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...

## Central Processing Unit (CPU)

- **Program Counter**: 00000000
- **Instruction Register**: 00110010
- **Register 0**: 00000000
- **Register 1**: 00000001
- **Register 2**: 00000010
- **Register 15**: 00000011

**r0 is always 0!**

## Memory Location

<table>
<thead>
<tr>
<th>Binary</th>
<th>Base 10</th>
</tr>
</thead>
<tbody>
<tr>
<td>00000000</td>
<td>0</td>
</tr>
<tr>
<td>00000001</td>
<td>1</td>
</tr>
<tr>
<td>00000010</td>
<td>2</td>
</tr>
<tr>
<td>00000011</td>
<td>3</td>
</tr>
<tr>
<td>00000100</td>
<td>4</td>
</tr>
<tr>
<td>11111111</td>
<td>255</td>
</tr>
</tbody>
</table>

## Machine Language Versus...

- 8 bit address
- 8 bit data in
- 8 bit data out
- Read
- Write

16 bits wide in Hmmm
Central Processing Unit (CPU)

Program Counter: 00000000
Instruction Register: 00110010
Register 0: 00000000
Register 1: 00000010
Register 3: 00000001
Register 15: 00000011

Memory Location

<table>
<thead>
<tr>
<th>Binary</th>
<th>Base 10</th>
</tr>
</thead>
<tbody>
<tr>
<td>00000000</td>
<td>0</td>
</tr>
<tr>
<td>00000001</td>
<td>1</td>
</tr>
<tr>
<td>00000010</td>
<td>2</td>
</tr>
<tr>
<td>00000011</td>
<td>3</td>
</tr>
<tr>
<td>00000100</td>
<td>4</td>
</tr>
<tr>
<td>11111111</td>
<td>255</td>
</tr>
</tbody>
</table>

8 bit address
- add r3 r0 r2
- add r1 r2 r2
- mul r1 r3 r1

8 bit data in

Read
Write

...Assembly Language!

16 bits wide in Hmmm
Hmmm Assembly Language

```
add r2 r2 r2  \quad \text{reg2} = \text{reg2} + \text{reg2} \\
\quad \quad \quad \quad \quad \quad \quad \quad \text{crazy, perhaps, but used ALL the time} \\
sub r2 r1 r4  \quad \text{reg2} = \text{reg1} - \text{reg4} \\
\quad \quad \quad \quad \quad \quad \quad \quad \text{which is why it is written this way in python!} \\
mul r7 r6 r2  \quad \text{reg7} = \text{reg6} \times \text{reg2} \\
div r1 r1 r1  \quad \text{reg1} = \text{reg1} / \text{reg1} \\
\quad \quad \quad \quad \quad \quad \quad \quad \text{INTEGER division—no remainders} \\
setn r1 42   \quad \text{reg1} = 42 \quad \text{you can replace 42 with anything from -128 to 127} \\
addn r1 -1   \quad \text{reg1} = \text{reg1} - 1 \quad \text{a shortcut} \\
read r10     \quad \text{read from keyboard and write to screen} \\
write r1     \quad \text{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 number 42.”

*Conditional* jumps

```
jeqzn r1 #
jgtzn r1 #
jltn zn r1 #
jnezn r1 #
```

IF $r1 == 0$ THEN jump to line number #

IF $r1 > 0$ THEN jump to the location in #

IF $r1 < 0$ THEN jump to the location in #

IF $r1 != 0$ THEN jump to the location in #

*Register* jump

```
jump r r1
```

Jump to the line # stored in reg1!

*This IS making me jumpy!*
Worksheet

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.

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

| r0 | 0 |
| r1 | X |
| r2 | X^2 |
| r3 | 3X |
| r4 | 4 |

Memory - RAM

0. read r1
1. mul r2 r1 r1
2. set r3 3
3. mul r3 r3 r3
4. add r13 r2 r3
5. set r4 4
6. add r18 r15 r4
7. write r13
8. halt
9. ?
10. ?

Memory - RAM

0
1
2
3
4
5
6
7
8
9
10
<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
<th>Aliases</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>halt</strong></td>
<td>Stop!</td>
<td></td>
</tr>
<tr>
<td><strong>read rX</strong></td>
<td>Place user input in register rX</td>
<td></td>
</tr>
<tr>
<td><strong>write rX</strong></td>
<td>Print contents of register rX</td>
<td></td>
</tr>
<tr>
<td><strong>nop</strong></td>
<td>Do nothing</td>
<td></td>
</tr>
</tbody>
</table>

**System instructions**

**Setting register data**

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
<th>Aliases</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>setn rX N</strong></td>
<td>Set register rX equal to the integer N (-128 to +127)</td>
<td></td>
</tr>
<tr>
<td><strong>addn rX N</strong></td>
<td>Add integer N (-128 to 127) to register rX</td>
<td></td>
</tr>
<tr>
<td><strong>copy rX rY</strong></td>
<td>Set rX = rY</td>
<td>mov</td>
</tr>
</tbody>
</table>

**Arithmetic**

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>add rX rY rZ</strong></td>
<td>Set rX = rY + rZ</td>
</tr>
<tr>
<td><strong>sub rX rY rZ</strong></td>
<td>Set rX = rY - rZ</td>
</tr>
<tr>
<td><strong>neg rX rY</strong></td>
<td>Set rX = -rY</td>
</tr>
<tr>
<td><strong>mul rX rY rZ</strong></td>
<td>Set rX = rY * rZ</td>
</tr>
<tr>
<td><strong>div rX rY rZ</strong></td>
<td>Set rX = rY / rZ (integer division; no remainder)</td>
</tr>
<tr>
<td><strong>mod rX rY rZ</strong></td>
<td>Set rX = rY % rZ (returns the remainder of integer division)</td>
</tr>
</tbody>
</table>

**Jumps!**

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>jumpn N</strong></td>
<td>Set program counter to address N</td>
</tr>
<tr>
<td><strong>jumpr rX</strong></td>
<td>Set program counter to address in rX</td>
</tr>
<tr>
<td><strong>jeqzn rX N</strong></td>
<td>If rX == 0, then jump to line N</td>
</tr>
<tr>
<td><strong>jnezn rX N</strong></td>
<td>If rX != 0, then jump to line N</td>
</tr>
<tr>
<td><strong>jgtzn rX N</strong></td>
<td>If rX &gt; 0, then jump to line N</td>
</tr>
<tr>
<td><strong>jltzn rX N</strong></td>
<td>If rX &lt; 0, then jump to line N</td>
</tr>
<tr>
<td><strong>calln rX N</strong></td>
<td>Copy the next address into rX and then jump to mem. addr. N</td>
</tr>
</tbody>
</table>

**Interacting with memory (RAM)**

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>loadn rX N</strong></td>
<td>Load register rX with the contents of memory address N</td>
</tr>
<tr>
<td><strong>storen rX N</strong></td>
<td>Store contents of register rX into memory address N</td>
</tr>
<tr>
<td><strong>load rX rY</strong></td>
<td>Load register rX with data from the address location held in reg. rY</td>
</tr>
<tr>
<td><strong>storer rX rY</strong></td>
<td>Store contents of register rX into memory address held in reg. rY</td>
</tr>
</tbody>
</table>
Why Assembly Language?

It’s only the foolish that never climb Mt. Fuji—or that climb it again.

Who writes most of the assembly language used?
The Compiler

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

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

the code

assembly or bytecode

```
setn r1 6
setn r2 7
mul r3 r1 r2
write r3
```

assembly or bytecode

```
0000 0001 0000 0001
1000 0010 0001 0001
0110 0010 0010 0001
0000 0010 0000 0010
0000 0000 0000 0000
```

executable machine code

```
0000 0001 ...
0010 0001 ...
0110 0010 ...
1000 0001 ...
```

machine code

```
interpreting byte code
```

```
interpreting byte code
```
Examples

Core 2 Duo

Each processor has its own endearing idiosyncrasies...

Power PC

```
x = 6
y = 7
z = x * y
```

The code

```
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
```
storer Goes TO Memory

Hmmm CPU

r1

... 42

r15

... 42

indirect store

Hmmm RAM

0
read r1

1
blah

2
blah

3
setn r15 42

4
storer r1 r15

5
blah

...
loadr Comes FROM Memory

Hmmm CPU

loadr r1 r15

indirect load

storier r1 r15

Hmmm RAM

loadr r1 r15

Blah blah blah

...
calln = setn + jumpn!

A function call in python:

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

def factorial(r1):
    # do work
    return result

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  jumper r14

puts NEXT line # into r14, then jumps to line 4
Factorial: *Function Call*

### Hmmm CPU

<table>
<thead>
<tr>
<th>Register</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>r0</td>
<td>0</td>
</tr>
<tr>
<td>r1</td>
<td></td>
</tr>
<tr>
<td>r13</td>
<td></td>
</tr>
<tr>
<td>r14</td>
<td>2</td>
</tr>
</tbody>
</table>

**Input value:** x

**Final result - return value - in progress**

### Hmmm RAM

<table>
<thead>
<tr>
<th>Address</th>
<th>Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>read r1</td>
</tr>
<tr>
<td>1</td>
<td>calln r14 4</td>
</tr>
<tr>
<td>2</td>
<td>write r13</td>
</tr>
<tr>
<td>3</td>
<td>halt</td>
</tr>
<tr>
<td>4</td>
<td>setn r13 1</td>
</tr>
<tr>
<td>5</td>
<td>jeqzn r1 9</td>
</tr>
<tr>
<td>6</td>
<td>mul r13 r13 r1</td>
</tr>
<tr>
<td>7</td>
<td>addn r1 -1</td>
</tr>
<tr>
<td>8</td>
<td>jumpn 5</td>
</tr>
<tr>
<td>9</td>
<td>jump r14</td>
</tr>
</tbody>
</table>

- **Input**
- **Output**
- **Function call**
- **Loop**
- **Return**
Which Factorial Is It?

```
0  read r1
1  setn r13 1
2  jeqzn r1 6
3  mul r13 r13 r1
4  addn r1 1
5  jumpn 2
6  write r13
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)
```
def main():
    r1 = input()
    r13 = emma(r1)
    r13 = r13 + r1
    print(r13)
    return

def emma(r1):
    r1 = r1 + 1
    r13 = sarah(r1)
    r13 = r13 + r1
    return r13

def sarah(r1):
    r1 = r1 + 42
    r13 = r1 + 1
    return r13
def main():
    r1 = input()  
    r13 = emma(r1)  
    r13 = r13 + r1
    print(r13)
    return

def emma(r1):
    r1 = r1 + 1
    r13 = sarah(r1)
    r13 = r13 + r1
    return r13

def sarah(r1):
    r1 = r1 + 42
    r13 = r1 + 1
    return r13

r1 = 3
emma(3)
r1 = 3
r1 = 4
sarah(4)
r1 = 4
r1 = 46
return(47)

Chew on this…
You should be worried!
def main():
    r1 = input()   # r1=3
    r13 = emma(r1) # emma(3)
    r13 = r13 + r1
    print(r13)
    return

def emma(r1):
    r1 = r1 + 1   # r1=4
    r13 = sarah(r1) # sarah(4) r13=47
    r13 = r13 + r1 # r13=??
    return r13

def sarah(r1):
    r1 = r1 + 42   # r1=4
    r13 = r1 + 1   # r13=46
    return r13      # return(47)
```python
def main():
    r1 = input()  # r1=3
    r13 = emma(r1)  # emma(3)  # r13=51
    r13 = r13 + r1  # r13=??
    print(r13)
    return

def emma(r1):
    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 = r1 + 42  # r1=4
    r13 = r1 + 1  # r1=46
    return r13  # return(47)
```

Function Calls…
def main():
    r1 = input()  # r1=3
    r13 = emma(r1)  # r13=47
    r13 = r13 + r1  # r13=54
    print(r13)  # 54
    return

def emma(r1):
    r1 = r1 + 1  # r1=4
    r13 = sarah(r1)  # r13=51
    r13 = r13 + r1  # r13=54
    return r13  # return(51)

def sarah(r1):
    r1 = r1 + 42  # r1=46
    r13 = r1 + 1  # r13=47
    return r13  # return(47)

Cool, but how does this work!?
The Stack!

- Insert ("push")
- Remove ("pop")

AFLAC, ShmAFLAC! Be careful up there!
Watch carefully…

What if I don’t give a hoot?!
def main():
    r1 = input()  # r1=3
    r13 = emma(r1)
    r13 = r13 + r1
    print(r13)
    return

def emma(r1):
    r1 = r1 + 1
    r13 = sarah(r1)
    r13 = r13 + r1
    return r13

def sarah(r1):
    r1 = r1 + 42
    r13 = r1 + 1
    return r13
def main():
    r1 = input()  # r1=3
    r13 = emma(r1)  # r13 = r13 + r1
    r13 = r13 + r1
    print(r13)
    return

def emma(r1):
    r1 = r1 + 1
    r13 = sarah(r1)  # r13 = r13 + r1
    r13 = r13 + r1
    return r13

def sarah(r1):
    r1 = r1 + 42
    r13 = r1 + 1
    return r13
def main():
    r1 = input()
    r1 = 3
    r13 = emma(r1)
    r13 = r13 + r1
    print(r13)
    return

def emma(r1):
    r1 = r1 + 1
    r1 = 4
    r13 = sarah(r1)
    r13 = r13 + r1
    return r13

def sarah(r1):
    r1 = r1 + 42
    r1 = 46
    r13 = r1 + 1
    return r13
def main():
    r1 = input()
    r13 = emma(r1)
    r13 = r13 + r1
    print(r13)
    return

def emma(r1):
    r1 = r1 + 1
    r13 = sarah(r1)
    r13 = r13 + r1
    return r13

def sarah(r1):
    r1 = r1 + 42
    r13 = r1 + 1
    return r13
def main():
    r1 = input()  # r1=3
    r13 = emma(r1)
    r13 = r13 + r1
    print(r13)
    return

def emma(r1):
    r1 = r1 + 1
    r13 = sarah(r1)
    r13 = r13 + r1
    return r13

def sarah(r1):
    r1 = r1 + 42
    r13 = r1 + 1
    return r13
def main():
    r1 = input()
    r13 = emma(r1)
    r13 = r13 + r1
    print(r13)
    return

def emma(r1):
    r1 = r1 + 1
    r13 = sarah(r1)
    r13 = r13 + r1
    return r13

def sarah(r1):
    r1 = r1 + 42
    r13 = r1 + 1
    return r13

It was better with pigs and geese!