8(r1)
    mtlr r0
    lmw r30,-8(r1)
    blr
```

Each processor has its own *endearing* idiosyncrasies...

`storer` Goes TO Memory



`loadr` Comes FROM Memory



calln = setn + jumpn!

A function call in python:

```
def main():
    r1 = input()
    result = factorial(r1)
    print(result)


def factorial(r1):
    # do work
    return result
```

puts NEXT line # into r14, then jumps to line 4

Hmmm's **call** operation:

```
0   read r1
1   calln r14 4
2   write r13
3   halt

4   do stuff and
5   answer in r13
6   jumpr r14
```

Factorial: *Function Call!*

Hmmm CPU

r0: 0

Input value: x

r1:

Final result - return value - *in progress*

r13:

location / line to return TO

r14: 2

Hmmm RAM

```
0  read r1          } input
1  calln r14 4      function call
2  write r13        } output
3  halt
4  setn r13 1       the function!
5  jeqzn r1 9       } loop
6  mul r13 r13 r1
7  addn r1 -1
8  jumpn 5
9  jumpr r14        return
```

Which Factorial Is It?

```
0  read r1          } input
1  setn r13 1       initialize result to 1
2  jeqzn r1 6       } loop
3  mul r13 r13 r1
4  addn r1 -1
5  jumpn 2
6  write r13        } output
7  halt
```

```
def fac1():
    r1 = input()
    r13 = 1
    while r1 != 0:
        r13 = r13 * r1
        r1 += -1
    print r13
    return

def fac2(r1):
    if r1 == 0:
        return 1
    else:
        return r1 * fac2(r1-1)
```

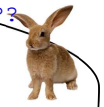
Function Calls…

```
def main():
    r1 = input()  <-- r1=3
    r13 = emma(r1)  <-- emma(3)
    r13 = r13 + r1
    print(r13)
    return

def emma(r1):  <-- r1=3
    r1 = r1 + 1  <-- r1=4
    r13 = sarah(r1)  <-- sarah(4) r13=47
    r13 = r13 + r1  <-- r13=??
    return r13

def sarah(r1):  <-- r1=4
    r1 = r1 + 42  <-- r1=46
    r13 = r1 + 1  <-- r13=47
    return r13  <-- return(47)
```

Chew on this…



Function Calls…

```
def main():
    r1 = input()<------  r1=3
    r13 = emma(r1) <-- emma(3)  r13=51
    r13 = r13 + r1 <-- r13=??
    print(r13)
    return

def emma(r1):  <------  r1=3
    r1 = r1 + 1 <------  r1=4
    r13 = sarah(r1)<-- sarah(4) r13=47
    r13 = r13 + r1 <--  r13=51
    return r13   <------return(51)

def sarah(r1): <------  r1=4
    r1 = r1 + 42 <------ r1=46
    r13 = r1 + 1 <------ r13=47
    return r13   <------return(47)
```

Function Calls…

```
def main():
    r1 = input()<------  r1=3
    r13 = emma(r1) <-- emma(3)   r13=51
    r13 = r13 + r1 <-- r13=54
    print(r13)
    return
```

54

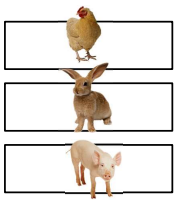
```
def emma(r1):  <------  r1=3
    r1 = r1 + 1 <------  r1=4
    r13 = sarah(r1)<-- sarah(4) r13=47
    r13 = r13 + r1 <--  r13=51
    return r13   <------return(51)

def sarah(r1): <------  r1=4
    r1 = r1 + 42 <------ r1=46
    r13 = r1 + 1 <------ r13=47
    return r13   <------return(47)
```

Cool, but how does this work!?

The Stack!

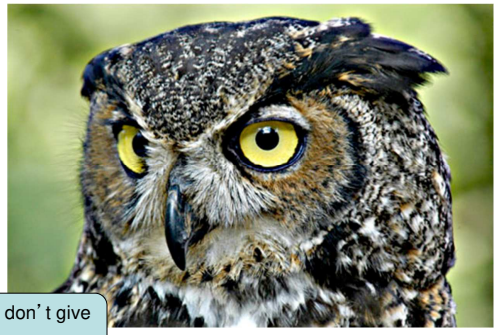
AFLAC, ShmAFLAC! Be careful up there!



Insert (“push”)

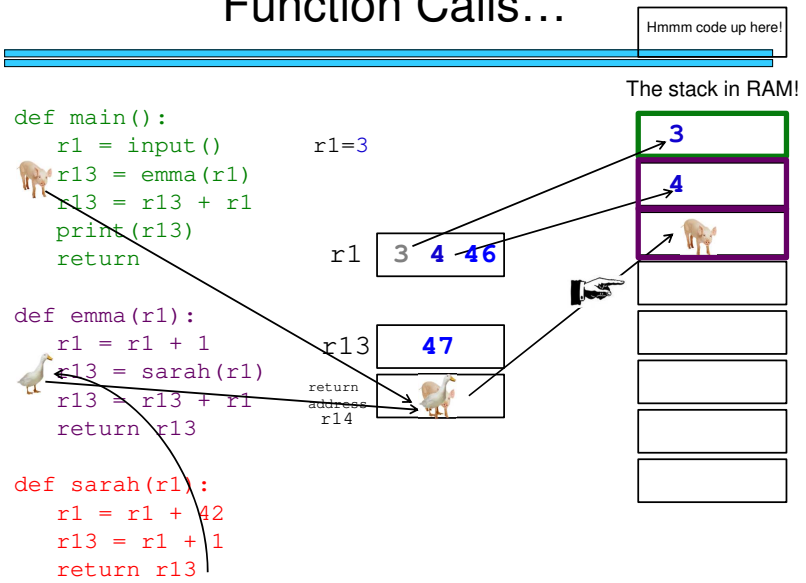
Remove (“pop”)

Watch carefully…

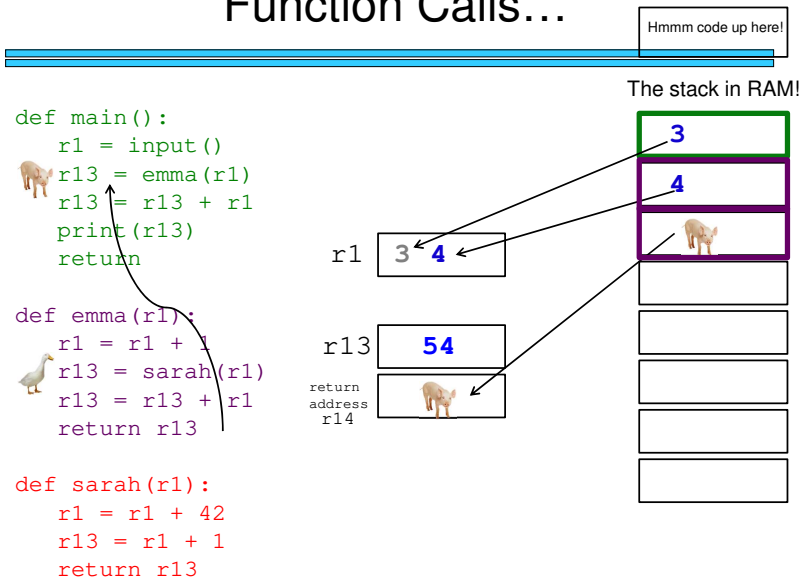


What if I don’t give a hoot?!

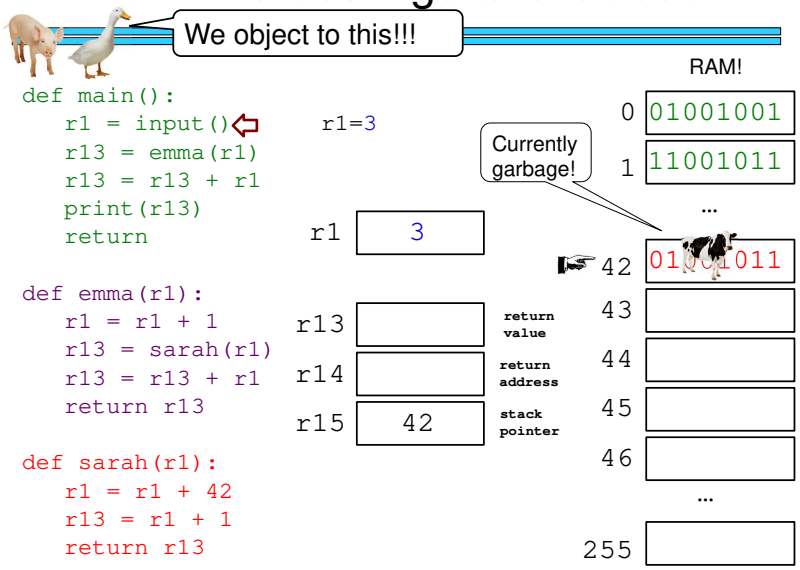
Function Calls…



Function Calls…



Now Without Pigs and Geese!



Now Without Pigs and Geese!