Lecture 2: Introduction to Variables and Control Flow

Armando Solar-Lezama

MIT COMPUTER SCIENCE AND ARTIFICIAL INTELLIGENCE LABORATORY

Course Policies

Grading

- Problem sets: 25%
  Quiz I: 15% (March 7)
  Quiz II: 20% (April 11) (Changed from lecture)
  Final: 35%
  Participation: 5%

MITx page up

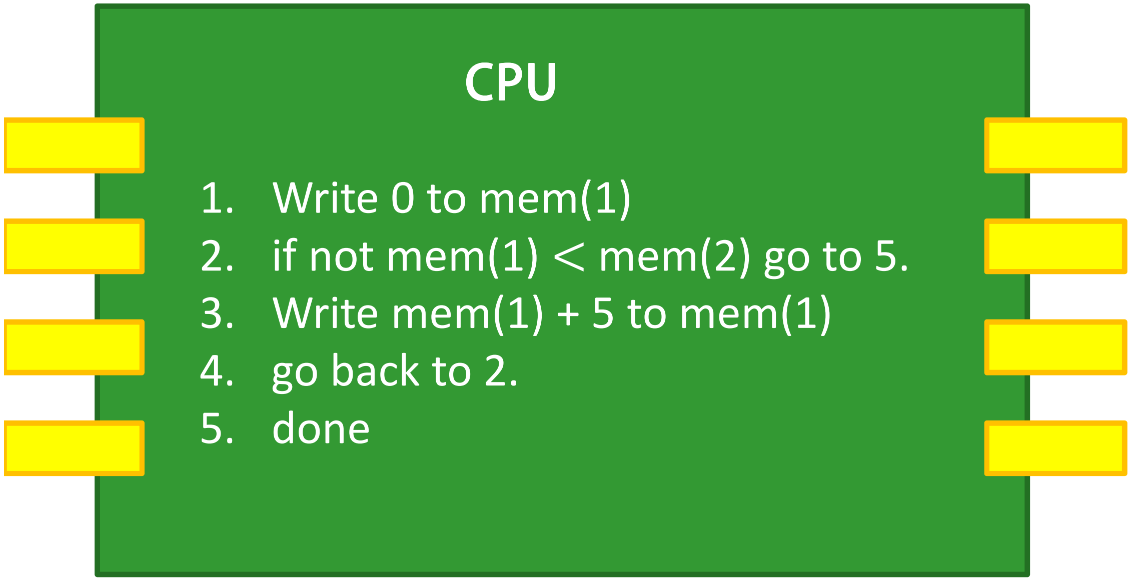
https://mit.edx.org/courses/MITx/6.00/MIT_2013_Spring/about

Lot's of resources to help you with the material in the course!

A simple computer

Just a (really big) table that stores values at different addresses

Ex: Make mem(1) equal to the lowest multiple of 5 ≥ mem(2)



CPU

1. Write 0 to mem(1)
2. if not mem(1) < mem(2) go to 5.
3. Write mem(1) + 5 to mem(1)
4. go back to 2.
5. done

Understands a very small number of instructions including

a) reads and updates to memory
b) basic arithmetic among memory locations
c) conditional jumps to other instructions

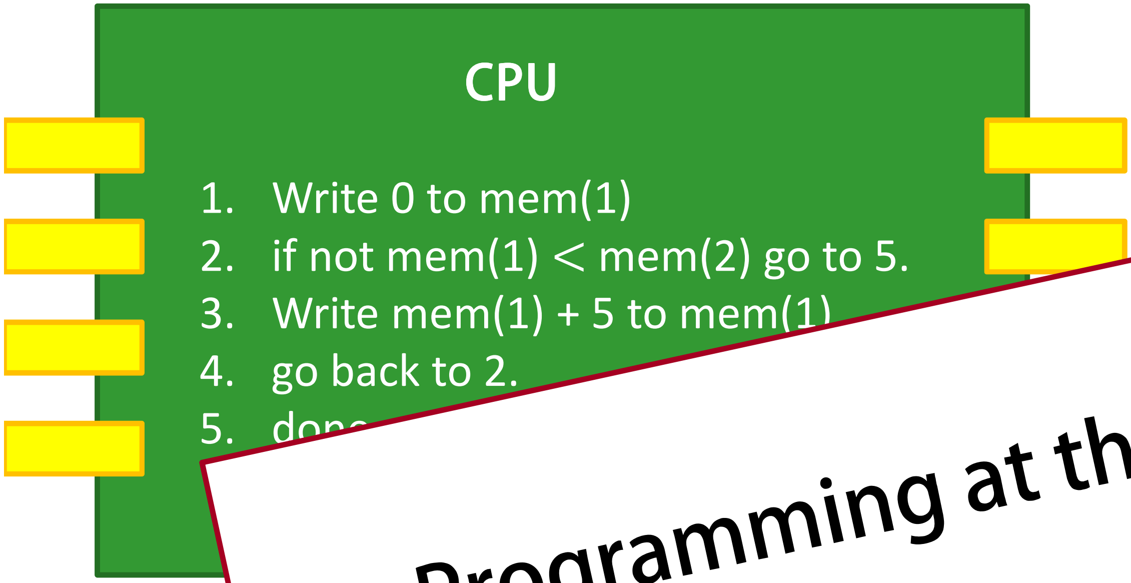
Random Access Memory (RAM)

1	55
2	23
3	13
4	44
…	…
123456	77
123457	109874
…	…

A simple computer

Just a (really big) table that stores values at different addresses

Ex: Make mem(1) equal to the lowest multiple of 5 ≥ mem(2)



CPU

1. Write 0 to mem(1)
2. if not mem(1) < mem(2) go to 5.
3. Write mem(1) + 5 to mem(1)
4. go back to 2.
5. don

Random Access Memory (RAM)

123456	77
123457	109874
…	…

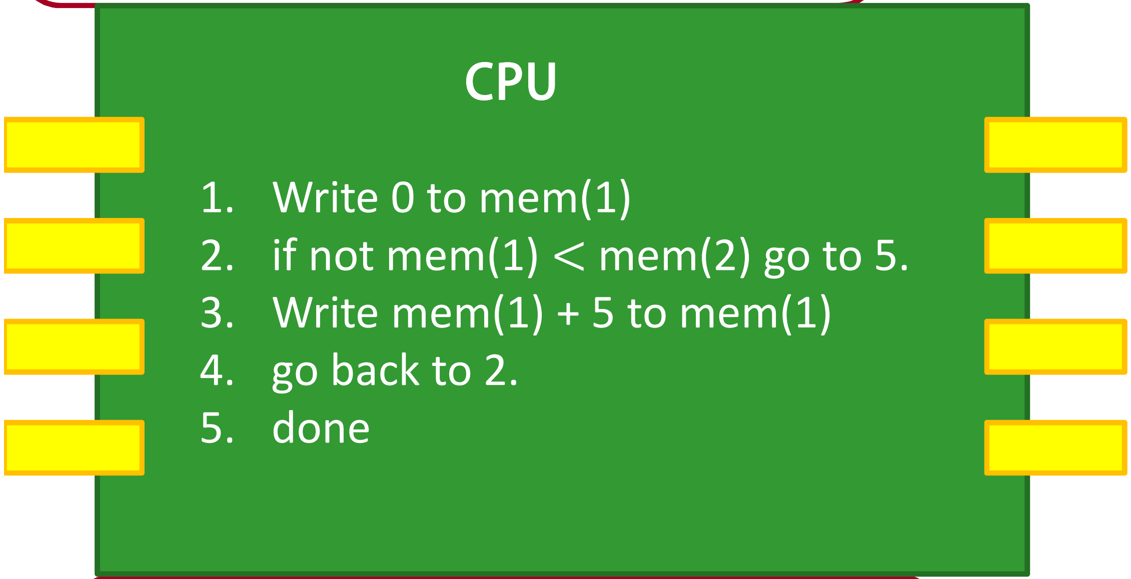
Programming at this level is hard!!

Underst ctions
includin
a) reads ates to memory
b) basic arithmetic among memory locations
c) conditional jumps to other instructions

Programming at this level is hard!!

It takes thousands of instructions to do even relatively simple things. This would be a lot of code to write.

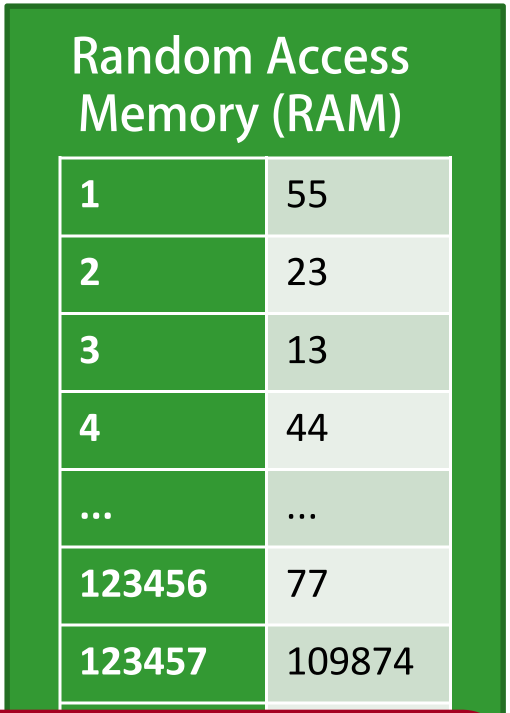
If you only have one big memory, it's hard to remember where you put what.



CPU

```
1. Write 0 to mem(1)
2. if not mem(1) < mem(2) go to 5.
3. Write mem(1) + 5 to mem(1)
4. go back to 2.
5. done
```

It's hard to reason about code that jumps all over the place.



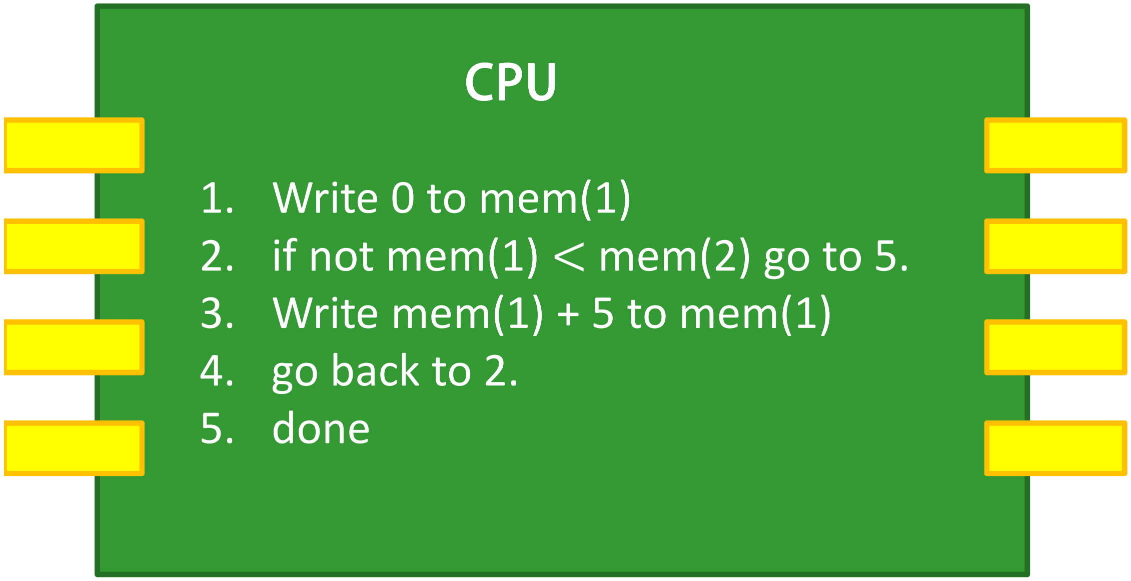
Random Access Memory (RAM)

1	55
2	23
3	13
4	44
…	…
123456	77
123457	109874

It's also hard to keep track of what the values in memory mean.

Programming at this level is hard!!

If you only have one big memory, it's hard to remember where you put what.



CPU

1. Write 0 to mem(1)
2. if not mem(1) < mem(2) go to 5.
3. Write mem(1) + 5 to mem(1)
4. go back to 2.
5. done

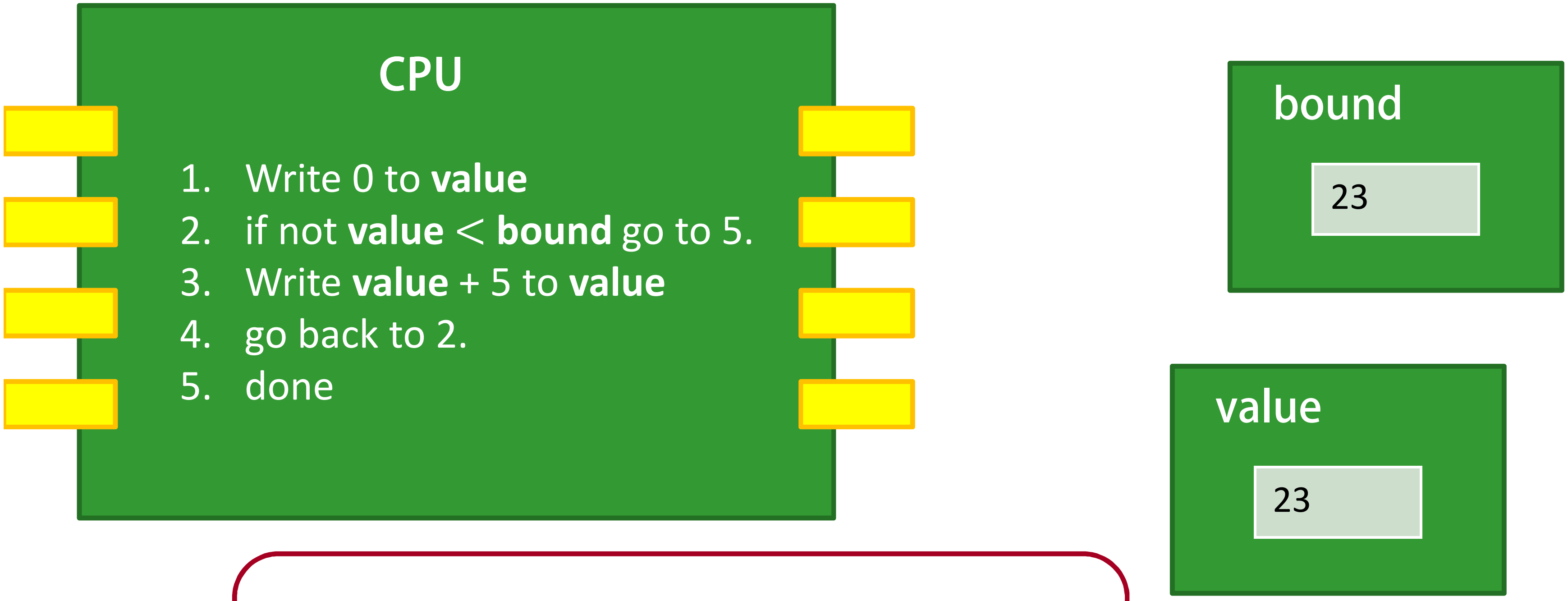
Random Access Memory (RAM)

1	55
2	23
3	13
4	44
…	…
123456	77
123457	109874
…	…

Solution: Have as many memories as you want with meaningful names to help you remember what they are for.

Programming languages to the rescue

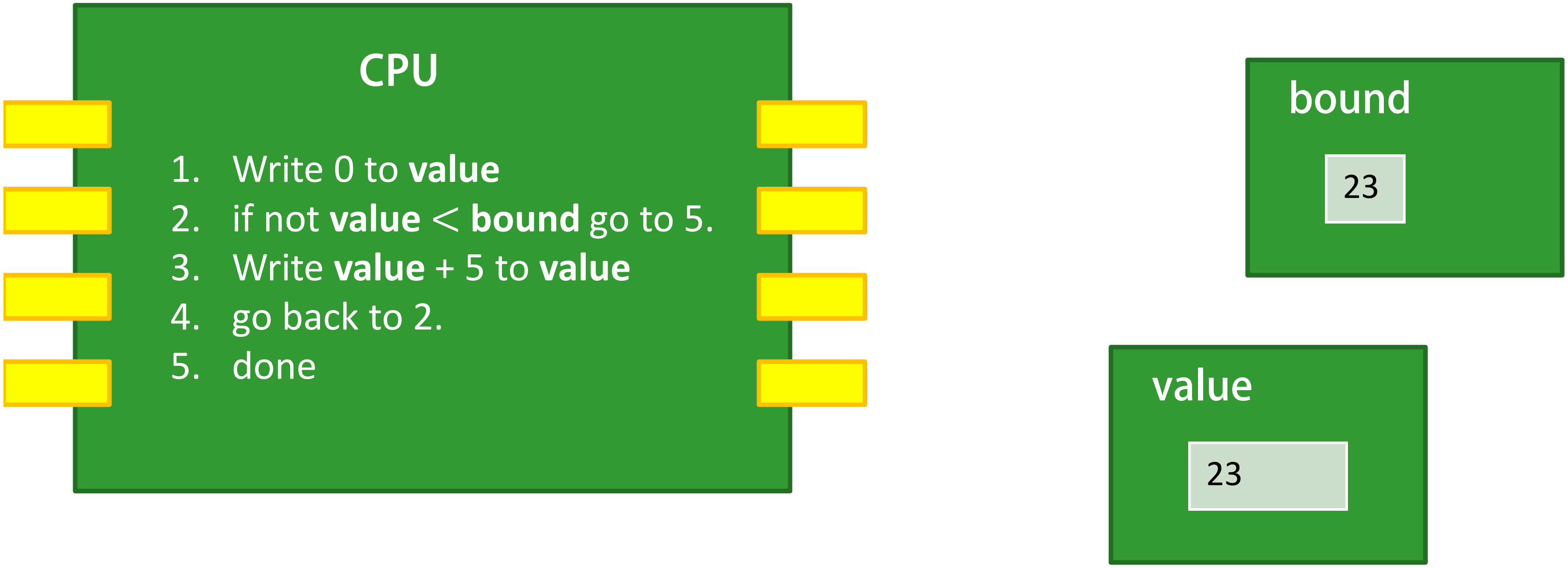
If you only have one big memory, it's hard to remember where you put what.



1. Write 0 to **value**
2. if not **value** < **bound** go to 5.
3. Write **value** + 5 to **value**
4. go back to 2.
5. done

Solution: Have as many memories as you want with meaningful names to help you remember what they are for.

Programming languages to the rescue

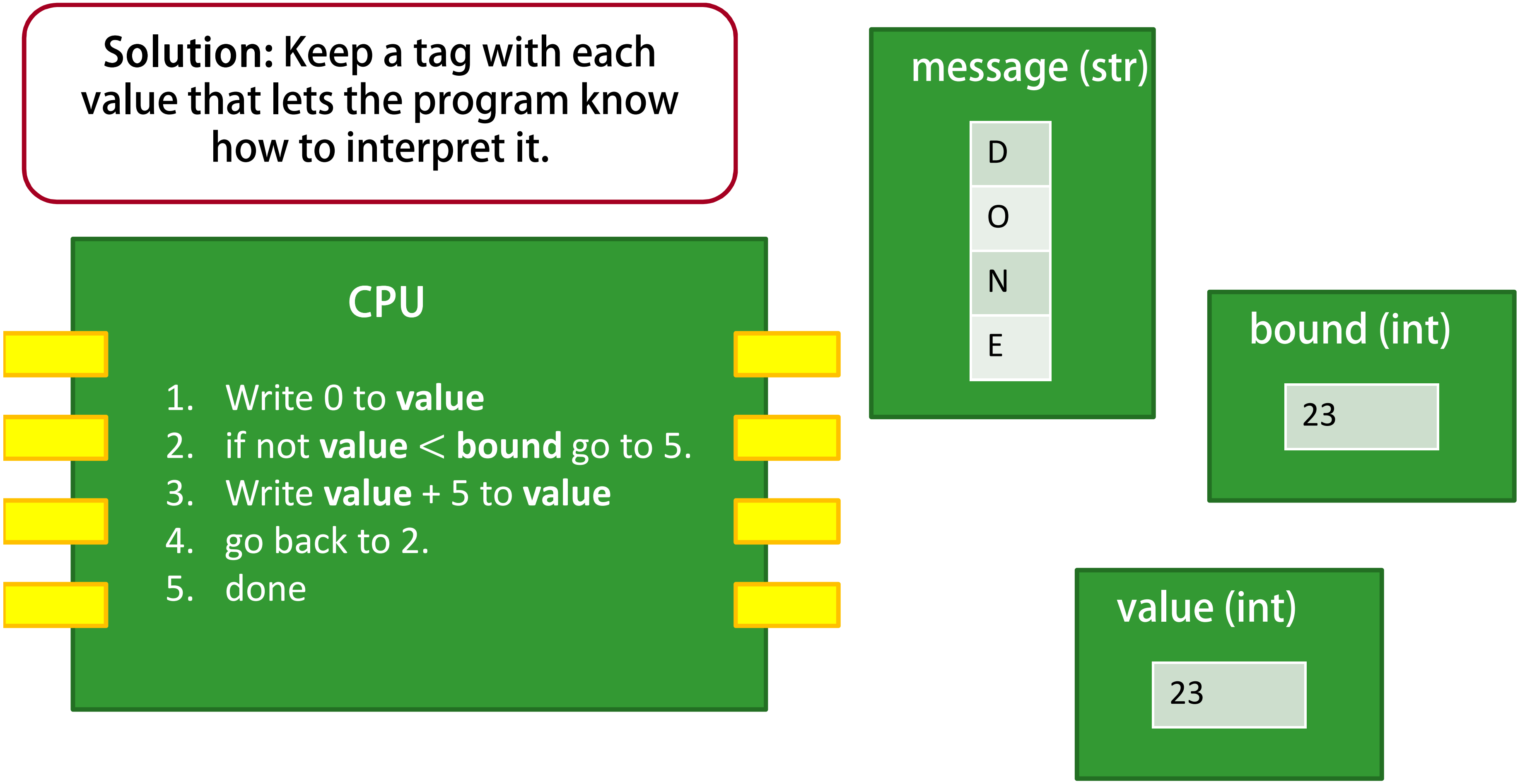


```
1. Write 0 to value
2. if not value < bound go to 5.
3. Write value + 5 to value
4. go back to 2.
5. done
```

It’s also hard to keep track of what the values in memory mean.

Programming languages to the rescue

Solution: Keep a tag with each value that lets the program know how to interpret it.



CPU

1. Write 0 to **value**
2. if not **value** < **bound** go to 5.
3. Write **value** + 5 to **value**
4. go back to 2.
5. done

message (str)

D
O
N
E

bound (int)

23

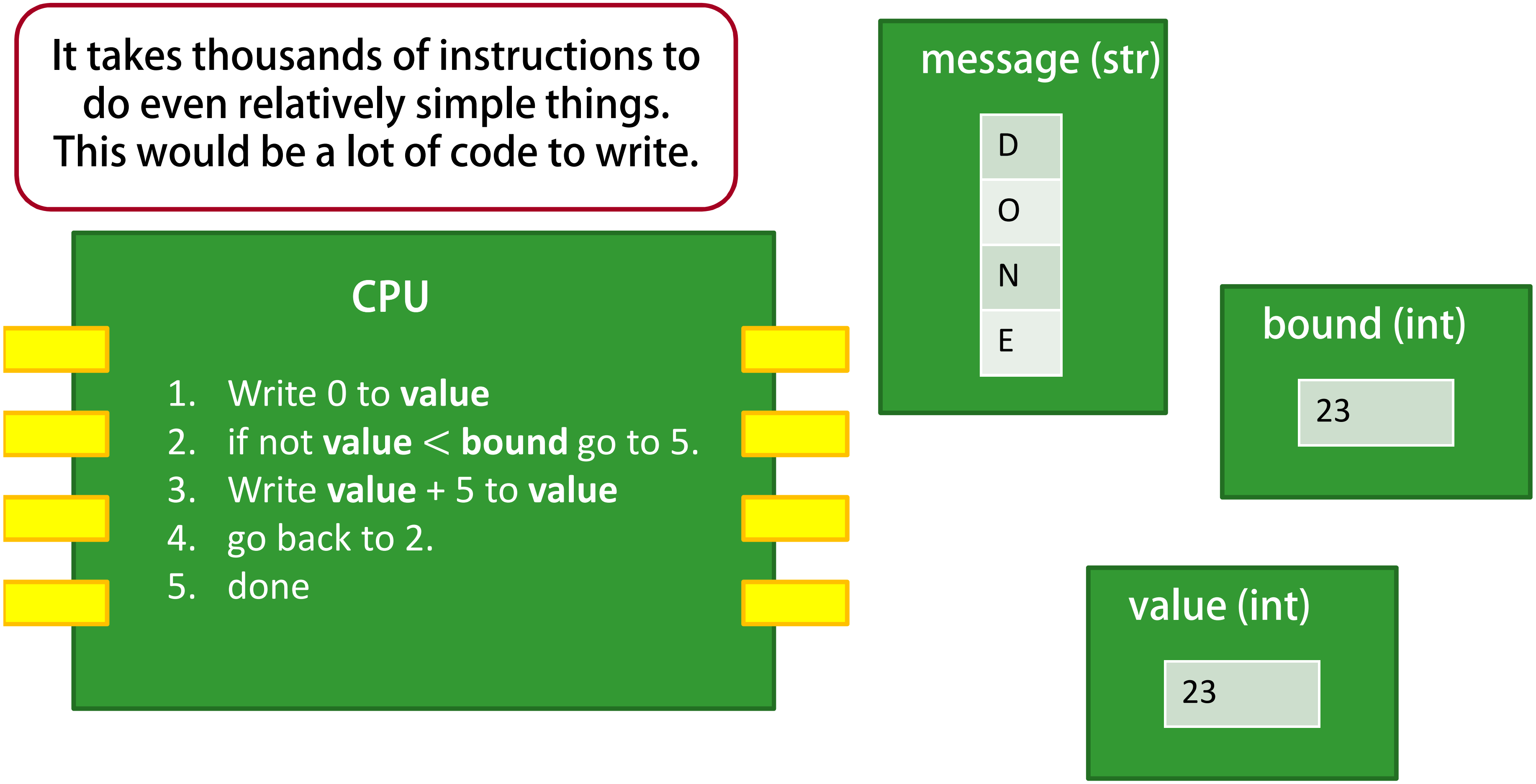
value (int)

23

It's also hard to keep track of what the values in memory mean.

Programming languages to the rescue

It takes thousands of instructions to do even relatively simple things. This would be a lot of code to write.



CPU

1. Write 0 to **value**
2. if not **value** < **bound** go to 5.
3. Write **value** + 5 to **value**
4. go back to 2.
5. done

message (str)

D
O
N
E

bound (int)

23

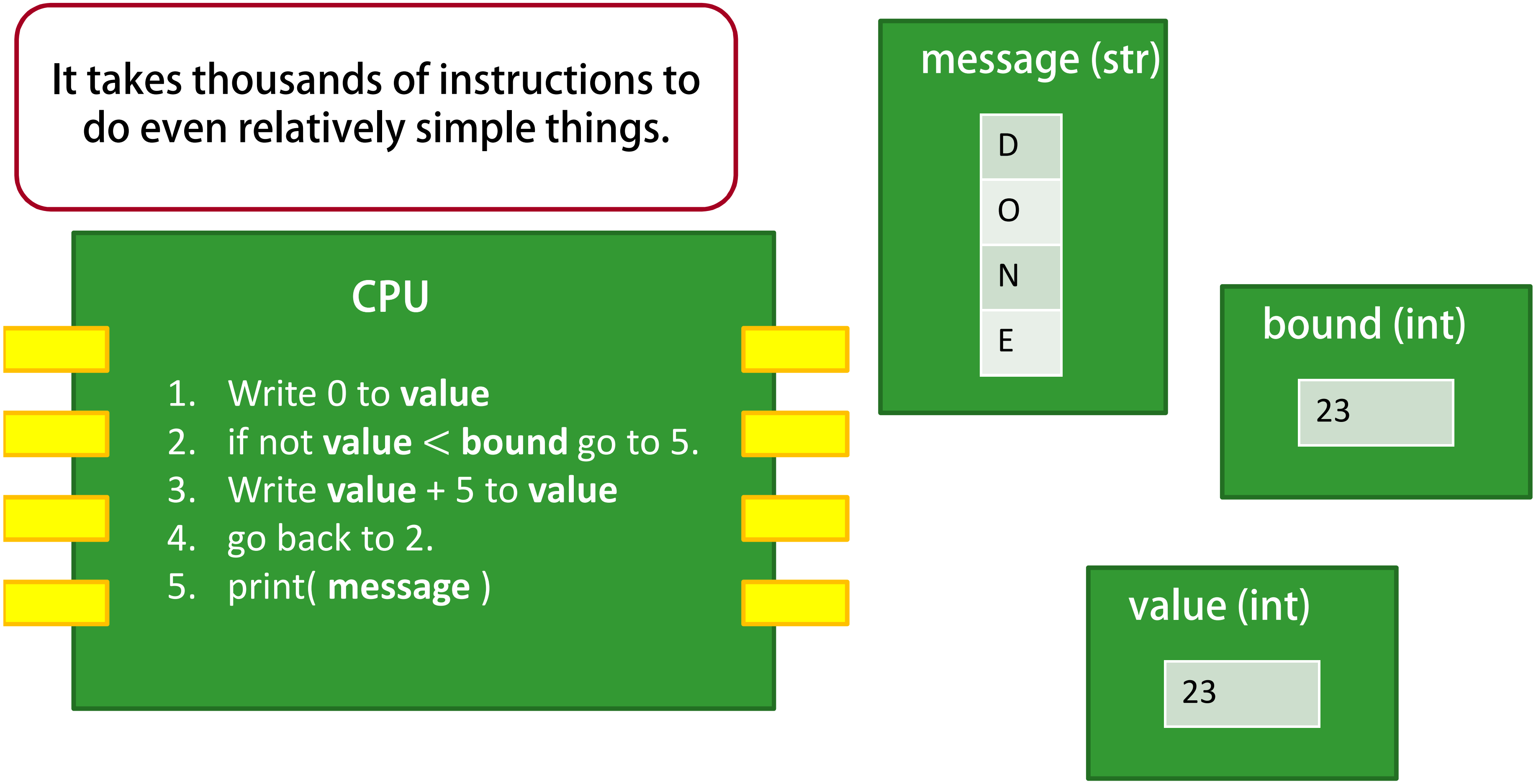
value (int)

23

Solution: Package complex sequences of instructions under easy-to-use procedures with intuitive names.

Programming languages to the rescue

It takes thousands of instructions to do even relatively simple things.

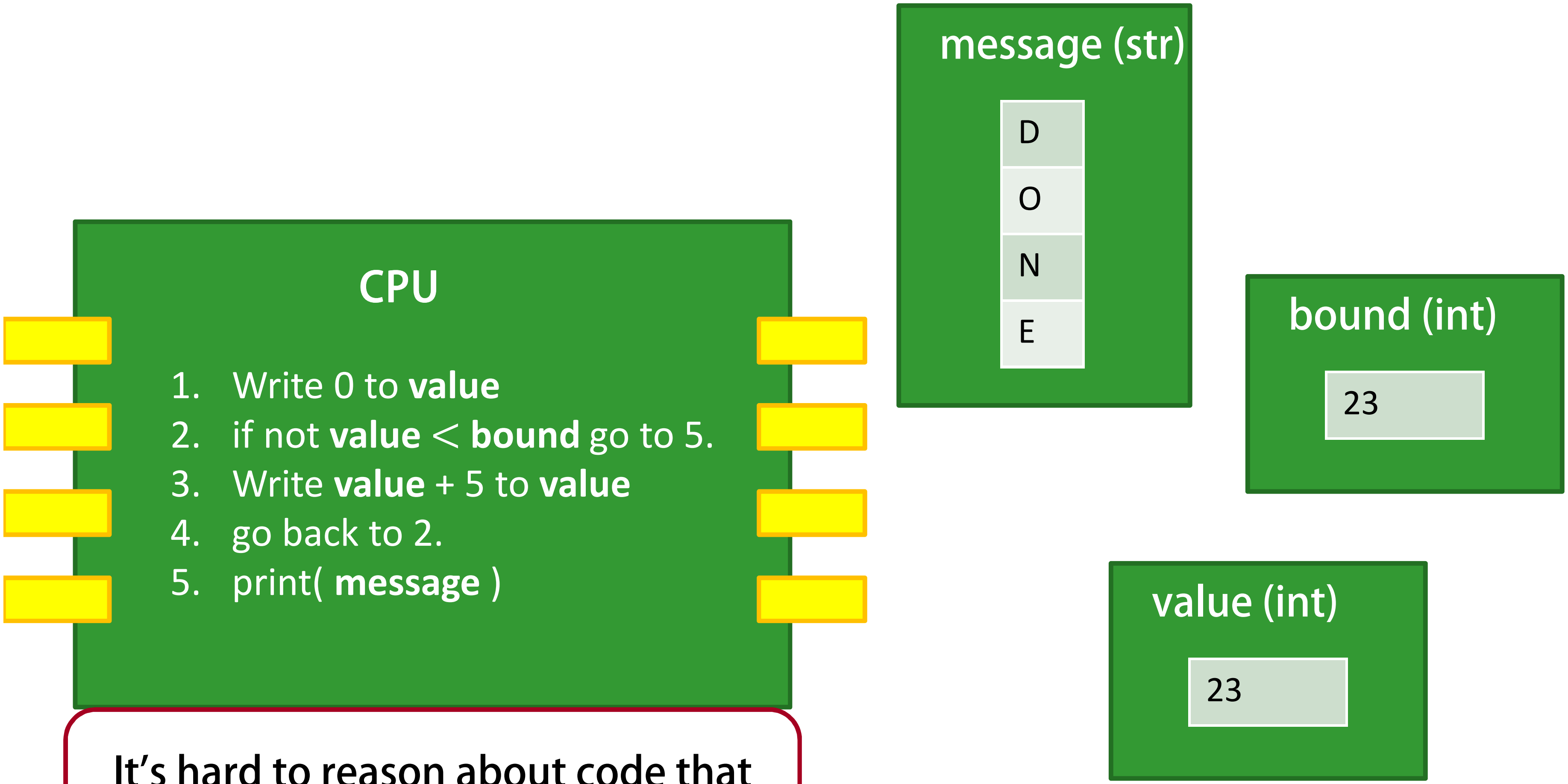


CPU

1. Write 0 to **value**
2. if not **value** < **bound** go to 5.
3. Write **value** + 5 to **value**
4. go back to 2.
5. print(**message**)

Solution: Package complex sequences of instructions under easy-to-use procedures with intuitive names.

Programming languages to the rescue



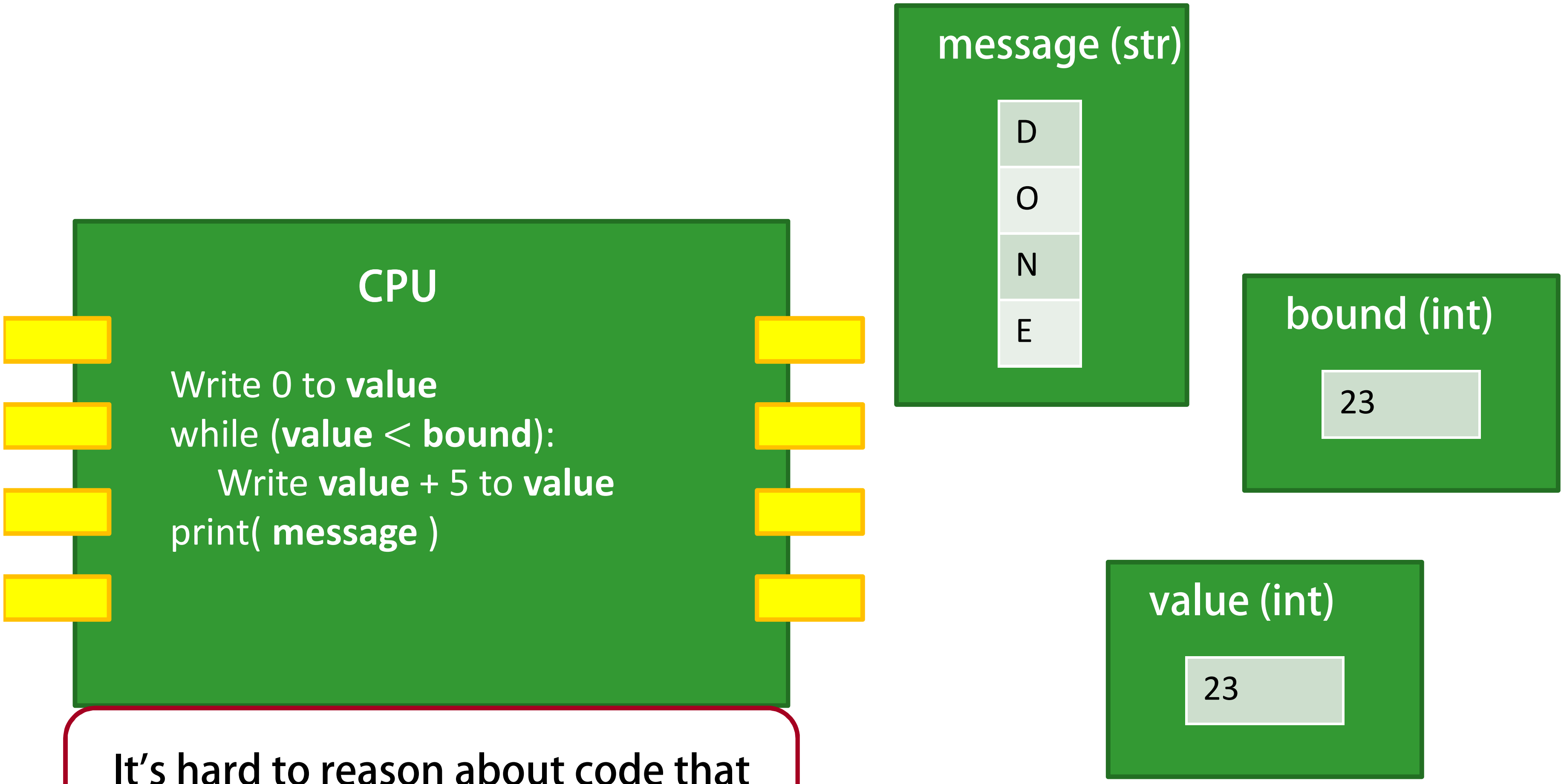
CPU

```
1.  Write 0 to value
2.  if not value < bound go to 5.
3.  Write value + 5 to value
4.  go back to 2.
5.  print( message )
```

It’s hard to reason about code that jumps all over the place.

Solution: Provide high-level control constructs

Programming languages to the rescue



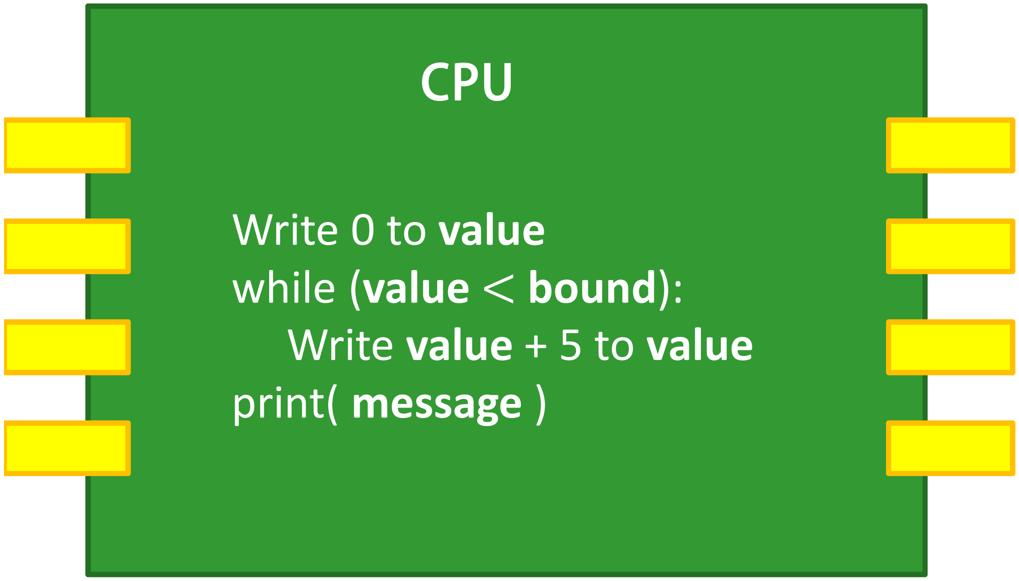
CPU

```
Write 0 to value
while (value < bound):
    Write value + 5 to value
print( message )
```

It’s hard to reason about code that jumps all over the place.

Solution: Provide high-level control constructs

Programming languages to the rescue



```
Write 0 to value
while (value < bound):
    Write value + 5 to value
print( message )
```

$x = 0$

$x = 5$

```
value = 0
while (value < bound):
    value =  value + 5
print( message )
```

This is what the program looks like in python

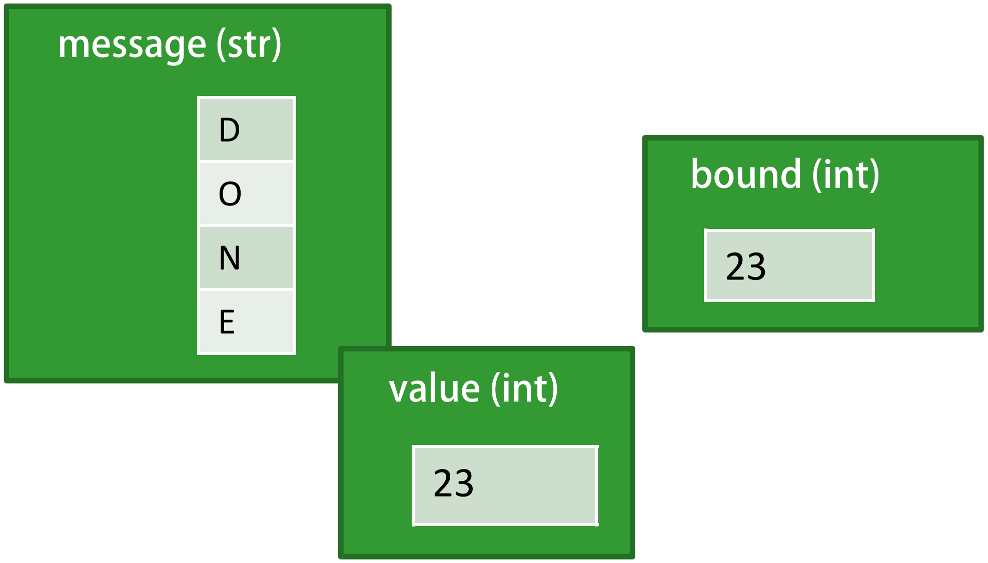
Key points

A variable is a name for a piece of memory

- assignment changes what that memory contains.

Lines are executed in sequence

- while repeats its body until the condition is satisfied



message (str)

D
O
N
E

bound (int)

23

value (int)

23

Running the examples

You will need to install python.

- Follow the Getting Started Guide from Pset 0

  http://bit.ly/UFhXVo

- If you want to use the animations you need to install matplotlib and numpy as well
  - Instructions are also in the Getting Started Guide

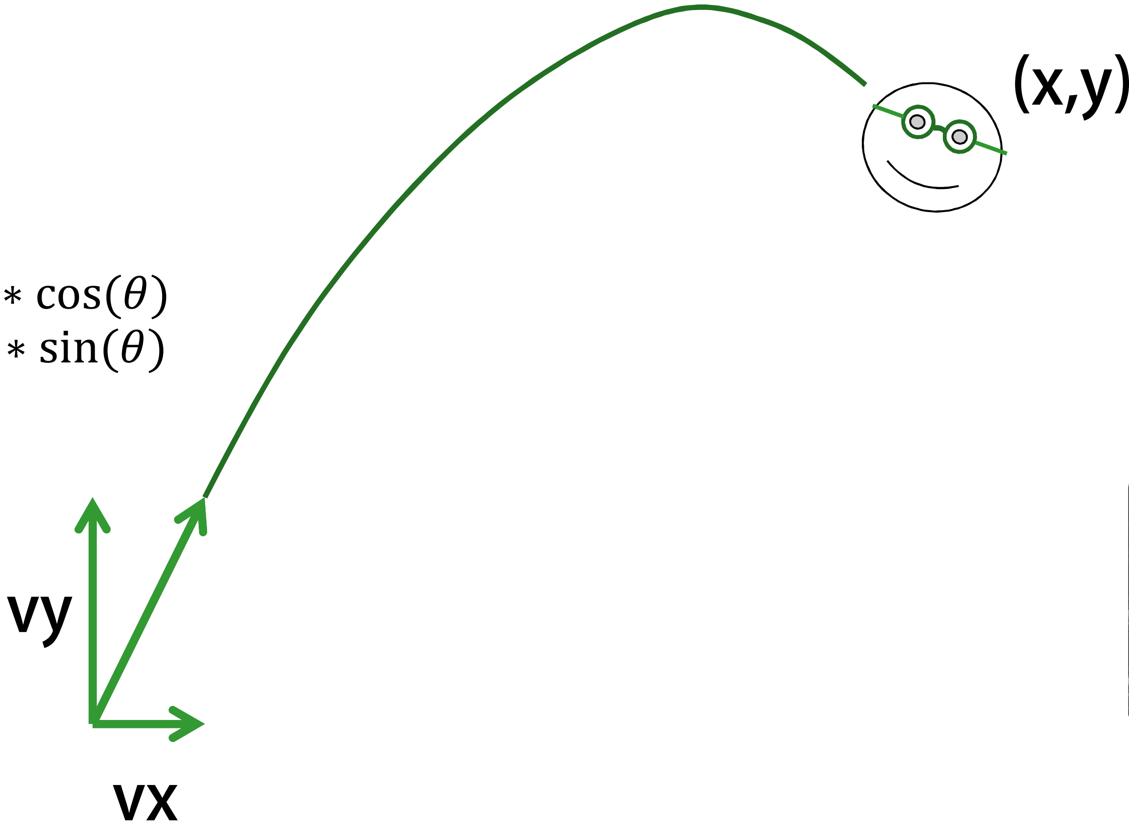
- Finally, if you want to use the simpleplot trajectory drawing you will need to have the simpleplot.py file in the same directory as your file

  http://bit.ly/YWyCoo

Angry Nerds

$$\Delta x = vx * \Delta t$$
$$\Delta y = vy * \Delta t + \frac{1}{2} g * \Delta t^2$$
$$\Delta vy = g * \Delta t$$

$$vx = v * \cos(\theta)$$
$$vy = v * \sin(\theta)$$



Code

```
import math
import simpleplot as sp

g = -9.8
dt = 0.01;

x = 0.1
y = 0.1
v = 25.0
ang = 30.0
vx = v*math.cos((ang/ 180.0) * math.pi)
vy = v*math.sin((ang/ 180.0) * math.pi)

while y > 0.0:
    x = x + vx*dt
    y = y + vy*dt + g*dt*dt/2
    vy = vy + g*dt
    sp.plotTrajectory((x,y))

print x
sp.doAnimation()
```

Programming in the old days