W2: Computer Architecture

With a history

What is the binary code of room 209

What is the binary code of room 209

2** 7 = 128	10 000 000	209 – 128 = 81
2** 6 = 64	1 000 000	81 – 64 = 17
2** 4 = 16	10 000	17 – 16 = 1
2 ** 0 = 1	1	
Add every binary number together	**11 010 001**	

So the answer is b(11010001)

What is the binary code of room 209

- 209 % 2 = 1 209 // 2 = 104
- 104 % 2 = 0 104 // 2 = 52
- 52 % 2 = 0 52 // 2 = 26
- 26 % 2 = 0 26 // 2 = 13
- 13 % 2 = 1 13 // 2 = 6
- 6 % 2 = 0 6 // 2 = 3
- 3 % 2 = 1 3 // 2 = 1
- 1 % 2 = 1

- So the answer is b(11010001)

Today

- BBC clips (#3, #4, #5)
- Inter-mixed with fundamentals of computer architectures

- State machine
- Program exercise: the register machine

BBC History of Computers

Clip #3

Takeaways

- Computer as a “crystal ball” has a long (and also short) history
  - ”Big data” often claims machines (algorithms + data) knows better!

- Which kind of ideas excite you more?
  - Bright ideas
  - Good ideas
  - Successful ideas

BBC History of Computers

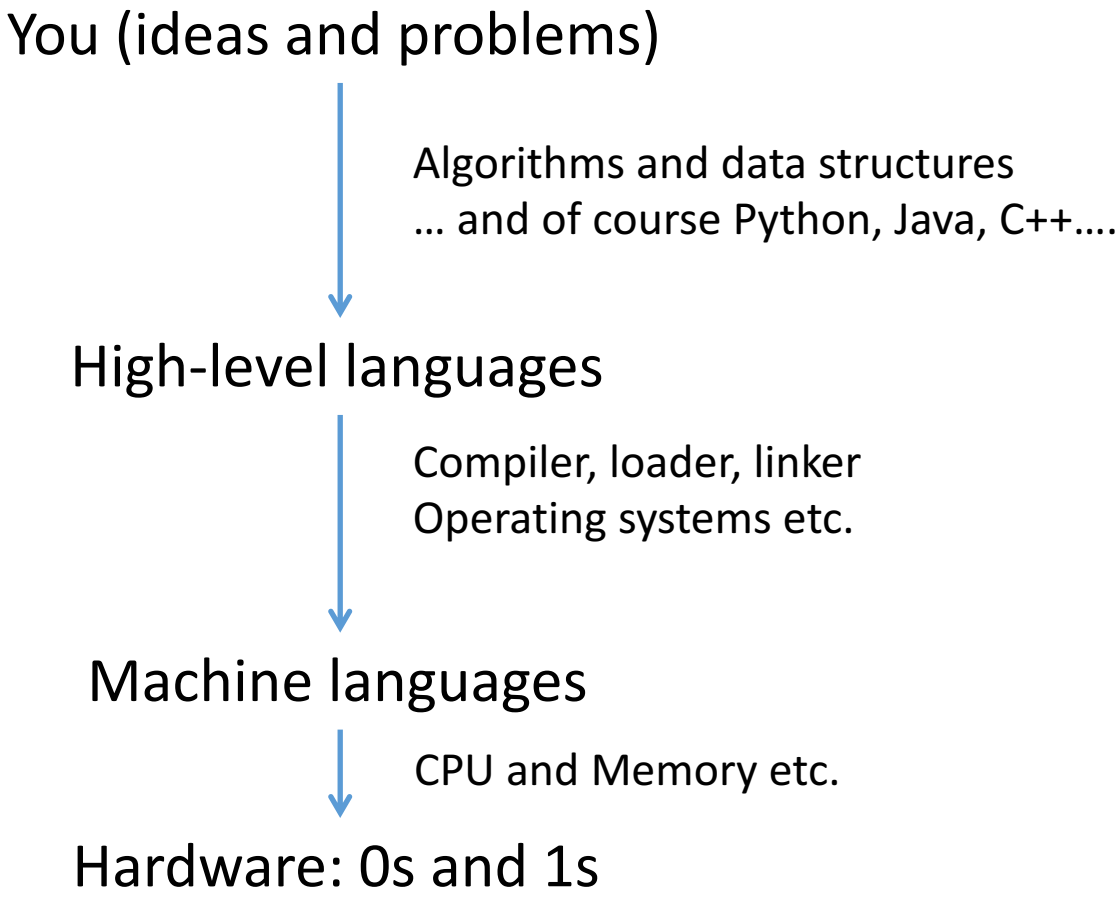
Clip #4

Layers and abstractions

You (ideas and problems)

Hardware: 0s and 1s

Layers and abstractions



Barbara Liskov (MIT)
The power of abstraction
Turning award speech
http://amturing.acm.org/vp/liskov_1108679.cfm

Abstraction

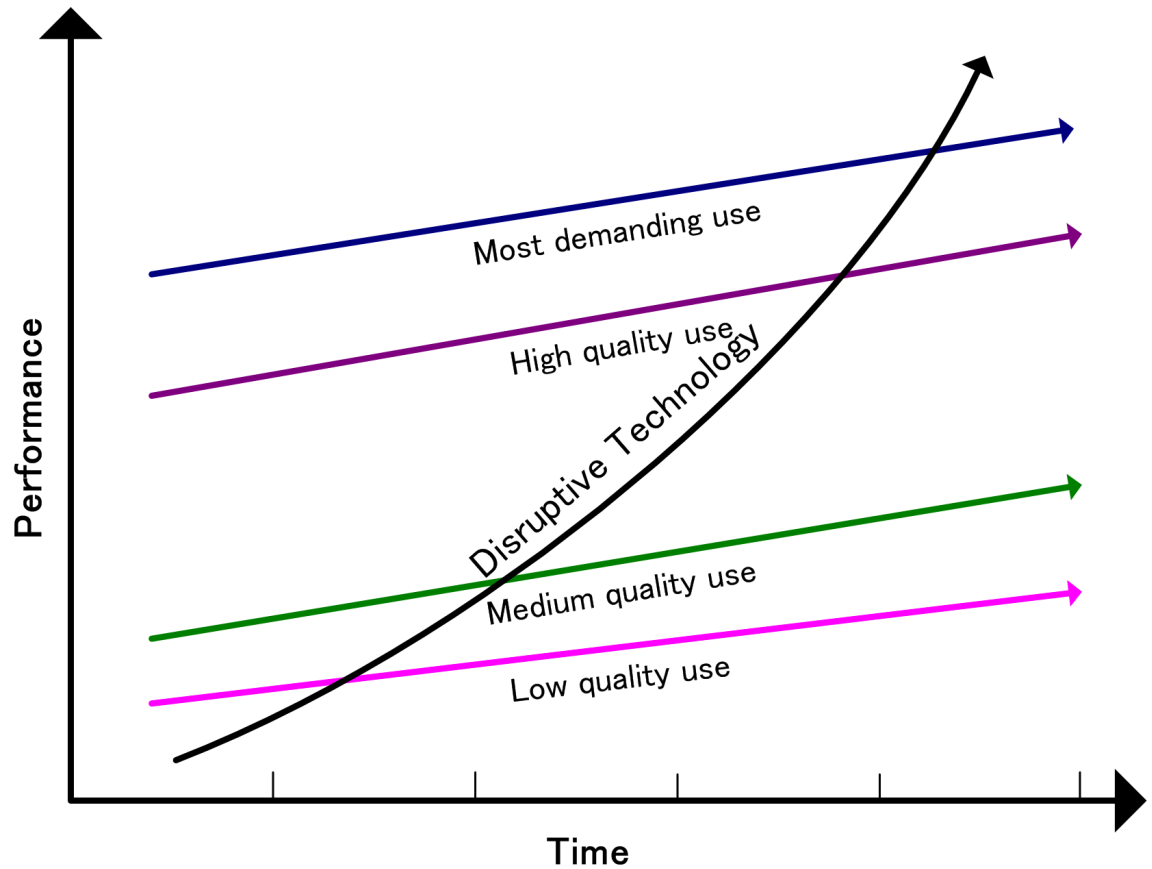
- Define interface, *not* implementation
- The isolation improves productivity

- Examples:
  - Python language syntax ➔
  - OS APIs ➔
  - Intel x86 ISAs (instruction set architecture) ➔
  - CPU/Memory bus signals/actions

BBC History of Computers

Clip #5

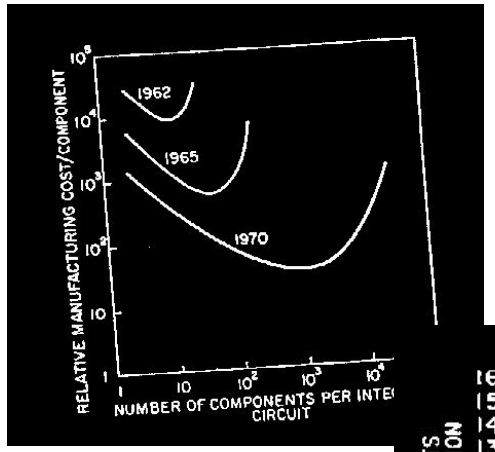
The Disruptive Technology LAW

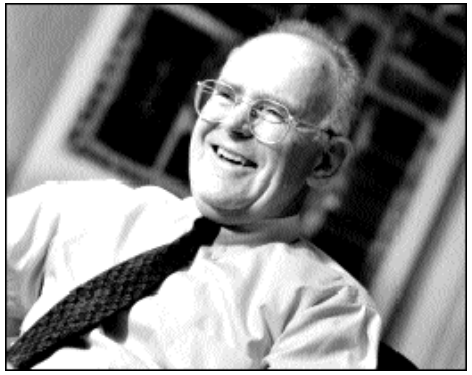


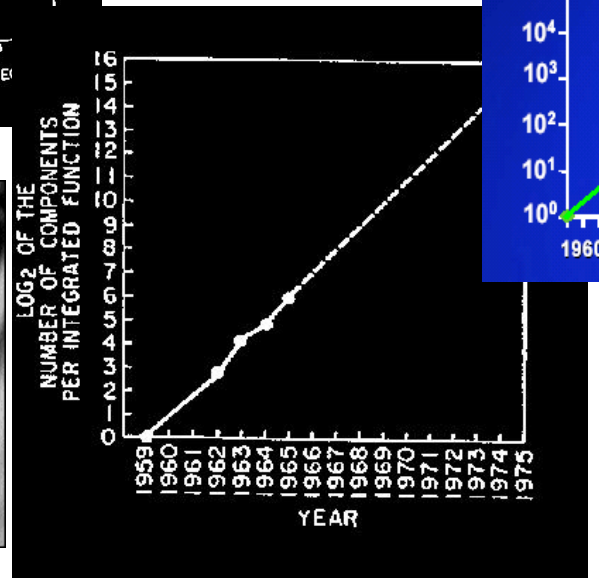
Examples:

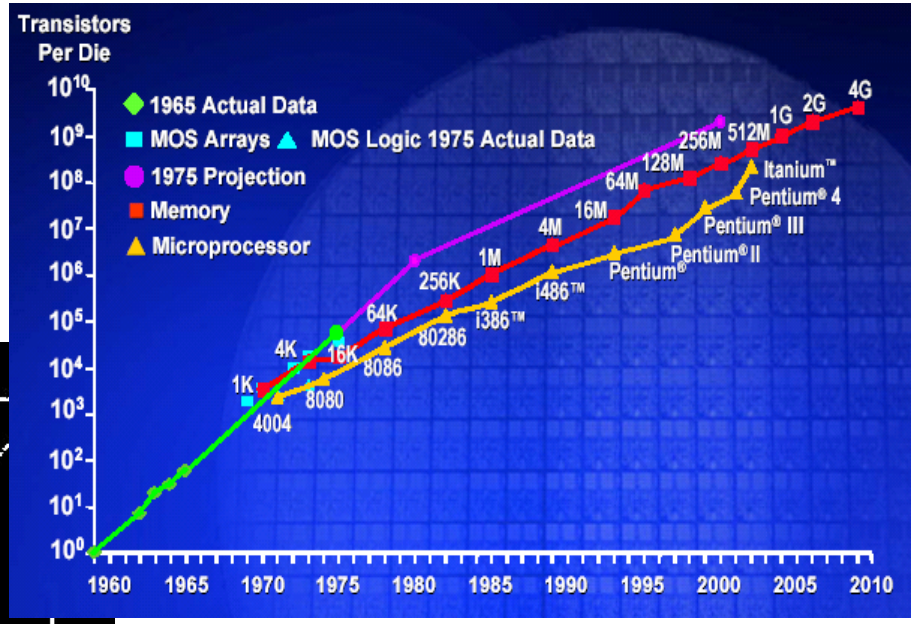
- Cell phone
- Flash disk
- PC
- What's in this clip?

The Moore's LAW





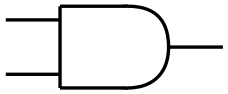
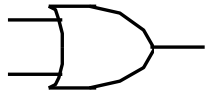
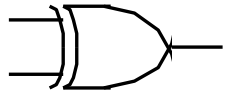




# of transistors on cost-effective integrated circuit double every 18 months

- "Cramming More Components onto Integrated Circuits"
  - Gordon Moore, Electronics, 1965
- Hitting a limit because of "power wall" and "memory wall"

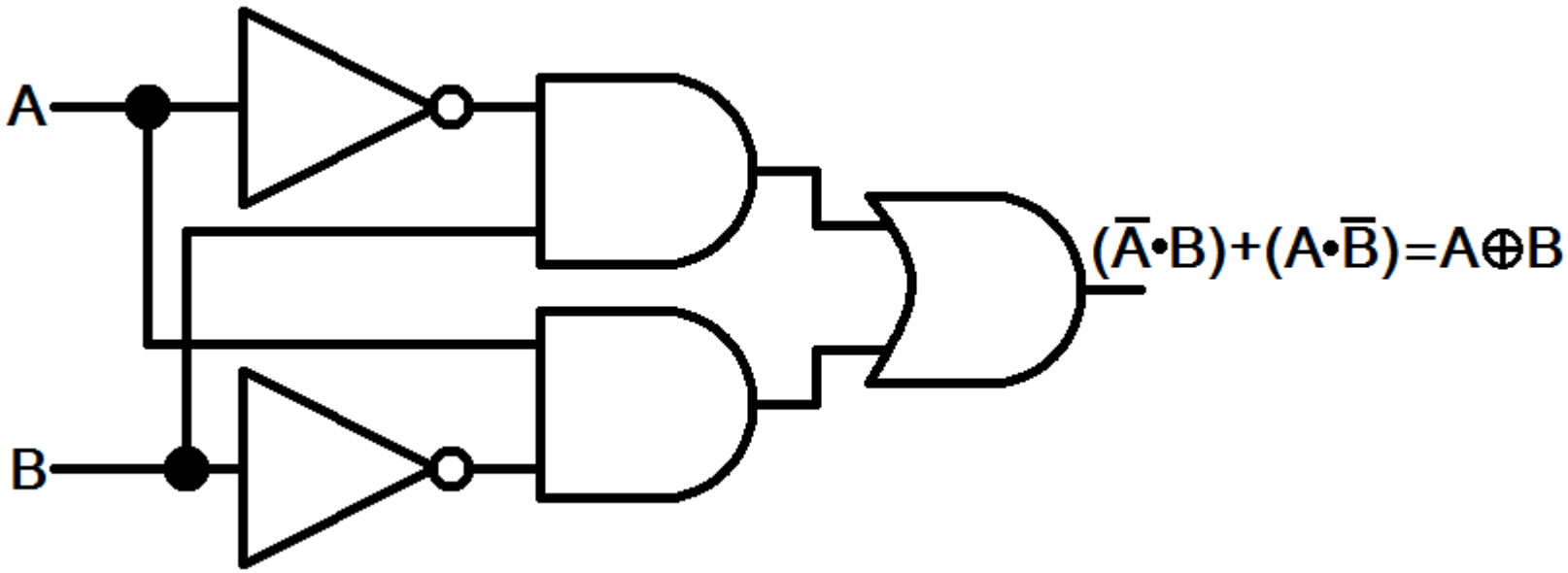
Signals, Logic Operators, and Gates

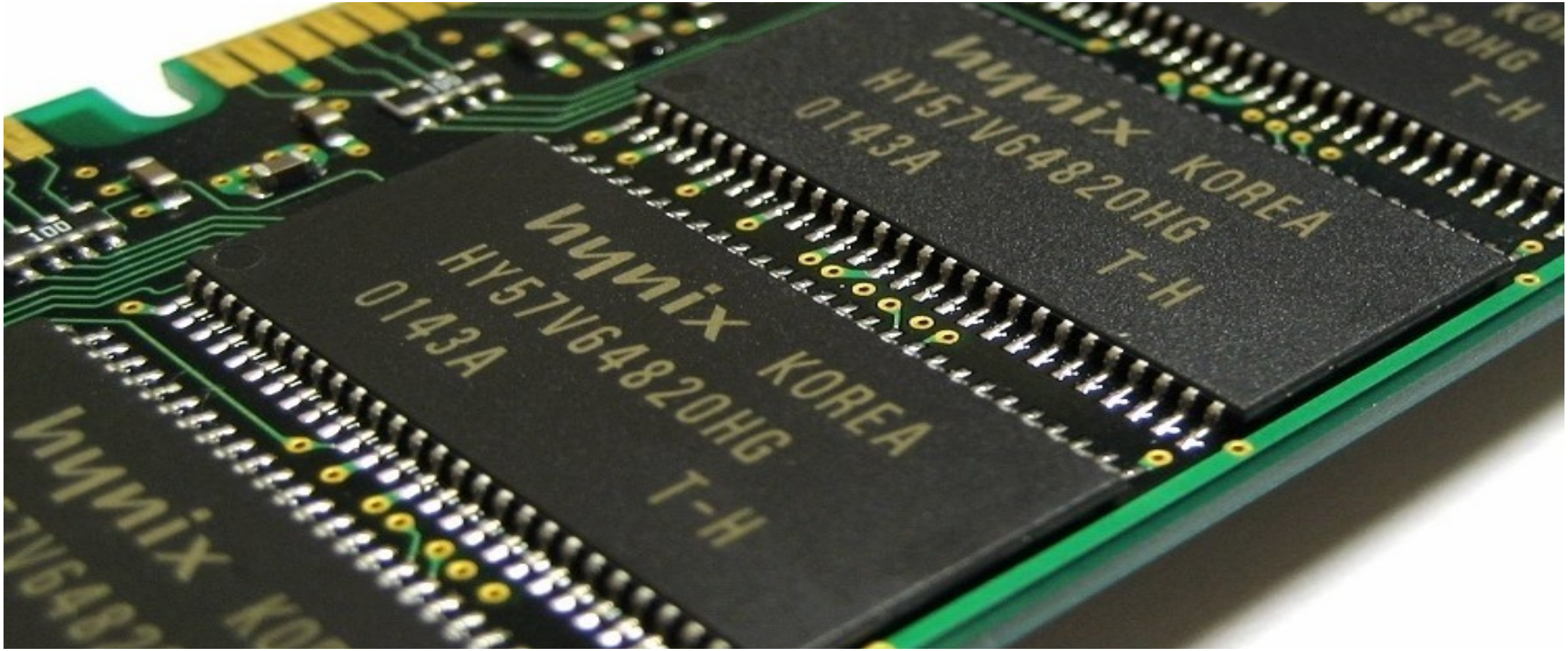
Name	NOT	AND	OR	XOR
Graphical symbol				
Operator sign and alternate(s)	x' $\neg x$ or $\overline{x}$	xy $x \wedge y$	$x \vee y$ $x + y$	$x \oplus y$ $x \not\equiv y$
Output is 1 iff:	Input is 0	Both inputs are 1s	At least one input is 1	Inputs are not equal
Arithmetic expression	$1 - x$	$x \times y$ or xy	$x + y - xy$	$x + y - 2xy$

The state of the input (x, y) decides the state of output (y)

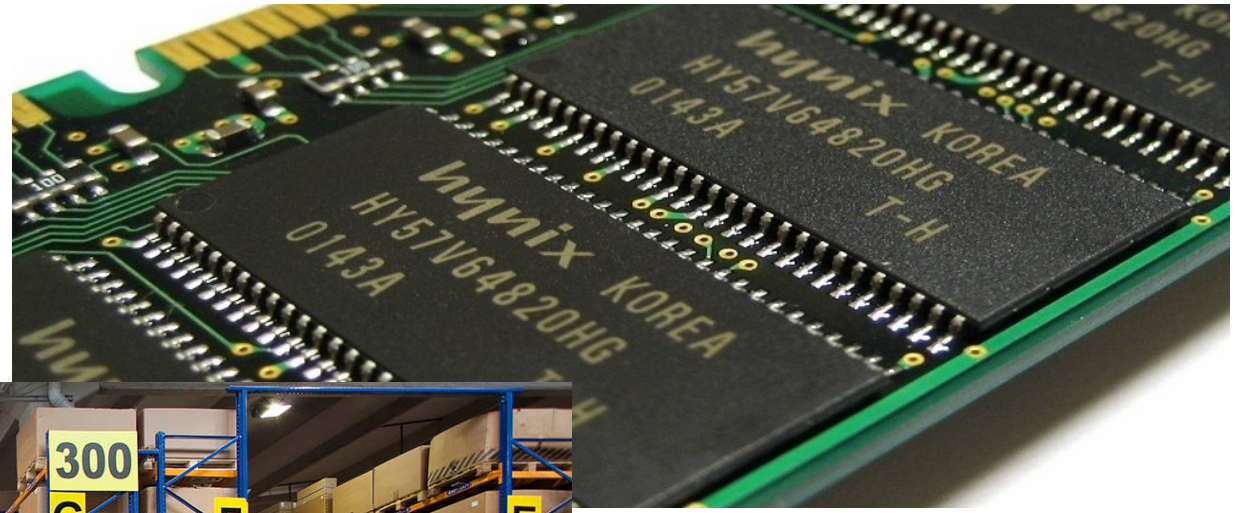
What does this circuitry do?

- Composing more complex functions out of simpler ones

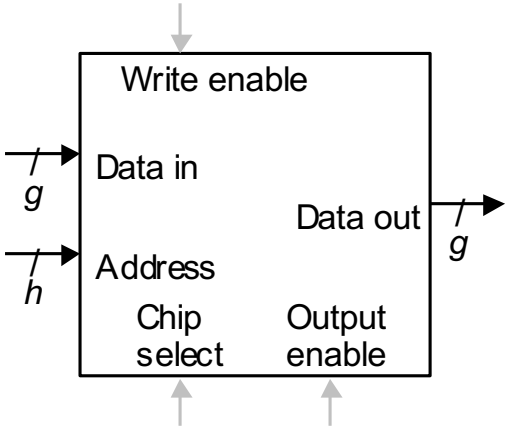



hynix
HY57V64820HG
0143A
KOREA
T-H

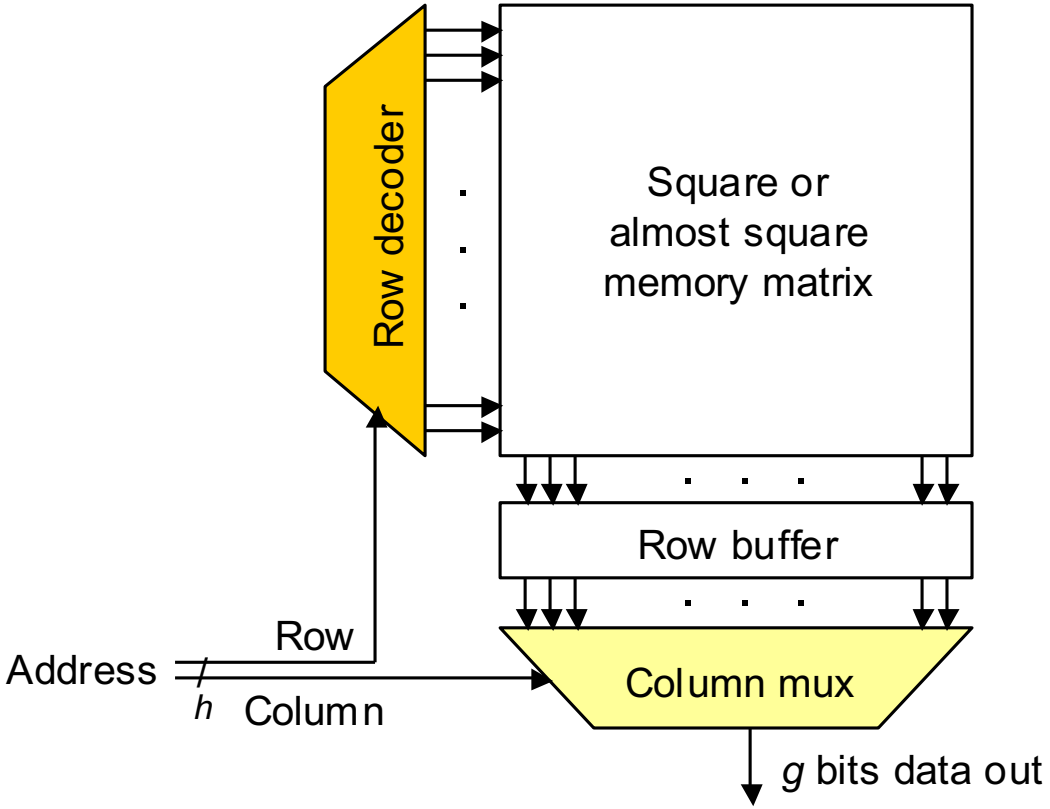
Not that different…



SRAM (the memory on your PC)



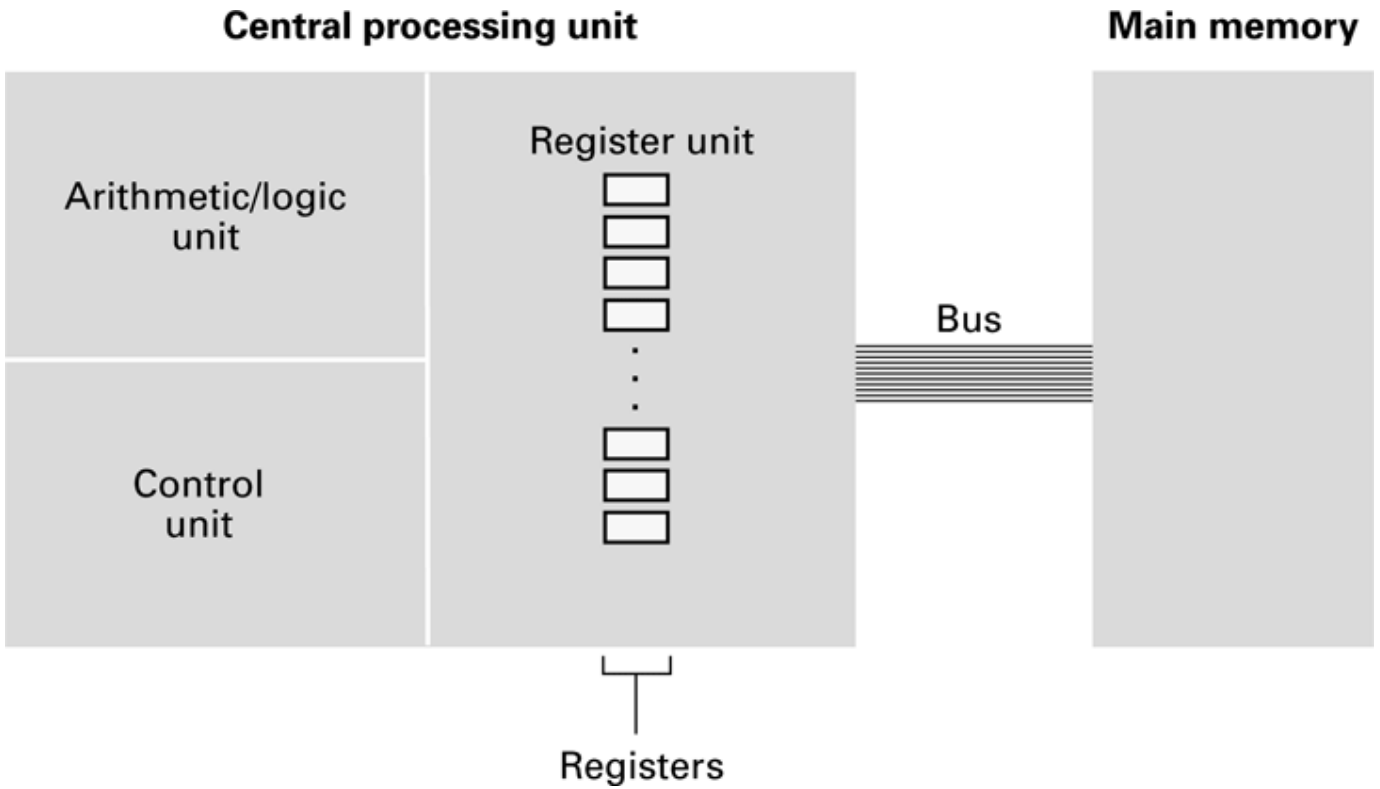
(a) SRAM block diagram



(b) SRAM read mechanism

Computer Architecture

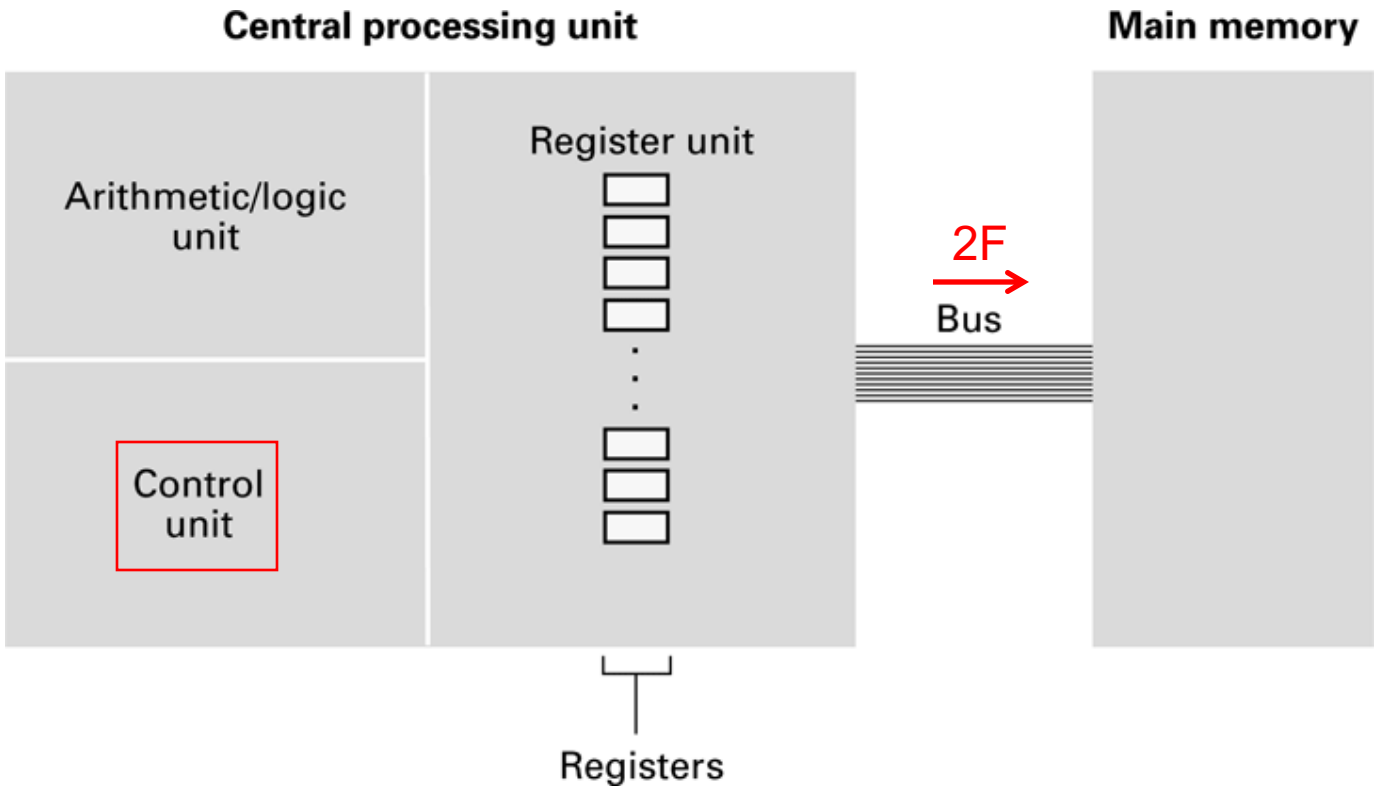
(microprocessor)



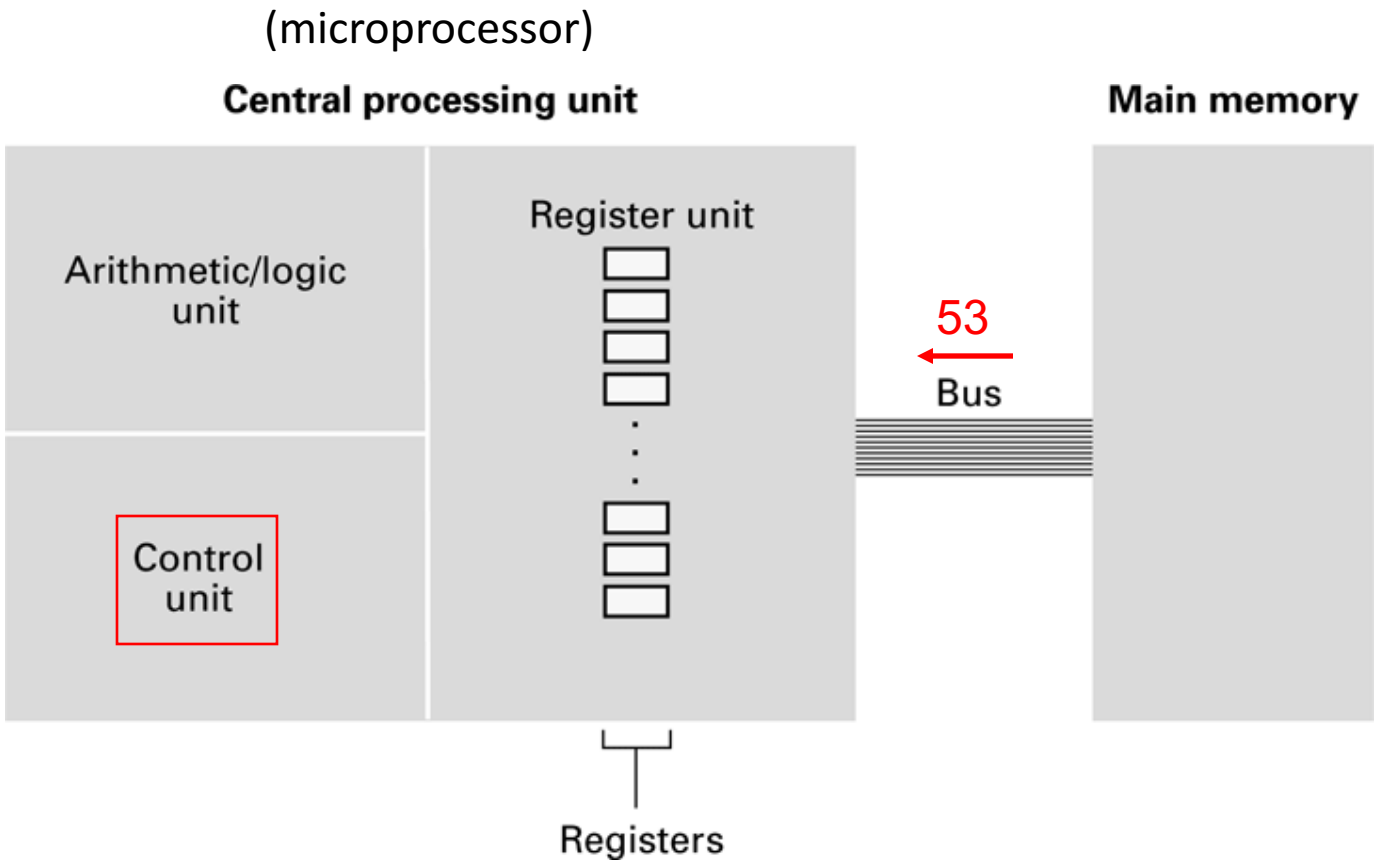
(Shamelessly copied from Keith)

Computer Architecture

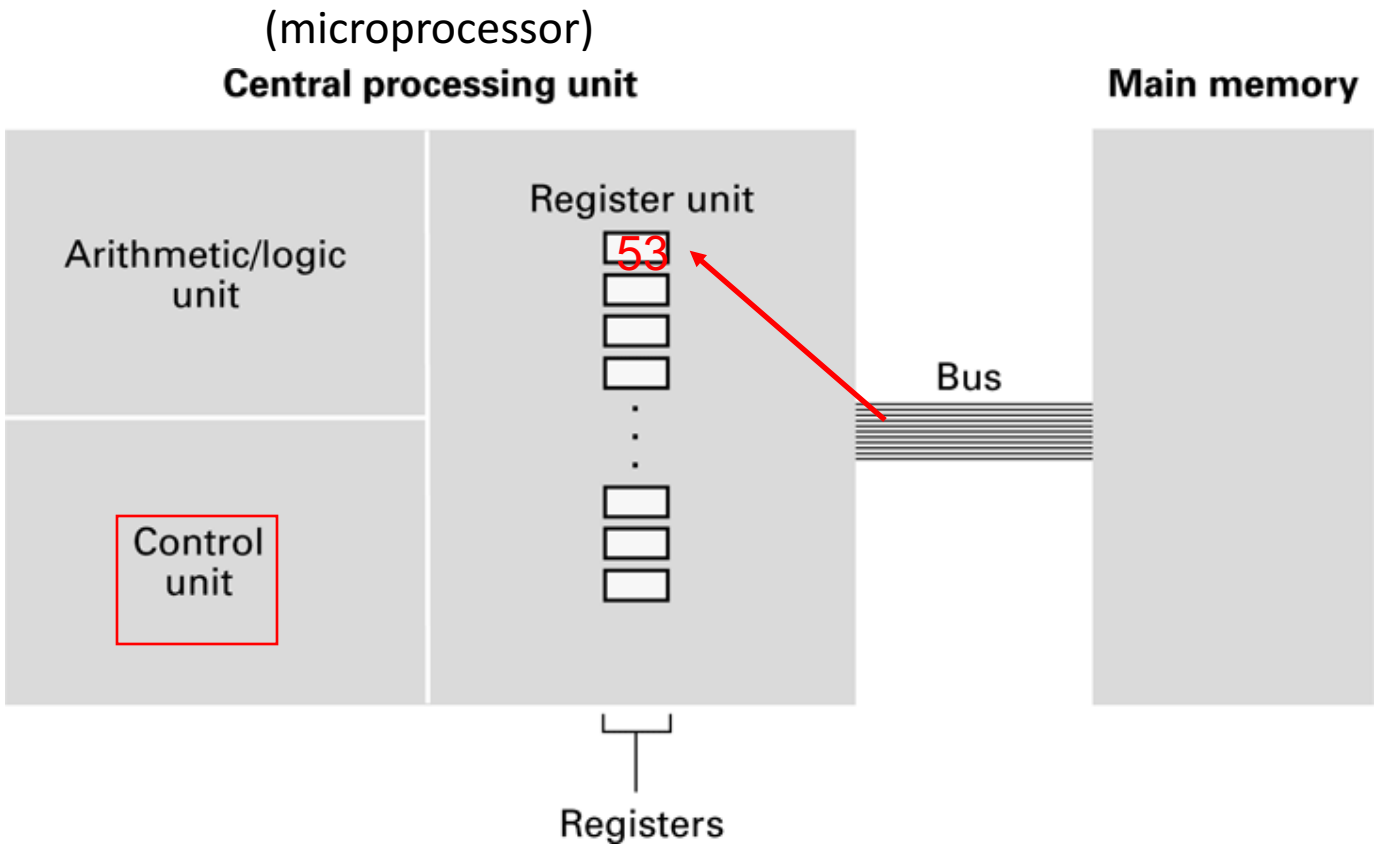
(microprocessor)



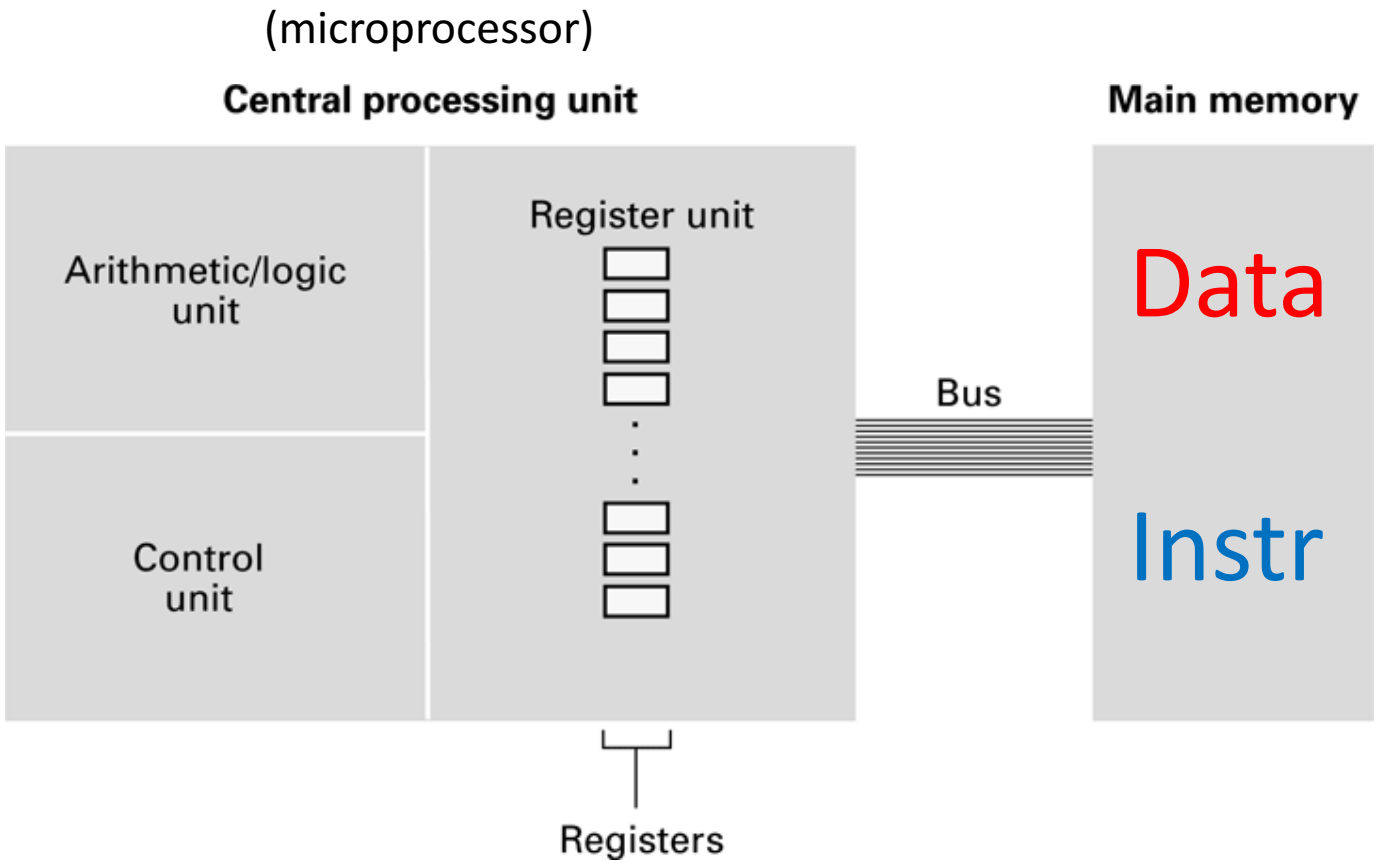
Computer Architecture



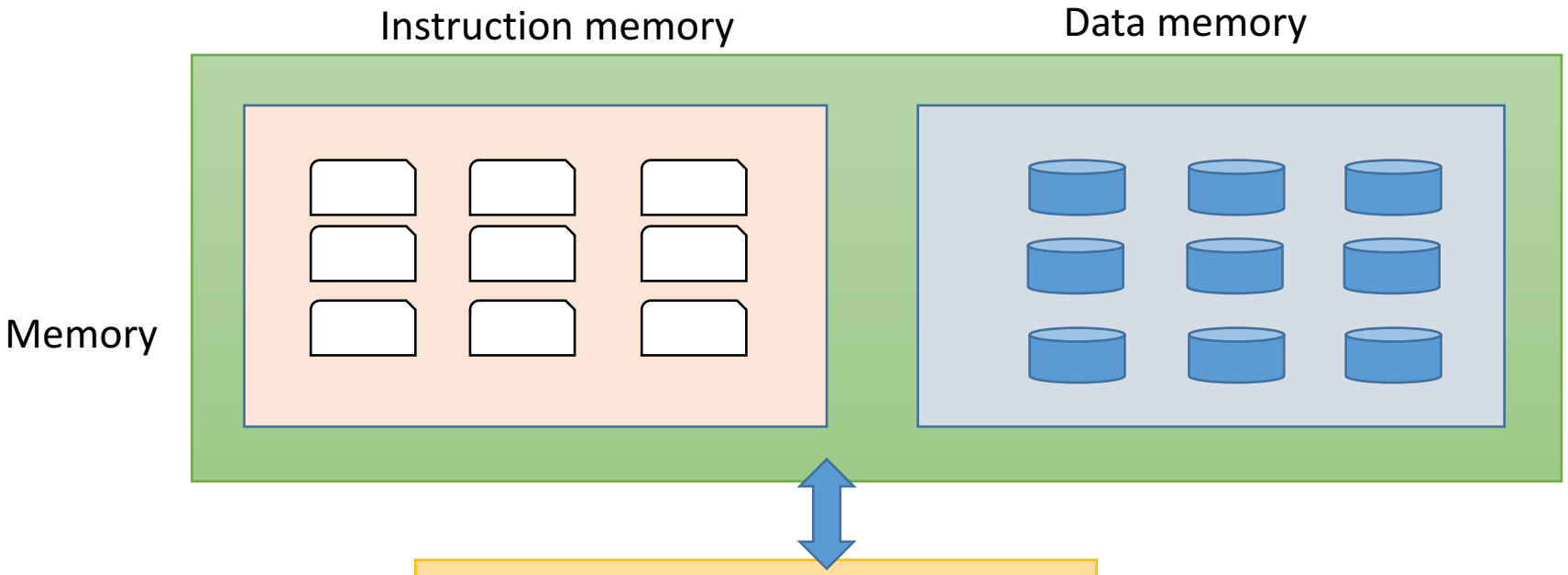
Computer Architecture



Computer Architecture



The cycle of execution



CPU

Loop:

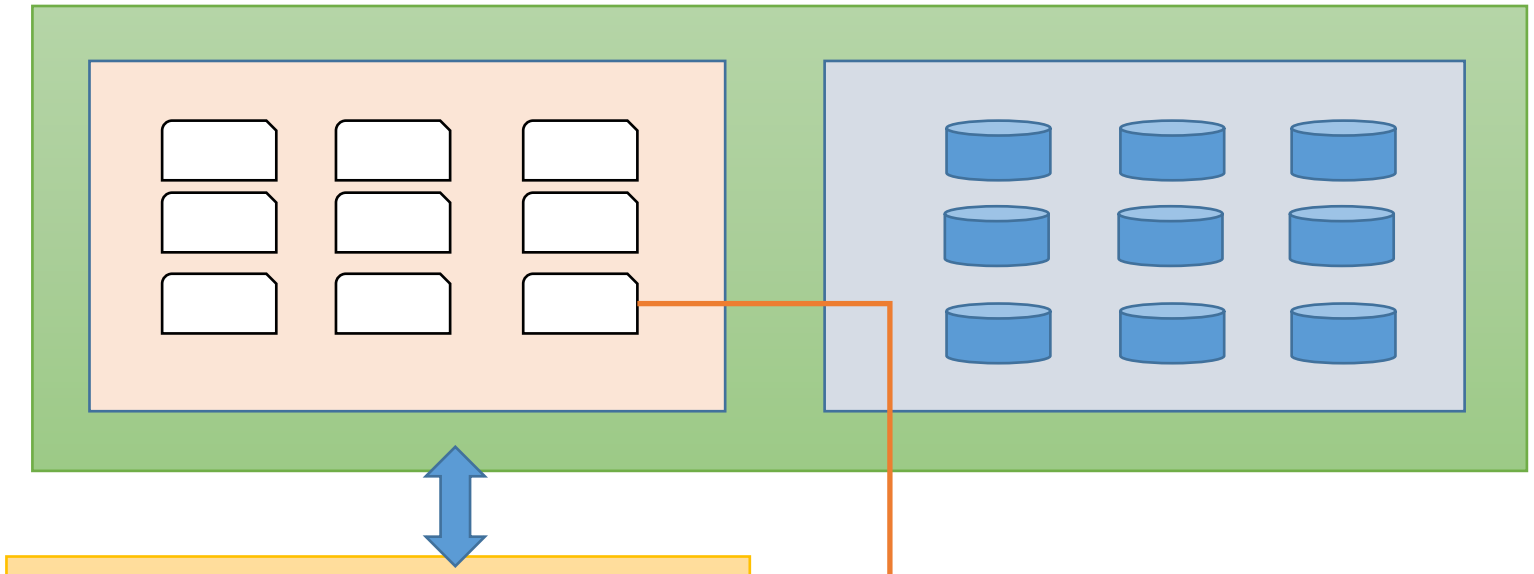
1. Fetch instruction
2. Decode instruction
3. (Fetch Data)
4. Execution
5. (Store Data)

The cycle of execution

Instruction memory

Data memory

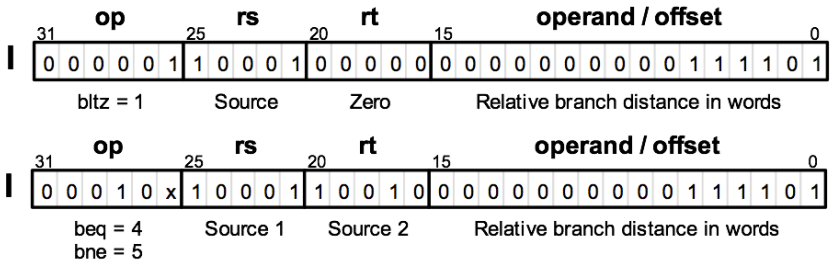
Memory



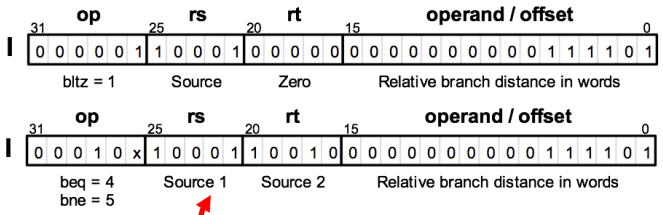
CPU

Loop:

1. Fetch instruction
2. Decode instruction
3. (Fetch Data)
4. Execution
5. (Store Data)



Some MiniMIPS Instructions



	Instruction	Usage	op	fn
Copy	Load upper immediate	`lui rt,imm`	15	
Arithmetic	Add	`add rd,rs,rt`	0	32
	Subtract	`sub rd,rs,rt`	0	34
	Set less than	`slt rd,rs,rt`	0	42
	Add immediate	`addi rt,rs,imm`	8	
	Set less than immediate	`slti rd,rs,imm`	10	
Logic	AND	`and rd,rs,rt`	0	36
	OR	`or rd,rs,rt`	0	37
	XOR	`xor rd,rs,rt`	0	38
	NOR	`nor rd,rs,rt`	0	39
	AND immediate	`andi rt,rs,imm`	12	
	OR immediate	`ori rt,rs,imm`	13	
	XOR immediate	`xori rt,rs,imm`	14	
Memory access	Load word	`lw rt,imm(rs)`	35	
	Store word	`sw rt,imm(rs)`	43	
Control transfer	Jump	`j L`	2	
	Jump register	`jr rs`	0	8
	Branch less than 0	`bltz rs,L`	1	
	Branch equal	`beq rs,rt,L`	4	
	Branch not equal	`bne rs,rt,L`	5	

Finding the max in a list

mymax.c mymax.py

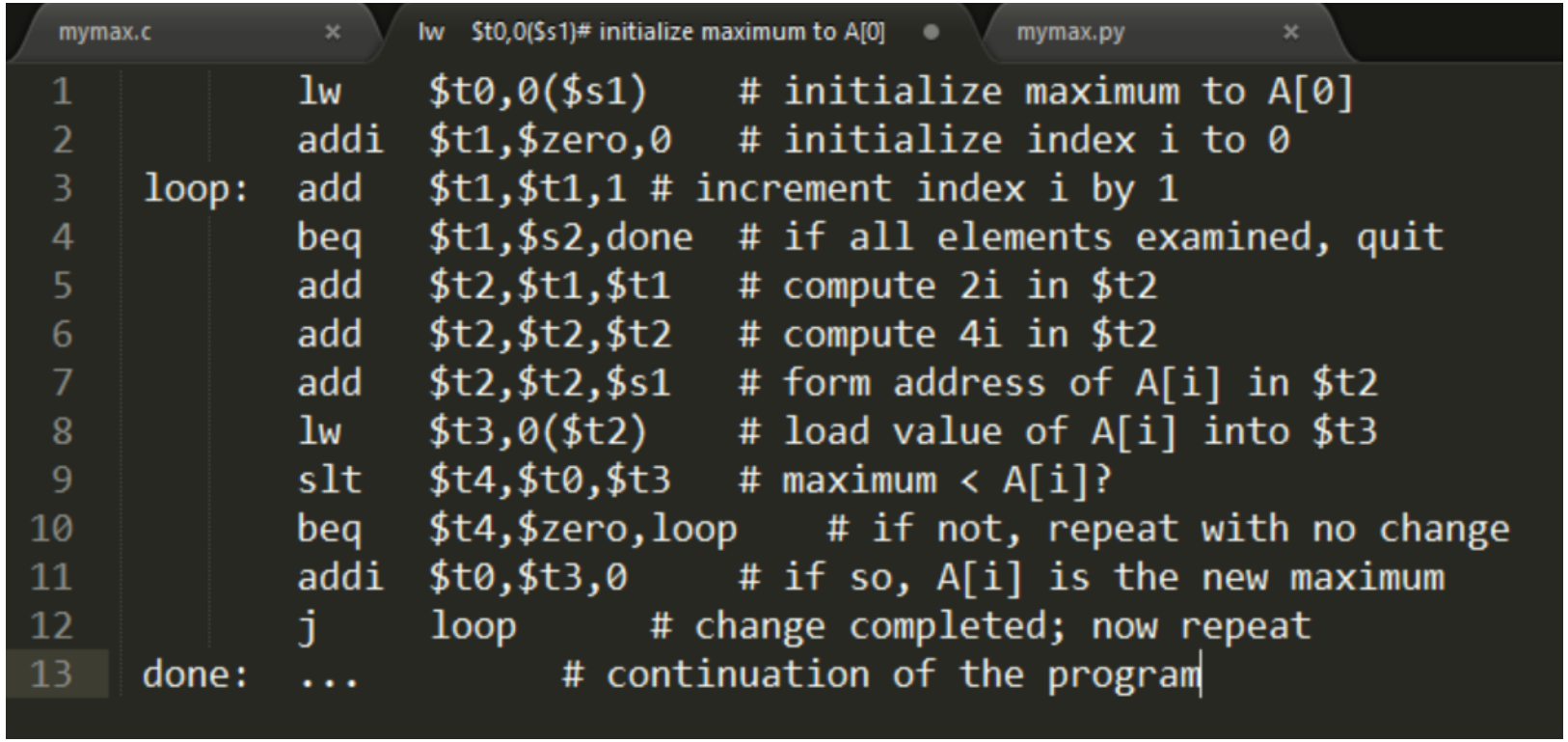
```
1
2  array_size = int(input('how many? '))
3  array = []
4  for i in range(array_size):
5      array.append(int(input('> ')))
6
7
8  my_max = array[0]
9  location = 0
10 for i in range(1, array_size):
11     if array[i] > my_max:
12         my_max = array[i]
13         location = i
14
15 print('max is: ', my_max, 'at ', location + 1)
```

Do it in C

mymax.c | mymax.py

```c
#include <stdio.h>

int main()
{
  int array[100], my_max, array_size, i, location = 1;

  printf("How many? \n");
  scanf("%d", &array_size);

  for (i = 0; i < array_size; i++)
    scanf("%d", &array[c]);

  my_max = array[0];
  location = 0
  for (i = 1; i < array_size; i++)
  {
    if (array[c] > my_max)
    {
       my_max  = array[i];
       location = i;
    }
  }

  printf("max is %d at location %d\n", my_max, location + 1);
}
```

Do it in assembly code: the job of the compiler



- $t0, $t1 etc. are ***registers***
- lw, addi, beq, j etc. are ***instructions***

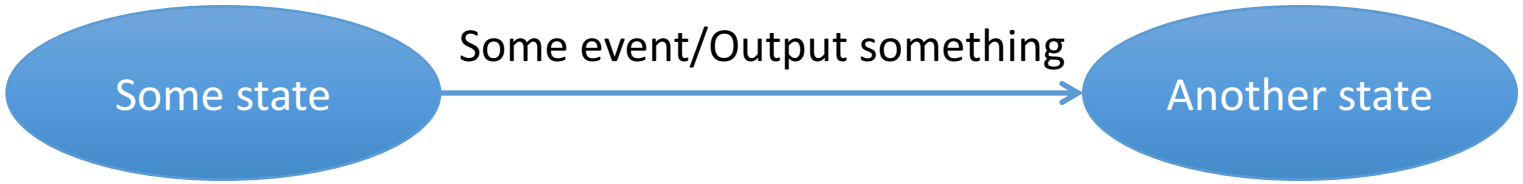
Key takeaways

- At the lowest level: 1s and 0s
- Bits are grouped into bytes
  - Each byte is 8 bits
  - This laptop has 16Giga-bytes (16*2^**30** bytes)
- What's stored in memory can be data or instructions
- Everything has a *state*!

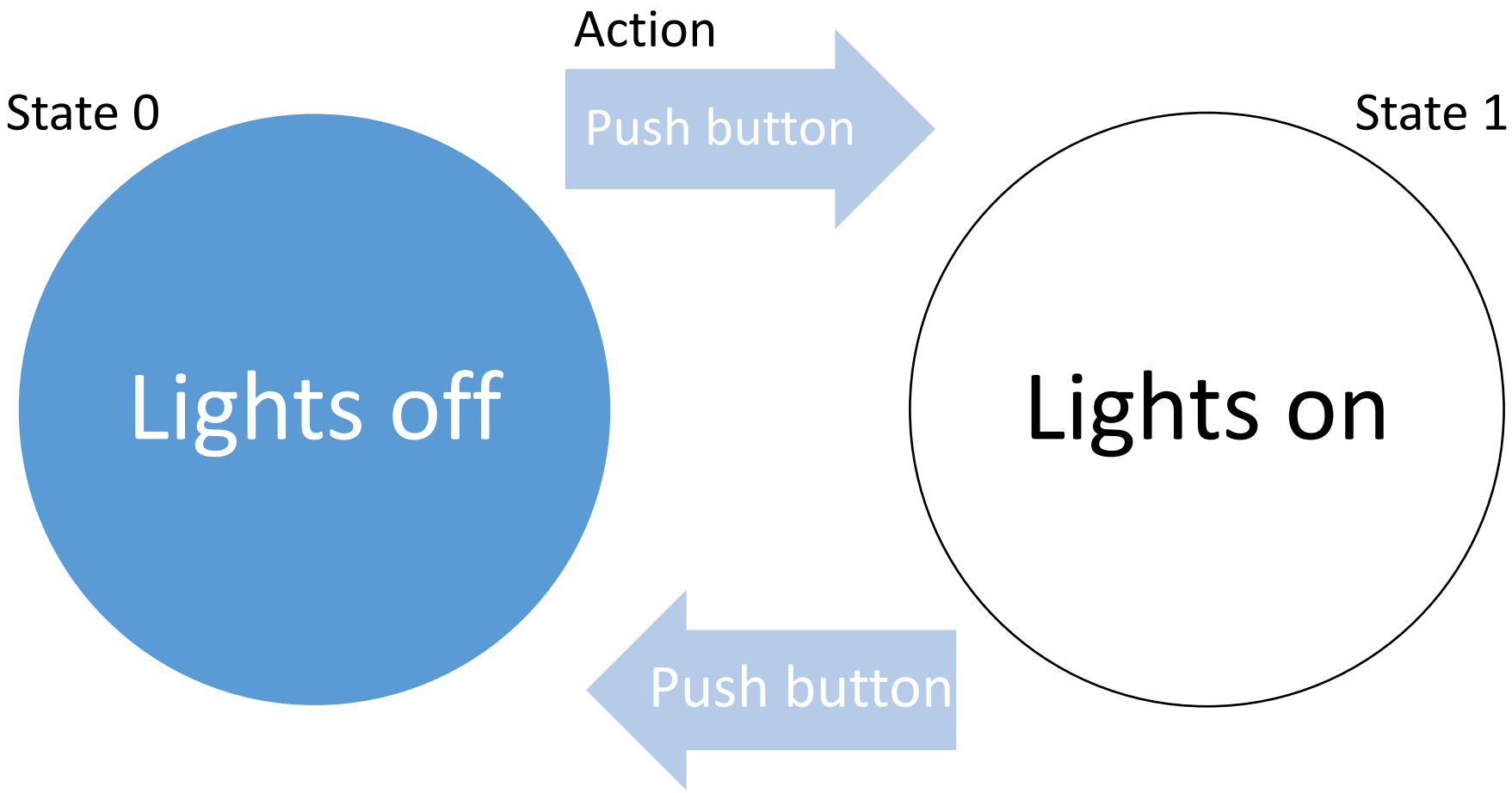
State machine

“3-instruction is enough”

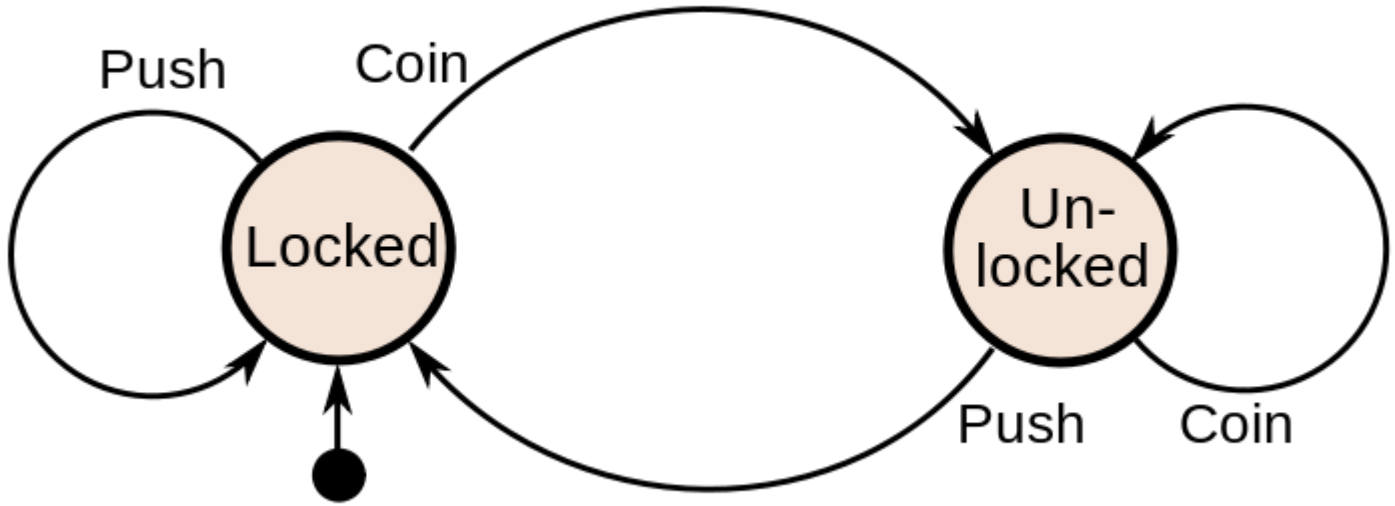
State space and (finite) state machine



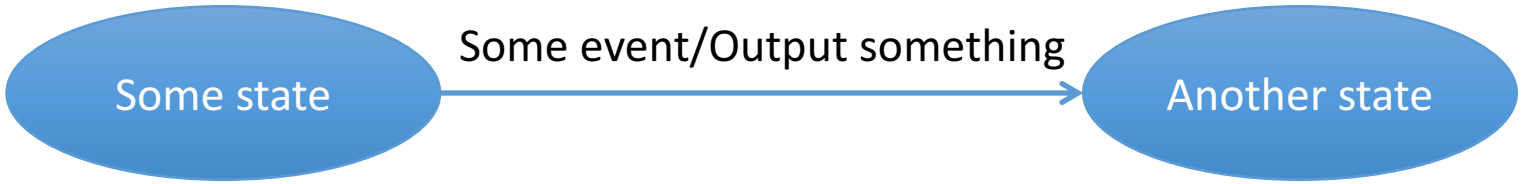
Finite State Machine: a lamp



Finite State Machine: turnstile

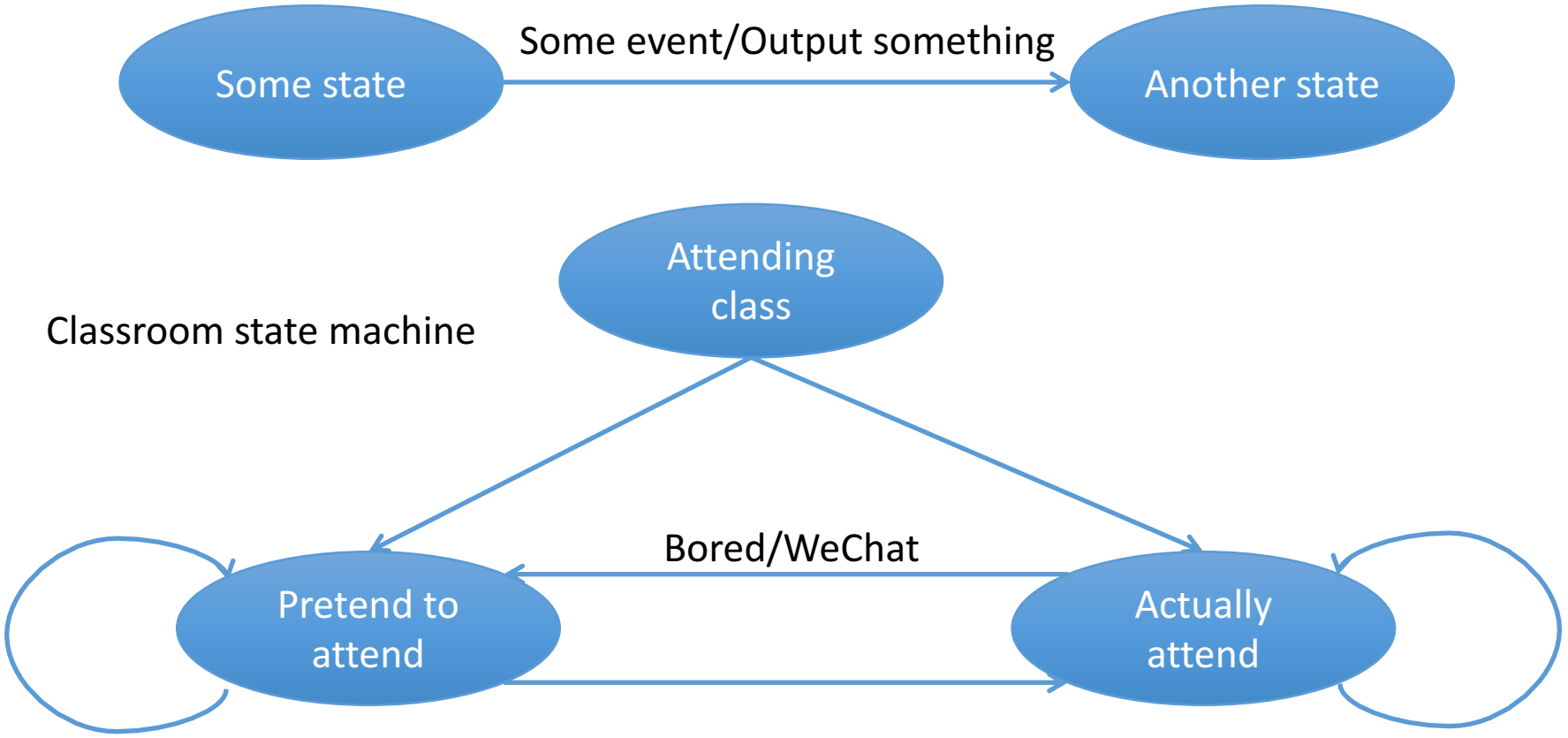


State space and state machine

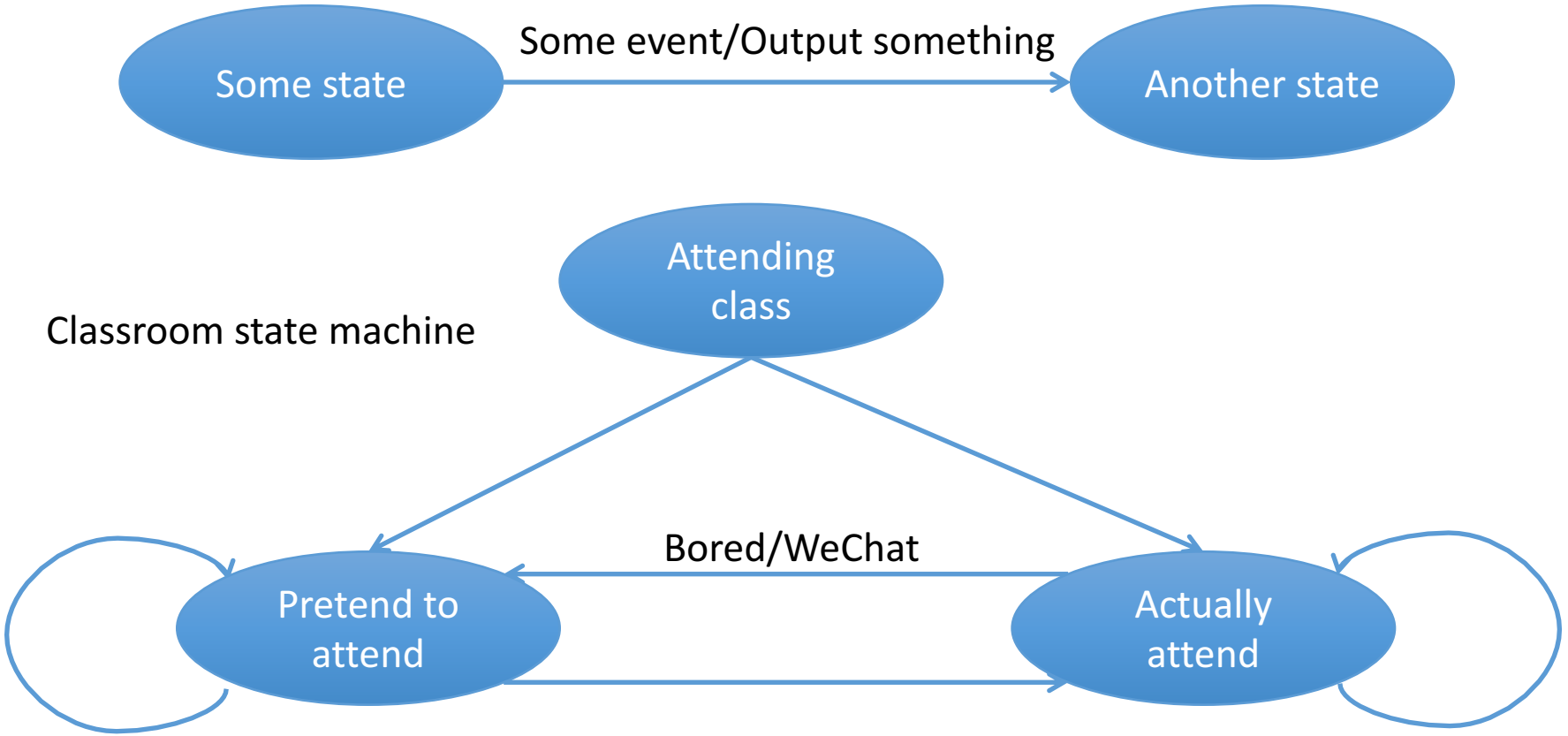


Classroom state machine

State space and state machine



State space and state machine



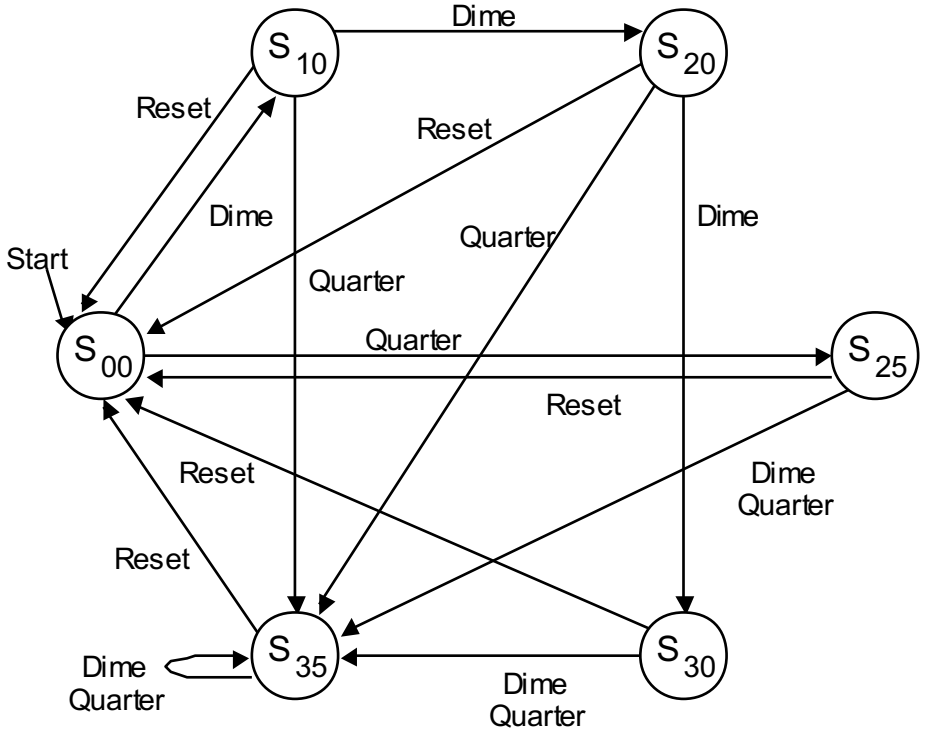
- Your program is a bunch of logics that move state from one region to the other

FSM of a vending machine

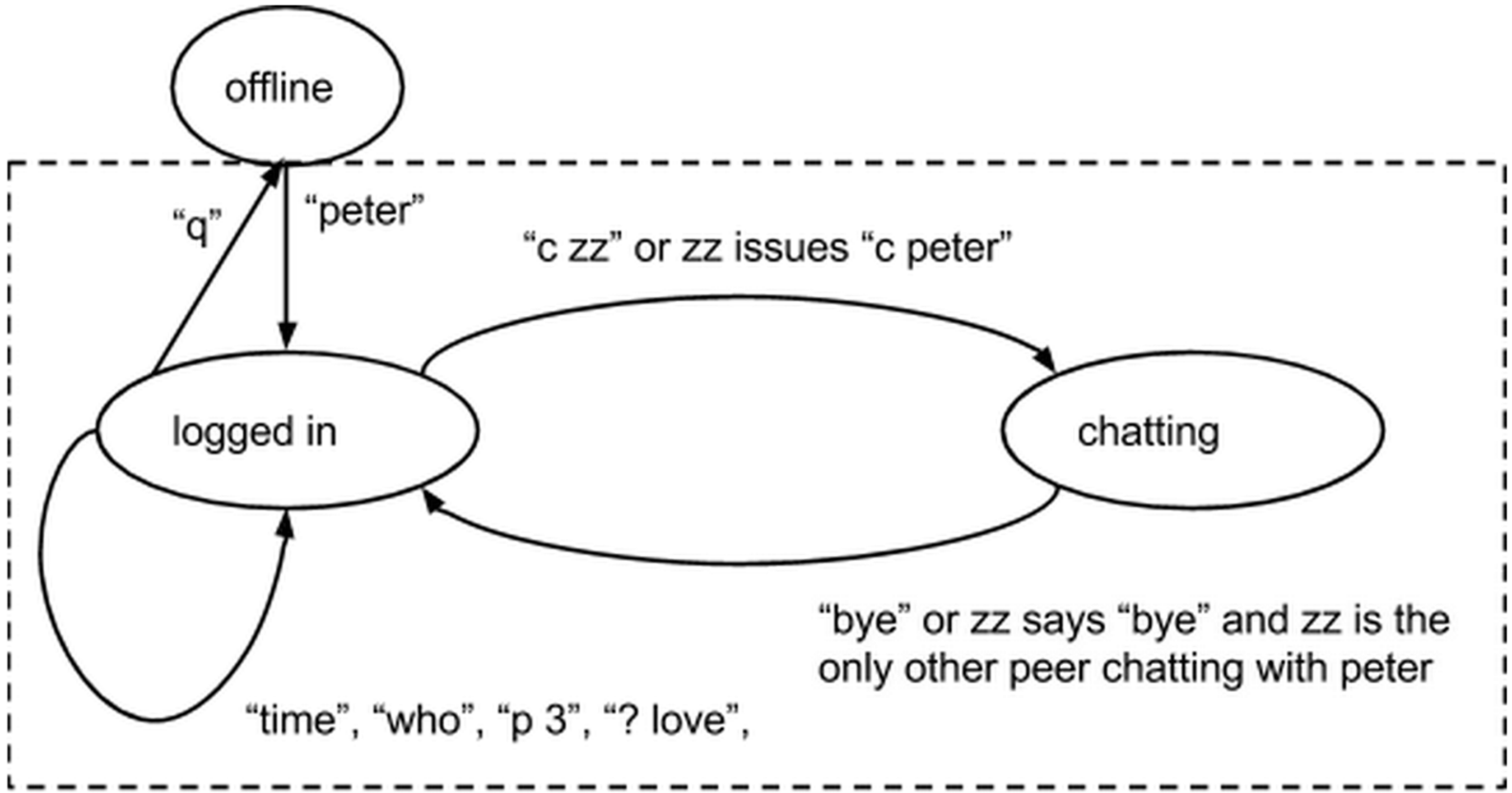
Current state	Input: Dime	Input: Quarter	Input: Reset
S_{00}	S_{10}	S_{25}	S_{00}
S_{10}	S_{20}	S_{35}	S_{00}
S_{20}	S_{30}	S_{35}	S_{00}
S_{25}	S_{35}	S_{35}	S_{00}
S_{30}	S_{35}	S_{35}	S_{00}
S_{35}	S_{35}	S_{35}	S_{00}

Next state

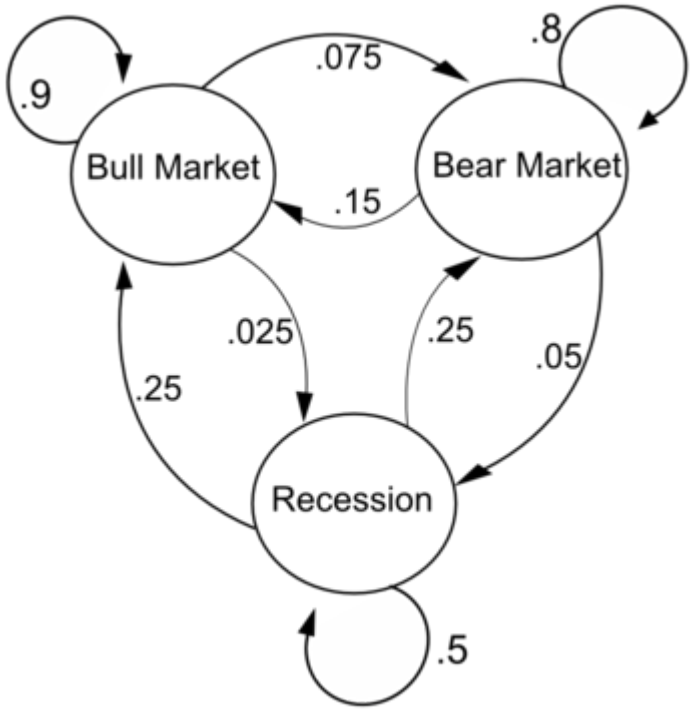
S_{00} is the initial state
S_{35} is the final state



Behind scene: state machine of the chat client



State Machine and Markov Model



Homework: FizzBuzz

Take an integer *n*; print:

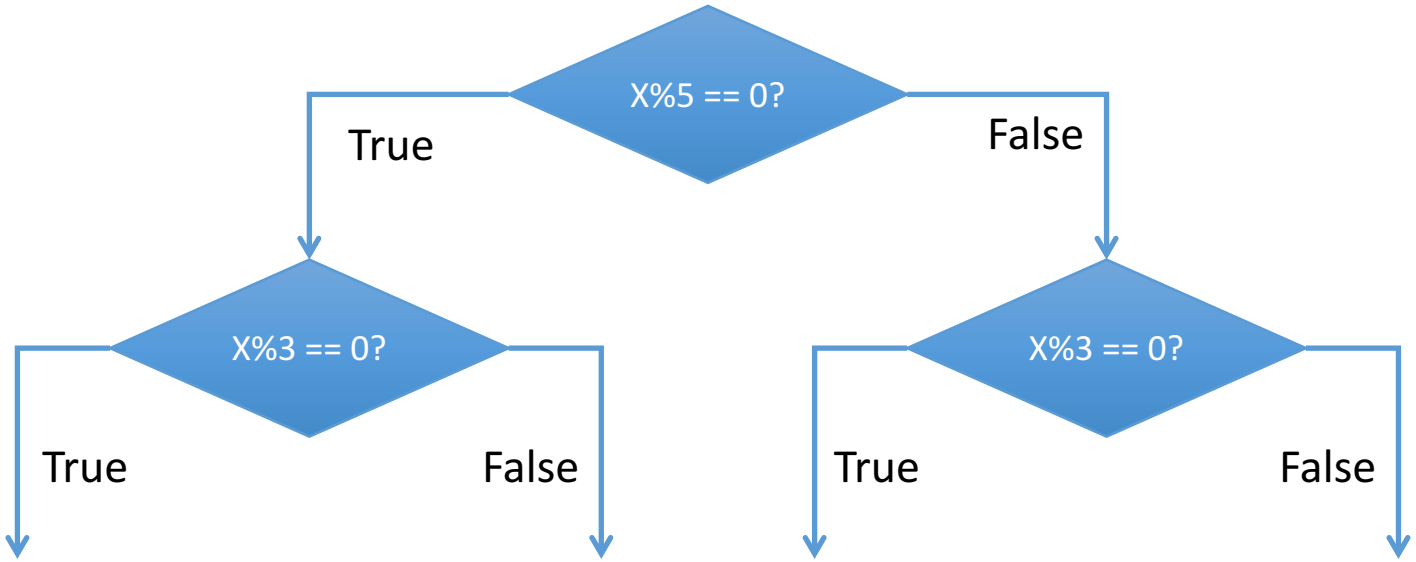
- “Fizz” if divisible ***only*** by 3
- “Buzz” if divisible ***only*** by 5
- “FizzBuzz” if divisible by ***both*** 3 and 5

vim (vim)

```
1
2 if x % 3 == 0:
3     print("fizz")
4 elif x % 5 == 0:
5     print("buzz")
6 elif x % 3 == 0 and x % 5 == 0:
7     print("fizzbuzz")
8
```

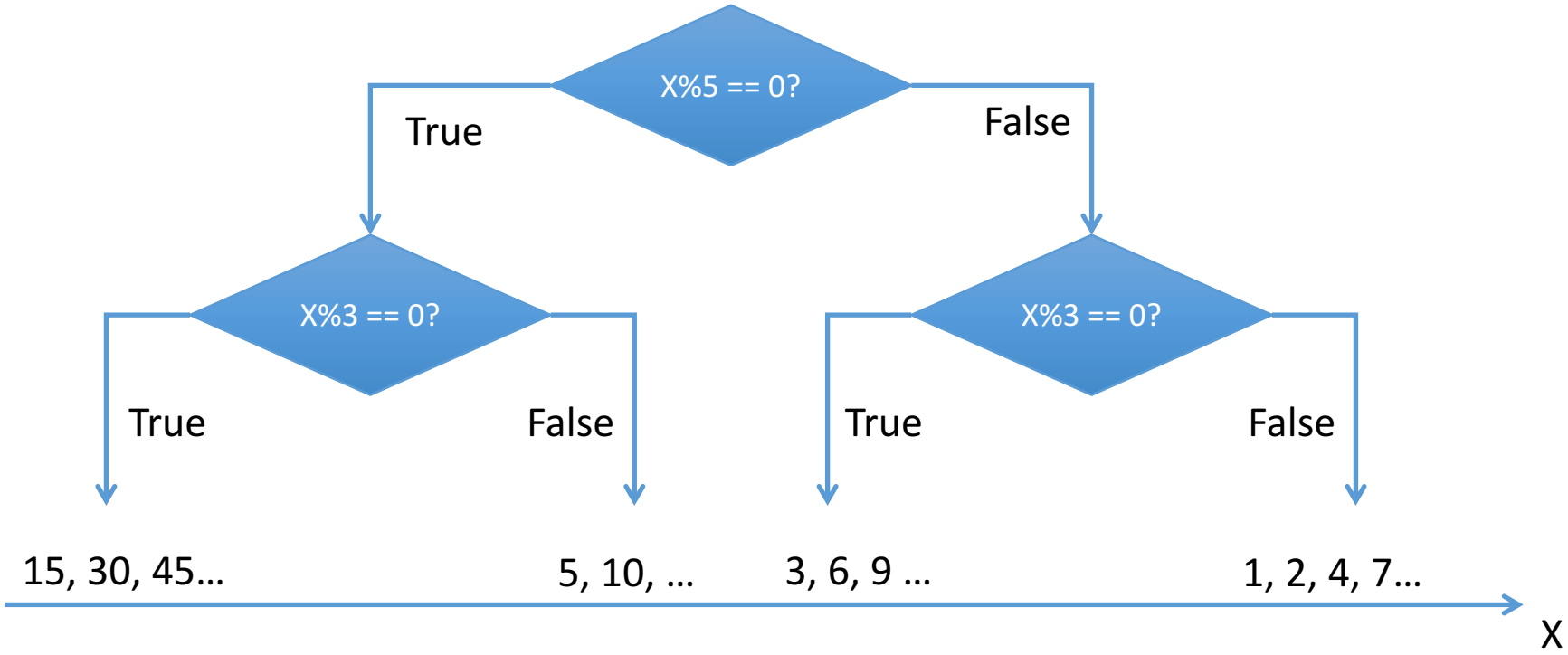
State space and state machine

- A one dimension example: FizzBuzz



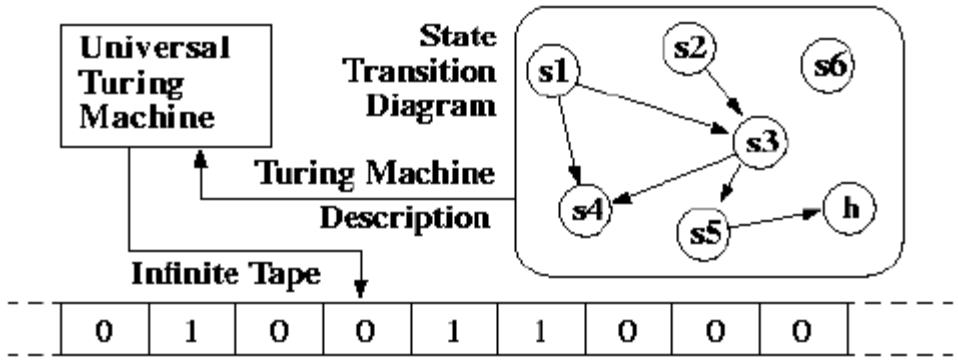
State space and state machine

- A one dimension example: FizzBuzz



Turing machine

- https://www.cl.cam.ac.uk/projects/raspberrypi/tutorials/
- http://morphett.info/turing/turing.html



Example: state table for the 3-state 2-symbol busy beaver reduced to 5-tuples

Current state	Scanned symbol		Print symbol	Move tape	Final (i.e. next) state	5-tuples
A	0		1	R	**B**	(**A**, 0, 1, R, **B**)
A	1		1	L	**C**	(**A**, 1, 1, L, **C**)
B	0		1	L	**A**	(**B**, 0, 1, L, **A**)
B	1		1	R	**B**	(**B**, 1, 1, R, **B**)
C	0		1	L	**B**	(**C**, 0, 1, L, **B**)
C	1		1	N	**H**	(**C**, 1, 1, N, **H**)

The register machine

The register machine

- Register (matchbox): a box that contains any *integer* (matches)
- There are infinite number of registers

- A possibly infinite number of steps
- each step is a state
- Each step is one of 3 instructions
  - **INC:** increment register(***n***) by 1, go to step ***x***
  - **DEB:** decrement register(***n***) by 1, go to step **y** , if R(***n***) is 0 branch to step ***z***
  - **END:** stop the program

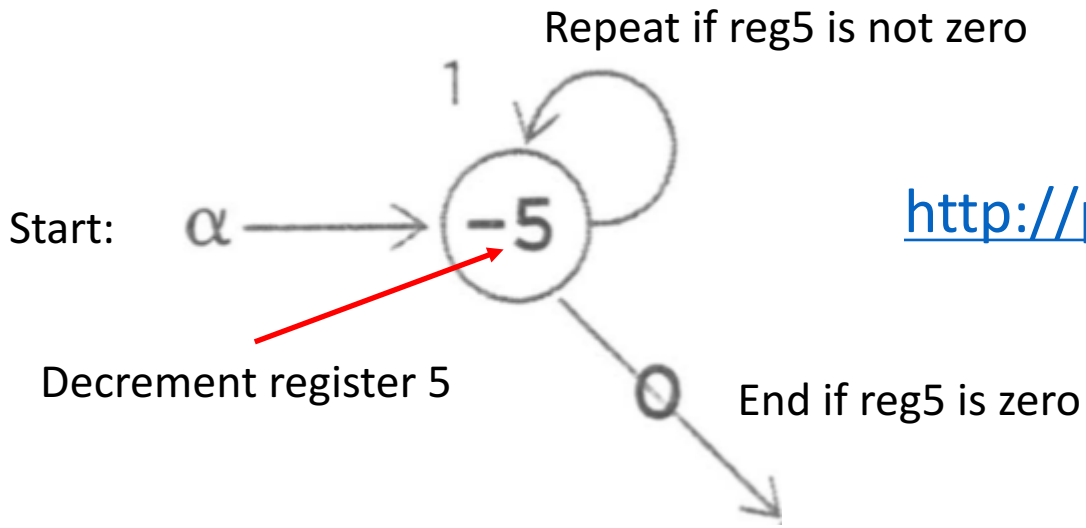
Register machine instructions

Example	Code
1. inc R(***1***), go to step ***3***	[1] INC 1 3
2. deb R(***n***), go to step 2, if R(***n***) is 0 go to step 3	[2] DEB 3 2 3
3. end	[3]END

Line number of the steps

CLEAR[5]

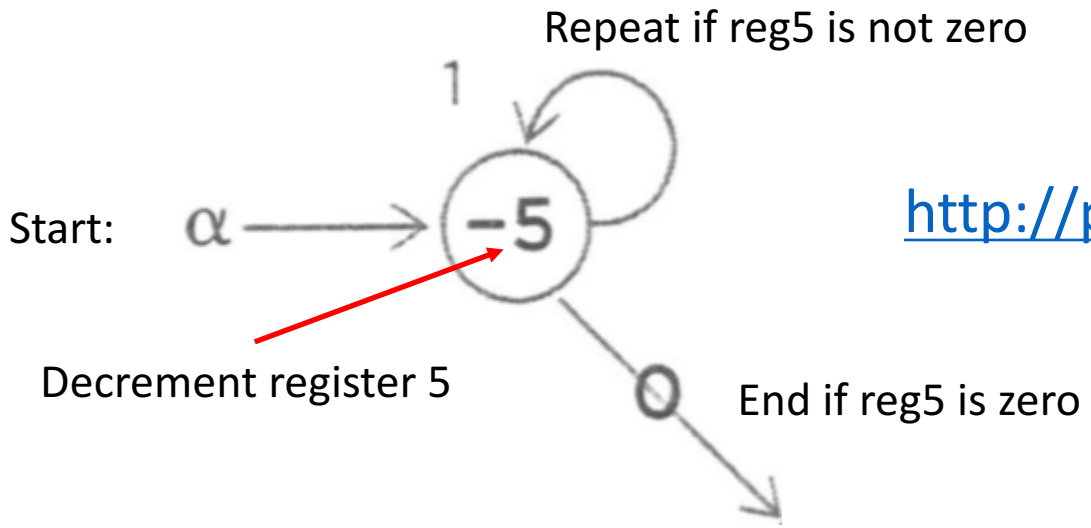
Step	Instruction	Register	Goto	Branch to
1	Deb	5	1	2
2	End			



http://proto.atech.tufts.edu/RodRego/

CLEAR[5]

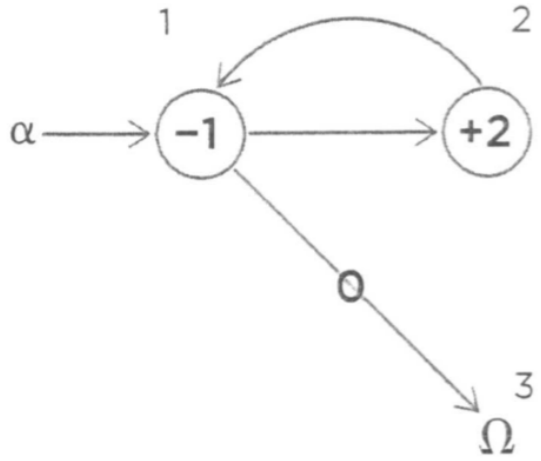
Step	Instruction	Register	Goto	Branch to
1	Deb	5	1	2
2	End			



http://proto.atech.tufts.edu/RodRego/

(Destructive) ADD[1, 2]: add content in R1 into R2

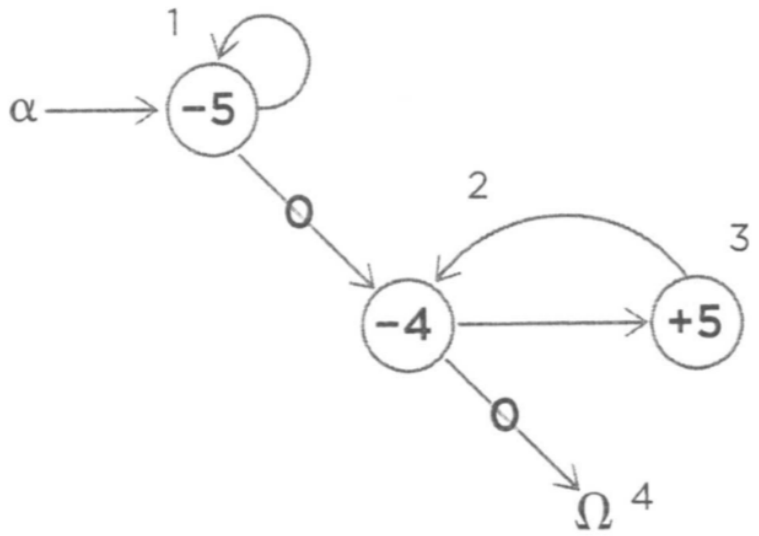
Step	Instruction	Register	Goto	Branch to
1	Deb	1	2	3
2	Inc	2	1	
3	End			



Destructive:
Content in R1 is gone!

MOVE[4, 5]: move R4 to R5 (clear R5, R4)

Step	Instruction	Register	Goto	Branch to
1	Deb	5	1	2
2	Deb	4	3	4
3	Inc	5	2	
4	End			



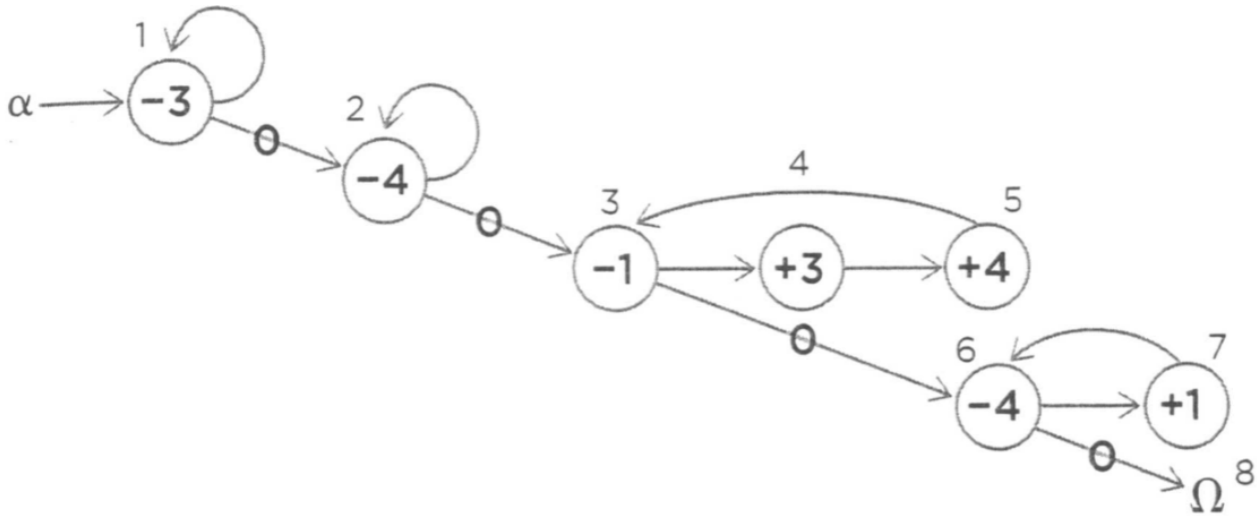
Next, do:

COPY[1, 3]

Before: r(1) = n, r(3) = m

After: r(1) = n, r(3) = n

COPY[1, 3]



program 3: COPY [1,3]

STEP	INSTRUCTION	REGISTER	GO TO STEP	[BRANCH TO STEP]
1.	*Deb*	3	1	2
2.	*Deb*	4	2	3
3.	*Deb*	1	4	6
4.	*Inc*	3	5	
5.	*Inc*	4	3	
6.	*Deb*	4	7	8
7.	*Inc*	1	6	
8.	*End*			

clear reg3

clear reg4

move reg1 to reg3 *and* reg4

move reg4 back to reg1

MULTIPLY[1, 2, 3]

- Can be composed by Deb and the other building blocks!

MULTIPLY[1, 2, 3]

- Can be composed by Deb and the other building blocks!

- # R1 = N
- # R2 = M
  - CLEAR(R4)
  - LOOP R2 times:
  - COPY(R1, R3)
  - ADD(R3, R4)

Non-destructive ADD[1, 2, 3]

- Contents in register 1 and 2 stay. How?

Non-destructive ADD[1, 2, 3]

1. COPY[1, 4]
2. COPY[2, 5]
3. ADD[4, 5]
4. CLEAR[3]
5. MOVE[5, 3]

Q: can you do better?

```
1 DEB 3  1 2
2 DEB 4 2 3
3 DEB 5 3 4
# add and make copy
# ADD R1 TO R3, R4
4 DEB 1 5 91
5 INC 3 6
6 INC 4 4
# ADD R2 TO R3, R5
91 DEB 2 7 92
7 INC 3 8
8 INC 5 91
# MOVE BACK R4 TO R1
92 DEB 4 10 93
10 INC  1 92
# MOVE BACK R5 TO R2
93 DEB 5 11 99
11 INC 2 93
99 END
```

Key takeaways

- State machine as a way of thinking, modeling, AND programming
- Building ***abstractions*** allows easier programming by composition
- 3-instructions is enough
- Programming in assembly language is *hard* but can be optimal

CS fields that are touched

- Theoretical computation

- Digital circuitry design
- Computer architecture

- Compiler
- Programming languages

Q&A

To see a World in a Grain of Sand
And a Heaven in a Wild Flower,
Hold Infinity in the palm of your hand
And Eternity in an hour.